Contents

Chapter 1

1

Záróvizsga tételsor

1. Függvények határértéke, folytonossága

Fekete Dóra

Függvények határértéke, folytonossága

Függvények határértéke, folytonossága. Kompakt halmazon folytonos függvények tulajdonságai: Heine-tétel, Weierstrass-tétel, Bolzano-tétel. A hatványsor fogalma, Cauchy–Hadamard-tétel, analitikus függvények.

1 Függvények határértéke

Adott $f \in \mathbb{K}_1 \to \mathbb{K}_2, a \in \mathcal{D}'_f$ (torlódási pont). Az f függvénynek az a helyen van határértéke, ha $\exists A \in \overline{\mathbb{K}}_2$, ahol $\overline{\mathbb{K}} = \mathbb{C} \vee \mathbb{R} \vee +\infty \vee -\infty$, amelyre tetszőleges $K(A) \subset \mathbb{K}_2$ környezetet is véve megadható az a-nak egy alkalmas $k(a) \subset \mathbb{K}_1$ környezete, amellyel $f[(k(a)\backslash\{a\}) \cap \mathcal{D}_f] \subset K(A)$ teljesül.
Másképp: $f(x) \in K(A), (a \neq x \in k(a) \cap \mathcal{D}_f)$.
Ekkor az egyértelműen létező $A \in \overline{\mathbb{K}}_2$ számot vagy valamelyik végtelent az f függvény a helyen vett határértékének nevezzük.
Jelölés: $\lim\limits_{x \to a} f(x) = \lim\limits_{a} f = A$

1.1 Torlódási pont

$A \subset \mathbb{K}$, ekkor az $\alpha \in \overline{\mathbb{K}}$ torlódási pontja az A halmaznak, ha bármely $K(\alpha)$ környezetre $(K(\alpha)\backslash\{\alpha\}) \cap A \neq \emptyset$.
Egyenlőtlenségekkel: $A \subset \mathbb{K}$, ekkor $\alpha \in \mathbb{K}$ szám torlódási pontja az A halmaznak, ha $\forall \varepsilon > 0$ esetén $\exists x \in A$, hogy $0 < |x - \alpha| < \varepsilon$.

1.2 Környezet

$A \subset \mathbb{K}, a \in A, r > 0 : K_r(a) = K(a) = \{x \in A : |x - a| < r\}$.

1.3 Függvény

$\emptyset \neq A, B$ halmazok, $f \subset A \times B$ reláció. Valamely $x \in A$ elemre legyen $f_x := \{y \in B : (x, y) \in f\}$. f reláció függvény, ha $\forall x \in A$-ra az f_x halmaz legfeljebb egy elemű.

2 Függvények folytonossága

Az $f \in \mathbb{K}_1 \to \mathbb{K}_2$ függvény az $a \in \mathcal{D}_f$ pontban folytonos, ha $\forall \varepsilon > 0$ számhoz megadható olyan $\delta > 0$ szám, amellyel bármely $x \in \mathcal{D}_f, |x - a| < \delta$ esetén $|f(x) - f(a)| < \varepsilon$.
Jelölés: $f \in \mathcal{C}\{a\}$, ha $\forall x \in \mathcal{D}_f : f \in \mathcal{C}\{x\}$, akkor $f \in \mathcal{C}$.

3 Összefüggés határérték és folytonosság között

Legyen $f \in \mathbb{K}_1 \to \mathbb{K}_2, a \in \mathcal{D}_f \cap \mathcal{D}'_f$. Ekkor $f \in \mathcal{C}\{a\} \iff \lim\limits_{a} f = f(a)$.

4 Fogalmak

4.1 Kompakt halmaz

$A \subset \mathbb{K}$ kompakt, ha bármely $(x_n) : \mathbb{N} \to A$ sorozatnak van olyan (x_{ν_n}) részsorozata, amely konvergens és $\lim (x_{\nu_n}) \in A$.
Ekkor A korlátos és zárt.

4.2 Konvergens

Egy $x = (x_n) : \mathbb{N} \to \mathbb{K}$ számsorozatot konvergensnek nevezünk, ha $\exists \alpha \in \mathbb{K}, \forall \varepsilon > 0, \exists N \in \mathbb{N}, \forall n \in \mathbb{N}, n > N : |x_n - \alpha| < \varepsilon$.
α az x sorozat határértéke.

4.3 Korlátos

Sorozatra: x_n korlátos $\Rightarrow \exists \nu : x \circ \nu$ konvergens
Halmazra: $\emptyset \neq A \subset \mathbb{K}$ korlátos, ha $\exists q \in \mathbb{R} : |x| \leq q, (x \in A)$

4.4 Zárt halmaz

Komplementere nyílt halmaz.

4.5 Nyílt halmaz

A nyílt halmaz $\iff \forall a \in A, \exists K(a) : K(a) \subset A$, vagyis az A halmaz minden pontja belső pont.

5 Heine-tétel

Ha az $f \in \mathbb{K}_1 \to \mathbb{K}_2$ függvény folytonos és $\mathcal{D}_f$ kompakt, akkor f egyenletesen folytonos.

5.1 Egyenletesen folytonos

$f \in \mathbb{K}_1 \to \mathbb{K}_2$ függvény egyenletesen folytonos, ha $\forall \varepsilon > 0, \exists \delta > 0 : |f(x) - f(t)| < \varepsilon, (x, t \in \mathcal{D}_f, |x - t| < \delta)$.

6 Weierstrass-tétel

Tegyük fel, hogy az $f \in \mathbb{K} \to \mathbb{R}$ függvény folytonos és $\mathcal{D}_f$ kompakt. Ekkor az $\mathcal{R}_f$ értékkészletnek van legnagyobb és legkisebb eleme ($\exists \max \mathcal{R}_f, \exists \min \mathcal{R}_f$).

7 Bolzano-tétel

Tegyük fel, hogy valamely $-\infty < a < b < +\infty$ esetén az $f : [a, b] \to \mathbb{R}$ függvény folytonos, és $f(a) \cdot f(b) < 0$ (ellenkező előjelű).
Ekkor van olyan $\xi \in (a, b)$, amelyre $f(\xi) = 0$.

8 A hatványsor fogalma

Legyen adott az $a \in \mathbb{K}$ középpont és az $(a_n) : \mathbb{N} \to \mathbb{K}$ együttható-sorozat, továbbá ezek segítségével tekintsük az alábbi függvényeket: $f_n(t) := a_n(t - a)^n, (t \in \mathcal{D} := \mathbb{K}, n \in \mathbb{N})$. Ekkor a $\sum (f_n)$ függvénysort hatványsornak nevezzük.

8.1 Sorozat

Az f függvényt sorozatnak nevezzük, ha $\mathcal{D}_f = \mathbb{N}$.

8.2 Függvénysorozat, függvénysor

(f_n) sorozat függvénysorozat, ha $\forall n \in \mathbb{N}$ esetén f_n függvény, és valamilyen $\mathcal{D} \neq \emptyset$ halmazzal $\mathcal{D}_{f_n} = \mathcal{D}, (n \in \mathbb{N})$.
Ha a szóban forgó (f_n) függvénysorozatra $\mathcal{R}_{f_n} \subset \mathbb{K}, (n \in \mathbb{N})$ is igaz, akkor az (f_n) függvénysorozat által meghatározott $\sum (f_n)$ függvénysor a következő függvénysorozat: $\sum (f_n) := (\sum_{k=0}^{n} f_k)$.

9 Cauchy–Hadamard-tétel

Tegyük fel, hogy az $(a_n) : \mathbb{N} \to \mathbb{K}$ sorozat esetén létezik a $\lim(\sqrt[n]{|a_n|})$ határérték, és legyen

$$r := \begin{cases} +\infty & ha \lim(\sqrt[n]{|a_n|}) = 0 \\ \dfrac{1}{\lim(\sqrt[n]{|a_n|})} & ha \lim(\sqrt[n]{|a_n|}) > 0 \end{cases}$$

r-t a konvergenciasugárnak nevezzük. Ekkor bármely $a \in \mathbb{K}$ mellett a $\sum (a_n(t-a)^n)$ hatványsorról a következőket mondhatjuk:

1. Ha $r > 0$, akkor minden $x \in \mathbb{K}, |x-a| < r$ helyen a $\sum (a_n(t-a)^n)$ hatványsor az x helyen abszolút konvergens.
2. Ha $r < +\infty$, akkor tetszőleges $x \in \mathbb{K}, |x-a| > r$ mellett a $\sum (a_n(t-a)^n)$ hatványsor az x helyen divergens.

9.1 Abszolút konvergens

A $\sum_{n=1}^{\infty} a_n$ végtelen sort abszolút konvergensnek nevezzük, ha a $\sum_{n=1}^{\infty} |a_n|$ sor konvergens.

9.2 Divergens

Ha a $\lim_{n\to\infty} a_n$ nem létezik, vagy nem véges, akkor a $\sum a_n$ végtelen sor divergens.

9.3 Cauchy–Hadamard-tételből következik

1. Ha $r = +\infty$, akkor $\mathcal{D}_0 = \mathbb{K}$, és $\forall x \in \mathbb{K}$-ra a hatványsor abszolút konvergens.
2. Ha $r = 0$, akkor $\mathcal{D}_0 = \{a\}$ és $\sum_{n=0}^{\infty} a_n(a-a)^n = a_0$.
3. Ha $0 < r < +\infty$, akkor $K_r(a) \subset \mathcal{D}_0 \subset G_r(a) = \{x \in \mathbb{K} : |x-a| \leq r\}$ és a $K_r(a)$ környezet $\forall x$ pontjára a hatványsor az x helyen abszolút konvergens.
4. Ha $r > 0$, akkor tetszőleges $0 \leq \rho < r$ számhoz válasszuk az $x \in K_r(a)$ elemet úgy, hogy $|x-a| = \rho$. Ekkor $\sum_{n=0}^{\infty} |a_n(x-a)^n| = \sum_{n=0}^{\infty} |a_n||x-a|^n) = \sum_{n=0}^{\infty} |a_n|\rho^n) < +\infty$

10 Analitikus függvények

Tegyük fel, hogy a $\sum (a_n(t-a)^n)$ hatványsor r konvergenciasugara (ld. C–H-tétel) nem nulla. Ekkor értelmezhetjük az alábbi függvényt: $f(x) := \sum_{n=0}^{\infty} a_n(x-a)^n, (x \in K_r(a))$, ami nem más, mint a $\sum (a_n(t-a)^n)$ függvénysor $F(x) := \sum_{n=0}^{\infty} a_n(x-a)^n, (x \in \mathcal{D}_0)$ összegfüggvényének a leszűkítése a $K_r(a)$ környezetre: $f = F|_{K_r(a)}$, (f a hatványsor összegfüggvénye). Az ilyen szerkezetű f függvényt analitikusnak nevezzük.
Megjegyzés: $\mathcal{D}_0$ a $\sum (a_n(t-a)^n)$ hatványsor konvergenciatartományát jelöli.

10.1 Fontos analitikus függvények

Olyan a_n-ek, amire $\lim(\sqrt[n]{|a_n|}) = 0$, tehát $r = +\infty$ a $\sum(a_n t^n)$ hatványsorok esetén. Ezek az a_n-ek: $\frac{1}{n!}, \frac{(-1)^n}{(2n+1)!}, \frac{(-1)^n}{(2n)!}, \frac{1}{(2n+1)!}, \frac{1}{(2n)!}$.

- $\exp x := \exp(x) = e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}, (x \in \mathbb{K})$
- $\sin x := \sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}, (x \in \mathbb{K})$
- $\cos x := \cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}, (x \in \mathbb{K})$
- $\sinh x := \sinh(x) = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!}, (x \in \mathbb{K})$
- $\cosh x := \cosh(x) = \sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!}, (x \in \mathbb{K})$

Chapter 2

2

Záróvizsga tételsor
2. Differenciál- és integrálszámítás

Dobreff András

Differenciál- és integrálszámítás
Jacobi-mátrix, gradiens, parciális derivált. Szélsőérték, függvényvizsgálat. Riemann-integrál, parciális integrálás, integrálás helyettesítéssel. Newton-Leibniz-formula. A kezdeti érték probléma. Lineáris, ill. magasabb rendű lineáris differenciálegyenletek.

1 Jacobi-mátrix, gradiens, parciális derivált

1.1 Jacobi-mátrix, gradiens

Differenciálhatóság

$1 \leq n, m \in \mathbb{N}, \quad 1 \leq p, q \leq +\infty,$

$(\mathbb{R}^n, \|.\|_p)$ és $(\mathbb{R}^m, \|.\|_q)$ normált terek

$f \in \mathbb{R}^n \to \mathbb{R}^m, \ a \in intD_f$

Az f függvény differenciálható az a pontban ($f \in D\{a\}$) , ha
létezik olyan $L \in L(\mathbb{R}^n, \mathbb{R}^m)$ korlátos lineáris leképezés és olyan $\eta \in \mathbb{R}^n \to \mathbb{R}^m$ függvény, hogy :

$$f(a+h) - f(a) = L(h) + \eta(h) \cdot \|h\|_p \quad (h \in \mathbb{R}^n, a+h \in D_f)$$

ahol

$$\eta(h) \longrightarrow 0 \quad (\|h\|_p \to 0)$$

Más szóval:

$$\frac{f(a+h) - f(a) - L(h)}{\|h\|_p} \longrightarrow 0 \quad (\|h\|_p \to 0)$$

Amennyiben $\forall a \in intD_f : f \in D\{a\}$, akkor az f differenciálható ($f \in D$)

Megjegyzés:
A $\mathbb{K}$ test feletti $(X, \|.\|_\star)$, $(X, \|.\|_\heartsuit)$ normált terek közötti folytonos leképezés, korlátos lineáris leképezés, ha

- *lineáris, azaz*

$$f(x + \lambda y) = f(x) + \lambda f(y) \quad (x, y \in X, \lambda \in \mathbb{K})$$

- *korlátos, azaz*

$$\exists M \geq 0 : \|f(x)\|_\heartsuit \leq M \|x\|_\star \quad (x \in X)$$

Derivált

$f \in \mathbb{R}^n \to \mathbb{R}^m$ függvény differenciálható egy $a \in intD_f$ pontban
$\Rightarrow \exists! L \in L(\mathbb{R}^n, \mathbb{R}^m)$ korlátos lineáris leképezés

Ezt az egyértelműen létező $L \in L(\mathbb{R}^n, \mathbb{R}^m)$ korlátos lineáris leképezést az f függvény a pontbeli deriváltjának nevezzük, és $f'(a)$ szimbólummal jelöljük.

Jacobi-mátrix

Az előzőekben szereplő $L := f'(a)$ korlátos lineáris leképezéshez $\exists! A \in \mathbb{R}^{m\times n}$ mátrix, melyre:

$$L(x) = Ax \quad (x \in \mathbb{R}^n)$$

Ezért: $f'(a) := A$
az f függvény a-beli deriváltja vagy derivált mátrixa, más néven Jacobi-mátrixa.

Gradiens

$m = 1$ esetén : $f \in \mathbb{R}^n \to \mathbb{R}$

$$\mathrm{grad} f(a) := f'(a) \in \mathbb{R}^{1\times n} \approx \mathbb{R}^n$$

Tehát ebben az esetben az $f'(a)$ Jacobi-mátrix tekinthető egy $\mathbb{R}^n$-beli vektornak, amit az f függvény a-beli gradiensének nevezünk.

Ha $D := \{a \in D_f : f \in D\{a\}\}$, akkor az

$$x \mapsto \mathrm{grad} f(x) \in \mathbb{R}^n \quad (x \in D)$$

függvényt az f függvény gradiensének nevezzük, és $\mathrm{grad} f \in \mathbb{R}^n \to \mathbb{R}^n$ jelöljük.

Gradiens mint Jacobi-mátrix sora

Legyen $1 \le n, m \in \mathbb{N}$. Az $f = (f_1, ..., f_m) \in \mathbb{R}^n \to \mathbb{R}^m$ függvény akkor és csak akkor differenciálható az $a \in intD_f$ helyen, ha minden $i = 1, ..., m$ esetén az $f_i \in \mathbb{R}^n \to \mathbb{R}$ koordináta-függvény differenciálható az a-ban.

Ha $f \in D\{a\}$, akkor az $f'(a)$ Jacobi-mátrix a következő alakú:

$$f'(a) = \begin{bmatrix} \mathrm{grad} f_1(a) \\ \mathrm{grad} f_2(a) \\ \vdots \\ \mathrm{grad} f_m(a) \end{bmatrix}$$

1.2 Parciális derivált

Definíció

Tekintsük a $h \in \mathbb{R}^n \to \mathbb{R}$ függvényt és az $a = (a_1, ..., a_n) \in D_h$ vektort. Legyen

$$D_{h,i}^{(a)} := \{t \in \mathbb{R} : (a_1, ..., a_{i-1}, t, a_{i+1}, ..., a_n) \in D_h\} \quad (i = 1, ..., n)$$

És legyen:

$$h_{a,i} : D_{h,i}^{(a)} \to \mathbb{R}, \quad \text{melyre:} \quad h_{a,i}(t) := h(a_1, ..., a_{i-1}, t, a_{i+1}, ..., a_n) \quad (t \in D_{h,i}^{(a)})$$

A $h_{a,i}$ parciális függvények mind egyváltozós valós függvények ($h_{a,i} \in \mathbb{R} \to \mathbb{R}$)

A h függvény az a-ban i-edig változó szerint parciálisan deriválható, ha $h_{a,i} \in D\{a_i\}$. Ekkor:

$$\partial_i h(a) := h'_{a,i}(a_i)$$

valós számot a h függvény a-beli, i-edik változó szerinti parcális deriváltjának nevezzük.

Parciális derivált függvény

Tegyük fel, hogy az előző h függvényre:

$$D_{h,i} := \{a \in D_h : \text{létezik a } \partial_i h(a)\text{parciáis derivált}\} \neq \emptyset \qquad (i = 1, ..., n)$$

Ekkor a

$$x \mapsto \partial_i h(x) \quad (x \in D_{h,i})$$

függvényt a h függvény i-edik változó szerinti parciális deriváltfüggvényének nevezzük, és a $\partial_i h$ szimbólummal jelöljük.

Differenciálhatóság és parciális differenciálhatóság

- Differenciálhatóság $\Rightarrow$ parciális differenciálhatóság
  $1 \leq n \in \mathbb{N},\ h \in \mathbb{R}^n \to \mathbb{R}$, és $h \in D\{a\}$ $(a \in D_h)$
  $\Rightarrow \forall i = 1, ..., n$: a h függvény i-edik változó szerint parciálisan differenciálható az a pontban, és

$$\text{grad}h(a) = (\partial_1 h(a), ..., \partial_n h(a))$$

- Differenciálhatóság $\Leftarrow$ parciális differenciálhatóság
  $1 \leq n \in \mathbb{N},\ h \in \mathbb{R}^n \to \mathbb{R}$, és $a \in intD_h$
  Valamilyen $i = 1, ..., n$ esetén:
  - Tetszőleges $x \in K_r(a)$ $(r > 0$ alkalmas) helyen léteznek a $\partial_j h(x)$ parciális deriváltak $(i \neq j = 1, ..., n)$ és ezek folytonosak
  - $\exists\ \partial_i h(a)$ parciális derivált

  $\Rightarrow h \in D(a)$

2 Szélsőérték, függvényvizsgálat

2.1 Szélsőérték

$f \in \mathbb{R} \to \mathbb{R}, a \in D_f$

Lokális maximum
f-nek a-ban lokális maximuma van, ha alkalmas $r > 0$ mellett:
$f(x) \leq f(a) \qquad (x \in D_f, |x - a| < r)$

Lokális minimum
f-nek a-ban lokális minimuma van, ha alkalmas $r > 0$ mellett:
$f(x) \geq f(a) \qquad (x \in D_f, |x - a| < r)$

Abszolút maximum
f-nek a-ban abszolút maximuma van, ha:
$f(x) \leq f(a) \qquad (x \in D_f)$

Abszolút mimimum
f-nek a-ban abszolút minumuma van, ha:
$f(x) \geq f(a) \qquad (x \in D_f)$

Lokális szélsőérték
f-nek a-ban lokális szélsőértéke van, ha a-ban lokális minimuma vagy maximuma van.

Abszolút szélsőérték
f-nek a-ban abszolút szélsőértéke van, ha a-ban abszolút minimuma vagy maximuma van.

Elsőrendű szükséges feltétel (lokális szélsőértékre)
$f \in \mathbb{R} \to \mathbb{R}$ függvénynek $a \in intD_f$ helyen lokális szélsőértéke van, és $f \in D\{a\}$
$\Rightarrow f'\{a\} = 0$

Az előbbi tétel segítségével már nem nehéz belátni a differenciálható függvények vizsgálata szempontjából alapvető fontosságú ún. középérték-tételeket

Rolle-tétel
$a, b \in \mathbb{R}\ (a < b),\ f : [a, b] \to \mathbb{R},\ f \in C$, és
$\forall x \in (a, b) : f \in D\{x\}$ és $f(a) = f(b)$
$\Rightarrow \exists\ \xi \in (a, b) : f'(\xi) = 0$

Lagrange-féle középértéktétel
$a, b \in \mathbb{R}\ (a < b),\ f : [a, b] \to \mathbb{R},\ f \in C$, és
$\forall x \in (a, b) : f \in D\{x\}$
$\Rightarrow \exists\ \xi \in (a, b) : f'(\xi) = \dfrac{f(b) - f(a)}{b - a}$

Cauchy-féle középértéktétel

$a, b \in \mathbb{R}\ (a < b),\ f, g : [a, b] \to \mathbb{R},\ f, g \in C$, és
$\forall x \in (a, b) : f, g \in D\{x\}$
$\Rightarrow \exists\ \xi \in (a, b) :$
$(f(b) - f(a)) \cdot g'(\xi) = (g(b) - g(a) \cdot f'(\xi)$

Jelváltás

$f \in \mathbb{R} \to \mathbb{R},\ a \in intD_f$ és $f(a) = 0,\ (K_r(a) \subset D_f, r > 0)$

1. f függvénynek $(-, +)$ jelváltása van, ha
   $f(x) \leq 0 \leq f(t), \quad (x, t \in K_r(a),\ x < a < t)$
2. f függvénynek $(+, -)$ jelváltása van, ha
   $f(x) \geq 0 \geq f(t), \quad (x, t \in K_r(a),\ x < a < t)$

Elsőrendű elégséges feltétel

$f \in \mathbb{R} \to \mathbb{R},\ a \in intD_f$ és $f \in D\{x\},\ (x \in K_r(a) \subset D_f, r > 0)$

1. f'-nak az a-ban $(+, -)$ jelváltása van $\Rightarrow$ f-nek a-ban lokális maximuma van.
2. f'-nak az a-ban $(-, +)$ jelváltása van $\Rightarrow$ f-nek a-ban lokális minimuma van.

2.2 Monotonitás

$f \in \mathbb{R} \to \mathbb{R}$

Monoton növekedés

f monoton növő ($\nearrow$), ha $\forall x, t \in D_f,\ x < t : f(x) \leq f(t)$.
Amennyiben $f(x) < f(t)$, akkor f szigorúan monoton növő ($\uparrow$).

Monoton fogyás (csökkenés)

f monoton fogyó ($\searrow$), ha $\forall x, t \in D_f,\ x < t : f(x) \geq f(t)$.
Amennyiben $f(x) > f(t)$, akkor f szigorúan monoton fogyó ($\downarrow$).

Derivált és monotonitás kapcsolata

$I \subset \mathbb{R}$ nyílt intervallum, $f : I \to \mathbb{R},\ f \in D$
$\Rightarrow$

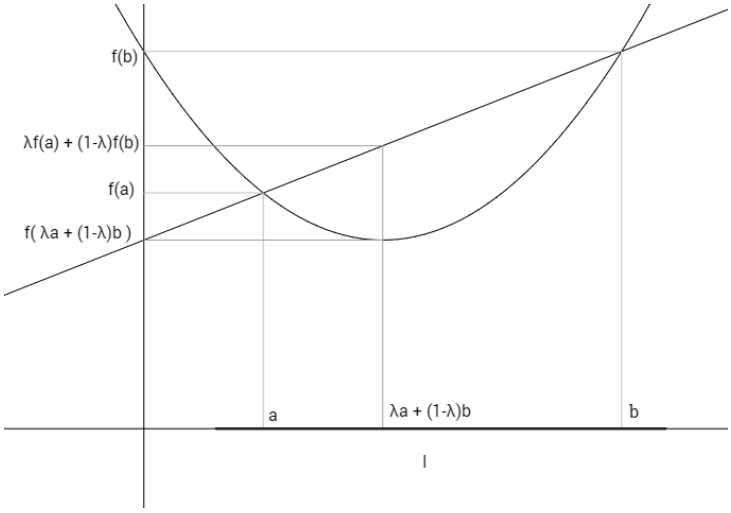
1. $f \nearrow \Leftrightarrow\ f' \geq 0$
2. $f \searrow \Leftrightarrow\ f' \leq 0$
3. $f\ konstans\ \Leftrightarrow\ \forall x \in I : f'(x) = 0$
4. $\forall x \in I : f'(x) > 0\ \Rightarrow\ f \uparrow$
5. $\forall x \in I : f'(x) < 0\ \Rightarrow\ f \downarrow$

2.3 Alaki viszonyok

Konvexitás, konkávitás

$I \subset \mathbb{R}$ intervallum, $f : I \to \mathbb{R}$

- f konvex, ha
  $\forall a, b \in I \quad \forall 0 \leq \lambda \leq 1 : f(\lambda a + (1 - \lambda)b) \leq \lambda f(a) + (1 - \lambda) f(b)$
- f konkáv, ha
  $\forall a, b \in I \quad \forall 0 \leq \lambda \leq 1 : f(\lambda a + (1 - \lambda)b) \geq \lambda f(a) + (1 - \lambda) f(b)$



ábra 1: Konvex függvény

Konvexitás és derivált

$I \subset \mathbb{R}$ nyílt intervallum, $f : I \to \mathbb{R}$, $f \in D$

- f konvex $\Leftrightarrow f' \nearrow$
- f konkáv $\Leftrightarrow f' \searrow$

Inflexió

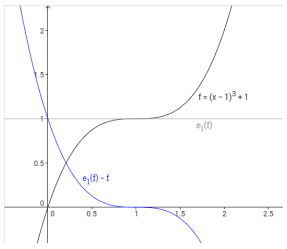
$f \in \mathbb{R} \to \mathbb{R}$, $a \in intD_f$, $f \in D\{a\}$:

Pontbeli érintő

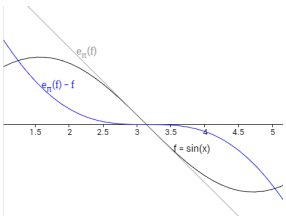
$e_a(x) := f(a) + f'(a)(x - a) \quad (x \in \mathbb{R})$

Inflexió

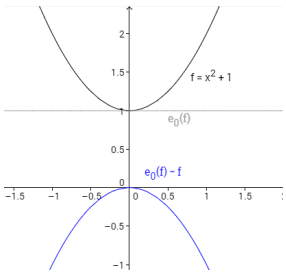
f-nek az a-ban inflexiója van, ha az $f - e_a(f)$ az a-ban jelet vált.



(a) x^3 függvény inflexiója



(b) szinusz függvény inflexiója



(c) x^2-nek nincs inflexiója

2.4 Többször differenciálható függvények

Második derivált

$f \in \mathbb{R} \to \mathbb{R}$, $a \in intD_f$, és
$f \in D\{x\} \quad (x \in K_r(a), r > 0)$, illetve $f' \in D\{a\}$
Ekkor f az a-ban kétszer deriválható és $f''(a) := (f')'(a)$ az f második deriváltja.

Differenciálás magasabb rendben

Hasonlóképpen az előzőhöz:
$f \in \mathbb{R} \to \mathbb{R}$, $a \in intD_f$, és
$f \in D^n\{x\} \quad (x \in K_r(a), r > 0)$, illetve $f^{(n)} \in D\{a\}$, $(1 \leq n \in \mathbb{N})$
Ekkor f az a-ban $(n+1)$-szer deriválható és $f^{(n+1)}(a) := (f^{(n)})'(a)$ az f $(n+1)$-edik deriváltja.

Másodrendű elégséges feltétel (lokális szélsőérték létezésére)

$f \in D^2\{a\}$ függvényre $f'(a) = 0$ és $f''(a) \neq 0$

$\Rightarrow f$-nek az a-ban szigorú lokális szélsőértéke van.
Ha $f''(a) < 0 \Rightarrow$ szigorú lokális maximum.
Ha $f''(a) > 0 \Rightarrow$ szigorú lokális minimum.

3 Riemann-integrál, parciális integrálás, integrálás helyettesítéssel.

Primitív függvény

$I \subset \mathbb{R}$ nyílt intervallum, $f \in I \to \mathbb{R}$
Ha $\exists F : I \to \mathbb{R}$, hogy $F \in D$, $F' = f$
akkor F az f primitív függvénye.

Határozatlan integrál

Legyen $\int f := \int f(x)dx := \{F : I \to \mathbb{R}, F \in D \text{ és } F' = f\}$ az f határozatlan integrálja.

Határozott integrál (Riemann-integrál)

$-\infty < a < b < \infty$, $f : [a, b] \to \mathbb{R}$, f korlátos

- A $\tau \subset [a, b]$ felosztása, ha τ véges és $a, b \in \tau$
  Ekkor $\tau = \{x_0, x_1, ..., x_n\}(n \in \mathbb{N})$, ahol $a := x_0 < x_1 < ... < x_n := b$
- $m_i := m_i(f) := inf\{f(x) : x_i \leq x \leq x_{i+1}\}(i = 0..n-1)$, illetve
  $M_i := M_i(f) := sup\{f(x) : x_i \leq x \leq x_{i+1}\}(i = 0..n-1)$
  és:
  $s(f, \tau) := \sum_{i=0}^{n-1} m_i(x_{i+1} - x_i)$ - alsó összeg
  $S(f, \tau) := \sum_{i=0}^{n-1} M_i(x_{i+1} - x_i)$ - felső összeg
- $\mathfrak{F} := \{\tau \subset [a, b]$ felosztás $\}$
- Az $\{s(f, \tau) : \tau \in \mathfrak{F}\}$ felülről korlátos és $\forall \mu \in \mathfrak{F} : S(f, \mu)$ felső korlát, illetve
  Az $\{S(f, \tau) : \tau \in \mathfrak{F}\}$ alulról korlátos és $\forall \mu \in \mathfrak{F} : s(f, \mu)$ alsó korlát
- Tehát legyen:
  $I_*(f) := sup\{s(f, \tau) : \tau \in \mathfrak{F}\}$ - Darboux alsó index, és
  $I^*(f) := inf\{S(f, \tau) : \tau \in \mathfrak{F}\}$ - Darboux felső index

$\Rightarrow \forall \tau, \mu \in \mathfrak{F} : s(f, \tau) \leq I_*(f) \leq I^*(f) \leq S(f, \mu)$

Az f függvény Riemann-integrálható ($f \in R[a, b]$), ha $I_*(f) = I^*(f)$, ekkor legyen
$\int_a^b f := \int_{[a,b]} f := \int_a^b f(x)dx := I_*(f) = I^*(f)$ az f függvény Riemann-integrálja (határozott integrálja).

Parciális integrálás

- Határozatlan esetben
  $I \subset \mathbb{R}$ nyílt intervallum, $f, g : I \to \mathbb{R}$, $f, g \in D$ és
  fg'-nek van primitív függvénye (azaz $\int fg' \neq \emptyset$)
  $\Rightarrow \int f'g \neq \emptyset$ és $\int f'g = fg - \int fg'$
- Határozott esetben
  $f, g \in D[a, b]$
  $f'g, fg' \in R[a, b]$
  $\Rightarrow \int_a^b f'g = f(b)g(b) - f(a)g(a) - \int_a^b fg'$

Integrálás helyettesítéssel

- Határozatlan esetben
  $I, J \subset \mathbb{R}$ nyílt intervallumok, $g : I \to J, g \in D, f : J \to \mathbb{R}, \int f \neq \emptyset$
  $\Rightarrow \int f \circ g \cdot g' \neq \emptyset$ és $(\int f) \circ g = \int (f \circ g \cdot g')$
- Határozott esetben
  $f \in C[a, b], g : [\alpha, \beta] \to [a, b], g \in C^1[\alpha, \beta]$,
  $g(\alpha) = a, g(\beta) = b$
  $\Rightarrow \int_a^b f = \int_\alpha^\beta (f \circ g \cdot g')$

4 Newton-Leibniz-formula

$f \in R[a,b], \exists F : [a,b] \to \mathbb{R}, F$ folytonos és $F \in D\{x\}, F'(x) = f(x), (a < x < b)$
$\Rightarrow \int_a^b f = F(b) - F(a)$

5 A kezdeti érték probléma

Differenciál egyenlet

$0 < n \in \mathbb{N},\ I \subset \mathbb{R}$ nyílt intervallum,
$\Omega := I_1 \times ... \times I_n \subset \mathbb{R}^n$, ahol $I_1, ..., I_n \subset \mathbb{R}$ nyílt intervallum
$f : I \times \Omega \to \mathbb{R}^n,\ f \in C$

Határozzuk meg a $\varphi \in I \to \Omega$ függvényt úgy, hogy:

- D_φ nyílt intervallum
- $\varphi \in D$
- $\varphi'(x) = f(x, \varphi(x)) \quad (x \in D_\varphi)$

Ezt a feladatot nevezzük differenciál egyenletnek.

Kezdeti érték probléma

Ha az előzőekhez még adottak: $\tau \in I$, és $\xi \in \Omega$
Illetve a φ függvényre még teljesül:

- $\tau \in D_\varphi$ és $\varphi(\tau) = \xi$

Akkor kezdeti érték problémának (Cauchy feladatnak) nevezzük.

6 Lineáris, ill. magasabb rendű lineáris differenciálegyenletek

6.1 Lineáris differenciálegyenletek

Definíció

A lineáris differenciálegyenlet olyan differenciálegyenlet, melyre:
$n = 1, \quad I, I_1 \subset \mathbb{R}$ nyílt intervallumok, $f : I \times I_1 \to \mathbb{R}$, ahol
$g, h : I \to \mathbb{R},\ g, h \in C,\ I_1 := \mathbb{R}$ és
$f(x,y) := g(x) \cdot y + h(x) \quad (x \in I, y \in I_1 = \mathbb{R})$
$\Rightarrow \varphi'(x) = f(x, \varphi(x)) = g(x) \cdot \varphi(x) + h(x) \quad (x \in D_\varphi)$

Homogenitás

A lineáris differenciálegyenlet homogén ha $h \equiv 0$ (különben inhomogén)

Kezdeti érték probléma

- Minden lineáris differenciálegyenletre vonatkozó kezdeti érték probléma megoldható és $\forall \varphi, \psi$ megoldásokra: $\varphi(t) = \psi(t) \quad (t \in D_\varphi \cap D_\psi)$
- Minden homogén lineáris differenciálegyenlet ($\varphi : I \to \mathbb{R}$) megoldása a következő alakú: $c\varphi_0$, ahol
  $c \in \mathbb{R}$ és $\varphi_0(t) = e^{G(t)} \quad (G : I \to \mathbb{R},\ G \in D$, és $G' = g)$
- Állandók variálásának módszere:
  $\exists m : I \to \mathbb{R},\ m \in D : m \cdot \varphi_0$ megoldása az (inhomogén) lineáris differenciálegyenletnek
- Partikuláris megoldás:
  $M := \{\varphi : I \to \mathbb{R} : \varphi'(t) = g(t) \cdot \varphi(t) + h(t)\ (t \in I)\}$
  $M_h := \{\varphi : I \to \mathbb{R} : \varphi'(t) = g(t) \cdot \varphi(t)\ (t \in I)\}$
  $\Rightarrow \forall \psi \in M : M = \psi + M_h = \{\varphi + \psi : \varphi \in M_h\}$
  (És itt ψ az előzőek alapján $m \cdot \varphi_0$ alakban írható)

- Példa: Radioaktív bomlás:
  $m_0 > 0$ - kezdeti anyagmennyiség
  $m \in \mathbb{R} \to \mathbb{R}$ - tömeg-idő függvénye, ahol
  $m(t)$ - a meglévő anyag mennyisége
  $m \in D \Rightarrow \dfrac{m(t) - m(t+\Delta t)}{\Delta t} \quad (\Delta t \neq 0)$ - átlagos bomlási sebesség
  $\dfrac{m(t) - m(t+\Delta t)}{\Delta t} \xrightarrow[\Delta t \to 0]{} -m'(t)$, ami megfigyelés alapján $\approx m(t)$

  azaz:
  $m'(t) = -\alpha \cdot m(t) \quad (t \in \mathbb{R}, 0 < \alpha \in \mathbb{R})$
  $m(0) = m_0$

  Homogén lineáris differenciálegyenlet (kezdeti érték probléma):
  $g \equiv -\alpha,\ \tau := 0,\ \xi := m_0$
  $\Rightarrow G(t) = -\alpha t \quad (t \in \mathbb{R}) \Rightarrow \varphi_0(t) = e^{-\alpha t} \quad (t \in \mathbb{R})$
  $\Rightarrow \exists c \in \mathbb{R} : m(t) = c \cdot e^{-\alpha t} \quad (t \in \mathbb{R})$, ahol
  $m(0) = c = m_0 \Longrightarrow m(t) = m_0 e^{-\alpha t} \quad (t \in \mathbb{R})$
  Ha $T \in \mathbb{R} : m(T) = \dfrac{m_0}{2}$ (felezési idő)
  $\Rightarrow \dfrac{m_0}{2} = m_0 e^{-\alpha T} \Rightarrow \dfrac{1}{2} = e^{-\alpha T} \Rightarrow e^{\alpha T} = 2$
  $\Rightarrow T = \dfrac{ln(2)}{\alpha}$

6.2 Magasabb rendű lineáris differenciálegyenletek

Definíció

$0 < n \in \mathbb{N}, I \subset \mathbb{R}$ nyílt, $a_0, ..., a_{n-1} : I \to \mathbb{R}$ folytonos és $c : I \to \mathbb{R}$ folytonos.
Keressünk olyan $\varphi \in I \to \mathbb{K}$ függvényt, melyre:

- $\varphi \in D^n$
- D_φ nyílt intervallum
- $\varphi^{(n)}(x) + \sum_{k=0}^{n-1} a_k(x) \cdot \varphi^{(k)}(x) = c(x) \quad (x \in D_\varphi)$

Ezt n-edrendű lineáris differenciálegyenletnek nevezzük. ($n = 1$ esetben Lineáris diff. egyenlet).
Ha még:
$\tau \in I,\ \xi_0, ..., \xi_{n-1} \in \mathbb{K}$ és

- $\tau \in D_\varphi$ és $\varphi^{(k)}(\tau) = \xi_k \quad (k = 0...n-1)$

Akkor Kezdeti érték problémáról beszélünk.

Homogenitás

Amennyiben $c(x) = 0$ homogén n-edrendű lineáris differenciálegyenletről beszélünk. Tehát homogén és inhomogén egyenletek megoldásainak halmazai:
$M_h := \{\varphi : I \to \mathbb{K} : \varphi \in D^n,\ \varphi^{(n)} + \sum_{k=0}^{n-1} a_k \cdot \varphi^{(k)} = 0\}$
$M := \{\varphi : I \to \mathbb{K} : \varphi \in D^n,\ \varphi^{(n)} + \sum_{k=0}^{n-1} a_k \cdot \varphi^{(k)} = c\}$
(Itt M_h n-dimenziós lineáris tér, így valamilyen $\varphi_1, ..., \varphi_n \in M_h$ bázist, más néven alaprendszert alkot.)

Állandó együtthatós eset

Ebben az esetben $a_0, ..., a_{n-1} \in \mathbb{R}$

- Karakterisztikus polinom szerepe
  Legyen $P(t) := t^n + \sum_{k=0}^{n-1} a_k t^k \quad (t \in \mathbb{K})$ karakterisztikus polinom és
  $\varphi_\lambda(x) := e^{\lambda x} \quad (x \in \mathbb{R}, \lambda \in \mathbb{K})$

Ekkor: $\varphi_\lambda \in M_h \iff P(\lambda) = 0$
Sőt ha λ r-szeres gyöke P-nek, és
$\varphi_{\lambda,j}(x) := x^j e^{\lambda x}$ $(j = 0..r-1, x \in \mathbb{R})$, akkor: $\varphi_{\lambda,j} \in M_h \iff \varphi_{\lambda,j}^{(n)} + \sum_{k=0}^{n-1} a_k \varphi_{\lambda,j}^{(k)}$
azaz $P(\lambda)^{(j)} = 0 \quad (j = 0..r-1)$

- Valós megoldások
  Legyen $\lambda = u + iv \quad (u, v \in \mathbb{R}, v \neq 0, i^2 = -1)$
  $\Rightarrow$ az $x \mapsto x^j e^{ux} cos(vx)$, és $x \mapsto x^j e^{ux} sin(vx)$ függvények valós alaprendszert (bázist) alkotnak (M_h-ban)

Példa: Rezgések

Írjuk le egy egyenes mentén, rögzített pont körül rezgőmozgást végző m tömegű tömegpont mozgását, ha ismerjük a megfigyelés kezdetekor elfoglalt helyét és az akkori sebességét!
$\varphi \in \mathbb{R} \to \mathbb{R}, \varphi \in D^2$: kitérés-idő függvény
$m > 0$: tömeg
$F \in \mathbb{R} \to \mathbb{R}$: kitérítő erő
$\alpha > 0$: visszatérítő erő, mely arányos φ-vel
$\beta \geq 0$: fékezőerő, mely arányos a sebességgel.
$\Longrightarrow$ (Newton-féle mozgástörvény alapján):
$m \cdot \varphi'' = F - \alpha\varphi - \beta\varphi'$
$\varphi(0) = s_0, \varphi'(0) = s_0'$

Másodrendű lineáris differenciál egyenlet (kezdeti érték probléma)
Standard alakba írva: $\varphi'' + \frac{\beta}{m}\varphi' + \frac{\alpha}{m}\varphi = \frac{F}{m}$
Tekintsük kényszerrezgésnek a periodikus külső kényszert, amikor:
$\frac{F(x)}{m} = Asin(\omega x) \quad [A > 0$ (amplitúdó), $\omega > 0$ (kényszerfrekvencia)$]$
Ekkor $\omega_0 := \sqrt{\frac{\beta}{m}}$ - saját frekvencia
és $\varphi''(x) + \omega_0^2 \varphi(x) = Asin(\omega x)$
Melynek karakterisztikus polinomja : $P(t) = t^2 + \omega_0^2 \quad (t \in \mathbb{R})$
Megoldásai: $\lambda = \pm\, \omega_0 i$

Korábban láttuk, hogy ha $\lambda = u + iv$ akkor $x \mapsto x^j e^{ux} cos(vx)$, és $x \mapsto x^j e^{ux} sin(vx)$ függvények valós alaprendszert (bázist) alkotnak (M_h-ban). Így $\varphi(x) = c_1 cos(\omega_0 x) + c_2 sin(\omega_0 x)$ alakban írható mely fázisszög segítségével: $d \cdot sin(\omega_0 x + \delta) \quad (d = \sqrt{c_1^2 + c_2^2}, \delta \in \mathbb{R})$ alakra átírható. Így:
$M_h = \{d \cdot sin(\omega_0 x + \delta)\}$

Ekkor már könnyen megadhatunk egy partikuláris megoldást:

- $\omega \neq \omega_0$ esetén partikuláris megoldás:
  $x \to q \cdot sin(\omega x)$
  És $q = \frac{A}{\omega_0^2 - \omega^2}$ kielégíti a $-q\omega^2 sin(\omega x) + \omega_0^2 q \cdot sin(\omega x) = Asin(\omega x)$ egyenletet. Tehát:
  $\varphi(x) = d \cdot sin(\omega_0 x + \delta) + \frac{A}{\omega_0^2 - \omega^2} sin(\omega x)$ megoldás két harmonikus rezgés összege.
- $\omega = \omega_0$ (rezonancia) esetén partikuláris megoldás:
  $x \to qx \cdot cos(\omega x)$
  És $q = \frac{-A}{2\omega}$ kielégíti a $-2q\omega \cdot sin(\omega x) - q\omega^2 x \cdot cos(\omega x) + \omega^2 qx \cdot cos(\omega x) = Asin(\omega x)$ egyenletet. Tehát:
  $\varphi(x) = d \cdot sin(\omega x + \delta) - \frac{A}{2\omega} x \cdot cos(\omega x)$ megoldás egy harmonikus és egy aperiodikus rezgés összege.
  (Ebben az esetben az idő (x) elteltével a φ értéke nő. Bizonyos modellekben ez a "rendszer szétesését" idézi elő)

Chapter 3

3

Záróvizsga tételsor
3. Numerikus módszerek

Ancsin Ádám

Numerikus módszerek
Iterációs módszerek: Lineáris egyenletrendszerekre és nemlineáris egyenletekre. Interpoláció: Lagrange-, Hermite- Spline interpoláció. Legkisebb négyzetek módszere.

1 Iterációs módszerek

1.1 Lineáris egyenletrendszerek iterációs módszerei

A lineáris egyenletrendszert (LER) vektorsorozatokkal közelítjük, törekedve a minél gyorsabb konvergenciára. Az iterációs módszereknek a lényege az $Ax = b \iff x = Bx + c$ átalakítás. Ilyen alak létezik, sőt nem egyértelmű, hanem sokféle lehet, és a különböző átalakítások szolgáltatják a különféle iterációs módszereket.

Definíció (kontrakció): Az $F : \mathbb{R}^n \to \mathbb{R}^n$ függvény kontrakció, ha $\exists 0 \leq q < 1 : \forall x, y \in \mathbb{R}^n$:

$$\|F(x) - F(y)\| \leq q\|x - y\|$$

.
A q értéket kontrakciós együtthatónak nevezzük.

Banach-féle fixponttétel: Legyen $F : \mathbb{R}^n \to \mathbb{R}^n$ kontrakció a q kontrakciós együtthatóval. Ekkor a következő állítások igazak:

1. $\exists! x^* \in \mathbb{R}^n : x^* = f(x^*)$. Azt mondjuk, hogy x^* az f függvény fixpontja.
2. $\forall x^{(0)} \in \mathbb{R}^n$ kezdőérték esetén az $x^{(k+1)} = f(x^{(k)})$ sorozat konvergens, és $\lim\limits_{k\to\infty} x^{(k)} = x^*$.
3. $\|x^{(k)} - x^*\| \leq \frac{q^k}{1-q}\|x^{(1)} - x^0\|$.
4. $\|x^{(k)} - x^*\| \leq q^k\|x^{(0)} - x^*\|$.

Vegyük észre, hogy az $Ax = b \iff x = Bx + c$ átírással megteremtettük a kapcsolatot a Banach-féle fixponttétellel, hisz most az $F(x) = Bx + c$ függvény fixpontját keressük. A fenti felírásban B-t átmenetmátrixnak nevezzük.

Tétel (elégséges feltétel a konvergenciára): Ha a LER B átmenetmátrixára $\|B\| < 1$, akkor tetszőleges $x^{(0)}$-ból indított $x^{(k+1)} := Bx^{(k)} + c$ iteráció konvergál az $Ax = b$ LER megoldásához.

Tétel (Szükséges és elégséges feltétel a konvergenciára): Tetszőleges $x^{(0)}$-ból indított $x^{(k+1)} := Bx^{(k)} + c$ iteráció konvergál az $Ax = b$ LER megoldásához $\iff \varrho(B) < 1$, ahol $\varrho(B) = \max_{1\leq i\leq n} |\lambda_i(B)|$ a B mátrix spektrálsugara.

1.1.1 Jacobi-iteráció

Tekintsük az $A \in \mathbb{R}^{n\times n}$ mátrix $L + D + U$ felbontását, ahol L a mátrix szigorú alsó része, U a szigorú felső része, D pedig a diagonális része. Ennek segítségével konstruáljuk meg a következő átírást:

$$Ax = b \iff (L + D + U)x = b \iff Dx = -(L + U)x + b \iff x = -D^{-1}(L + U)x + D^{-1}b$$

A Jacobi-iteráció átmenetmátrixa tehát $B_J = -D^{-1}(L+U)$, maga az iteráció pedig:

$$x^{(k+1)} := -D^{-1}(L+U)x^{(k)} + D^{-1}b$$

Koordinátás alakban felírva:

$$x_i^{(k+1)} := -\frac{1}{a_{ii}}\left[\sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j^{(k)} - b_i\right] \ (i = 1, ..., n)$$

Tétel: Ha az A mátrix szigorúan diagonálisan domináns a soraira, akkor $\|B_J\|_\infty < 1$ (azaz konvergens a módszer).

Tétel: Ha az A mátrix szigorúan diagonálisan domináns az oszlopaira, akkor $\|B_J\|_1 < 1$ (azaz konvergens a módszer).

1.1.2 Csillapított Jacobi-iteráció

Továbbra is a Jacobi-iterációval foglalkozunk, csak egy plusz ω paraméter bevezetésével próbáljuk finomítani a módszert. Tekintsük a $Dx = -(L+U)x + b$ egyenletet, valamint a triviális $Dx = Dx$ egyenletet. Ezeket rendre szorozzuk meg ω, illetve $1-\omega$ értékekkel, majd adjuk össze a két egyenletet:

$$Dx = (1-\omega)Dx - \omega(L+U)x + \omega b$$

Szorozzunk D^{-1}-zel:

$$x = (1-\omega)Ix - \omega D^{-1}(L+U)x + \omega D^{-1}b \iff x = ((1-\omega)I - \omega D^{-1}(L+U))x + \omega D^{-1}b$$

Ez alapján $B_{J(\omega)} = (1-\omega)I - \omega D^{-1}(L+U)$ és $c_J(\omega) = \omega D^{-1}b$.

Észrevehető, hogy $\omega = 1$ esetén pont a Jacobi-iterációt kapjuk vissza.

Koordinátás alakban felírva:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} - \frac{\omega}{a_{ii}}\left[\sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j^{(k)} - b_i\right] \ (i = 1, ..., n)$$

Tétel: Ha $J(1)$ konvergens, akkor $\omega \in (0,1)$-re $J(\omega)$ is az.

1.1.3 Gauss-Seidel iteráció

Egy másik lehetséges iteráció konstruálásának az ötlete a következő:

$$Ax = b \iff (L+D+U)x = b \iff (L+D)x = -Ux + b \iff x = -(L+D)^{-1}Ux + (L+D)^{-1}b$$

Ez az ötlet szüli a Gauss-Seidel iterációt, vagyis:

$$x^{(k+1)} := -(L+D)^{-1}Ux^{(k)} + (L+D)^{-1}b$$

Az iteráció átmenetmátrixa tehát $B_S = -(L+D)^{-1}U$. A koordinátás alak felírásához kicsit átírjuk az iterációt:

$$(L+D)x^{(k+1)} = -Ux^{(k)} + b$$

$$Dx^{(k+1)} = -Lx^{(k+1)} - Ux^{(k)} + b$$

$$x^{(k+1)} = -D^{-1}\left[Lx^{(k+1)} + Ux^{(k)} - b\right]$$

$$x_i^{(k+1)} = -\frac{1}{a_{ii}}\left[\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} - b_i\right] \ (i = 1, ..., n)$$

Megjegyzés: Az implementáció során elég egyetlen x vektort eltárolni, és annak a komponenseit sorban felülírni, ugyanis láthatjuk, hogy az első $i-1$ komponenst már az "új", $x^{(k+1)}$ vektorból vesszük.

Tétel: Ha A szigorúan diagonálisan domináns

1. a soraira, akkor $\|B_S\|_\infty \leq \|B_J\|_\infty < 1$.
2. az oszlopaira, akkor $\|B_S\|_1 \leq \|B_J\|_1 < 1$.

Azaz a Gauss-Seidel is konvergens, és legalább olyan gyors, mint a Jacobi.

1.1.4 Relaxációs módszer

A relaxációs módszer lényegében a csillapított Gauss-Seidel iterációt jelenti. Ennek megkonstruálásához tekintsük az $(L + D)x = -Ux + b$ és $Dx = Dx$ egyenleteket. Ezeket rendre szorozzuk meg ω, illetve $1 - \omega$ értékekkel, majd adjuk össze a két egyenletet:

$$(D + \omega L)x = (1 - \omega)Dx - \omega Ux + \omega b$$

$$x = (D + \omega L)^{-1}\big[(1 - \omega)D - \omega U\big]x + \omega(D + \omega L)^{-1}b$$

Az iteráció tehát: $x^{(k+1)} = (D+\omega L)^{-1}\big[(1-\omega)D-\omega U\big]x^{(k)}+\omega(D+\omega L)^{-1}b$, ahol az átmenetmátrix: $B_{S(\omega)} = (D + \omega L)^{-1}\big[(1 - \omega)D - \omega U\big]$. A koordinátás alak felírásához itt is átírjuk kicsit az iterációt:

$$(D + \omega L)x^{(k+1)} = (1 - \omega)Dx^{(k)} - \omega Ux^{(k)} + \omega b$$

$$Dx^{(k+1)} = -\omega Lx^{(k+1)} - \omega Ux^{(k)} + \omega b + (1 - \omega)Dx^{(k)}$$

$$x^{(k+1)} = -\omega D^{-1}\big[Lx^{(k+1)} + Ux^{(k)} - b\big] + (1 - \omega)x^{(k)}$$

$$x_i^{(k+1)} = -\frac{\omega}{a_{ii}}\left[\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} - b_i\right] + (1 - \omega)x_i^{(k)} \ (i = 1, ..., n)$$

Vegyük észre, hogy $\omega = 1$ esetén a Gauss-Seidel iterációt kapjuk.

Tétel: Ha a relaxációs módszer konvergens minden kezdővektorból indítva, akkor $\omega \in (0, 2)$.

Megjegyzés: Ha $\omega \notin (0, 2)$, akkor általában nem konvergens a módszer (bár adott feladat esetén előfordulhat, hogy találunk olyan kezdővektort, amelyből indítva konvergál a módszer).

Tétel: Ha A szimmetrikus és pozitív definit és $\omega \in (0, 2)$, akkor a relaxációs módszer konvergens. Ennek következménye a Gauss-Seidel iteráció konvergenciája ($\omega = 1$ eset).

Tétel: Ha A tridiagonális, akkor $\varrho(B_S) = \varrho(B_J)^2$, azaz a Jacobi és Gauss-Seidel iteráció egyszerre konvergens, illetve divergens.

Tétel: Ha A szimmetrikus, pozitív definit és tridiagonális, akkor a $J(1)$, $S(1)$ és $S(\omega)$ $\omega \in (0, 2)$- re konvergens, és $S(\omega)$-ra az optimális paraméter értéke:

$$\omega_0 = \frac{2}{1 + \sqrt{1 - \varrho(B_J)^2}}$$

1.1.5 Richardson-iteráció

Legyen $p \in \mathbb{R}$. Így

$$Ax = b \iff 0 = -Ax + b \iff 0 = -pAx + pb \iff x = (I - pA)x + pb$$

Az iteráció tehát $x^{(k+1)} := (I - pA)x^{(k)} + pb$. Az átmenetmátrix: $B_{R(p)} = I - pA$). Az $r^{(k)} := b - Ax^{(k)}$ vektort maradékvektornak (reziduumvektornak) nevezzük, hiszen

$$x^{(k+1)} = x^{(k)} - pAx^{(k)} + pb = x^{(k)} + pr^{(k)}$$

$$r^{(k+1)} = b - Ax^{(k+1)} = b - A(x^{(k)} + pr^{(k)}) = r^{(k)} - pAr^{(k)}$$

Tekintsük az előállítás algoritmusát: $r^{(0)} := b - Ax^{(0)}$, továbbá a fentiek miatt:

$$x^{(k+1)} := x^{(k)} + pr^{(k)}$$

$$r^{(k+1)} := r^{(k)} - pAr^{(k)}$$

Tétel: Ha A szimmetrikus, pozitív definit, a sajátértékei pedig a következők:

$$0 < m := \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n =: M$$

akkor $p \in (0, \frac{2}{M})$ esetén $R(p)$ konvergens, és az optimális paraméter: $p_0 = \frac{2}{m+M}$. Továbbá igaz, hogy: $\varrho(B_{R(p_0)}) = \frac{M-m}{M+m}$.

1.2 Nemlineáris egyenletek iterációs módszerei

Eddig egyenletrendszerekkel foglalkoztunk, melyekben minden egyenlet lineáris volt. Most módszereket fogunk keresni az $f(x) = 0$ típusú egyenletek megoldására, ahol $f \in \mathbb{R} \to \mathbb{R}$. A módszerek lényege az lesz, hogy valamilyen szempont szerint egy számsorozatot állítunk elő, melyek bizonyos feltételek mellett az egyenlet gyökéhez konvergálnak.

Bolzano-tétel: Legyen $f \in C[a, b]$ és $f(a)f(b) < 0$, azaz az f függvény az a és b pontokban nem 0, valamint ellenkező előjelű. Ekkor létezik (a, b) intervallumbeli gyöke az f-nek, azaz $\exists x^* \in (a, b) : f(x^*) = 0$.

A Bolzano-tétel következménye: Ha a Bolzano-tétel feltételei mellett még f szigorúan monoton is, akkor az x^* egyértelműen létezik (hiszen f invertálható).

Brouwer-féle fixponttétel: Legyen $f : [a, b] \to [a, b]$ és $f \in C[a, b]$. Ekkor $\exists x^* \in [a, b] : x^* = f(x^*)$.

Tétel: Legyen $f : [a, b] \to [a, b]$, $f \in C^1[a, b]$ és f' állandó előjelű. Ekkor $\exists! x^* \in [a, b] : x^* = f(x^*)$.

Fixponttétel [a,b]-re: Legyen $f : [a, b] \to [a, b]$ kontrakció a q kontrakciós együtthatóval. Ekkor:

1. $\exists! x^* \in [a, b] : x^* = f^{x*}$,
2. $\forall x_0 \in [a, b] : x_{k+1} = f(x_k)$ konvergens és $x^* = \lim_{k\to\infty} x_k$,
3. $|x_k - x^*| \leq q^k |x_0 - x^*| \leq q^k (b - a)$.

p-adrendű konvergencia: Az (x_k) konvergens sorozat ($\lim_{k\to\infty} x_k = x^*$) p-adrendben konvergens, ha

$$\lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^p} = c > 0$$

Néhány megjegyzés a fenti definícióhoz:

1. p egyértelmű és $p \geq 1$
2. $p = 1$ esetén lineáris $p = 2$ esetén kvadratikus, $1 < p < 2$ esetén szuperlineáris

3. A gyakorlatban az $|x_{k+1} - x^*| \leq M|x_k - x^*|^p$ alakot használják, azt jelenti, hogy legalább p-adrendben konvergens.

Tétel: Tegyük fel, hogy az (x_k) sorozat konvergens, $x_{k+1} = f(x_k)$ és $f'(x^*) = f''(x^*) = ... = f^{(p-1)}(x^*) = 0$, de $f^{(p)}(x^*) \neq 0$. Ekkor az (x_k) p-adrendben konvergens.

1.2.1 Newton-módszer

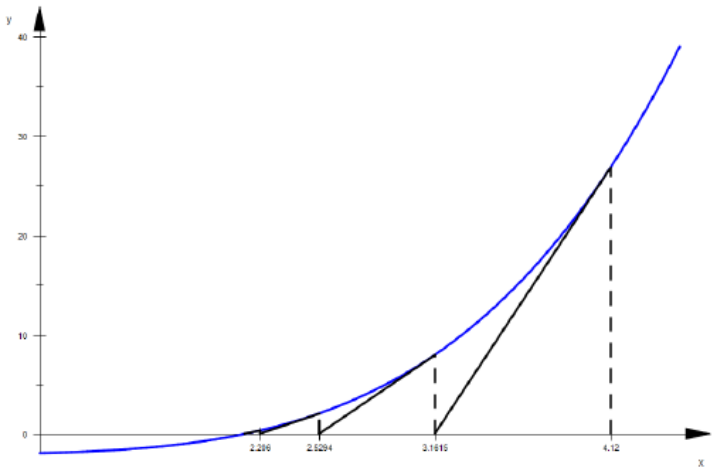


Figure 1: A Newton-módszer ötlete.

Az ábrán a legszélső, 4.12-es pontból indulunk, felvesszük a függvény ehhez a ponthoz tartozó érintőjét, majd ennek az érintőnek a gyöke lesz a következő pont, és így tovább. Általánosan, tekintsük az f függvény x_k ponthoz tartozó érintőjének egyenletét:

$$y - f(x_k) = f'(x_k)(x - x_k)$$

Mint mondtuk, az iteráció x_{k+1}. elemét az x_k-hoz tartozó érintő gyöke adja meg:

$$0 - f(x_k) = f'(x_k)(x_{k+1} - x_k) \Rightarrow -\frac{f(x_k)}{f'(x_k)} = x_{k+1} - x_k \Rightarrow x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

A fenti képlet a Newton-módszer képlete. Megjegyezhető, hogy a fenti módszer is $x_{k+1} = g(x_k)$ alakú (fixpontiteráció).

Fontos megemlíteni, hogy $f'(x_k) = 0$ esetén nem értelmezhető a módszer. A gyakorlatban $f'(x_k) \approx 0$ is probléma. Ha x^* többszörös gyök, akkor $f'(x^*) = 0$, vagyis x^* közelében $f'(x_k)$ egyre jobban közelít 0-hoz, numerikusan instabil.

Monoton konvergencia tétele: Tegyük fel, hogy $f \in C^2[a, b]$ és

1. $\exists x^* \in [a, b] : f(x^*) = 0$, azaz van gyök
2. f' és f'' állandó előjelű
3. $x_0 \in [a, b]$ legyen olyan, hogy $f(x_0)f''(x_0)) > 0$, azaz $f(x_0) \neq 0$, valamint $f(x_0)$ és $f''(x_0)$ azonos előjelűek

Ekkor az x_0-ból indított Newton-módszer monoton konvergál x^*-hoz.
Lokális konvergencia tétele: Tegyük fel, hogy $f \in C^2[a, b]$ és

1. $\exists x^* \in [a, b] : f(x^*) = 0$, azaz van gyök
2. f' állandó előjelű
3. $0 < m_1 \leq |f'(x)|(x \in [a, b])$ alsó korlát
4. $f''(x) \leq M_2(x \in [a, b])$ felső korlát, $M := \frac{M}{2m_1}$

5. $x_0 \in [a,b] : |x_0 - x^*| < r = \min\left\{\frac{1}{M}, |x^* - a|, |x^* - b|\right\}$

Ekkor az x_0-ból indított Newton-módszer másodrendben konvergens, hibabecslése:

$$|x_{k+1} - x^*| \leq M|x_0 - x^*|^2$$

1.2.2 Húrmódszer

Továbbra is $f(x) = 0$ megoldása a cél egy adott $[a,b]$ intervallumon. Az eljárás lényege a következő. Kezdetben $x_0 := a, x_1 := b$, majd meghúzzuk ezen pontok által képzett egyenest. Legyen x_2 a húr gyöke. Ha $f(x_2) = 0$, akkor megtaláltuk a gyököt. Ha $f(x_2) \neq 0$, akkor folytatjuk a keresést az $[x_0, x_2]$ vagy $[x_2, x_1]$ intervallumban. Ha $f(x_0)f(x_2) < 0$, akkor $[x_0, x_2]$ intervallumban folytatjuk, ha $f(x_2)f(x_1) < 0$, akkor $[x_2, x_1]$ intervallumban. Stb.

Általánosan: Legyen $x_0 := a, x_1 := b$ és $f(a)f(b) < 0$. Az $(x_k, f(x_k))$ és $(x_s, f(x_s))$ pontokon átmenő egyenesekkel közelítjük a függvényt ahol x_s-re $f(x_s)f(x_k) < 0$ és s a legnagyobb ilyen index. x_{k+1}-et a következőképpen határozhatjuk meg:

$$x_{k+1} := x_k - \frac{f(x_k)}{\frac{f(x_k)-f(x_s)}{x_k - x_s}} = x_k - \frac{f(x_k)(x_k - x_s)}{f(x_k) - f(x_s)}$$

Tétel: Legyen $f \in C^2[a,b]$ és

1. $f(a)f(b) < 0$
2. $M = \frac{M_2}{2m_1}$, ahol $0 < m_1 \leq |f'(x)|$ és $f''(x) \leq M_2 (x \in (a,b))$
3. $M(b-a) < 1$

Ekkor a húrmódszer konvergens, hibabecslése pedig:

$$|x_{k+1} - x^*| \leq \frac{1}{M}(M|x_0 - x^*|)^{k+1}$$

1.2.3 Szelőmódszer

A szelőmódszer lényege, hogy az $(x_k, f(x_k))$ és $(x_{k-1}, f(x_{k-1}))$ pontokon átmenő egyenessel közelítjük f-et, a kapott egyenes x tengellyel vett metszéspontja (x_{k+1}) lesz a következő pont. Ez tulajdonképpen a húrmódszer $s := k-1$-re.

$$x_{k+1} := x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}$$

Tétel: Ha teljesülnek a Newton-módszer lokális konvergencia tételének feltételei, akkor a szelőmódszer konvergens $p = \frac{1+\sqrt{5}}{2}$ rendben ,és hibabecslése:

$$|x_{k+1} - x^*| \leq M|x_k - x^*||x_{k-1} - x^*|$$

1.2.4 Többváltozós Newton-módszer

Most $F(x) = 0$ megoldásait keressük, ahol $F \in \mathbb{R}^n \to \mathbb{R}^n$. Tekintsük a többváltozós Newton-módszert:

$$x^{(k+1)} := x^{(k)} - [F'(x^{(k)})]^{-1} F(x^{(k)})$$

ahol

$$F(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ ... \\ f_n(x) \end{bmatrix}, F'(x) = \begin{bmatrix} \partial_1 f_1(x) & \partial_2 f_1(x) & ... \\ \partial_1 f_2(x) & \partial_2 f_2(x) & ... \\ ... & ... & ... \end{bmatrix}$$

Ténylegesen az $F'(x^{(k)})(x^{(k+1)} - x^{(k)}) = -F(x^{(k)})$ egyenletrendszert oldjuk meg.

2 Interpoláció

A gyakorlatban sokszor felmerül olyan probléma, hogy egy többségében ismeretlen (csak néhány pontbeli érték ismert) vagy nagyon költségesen kiszámítható függvénnyel kellene egy megadott intervallumon dolgoznunk. Ekkor például azt tehetjük, hogy néhány pontban kiszámítjuk a függvény értékét, majd keresünk olyan egyszerűbben számítható függvényt, amelyik illeszkedik az adott pontokra. Ezután az intervallum bármely további pontjában az illesztett függvény értékeit használjuk, mint az eredeti függvény értékeinek a közelítéseit. Ilyen egyszerűbben kiszámolható függvények pl. a polinomok (polinom interpoláció).

Példa alkalmazásra: Animáció készítésénél nem szeretnénk minden egyes képkockát saját magunk elkészíteni, hanem csak bizonyos képkockákat, ún. kulcskockákat. A köztes képkockákon az egyes objektumok helyzetét szeretnénk a számítógéppel kiszámíttatni (például szeretnénk, hogy ha egy objektum egyenes vonalú, egyenletes mozgást végezne két adott pozíció között).

2.1 Polinom interpoláció

2.1.1 A polinom interpoláció feladata

Legyenek adva $n \in \mathbb{N}$ és az $x_k \in \mathbb{R}, k = 0, 1, .., n$ különböző számok, az ún. interpolációs alappontok, valamint az $f(x_k), k = 0, 1, .., n$ számok, az ismert függvényértékek. Keressük azt a legfeljebb n-edfokú p_n polinomot ($p_n \in P_n$), amelyre:

$$p_n(x_k) = f(x_k) \quad k = 0, 1, ..., n$$

Azaz a keresett polinom az interpolációs alappontokban a megadott függvényértékeket veszi fel.

Tétel: A fenti interpolációs feladatnak egyértelműen létezik megoldása.

2.1.2 Lagrange-interpoláció

Az interpolációs polinom kiszámolására explicit képletet ad a Lagrange-interpoláció.

Lagrange-alappolinomok: Adott $n \in \mathbb{N}$ és $x_k \in \mathbb{R}, k = 0, 1, ..., n$ különböző alappontokra a Lagrange-alappolinomokat a következőképpen definiáljuk:

$$l_k(x) = \frac{\prod\limits_{\substack{j=0 \\ j \neq k}}^{n} (x - x_j)}{\prod\limits_{\substack{j=0 \\ j \neq k}}^{n} (x_k - x_j)}$$

$k = 0, 1, ..., n$ esetén.

Az alappontok mind n-edfokú polinomok, és a következő tulajdonsággal rendelkeznek:

$$l_k(x_j) = \left\{ \begin{array}{ll} 1 & ha\ j = k \\ 0 & ha\ j \neq k \end{array} \right\}$$

Tétel: Az interpolációs feladat megoldása az alábbi polinom, amelyet az interpolációs polinom Lagrange-alakjának hívunk:

$$L_n(x) = \sum_{k=0}^{n} f(x_k) l_k(x)$$

A továbbiakban jelölje $[a, b]$ az $x_0, x_1, ..., x_n$ alappontok által kifeszített intervallumot.

A Lagrange-interpoláció hibája: Ha $f \in C^{n+1}[a, b]$, akkor $\forall x \in [a, b]$ esetén

$$|f(x) - L_n(x)| \leq \frac{M_{n+1}}{(n+1)!} |\omega_n(x)|$$

ahol $\omega_n(x) = \prod_{i=0}^{n}(x - x_i)$, valamint $M_k = \max_{[a,b]} |f^{(k)}(x)|$.

Egyenletes konvergencia: Legyen $f \in C^\infty[a,b]$ és legyen adva egy $[a,b]$ intervallumbeli alappontrendszerek sorozata: $x_k^{(n)}$, $k = 0,1,...,n$, $n = 0,1,2,....$ Legyen L_n az $x_0^{(n)},...,x_n^{(n)}$ alappontrendszerre illesztett Lagrange-interpolációs polinom ($n = 0,1,2,...$). Ekkor ha $\exists M > 0$ úgy, hogy $M_n \leq M^n \; \forall n \in \mathbb{N}$, akkor az L_n sorozat egyenletesen konvergál az f függvényhez.

Marcinkiewicz tétele: Minden $f \in C[a,b]$ esetén létezik a fenti módon definiált alappontrendszer úgy, hogy $\|f - L_n\|_\infty \to 0$.

Faber tétele: Minden a fenti módon definiált alappontrendszer esetén van olyan $f \in C[a,b]$ függvény, hogy $\|f - L_n\|_\infty \not\to 0$.

Osztott differencia: Legyenek adva az $x_k \in [a,b], k = 0,1,...,n$ különböző alappontok. Az $f : [a,b] \to \mathbb{R}$ függvénynek a megadott alappontrendszerre vonatkozó elsőrendű osztott differenciái

$$f[x_i, x_{i+1}] = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

a magasabb rendű osztott differenciákat rekurzívan definiáljuk. Tegyük fel, hogy a $k-1$ rendű osztott differenciák már definiálva lettek, akkor a k-adrendű osztott differenciák az alábbiak:

$$f[x_i, x_{i+1}, ..., x_{i+k}] = \frac{f[x_{i+1}, x_{i+2}, ..., x_{i+k}] - f[x_i, x_{i+1}, ..., x_{i+k-1})]}{x_{i+k} - x_i}$$

Látható, hogy a k-adrenű osztott differencia $k+1$ alappontra támaszkodik.

Ha adott egy interpoláció alappontrendszer függvényértékekkel, akkor a hozzá tartozó osztott differenciákat az alábbi táblázat szerint érdemes elrendezni, és ez az elrendezés egyúttal a kiszámolást is segíti.

x_0	$f(x_0)$					
x_1	$f(x_1)$	$f[x_0,x_1]$				
x_2	$f(x_2)$	$f[x_1,x_2]$				
$\vdots$	$\vdots$					
x_k	$f(x_k)$	$f[x_{k-1},x_k]$	...	$f[x_0,x_1,...,x_k]$		
$\vdots$	$\vdots$					
x_{n-1}	$f(x_{n-1})$	$f[x_{n-2},x_{n-1}]$	...	$f[x_{n-1-k},x_{n-k},...,x_{n-1}]$	...	$f[x_0,x_1,...,x_{n-1}]$
x_n	$f(x_n)$	$f[x_{n-1},x_n]$	...	$f[x_{n-k},x_{n-k+1},...,x_n]$	...	$f[x_1,x_2,...,x_n]$ $\quad$ $f[x_0,x_1,...,x_n]$

Az interpolációs polinom Newton-alakja: Az interpolációs polinom az alábbi alakban felírható:

$$N_n(x) = f(x_0) + \sum_{k=1}^{n} f[x_0, x_1, ..., x_k]\omega_{k-1}(x)$$

ahol $\omega_j(x) = (x-x_0)(x-x_1)...(x-x_j)$. Ezt az alakot az interpolációs polinom Newton-alakjának hívjuk.

2.1.3 Hermite-interpoláció

Az előbbi interpolációs feladatot a következőképpen általánosíthatjuk. Legyenek adva az egyes alappontokban a függvényértékek mellett a függvény derivált értékei is valamely rendig bezárólag. Ekkor olyan polinomot keresünk, amelyik deriváltjaival együtt illeszkedik a megadott értékekre, vagyis:

Legyenek adva $n, m_0, m_1, ..., m_n \in \mathbb{N}$ és az $x_j \in \mathbb{R}, j = 0,1,...,n$ interpolációs alappontok, valamint az $f^{(k)}(x_j) \;\; k = 0,1,...,m_j - 1, \quad j = 0,1,...,n$ függvény- és derivált értékek. Legyen $m = \sum_{j=0}^{n} m_j$ Keressük azt a legfeljebb $(m-1)$-edfokú p_{m-1} polinomot, melyre:

$$p_{m-1}^{(k)}(x_j) = f^{(k)}(x_j) \;\; k = 0,1,...,m_j - 1, \quad j = 0,1,...,n$$

Megjegyzések:

1. Ha $m_j = 2, j = 0, 1, ..., n$, akkor a feladatot Hermite-Fejér-féle interpolációnak nevezzük. Ekkor minden alappontban a függvény- és az első derivált érték adott. A keresett polinom pedig legfeljebb $(2n + 1)$-edfokú.
2. Ha $m_j = 1, j = 0, 1, ..., n$, akkor a Lagrange-interpolációt kapjuk vissza.

Osztott differencia ismétlődő allapontokra: Ha x_k j-szer szerepel:

$$f[x_k, ...x_k] = \frac{f^{(j)}(x_k)}{j!}$$

Tétel: A Hermite-féle interpolációs polinom egyértelműen létezik.

A Hermite interpolációs polinom előállítása: Könnyen felírható a Newton-féle formában. Csak annyit kell tennünk, hogy kiindulunk az alappontok és a függvényértékek táblázatával és legyártjuk az osztott differenciák táblázatát. Az az egyetlen különbség most, hogy az x_j alappontot m_j-szer soroljuk fel.

Hermite-interpoláció hibája: Ha $f \in C^m[a, b]$, akkor $\forall x \in [a, b]$:

$$|f(x) - H_{m-1}(x)| \leq \frac{M_m}{m!}|\Omega_m(x)|$$

ahol $\Omega_m(x) = (x - x_0)^{m_0}(x - x_1)^{m_1}...(x - x_n)^{m_n}$

2.2 Spline-interpoláció

Az eddig említett interpolációs módszerekben polinomokkal dolgoztunk. Lehetőség van arra is, hogy a megadott pontrendszerre más típusú függvényt próbáljunk illeszteni. Igen előnyös tulajdonságokkal rendelkeznek a bizonyos folytonossági előírásoknak is megfelelő, szakaszonként polinom függvények, a spline-ok.

l-edfokú spline: Legyen adott $\Omega_n = \{x_0, x_1, ..., x_n\}$ az $[a, b]$ intervallum egy felosztása, ahol $x_0 = a, x_n = b$ és $l \in \mathbb{N}$. Az $s : [a, b] \to \mathbb{R}$ függvény egy l-edfokú spline az Ω_n-re vonatkozóan, ha:

1. $s_{|[x_{k-1}, x_k]}$ egy l-edfokú polinom $\forall k = 1, .., n$
2. $s \in C^{l-1}[a, b]$, tehát a teljes intervallumon $(l - 1)$-szer folytonosan derviálható

Jelölés: $s_l(\Omega_n)$ az Ω_n-hez tartozó l-edfokú spline-ok halmaza.

Spline-interpoláció: Legyenek adottak $x_k, f(x_k)$ értékek $k = 0, 1, .., n$-re és $l \in \mathbb{N}$. Keressük azt az $s \in s_l(\Omega_n)$ spline-t, amelyre $s(x_k) = f(x_k)$. Ehhez elő kell állítanunk minden intervallumra egy l-edfokú polinomot. Ha a polinomokat az együtthatóikkal reprezentáljuk, akkor ez $n(l + 1)$ ismeretlen. Az előírt feltételek száma: $2n$ interpolációs és $(l - 1)(n - 1)$ folytonossági feltétel, hiszen csak a belső pontokban kell előírni az illető deriváltakra vonatkozó megfelelő folytonossági feltételt. Az így kapott összes feltétel darabszáma $(l + 1)n - (l - 1)$, tehát az egyértelműséghez $l - 1$ feltétel hiányzik még. Ezeket úgynevezett peremfeltételekkel adjuk meg. Pl. a harmadfokú spline-interpolációhoz 2 peremfeltétel szükséges. Ezek a következők (ezek közül elég egyet választani, mert mindegyik 2 feltételt tartalmaz):

1. Természetes peremfeltétel: $s''(a) = s''(b) = 0$.
2. Hermite-féle peremfeltétel: $s'(a) = s_a, s'(b) = s_b$, ahol s_a, s_b előre megadott számok.
3. Periodikus peremfeltétel (ekkor feltételezzük, hogy $s(a) = s(b)$ is teljesül): $s'(a) = s'(b)$ és $s''(a) = s''(b)$

Elsőfokú spline előállítása: Az elsőfokú spline előállítása triviális szakaszonkénti lineáris Lagrange-interpolációval.

Másodfokú spline előállítása: Egyetlen peremfeltétel szükséges, legyen a következő: $s'(a) = s_a$ valamilyen s_a számra. Az $[x_0, x_1]$ szakaszon Hermite-interpolációval előállítjuk azt a H_2 másodfokú polinomot,

amely megfelel az interpolációs feltételeknek és a peremfeltételnek. Az így kapott polinom x_1-beli deriváltja meghatározott, tehát a folytonos deriválhatóság miatt az $[x_1, x_2]$ szakaszon a bal végpontban adott a derivált értéke. Ismét Hermite-interpolációt alkalmazva megkapjuk az $[x_1, x_2]$ szakaszhoz tartozó polinomot. Ezt az eljárást ismételve állíthatjuk elő a másodfokú interpolációs spline-t.

Függvény tartója: A $supp(f) := \overline{\{x \in \mathbb{R} : f(x) \neq 0\}}$ halmazt az f függvény tartójának nevezzük.

Számegyenes felosztása: $\Omega_\infty := \{..., x_{-2}, x_{-1}, x_0, x_1, ...x_n, x_{n+1}, ...\}$

B-spline: A $B_{l,k}, k \in \mathbb{Z}$ l-edfokú spline függvények rendszerét B-spline függvényeknek nevezzük, ha az alábbi feltételek teljesülnek:

1. $supp(B_{l,k}) = [x_k, x_{k+l+1}]$, azaz a tartója minimális
2. $B_{l,k}(x) \geq 0$
3. $\sum_{k \in \mathbb{Z}} B_{l,k}(x) = 1$

3 Legkisebb négyzetek módszere

Gyakorlati feladatok során adódik a következő probléma. Egy elsőfokú függvényt mérünk bizonyos pontokban, de a mérési hibák miatt ezek nem lesznek egye egyenesen. Ekkor olyan egyenest keresünk, amelyik az alábbi értelemben legjobban illeszkedik a megadott mérési ponthalmazra.

Legyenek adva az $(x_i, y_i), i = 1, 2, .., m$ mérési pontok. Keressük azt a $p_1(x) = a + bx$ legfeljebb elsőfokú polinomot, amelyre a

$$\sum_{i=1}^{m} (y_i - p_1(x_i))^2 = \sum_{i=1}^{m} (y_i - a - bx_i)^2$$

kifejezés minimális. Ez azt jelenti, hogy azt az egyenes keressük, amelyre a függvényértékek hibáinak négyzetösszege minimális.

Az általános feladat az alábbi.

Adottak az $m, n \in \mathbb{N}$, ahol $m >> n$ és $(x_i, y_i), i = 1, 2, ..., m$ mérési pontok, ahol az x_i alappontok különbözők. Keressük azt a $p_n(x) = a_0 + a_1 x + ...a_n x^n$ legfeljebb n-edfokú polinomot, melyre a

$$\sum_{i=1}^{m} (y_i - p_n(x_i))^2$$

kifejezés minimális.

A feladat megoldásához tekintsük annak egy átfogalmazását.

Vegyük a $p_n(x_i) = y_i, i = 1, 2, ..., m)$ egyenletrendszert. Ez a rendszer az ismeretlen a_i együtthatókra nézve lineáris, mégpedig túlhatározott, amelynek az A mátrixa egy téglalap alakú Vandermonde-mátrix $A \in \mathbb{R}^{m \times (n+1)}$, a $b \in \mathbb{R}^m$ jobb oldali vektora pedig a függvényértékekből adódik:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & ... & x_1^n \\ 1 & x_2 & x_2^2 & ... & x_2^n \\ 1 & x_3 & x_3^2 & ... & x_3^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 & ... & x_m^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

Ezen jelölésekkel a minimalizálandó kifejezés $\|Az - b\|_2^2$, ahol $z = [a_0, a_1, ..., a_n]^T$ a keresett együtthatók vektora. A feladat megoldását a Gauss-féle normálegyenletek adják:

$$A^T A z = A^T b$$

A fenti LER-t kell megoldani z-re.

n=1 eset: Ekkor a feladatot gyakran lineáris regressziónak is hívjuk. Ebben az esetben

$$A^T A = \begin{bmatrix} m & \sum_{i=1}^m x_i \\ \sum_{i=1}^m x_i & \sum_{i=1}^m x_i^2 \end{bmatrix}, A^T b = \begin{bmatrix} \sum_{i=1}^m y_i \\ \sum_{i=1}^m x_i y_i \end{bmatrix}, z = \begin{bmatrix} b \\ a \end{bmatrix}$$

Chapter 4

4

Záróvizsga tételsor
4. Számelmélet, gráfok, kódoláselmélet

Dobreff András

Számelmélet, gráfok, kódoláselmélet
Relációk, rendezések. Függvények és műveletek. Számfogalom, komplex számok. Leszámlálások véges halmazokon. Számelméleti alapfogalmak, lineáris kongruencia-egyenletek. Általános és síkgráfok, fák, Euler- és Hamilton-gráfok, gráfok adatszerkezetei. Polinomok és műveleteik, maradékos osztás, Horner-séma. Betűnkénti kódolás, Shannon- és Huffman-kód. Hibajavító kódok, kódtávolság. Lineáris kódok.

1 Számelmélet

1.1 Relációk, rendezések

Alapfogalmak

- Rendezett pár
  (x, y) rendezett pár, ha $(x, y) = (u, v) \iff x = u \ \wedge \ y = v$. Ezt a tulajdonságot halmazokkal definiáljuk:
  $$(x, y) := \{\{x\}, \{x, y\}\}$$
- Descartes-szorzat
  X, Y halmazok Descartes-szorzata vagy direkt szorzata:
  $$X \times Y := \{(x, y) : x \in X, y \in Y\}$$
- Binér reláció
  Egy halmazt binér relációnak nevezünk, ha minden eleme rendezett pár. Ha R binér reláció és $(x, y) \in R$, akkor gyakran írjuk: xRy
- Reláció
  Ha X, Y halmazokra $R \subset X \times Y$, akkor R reláció X és Y között.
- Értelmezési tartomány
  Az R binér reláció értelmezési tartománya:
  $$\mathrm{dmn}(R) := \{x \mid \exists y : (x, y) \in R\}$$
- Érték készlet
  Az R binér reláció érték készlete:
  $$\mathrm{rng}(R) := \{y \mid \exists x : (x, y) \in R\}$$
- Inverz
  Egy R binér reláció inverze:
  $$R^{-1} := \{(a, b) : (b, a) \in R\}$$
- Halmaz képe
  Legyen R binér reláció, és A halmaz. Az A halmaz képe:
  $$R(A) := \{y \mid \exists x \in A : (x, y) \in R\}$$
- Kompozíció
  R és S binér relációk kompozíciója:
  $$R \circ S := \{(x, y) \mid \exists z : (x, z) \in S \ \wedge \ (z, y) \in R \ \}$$

Tulajdonságok

Az R egy X-beli binér reláció (azaz $R \subset X \times X$)

1. tranzitív

$$\forall x, y, z : (x, y) \in R \ \wedge \ (y, z) \in R \Longrightarrow (x, z) \in R$$

2. szimmetrikus

$$\forall x, y : (x, y) \in R \Longrightarrow (y, x) \in R$$

3. antiszimmetrikus

$$\forall x, y : (x, y) \in R \ \wedge \ (y, x) \in R \Longrightarrow x = y$$

4. szigorúan antiszimmetrikus

$$\forall x, y : (x, y) \in R \Longrightarrow (y, x) \notin R$$

5. reflexív

$$\forall x \in X : (x, x) \in R$$

6. irreflexív

$$\forall x \in X : (x, x) \notin R$$

7. trichotóm
   Ha minden $x, y \in X$ esetén az alábbiak közül pontosan egy teljesül
   a) $x = y$
   b) $(x, y) \in R$
   c) $(y, x) \in R$
8. dichotóm

$$\forall x, y \in X : (x, y) \in R \ \vee \ (y, x) \in R$$

   Más néven az elemek összehasonlíthatóak.

Rendezések

- Ekvivalenciareláció, osztályozás
  X halmaz, R X-beli binér reláció ekvivalenciareláció, ha
  - Reflexív
  - Tranzitív
  - Szimmetrikus

  X részhalmazainak egy $\mathcal{O}$ rendszerét osztályozásnak hívjuk, ha $\mathcal{O}$ páronként diszjunkt nemüres halmazokból álló halmazrendszer, melyre $\cup\mathcal{O} = X$

  Tétel:
  Egy ekvivalenciareláció meghatároz egy osztályozást. Fordítva: $\mathcal{O}$ osztályozásra $R = \cup\{Y \times Y : Y \in \mathcal{O}\}$ ekvivalenciareláció.
- Részbenrendezés
  X halmaz, R X-beli binér reláció részbenrendezés, ha
  - Reflexív
  - Tranzitív
  - Antiszimmetrikus
- Teljes rendezés
  X halmaz, R X-beli binér reláció (teljes) rendezés, ha
  - Reflexív

– Tranzitív
– Antiszimmetrikus
– Dichotóm

Magyarul ha egy részbenrendezés dichotóm (tehát minden eleme összehasonlítható), akkor (teljes) rendezés.

- Szigorú és gyenge reláció, rendezés
X halmaz, R,S relációk X-beliek. Ha

$$xRy \ \wedge \ x \neq y \Rightarrow xSy$$

akkor S-et az R szigorításának nevezzük.
Megfordítva, ha

$$xRy \ \vee \ x = y \Rightarrow xTy$$

akkor T az R-hez megfelelő gyenge reláció.

Megjegyzés: Tulajdonképpen a reflexívitás elvételéről és hozzáadásáról van szó. Egy részbenrendezés esetén a megfelelő szigorú reláció (szigorú részbenrendezés) tehát irreflexív, következésképpen szigorúan antiszimmetrikus is. Megfordítva: Egy X-beli szigorú részbenrendezés (tran., irrefl., szig. ant.) megfelelő gyenge relációja részbenrendezés.

Korlátok

- Legkisebb, legnagyobb, minimális, maximális elem
X halmazbeli részbenrendezés ($\preccurlyeq$) legkisebb (legelső) elemén egy olyan $x \in X$ elemet értünk, melyre: $\forall y \in X : x \preccurlyeq y$. (Ilyen nem biztos, hogy létezik, de ha igen, akkor egyértelmű).
Hasonlóan a legnagyobb (utolsó) elem olyan $x \in X$, hogy $\forall y \in X : y \preccurlyeq x$.

x-et minimálisnak nevezzük, ha nincs nála kisebb elem, maximálisnak, ha nincs nála nagyobb elem. (Szemben a legkisebb/legnagyobb elemekkel, minimális/maximális elemből több is lehet. Ha viszont X rendezett, akkor legkisebb=minimális, legnagyobb=maximális.)

- Alsó, felső korlát
X részbenrendezett halmaz, $Y \subset X$. Az $x \in X$ elem az Y alsó korlátja $\forall y \in Y : x \preccurlyeq y$. (felső korlátja: $\forall y \in Y : y \preccurlyeq x$). Látható, hogy x nem feltétlenül eleme Y-nak, sőt az is lehet, hogy Y-nak nincs alsó/felső korlátja, vagy akár több is van. Ha azonban $x \in Y$, akkor egyértelmű és ez Y legkisebb eleme.

- Infimum, szuprémum
Ha az alsó korlátok között van legnagyobb elem, azt Y alsó határának, infimumának nevezzük. (Jele: inf$Y$)
Ha a felső korlátok között van legnagyobb elem, azt Y felső határának, szuprémumának nevezzük. (Jele: sup$Y$)

- Alsó, felső határ tulajdonság
X részbenrendezett halmaz. Ha $\forall \emptyset \neq Y \subset X : Y$ felülről korlátos és van szuprémuma, akkor felső határ tulajdonságú. Illetve ha $\forall \emptyset \neq Y \subset X : Y$ alulról korlátos és van infimuma, akkor alsó határ tulajdonságú.

1.2 Függvények és műveletek

1.2.1 Függvények

Definíció

Egy f reláció függvény, ha

$$(x, y) \in f \ \wedge (x, y') \ \in f \Longrightarrow y = y'$$

Más szóval minden x-hez legfeljebb egy olyan y létezik, hogy $(x, y) \in f$

Így minden $x \in \mathrm{dmn}(f)$-re az $f(x) = \{y\}$, melyet $f(x) = y$ vagy $f : x \mapsto y$ vagy $f_x = y$ is szoktunk jelölni.

Értelmezési tartomány, értékkészlet

Az $f : X \to Y$ jelölést használjuk, ha $\mathrm{dmn}(f) = X$.
Az $f \in X \to Y$ jelölést használjuk, ha $\mathrm{dmn}(f) \subset X$ (amikor $\mathrm{dmn}(f) \subsetneq X$ is előfordulhat).
Mindkét esetben $\mathrm{rng}(f) \subset Y$.

Injektív

f függvény kölcsönösen egyértelmű/injektív, ha

$$f(x) = y \ \wedge \ f(x') = y \implies x = x'$$

Ez azzal ekvivalens, hogy f^{-1} reláció is függvény.

Szürjektív

Az f függvény szürjektív, ha

$$\forall y \in Y : \exists x \in X : f(x) = y$$

Azaz $\mathrm{rng}(f) = Y$. Magyarul az f függvény az egész Y-ra képez.

Bijektív

Ha az f függvény injektív és szürjektív, akkor bijektív.

Indexelt család

Az x függvény i helyen felvett értékét x_i-vel is szoktuk jelölni. Ilyenkor gyakran $\mathrm{dmn}(f) = I$ értelmezési tartományt indexhalmaznak, elemeit indexeknek, $\mathrm{rng}(f)$-et indexelt halmaznak, és magát az x függvényt indexelt családnak szoktuk nevezni.

1.2.2 Műveletek

Definíciók

- Binér művelet
  X halmazon egy $f : X \times X \to X$ függvény binér művelet.
- Unér művelet
  X halmazon egy $f : X \to X$ függvény unér művelet.
- Nullér művelet
  X halmaz, $f : \{\emptyset\} \to X$ nullér művelet. (Gyakorlatilag elemkiválasztás)

Tulajdonságok

- Legyen $\spadesuit, ©$ binér műveletek X-en.
  1. $\spadesuit$ asszociatív, ha
     $$\forall x, y, z \in X : \quad (x \spadesuit y) \spadesuit z = x \spadesuit (y \spadesuit z)$$
  2. $\spadesuit$ kommutatív, ha
     $$\forall x, y \in X : \quad x \spadesuit y = y \spadesuit x$$
  3. $\spadesuit$ disztributív a ©-ra, ha $\forall x, y, z \in X$:
     $$x \spadesuit (y \copyright z) = (x \spadesuit y) \copyright (x \spadesuit z) \quad \text{- baloldali}$$
     $$(y \copyright z) \spadesuit x = (y \spadesuit x) \copyright (z \spadesuit x) \quad \text{- jobboldali}$$
- Legyen $\heartsuit$ binér művelet X-en és $\S$ binér művelet Y-on $f : X \to Y$ művelettartó ha:
  $$\forall x_1, x_2 \in X : f(x_1 \heartsuit x_2) = f(x_1) \ \S \ f(x_2)$$

1.3 Számfogalom, komplex számok

1.3.1 Számfogalom

Algebrai Struktúrák

1. Grupoid
   G halmaz egy $\star$ művelettel, azaz a $(G, \star)$ párt grupoidnak nevezzük.
2. Félcsoport
   Ha egy grupoidban a $\star$ művelet asszociatív, akkor a grupoid félcsoport.
3. Monoid
   Semleges elemes félcsoportot monoidnak nevezzük.
   Megyjegyzés: $a \in G$ semeleges elem, ha $\forall g \in G : a \star g = g \star a = g$
4. Csoport
   Ha egy monoidban minden elemnek van inverze, akkor csoportról beszélünk.
   Megyjegyzés: $g, g^{-1} \in G$ és $\xi \in G$ semleges elem, akkor a g^{-1} a g inverze, ha $g \star g^{-1} = \xi$ és $g^{-1} \star g = \xi$
5. Ábel-csoport
   Ha egy csoportban a művelet kommutatív, akkor Abel-csoport.
6. Gyűrű
   $(R, +, \cdot)$ gyűrű, ha az összeadással Abel-csoport, a szorzással félcsoport és teljesül mindkét oldali disztributivitás.
   Ha a szorzás kommutatív, akkor kommutatív gyűrű.
   Ha a szorzásnak van egységeleme, akkor egységelemes gyűrű.
7. Integritási tartomány
   Nullosztó mentes kommutatív gyűrű.
   Nullosztó: x, y nullától különböző elemek, de $x \cdot y = 0$
8. Rendezett integritási tartomány
   R integritási tartomány rendezett integritási tartomány, ha rendezett halmaz, továbbá az összeadás és szorzás monoton.
   Összeadás monoton: $x, y, z \in R$ és $x \leq y \Rightarrow x + z \leq y + z$
   Szorzás monoton: $x, y \in R$ és $x, y \geq 0 \Rightarrow x \cdot y \geq 0$
9. Test
   Egy R gyűrűt, ha $R \setminus \{0\}$ szorzással Abel-csoport, akkor test.
10. Rendezett test
    Ha egy test rendezett integritási tartomány, akkor rendezett test.

Természetes számok

- Peano-axiómák
  Legyen $\mathbb{N}$ egy halmaz és a $^+$ egy $\mathbb{N}$-en értelmezett függvény. Az alábbi feltételeket Peano-axiómáknak nevezzük:
  1. $0 \in \mathbb{N}$ - 0 egy nullér művelet $\mathbb{N}$-en
  2. ha $n \in \mathbb{N}$, akkor $n^+ \in N$ - $^+$ egy unér művelet $\mathbb{N}$-en
  3. ha $n \in \mathbb{N}$, akkor $n^+ \neq 0$ - 0 nincs a $^+$ értékkészletében
  4. ha $n, m \in \mathbb{N}$, és $m^+ = n^+$, akkor $n = m$ - $^+$ injektív
  5. ha $S \subset \mathbb{N}, 0 \in S$, továbbá $n \in S : n^+ \in S$, akkor $S = \mathbb{N}$ - a matematikai indukció elve
- Műveletek
  - összeadás
    $k, m, n \in \mathbb{N}$, akkor:
    1. $(k + m) + n = k + (m + n)$ - *asszociativitás*
    2. $n + 0 = 0 + n = n$ - *0 a nullelem (additív semleges elem)*
    3. $n + k = k + n$ - *kommutativitás*
    4. $n + k = m + k$ vagy $k + n = k + m$, akkor $m = n$ - *egyszerűsítési szabály*

– szorzás

$k, m, n \in \mathbb{N}$, akkor:

1. $(k \cdot m) \cdot n = k \cdot (m \cdot n)$ - *asszociativitás*
2. $0 \cdot n = n \cdot 0 = 0$
3. $n \cdot 1 = 1 \cdot n = n$ - *1 az egységelem (multiplikatív semleges elem)*
4. $n \cdot k = k \cdot n$ - *kommutativitás*
5. $k \cdot (m + n) = k \cdot m + \cdot n$, illetve $(m + n) \cdot k = m \cdot k + n \cdot k$ - *disztributivitás*
6. $k \neq 0$ esetén: $n \cdot k = m \cdot k$, akkor $m = n$ - *egyszerűsítési szabály*

Egész számok

Természetes számok körében az összeadásra nézve csak a nullának van inverze, másként szólva, a kivonás általában nem végezhető el.

Tekintsük a $\sim \subset \mathbb{N} \times \mathbb{N}$ relációt, melyre $(m, n) \sim (m', n')$, ha $m+n' = m'+n$. És vegyük az $(m, n)+(m', n') = (m + m', n + n')$ összeadást. A $\sim$ reláció ekvivalenciareláció, az ekvivalenciaosztályok halmazát jelöljük $\mathbb{Z}$-vel. $\mathbb{Z}$ elemeit egész számoknak nevezzük.

Az összeadás kompatibilis az ekvivalenciával, így az egész számok között értelmezve van, és $(\mathbb{Z}, +)$ Ábel-csoport.

Tehát $(\mathbb{Z}, +, \cdot)$ gyűrű.

Megjegyzés: $*$ *művelet kompatibilis a* $\asymp$ *ekvivalenciarelációval, ha teljesül:* $x \asymp x' \ \wedge \ y \asymp y' \implies x * y \asymp x' * y'$

Racionális számok

Az egész számok körében a nem nulla elemek közül csak az 1-nek és a -1-nek van multiplikatív inverze, másként szólva az osztás általában nem végezhető el.

Tekintsük a $\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$-n a $\sim$ relációt, melyre $(m, n) \sim (m', n')$, ha $mn' = nm'$. És vegyük az $(m, n) + (m', n') = (mn' + nm', nn')$ összeadást és az $(m, n) \cdot (m', n') = (mm', nn')$ szorzást. A $\sim$ reláció ekvivalenciareláció, az ekvivalenciaosztályok halmazát jelöljük $\mathbb{Q}$-val. $\mathbb{Q}$ elemeit racionális számoknak nevezzük.

$(\mathbb{Q}, +, \cdot)$ rendezett test.

Valós számok

Nincs olyan $a \in \mathbb{Q}$ szám, melynek négyzete 2. Tehát nem minden szám írható fel m/n $(m, n \in \mathbb{N}^+)$ alakban.

Archimédeszi rendezettség:
Egy F rendezett testet archimédeszien rendezett, ha $x, y \in F : \exists n \in \mathbb{N} : nx \geq y \quad (x > 0)$

A racionális számok rendezett teste archimédeszien rendezett, de nem felső határ tulajdonságú.

Egy felső határ tulajdonságú rendezett testet a valós számok testének nevezünk, és $\mathbb{R}$-rel jelöljük. $(\exists! \mathbb{R})$

1.3.2 Komplex számok

A komplex számok szükségét a harmadfokú egyenletek megoldására való Cardano-képlet szülte. Ugyanis abban az esetben, amikor az egyenletnek három különböző valós gyöke van, a képletben a gyökjel alá negatív szám kerül. Fokozatosan tisztult a "képzetes" számokkal való számolás szabályai, és a trigonometrikus függvényekkel való kapcsolat.

Definíció

A komplex számok halmaza $\mathbb{C} = \mathbb{R} \times \mathbb{R}$. $\mathbb{C}$ az $(x, y) + (x', y') = (x + x', y + y')$ összeadással és az $(x, y) \cdot (x', y') = (xx' - yy', y'x + yx')$ szorzással test. A komplex számok halmaza nem rendezett test, mivel (tétel alapján) egy rendezett integritási tartományban $x \neq 0 \ \Rightarrow x^2 > 0$. (Ez azonban $(0, 1)^2 = i^2 = -1$-re nem teljesül).

[A komplex számok körében (0,0) a nullelem, $(1, 0)$ egységelem, (x, y) additív inverze $(-x, -y)$, és $(0, 0) \neq (x, y)$ pár multiplikatív inverze az $(\frac{x}{x^2+y^2}, \frac{-y}{x^2+y^2})$ pár.]

Valós számok azonosítása

Mivel $(x, 0) + (x', 0) = (x + x', 0)$ és $(x, 0) \cdot (x', 0) = (xx', 0)$ így az összes $(x, 0), x \in \mathbb{R}$ komplex számot azonosíthatjuk $\mathbb{R}$-rel.

Komplex számok algebrai alakja

Mivel

$$(x, y) = (x, 0) + (y, 0) \cdot i = x + yi$$

így a komplex számokat $a + bi$ algebrai alakban is írhatjuk.

Ekkor az $\mathrm{Re}(z) = x$ valós számot a $z = (x, y)$ komplex szám valós részének, az $\mathrm{Im}(z) = y$ valós számot pedig a képzetes részének nevezzük.

Konjugált

$z = x + yi$ komplex szám konjugáltja: $\overline{z} = x - yi$

Tulajdonságai:

1. $\overline{z + w} = \overline{z} + \overline{w}$
2. $\overline{z \cdot w} = \overline{z} \cdot \overline{w}$
3. $\overline{\overline{z}} = z$
4. $z + \overline{z} = 2\mathrm{Re}(z)$
5. $z - \overline{z} = i \cdot 2\mathrm{Im}(z)$

Abszolút érték

A $z = (x, y)$ komplex szám abszolút értéke: $|z| = \sqrt{x^2 + y^2}$

Tulajdonságai:

1. $z \cdot \overline{z} = |z|^2$
2. $\frac{1}{z} = \frac{\overline{z}}{|z|^2}$
3. $|z| = \overline{|z|}$
4. $|z \cdot w| = |z| \cdot |w|$
5. $|z + w| \leq |z| + |w|$

Trigonometrikus alak

- Argumentum
  $z \neq 0$ esetén az a z argumentuma $\forall t \in \mathbb{R}$, melyre $\mathrm{Re}(z) = |z|\cos(t)$, és $\mathrm{Im}(z) = |z|\sin(t)$. Más szóval a z argumentuma az origóból a z-be mutató vektor és a pozitív valós tengellyel bezárt szöge.
- Trigonometrikus alak
  A z komplex szám trigonometrikus alakja: $z = |z|(\cos(t)+i\cdot\sin(t)$
- Moivre-azonosságok
  Legyen $z = |z|(\cos(t)+i\cdot\sin(t))$, és $w = |w|(\cos(s)+i\cdot\sin(s))$. Ekkor

$$z \cdot w = |z||w|(\cos(t + s) + i \cdot \sin(t + s))$$

$$\frac{z}{w} = \frac{|z|}{|w|}(\cos(t - s) + i \cdot \sin(t - s)) \quad (w \neq 0)$$

$$z^n = |z|^n(\cos(nt) + i \cdot \sin(nt)) \quad (n \in \mathbb{Z})$$

- Gyökvonás
  Legyen $z^n = w$ ekkor:

$$\sqrt[n]{w} = \left\{ z_k = \sqrt[n]{|w|}\left(\cos\left(\frac{t + 2k\pi}{n}\right) + \sin\left(\frac{t + 2k\pi}{n}\right)\right), k = 0, ..., n - 1 \right\}$$

  De mivel ez a jelöltés összetéveszthető a valósak között (egyértelművé tett) valós gyökvonással. így ezt a jelölést nem használjuk. Vezessük be helyette a n-edik komplex egységgyök fogalmát:

$$\varepsilon_k = \cos\left(\frac{2k\pi}{n}\right) + i \cdot \sin\left(\frac{2k\pi}{n}\right), \quad k = 0, ..., n - 1$$

  Ezek után a w gyökeit a z és az n-edik komplex egységgyökök segítségével kaphatjuk meg: $z\varepsilon_0, ..., z\varepsilon_{n-1}$

1.4 Leszámlálások véges halmazokon

Véges halmazok

- Halmazok ekvivalenciája
  X, Y halmazok ekvivalensek, ha létezik X-et Y-ra képező bijekció.
  Jele: $X \sim Y$
- Véges és végtelen halmazok
  X halmaz véges, ha $\exists n \in \mathbb{N} : X \sim \{1, 2, ..., n\}$, egyébként végtelen. Ha létezik n, akkor az egyértelmű, és ekkor a halmaz elemszámának/számosságának nevezzük. Jele: $\#(X)$

Skatulya elv

Ha X, Y véges halmazok és $\#(X) > \#(Y)$, akkor egy $f : X \to Y$ leképezés nem lehet kölcsönösen egyértelmű (azaz bijekció).

Leszámolások

- Permutáció
  A halmaz egy permutációja az önmagára való kölcsönösen egyértelmű leképezése. Az A halmaz összes permutációjának száma:
  $$P_n = \prod_{k=1}^{n} k \ = \ n!$$
- Variáció
  Az A halmaz elemeiből készíthető, különböző tagokból álló $a_1, a_2, ..., a_k$ sorozatokat az A halmaz k-ad osztályú variációinak nevezzük. Ha A véges ($\#(A) = n$), akkor V_n^k száma megegyezik az $\{1, 2, ..., k\}$-t $\{1, 2, ..., n\}$-be képező kölcsönösen egyértelmű leképezések számával:
  $$V_n^k = \frac{n!}{(n-k)!}$$
- Kombináció
  Ha A halmaz $k \in \mathbb{N}$ elemű részhalmazait k-ad osztályú kombinációinak nevezzük. Ha A véges, akkor C_n^k száma megegyezik $\{1, 2, ..., n\}$ k elemű részhalmazainak számával.
  $$C_n^k = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$
- Ismétléses permutáció
  $A = \{a_1, \ldots, a_r\}$ halmaz elemeinek ismétlődései $i_1, \ldots, i_r$. (Az elemek ismétléses permutációi olyan $i_1 + \cdots + i_r = n$ tagú sorozatok, melyben az a_j elem i_j-szer fordul elő.)
  $$P_n^{i_1,\ldots,i_r} = \frac{n!}{i_1! i_2! \cdots i_r!}$$
- Ismétléses variáció
  Az A véges halmaz elemeiből készíthető (nem feltétlenül különböző) $a_1, \cdots, a_k$ sorozatokat, az A halmaz k-ad osztályú ismétléses variációinak nevezzük.
  $$^{i}V_n^k = n^k$$
- Ismétléses kombináció
  Az A véges halmaz. A halmazból k elemet kiválasztva, ismétléseket megengedve, de a sorrend figyelmen kívül hagyva, az A halmaz k-ad osztályú ismétléses kombinációit kapjuk.
  $$^{i}C_n^k = \binom{n+k-1}{k}$$

Tételek

- Binomiális tétel
  $x, y \in R$ (kommutatív egységelemes gyűrű), $n \in \mathbb{R}$. Ekkor

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

- Polinomiális tétel
  $r, n \in \mathbb{N}$ és $x_1, x_2, \cdots, x_r \in R$ (kommutatív egységelemes gyűrű), ekkor

$$(x_1 + \cdots + x_r)^n = \sum_{i_1+\cdots+i_r=n} P_n^{i_1,\cdots,i_r} x_1^{i_1} x_2^{i_2} \cdots x_r^{i_r} \qquad (i_1, \cdots, i_r \in \mathbb{N})$$

- Szita formula
  $X_1, \cdots, X_k \subset X$ (véges halmaz). f az X-en értelmezett, egy Abel-csoportba képző függvény. Legyen:

$$S = \sum_{x \in X} f(x)$$

$$S_r = \sum_{1 \le i_1 \le \cdots \le i_r \le k} \left(\sum_{x \in X_{i_1} \cap \cdots \cap X_{i_r}} f(x) \right)$$

  és

$$S_0 = \sum_{x \in X \setminus \cup_{i=1}^{k} X i} f(x)$$

  Ekkor

$$S_0 = S - S_1 + S_2 - S_3 + \cdots + (-1)^k S_k$$

1.5 Számelméleti alapfogalmak, lineáris kongruencia-egyenletek

1.5.1 Számelméleti alapfogalmak

Oszthatóság egységelemes integritási tartományban

R egységelemes integritási tartomány, $a, b \in R$. Ha $\exists c \in R : a = bc$, akkor b osztója a-nak (a a b többszöröse). Jele: $b|a$

A $b = 0$-t kivéve legfeljebb egy ilyen c létezik.

Az oszthatóság tulajdonságai egységelemes integritási tartományban.

- Ha $b|a$ és $b'|a'$, akkor $bb'|aa'$
- $\forall a \in R : a|0$ (a nullának minden elem osztója)
- $0|a \Leftrightarrow a = 0$ (a null csak saját magának osztója)
- $\forall a \in R : 1|a$ (az egységelem minden elem osztója)
- $b|a \Rightarrow \forall c \in R : bc|ac$
- $bc|ac$ és $c \neq 0 \Rightarrow b|a$
- $b|a_i$ és $c_i \in R$, $(i = 1, \cdots, j) \;\Rightarrow\; b| \sum_{i=1}^{j} a_i c_i$
- az $|$ reláció reflexív és tranzitív

Felbonthatatlan elem és prímelem

$0, 1 \neq a \in R$ felbonthatatlan (irreducibilis), ha $a = bc$ esetén b vagy c egység ($b, c \in R$).

$0, 1 \neq p \in R$ prím, ha $\forall a, b \in R : p|ab$ esetén $p|a$ vagy $p|b$

Legnagyobb közös osztó, legkisebb közös többszörös, relatív prím

R egységelemes integritási tartomány. $a_1, \cdots, a_n \in R$ elemeknek $b \in R$ legnagyobb közös osztója, ha $b|a_i$ és $b'|a_i$ esetén $b'|b$. Ha b egység, akkor $a_1, \cdots, a_n$ relatív prímek.

$a_1, \cdots, a_n \in R$ elemeknek legkisebb közös többszöröse $b \in R$, ha $a_i|b$ és $a_i|b'$ esetén $b|b'$.

Bővített euklideszi algoritmus

Az eljárás meghatározza az $a, b \in \mathbb{Z}$ számok legnagyobb közös osztóját ($d \in \mathbb{Z}$), valamint $x, y \in \mathbb{Z}$ számokat úgy, hogy $d = ax + by$

A számelmélet alaptétele

Minden pozitív természetes szám (sorrendtől eltekintve) egyértelműen felbontható prímszámok szorzataként.

Erathoszthenész szitája

Adott n-ig a prímek meghatározásához: Írjuk fel a számokat 2-től n-ig. Az első szám (2) prím, összes többszöröse összetett, ezeket húzzuk ki. A fennmaradó számok közül az első (3) ugyancsak prím, stb. Az eljárás végén az n-nél nem nagyobb prímek maradnak.

1.5.2 Lineáris kongruencia egyenletek

Kongruencia

Ha $a, b, m \in \mathbb{Z}$ és $m|(a-b)$, akkor azt mondjuk, hogy a és b kongruensek modulo m (Jele: $a \equiv b \mod m$).

A kongruencia ekvivalenciareláció bármely m-re. Ha $a \in \mathbb{Z}$ akkor az ekvivalenciaosztály elemei $a + km, k \in \mathbb{Z}$ alakúak.

Maradékosztályok

Az $m \in \mathbb{Z}$ modulus szerinti ekvivalenciaosztályoknak nevezzük. A maradékosztályokat elemeikkel reprezentáljuk. (Az a elem által reprezentált maradékosztály $\widetilde{a} \mod m$).

Ha egy maradékosztály valamely eleme relatív prím a modulushoz, akkor mindegyik az és a maradékosztályt redukált maradékosztálynak nevezzük.

Páronként inkongruens egészek egy rendszerét maradékrendszernek nevezzük.

Ha egy maradékrendszer minden maradékosztályból tartalmaz elemet, akkor teljes maradékrendszer.

Ha maradékrendszer pontosan a redukált maradékosztályokból tartalmaz elemet, akkor redukált maradékrendszer.

Euler-féle φ függvény

$m > 0$ egész szám. Az Euler-féle $\varphi(m)$ függvény a modulo m redukált maradékosztályok számát adja meg. Ez nyilván megegyezik a $0, 1, \cdots, m-1$ számok közötti, m-hez relatív prímek számával.

Euler-Fermat tétel

$m > 1$ egész, a relatív prím m-hez, ekkor:

$$a^{\varphi(m)} \equiv 1 \mod m$$

Fermat tétel

Legyen p prím, és $a \in \mathbb{Z} : p \nmid a$, ekkor

$$a^{p-1} \equiv 1 \mod p$$

Lineáris kongruencia megoldása

Keressük az $ax \equiv b \mod m$ kongruencia megoldásait ($a, b, m \in \mathbb{Z}$ ismert). Ez ekvivalens azzal, hogy keressünk olyan x-et, melyre (valamely y-nal) $ax + my = b$.

Legyen $d = \text{lnko}(a, m)$. Mivel d osztója $ax + my$-nak, b-t is osztania kell, különben nincs megoldás. Így $\frac{a}{d}x + \frac{m}{d}y = \frac{b}{d}$. Ekkor $a'x + m'y = 1$. A bővített euklideszi algoritmus segítségével olyan u, v számokat kapunk, melyekkel $a'u + m'v = 1$ (ui.: a', m' relatív prímek). Az egyenletet b'-vel beszorozva $a'ub' + m'vb' = b' \Rightarrow x \equiv ub' \mod m'$

Lineáris kongruenciarendszer megoldása

Két lineáris kongruencia esetén a megoldások $x \equiv a \mod m$ és $x \equiv b \mod n$. A közös megoldáshoz $x = a+my = b+nz \Leftrightarrow my-nz = b-a$ egyenletet kell megoldani. Akkor és csak akkor van megoldás, ha $d = \text{lnko}(m, n)$ osztója $b-a$-nak. Ekkor a megoldás valamely x_1 egésszel $x \equiv x_1 \mod \text{lkkt}(m, n)$ alakban írható. (Több kongruencia esetén az eljárás folytatható.)

Kínai maradéktétel

$1 < m_1, \cdots, m_n \in \mathbb{N}$ páronként relatív prímek, és $c_1, \cdots, c_n \in \mathbb{Z}$. Az $x \equiv c_j \mod m_j$ ($j = 1, \cdots, n$) kongruenciarendszer megoldható, és bármely két megoldása kongruens $\mod m_1 m_2 \cdots m_n$

2 Gráfok

2.1 Általános és síkgráfok

Alapfogalmak

- Irányítatlan gráf
  Egy irányítatlan gráf a $G = (V, E, \varphi)$ rendezett 3-as, ahol:
  V - a csúcsok halmaza
  E - élek halmaza
  φ - illeszkedési reláció ($\varphi \in E \times V$)
  Ha $v \in \varphi(e)$, akkor v illeszkedik az e élre. ($v \in V, e \in E$). Egy élnek mindig két vége van
- Él-, és csúcstípusok
  - Izolált csúcs
    $v \in V$ izolált csúcs, ha $\nexists e \in E : v \in \varphi(e)$
  - Párhuzamos él
    $e, e' \in E$ élek párhuzamos élek, ha $\varphi(e) = \varphi(e')$
  - Hurokél
    $e \in E$ hurokél, ha $|\varphi(e)| = 1$
- Irányított gráf
  Egy irányítatott gráf a $G = (V, E, \psi)$ rendezett 3-as, ahol:
  V - a csúcsok halmaza
  E - élek halmaza
  ψ - illeszkedési reláció ($\psi \in E \to V \times V$)
  $\psi(e) = (v, v')$, *ahol v az e él kezdőpontja, v' a végpontja.*

Véges, egyszerű gráfok - alapfogalmak

- Egyszerű gráf
  G gráf egyszerű, ha nem tartalmaz párhuzamos vagy hurokéleket
- Véges gráf $G = (V, E, \varphi)$ gráf véges, ha V, E véges halmazok.
- Szomszédság, fok
  Két él szomszédos, ha van közös pontjuk.
  Két csúcs szomszédos, ha van közös élük.
  $v \in V$ szomszédjainak száma a v foka. [Jele: $\deg(v) = \mathrm{d}(v)$]
- r-reguláris gráfok
  G gráf r-reguláris, ha minden pont foka r
- Teljes gráf
  G gráf teljes gráf, ha minden él be van húzva, más szóval $(|V| - 1)$-reguláris. (Jele: $K_{|V|}$)
- Páros gráf
  G páros gráf, ha $V = V' \cup V''$ és $V' \cap V'' = \emptyset$ (diszjunkt), valamint él csak V' és V'' között fut.
  Ha viszont így V' és V'' között minden él be húzva, akkor teljes páros gráf. (Jele: $K_{n,m}$, ahol $n = |V'|, m = |V''|$)
- Részgráf
  $G = (V, E, \varphi)$ részgráfja $G' = (V', G', \varphi')$-nek, ha $V \subset V \ \wedge \ E \subset E' \ \wedge \ \varphi \subset \varphi'$
- Séta, vonal, út
  G gráfban egy n hosszú séta v-ből v'-be egy olyan

  $$v_0, e_1, v_1, \cdots, v_{n-1}, e_n, v_n$$

  sorozat, melyre $v = v_1, v' = v_n$ és $v_{i-1}, v_i \in \varphi(e_i)$
  Egy séta vonal, ha minden él legfeljebb egyszer szerepel a sorozatban.
  Egy vonal út, ha minden csúcs legfeljebb egyszer szerepel a sorozatban.
  Egy séta/vonal/út zárt, ha kezdő és végpontja megegyezik, egyébként nyílt.

- Összefüggő gráf Egy gráf összefüggő, ha bármely két csúcs közt van út.
  Ez a reláció ekvivalenciareláció, melynek ekvivalenciaosztályait komponenseknek nevezzük.
- Címkézett, Súlyozott gráf
  $G = V, E, \varphi, C_e, c_e, C_v, c_v)$ rendezett 7-es címkézett gráfot jelöl, ahol C_e, C_v tetszőleges halmazok, és

$$c_e : E \to C_e$$
$$c_v : E \to C_v$$

  Ha $C_e = C_v = \mathbb{R}^+$, akkor a gráfot súlyozott gráfnak nevezzük, és w a csúcs/él súlya. $(w(e) = c_e(e),\ w(v) = c_v(v))$

Síkba rajzolhatóság

Fogalmak

- Síkba rajzolhatóság
  Egy gráf síkba rajzolható, ha lerajzolható úgy, hogy az elei nem keresztezik egymást.
- Topologikus izomorfia
  Két gráf topologikusan izomorf, ha a következő lépést illetve fordítottját véges sok ismétlésével egyikből a másikat kapjuk: Egy másodfokú csúcsot elhagyunk, és a szomszédjait összekötjük.
- Tartomány
  Ha G gráf síkba rajzolható, akkor a tartományok az élek által határolt síkidomok. (A nem korlátolt síkidom is tartomány.)

Tételek

1. Minden véges gráf $\mathbb{R}^3$-ban lerajzolható.
2. Ha egy véges gráf síkba rajzolható $\iff$ gömbre rajzolható
3. Euler-tétel:
   Ha a G véges gráf összefüggő, síkba rajzolható gráf, akkor:

$$|E| + 2 = |V| + |T|$$

4. Kuratowsky-tétel:
   Egy véges gráf pontosan akkor síkba rajzolható, ha nem tartalmaz K_5-tel, vagy $K_{3,3}$-mal topologikusan izomorf részgráfot.

2.2 Fák

Fa

Egy gráfot fának nevezünk, ha összefüggő és körmentes.

Feszítőfa

F részgráfja G-nek. Ha F fa és csúcsainak halmaza megegyezik G csúcsainak halmazával, akkor F-et a G feszítőfájának nevezzük.

Tételek

- Ha G egyszerű gráf, akkor a következő feltételek ekvivalensek:
  1. G fa
  2. G összefüggő, de bármely él törlésével már nem az
  3. Két különböző csúcs között csak egy út van
  4. G körmentes, de egy él hozzáadásával már nem az
- Ha G egyszerű véges gráf, akkor a következő feltételek ekvivalensek:
  1. G fa
  2. G-ben nincs kör és $n - 1$ éle van
  3. G összefüggő és $n - 1$ éle van

Irányított fa

Olyan fa, melyre: $\exists v \in V : d^-(v) = 0$ és $\forall v' \neq v : d^-(v') = 1$ (Egy csúscs befoka 0, a többié 1)

További fogalmak:

- $r \in V, d^-(r) = 0$ csúcsot gyökérnek nevezzük
- v' csúcs szintje a r, v' út hossza
- $(v, v') \in \psi(e)$, a v szülője v'-nek, v' gyereke, v-nek.
- v levél, ha $d^+(v) = 0$

2.3 Euler- és Hamilton-gráfok

2.3.1 Euler-gráf

Euler-vonal

Az Euler-vonal olyan vonal v-ből v'-be a gráfban, amelyben minden él szerepel. Ha $v = v'$ akkor ezt a vonalat Euler-körvonalnak is szokás nevezni. Euler-vonallal rendelkező gráfot Euler-gráfnak nevezik.

Tétel

Egy összefüggő véges gráfban pontosan akkor létezik Euler-körvonal, ha minden csúcs páros fokú.

2.3.2 Hamilton-gráf

A Hamilton-út egy olyan út v-ből v'-be a gráfban, mely minden csúcsot tartalmat. Ha $v = v'$ akkor ezt az utat Hamilton-körnek is szokás nevezni. Hamilton-úttal rendelkező gráfot Hamilton-gráfnak nevezik.

2.4 Gráfok adatszerkezetei

Gráfok számítógépes reprezentációjához legtöbbször láncolt listákat, vagy mátrixokat szoktak használni. A láncolt listák inkább ritka gráfokra, míg a mátrixok sűrű gráfok esetén gazdaságosak.

Illeszkedési mátrix

$G = (V, E, \psi)$ irányított gráf esetén a gráfot egy $A = \{0, 1, -1\}^{n \times m}$ mátrix segítségével tudjuk reprezentálni, ahol $V = \{v_1, \cdots, v_n\}$, és $E = \{e_1, \cdots, e_m\}$. Ekkor a mátrix egyes elemei:

$$a_{ij} = \begin{cases} 1 & \text{ha } v_i \text{ kezdőpontja } e_j\text{-nek} \\ -1 & \text{ha } v_i \text{ végpontja } e_j\text{-nek} \\ 0 & \text{különben} \end{cases}$$

Ha G nem irányított, akkor $a_{ij} = |a_{i,j}|$

Csúcsmátrix

A fenti jelölésekkel irányított esetben $B \in \mathbb{Z}^{n \times n}$, ahol b_{ij} a v_i-ből v_j-be menő élek számát jelöli.

Ha G irányítatlan, akkor b_{ii} v_i hurokéleinek száma, egyébként b_{ij} a v_i és v_j csúcsok közötti élek száma.

3 Kódoláselmélet

3.1 Polinomok és műveleteik

Definíció

Legyen R gyűrű. Egy polinomot egy $\sum_{i=0}^{n} f_i x^i$ alakú véges összegnek tekintünk, ahol $n \in \mathbb{N}, f_i \in R$.

Az f_n tagot a polinom főegyütthatójának nevezzük.

Műveletek

Legyen $R[x]$ az $f = (f_0, f_1, \cdots)$ végtelen sorozatok feletti gyűrű (polinomok gyűrűje), ahol $f_i \in R$. Ekkor az $R[x]$-beli műveletek:

- Összeadás:

$$f + g = (f_0 + g_0, f_1 + g_1, \cdots) \qquad (f, g \in R[x])$$

- Szorzás:

$$f \cdot g = h = (h_0, h_1, \cdots) \qquad (f, g, h \in R[x]), \text{ ahol}$$
$$h_k = \sum_{i+j=k} f_i g_j$$

Megjegyzés: Ha R kommutatív, akkor $R[x]$ is az. Ha R egységelemes az 1 egységelemmel, akkor $R[x]$ is az az $(1, 0, 0, \cdots)$ egységelemmel.

3.2 Maradékos osztás

Legyen R egységelemes integritási tartomány, $f, g \in R[x], g \neq 0$ és tegyük fel, hogy g főegyüttható egység R-ben. Ekkor

$$\exists! q, r \in R[x] : f = g \cdot q + r \qquad (\deg(r) < \deg(g))$$

3.3 Horner-séma

A Horner-módszer egy polinom helyettesítési értékének kiszámítására alkalmas. (Ezzel együtt természetesen az is eldönthető, hogy adott c érték a polinom gyöke-e vagy nem. 4-ed fok felett erre még analitikus megoldás sincs.)

A módszer lényege, hogy az egyébként $f_n x^n + f_{n-1} x^{n-1} + \cdots + f_0$ polinom helyettesítési értékének kiszámolásához rendkívül sok szorzásra és összeadásra lenne szükség. A polinom átalakításával azonban a műveletek számát lecsökkenthetjük. A maradékos osztást alkalmazva:

$$f_n x^n + f_{n-1} x^{n-1} + \cdots + f_0 = (f_n x^{n-1} + f_{n-1} x^{n-2} + \cdots) x + f_0$$

Ezt rekurzívan folytatva a következő alakra jutunk:

$$(((f_n x + f_n - 1) x + f_n - 2) x + \cdots) x + f_0$$

A helyettesítési érték kiszámítását egy táblázatban könnyebben elvégezhetjük.

	f_n	f_{n-1}	f_{n-2}	$\cdots$	f_0
c	f_n	$f_n c + f_{n-1}$	$(f_n c + f_{n-1})c + f_{n-2}$	$\cdots$	$f(c)$

A táblázat kitöltése a következőképp zajlik:

1. Az első sorba felírjuk a polinom együtthatóit
2. A második sor első cellájába beírjuk az argumentum értékét.
3. A főegyüttható alá beírjuk önmagát.
4. A második sor celláinak kitöltésével folytatjuk
5. Az előző cella elemét megszorozzuk az argumentummal
6. A szorzathoz adjuk hozzá az aktuális együtthatót
7. Az összeget írjuk be az aktuális cellába
8. Folytassuk az 5. ponttal, míg el nem jutunk az utolsó celláig

Az utolsó cellába a polinom helyettesítési értéke kerül. (Ha ez nulla, akkor az argumentum a polinom gyöke.)

3.4 Betűnkénti kódolás

A kódolás a legáltalánosabb értelemben az üzentek halmazának egy másik halmazba való leképzését jelenti. Gyakran az üzenetet valamilyen karakterkészlet elemeiből alkotott sorozattal adjuk meg. Ekkor az üzenetet felbontjuk előre rögzített olyan elemi részekre, hogy minden üzenet egyértelműen előálljon ilyen elemi részek sorozataként. A kódoláshoz megadjuk az elemi részek kódját, amelyet egy szótár tartalmaz. Az ilyen kódolást betűnkénti kódolásnak nevezzük.

A kódolandó üzenetek egy A ábécé betűi, és egy-egy betű kódja egy másik, B ábécé (kódábécé) betűinek felel meg. Tegyük fel, hogy mind két ábécé nem üres és véges.

Egy A ábécé betűiből felírható szavak halmazát A^+-szal jelöljük, míg az üres szóval kiterjesztettet A^*-gal.

Ez alapján a betűnkénti kódolást egy $\varphi : A \to B^*$ leképezés határozza meg, amelyet kiterjeszthetünk egy $\psi : A^* \to B^*$ leképezéssé, alábbi módon: Ha $\alpha_1\alpha_2...\alpha_n = \alpha \in A$, akkor α kódja $\psi(\alpha) = \varphi(\alpha_1)\varphi(\alpha_2)...\varphi(\alpha_3)$. Nyilván ha φ nem injektív (vagy az üres szó benne van az értékkészletében), akkor a ψ kódolás sem injektív, azaz nem egyértelműen dekódolható. Emiatt feltehetjük, hogy φ injektív, és B^+-ba képez.

3.5 Shannon- és Huffman-kód

Alapfogalmak

- Gyakoriság, relatív gyakoriság, eloszlás
  Az információforrás n üzenetet bocsájt ki. A különböző üzeneteket jelöljük $a_1, \cdots, a_m$-mel. a_i üzenet k_i-szer fordul elő, melyet gyakoriságnak nevezzük. Az a_i relatív gyakorisága a $p_i = k_i/n$. A $p_1, \cdots, p_m$ szám m-est az üzentek eloszlásának nevezzük. ($\sum_{i=1}^{m} p_i = 1$)
- Információtartalom
  Az a_i üzenet egyedi információtartalma $I_i = -\log_r p_i$, ahol $r > 1$ az információ egysége. ($r = 2$ esetén az egység a bit).
- Entrópia
  Az üzenetforrás által kibocsátott átlagos információtartalmat nevezzük entrópiának:

$$H_r(p_1, \cdots, p_m) = -\sum_{i=1}^{m} p_i \log_r p_i$$

- Prefix, szuffix, infix
  Legyen $\alpha, \beta, \gamma \in A$ szavak. Ekkor az $\alpha\beta\gamma$ szónak α prefixe, β infixe, γ pedig szuffixe.
- Kódfa
  A betűnkénti kódoláshoz egyértelműen adható meg egy szemléletes irányított, élcímkézett fa. Legyen $\varphi : A \to B^*$ a betűnkénti kódolás. Készítsünk el egy olyan fát, melynek a gyökere az üres szó és ha $\beta = \alpha b \quad (b \in B)$-re, akkor α-ból húzódjon olyan él β-ba, melynek b címkéje van. Ekkor minden azonos hosszú szó egy szinten lesz. Azokat a csúcsokat, melyekből minden $b \in B$ címkével vezet ki él teljes csúcsnak nevezzük, különben csonka csúcsok.
- Prefix kód, egyenletes kód, vesszős kód
  A $\varphi : A \to B^+$ injektív leképezés által meghatározott $\psi : A^* \to B^*$ betűnkénti kódolás
  1. felbontható (egyértelműen dekódolható), ha ψ injektív
  2. prefix kód, ha φ értékkészlete prefixmentes.
  3. egyenletes kód (fix hosszúságú), ha ψ értékkészletében minden elem megegyező hosszú
  4. vesszős kód, ha $\exists \vartheta \in B^+$ vessző, hogy ϑ szuffixe minden kódszónak, de sem prefixe, sem infixe semelyik kódszónak.
- Átlagos szóhosszúság
  Legyen $A = \{a_1, \cdots, a_n\}$ a kódolandó ábécé. Az a_i kódjának hossza l_i. Ekkor $\bar{l} = \sum_{i=1}^{n} p_i l_i$ a kód átlagos szóhosszúsága.
- Optimális kód
  Ha egy adott elemszámú ábécével és adott eloszlással egy felbontható betűnkénti kód átlagos szóhoszúsága minimális, akkor optimális kódnak nevezzük.

Shannon-kód

Shannon kód egy optimális kód (r elemszámú ábécével és p_i gyakoriságokkal), melyet a következő módon állítunk elő.

1. Rendezzük a betűket relatív gyakoriságaik alapján csökkenő sorrendbe.
2. Határozzuk meg az $l_1, \cdots, l_n$ szóhosszúságokat a következő módon:

$$r^{-l_i} \leq p_i < r^{-l_i+1}$$

3. Osszuk el az ábécé elemeit az egyes helyiértékeken.

Példa:

Legyen a kódábécé a $0, 1, 2$ halmaz, az kódolandó betűk és gyakoriságaik pedig a következők:

a	b	c	d	e	f	g	h	i	j
0,17	0,02	0,13	0,02	0,01	0,31	0,02	0,17	0,06	0,09

A relatív gyakoriságok rendezése után:

f	a	h	c	j	i	b	d	g	e
0,31	0,17	0,17	0,13	0,09	0,06	0,02	0,02	0,02	0,01

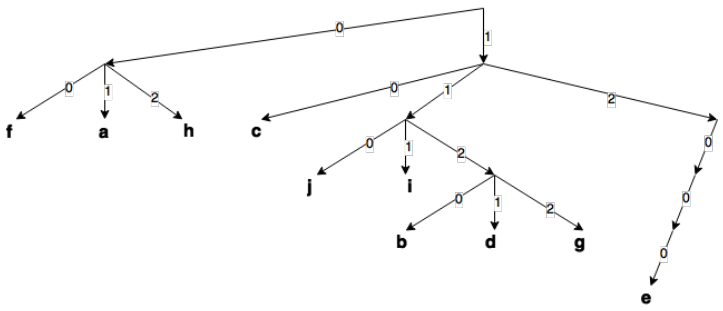
Határozzuk meg szóhosszúságokat. Az f, a, h és c esetében: $3^{-2} = r^{-l_i} \leq p_i < r^{-l_i+1} = 3^{-1}$ Tehát azok szóhosszúsága 2. A többi esetben is így járunk el:

f	a	h	c	j	i	b	d	g	e
0,31	0,17	0,17	0,13	0,09	0,06	0,02	0,02	0,02	0,01
2	2	2	2	3	3	4	4	4	5

Ezek alapján f kódszava a 00, a kódszava a 01, h-hoz a 02 tartozik, míg c-hez 10. A j-hez ezek után 11 tartozna, de mivel az 3 hosszú, így 110. A kódszavak tehát a következőképp alakulnak:

f	a	h	c	j	i	b	d	g	e
0,31	0,17	0,17	0,13	0,09	0,06	0,02	0,02	0,02	0,01
2	2	2	2	3	3	4	4	4	5
00	01	02	10	110	111	1120	1121	1122	12000

A kódfát 1. ábrán láthatjuk.



ábra 1: Shannon-kód példa kódfája

Huffman-kód

A Huffman-kód is optimális kód (r elemszámú ábécével és p_i gyakoriságokkal), melyet a következő módon állítunk elő.

1. Rendezzük a betűket relatív gyakoriságaik alapján csökkenő sorrendbe.
2. Annak érdekében, hogy csak egy csonka csúcs keletkezzen

$$m \equiv n \mod r - 1$$

kongruenciának teljesülnie kell, ahol m az egyetlen csonka csúcs kifoka. Ami ekvivalens azzal, hogy $m = 2 + ((n-2) \mod r - 1)$. Tehát osszuk el $n-2$-t $r-1$-gyel, és így m a maradék+2 lesz.

3. Az első lépésben a sorozat m utolsó betűjét összevonjuk (új jelölést/betűt adunk neki), és ennek a relatív gyakorisága a tagok relatív gyakoriságának összege lesz. Rendezzük a sorozatot. Ezen lépés után már a betűk száma kongruens $r-1$-gyel, így a következő redukciós lépésekben mindig teljes csúcsokat tudunk készíteni.
4. Az utolsó r betűt vonjunk össze, helyettesítsük egy új betűvel és relatív gyakoriság legyen a relatív gyakoriságok összege.
5. A 4-beli redukciós lépést addig ismételjük míg r db betű nem marad. Ekkor rendre minden betűhöz a kódábécé egy-egy betűjét rendeljük.
6. Ha redukált elemmel találkozunk szétbontjuk, majd az ő elemeihez is a kódábécé betűit rendeljük, de konkatenáljuk az előzővel.
7. A 6-beli lépést addig ismételjük míg marad redukált elem.

Példa:

A Shannon-kódnál látott forrást kódoljuk be ugyanúgy $\{0,1,2\}$ kódábécével.

a	b	c	d	e	f	g	h	i	j
0,17	0,02	0,13	0,02	0,01	0,31	0,02	0,17	0,06	0,09

Rendezzük relatív gyakoriság szerint:

f	a	h	c	j	i	b	d	g	e
0,31	0,17	0,17	0,13	0,09	0,06	0,02	0,02	0,02	0,01

Osszuk el $n-2$-t $r-1$-gyel: $10-2=4*(3-1)+0$. Így m a maradék+2, azaz $m=2$. Az utolsó m betűt összevonjuk, és rendezzük a sorozatot:

f	a	h	c	j	i	(g,e)	b	d
0,31	0,17	0,17	0,13	0,09	0,06	0,03	0,02	0,02

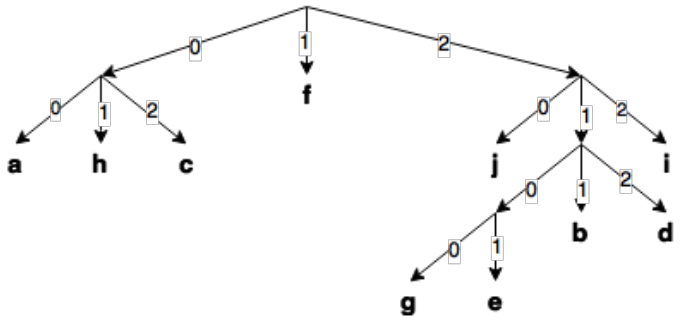
Innentől kezdve minden redukciós lépésben az utolsó r db azaz 3 betűt vonjuk össze:

f	a	h	c	j	((g,e), b, d)	i
0,31	0,17	0,17	0,13	0,09	0,07	0,06

Ezt addig ismételjük, míg r darab betű marad:

(a,h,c)	f	(j,((g,e),b,d),i)
0,47	0,31	0,22

A szétbontás alapján a 2. ábrán látható fát tudjuk összeállítani.



ábra 2: Huffman-kód példa kódfája

Ezek alapján a kódtábla:

betű	gyakoriság	kód
f	0,31	1
a	0,17	00
h	0,17	01
c	0,13	02
j	0,09	20
i	0,06	22
b	0,02	211
d	0,02	212
g	0,02	2100
e	0,01	2101

3.6 Hibajavító kódok, kódtávolság

Hibakorlátozó kódolás

A hibakorlátozó kódokat két csoportba sorolhatjuk: hibajelző és hibajavító kódok. Mindkét esetben az üzenetekhez kódszavakat rendelünk, amik alapján az átvitel során keletkező hibákat kezelni tudjuk. Ha az üzenet könnyen ismételhető hibajelző, ha nehezen ismételhető hibajavító kódot alkalmazunk. A hibakorlátozó kódoknál mindig azonos hosszúságú kódszavakat használunk.

Kódok távolsága, súlya

A kódábécé u és v szavának Hamming-távolsága $d(u, v)$ az azonos pozícióban levő, eltérő jegyek száma. A Hamming-távolság rendelkezik a távolság szokásos tulajdonságaival, vagyis $\forall u, v, z$:

- $d(u, v) \geq 0$
- $d(u, v) = 0 \iff u = v$
- $d(u, v) = d(v, u)$ - szimmetria
- $d(u, z) \leq d(u, v) + d(v, z)$ - háromszög egyenlőtlenség

A kód távolsága $d(C) = \min\limits_{u \neq v} d(u, v) \qquad (u, v \in C)$

Amennyiben az A kódábécé Abel-csoport a 0 nullelemmel. Ekkor egy u szó Hamming-súlya ($w(u)$) a szóban szereplő nem nulla elemek száma. Ekkor a kód súlya $w(C) = \min\limits_{u \neq 0} w(u)$

Hibajavító kód

Amikor egy olyan szót kapunk, ami nem kódszó, a hozzá legkisebb Hamming-távolságú kódszóra javítjuk.

A K kód t-hibajavító, ha egy legfeljebb t helyen megváltozott kódot helyesen javít. A K kód pontosan t-hibajavító, ha t-hibajavító, de nem $t + 1$-hibajavító.

Megjegyzés: d minimális távolságú kód esetén d/2-nél kevesebb hibát biztosan egyértelműen tudunk javítani.

Hamming-korlát

Egy q elemű ábécé n hosszú szavaiból álló C kód t-hibajavító. Ekkor bármely két kódszóra a tőlünk legfeljebb t távolságra lévő szavak halmazai diszjunktak.

Mivel egy kódszótól j távolságra pontosan $\binom{n}{j}(q-1)^j$ szó van, így a Hamming-korlát a kódszavak számára adott t-nél:

$$\#(C) \cdot \sum_{j=0}^{t} \binom{n}{j} (q-1)^j \leq q^n$$

Amennyiben egyenlőség áll fent tökéletes kódról beszélünk.

3.7 Lineáris kódok

Definíció

A véges test és A^n lineáris tér. Minden $K \leq A^n$ alteret lineáris kódnak nevezzük. Ha az altér k dimenziós, a kód távolsága d és $\#(A) = q$, akkor az ilyen kódot $[n, k, d]_q$ kódnak nevezzük.

Egy lineáris kódnál feltesszük, hogy kódolandó üzenetek K^k elemei, azaz a kódábécé elemeiből képzett k-asok.

Generátormátrix

K véges test feletti $[n, k, d]_q$ lineáris kódolást válasszuk egy (kölcsönösen egyértelmű) lineáris leképezésnek:

$$G : K^k \to K^n$$

Ezt egy mátrixszal, az úgy nevezett generátormátrixszal jellemezhetjük.

Polinomkódok

Egy lineáris kód esetén az üzeneteket megfeleltethetjük $\mathbb{F}_q$ (q elemű véges test) feletti k-nál alacsonyabb fokú polinomoknak.

$$(a_0, a_1, \cdots, a_{k-1}) \to a_0 + a_1 x + \cdots + a_{k-1} x^{k-1}$$

Legyen $g(x)$ rögzített m-edfokú polinom. A $p(x)$ polinomot (üzenet) $g(x)$-szel szorozva lineáris kódolást kapunk (mivel a $p \to pg$ kölcsönösen egyértelmű). Ekkor a kódszavak hossza $n = k + m$. Az ilyen típusú lineáris kódolást polinomkódolásnak nevezzük.

Megjegyzés: Feltehetjük, hogy $g(x)$ főpolinom (együtthatója egység), illetve a konstans tag nem nulla (ha nulla lenne, a szorzatban kiesne a konstans tag, így a kódban a nulla indexű betű soha nem hordozna információt)

CRC - Cyclic Redundancy Check

Ha egy polinomkódban $g(x)|x^n - 1$, akkor ciklikus kódról beszélünk. Ekkor, ha $a_0 a_1 \cdots a_{n-1}$ kódszó, akkor $a_{n-1} a_0 \cdots a_{n-2}$ is az, mivel:

$$a_{n-1} + a_0 x + \cdots + a_{n-2} x^{n-1} \;=\; x \cdot (a_0 + a_1 x + \cdots a_{n-1} x^{n-1}) - a_{n-1}(x^n - 1)$$

osztható $g(x)$-szel.

A CRC az $\mathbb{F}_2$ feletti ciklikus kódokat foglalja magába. Csak hibajelzésre alkalmas, a kódolás a következő: Vegyük $p(x)x^m = (0, 0, \cdots, 0, a_m, a_{m+1}, \cdots, a_{n-1})$. Ezt osszuk el $g(x)$-el maradékosan. $p(x)x^m = q(x)g(x) + r(x)$. Ekkor a kódszó legyen: $p(x)x^m - r(x) = q(x)g(x)$, amely osztható $g(x)$-szel és magas fokszámokon az eredeti üzenet betűi helyezkednek el. A vett szó ellenőrzése egyszerű: Megnézzük, hogy osztható-e $g(x)$-szel, ha nem, hiba történt.

Chapter 5

5

Záróvizsga tételsor

5. Valószínűségszámítási és statisztikai alapok

Fekete Dóra

Valószínűségszámítási és statisztikai alapok

Diszkrét és folytonos valószínűségi változók, nagy számok törvénye, centrális határeloszlás tétel. Statisztikai becslések, klasszikus statisztikai próbák.

1 Kolmogorov-féle valószínűségi mező

$(\Omega, \mathcal{A}, P)$ hármas, ahol:

Ω nem üres halmaz, eseménytér, ω elemi esemény.

$\mathcal{A}$ Ω részhalmazainak egy rendszere, $\mathcal{A} \subset 2^{\Omega}, A \in \mathcal{A}$ események, $\mathcal{A}$ σ-algebra.
σ-algebra:

1. $\Omega \in \mathcal{A}$
2. $A \in \mathcal{A} \Rightarrow \overline{A} = \Omega \backslash A \in \mathcal{A}$
3. $A_1, A_2 ... \in \mathcal{A} \Rightarrow \bigcup_{i=1}^{\infty} A_i \in \mathcal{A}$

$P : \mathcal{A} \to [0,1]$ halmazfüggvény, valószínűség, amelyre $P(\Omega) = 1, P(A) \geq 0, \forall A \in \mathcal{A}$-ra, páronként kizáró $(A_i \cdot A_j = \emptyset, i \neq j)$ $A_1, A_2 ... \in \mathcal{A}$ eseményekre $P(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} P(A_i)$.

2 Diszkrét és folytonos valószínűségi változók

- *Valószínűségi változó*: $\xi : \Omega \to \mathbb{R}$ mérhető függvény, azaz amire $\{\omega : \xi(\omega) < x\} \in \mathcal{A}, \forall x \in \mathbb{R}$, ahol $\mathcal{A}$ az eseménytér (Ω) részhalmazainak egy rendszere. ($\omega \in \Omega$ elemi esemény).
- *Valószínűségi változó eloszlása/eloszlásfüggvénye*: $F_{\xi}(x) = P(\xi < x)$, $\forall x \in \mathbb{R}$.
  Tulajdonságai:
  1. $0 \leq F_{\xi}(x) \leq 1$
  2. monoton növő
  3. balról folytonos
  4. $\lim\limits_{x \to -\infty} F_{\xi}(x) = 0$, $\lim\limits_{x \to \infty} F_{\xi}(x) = 1$

2.1 Diszkrét valószínűségi változók

Értékkészlete legfeljebb megszámlálhatóan végtelen, azaz $\{x_1 ... x_n ...\}$ elemekből áll. Ekkor eloszlása: $p_k := P(\xi = x_k)$.

Név	Értelmezés	Eloszlás	EX	D^2X
indikátor $Ind(p)$	Egy p valószínűségű esemény bekövetkezik-e vagy sem.	$P(X=1)=p$ $P(X=0)=1-p$	p	$p(1-p)$
geometriai (Pascal) $Geo(p)$	Hányadikra következik be először egy p valószínűségű esemény.	$P(X=k)=p(1-p)^{k-1}$ $k=1,2...$	$\frac{1}{p}$	$\frac{1-p}{p^2}$
hipergeometriai $Hipgeo(N,M,n)$	Visszatevés nélküli mintavétel.	$P(X=k)=\frac{\binom{M}{k}\binom{N-M}{n-k}}{\binom{N}{n}}$ $k=0,1,...,n$	$n\frac{M}{N}$	$n\frac{M}{N}(1-\frac{M}{N})(1-\frac{n-1}{N-1})$
binomiális $Bin(n,p)$	Visszatevéses mintavétel.	$P(X=k)=\binom{n}{k}p^k(1-p)^{n-k}$ $k=0,1,...,n$	np	$np(1-p)$
negatív binomiális $Negbin(n,p)$	Hányadikra következik be $n.$ alkalommal egy p valószínűségű esemény.	$P(X=k)=\binom{k-1}{n-1}p^n(1-p)^{k-n}$ $k=n,n+1,...$	$\frac{n}{p}$	$\frac{n(1-p)}{p^2}$
Poisson $Poi(\lambda)$	Ritka esemény.	$P(X=k)=\frac{\lambda^k}{k!}e^{-\lambda}$	λ	λ

2.2 Folytonos valószínűségi változók

Egy ξ valószínűségi változó abszolút folytonos, ha létezik olyan $f(x)$ függvény, amelyre $F(x)=\int_{-\infty}^{x} f(t)dt$. Ilyenkor $f(x)$ *sűrűségfüggvény.* ($F(x)$ pedig az eloszlásfüggvény.)
Másik megfogalmazás: $\forall a<b$-re $P(a<\xi<b)=\int_a^b f(t)dt$, $F_\xi(x)=P(\xi<x)=\lim_{a\to-\infty} P(a<\xi\ b)=\int_{-\infty}^{x} f(t)dt$.
Sűrűségfüggvény tulajdonságai:

1. $f(x)=F'(x)$
2. $f(x)\geq 0$
3. $\int_{-\infty}^{\infty} f(x)dx=1$

Név	Eloszlásfüggvény	Sűrűségfüggvény	EX	D^2X
egyenletes $E(a,b)$	$\begin{cases} 0 & x\leq a \\ \frac{x-a}{b-a} & a<x\leq b \\ 1 & b<x \end{cases}$	$\begin{cases} \frac{1}{b-a} & a<x\leq b \\ 0 & otherwise \end{cases}$	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$
exponenciális $Exp(\lambda)$	$\begin{cases} 1-e^{-\lambda x} & x\geq 0 \\ 0 & otherwise \end{cases}$	$\begin{cases} \lambda\cdot e^{-\lambda x} & x\geq 0 \\ 0 & otherwise \end{cases}$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$
normális $N(m,\sigma^2)$	...	$\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-m)^2}{2\sigma^2}}\ x\in\mathbb{R}$	m	σ^2
standard normális $N(0,1^2)$	$\Phi(x)=...$	$\frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}\ x\in\mathbb{R}$	0	1
gamma $\Gamma(\alpha,\lambda)$	...	$\begin{cases} \frac{1}{\Gamma(\alpha)}\lambda^\alpha x^{\alpha-1}e^{-\lambda x} & x\geq 0 \\ 0 & otherwise \end{cases}$	$\frac{\alpha}{\lambda}$	$\frac{\alpha}{\lambda^2}$

2.3 Fogalmak

- *Konvolúció*: X,Y független valószínűségi változók, konvolúciójuk az $X+Y$ v. v.
- *Függetlenség*: $P(\xi_1<x_1,...,\xi_n<x_n)=\prod_{i=1}^n P(\xi_i<x_i)$ vagy diszkrét esetben: $P(\xi_1)x_1,...,\xi_n=x_n)=\prod_{i=1}^n P(\xi_i=x_i)$
- *Várható érték*: $(\Omega,\mathcal{A},P)$ valószínűségi mező, $X:\Omega\to\mathbb{R}$ valószínűségi változó, $EX=\int_\Omega X dP$, ha ez létezik. Diszkrét esetben $EX=\sum_k x_k\cdot p_k$, ha abszolút konvergens. Abszolút folytonos esetben $EX=\int_{-\infty}^{\infty} x\cdot f(x)dx$, ha abszolút folytonos.
- *Szórásnégyzet*: $D^2X=E((X-EX)^2)=EX^2-E^2X$

- *l. momentum*: $EX^l = \int_\Omega x^l dP$, ha létezik.
- *Szórás*: $DX = \sqrt{D^2X}$
- *Kovariancia*: $cov(X,Y) = E((X-EX)(Y-EY) = E(XY) - EX \cdot EY$. Ha $cov(X,Y) = 0$, akkor X és Y korrelálatlan. (Megjegyzés: ha két v.v. független, akkor $cov(X,Y) = 0$, vagyis korrelálatlanok; illetve $cov(X,X) = D^2X$.)
- *Korreláció*: $R(X,Y) = \frac{cov(X,Y)}{DX \cdot DY}$, két v.v. lineáris kapcsolatát méri. $R > 0 \to$ pozitív, $R < 0 \to$ negatív; $R^2 \sim 1 \to$ erős, $R^2 \sim 0.5 \to$ közepes, $R^2 \sim 0 \to$ gyenge.

3 Nagy számok törvénye

3.1 Gyenge törvény

$X_1, X_2, ...$ függetlenek, azonos eloszlásúak, $EX_i = m < \infty$, $D^2X_i = \sigma^2 < \infty$.
$P(\frac{X_1+...+X_n}{n} - m \geq \varepsilon) \to 0$ $(n \to \infty)$ $\forall \varepsilon > 0$-ra (sztochasztikus konvergencia).

3.2 Erős törvény

$X_1, X_2, ...$ függetlenek, azonos eloszlásúak, $EX_1 = m < \infty$, $D^2X_1 = \sigma^2 < \infty$.
$\frac{X_1+...+X_n}{n} \to m$ $(n \to \infty)$ 1 valószínűséggel.
Megjegyzés: Csebisev-egyenlőtlenséggel bizonyítjuk. ($\frac{\sigma^2}{n\varepsilon^2} \to 0$ $(n \to \infty)$)

3.2.1 Csebisev-egyenlőtlenség

EX véges.
Ekkor $P(|X - EX| \geq \lambda) \leq \frac{D^2X}{\lambda^2}$
Megjegyzés: Bizonyítás Markov-egyenlőtlenséggel.

3.2.2 Markov-egyenlőtlenség

$X \geq 0, c > 0$.
Ekkor $P(X \geq c) \leq \frac{EX}{c}$

3.3 Konvergenciafajták

$\xi_n \to \xi$, vagyis ξ konvergens.

- *sztochasztikusan*: ha $\forall \varepsilon > 0$-ra $P(|\xi_n - \xi| \geq \varepsilon) \to 0$ $(n \to \infty)$.
- *1 valószínűséggel (majdnem mindenütt)*: ha $P(\omega : \xi_n(\omega) \to \xi(\omega)) = 1$.
- *L^p-ben*: ha $E(|\xi_n - \xi|^p) \to 0$ $(n \to \infty)$ $(p > 0$ rögzített).
- *eloszlásban*: ha $F_{\xi_n}(x) \to F_\xi(x)$ $(n \to \infty)$ az utóbbi minden folytonossági pontjában.

Kapcsolataik: 1 valószínűségű és L^p-beli a legerősebb, ezekből következik a sztochasztikus, ebből pedig az eloszlásbeli.

4 Centrális határeloszlás tétel

$X_1, X_2, ...$ függetlenek, azonos eloszlásúak, $EX_1 = m < \infty$, $D^2X_1 = \sigma^2 < \infty$.
Ekkor $\frac{X_1+...+X_n-nm}{\sqrt{n}\sigma} \to N(0,1)$ $(n \to \infty)$ eloszlásban, azaz $P(\frac{X_1+...+X_n-nm}{\sqrt{n}\sigma} < x) \to \Phi(x)$ $(n \to \infty)$.

5 Statisztikai mező

$(\Omega, \mathcal{A}, \mathcal{P})$ hármas, ha $\mathcal{P} = \{P_\vartheta\}_{\vartheta \in \Theta}$ és $(\Omega, \mathcal{A}, P_\vartheta)$ Kolmogorov-féle valószínűségi mező $\forall \vartheta \in \Theta$-ra.

5.1 Fogalmak

- *Minta*: $\underline{\xi} = (\xi_1, ..., \xi_n) : \Omega \to \Xi \in \mathbb{R}^n$. ($\xi_i$ valószínűségi változó)
- *Mintatér*: Ξ, minta lehetséges értékeinek halmaza, gyakran $\mathbb{R}^n, \mathbb{Z}^n$.
- *Minta [realizációja]*: $\underline{x} = (x_1, ..., x_n)$, konkrét megfigyelés.
- *Statisztika*: $T : \Xi \to \mathbb{R}^k$.
- *Statisztika alaptétele*: (Glivenko–Cantelli-tétel) $\xi_1, \xi_2, ...$ független, azonos eloszlású F eloszlásfüggvénnyel. Ekkor az F_n tapasztalati eloszlásfüggvényre teljesül, hogy $\sup_{-\infty<x<\infty} |F_n(x) - F(x)| \to 0$ ($n \to \infty$) 1 valószínűséggel.

6 Statisztikai becslések

$(\Omega, \mathcal{A}, \mathcal{P})$ statisztikai mező, $\vartheta \in \Theta$, $P_\vartheta(\xi_1 < x_1, ..., \xi_n < x_n) = F_\vartheta(\underline{x})$
$T(\underline{\xi})$ a ϑ *becslése*, ha $T : \mathbb{R}^n \to \Theta$.
$T(\underline{\xi})$ a $h(\vartheta)$ *becslése*, ha $T : \mathbb{R}^n \to h(\Theta)$.
Torzítatlanság: $T(\underline{\xi})$ torzítatlan becslése $h(\vartheta)$-nak, ha $E_\vartheta T(\underline{\xi}) = h(\vartheta)$ $\forall \vartheta \in \Theta$.
Aszimptotikusan torzítatlan: $T(\underline{\xi})$ aszimptotikusan torzítatlan a $h(\vartheta)$-ra, ha $E_\vartheta T(\underline{\xi}) \to h(\vartheta)$ ($n \to \infty$) $\forall \vartheta \in \Theta$.
A T_1 torzítatlan becslés *hatásosabb* T_2 torzítatlan becslésnél, ha $D^2_\vartheta T_1 \leq D^2_\vartheta T_2$ $\forall \vartheta \in \Theta$.
Hatásos, ha minden más torzítatlan becslésnél hatásosabb. Ha van hatásos becslés, akkor az egyértelmű.

- *Maximum-likelihood becslés*: Likelihood függvény: $L(\vartheta, \underline{x}) = \begin{cases} P_\vartheta(\underline{\xi} = \underline{x}) & diszkr. \\ f_{\vartheta,\underline{\xi}}(\underline{x}) & absz.folyt. \end{cases}$
  Független esetben: $L(\vartheta, \underline{x}) = \begin{cases} \prod_{i=1}^n P_\vartheta(\xi_i = x_i) & diszkr. \\ \prod_{i=1}^n f_{\vartheta,\xi_i}(x_i) & absz.folyt. \end{cases}$
  $\hat{\vartheta}$ a ϑ ismeretlen paraméter maximum-likelihood becslése, ha $L(\hat{\vartheta}, \underline{\xi}) = \max_{\vartheta\in\Theta} L(\vartheta, \underline{\xi})$.
- *Momentum-módszer becslés*: $\vartheta = (\vartheta_1, ..., \vartheta_k)$, $\xi_1, ..., \xi_n$, l. momentum: $M_l(\underline{\vartheta}) = E_{\underline{\vartheta}}\xi_i^l$, tapasztalati l. momentum: $\hat{M}_l = \frac{\sum_{i=1}^n \xi_i^l}{n}$.
  $\hat{\underline{\vartheta}}$ a $\underline{\vartheta}$ momentum módszer szerinti becslése, ha megoldása az $M_l(\underline{\vartheta}) = \hat{M}_l$, $l = 1..k$ egyenletrendszernek.

7 Hipotézisvizsgálat

Felteszünk egy hipotézist, és vizsgáljuk, hogy igaz-e. Elfogadjuk vagy elutasítjuk. Lehet paraméteres vagy nem paraméteres, vizsgálhatjuk várható értékek, szórások egyezőségét, értékét, teljes eseményrendszerek függetlenségét. Illeszkedésvizsgálattal megállapíthatjuk, hogy a valószínűségi változók adott eloszlásfüggvényűek-e, homogenitásvizsgálattal pedig azt, hogy ugyanolyan eloszlású-e két minta.
H_0: nullhipotézis, $\vartheta \in \Theta_0$; H_1: ellenhipotézis, $\vartheta \in \Theta_1$; $\Theta = \Theta_0 \bigcup \Theta_1$.
Egy- és kétoldali vizsgálat: Kétoldali ellenhipotézisnél a nem egyezőséget tesszük fel, egyoldalinál valamilyen relációt. Kétoldalinál a próba értékének abszolút értékét vizsgáljuk, hogy az elfogadási tartományon belül van-e, ekkor például az u-próbánál az adott hibaszázalékot meg kell felezni a számításhoz, hiszen a Φ függvény szimmetrikus az y tengelyre.

- *Statisztikai próba*: $\Xi = \Xi_e \bigcup \Xi_k$ (diszjunkt halmazok) elfogadási és kritikus tartomány. Ez a felbontás a statisztikai próba. Ha a megfigyelés eleme a kritikus tartománynak, akkor elutasítjuk a nullhipotézist, ha nem eleme, akkor elfogadjuk. $T(\underline{x}) = \begin{cases} 1 & x \in \Xi_k \\ 0 & otherwise \end{cases}$
- *Elsőfajú hiba*: H_0 igaz, de elutasítjuk. Valószínűsége: $P_\vartheta(\underline{\xi} \in \Xi_k)$, $\vartheta \in \Theta_0$.
- *Másodfajú hiba*: H_0 hamis, de elfogadjuk. Valószínűsége: $P_\vartheta(\underline{\xi} \notin \Xi_k)$, $\vartheta \in \Theta_1$.
  Az a cél, hogy ezek a hibák minél kisebbek legyenek. Egymás kárára javítható a két valószínűség, ha a megfigyelések száma rögzített.
- *Próba terjedelme*: α a próba terjedelme, ha $P_\vartheta(\underline{\xi} \in \Xi_k) \leq \alpha$, $\vartheta \in \Theta_0$.
  α a próba pontos terjedelme, ha $\sup_{\vartheta\in\Theta_0} P_\vartheta(\underline{\xi} \in \Xi_k) = \alpha$.

8 Klasszikus statisztikai próbák

- *u-próba*: Feltételezzük, hogy a minta normális eloszlású ($\xi_i \sim N(m, \sigma^2)$), $i = 1..n$, és hogy a szórás ismert.
  - *Egymintás*: A nullhipotézis az, hogy a várható érték megegyezik-e egy konkrét értékkel (m_0), másképpen fogalmazva azt vizsgáljuk, hogy a mintabeli átlag nem tér-e el szignifikánsan m_0-tól. Tehát $H_0 : m = m_0$, és kétoldali esetben $H_1 : m \neq m_0$, egyoldaliban pedig például $H_1 : m \geq m_0$ vagy $H_1 : m < m_0$.
    Az u-próba értéke: $u = \sqrt{n}\frac{\overline{\xi}-m_0}{\sigma}$. Ha igaz a nullhipotézis, akkor ez közel standard normális eloszlású.
    ε hibavalószínűséggel vizsgáljuk a hipotézist, ehhez szükségünk van a $\Phi(u_{1-\varepsilon}) = 1-\varepsilon$ értékre.
    Kétoldali esetben H_0-t elutasítjuk, ha $|u| > u_{1-\frac{\varepsilon}{2}}$, és elfogadjuk, ha $|u| \leq u_{1-\frac{\varepsilon}{2}}$.
    Egyoldali esetben $u > u_{1-\varepsilon}$ (jobb) és $u < u_{1-\varepsilon}$ (bal) esetét vizsgáljuk, ezen esetekben utasítjuk el H_0-t.
  - *Kétmintás*: Itt a feltételek a következők: $\xi_i \sim N(m_1, \sigma_1^2)$, $i = 1..n$ és $\eta_j \sim N(m_2, \sigma_2^2)$, $j = 1..m$. A szórások szintén ismertek. $H_0 : m_1 = m_2$, és $u = \frac{\overline{\xi}-\overline{\eta}}{\sqrt{\frac{\sigma_1^2}{n}+\frac{\sigma_2^2}{m}}}$. $H_1 : m_1 > m_2$, ez a felső (jobb?) oldali, $H_1 : m_1 < m_2$ pedig az alsó (bal?) ellenhipotézis.
- *t-próba*: Ennél a próbánál nem ismert a szórás, viszont ugyanúgy normális eloszlást feltételezünk, mint az u-próbánál. $\xi_i \sim N(m, \sigma^2)$, $i = 1..n$.
  - *Egymintás*: $H_0 : m = m_0$. Ellenhipotézis az u-próbához hasonlóan. $t = \sqrt{n}\frac{\overline{\xi}-m_0}{\sqrt{\sigma_*^2}}$, ahol σ_*^2 a korrigált tapasztalati szórásnégyzet, amit a mintából számíthatunk ki. (Megjegyzés: n helyett $n-1$-gyel osztunk a képletben.) Ez az érték t-eloszlású H_0 esetén, ami $n-1$ szabadságfokú. Más néven szokás ezt a próbát Student-próbának is nevezni.
  - *Kétmintás*: $\xi_i \sim N(m_1, \sigma_1^2)$, $i = 1..n$ és $\eta_j \sim N(m_2, \sigma_2^2)$, $j = 1..m$. Ez esetben sem ismert a szórás, viszont feltételezzük, hogy a két minta szórása megegyezik. Ekkor $t_{n+m-2} = \sqrt{\frac{nm(n+m-2)}{n+m}}\frac{\overline{\xi}-\overline{\eta}}{\sqrt{\sum(\xi_i-\overline{\xi})^2+\sum(\eta_j-\overline{\eta})^2}}$. $n+m-2$ a próba szabadságfoka.
- *f-próba*: Két minta esetén használható. Ez a próba szórások egyezőségének vizsgálatára alkalmas, tehát itt $H_0 : \sigma_1 = \sigma_2$. Ha a két minta szórásnégyzete megegyezik, akkor a hányadosuk 1-hez tart. $f_{n-1,m-1} = max(\frac{\sigma_1^2}{\sigma_2^2}, \frac{\sigma_2^2}{\sigma_1^2})$. A két szabadsági fok közül az első az f számlálójához tartozó minta elemszáma -1, a második a nevezőjéhez.
- *Welch-próba*: Más néven d-próba. Hasonló, mint a kétmintás t-próba, de itt a szórások egyezőségét nem kell feltenni. Szabadsági foka bonyolult képlettel számítható.
- *szekvenciális próbák*: $V_n = \frac{\prod f_1(x_i)}{\prod f_0(x_i)} = \frac{L_1(x)}{L_0(x)}$. f_0 a nullhipotézis szerinti sűrűségfüggvény, f_1 az ellenhipotézis szerinti. Adott egy A és egy B érték, $A < B$. Ha $V_n \geq B$, akkor elutasítjuk H_0-t, ha $V_n \leq A$, akkor elfogadjuk, és ha $A < V_n < B$, akkor új mintaelemet veszünk.
  Stein tétele szerint N 1 valószínűséggel véges. $N = min\{n : V_n \leq A \vee V_n \geq B\}$.
- *Minőség-ellenőrzés*: n_1 elemet nézünk, $c_1 < c_2$ és c_3 határértékek. Ha $X_1 \leq c_1$, akkor elfogadjuk H_0-t, ha $X_1 \geq c_2$, akkor elutasítjuk. Ha $c_1 < X_1 < c_2$, akkor megnézünk n_2 elemet, és ha $X_1 + X_2 \leq c_3$, akkor szintén elfogadjuk H_0-t. A várható mintaelemszám méri a hatékonyságát.
- *χ^2-próba*: $H_0 : A_1, ..., A_n$ teljes eseményrendszer. $P(A_i) = p_i, i = 1..n$, ν_i a gyakoriság. Ha teljesül a nullhipotézis, akkor $\frac{\nu_i}{n} \sim p_i$.
  $\chi^2 = \sum \frac{(\nu_i - np_i)^2}{np_i}$. Ez χ^2 eloszlású, aminek $r-1$ szabadságfoka van. r az összeadott csoportok száma. (Megjegyzés: ha túl kicsi lenne 1-1 csoportban a gyakoriság, akkor azokat összevonjuk.)
  χ^2-próbát használhatunk illeszkedés-, homogenitás- és függetlenségvizsgálatra is. (Megjegyzés: más képlet van mindhez.)
- *Egyéb próbák*:
  - *Kolmogorov–Szmirnov-próba*: 2 tapasztalati eloszlásfüggvény megegyezik-e (homogenitásvizsgálat), vagy 1 minta esetén megegyezik-e valamilyen eloszlásfüggvénnyel. $D_{m,n} = \max_x |F_n(x) - G_m(x)|$.
    X_i F eloszlásfüggvénnyel, Y_j G-vel. $H_0 : F \equiv G$.

– *Előjel-próba*: Hányszor teljesül, hogy valami pozitív.
– *Wilcoxon-próba*: (rangstatisztika), $P(X > Y) = \frac{1}{2}$ tesztelésére összeszámoljuk, hogy hány párra teljesül, hogy $X_i > Y_j$.

Chapter 6

6

Záróvizsga tételsor

6. Mesterséges intelligencia

Fekete Dóra, Gregorics Tibor

Mesterséges intelligencia

MI problémák és az útkeresési feladat kapcsolata. Állapottér reprezentáció. Heurisztikus útkereső algoritmusok: lokális keresések (hegymászó módszer, tabu-keresés, szimulált hűtés), visszalépéses keresés, heurisztikus gráfkereső eljárások (A, A*, A^C , B algoritmusok). Kétszemélyes játékok.

1 Bevezetés

Az MI az intelligens gondolkodás számítógépes reprodukálása szempontjából hasznos elveket, módszereket, technikákat kutatja, fejleszti, rendszerezi. Megoldandó feladatai: nehezek, mert ezek problématere hatalmas, a megoldás megkeresése kellő intuíció hiányában kombinatorikus robbanáshoz vezethet. A szoftver intelligensen viselkedik, és sajátos eszközöket használ. A reprezentáció átgondolt a feladat modellezéséhez, és az algoritmusok hatékonyak, heurisztikával megerősítve.

2 Útkeresési problémák

Útkeresési problémaként sok MI feladat fogalmazható meg úgy, hogy a feladat modellje alapján megadunk egy olyan élsúlyozott irányított gráfot, amelyben adott csúcsból adott csúcsba vezető utak jelképezik a feladat egy-egy megoldását. Ezt a feladat *gráfreprezentációjának* is szokás nevezni, amely magába foglal egy úgynevezett *δ-gráfot* (olyan élsúlyozott irányított gráf, ahol egy csúcsból kivezető élek száma véges, és az élek költségére megadható egy δ pozitív alsó korlát), az abban kijelölt startcsúcsot és egy vagy több célcsúcsot. Ebben a reprezentációs gráfban keresünk egy startcsúcsból kiinduló célcsúcsba futó utat, esetenként egy legolcsóbb ilyet.

3 Állapottér-reprezentáció

Állapottér-reprezentáció egy olyan lehetséges (de nem az egyetlen) módszer a feladatok modellezésére, amelyet aztán természetes módon lehet gráfreprezentációkét is megfogalmazni. Négy eleme van:

- *Állapottér*, amely a probléma homlokterében álló adat (objektum) lehetséges értékeinek (állapotainak) halmaza. Gyakran egy alaphalmaz, amelyet egy alkalmas invariáns leszűkít.
- *Műveletek* (előfeltétel+hatás), amelyek állapotból állapotba vezetnek.
- *Kezdőállapot*(ok) vagy azokat leíró kezdőfeltétel.
- *Célállapot*(ok) vagy célfeltétel.

Az *állapot-gráf* (egy speciális reprezentációs gráf) az állapotokat, mint csúcsokat, a műveletek hatásait, mint éleket tartalmazza. Az állapottér nem azonos a problématérrel, hiszen a problématér elemei a startcsúcsból kivezető utak (műveletsorozatok), nem pedig az állapotok (csúcsok). A megoldás a problématér egy eleme, ami egy olyan műveletsorozat, ami startcsúcsból célcsúcsba vezet.

4 Keresések

Egy általános *kereső rendszer* részei: a *globális munkaterület* (a keresés memóriája), a *keresési szabályok* (a memória tartalmát változtatják meg), és a *vezérlési stratégia* (adott pillanatban alkalmas szabályt választ). A vezérlési stratégiának van egy általános, elsődleges eleme (ez lehet nemmódosítható vagy

módosítható), lehet egy másodlagos (az alkalmazott reprezentációs modell sajátosságait kihasználó) eleme és a konkrét feladatra építő eleme. Ez utóbbi a *heurisztika*, a konkrét feladatból származó extra ismeret, amelyet közvetlenül a vezérlési stratégiába építünk be az eredményesség és a hatékonyság javítása céljából.

5 Lokális keresések

A lokális keresések egyetlen aktuális csúcsot és annak szűk környezetét tárolják a globális munkaterületen. Keresési szabályai az aktuális csúcsot minden lépésben a szomszédjai közül vett lehetőleg „jobb" gyerekcsúccsal cserélik le. A vezérlési stratégiájuk a „jobbság" eldöntéséhez egy *rátermettségi függvényt* használ, amely annál jobb értéket ad egy csúcsra, minél közelebb esik az a célhoz. Mivel a keresés „elfelejti", hogy honnan jött, a döntések nem vonhatók vissza, ez egy *nem-módosítható vezérlési stratégia*. Lokális kereséssel megoldható feladatok azok, ahol egy lokálisan hozott rossz döntés nem zárja ki a cél megtalálását. Ehhez vagy egy erősen összefüggő reprezentációs-gráf, vagy jó heurisztikára épített célfüggvény kell. Jellemző alkalmazás: adott tulajdonságú elem keresése, függvény optimumának keresése.

- *Hegymászó algoritmus*: Minden lépésben az aktuális csúcs legjobb gyermekére lép, de kizárja a szülőre való visszalépést. Zsákutcába (aktuális csúcsból nem vezet ki él) beragad, körök mentén végtelen ciklusba kerülhet, ha a rátermettségi függvény nem tökéletes.

- *Tabu keresés*: Az aktuális csúcson (n) kívül nyilvántartja még az eddig legjobbnak bizonyult csúcsot (n*) és az utolsó néhány érintett csúcsot; ez a (sor tulajdonságú) tabu halmaz. Minden lépésben az aktuális csúcs gyermekei közül, kivéve a tabu halmazban levőket, a legjobbat választja új aktuális csúcsnak, (ezáltal felismeri a tabu halmaz méreténél nem nagyobb köröket), frissíti a tabu halmazt, és ha n jobb, mint az n*, akkor n*-ot lecseréli n-re.

- *Szimulált hűtés algoritmusa*: A következő csúcs választása véletlenszerű. Ha a kiválasztott csúcs (r) célfüggvény-értéke jobb, mint az aktuális csúcsé (n), akkor odalép, ha rosszabb, akkor az új csúcs elfogadásának valószínűsége fordítottan arányos f(n) - f(r) különbséggel. Ez az arány ráadásul folyamatosan változik a keresés során: ugyanolyan különbség esetén kezdetben nagyobb, később kisebb valószínűséggel fogja a rosszabb értékű r csúcsot választani.

6 Visszalépéses keresések

A startcsúcsból az aktuális csúcsba vezető utat (és az arról leágazó még ki nem próbált éleket) tartja nyilván (globális munkaterületen), a nyilvántartott út végéhez egy új (ki nem próbált) élt fűzhet vagy a legutolsó élt törölheti (visszalépés szabálya), a visszalépést a legvégső esetben alkalmazza. A visszalépés teszi lehetővé azt, hogy egy korábbi továbblépésről hozott döntés megváltozhasson. Ez tehát egy *módosítható vezérlési stratégia*. A keresésbe sorrendi és vágó heurisztika építhető. Mindkettő lokálisan, az aktuális csúcsból kivezető, még ki nem próbált élekre vonatkozik. Visszalépés feltételei: zsákutca, zsákutca torkolat, kör, mélységi korlát.

- VL1 (nincs kör- és mélységi korlát figyelés) véges körmentes irányított gráfokon terminál, és ha van megoldás, akkor talál egyet.

- VL2 (általános) δ-gráfokon terminál, és ha van megoldás a mélységi korláton belül, akkor talál egyet.

Könnyen implementálható, kicsi memória igényű, mindig terminál, és ha van (a mélységi korlát alatt), akkor megoldást talál. De nem garantál optimális megoldást, egy kezdetben hozott rossz döntést csak nagyon sok lépés után képes korrigálni és egy zsákutca-szakaszt többször is bejárhat, ha abba többféle úton is el lehet jutni.

7 Gráfkeresések

A globális munkaterületén a startcsúcsból kiinduló már feltárt utak találhatók (ez az ún. *kereső gráf*), külön megjelölve az utak azon csúcsait, amelyeknek még nem (vagy nem eléggé jól) ismerjük a rákövetkezőit. Ezek a *nyílt csúcsok*. A keresés szabályai egy nyílt csúcsot terjesztenek ki, azaz előállítják (vagy újra előállítják) a csúcs összes rákövetkezőjét. A vezérlési stratégia a legkedvezőbb nyílt csúcs kiválasztására

törekszik, ehhez egy *kiértékelő függvényt* (f) használ. Mivel egy nyílt csúcs, amely egy adott pillanatban nem kerül kiválasztásra, később még kiválasztódhat, ezért itt egy módosítható vezérlési stratégia valósul meg. A keresés minden csúcshoz nyilvántart egy odavezető utat (π visszamutató pointerek segítségével), valamint az út költségét (g). Ezeket az értékeket működés közben alakítja ki, amikor a csúcsot először felfedezi vagy később egy olcsóbb utat talál hozzá. Mindkét esetben (amikor módosultak a csúcs ezen értékei) a csúcs nyílttá válik. Amikor egy már korábban kiterjesztett csúcs újra nyílt lesz, akkor a már korábban felfedezett leszármazottainál a visszafelé mutató pointerekkel kijelölt út költsége nem feltétlenül egyezik majd meg a nyilvántartott g értékkel, és az sem biztos, hogy ezek az értékek az eddig talált legolcsóbb útra vonatkoznak, vagyis előfordulhat, hogy elromlik a keresőgráf korrektsége.

- Nem-informált gráfkeresések: *mélységi gráfkeresés* ($f = -g$, minden (n, m) élre $c(n, m) = 1$), *szélességi gráfkeresés* ($f = g$, $c(n, m) = 1$), *egyenletes gráfkeresés* ($f = g$)
- Heurisztikus gráfkeresések f-je a h heurisztikus függvényre épül, amely minden csúcsban a hátralevő optimális h* költséget becsli. Ilyen az *előre tekintő gráfkeresés* ($f = h$), az *A algoritmus* ($f = g + h, h \geq 0$), az *A* algoritmus* ($f = g + h, h^* \geq h \geq 0$ − h megengedhető), az *A^C algoritmus* ($f = g + h, h^* \geq h \geq 0$, minden (n, m) élre $h(n) - h(m) \leq c(n, m)$), és *B algoritmus* (ahol az $f = g + h$, $h \geq 0$ helyett a g-t használjuk a kiterjesztendő csúcs kiválasztására azon nyílt csúcsok közül, amelyek f értéke kisebb, mint az eddig kiterjesztett csúcsok f értékeinek maximuma).

Véges δ-gráfokon minden gráfkeresés terminál, és ha van megoldás, talál egyet. A nevezetes gráfkeresések többsége végtelen nagy gráfokon is találnak megoldást, ha van megoldás. (Kivétel az előre-tekintő keresés és a mélységi korlátot nem használó mélységi gráfkeresés.) Az A*, A^C algoritmusok optimális megoldást találnak, ha van megoldás. Az A^C algoritmus egy csúcsot legfeljebb egyszer terjeszt csak ki. Egy gráfkeresés memória igényét a kiterjesztett csúcsok számával, futási idejét ezek kiterjesztéseinek számával mérjük. (Egy csúcs általában többször is kiterjesztődhet, de δ-gráfokban csak véges sokszor.) A* algoritmusnál a futási idő legrosszabb esetben exponenciálisan függ a kiterjesztett csúcsok számától, de ha olyan heurisztikát választunk, amelyre már A^C algoritmust kapunk, akkor a futási idő lineáris lesz. Persze ezzel a másik heurisztikával változik a kiterjesztett csúcsok száma is, így nem biztos, hogy egy A^C algoritmus ugyanazon a gráfon összességében kevesebb kiterjesztést végez, mint egy csúcsot többször is kiterjesztő A* algoritmus. A B algoritmus futási ideje négyzetes, és ha olyan heurisztikus függvényt használ, mint az A* algoritmus (azaz megengedhetőt), akkor ugyanúgy optimális megoldást talál (ha van megoldás) és a kiterjesztett csúcsok száma (mellesleg a halmaza is) megegyezik az A* algoritmus által kiterjesztett csúcsokéval.

8 Kétszemélyes (teljes információjú, zéró összegű, véges) játékok

A játékokat állapottér-reprezentációval szokás leírni, és az állapot-gráfot faként ábrázolják. A *győztes (vagy nem-vesztes) stratégia* egy olyan elv, amelyet betartva egy játékos az ellenfél minden lépésére tud olyan választ adni, hogy megnyerje (ne veszítse el) a játékot. Valamelyik játékosnak biztosan van győztes (nem-vesztes) stratégiája. Győztes (nem-vesztes) stratégia keresése a *játékfában* kombinatorikus robbanást okozhat, ezért e helyett részfa kiértékelést szoktak alkalmazni a soron következő jó lépés meghatározásához. A *minimax* algoritmus az aktuális állásból felépíti a játékfa egy részét, kiértékeli annak leveleit aszerint, hogy azok által képviselt állások milyen mértékben kedveznek nekünk vagy az ellenfélnek, majd szintenként váltakozva az ellenfél szintjein a gyerekcsúcsok értékeinek minimumát, a saját szintjeinken azok maximumát futtatjuk fel a szülőcsúcshoz. Ahonnan a gyökérhez kerül érték, az lesz soron következő lépésünk. A minimax legismertebb módosítása az *alfa-béta* algoritmus, amely egyfelől kisebb memória igényű (egyszerre csak egy ágat tárol a vizsgált részfából), másfelől egy sajátos vágási stratégia miatt jóval kevesebb csúcsot vizsgál meg, mint a minimax. Saját szinten α, ellenfelén β értéket adunk meg, kezdetben $-\infty$, illetve $+\infty$ értékkel. Visszalépéskor változtatunk rajta, α-t növeljük, β-t csökkentjük. Vágás akkor történik, ha az úton vannak olyan értékek, hogy $\alpha \geq \beta$. További módosítások még az átlagoló (legnagyobb m és legkisebb n darab érték átlagát vesszük), illetve a váltakozó mélységű kiértékelésű minimax (minden ágon reális értéket mutasson a kiértékelő függvény nyugalmi teszttel), továbbá a negamax algoritmus (ellenfél szintjén (-1)-szeres érték, maximumot választunk minden szinten az ellentettekből).

Chapter 7

7

Záróvizsga tételsor
7. Programozás

Ancsin Ádám

Programozás

Egyszerű programozási feladat megoldásának lépései (specifikálás, tervezés, megvalósítás, tesztelés). Az adattípus fogalma (típusspecifikáció, műveletek, reprezentáció, invariáns, implementáció). A visszavezetés módszere. A felsoroló típus specifikációja. Felsorolóra megfogalmazott programozási tételek (összegzés, számlálás, maximum kiválasztás, feltételes maximumkeresés, lineáris keresés, kiválasztás). Nevezetes gyűjtemények (intervallum, tömb, sorozat, halmaz, szekvenciális inputfájl) felsorolói.

1 Egyszerű programozási feladat megoldásának lépései

1.1 Bevezetés

Egy programozási feladat megoldása a kódoláson túl jó néhány tevékenységet tartalmaz. Az első teendő a feladat pontos meghatározása, a specifikáció. Ez a feladat szöveges és formalizált, matematikai leírásán (a specifikáció ún. szűkebb értelmezésén) túl tartalmazza a megoldással szemben támasztott követelményeket, környezeti igényeket is (ami a specifikáció ún. tágabb értelmezése).

A specifikáció alapján meg lehet tervezni a programot, elkészülhet a megoldás algoritmusa és az algoritmus által használt adatok leírása. Az algoritmus és az adatszerkezet finomítása egymással párhuzamosan halad, egészen addig a szintig, amelyet a programozó ismeretei alapján már könnyen, hibamentesen képes kódolni. Gyakran előfordul, hogy a tervezés során derül fény a specifikáció hiányosságaira, így itt visszalépésekre számíthatunk.

Az algoritmusírás után következhet a kódolás. Ha a feladat kitűzője nem rögzítette, akkor ez előtt választhatunk a megoldáshoz programozási nyelvet. A kódolás eredménye a programozási nyelven leírt program.

A program első változatban általában sohasem hibátlan, a helyességéről csak akkor beszélhetünk, ha meggyőződtünk róla. A helyesség vizsgálatának egyik lehetséges módszere a tesztelés. Ennek során próbaadatokkal próbáljuk ki a programot, s az ezekre adott eredményből következtetünk a helyességre. (Ne legyenek illúzióink afelől, hogy teszteléssel eldönthető egy program helyessége. Hisz hogy valójában helyes-e a program – sajnos – nem következik abból, hogy nem találtunk hibát.)

Ha a tesztelés során hibajelenséggel találkozunk, akkor következhet a hibakeresés, a hibajelenséget okozó utasítás megtalálása, majd pedig a hibajavítás. A hiba kijavítása több fázisba is visszanyúlhat. Elképzelhető, hogy kódolási hibát kell javítanunk, de az is lehet, hogy a hibát már a tervezésnél követtük el. Javítás után újra tesztelni kell, hiszen – legyünk őszinték magunkhoz!– nem kizárt, hogy hibásan javítunk, illetőleg – enyhe optimizmussal állítjuk:– a javítás újabb hibákat fed fel, ...

E folyamat végeredménye a helyes program. Ezzel azonban még korántsem fejeződik be a programkészítés. Most következnek a minőségi követelmények. Egyrészt a hatékonyságot kell vizsgálnunk (végrehajtási idő, helyfoglalás), másrészt a kényelmes használhatóságot. Itt újra visszaléphetünk a kódolási, illetve a tervezési fázisba is. Ezzel elérkeztünk a jó programhoz.

1.2 Specifikáció

A programkészítés menetének első lépése a feladat meghatározása, precíz "újrafogalmazása". Milyen is legyen, mit várjunk el tőle? Nézzünk meg néhány – jónak tűnő – követelményt egyelőre címszavakban! (A továbbiakban a specifikáció szűkebb értelmezéséről lesz szó.) A specifikáció legyen:

- helyes, egyértelmű, pontos, teljes
- rövid, tömör, ami legegyszerűbben úgy érhető el, hogy ismert formalizmusokra építjük

- szemléletes, érthető (amit időnként nehezít a formalizáltság)

A specifikáció első közelítésben lehetne a feladatok szövege. Ez azonban több problémát vethet fel:

- mi alapján adjuk meg a megoldást
- mit is kell pontosan megadni?

Például az a feladat, hogy adjuk meg N ember közül a legmagasabbat. A legmagasabb ember megadása mit jelent? Adjuk meg a sorszámát, vagy a nevét, vagy a személyi számát, vagy a magasságát, esetleg ezek közül mindegyiket? Tanulságként megállapíthatjuk, hogy a specifikációnak tartalmaznia kell a bemenő és a kimenő adatok leírását.

```
Bemenet:
    N : az emberek száma,
    A : a magasságukat tartalmazó sorozat.

Kimenet:
    MAX : a legmagasabb ember sorszáma.
```

Tudjuk-e, hogy a bemenő, illetve a kimenő változók milyen értéket vehetnek fel? Például az emberek magasságát milyen mértékegységben kell megadni? Az eredményül kapott sorszám milyen érték lehet: 1-től sorszámozunk, vagy 0-tól? Megállapíthatjuk tehát, hogy a specifikációban a bemeneti és a kimeneti változók értékhalmazát is meg kell adnunk.

```
Bemenet:
    N : az emberek száma, természetes szám;
    A : a magasságukat tartalmazó sorozat, egész számok, amelyek a magasságot
        centiméterben fejezik ki (a sorozatot 1-től N-ig indexeljük).
Kimenet:
    MAX : a legmagasabb ember sorszáma, 1 és N közötti természetes szám.
```

Most már a bemenő és a kimenő változók értékhalmazát pontosan meghatároztuk, csupán az a probléma, hogy a feladatban használt fogalmakat és az eredmények kiszámítási szabályát nem definiáltuk. A specifikációnak tehát tartalmaznia kell a feladatban használt fogalmak definícióját, valamint az eredmény kiszámítási szabályát. Itt lehetne megadni a bemenő adatokra vonatkozó összefüggéseket is. A bemenő, illetve a kimenő adatokra kirótt feltételeket nevezzük előfeltételnek, illetve utófeltételnek. Az előfeltétel nagyon sokszor egy azonosan igaz állítás, azaz a bemenő adatok értékhalmazát semmilyen "külön" feltétellel nem szorítjuk meg.

```
Bemenet:
      N : az emberek száma, természetes szám,
      A : a magasságukat tartalmazó sorozat, egész számok,
         amelyek a magasságot centiméterben tartalmazzák (a sorozatot 1-tol N-ig indexeljük).
Kimenet:
      MAX : a legmagasabb ember sorszáma, 1 és N közötti természetes szám.

Elofeltétel:
     A[i]-k pozitívak.

Utófeltétel:
    MAX olyan 1 és N közötti szám, amelyre A[MAX] nagyobb vagy egyenlo,
    mint a sorozat bármely eleme (az 1. és az N. között).
```

Újabb probléma merülhet fel bármelyik feladattal kapcsolatban: az eddigiek alapján a "várttól" lényegesen különböző – nyugodtan állíthatjuk: "banális" –, az elő- és utófeltételnek megfelelő megoldást is tudunk készíteni.

Itt persze arról a hallgatólagos (tehát még meg nem fogalmazott, ki nem mondott) feltételezésről van szó, hogy a bemeneti változók értéke nem változik meg. Ez sajnos nem feltétlenül igaz. A probléma megoldására kétféle utat követhetünk (a későbbiekben mindkettőt alkalmazni fogjuk):

- az utófeltételbe automatikusan beleértjük, hogy "és a bemeneti változók értéke nem változik meg", s külön kiemeljük, ha mégsem így van;
- az elő- és az utófeltételt a program paramétereire fogalmazzuk meg, amelyeket formailag megkülönböztetünk a program változóitól, és emiatt nem a paraméterek fognak változni, hanem a programbeli változók (ebben az esetben természetesen az elő- és az utófeltételben meg kell fogalmazni a paraméterek és a megfelelő programbeli változók értékének azonosságát).

A második megoldásból az következik, hogy meg kell különböztetnünk egymástól a feladat és a program elő–, illetve utófeltételét! Ez hosszadalmasabbá – bár precízebbé – teszi a feladat megfogalmazását, emiatt ritkábban fogjuk alkalmazni.

Előfordulhat, hogy a feladat megfogalmazása alapján nem lehet egyértelműen meghatározni az eredményt, ugyanis az utófeltételnek megfelelő több megoldás is létezik. Ez a jelenség a feladat ún. nemdeterminisztikussága. Ehhez a nemdeterminisztikus feladathoz tehát determinisztikus programot kell írnunk, aminek az utófeltétele már nem engedheti meg a nem egyértelműséget, a nemdeterminisztikusságot. E probléma miatt tehát mindenképpen meg kell különböztetnünk egymástól a feladat és a program elő–, illetve utófeltételét!

```
Bemenet:
     N : az emberek száma, természetes szám,
     A : a magasságukat tartalmazó sorozat, egész számok,
         amelyek a magasságot centiméterben tartalmazzák (a sorozatot 1-tol N-ig indexeljük).
Kimenet:
     MAX : a legmagasabb ember sorszáma, 1 és N közötti természetes szám.

Elofeltétel:
   A[i]-k pozitívak.

    Utófeltétel:
      MAX olyan 1 és N közötti szám, amelyre A[MAX] nagyobb vagy egyenlo,
    mint a sorozat bármely eleme (az 1. és az N. között).

Program utófeltétel:
    MAX olyan 1 és N közötti szám, amelyre A[MAX] nagyobb vagy egyenlo,
    mint a sorozat bármely eleme (az 1. és az N. között) és elotte
    nincs vele egyenlo.
```

Megállapíthatjuk ebből, hogy a program utófeltétele lehet szigorúbb, mint a feladaté, emellett az előfeltétele pedig lehet gyengébb.

Visszatekintve a specifikáció eddig "bejárt pályájára" egy szemléletes modellje körvonalazódik a feladatmegoldásnak. Nevezetesen: nyugodtan mondhatjuk azt, hogy a feladatot megoldó program egy olyan automatát határoz meg, amelynek pillanatnyi állapota a feladat paraméterei (a program változói) által "kifeszített" halmaz egy eleme. (E halmaz annyi dimenziós, ahány paraméterváltozója van a programnak; minden dimenzió egyik változó értékhalmaza. Tehát egy konkrét időpillanatban e "gép" állapota: a változóinak abban a pillanatban érvényes értékeinek együttese.) Ezt a halmazt nevezzük a program állapotterének. Amikor megfogalmazzuk az előfeltételt, akkor tulajdonképpen kihasítjuk ebből az állapottérből azt a részt (azt az altért), amelyből indítva elvárhatjuk az automatánktól (amit a megoldó program vezérel), hogy a helyes eredményt előállítja egy végállapotában. A végállapotot jelöltük ki az utófeltétellel.

Ezt a modellt elfogadva adódik még egy további megoldásra váró kérdés. Akkor ugyanis, amikor a programot írjuk, lépten-nyomon a részeredmények tárolására újabb és újabb változókat vezetünk be. Fölvetődik a kérdés: hogyan egyeztethető össze az imént elképzelt modellel? A válasz egyszerű: minden egyes újabb változó egy újabb dimenziót illeszt az eddig létrejött állapottérhez. Tehát a programozás folyamata – leegyszerűsítve a dolgot – nem áll másból, mint annak pontosításából, hogy hogyan is nézzen ki a megoldó automata állapottere (és persze: hogyan kell az egyik állapotból a másik állapotba jutnia). A feladatban szereplő paraméterek meghatározta "embrionális" állapotteret hívhatjuk paramétertérnek, ami csak altere a program valódi állapotterének. Ez is azt sugallja, hogy a feladat előfeltétele gyengébb (azaz az általa kijelölt állapothalmaz) lehet, mint a program előfeltétele.

Foglaljuk most össze, hogy melyek a specifikáció részei! Ezek az eddigiek, valamint a programra vonatkozó további megkötések lesznek.

1. A feladat specifikálása
   - a feladat szövege,
   - a bemenő és a kimenő adatok elnevezése, értékhalmazának leírása,
   - a feladat szövegében használt fogalmak definíciói (a fogalmak fölhasználásával),
   - a bemenő adatokra felírt előfeltétel (a fogalmak fölhasználásával),
   - a kimenő adatokra felírt utófeltétel.
2. A program specifikálása
   - a bemenő és a kimenő adatok elnevezése, értékhalmazának leírása,
   - (a feladat elő-, illetve utófeltételétől esetleg különböző) program elő- és utófeltétel,
   - a feladat megfogalmazásában használt fogalmak definíciói.

Ezek az absztrakt specifikáció elemei. Az alábbiak másodlagos, mondhatjuk: technikai specifikáció részei:

- a program környezetének leírása (számítógép, memória- és perifériaigény, programozási nyelv, szükséges fájlok stb.),
- a programmal szembeni egyéb követelmények (minőség, hatékonyság, hordozhatóság stb.).

A technikai specifikáció nélküli leírást a program szűkebb specifikációjának nevezik.

Progos specifikáció:

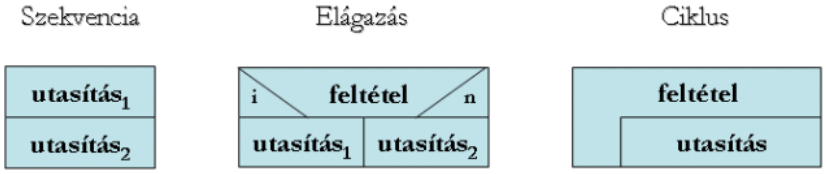
$A = (N : \mathbb{N}, A : \mathbb{N}^{1..N})$
$Ef = (\forall i \in 1..N : A_i > 0)$
$Uf = (Ef \wedge \forall i \in 1..N : A_{MAX} >= A_i \wedge \forall j \in 1..MAX - 1 : A_i < A_{MAX})$

1.3 Tervezés

A tervezés során algoritmusleíró eszközöket használunk, amelynek célja a feladatok megoldásának leírása programozási nyelvtől független nyelven. A programozási nyelvek ugyanis szigorú szintaxisúak, a tervezés szempontjából lényegtelen sallangokat tartalmaznak. A programozási nyelven történő tervezés esetén nehézzé válhat a program átírása más nyelvre, más gépre.

Többféle algoritmusleíró eszköz is létezik, mi tanulmányaink során a struktogramot alkalmaztuk.

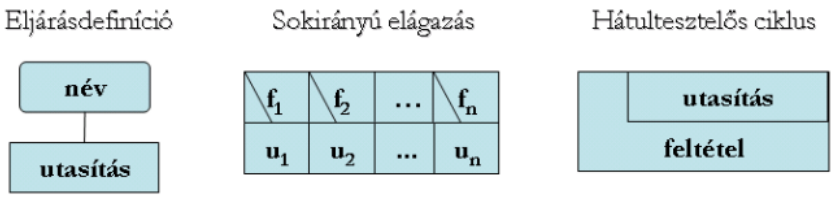
A struktogram a programgráfot élek nélkül ábrázolja. Így egyetlen egy alapelem marad, a téglalap. Ezzel az alapelemmel építhetjük fel a szokásos strukturált alapszerkezeteket (és csak azokat).



ábra 1: A struktogram összetett alapszerkezetei.

Szekvenciánál a téglalapok egymás alatti sorrendje dönti el a végrehajtás sorrendjét. Az elágazásfeltétel igaz értéke esetén az i betűvel jelölt bal oldali téglalap utasítását kell végrehajtani, hamis értéke esetén pedig az n betűvel jelölt jobb oldali téglalapét. Ha az elágazás valamelyik ága üres, akkor a neki megfelelő téglalap is üres marad. A ciklus elöltesztelős, azaz a benne levő utasítást mindaddig végre kell hajtani, amíg a feltétel igaz.

Az utasítások helyén lehet egyetlen elemi utasítás, lehet a három algoritmikus szerkezet valamelyike, és lehet egy eljáráshívás. Ezt a leíróeszközt még többféle elemmel szokták bővíteni: az eljárásdefinícióval, a sokirányú elágazással, illetve a hátultesztelős ciklussal.



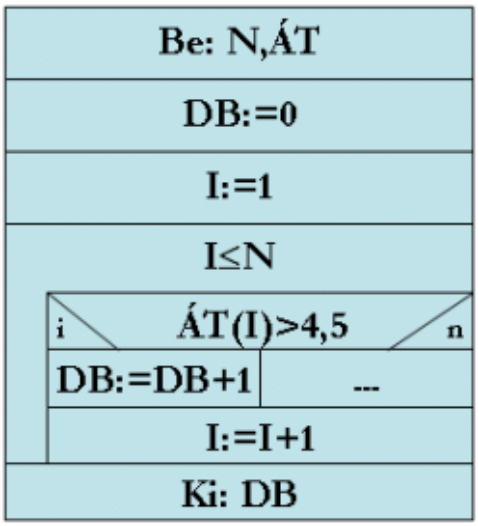
ábra 2: A struktogram további összetett alapszerkezetei.

Sokirányú elágazásnál azt az ágat kell végrehajtani, amelynek igaz értékű a feltétele (közülük minden esetben pontosan egy teljesülhet).

A lokális adatokat az eljárások téglalapjai mellett, az eljárásnév után sorolhatjuk fel.

Nézzük meg ezzel az eszközzel leírva a következő példát!

Feladat: N tanuló év végi átlagának ismeretében adjuk meg a jeles átlagú tanulók számát!



ábra 3: A példafeladat megoldása struktogrammal.

1.4 Megvalósítás

A.k.a. kódolás.

1.5 Tesztelés

A tesztelés célja, hogy minél több hibát megtaláljunk a programban. Ahhoz, hogy az összes hibát fölfedezzük, kézenfekvőnek tűnik a programot kipróbálni az összes lehetséges bemenő adattal. Ez azonban sajnos nem lehetséges.
Példaként tekintsük a következő - pszeudokóddal megadott - egyszerű programot:

```
Program:
    Változó A,B:Egész
    Be: A,B
    Ki: A/B
Program vége.
```

Mivel 2^{16} különböző értékű egész számot tudunk tárolni, ezért az összes lehetőség 2^{32}, aminek a leírásához már 9 számjegyre van szükség. Ez rengeteg időt venne igénybe, így nem is járható út.

Ha ezt a programot olyan bemenő adatokkal próbáljuk ki, amelyben A=0 vagy B=1, akkor a program helyesen működik, a hibát nem tudjuk felfedezni. Ezután azt gondolhatnánk, hogy reménytelen

helyzetbe kerültünk: hiszen minden lehetséges adattal nem tudjuk kipróbálni a programot; ha pedig kevesebbel próbáljuk ki, akkor lehet, hogy nem vesszük észre a hibákat. A helyzet azért nem ennyire rossz: célunk csak az lehet, hogy a tesztelést olyan módszerrel hajtsuk végre, amellyel a próbák száma erősen lecsökkenthető.

Tesztesetnek a be- és kimeneti adatok és feltételek együttes megadását nevezzük. Akkor tudunk a tesztelés eredményeiről bármit is mondani, ha van elképzelésünk arról, hogy adott bemenő adatra milyen eredményt várunk.

Fogalmazzuk meg a tesztelés alapelveit:

- A jó teszteset az, ami nagy valószínűséggel egy még felfedetlen hibát mutat ki a programban. Például két szám legnagyobb közös osztóját számoló programot az [5,5] adatpár után a [6,6]-tal teljesen felesleges kipróbálni (ugyanis igencsak rafinált, valószínűtlen elírás esetén viselkedhet a program [6,6]-ra másként, mint [5,5]-re).
- A teszteset nemcsak bemenő adatokból, hanem a hozzájuk tartozó eredményekből is áll. Egyébként nem tudnánk a kapott eredmény helyes vagy hibás voltáról beszélni. A későbbi felhasználás miatt célszerű a teszteseteket is leírni a fejlesztői dokumentációban vagy egy önálló tesztelési jegyzőkönyvben.
- A meg nem ismételhető tesztesetek kerülendők, feleslegesen megnövelik a program-tesztelés költségeit, idejét. Nem is beszélve arról a bosszúságról, amikor a programunk egy hibás futását nem tudjuk megismételni, és így a hiba is felfedetlen marad.
- Teszteseteket mind az érvénytelen, mind az érvényes adatokra kell készíteni.
- Minden tesztesetből a lehető legtöbb információt "ki kell bányászni", azaz minden teszteset eredményét alaposan végig kell vizsgálni. Ezzel jelentősen csökkenthető a szükséges próbák száma.
- Egy próba eredményeinek vizsgálata során egyaránt fontos megállapítani, hogy miért nem valósít meg a program valamilyen funkciót, amit elvárunk tőle, illetve hogy miért végez olyan tevékenységeket is, amelyeket nem feltételeztünk róla.
- A program tesztelését csak a program írójától különböző személy képes hatékonyan elvégezni. Ennek oka, hogy a tesztelés nem "jóindulatú" tevékenység, saját munkájának vizsgálatához mindenki úgy áll hozzá, hogy önkéntelenül jónak feltételezi.

A programtesztelés módszereit két csoportba oszthatjuk aszerint, hogy a tesztelés során végrehajtjuk-e a programot, vagy nem. Ha csak a program kódját vizsgáljuk, akkor statikus (erről nem esik több szó), ha a programot végre is hajtjuk a tesztelés során, akkor dinamikus tesztelésről beszélünk.

Dinamikus tesztelési módszerek

A dinamikus tesztelési módszerek alapelve az, hogy a programot működés közben vizsgáljuk. Teszteseteket kétféle módon tudunk választani. Egy lehetőség az ún. feketedoboz-módszer, más néven adatvezérelt tesztelés. E módszer alkalmazásakor a tesztelő nem veszi figyelembe a program belső szerkezetét, pontosabban nem azt tekinti elsődleges szempontnak, hanem a teszteseteket a feladat meghatározás alapján választja meg.

A cél természetesen a lehető leghatékonyabb tesztelés elvégzése, azaz az összes hiba megtalálása a programban. Ez ugyan elvileg lehetséges, kimerítő bemenet tesztelést kell végrehajtani, a programot ki kell próbálni az összes lehetséges bemenő adatra. Ezzel a módszerrel azonban, mint korábban láttuk, mennyiségi akadályba ütközhetünk.

Egy másik lehetőség a fehérdoboz-módszer (logika vezérelt tesztelés). Ebben a módszerben a tesztesetek megválasztásánál lehetőség van a program belső szerkezetének figyelembevételére is.

A cél a program minél alaposabb tesztelése, erre jó módszer a kimerítő út tesztelés. Ez azt jelenti, hogy a programban az összes lehetséges utat végigjárjuk, azaz annyi tesztesetet hozunk létre, hogy ezt elérhessük vele. Az a probléma, hogy még viszonylag kis programok esetén is igen nagy lehet a tesztelési utak száma. Gondoljunk a ciklusokra! Sőt ezzel a módszerrel a hiányzó utakat nem lehet felderíteni.

Mivel sem a fehérdoboz-módszerrel, sem a feketedoboz-módszerrel nem lehetséges a kimerítő tesztelés, el kell fogadnunk, hogy nem tudjuk egyetlen program hibamentességét sem szavatolni. A további cél

ezek után az összes lehetséges teszteset halmazából a lehető leghatékonyabb teszteset-csoport kiválasztása lehet.

A tesztelés hatékonyságát kétféle jellemző határozza meg: a tesztelés költsége és a felfedett hibák aránya. A leghatékonyabb teszteset-csoport tehát minimális költséggel maximális számú hibát fed fel.

A feketedoboz- és fehérdoboz-teszteken kívül még érdemes megemlíteni olyan speciális teszteket, amikor nem a helyesség belátása a cél. Ilyen pl. a stresszteszt (nagy adatmennyiséget hogyan bír kezelni a program, jól skálázódik-e) vagy a hatékonysági teszt (végrehajtási idő tesztelése).

2 Az adattípus fogalma

2.1 Alapfogalmak, jelölések

- A^* az A-beli véges sorozatok halmazát, A^∞ az A-beli végtelen sorozatok halmazát jelöli. A kettő uniója $A^{**} = A^* \cup A^\infty$ pedig az A-beli véges vagy végtelen sorozatok halmazát jelenti.
- Legyen $R \subseteq A \times \mathbb{L}$ egy logikai reláció. Ekkor az R igazsághalmaza $\lceil R \rceil ::= R^{-1}(\{igaz\})$
- Legyen I egy véges halmaz és legyenek $A_i, i \in I$ tetszőlege véges vagy megszámolható, nem üres halmazok. Ekkor az $A = \underset{i \in I}{\times} A_i$ halmazt állapottérnek, az A_i halmazokat pedig típusértékhalmazoknak nevezzük.
- Feladat: feladatnak nevezünk egy $F \subseteq A \times A$ relációt.
  A feladat fenti definíciója természetes módon adódik abból, hogy a feladatot egy leképezésnek tekintjük az állapottéren, és az állapottér minden pontjára megmondjuk, hova kell belőle eljutni, ha egyáltalán el kell jutni belőle valahova.
- Program:
  Programnak nevezzük az $S \subseteq A \times A^{**}$ relációt, ha
  1. $\mathcal{D}_S = A$ (az állapottér minden pontjához rendel valamit, azaz a program minden pontban csinál valamit)
  2. $\forall \alpha \in \mathcal{R}_S : \alpha = red(\alpha)$ (az állapot megváltozik, vagy ha mégsem, az az abnormális működés jele)
  3. $\forall a \in A : \forall \alpha \in S(A) : |\alpha| \neq 0$ és $\alpha_1 = a$

  A fenti definícióval a "működés" fogalmát akarjuk absztrakt módon megfogalmazni.

2.2 Típusspecifikáció

Először bevezetünk egy olyan fogalmat, amelyet arra használhatunk, hogy pontosan leírjuk a követelményeinket egy típusértékhalmazzal és a rajta végezhető műveletekkel szemben.

A $\mathcal{T}_\mathcal{S} = (H, I_S, \mathbb{F})$ hármast típusspecifikációnak nevezzük, ha teljesülnek rá a következő feltételek:

1. H az alaphalmaz,
2. $I_S : H \to \mathbb{L}$ a specifikációs invariáns,
3. $T_\mathcal{T} = \{(\mathcal{T}, x) | x \in \lceil I_S \rceil\}$ a típusértékhalmaz,
4. $\mathbb{F} = \{F_1, F_2, ..., F_n\}$ a típusműveletek specifikációja, ahol
   $\forall i \in [1..n] : F_i \subseteq A_i \times A_i$, $A_i = A_{i_1} \times ... \times A_{i_{n_i}}$ úgy,
   hogy $\exists j \in [1..n_i] : A_{i_j} = T_\mathcal{T}$

Az alaphalmaz és az invariáns tulajdonság segítségével azt fogalmazzuk meg, hogy mi az a halmaz, $T_\mathcal{T}$, amelynek elemeivel foglalkozni akarunk, míg a feladatok halmazával azt írjuk le, hogy ezekkel az elemekkel milyen műveletek végezhetők el.

Az állapottér definíciójában szereplő típusértékhalmazok mind ilyen típusspecifikációban vannak definiálva. Az állapottér egy komponensét egy program csak a típusműveleteken keresztül változtathatja meg.

2.3 Típus

Vizsgáljuk meg, hogy a típusspecifikációban leírt követelményeket hogyan valósítjuk meg.

A $\mathcal{T} = (\rho, I, \mathbb{S})$ hármast típusnak nevezzük, ha

1. $\rho \subseteq E^* \times T$ a reprezentációs függvény (reláció),
   T a típusértékhalmaz,
   E az elemi típusértékhalmaz
2. $I : E^* \to \mathbb{L}$ típusinvariáns
3. $\mathbb{S} = \{S_1, S_2, ..., S_m\}$, ahol
   $\forall i \in [1..m] : S_i \subseteq B_i \times B_i^{**}$ program, $B_i = B_{i_1} \times ... \times B_{i_{m_i}}$ úgy,
   hogy $\exists j \in [1..m_i] : B_{i_j} = E^*$ és $\nexists j \in [1..m_i] : B_{i_j} = T$

A típus első két komponense az absztrakt típusértékek reprezentációját írja le, míg a programhalmaz a típusműveletek implementációját tartalmazza. Az elemi típusértékhalmaz lehet egy tetszőleges másik típus típusértékhalmaza vagy egy, valamilyen módon definiált legfeljebb megszámolható halmaz.

2.4 Invariáns

Az invariáns lényege, hogy ezt a tulajdonságot soha nem sérthetjük meg. Például halmaz típus esetén nem szabad, hogy megsérüljön az az invariáns tulajdonság, hogy egy halmazban egy elem csak egyszer fordulhat elő.

2.5 Reprezentáció

Azt, hogy egy típust milyen típusok segítségével, milyen módszerrel, stb., valósítottunk meg, reprezentációnak nevezzük. Például egy verem típust meg lehet valósítani tömb segítségével, de láncolt listával is.

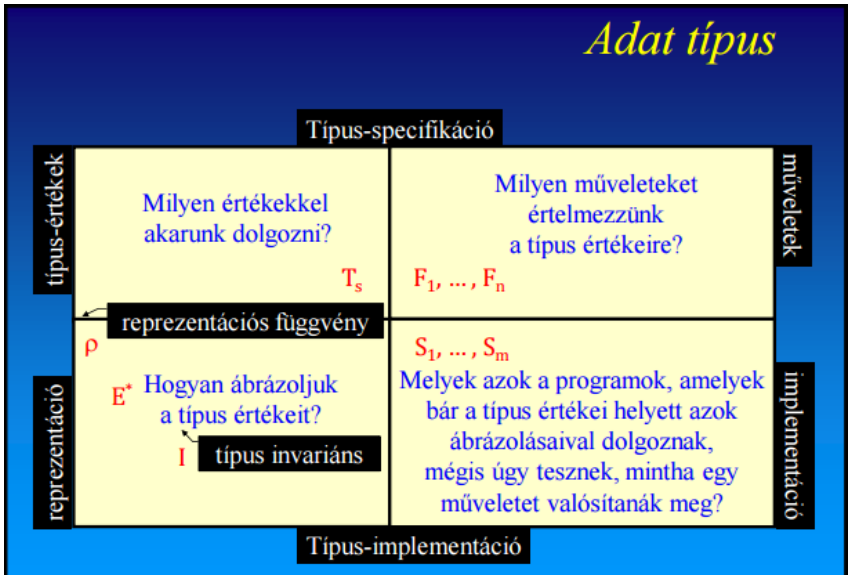
A reprezentáció a típusspecifikáció típusértékhalmazának leképezése a konkrét típusban, amit a reprezentációs függvény ad meg.

2.6 Implementáció

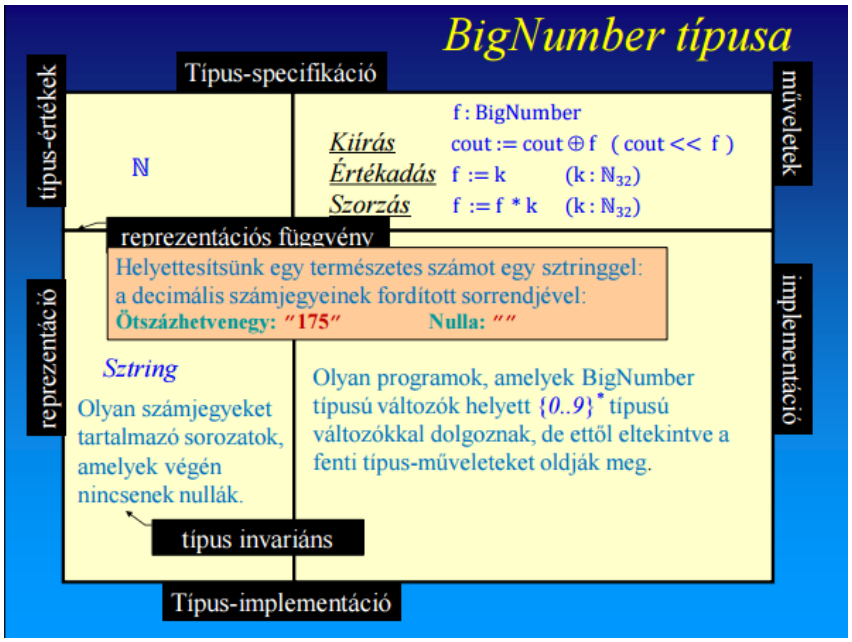
Az implementáció a típusspecifikáció típusműveleteinek megvalósítása a konkrét típus programhalmaza által.

Az implementáció során a típus megvalósításakor a típusértékhalmaz megadását követően definiálni kell a típusműveleteket. Ahogyan a modellben is, a gyakorlatban is az állapottér változásait a program csak a típusműveleteken keresztül végezheti el.

2.7 Emészthetőbb módon



ábra 4: Adattípus



ábra 5: BigNumber példa

3 A visszavezetés módszere

A programozási feladatok megoldásához különböző programozási mintákat, ún. programozási tételeket használunk fel, ezekre vezetjük vissza a megoldást.

Lépései:

1. Megsejtjük a feladatot megoldó programozási tételt.
2. Specifikáljuk a feladatot a programozási tétel jelöléseivel.
3. Megadjuk a programozási tétel és a feladat közötti eltéréseket:

- intervallum határok: konkrét érték vagy kifejezés (pl. $[1..\frac{n}{2}]$), a típusuk a $\mathbb{Z}$ helyett lehet annak valamely része (pl. $\mathbb{N}$)
- $\beta : [m..n] \to \mathbb{L}$ és/vagy $f : [m..n] \to H$ konkrét megfelelői
- a H megfelelője a szükséges művelettel
  - $(H, >)$ helyett pl. $(\mathbb{Z}, >)$ vagy $(\mathbb{Z}, <)$
  - $(H, +)$ helyett pl. $(\mathbb{Z}, +)$ vagy $(\mathbb{R}, *)$
- a változók átnevezése

4. A különbségek figyelembe vételével a tétel algoritmusából elkészítjük a konkrét feladatot megoldó algoritmust.

4 Felsoroló, a felsoroló típus specifikációja

A gyűjtemény (tároló, kollekció, iterált) egy olyan adat (objektum), amely valamilyen elemek tárolására alkalmas.

- Ilyenek az összetett szerkezetű, de különösen az iterált szerkezetű típusok értékei: halmaz, sorozat (verem, sor, fájl), fa, gráf
- De vannak úgynevezett virtuális gyűjtemények is: pl. egész számok egy intervallumának elemei, vagy egy természetes szám prím-osztói

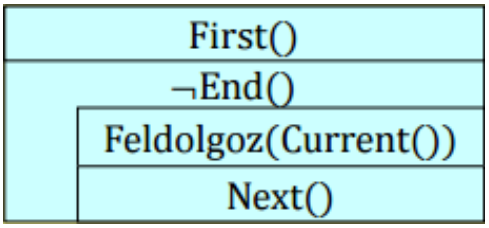
Egy gyűjtemény feldolgozásán a benne levő elemek feldolgozását értjük.

- Keressük a halmaz legnagyobb elemét!
- Hány negatív szám van egy számsorozatban?
- Válogassuk ki egy fa leveleiben elhelyezett értékeket!
- Járjuk be az [m .. n] intervallum minden második elemét visszafelé!
- Adjuk össze az n természetes szám prím-osztóit!

A feldolgozni kívánt elemek felsorolását (bejárását) az alábbi műveletekkel szabványosítjuk:

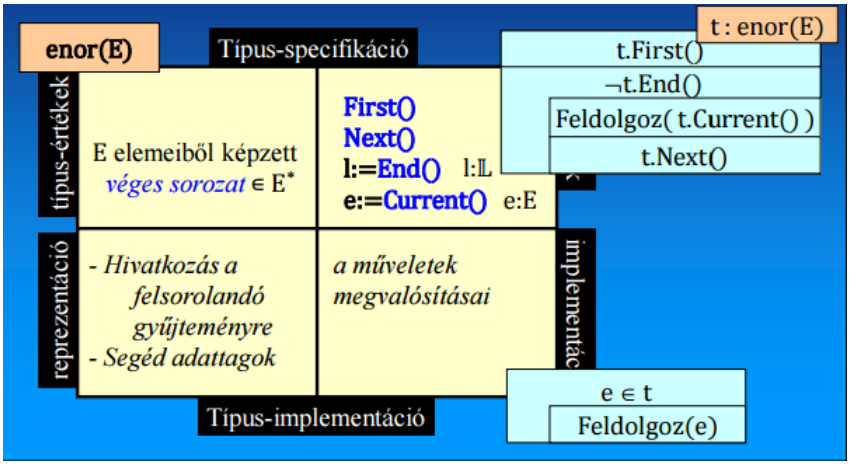
- First() : Rááll a felsorolás első elemére, azaz elkezdi a felsorolást
- Next() : Rááll az elkezdett felsorolás soron következő elemére
- End() : Mutatja, ha a felsorolás végére értünk
- Current() : Visszaadja a felsorolás aktuális elemét

Egy felsorolásnak különböző állapotai vannak (indulásra kész, folyamatban van, befejeződött), és a műveletek csak bizonyos állapotokban értelmezhetők (máshol a hatásuk nem definiált). A feldolgozó algoritmus garantálja, hogy a felsoroló műveletek mindig megfelelő állapotban kerüljenek végrehajtásra.



ábra 6: A felsorolás algoritmusa

A felsorolást sohasem a felsorolni kívánt gyűjtemény, hanem egy külön felsoroló objektum végzi.



ábra 7: A felsoroló objektum és típusa

5 Felsorolóra megfogalmazott programozási tételek

Programozási tételek felsorolókra

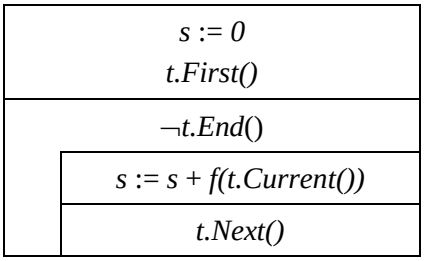
Összegzés

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum és egy $f:E\rightarrow H$ függvény. A *H* halmazon értelmezzük az összeadás asszociatív, baloldali nullelemes műveletét. Határozzuk meg a függvénynek a *t* elemeihez rendelt értékeinek összegét! (Üres felsorolás esetén az összeg értéke definíció szerint a nullelem: *0*).

Specifikáció:

$A = (t{:}enor(E), s{:}H)$
$Ef = (t=t')$
$Uf = (s = \sum_{e \in t'} f(e))$

Algoritmus:



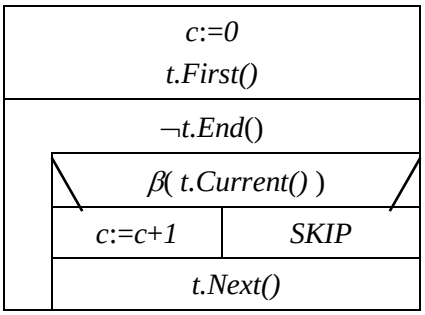
Számlálás

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum és egy $\beta:E\rightarrow\mathbb{L}$ feltétel. A felsoroló objektum hány elemére teljesül a feltétel?

Specifikáció:

$A = (t{:}enor(E), c{:}\mathbb{N})$
$Ef = (t=t')$
$Uf = (c = \sum_{\substack{e \in t' \\ \beta(e)}} 1)$

Algoritmus:



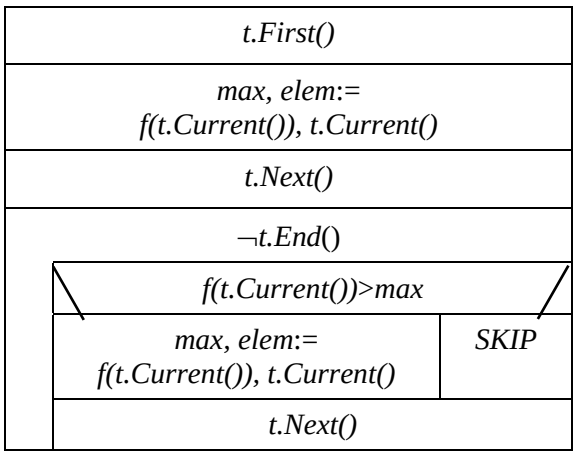
Maximum kiválasztás

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum és egy $f:E\rightarrow H$ függvény. A *H* halmazon definiáltunk egy teljes rendezési relációt. Feltesszük, hogy *t* nem üres. Hol veszi fel az *f* függvény a *t* elemein a maximális értékét?

Specifikáció:

$A = (t{:}enor(E), max{:}H, elem{:}E)$
$Ef = (t=t' \wedge |t|>0)$
$Uf = ((max, elem) = \mathbf{max}_{e \in t'} f(e))$

Algoritmus:



Kiválasztás

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum és egy β:$E \to \mathbb{L}$ feltétel. Keressük a *t* bejárása során az első olyan elemi értéket, amely kielégíti a β:$E \to \mathbb{L}$ feltételt, ha tudjuk, hogy biztosan van ilyen.

Specifikáció:

$A = (t{:}enor(E), elem{:}E)$

$Ef = (t = t' \wedge \exists i \in [1..|t|]: \beta(t_i))$

$Uf = ((elem, t) = \underset{elem \in t'}{\textbf{select}} \beta(elem))$

Algoritmus:

$t.First()$
$\neg\beta(t.Current())$
$t.Next()$
$elem{:=}t.Current()$

Lineáris keresés

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum és egy β:$E \to \mathbb{L}$ feltétel. Keressük a *t* bejárása során az első olyan elemi értéket, amely kielégíti a β:$E \to \mathbb{L}$ feltételt.

Specifikáció:

$A = (t{:}enor(E), l{:}\mathbb{L}, elem{:}E)$

$Ef = (t = t')$

$Uf = ((l, elem, t) = \underset{e \in t'}{\textbf{search}} \beta(e))$

Algoritmus:

$l := hamis; t.First()$
$\neg l \wedge \neg t.End()$
$elem := t.Current()$
$l := \beta(elem)$
$t.Next()$

Feltételes maximumkeresés

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum, egy β:$E \to \mathbb{L}$ feltétel és egy $f{:}E \to H$ függvény. A *H* halmazon definiáltunk egy teljes rendezési relációt. Határozzuk meg *t* azon elemeihez rendelt *f* szerinti értékek között a legnagyobbat, amelyek kielégítik a β feltételt.

Specifikáció:

$A = (t{:}enor(E), l{:}\mathbb{L}, max{:}H, elem{:}E)$

$Ef = (t = t')$

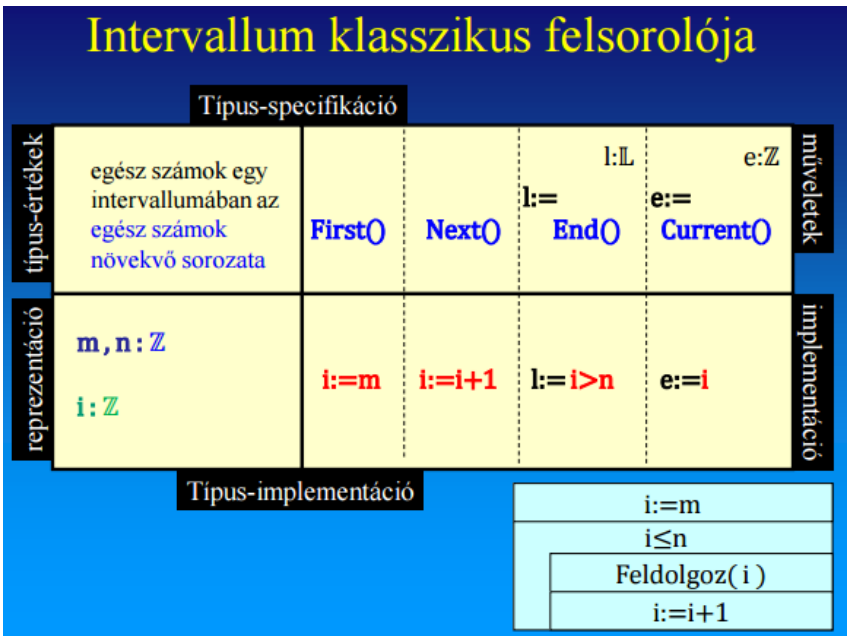
$Uf = ((l, max, elem) = \underset{\substack{e \in t' \\ \beta(e)}}{\textbf{max}} f(e))$

Algoritmus:

<table>
<tr><td colspan="4">$l{:=}\ hamis;\ t.First()$</td></tr>
<tr><td colspan="4">$\neg t.End()$</td></tr>
<tr><td>$\neg\beta(t.Current())$</td><td colspan="2">$\beta(t.Current()) \wedge l$</td><td>$\beta(\ t.Current()) \wedge \neg l$</td></tr>
<tr><td>$SKIP$</td><td colspan="2">$f(t.Current()) > max$</td><td rowspan="2">$l, max, elem :=$
$igaz, f(t.Current()), t.Current()$</td></tr>
<tr><td></td><td>$max, elem{:=}$
$f(t.Current()), t.Current()$</td><td>$SKIP$</td></tr>
<tr><td colspan="4">$t.Next()$</td></tr>
</table>

6 Nevezetes gyűjtemények felsorolói

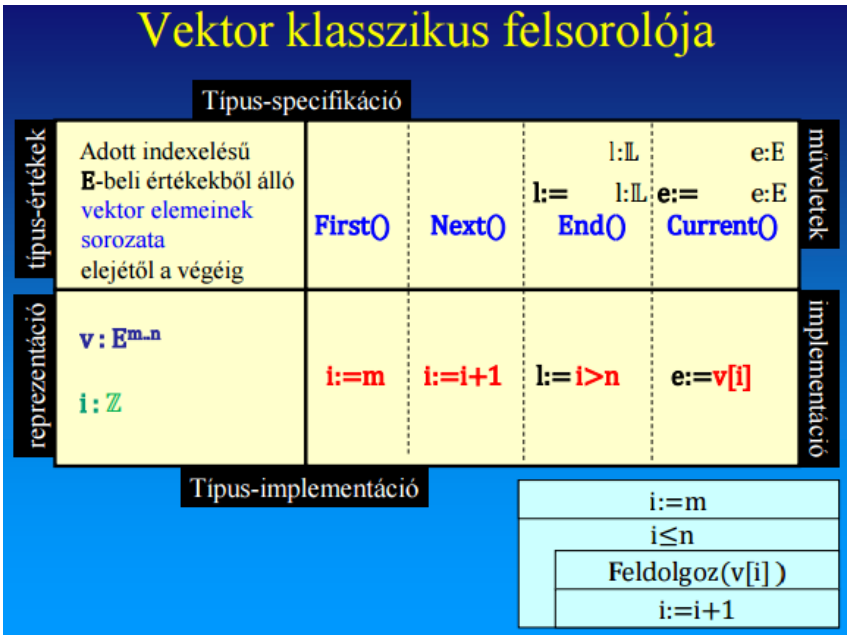
6.1 Intervallum



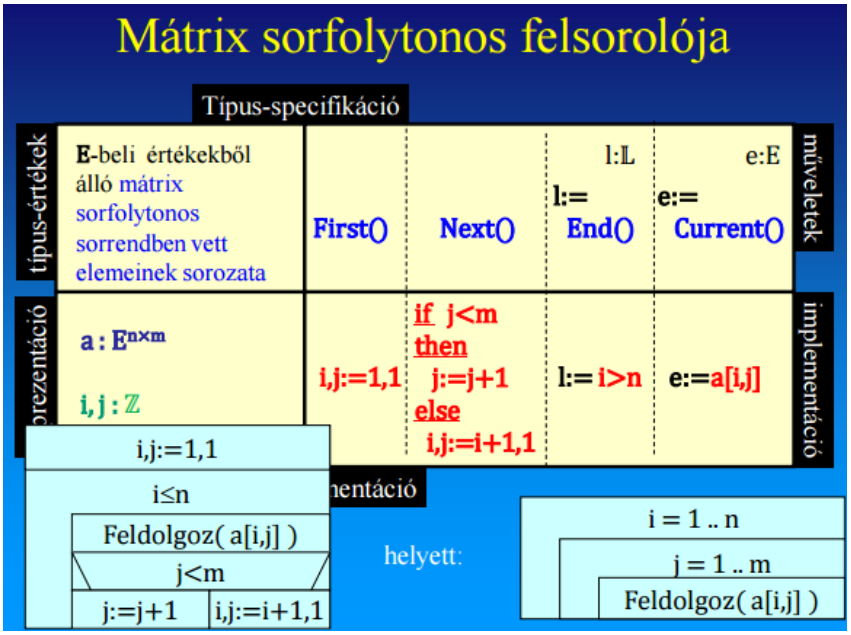
ábra 8: Intervallum felsorolója

6.2 Tömb

Itt két különböző tömbtípus felsorolóját mutatjuk be: az egydimenziós (vektor) és a kétdimenziós tömbét (mátrix).



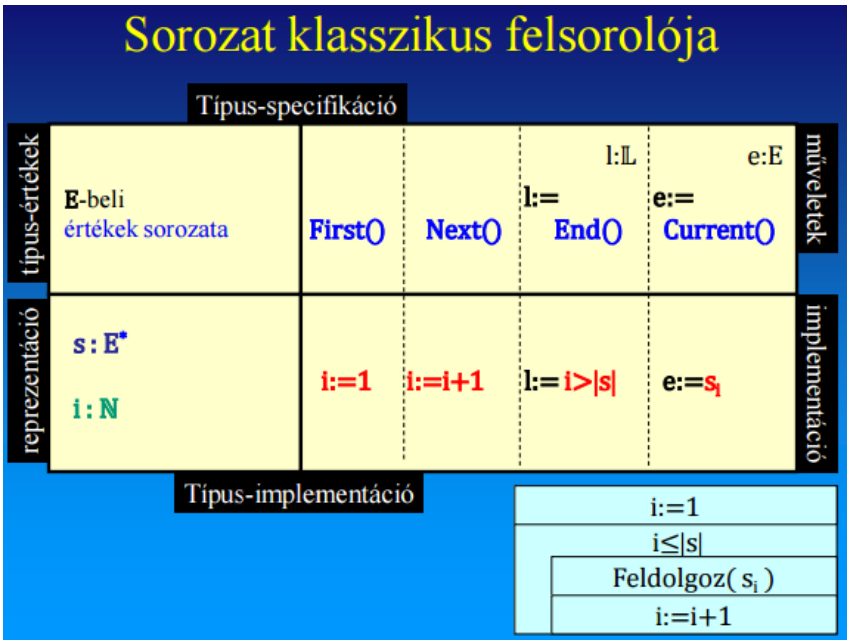
ábra 9: Vektor felsorolója



ábra 10: Mátrix sorfolytonos felsorolója

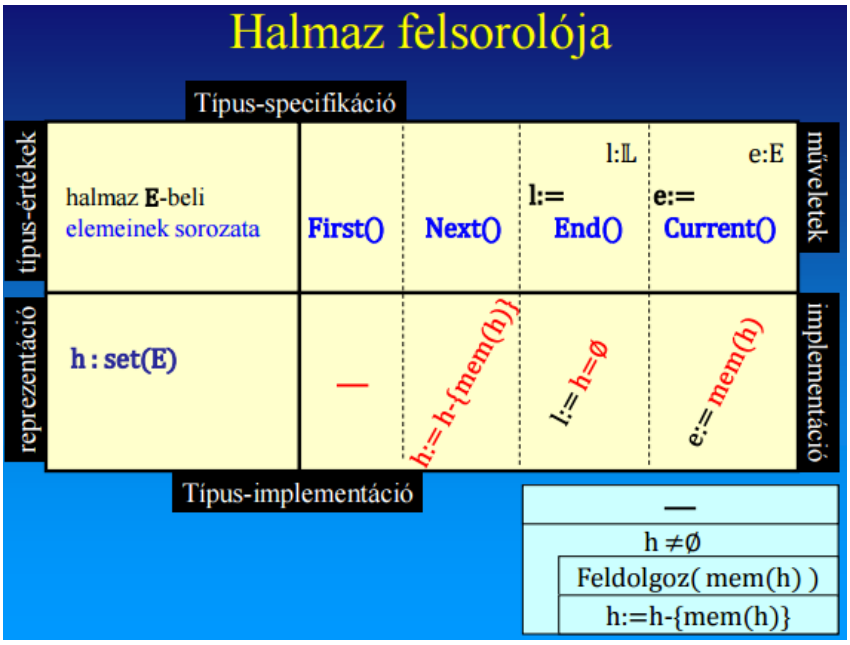
Megjegyzés: a felsorolás történhet másképpen is, például vektor esetén végezhetjük a felsorolást visszafelé, a tömb végétől kezdve, vagy mátrixnál alkalmazhatunk pl. oszlopfolytonos bejárást.

6.3 Sorozat



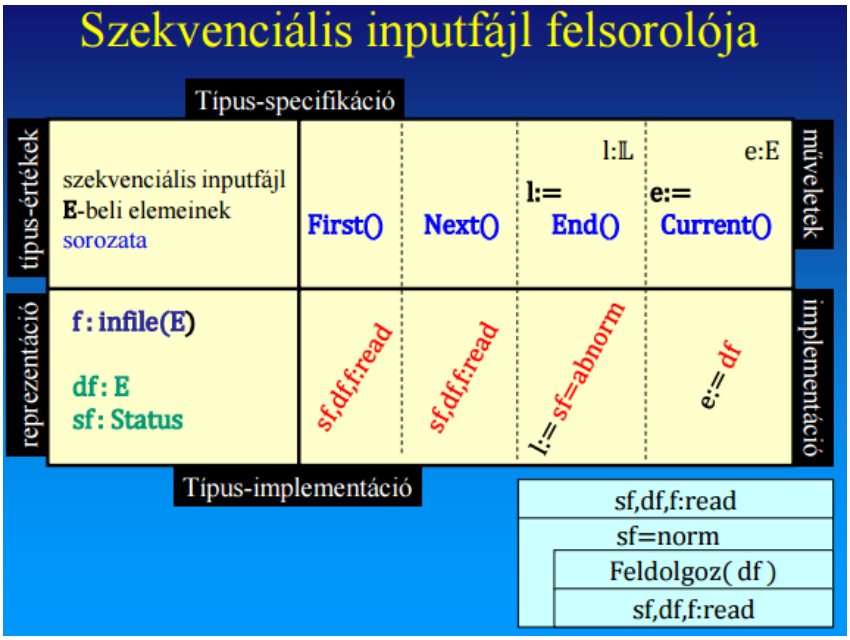
ábra 11: Sorozat felsorolója

6.4 Halmaz



ábra 12: Halmaz felsorolója

6.5 Szekvenciális inputfájl



ábra 13: Szekvenciális inputfájl felsorolója

Chapter 8

8

Záróvizsga tételsor
8. Programfejlesztési modellek

Dobreff András

Programfejlesztési modellek
Nagy rendszerek fejlesztési fázisai, kapcsolataik. Az objektumelvű modellezés nézetrendszerei. Statikus modell (osztálydiagram, objektumdiagram). Dinamikus modell (állapotdiagram, szekvenciadiagram, együttműködési diagram, tevékenységdiagram). Használati esetek diagramja.

1 Nagy rendszerek fejlesztési fázisai, kapcsolataik

1.1 Fejlesztési fázisok

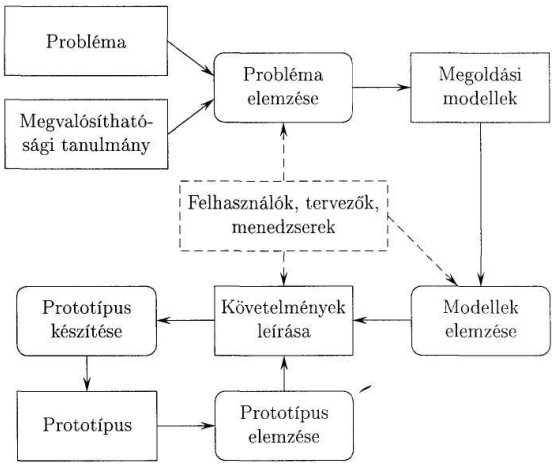
1. A probléma megoldásának előzménye

   Egy probléma megoldása előtt meg kell vizsgálni a megvalósíthatóságát, és annak mikéntjét. Eredmény: *Megvalósíthatósági tanulmány*, mely a következőkre válaszol:

   - Erőforrások (hardver, szoftver, szakember)
   - Költségek
   - Határidő
   - Üzemeltetés

2. Követelmények leírása

   Rendszerint iteratív módon állítjuk elő, és a prototípust használjuk a finomításra (ábra 1).



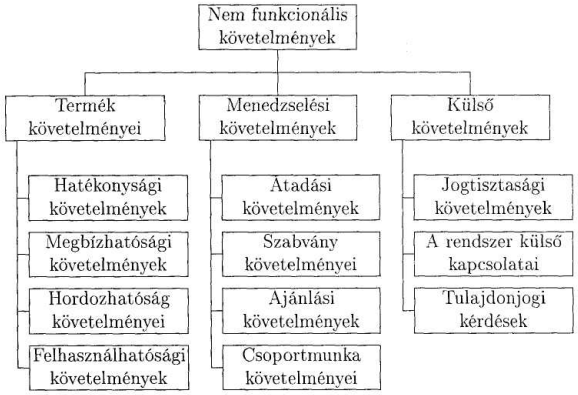
ábra 1: Követelményleírás elkészítésének folyamata

Követelmények leírásának tartalma:

- Probléma
- Korlátozó tényezők (hardver, szoftver, stb.)
- Elfogadható megoldás

Követelmények leírásának fajtái:

- Funkcionális követelmények
  A rendszer szolgáltatásainak, leképzéseinek leírása:
  - Elindítás formája
  - Bemenő adatok (és azok megadásának formája)
  - Igénybevétel előfeltétele, korlátozások
  - Szolgáltatás kezdeményezésére a válasz, eredmények
  - Válasz megjelenési formája
  - Bemenő adatok és válasz közti reláció
- Nem funkcionális követelmények
  A nem funkcionális követelményeket rendszerint három osztályba soroljuk: a termék követelményei, menedzselési követelmények, külső követelmények. Az osztályokat tovább lehet bontani (ábra 2).



ábra 2: Nem funkcionális követelmények osztályozása

3. Követelmények elemzése és prototípus

   A következőket kell megvizsgálni:

   - Önmagában jó-e a követelmények leírása?
     - Konzisztens (nincs ellentmondás)
     - Komplett (teljes)
   - Validáció vizsgálat
     (Megfelel-e a felhasználó által elképzelt problémának?)
   - Megvalósíthatósági vizsgálat
     (A követelményeknek megfelelő megoldás megvalósítható-e?)
   - Tesztelhetőségi vizsgálat
     (A követelmények úgy vannak-e megfogalmazva, hogy azok tesztelhetők?)
   - Nyíltság kritériumainak vizsgálata.
     (A követelmények nem mondanak-e ellent a módosíthatóság, a továbbfejleszthetőség követelményének?)

   A követelmények elemzésének egyik eszköze a prototípus-készítés. A prototípus magas szintű programozási környezetben létrehozott, a külső viselkedés szempontjából helyes megoldása a problémának.

4. Programspecifikáció

   A programspecifikáció a következő kérdésekre kell, hogy válaszoljon a követelmények leírása alapján:

   - Mik a bemenő adatok? (Forma, jelentés, megjelenés.)
   - Mik az eredmények? (Forma, jelentés, megjelenés.)
   - Mi a reláció a bemenő adatok és az eredmény adatok között?

5. Tervezés

   A tervezés során a következő kérdésekre adjuk meg a választ:

   (a) Statikus modell
   - Rendszer szerkezete
   - Programegységek, azok feladata és kapcsolata

   (b) Dinamikus modell
   - Hogyan oldja meg a rendszer a problémát?
   - Milyen egységek működnek együtt?
   - Milyen üzenetek játszódnak le?
   - Rendszer és egységek állapotai
   - Események (melyek hatására állapotváltás történik)

   (c) Funkcionális modell
   - Milyen adatáramlások révén valósulnak meg a szolgáltatások?
   - Milyen leképezések játszanak szerepet az adatáramlásokban?
   - Mik az ajánlások az implementáció számára?
     - Implementációs stratégiára vonatkozó ajánlás.
     - Programozási nyelvre vonatkozó előírás, ajánlás.
     - Tesztelési stratégiára vonatkozó ajánlás.

   A gyakorlatban két tervezési módszer terjedt el: *procedurális* és a *objektumelvű*

   (*procedurális*: megvalósítandó funkciókból, műveletekből indulunk ki, és ezek alapján bontjuk fel a rendszert kisebb összetevőkre, modulokra
   objektumelvű: a rendszer funkciói helyett az adatokat állítjuk a tervezés középpontjába. A rendszer által használt adatok felelnek meg majd bizonyos értelemben az objektumoknak.)

6. Implementáció

   Fontos szempontok:

   - Reprezentáció (Adatok ábrázolása)
   - Események leképezések megvalósítása
     Algoritmusok és optimalizálások

   Az implementáció egyik alapvető kérdése az implementációs stílus. A jó programozási stílus néhány fontos eleme:

   - absztrakció különböző szintjeinek alkalmazása
   - öröklődési technika használata, absztrakciós szintek hierarchikus rendszere
   - absztrakciós szintekre bontás osztályon belül (deklaráció + megvalósítás)
   - korlátolt láthatóság;
   - információ elrejtés (information hiding);
   - információ beburkolás (encapsulation).

7. Verifikáció, validáció

   A rendszer eleget tesz-e a vele szemben támasztott elvárásoknak?

   Verifikáció: a specifikációszerinti helyesség igazolása

   Validáció: Minőségi előírások teljesítése (robosztusság hatékonyság, erőforrásigény)

   Ennek folyamata: tesztelés, melynek szakaszai:

   - Egységteszt
   - Rendszerteszt

   A tesztelésnek két módja lehet:

- fekete doboz - Csak a maguknak a hibáknak a felderítése
- fehér doboz - Hibák helyének felderítése

8. Rendszerkövetés és karbantartás (maintenance)

   Karbantartás: Üzemebe helyezés után szükségessé váló szoftver jellegű munkák [pl.: rejtett hibák kijavítása, adaptációs munkák (új harver-, szoftverkörnyezet), továbbfejlesztési munkák]

   Rendszerkövetés: a felhasználókkal való kapcsolattartás menedzsment jellegű, dokumentációs feladatai [pl.: konfigurációk nyilvántartása, verziók menedzselése, dokumentáció menedzselése]

9. Dokumentáció

   Egy nagy méretű program önmagában nem tekinthető szoftverterméknek dokumentáció nélkül. Egy jó dokumentáció a következőképp épül fel.

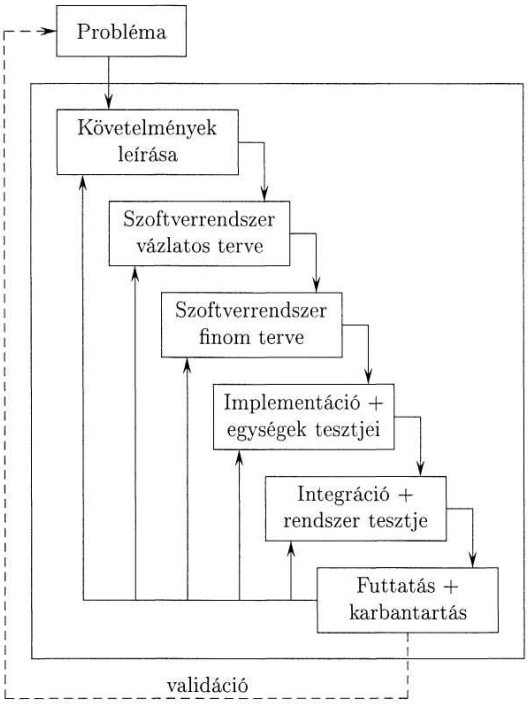
   - Felhasználói leírás
     - Feladatleírás
     - Futtató környezet
     - Fejlesztések, verziók
     - Installálás
     - Használat
     - Készítők
   - Fejlesztői leírás
     - Modulok (és azok szerkezete)
     - Osztályok (és azok kapcsolata)
     - Rendszer dinamikus viselkedése
     - Osztályok implementálása (adatszerkezetek, sablon osztályok)
     - Tesztelés

1.2 Fejlesztési fázisok kapcsolatai

A fejlesztési fázisok leírására többféle modellt használhatunk

1. Vízesés modell

   Az egyes fázisok egymást követik, a módosítások a futtatási eredmények ismeretében történnek. Egy bizonyos fázisban elvégzett módosítás az összes rákövetkező fázist is érinti.



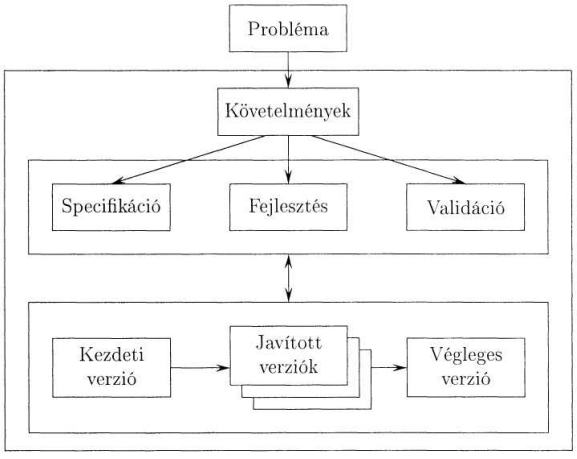
ábra 3: Vízesés modell

Hátrányai:

- Új szolgáltatás minden fázison módosítást igényel
- Validáció az egész életciklus megismétlését követelheti meg

2. Evolúciós modell

   A megoldást közelítő verzióinak, prototípusainak sorozatát állítjuk egymás után elő, és így haladunk lépésenként egészen a végleges megoldásig. Ennek során egy verzió elkészítésekor a specifikáció, a fejlesztés és a validáció párhuzamosan történik.



ábra 4: Evolúciós modell

Hátrányai:

- Nehéz a projekt áttekintése
- A gyors fejlesztés rendszerint a dokumentáltság rovására megy.

3. Boehm-féle spirális modell

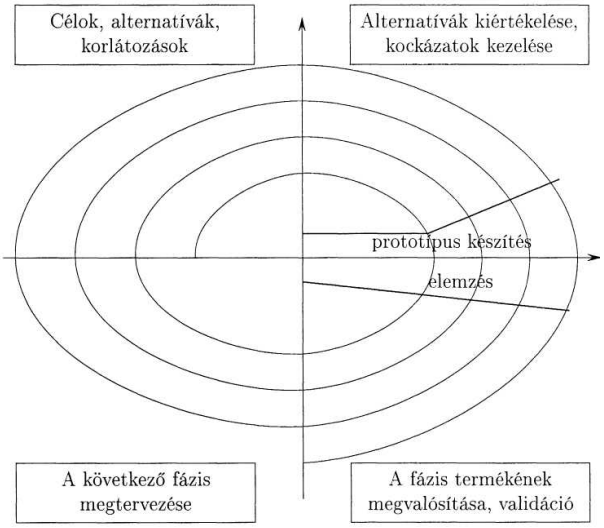
   Ez a modell egy iterációs modell. Az iteráció a spirális egy fázisával modellezhető, amely négy szakaszra bontható:

   (a) Célok, utak, alternatívák, korlátozások definiálása

   (b) Kockázatelemzés, stratégia kidolgozás

   (c) Feladat megoldása, validáció

   (d) Következő iteráció megtervezése



ábra 5: Evolúciós modell

Hátrányai:

- A modell alkalmazása általában munkaigényes, bonyolult feladat.
- A projekt kidolgozásához szükséges szakembereket nem könnyű gazdaságosan foglalkoztatni.

2 Az objektumelvű modellezés nézetrendszerei

Objektumelvű programozás = adatabsztrakció + absztrakt adattípus + típusöröklődés

- Absztrakció

  Programozás adott szintjén a megoldás szempontjából lényegtelen részletek elhanyagolása.

- Adattípus

  Az adattípus egy (A, F) rendezett pár, ahol A az adatok halmaza F pedig a műveletek véges halmaza. $(\forall f \in F : f : A^n \to A)$ Létezik egyszerű és összetett adattípus.
  Típusosztály: A típus komplex leírása, mely az adott adattípus absztrakt (PAR + EXP) és konkrét (IMP + BODY) leírását szolgálja. Tehát:

  Típusosztály = (PAR, EXP, IMP, BODY), ahol:
  PAR = <paraméterek tulajdonságai >
  EXP = <típusobjektumok halmaza és műveltei neve, szintaktikája, szemantikája >
  IMP = <más osztályból átvett szolgáltatások >
  BODY = <típusosztály ábrázolása, megvalósítása >

- Típusöröklődés

  A típusöröklődés két fő formája: *specializáció* és *újrafelhasználás*

1. A subclass átveszi az absztakt tulajdonságokat és azt az export részben használja fel
2. Típushalmaz, paraméterhalmazok, műveletek nevei átdefiniálódhatnak.
3. subclass típushalmaza = superclass típushalmaza

A specializáció következményei:

- polimorfizmus
  Minden változónak két típusa van: *statikus* (deklaráció során kapott) és *dinamikus* (deklaráció pillanatában megegyezik a statikussal, de később megváltozhat, ha egy superclass példánynak adunk értékül egy subclass példányt)
- dinamikus kötés
  A dinamikus típusnak megfelelő kiszámítási szabály hozzárendelése a függvényhez attribútumhoz, a végrehajtás pillanatában

Nézetrendszerek

- használati szempont
  Kinek nyújt a rendszer szolgáltatást? (Személyek vagy más rendszerek, programok)
- Szerkezetei strukturális, statikus szempont
  Milyen egységek vannak, ezeknek mi a feladata, és hogyan kapcsolódnak egymáshoz?
- Dinamikus szempont
  Az egyes részegységek hogyan viselkednek, milyen állapotokat vesznek fel, azokat milyen események hatására váltják? Milyen az egységek között együttműködés mechanizmusa? Időben hogyan játszódnak le közöttük az üzentek?
- Implementációs szempont
  Milyen szoftverkomponensek, és azok között milyen kapcsolatok vannak?
- Környezeti szempont
  A rendszer milyen hardver és szoftver erőforrást igényel a megoldás során?

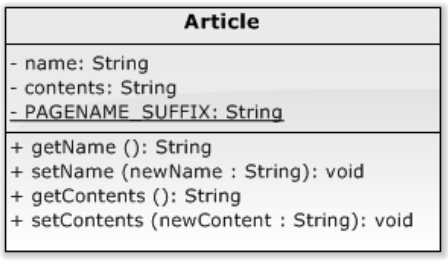
3 Statikus modell (osztálydiagram, objektumdiagram)

3.1 Osztáldiagram

A megoldás szerkezetét leíró összefüggő gráf, melynek csomópontjaihoz az osztályokat, éleihez pedig az osztályok közötti relációkat (öröklődés, asszociáció, aggregáció, kompozíció) rendeljük.
A rendszerhez csak egy osztálydiagram tartozik.

Osztályok

Egy osztály a következőképp néz ki:



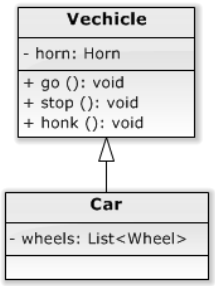
ábra 6: Osztály

Az osztályt leíró téglalap 3 részre van osztva.

- Az első részbe az osztály neve kerül.
  Ha az osztály absztrakt, a nevét dőlt betűvel írjuk.
- A második részbe az osztály attribútumai kerülnek.
  Az attribútum formátuma: *Attribútumnév : Típus* A statikusságot aláhúzással jelöljük. Az attribútumok láthatóságát is fel lehet tüntetni: *publikus (+), privát (-), védett (#)*
- A harmadik részbe az osztály metódusai kerülnek.
  A metódusok formátuma: *Metódusnév(Paraméterlista):Visszatérési érték*, ahol a paraméterlista *Paraméternév:Típus* fomrátumú paraméterekből áll. Absztraktságot, statikusságot, és láthatóságot az előzőekben leírtakkal azonosan jelöljük.

Osztályok közötti kapcsolatok:

- öröklődés
  Két osztály közötti absztrakciós kapcsolatot jelöl



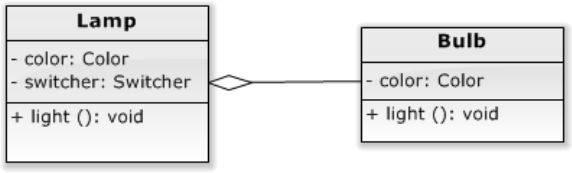
ábra 7: Öröklődés

- asszociáció
  Ez a legáltalánosabb reláció két osztály között. Az asszociáció két osztály közötti absztrakt reláció, amely kétirányú társítást fejez ki. A reláció absztrakt volta azt jelenti, hogy a reláció konkretizálása osztályok objektumainak összekapcsolásában valósul meg.
  Az asszociációnak lehet:
  - Neve, azonosítója
  - Iránya
  - Multiplicitása
    Akár egy érték, akár intervallum. A * szimbólum kitüntett szerepet kap, jelentése: bármennyi, akár nulla is. (Pl: 3, 1..4, 5..*, *)
  - Szerepe
  - Navigálhatósága
    A társított osztályok közül csak az egyik ismeri a másikat. (Ha nem tüntetjük fel, kölcsönös elérhetőséget feltételezünk)

ábra 8: Asszociáció

- aggregáció
  Az aggregáció egy speciális asszociáció, mely egész-rész kapcsolatot fejez ki. Azonban ha két osztály között aggregációs reláció áll fenn, a két osztály objektumai egymástól függetlenül is létezhetnek (Ezt un. laza tartalmazási relációnak nevezik) A relációt jellemzi ezen kívül

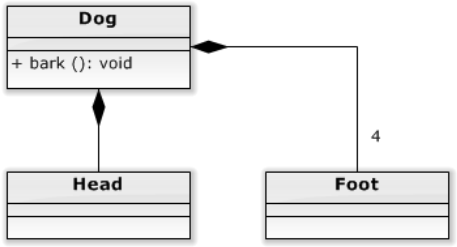
a tranzitivitás, és a közös attribútumok illetve szolgáltatások. Különböző aggregátumoknak lehetnek közös komponenseik.



ábra 9: Aggregáció

- kompozíció

  A kompozíció egy speciális aggregáció, mely *fizikai* tartalmazást jelöl. Nem jellemzi többé az objektumok független létezése, a két objektum egyszerre jön létre és szűnik meg. Tehát a tartalmazó objektumnak gondoskodnia kell a tartalmazott létrehozásáról és megszüntetéséről. Egy komponens legfeljebb egy tartalmazó objektumhoz tartozhat. A kompozíciós kapcsolat és az attribútum jellegű kapcsolat két objektum között szemantikailag azonos.



ábra 10: Kompozíció

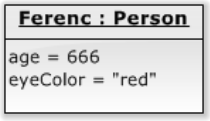
3.2 Objektumdiagram

Az objektumdiagram egyszeresen összefüggő gráf, amelynek csomópontjaihoz az objektumokat, éleihez pedig az objektumok közötti összekapcsolásokat rendeljük.

A rendszerhez különböző időpillanatokban más-más objektumdiagram tartozhat. (Viszont mindegyiknek meg kell felelnie az osztálydiagramnak)

Objektumok

Az objektumokat az osztályokhoz hasonlóan egy téglalap írja le. Egy ilyen téglalapnak két része van. Az első részben az objektum neve (opcionális) és típusa található a következő formátumban: *Objektumnév : Típus*, melyet aláhúzással tarkítunk. A második részbe az objektum attribútumai és azok értékei kerülhetnek, a következő formátumban: *Attribútumnév=érték*.

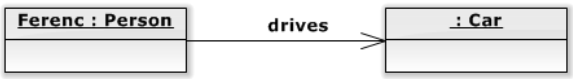


ábra 11: Objektum

Objektumok közötti kapcsolatok

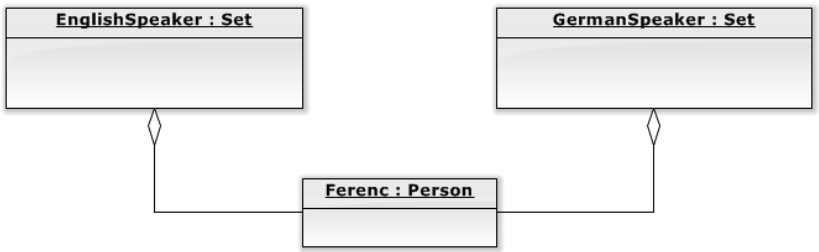
Az objektumokat összekötő relációk az osztálydiagramon lévőkkel megegyezőek (Öröklődésnek ezen a szinten nincs értelme):

- asszociáció



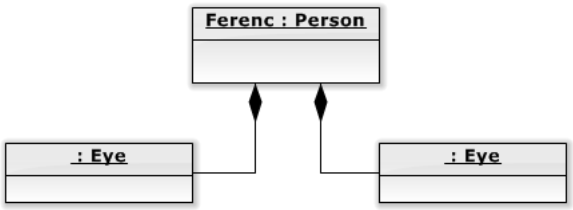
ábra 12: Asszociáció

- aggregáció



ábra 13: Aggregáció

- kompozíció



ábra 14: Aggregáció

4 Dinamikus modell (állapotdiagram, szekvenciadiagram, együttműködési diagram, tevékenységdiagram)

4.1 Állapotdiagram

Az állapotdiagram egy összefüggő irányított gráf, amelynek csomópontjaihoz az állapotokat rendeljük, éleihez pedig az eseményeket. (Két csúcs között több állapotátmenetet is jelölhetünk, hiszen több esemény hatására is létrejöhet)

Állapot

Az objektum állapotát az attribútumok konkrét értékeinek n-esével jellemezzük.

Az állapotnak van azonosítója, mely legtöbbször az állapot neve (de lehet maga az invariáns, vagy az attribútumok konkrét értéke). Az állapotot esemény hozza létre és szünteti meg. Az állapot mindaddig fennmarad, míg az attribútumok kielégítik az állapotot leíró invariánst. Az állapotot egy lekerekített téglalappal jelöljük, melyben az azonosítót tüntetjük fel.

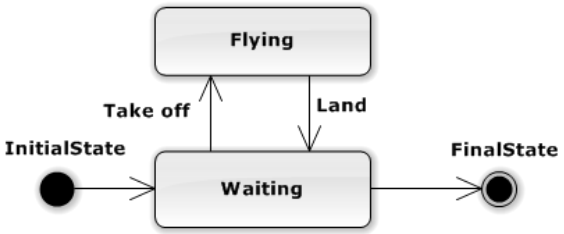
Speciális (rendszeren kívüli) állapotok: Kezdőállapot, Végállapot

Esemény

Eseménynek nevezzük azt a tevékenységet, történést, amely valamely objektum állapotát megváltoztatja.

Az esemény lehet paraméteres vagy paraméter nélküli, és lehet előfeltétele. Az események között sorrendiség áll fent, így egy esemény lehet megelőző eseménye. Egy eseményt a következőképp írhatunk le:
<esemény >(<paraméterek>)[<feltétel>]/<megelőző esemény>

Az eseményeket az állapotok közötti állapotátmenetekre írjuk.



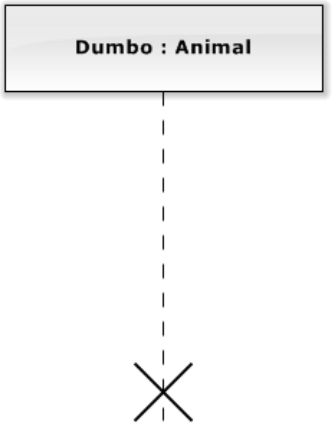
ábra 15: Repülőgép állapotgépe

4.2 Szekvenciadiagram

A szekvencia diagram az objektumok közötti üzenetváltások időbeli menetét szemlélteti.

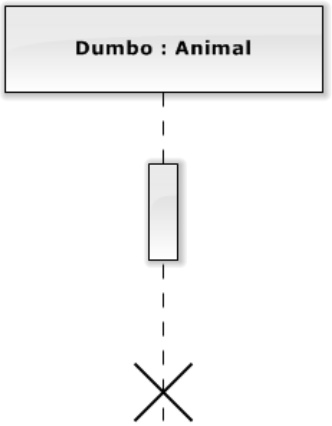
Osztályszerep
Az osztály szerepét olyan egy vagy több objektum testesíti meg, melyek az üzenetküldés szempontjából konform módon viselkednek.

Osztályszerep életvonal
Az életvonal az osztályszerep időben való létezését jelenti.



ábra 16: Életvonal

Aktivációs életvonal
Az aktivációs életvonal azt az állapotot jelenti, amikor az osztályszerep megtestesítői műveleteket hajtanak végre, vagy más objektumok vezérlése alatt állnak.



ábra 17: Aktivációs életvonal

Üzenet

Az üzenet az objektumok közötti információátadás formája. Az üzenet küldésének az a célja, hogy az objektum működésbe hozza a másik objektumot. Az üzenet azok között az objektumok között jöhet létre, amelyek az objektumdiagramban kapcsolatban állnak. Az üzenetnek van azonosítója (neve, szövege), lehet paramétere, sorszáma.

4.3 Együttműködési diagram

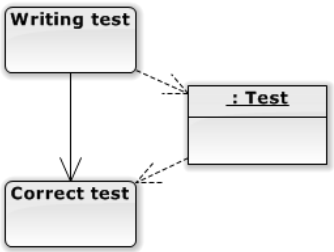
Az együttműködési diagram azt hivatott bemutatni, hogy miként működnek együtt az osztályok objektumai, milyen üzenetek cseréje révén valósul meg ez az együttműködés. (Csak azok az objektumok relevánsak, amelyek osztályait az osztálydiagramban asszociációs kapcsolat köt össze. A diagram mutatja ezt az összekapcsolást és az ehhez tartozó üzenetváltásokat, ezért az együttműködési diagram az objektumdiagram bizonyos értelemben vett kiterjesztésének tekinthető.)

Az üzenet küldését egy nyíl mutatja, amely az asszociáció mellett kap helyet és a címzett irányába mutat. Az üzenet azonosítója a nyíl mentén helyezkedik el. Az üzenetnek lehet argumentuma és eredménye. Ezeket egy kis körből induló nyíl mellett helyezzük el, ahol a nyíl az információ áramlásának irányát mutatja.

4.4 Tevékenységdiagram

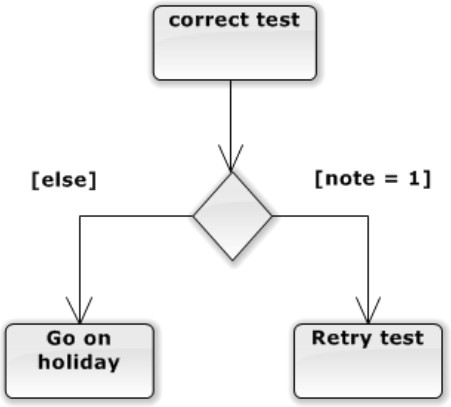
A tevékenységdiagram (aktivációs diagram) a probléma megoldásának lépéseit szemlélteti, a párhuzamosan zajló vezérlési folyamatokkal együtt.

Ha egy tevékenységet egy másik tevékenység követ közvetlenül, akkor a két tevékenységet nyíllal kötjük össze. Ha adatot (objektumot) ad át egy tevékenység egy másik tevékenységnek, akkor a küldő tevékenységből szaggatott nyíl vezet az objektumot reprezentáló téglalaphoz, és a téglalaptól szaggatott nyíl mutat a fogadó tevékenységre. A téglalapban szögletes zárójelek között megadhatjuk az objektum állapotát, státuszát is.



ábra 18: Objektum átadás

Lehetőség van arra, hogy bizonyos feltételek teljesülése esetén eltérő tevékenységeket hajtsunk végre, illetve tevékenységek végrehajtását feltételekhez kössük. Ekkor egy rombuszt kell elhelyeznünk a diagramban, amelyből kivezető nyilakra írjuk a feltételeket.



ábra 19: Feltétel ábrázolása

5 Használati esetek diagramja

A használati esetek diagramja a felhasználók szempontjából kívánja szemléltetni azt, hogy a rendszer miként működik, függetlenül attól, hogy a szolgáltatásait hogyan valósítja meg.
A diagram részei:

- használati esetek
- felhasználók
- felhasználási relációk

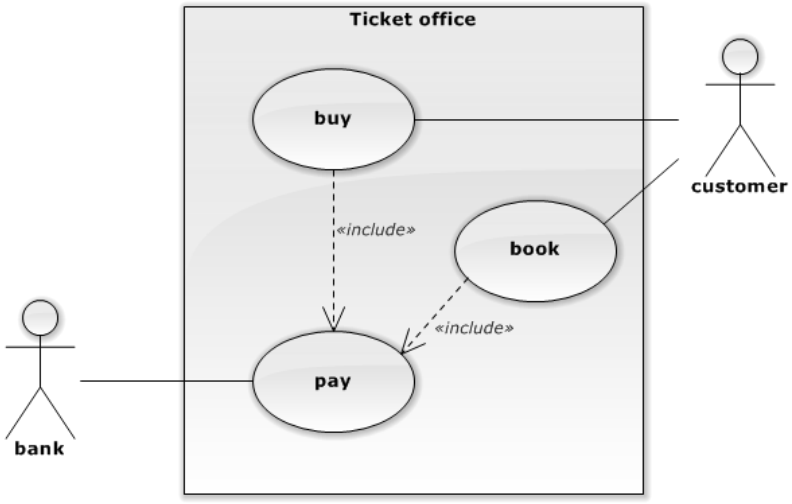
A használati esetek a rendszer funkcióinak összefoglalásai, szolgáltatási egységek. Ez az egység az akcióknak egy olyan sorozata, amelyekkel a rendszer a felhasználók egy csoportjával működik együtt.

A használati esetet egy ovális alakzattal jelöljük. A használati eseteket téglalapba foglaljuk, ez jelzi a rendszer határait.

A felhasználók az adott rendszeren kívüli egységek, más programrendszerek, alrendszerek, osztályok, illetve személyek lehetnek. Ezek aktor szerepet töltenek be. A diagramon egy pálcikaember figurával jelöljük.

A felhasználási relációk kapcsolják össze a használati eseteket a felhasználókkal. A relációk egymással is kapcsolatban állhatnak, amit a diagramban fel lehet tüntetni. A lehetséges relációk a következők:

- asszociáció
  Egy felhasználó és egy használati eset közötti kapcsolatot jelez. (Egyszerű vonal)
- általánosítás
  Az egyik használati eset a másik általánosabb formája. (Egyszerű vonal, végén fehér háromszöggel)
- kiterjesztés
  Az egyik használati eset a másikat terjeszti ki. Ennek során viselkedéseket illeszt be megadott beszúrási pontoknál. (Szaggatott nyíl ≪extend≫ felirattal)
- tartalmazás
  Az egyik használati eset tartalmazza a másik viselkedését (Szaggatott nyíl ≪include≫ felirattal)



ábra 20: Használati esetek diagramja

Chapter 9

9

Záróvizsga tételsor

9. Programok fordítása és végrehajtása

Programok fordítása és végrehajtása

Fordítás és interpretálás összehasonlítása. Fordítási egység és a szerkesztés fogalma. Fordítóprogramok komponenseinek feladata és működési elveik vázlatos ismertetése. Kódgenerálás assemblyben alapvető imperatív vezérlési szerkezetekhez. A szekvenciális és párhuzamos/elosztott végrehajtás összehasonlítása

1 Bevezetés

Amikor programot írunk, azt valamilyen programozási nyelven tesszük. Ezután a nyelvtől függően vagy lefordítjuk, vagy interpreterrel futtatjuk.

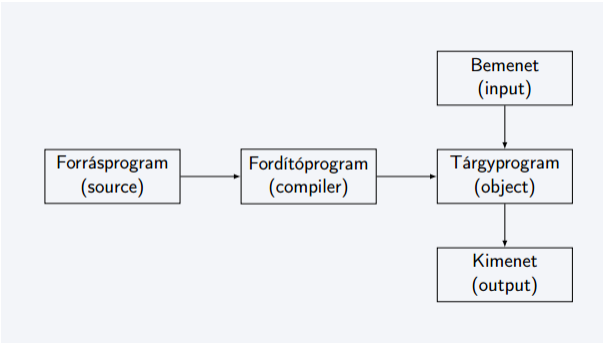
2 Fordítás és Interpretálás

2.1 Fordítás

A fordítás során általában egy magas szintű programozási nyelvből gépi kód keletkezik, amelyet a processzor már képes értelmezni és futtatni. Előnye, hogy gyors, mivel a lexikális, szintaktikus és szemantikus elemzés fordítási időben, egyszer fut le, valamint ekkor optimalizáljuk a kódot. Fordítási időben sok hibát ki lehet szűrni, ezáltal megkönnyítve a debugolást. A gépi kód nehezen visszafejthető. Általában nagyobb programokhoz használjuk, ahol fontos a hatékonyság. A lefordított kódon később már nem (vagy csak nagyon nehezen) tudunk változtatni.

Hátránya, hogy a keletkezett kód nem platformfüggetlen, minden architektúrára külön-külön le kell fordítani.

Példák: C, C++, Ada, Haskell

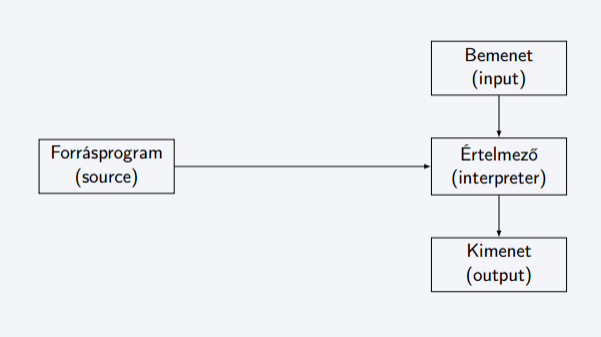


ábra 1: a fordítás folyamata

2.2 Interpretálás

Az interpretálás során a programkódot az értelmező futás közben hajtja végre. Platformfüggetlen, csak az interpretert kell minden rendszerre egyszer megírni. Nehéz benne a hibakeresés, mivel sok olyan hiba maradhat a kódban, amit egy fordító kiszűrt volna (pl. típus egyezőség).

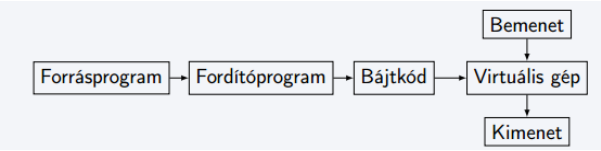
Példák: PHP, JavaScript, ShellScript



ábra 2: az interpretálás folyamata

2.3 Fordítás és Interpretálás együtt

Egyes nyelvek (pl. Java) előfordítást használnak, melynek eredménye a *bájtkód*, amely gépi kód egy virtuális gép számára. Ezzel elérhető a fordítási idejű hibaellenőrzés és optimalizálás, de megmarad a platformfüggetlenség.



ábra 3: az interpretálás folyamata

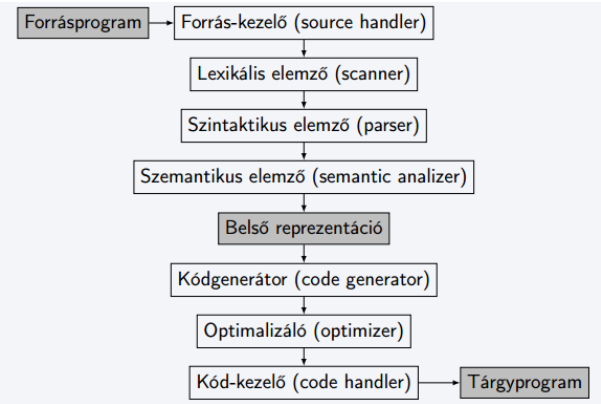
3 Fordítási egység és a szerkesztés

A tárgykód létrehozása két fázisban történik. Először a forrásfájlokat *lefordítjuk*, ebből keletkezik az un. *objektumkód* (pl.: .obj, .class). Ebben a gépi utasítások már megvannak, de hiányzik belőle a hivatkozások (pl változók, függvények), melyek más fájlokban vannak megvalósítva. *Fordítási egységnek* nevezzük azt, amiből egy objektumkód keletkezik.

A *linker* (szerkesztő) feladata, hogy a hiányzó referenciákat kitöltse, hogy egyetlen fájlt generálva futtatható kódot kapjunk.

A linkelés lehet statikus, amikor a fordító tölti fel a hiányzó referenciákat; vagy dinamikus, mikor fordítási időben, jellemzően egy másik fájlból (pl.: .dll) tölti be a hiányzó kódot. Az utóbbi akkor praktikus, ha egy modult több, különálló program használ.

4 A fordítóprogram komponensei



ábra 4: a fordítás lépései

4.1 Lexikális elemző

Bemenete maga a forráskód. A lexikális elemző feladata, hogy tokenekre bontsa a forráskódot. Adott egy reguláris (hármas típusú) nyelvtan, mely a nyelvre jellemző. Ez adja meg, hogy milyen típusú tokenek szerepelhetnek a forrásban. A tokenekhez tulajdonságokat rendelhet (pl. változó neve, literál értéke). Kimenete ez a tokensorozat. Amennyiben az elemző olyan karaktersorozatot talál, amelynek nem feleltethető meg token, akkor az lexikális hibát vált ki.

megjegyzés: Lexikális hibánál nem feltétlen szakad meg a fordítás folyamata, megpróbálhatjuk átugrani az adott részt és folytatni az elemzést, így ha több hiba is van, akkor azokat egyszerre jelezhetjük.

A reguláris kifejezéseket *véges determinisztikus automatákkal* ismerjük fel. Amennyiben egy lexikális elemre az egyik automata elfogadó állapotba kerül, úgy felismertünk egy tokent. Egy karaktersorozatot egyszerre több automata is felismerhet. Amennyiben ezek azonosan hosszúak, akkor a nyelv konfliktusos. Ennek nem szabad előfordulnia. Az viszont lehetséges, hogy egy szót, és az ő prefixét is felismerte egy automata. Ekkor mindig a hosszabbat választjuk.

4.2 Szintaktikus elemző

Bemenete a lexikális elemző kimenete. Feladata, hogy *szintaxisfát* építsen a tokenekből, a nyelvez tartozó egy környezetfüggetlen (kettes típusú) grammatika alapján, vagy ha ez lehetetlen, akkor jelezze ezt *szintaktikus hiba*ként.

4.2.1 LR0 elemzés

A lexikális elemző által előállított szimbólumsorozatot balról jobbra olvassuk, a szimbólumokat az elemző vermébe tesszük.

Léptetés: egy új szimbólumot teszünk a bemenetről a verem tetejére.

Redukálás: a verem tetején lévő szabály-jobboldalt helyettesítjük a szabály bal oldalán álló nemterminálissal

A háttérben egy véges determinisztikus automata működik: az automata átmeneteit a verem tetejére kerülő szimbólumok határozzák meg ha az automata végállapotba jut, redukálni kell egyéb állapotban pedig léptetni.

Az automata bizonyos nyelvek esetén konfliktusos lehet: nem tudjuk eldönteni, hogy léptessünk vagy redukáljunk.

4.2.2 LR1 elemzés

Az előző problémára kínál megoldást, kibővítve a lehetséges nyelvek halmazát.

Az ötlet, hogy *olvassunk előre* egy szimbólumot.

Ha az aktuális állapot i, és az előreolvasás eredménye az a szimbólum:

ha $[A \to \alpha.a\beta, b] \in I_i$ és $read(I_i, a) = I_j$ akkor léptetni kell, és átlépni a j állapotba.

ha $[A \to \alpha., a] \in I_i (A \neq S')$, akkor redukálni kell az $A \to \alpha$ szabály szerint.

ha $[S' \to S., \#] \in I_i$ és $a = \#$, akkor el kell fogadni a szöveget, minden más esetben hibát kell jelezni.

Ha az i állapotban A kerül a verem tetejére: ha$read(I_i, A) = I_j$, akkor át kell lépni a j állapotba, egyébként hibát kell jelezni.

4.2.3 Jelmagyarázat/Kanonikus halmazok

Closure/lezárás Ha I a grammatika egy $LR(1)$ elemhalmaza, akkor $closure(I)$ a legszűkebb olyan halmaz, amely az alábbi tulajdonságokkal rendelkezik:

$I \subseteq closure(I)$ ha $[A \to \alpha.B\gamma, a] \in closure(I)$,

és $B \to \beta$ a grammatika egy szabálya, akkor $\forall b \in FIRST1(\gamma a)$ esetén $[B \to .\beta, b] \in closure(I)$

Read/olvasás Ha I a grammatika egy $LR(1)$ elemhalmaza, X pedig terminális vagy nemterminális szimbóluma, akkor $read(I, X)$ a legszűkebb olyan halmaz, amely az alábbi tulajdonsággal rendelkezik:

Ha $[A \to \alpha.X\beta, a] \in I$, akkor $closure([A \to \alpha X.\beta, a]) \subseteq read(I, X)$.

LR(1) kanonikus halmazok (I_n)

- $closure([S' \to .S, \#])$ a grammatika egy kanonikus halmaza.
- Ha I a grammatika egy kanonikus elemhalmaza, X egy terminális vagy nemterminális szimbóluma, és $read(I, X)$ nem üres, akkor $read(I, X)$ is a grammatika egy kanonikus halmaza.
- Az első két szabállyal az összes kanonikus halmaz előáll.

4.3 Szemantikus elemző

A szemantikus elemzés jellemzően a környezetfüggő ellenőrzéseket valósítja meg.

- deklarációk kezelése: változók, függvények, eljárások, operátorok, típusok
- láthatósági szabályok
- aritmetikai ellenőrzések
- a program szintaxisának környezetfüggő részei
- típusellenőrzés
- stb.

A szemantikus elemzéshez ki kell egészítenünk a grammatikát. Rendeljünk a szimbólumokhoz attribútumokat és a szabályokhoz akciókat! Egy adott szabályhoz tartozó feltételek csak a szabályban előforduló attribútumoktól függhetnek. (Ha egy feltétel nem teljesül, akkor szemantikus hibát kell jelezni!). A szemantikus rutinok csak annak a szabálynak az attribútumait használhatják és számíthatják ki, amelyikhez az őket reprezentáló akciószimbólum tartozik. Minden szintaxisfában minden attribútumértéket pontosan egy szemantikus rutin határozhat meg. Az így létrejövő nyelvtant *attribútum fordítási grammatikának* (ATG) hívjuk.

A *jól definiált attribútum fordítási grammatika*, olyan attribútum fordítási grammatika, amelyre igaz, hogy a grammatika által definiált nyelv mondataihoz tartozó minden szintaxisfában minden attribútum értéke egyértelműen kiszámítható.

Egy attribútumot kétféleképpen lehet meghatározni:

Szintézissel a szintaxisfában alulról felfelé terjed az információ, egy szülő attribútumát a gyerekekből számoljuk. Kitüntetettnek hívjuk azokat az attribútumokat, melyeket a lexikális elemző szolgáltat.

Öröklődéssel a szintaxisfában felülről lefelé terjed az információ. A gyerekek attribútumait a szülőé határozza meg.

Az *L-ATG* olyan attribútum fordítási grammatika, amelyben minden $A \to X_1X_2...X_n$szabályban az attribútumértékek az alábbi sorrendben meghatározhatók:

- A örökölt attribútumai
- X_1 örökölt attribútumai
- X_1 szintetizált attribútumai
- X_2 örökölt attribútumai
- X_2 szintetizált attribútumai
- ...
- X_n örökölt attribútumai
- X_n szintetizált attribútumai
- A szintetizált attribútumai

Amennyiben a nyelvtanunk ennek eleget tesz, úgy hatékonyan meghatározható minden attribútum.

A szemantikus elemzéshez jellemzően szimbólumtáblát használunk, verem szerkezettel és keresőfával vagy hash-táblával. Minden blokk egy új szint a veremben, egy szimbólum keresése a verem tetejéről indul.

5 Kódgenerálás alapvető vezérlési szerkezetekhez

A kódgenerálás feladata, hogy a szintaktikusan és szemantikusan elemzett programot tárgykóddá alakítsa. Általában szorosan összekapcsolódik a szemantikus elemzéssel.

5.1 Értékadás

assignment → variable assignmentOperator expression

```
a kifejezést az eax regiszterbe kiértékelö kód
2 mov [Változó],eax
```

5.2 Egy ágú elágazás

statement → if condition then program end

```
1 a feltételt az al regiszterbe kiértékelö kód
2 cmp al,1
3 je Then
4 jmp Vége
5 Then: a then-ág programjának kódja
6 Vége:
```

megjegyzés: a dupla ugrásra azért van szükség, mert a feltételes ugrás hatóköre limitált.

5.3 Több ágú elágazás

statement →
if $condition_1$ then $program_1$
elseif $condition_2$ then $program_2$
...
elseif $condition_n$ then $program_n$
else $program_{n+1}$ end

```
1 az 1. feltétel kiértékelése az al regiszterbe
2 cmp al,1
3 jne near Feltétel_2
4 az 1. ág programjának kódja
5 jmp Vége
6
...
7 Feltétel_n: az n-edik feltétel kiértékelése az al regiszterbe
8 cmp al,1
9 jne near Else
10 az n-edik ág programjának kódja
11 jmp Vége
12 Else: az else ág programjának kódja
13 Vége:
```

5.4 Switch-case

statement → switch variable
case $value_1$: $program_1$
...
case $value_n$: $program_n$

```
1 cmp [Változó],Érték_1
2 je near Program_1
3 cmp [Változó],Érték_2
4 je near Program_2
```

```
5
. ..
6 cmp [Változó],Érték_n
7 je near Program_n
8 jmp Vége
9 Program_1: az 1. ág programjának kódja
10
. ..
11 Program_n: az n-edik ág programjának kódja
12 Vége:
```

5.5 Ciklus

5.5.1 Elől tesztelő

statement → while condition statements end

```
1 Eleje: a ciklusfeltétel kiértékelése az al regiszterbe
2 cmp al,1
3 jne near Vége
4 a ciklusmag programjának kódja
5 jmp Eleje
6 Vége:
```

5.5.2 Hátul tesztelő

statement → loop statements while condition

```
1 Eleje: a ciklusmag programjának kódja
2 a ciklusfeltétel kiértékelése az al regiszterbe
3 cmp al,1
4 je near Eleje
```

5.5.3 For ciklus

statement → for variable from $value_1$ to $value_2$ statements end

```
1 a "from" érték kiszámítása a [Változó] memóriahelyre
2 Eleje: a "to" érték kiszámítása az eax regiszterbe
3 cmp [Változó],eax
4 ja near Vége
5 a ciklusmag kódja
6 inc [Változó]
7 jmp Eleje
8 Vége:
```

5.6 Statikus változók

Kezdőérték nélküli változódefiníció fordítása:

```
section .bss
; a korábban definiált változók...
Lab12: resd 1 ; 1 x 4 bájtnyi terület
```

Kezdőértékkel adott változódefiníció fordítása:

```
section .data
; a korábban definiált változók...
Lab12: dd 5 ; 4 bájton tárolva az 5-ös érték
```

5.7 Logikai kifejezések

5.7.1 kifejezés1 < kifejezés2

```
; a 2. kifejezés kiértékelése az eax regiszterbe
push eax
; az 1. kifejezés kiértékelése az eax regiszterbe
pop ebx
cmp eax,ebx
jb Kisebb
mov al,0 ; hamis
jmp Vége
Kisebb:
mov al,1 ; igaz
Vége:
```

5.7.2 kifejezés1 { és, vagy, nem, kizáróvagy } kifejezés2

```
; a 2. kifejezés kiértékelése az al regiszterbe
push ax ; nem lehet 1 bájtot a verembe tenni!
; az 1. kifejezés kiértékelése az al regiszterbe
pop bx ; bx-nek a bl részében van,
; ami nekünk fontos
and al,bl
```

5.7.3 lusta ”és” kiértékelés

```
; az 1. kifejezés kiértékelése az al regiszterbe
cmp al,0
je Vége
push ax
; a 2. kifejezés kiértékelése az al regiszterbe
mov bl,al
pop ax
and al,bl
Vége:
```

5.7.4 Alprogramok megvalósítása

```
; az 1. kifejezés kiértékelése az al regiszterbe
cmp al,0
je Vége
push ax
; a 2. kifejezés kiértékelése az al regiszterbe
mov bl,al
pop ax
and al,bl
Vége:
```

5.7.5 Alprogramok hívása

```
Alprogramok sémája
; utolsó paraméter kiértékelése eax-be
push eax
; ...
; 1. paraméter kiértékelése eax-be
push eax
call alprogram
add esp,'a paraméterek összhossza'
```

6 Kódoptimalizáló

Az optimalizálás feladata, hogy a keletkezett kód kisebb és gyorsabb legyen, úgy hogy a futás eredménye nem változik. A gyorsaság és a tömörség gyakran ellentmondanak egymásnak, és az egyik csak a másik rovására lehet javítani. Általában három lépésben szokás elvégezni:

- Optimalizálási lépések végrehajtása az eredeti programon (vagy annak egyszerűsített változatán)
- Kódgenerálás
- Gépfüggő optimalizálás végrehajtása a generált kódon

6.1 Lokális optimalizáció

Egy programban egymást követő utasítások sorozatát *alapblokknak* nevezzük, ha az első utasítás kivételével egyik utasítására sem lehet távolról átadni a vezérlést (assembly programokban: ahová a jmp, call, ret utasítások "ugranak"; magas szintű nyelvekben: eljárások, ciklusok eleje, elágazások ágainak első utasítása, goto utasítások célpontjai). Az utolsó utasítás kivételével nincs benne vezérlés-átadó utasítás (assembly programban: jmp, call, ret magas szintű nyelvekben: elágazás vége, ciklus vége, eljárás vége, goto). Az utasítás-sorozat nem bővíthető a fenti két szabály megsértése nélkül.

Ha az optimalizálás az alapblokkok keretein belül történik, akkor garantált, hogy az átalakításnak nincs mellékhatása. Ez a *lokális optimalizálás.*

Ablakoptimalizálás Ez egy módszer a lokális optimalizálás egyes fajtáihoz. Egyszerre csak egy néhány utasításnyi részt vizsgálunk a kódból. A vizsgált részt előre megadott mintákkal hasonlítjuk össze. Ha illeszkedik, akkor a mintához megadott szabály szerint átalakítjuk ezt az "ablakot" végigcsúsztatjuk a programon. Az átalakítások megadása:

{ minta $\rightarrow$ helyettesítés } szabályhalmazzal (a mintában lehet paramétereket is használni)

Példák:

- felesleges műveletek törlése: nulla hozzáadása vagy kivonása
- egyszerűsítések: nullával szorzás helyett a regiszter törlése
- regiszterbe töltés és ugyanoda visszaírás esetén a visszaírás elhagyható
- utasításismétlések törlése: ha lehetséges, az ismétlések törlése

6.2 Globális optimalizáció

A teljes program szerkezetét meg kell vizsgálni. Ennek módszere az adatáram-analízis:

- Mely változók értékeit számolja ki egy adott alapblokk?
- Mely változók értékeit melyik alapblokk használja fel?

Ez lehetővé teszi az azonos kifejezések többszöri kiszámításának kiküszöbölését akkor is, ha különböző alapblokkokban szerepelnek; valamint a konstansok és változók továbbterjesztését alapblokkok között is elágazások, ciklusok optimalizálását.

7 A szekvenciális és párhuzamos/elosztott végrehajtás összehasonlítása

7.1 Szekvenciális végrehajtás:

Ilyenkor a végrehajtás egy processzoron történik. Minden művelet atomi. Egy inputhoz egy output tartozik. Két szekvenciális program ekvivalens, ha ezek a párosok megegyeznek. Nem használja fel az összes rendelkezésre álló erőforrást.

7.2 Párhuzamos végrehajtás:

Több processzoron hajtódik végre a program. A párhuzamos folyamatok egymással kommunikálva, szinkronban oldják meg az adott problémát. A konkurens program szétbontható elemi szekvenciális programokra, ezek a folyamatok. A folyamatok használhatnak közös erőforrásokat: pl. változók, adattípus objektumok, kommunikációs csatornák.

A kommunikációt általában kétféleképpen szokták megvalósítani.

Osztott memóriával. Ekkor szinkronizálni kell, hogy ki mikor fér hozzá, hogy ne legyen ütközés.

Kommunikációs csatornával. Garantálni kell, hogy ha egy folyamat üzenetet küld egy másiknak, akkor az meg is kapja azt, és jelezzen is vissza. Ügyelni kell, nehogy deadlock alakuljon ki.

Chapter 10

10

Záróvizsga tételsor
10. Programnyelvi alapok

Ancsin Ádám

Programnyelvi alapok

Kifejezések kiértékelésének szabályai. Vezérlési szerkezetek: utasítások, rekurzió. Típusok: tömb, rekord, osztály, öröklődés, statikus és dinamikus kötés, polimorfizmus. Generikusok. Hatókör/láthatóság. Automatikus, statikus és dinamikus élettartam, szemétgyűjtés. Konstruktor, destruktor. Objektumok másolása, összehasonlítása. Alprogramok, paraméterátadás, túlterhelés.

1 Kifejezések kiértékelésének szabályai

Fogalmak:

- Operandusok: Változók, konstansok, függvény- és eljáráshívások.
- Operátorok: Műveleti jelek, amelyek összekapcsolják egy kifejezésben az operandusokat és valamilyen műveletet jelölnek.
- Kifejezés: operátorok és operandusok sorozata
- Precedencia: A műveletek kiértékelési sorrendjét határozza meg.
- Asszociativitás iránya: Az azonos precedenciájú operátorokat tartalmazó kifejezésekben a kiértékelés iránya. Megkülönböztetünk bal-asszociatív és jobb-asszociatív operátorokat.

Az operátorokat háromféleképpen írhatjuk az operandusokhoz képest:

- Infix : Egy operátort a két operandusa közé kell írni (tehát csak kétoperandusú műveletek operátorait lehet így írni). Amikor egy kifejezésben több operátor is szerepel, akkor a különböző operátorok végrehajtási sorrendjét az operátorok precedenciája dönti el. Amelyik operátor precedenciája magasabb (pl. a szorzásé magasabb, mint az összeadásé), az általa jelölt műveletet értékeljük ki először. Ugyanazon operátorok végrehajtási sorrendjét pedig az asszociativitás iránya dönti el (pl. a bal-asszociatív azt jelenti, hogy balról jobbra haladva kell végrehajtani). Ezeket a (programozási nyelvekbe beépített) szabályokat zárójelek segítségével lehet felülírni.
  Példa: A * (B + C) / D
- Postfix (Lengyelforma) : Az operátorokat az operandusaik mögé írjuk. A kiértékelés sorrendje mindig balról jobbra történik – tehát egy n operandusú operátor a tőle balra levő első n operandusra érvényes.
  Példa: A B C + * D /
  Ugyanez zárójelezve (felesleges): ((A (B C +) *) D /)
- Prefix : Az operátorokat az operandusuk elé írjuk. A kiértékelés sorrendje balról jobbra történik.
  Példa: / * A + B C D
  Ugyanez zárójelezve (felesleges): (/ (* A (+ B C)) D)
  Habár a prefix operátorok esetén is balról jobbra történik a kiértékelés, viszont ha egy operátortól jobbra egy másik operátor következik, akkor értelemszerűen az ehhez az operátorhoz tartozó műveletet kell először végrehajtani, hogy a bal oldalit is végre tudjuk hajtani. A fenti példában is a szorzást az osztás előtt, az összeadást pedig a szorzás előtt kell elvégezni.

Logikai operátorokat tartalmazó kifejezések kiértékelése

Az ilyen kifejezéseknek kétféle kiértékelése létezik:

- Lusta kiértékelés : Ha az első argumentumból meghatározható a kifejezés értéke, akkor a másodikat már nem értékeli ki.
- Mohó kiértékelés : Mindenféleképpen megállapítja mindkét argumentum logikai értékét.

A két kiértékelési módszer bizonyos esetekben különböző eredményt adhat:

- A 2. argumentum nem mindig értelmes
  Példa (C++):

  ```
  if ((i>=0) && (T[i]>=10))
  {
      //...
  }
  ```

  Tegyük fel, hogy a T egy `int` tömb, 0-tól indexelődik. Itt ha az $i >= 0$ hamis, akkor T-t alul indexelnénk. Ez mohó kiértékelés esetén futási idejű hibát okozna. Lusta kiértékelés esetén (a C++ alapértelmezetten ezt használja) viszont tudhatjuk, hogy a feltétel már nem lehet igaz, emiatt $T[i] >= 10$-et már nem kell kiértékelni.

- A 2. argumentumnak van valamilyen mellékhatása.
  Példa (C++):

  ```
  if ((i>0) || (++j>0))
  {
      T[j] = 100;
  }
  ```

  Ebben az esetben ha $i > 0$ igaz, akkor a feltétel biztosan igaz, viszont a $++j > 0$ kifejezés mellékhatásos, növeli j értékét. Mivel a C++ lusta kiértékelést használ a || operátor esetén (| operátor esetén mohó a kiértékelés), ezért ebben az esetben nem növeli j értékét. (csak akkor, ha $i > 0$ hamis).

2 Utasítások, vezérlési szerkezetek

2.1 Egyszerű utasítások

- Értékadás : Az értékadás bal oldalán egy változó, a jobb oldalán bármilyen kifejezés állhat. Az értékadással a változóhoz rendeljük a jobb oldali kifejezést. Figyelni kell arra, hogy a bal oldali változó típusának megfelelő kifejezés álljon a jobb oldalon (vagy létezik implicit konverzió, pl. C++-ban az egész és logikai típus között). A legtöbb nyelvben az értékadás operátora az `=` (például C++, Java, C#), vagy a `:=` (például Pascal, Ada).
- Üres utasítás : Nem mindenhol lehet ilyet írni. A lényege, hogy nem csinál semmit. Azokban a nyelvekben lehet létjogosultsága, ahol üres blokkot nem írhatunk, muszáj legalább 1 utasításnak szerepelnie benne. (pl. Ada) Erre szolgál az üres utasítás. (Ada-ban ez a `null` utasítás)
- Alprogramhívás : Alprogramokat nevük és paramétereik megadásával hívhatunk. Példák:
  - `System.out.println("Hello");`
  - `int x = sum(3,4);`
- Visszatérés utasítás : Az utasítás hatására az alprogram végrehajtása befejeződik. Ha az alprogram egy függvény, akkor meg kell adni a visszatérési értéket is.
  Példák (C++):

– Ebben a példában a `doSomething` egy eljárás, nincs visszatérési értéke. A paraméterül kapott x változót értékül adjuk a j-nek. Ha ez az érték nem 0, akkor visszatérünk, azaz megszakítjuk az alprogram végrehajtását (konkrét értéket viszont nem adunk vissza). Ha ez az érték 0, akkor a végrehajtás folytatódik tovább, meghívjuk a `doSomethingElse` függvényt a j paraméterrel.

```
void doSomething(int x)
{
    int j;
    if(j=x)
        return; // j!=0, do nothing

    doSomethingElse(j);
}
```

– Ebben a példában az `isOdd` egy függvény, `int` visszatérési értékkel. Megmondja a paraméterül kapott x egész számról, hogy páratlan-e. Ehhez bitenkénti ÉS művelettel "összeéseli" az x-et az 1-gyel (0...01). Ha az eredmény nem 0, akkor páratlan, visszatérünk igaz értékkel. Különben folytatjuk a működést, majd visszatérünk hamis értékkel.

```
int isOdd(int x)
{
    if(x & 1) //bitwise AND with 0...01 is not 0...0
        return true;

    return false;
}
```

- Utasításblokk: A blokkon belüli utasítások "összetartoznak". Ez több esetben is jól alkalmazható nyelvi elem:
  – Vezérlési szerkezetekben: Az adott vezérlési szerkezetekhez tartozó utasításokat különíthetjük el.
  – Az olyan nyelvekben, amelyekben deklaráció csak a program elején található deklarációs blokkokban lehetséges (pl. Ada), van lehetőség arra, hogy a programkód későbbi részében nyissunk egy blokkot, ahol deklarációk is szerepelhetnek.
  – Osztályok inicializáló blokkja pl. Java-ban (konstruktor előtt hajtódik végre):

```
public class MyClass{
    private ResourceSet resourceSet;

    {
        resourceSet = new ResourceSetImpl();
        UMLResourcesUtil.init(resourceSet);
    }

    /* ...   */
}
```

  – Osztályok statikus inicializáló blokkja pl. Java-ban:

```
public class MyClass{
    private static ResourceSet resourceSet;

    static{
        resourceSet = new ResourceSetImpl();
        UMLResourcesUtil.init(resourceSet);
    }

    /*  ...   */
}
```

2.2 Vezérlési szerkezetek

- Elágazás : Az elágazás egy olyan vezérlési szerkezet, amellyel meghatározhatjuk, hogy bizonyos (blokkban megadott) utasítások csak a megadott feltétellel jöhessenek létre. Általában több feltételt is megadhatunk egymás után (if L1 then ... else if L2 then ... else if L3 then ...). Megadhatjuk azt is, hogy mi történjen, ha egyik feltétel sem teljesül (if L then ... else ...).

  Elágazásokat lehet egymásba ágyazni (if ... then if ...).

  "Csellengő else" ("Dangling else") probléma: Azokban a nyelvekben lép fel, ahol egy feltétel egy utasításblokkját nem zárja le külön kódszó (pl. endif). Ekkor abban az esetben, ha elágazásokat egymásba ágyazunk, a következő probléma léphet fel:
  Példa: `if (A>B) then if (C>D) then E:=100; else F:=100;`

  A fenti esetben nem lehet megállapítani, hogy a programozó az else kulcsszót melyik elágazásra értette.

- Ciklus : Egy utasításblokk (ciklusmag) valahányszori végrehajtását jelenti.

  - Feltétel nélküli ciklus : Végtelen ciklust kódol, kilépni belőle a strukturálatlan utasításokkal lehet (ld. lentebb), vagy return-nel, esetleg hiba fellépése esetén.
    Példa (ADA):

    ```
    loop
        null;
    end loop;
    ```

  - Elöltesztelő ciklus : A ciklus a ciklusmag minden végrehajtása előtt megvizsgálja, hogy az adott feltétel teljesül-e. Ha teljesül, akkor végrehajtja a magot, majd újra ellenőriz. Különben a ciklus után folytatódik a futás.
    Példák:

    * C++:

      ```
      int x=0,y=0;
      while(x<5 && y<5)
      {
          x+=y+1;
          y=x-1;
      }
      cout<<x+y<<endl;
      ```

    * ADA:

      ```
      declare
          X,Y: Integer;
      begin
          X:=0;
          Y:=0;
          while X<5 and Y<5 loop
              X:=X+Y+1;
              Y:=X-1;
          end loop;
      end;
      ```

  - Számlálásos ciklus : Ebben a vezérlési szerkezetben megadhatjuk, hogy a ciklusmag hányszor hajtódjon végre. Példa (ADA):

    ```
    for i in 1..10 loop
        null;
    end loop;
    ```

    A számlálás úgy történik, hogy egy változóban (ciklusváltozó) tároljuk, hogy "hol tartunk" - ezt hasonlítjuk össze minden ciklus elején a kifejezéssel, amit meg kell haladnia a változónak (i). Tehát felfogható egy elöltesztelő cikluskénti is, ahol a ciklusfeltétel az `i<=10` és a ciklusmag végén növelni kell i-t.

– Hátultesztelő ciklus : Az a különbség az elöltesztelőhöz képest, hogy a feltételt a ciklusmag végrehajtása után ellenőrizzük – tehát itt a ciklusmag 1-szer mindenképpen lefut.
Példa (C++):

```
int i=0;
do
{
    ++i;
} while(i<5);
```

Megjegyzés: vannak olyan nyelvek (pl. Pascal), amelyekben a hátultesztelő ciklus feltétele nem bennmaradási, hanem leállási feltétel. Azaz nem azt adjuk meg, hogy minek kell teljesülnie ahhoz, hogy még egyszer végrehajtásra kerüljön a ciklusmag, hanem azt, hogy minek kell teljesülnie ahhoz, hogy a ciklusmag ne hajtódjon végre többször.
Az előbbi C++-os példa Pascal-os megfelelője:

```
i:=0;
repeat
    i:=i+1;
until i=5;
```

Megjegyzés: ADA-ban nincs igazi hátultesztelős ciklus. Ebben a nyelvben hátultesztelő ciklust úgy írhatunk, hogy ha írunk egy feltétel nélküli ciklust, amelynek utolsó utasítása egy feltételhez kötött kilépés.
Példa:

```
i:=0;
loop
    i:=i+1;
    exit when i=5;
end loop;
```

– foreach : Akkor használatos, ha egy adatszerkezet minden elemére végre akarjuk hajtani a magot. Tulajdonképpen ez is egy elöltesztelő ciklus (a ciklusfeltétel az, hogy a végére értünk-e az adatszerkezetnek). Példa:

```
foreach (int v in Vect)
{
    ++v;
}
```

Megjegyzések:

1. Nincs minden programozási nyelvben ilyen ciklus. Leginkább az újabb nyelvekben terjedt el.
2. Nem minden programozási nyelvben a foreach a kulcsszó ehhez a ciklushoz. Például Java-ban, C++11-ben (régebbi változatokban nincs ilyen) a for ciklust használhatjuk foreach-ként:

```
int x=0;
for (int v : vect){
    x+=v;
}
```

2.3 Strukturálatlan utasítások

- Ciklus megszakítása : A ciklusból való "kiugrásra" (tehát annak azonnali befejezésére) használható. Gyakran végtelen ciklus megszakítására használjuk, vagy hátultesztelő ciklus kódolására (ahol nincs erre beépített vezérlési szerkezet, például Ada).

  Ilyen utasítás pl. C/C++/C#-ban, vagy Java-ban a `break`, illetve Ada-ban az `exit`.

- `goto` utasítás : A programkódban címkéket definiálhatunk, majd a `goto` utasítással egy ilyen címkéhez irányíthatjuk a vezérlést. Vezérlési szerkezeteket is lehet vele kódolni. Túlzott használata olvashatatlan kódhoz vezethet.

2.4 Rekurzió

Rekurzív alprogram: Olyan alprogram, amelynek kódjában szerepel önmagának a meghívása. Mindenképpen kell, hogy legyen a rekurziónak megállási feltétele – tehát egy olyan feltétel (egy elágazással együtt), amelynek teljesülése esetén nem történik rekurzív hívás, így az összes, folyamatban levő rekurzív hívás végre tud hajtódni (ellenkező esetben végtelen rekurzió lép fel, ilyenkor általában előbb-utóbb betelik a stack és leáll a program).

Példák (faktoriális):

- C++:

```
int fact (int n)
{
    if(n>0)
        return n * fact (n-1);
    else
        return 1;
}
```

- Haskell:

```
fact :: (Integral a) => a -> a
fact n
    | n>0 = n * fact (n - 1)
    | otherwise = 0
```

3 Típusok

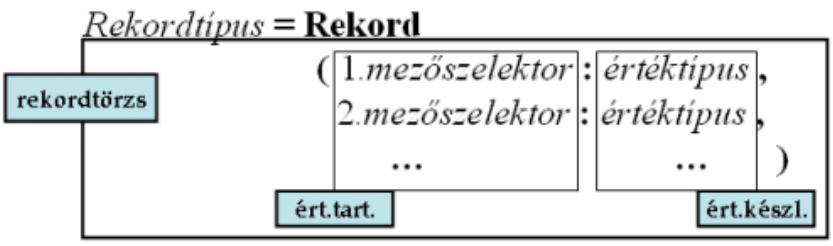
3.1 Tömb

A tömb (angolul array) olyan adatszerkezet, amelyet nevesített elemek csoportja alkot, melyekre sorszámukkal (indexükkel) lehet hivatkozni. Vektornak is nevezik, ha egydimenziós, mátrixnak esetenként, ha többdimenziós. A legtöbb programozási nyelvben minden egyes elemnek azonos adattípusa van és a tömb folytonosan helyezkedik el a számítógép memóriájában. A készítés módja alapján lehet:

- statikus :a méret fix, deklarációban szabályozott
- dinamikus tömb: a mérete változik, folyamatosan bővíthető

3.2 Rekord

A rekord egy összetett értékek leírásához használható konstrukció. Névvel és típussal ellátott összetevői vannak, ezeket mezőknek nevezzük. Értékhalmaz a mezők értéktípusai által meghatározott alaphalmazok direktszorzata.



ábra 1: A rekord-típuskonstrukciók általános szintaxisa.

Műveletek:

- Szelekciós függvény (.mezőszelektor szintaxisú);

- konstrukciós függvény (Rekordtípus(mezőértékek) szintaxisú);
- elképzelhetők transzformációs függvények, amelyek a teljes rekordstruktúrát érintik.

Példa rekordra és használatára C-ben:

```
//definition of Point
struct Point
{
    int xCoord;
    int yCoord;
};

const struct Point ORIGIN = {0,0};
```

3.3 Osztály

Az osztály egy felhasználói típus, amelynek alapján példányok (objektumok) hozhatók létre. Az osztály alapvetően attribútum és metódus (művelet) definíciókat tartalmaz. Az osztály írja le az objektum típusát: megadja a tulajdonságait és azok lehetséges értékeit (azaz a típusértékeket), valamint az objektumon végrehajtható műveleteket (típusműveletek).

Példa C++-ban:

```
//definition of Point
class Point {
private:
    int xCoord;
    int yCoord;
public:
    //constructor
    Point(int xCoord, int yCoord) : xCoord(xCoord),yCoord(yCoord) {}

    void Translate(int dx, int dy) {
        xCoord+=dx;
        yCoord+=dy;
    }

    void Translate(Point delta) {
        xCoord+=delta.xCoord;
        yCoord+=delta.yCoord;
    }

    int getX() { return xCoord; }

    int getY() { return yCoord; }
};

int main(int argc, char* argv[])
{
    Point point(0,0);
    point.Translate(5,-2);
    cout<<point.getX()<<","<<point.getY()<<endl; //5,-2
    Point delta(-2,1);
    point.Translate(delta);
    cout<<point.getX()<<","<<point.getY()<<endl; //3,-1

    return 0;
}
```

Megjegyzés: A nem objektum-orientált nyelvekben nincsenek osztályok, pl. régebbi nyelvekben, mint a C, ADA, Pascal, vagy funkcionális nyelvekben (Haskell, Clean, stb.).

3.4 Öröklődés

Egy osztály legegyszerűbben adattagjainak és metódusainak felsorolásával hozható létre. Azonban az objektum-orientált paradigma lehetőséget ad egy másik, hatékonyabb módszerre is, az öröklődésre. Az öröklődés az újrafelhasználhatóságot szem előtt tartva arra ad lehetőséget, hogy már meglévő (szülő-, ős-) osztályból kiindulva hozzunk létre új (gyermek-, leszármazott-, al-) osztályt. Az öröklés két osztály között fennálló olyan kapcsolat, amely során a leszármazott osztály rendelkezik a szülő osztály majdnem összes tulajdonságával (nem privát adattagjait és metódusait sajátjaként kezeli), s ezeket újabbakkal egészítheti ki. Az így létrehozott osztály is lehet más osztályok őse (kivéve pl. Java-ban a `final` kulcsszóval ellátott osztályok, ezekből már nem származtathatunk), így ezek az osztályok egy öröklési hierarchiába szerveződnek. Attól függően, hogy egy osztálynak egy- vagy több őse van, beszélünk egyszeres- ill. többszörös öröklődésről. A Java az egyszeres öröklődést támogatja, de pl. C++-ban van többszörös öröklődés.

A Java-ban az osztályhierarchia legfelső eleme az Object osztály, amelyből minden más osztály (közvetve vagy közvetlenül) származik.

Egy alosztály az örökölt metódusokat újraimplementálhatja. Ilyenkor az adott metódus ugyanolyan néven, de más, módosított (alosztályra specifikált) tartalommal kerül megvalósításra. Az ilyen metódusokat polimorfnak nevezzük. Java-ban minden olyan metódust, ami nincs ellátva a `final` kulcsszóval, újra lehet definiálni. (C++-ban mindent, ami nem privát)

Példák:

- C++:

```
class RegularPolygon
{
protected:
    const double radius;
public:
    RegularPolygon(double radius) : radius(radius) {}
    virtual double area() = 0;
};

class EquilateralTriangle : public RegularPolygon
{
public:
    EquilateralTriangle(double radius) : RegularPolygon(radius) {}
    virtual double area()
    {
         return 0.75*sqrt(3.0)*radius*radius;
    }
};
```

- Java:

```
public abstract class RegularPolygon{
    protected final double radius;

    public RegularPolygon(double radius){
        this.radius = radius;
    }
    public abstract double area();
}

public class EquilateralTriangle extends RegularPolygon{
    public EquilateralTriangle(double radius){
```

```
        super(radius);
    }
    public double area(){
          return 0.75*Math.sqrt(3.0)*radius*radius;
    }
}
```

3.5 Statikus és dinamikus kötés

- statikus típus: A változó deklarációjában megadott típus. Fordítás során egyértelműen eldől, nem változhat futás során. A statikus típus határozza meg, hogy mit szabad csinálni az objektummal (pl. hogy milyen műveletek hívhatók meg rá).
- dinamikus típus: A változó által hivatkozott objektum típusa. Vagy a statikus típus leszármazottja, vagy maga a statikus típus. Futás során változhat.

Példa(Java):

```
Object o = new String("Hello");
```

Itt az o változó statikus típusa `Object`, dinamikus típusa pedig `String`.

Kötések:

- statikus kötés: A változó statikus típusa szerinti adattagokra lehet hivatkozni.
- dinamikus kötés: A változó dinamikus típusa szerinti adattagok használhatók.

A kötés akkor fontos, ha a hívott művelet, vagy a hivatkozott változó a statikus és a dinamikus típusban különbözik, vagy esetleg a statikus típusban nem is létezik (ekkor a dinamikus típusban sem hivatkozhatunk az adattagra).

Többféleképpen lehet a kötéseket meghatározni a különböző nyelvekben:

- Java : Minden esetben dinamikus kötés van (az örökölt metódusok törzsében is).
- C++ : A művelet definiálásakor lehet jelezni, ha dinamikus kötést szeretnénk (`virtual`).
- Ada : A híváskor lehet jelezni, ha dinamikus kötést szeretnénk.

4 Generikusok

Generikus programozás: Algoritmusok, adatszerkezetek általánosított, több típusra is működő leprogramozása (pl. generikus rendezés, generikus verem, ...). Ezt az általános kódot nevezzük sablonnak (template).

Bizonyos nyelvekben (Ada, C++) egy sablont példányosítani kell – ekkor a kívánt típusokat, objektumokat (a sablon definíciónak megfelelően) a sablonnak paraméterként megadva példányosul a szóban forgó sablon. (Ugyanúgy, mint pl. amikor egy függvénynek megadjuk az aktuális paraméterét.) Ezt nevezzük generatív programozásnak: a program a megadott sablon alapján létrehoz egy "igazi" programegységet (program generál programot).

Példa: Egy verem sablon példányosításakor megadjuk, hogy milyen típusú elemeket tároljon a verem (típus paraméter) és hogy hány elem fér a verembe (objektum paraméter). C++ és Ada esetén egy sablon paramétere alprogram is lehet.

Példa: Egy rendezésnek megadjuk, hogy milyen művelet alapján rendezzen. Természetesen lehet alapértelmezett sablonparaméter is.

Egy sablon definiálása esetén természetesen a sablont nem lehet minden típusra használni. Egy sablon attól lesz sablon, hogy több típusra is működik. Azonban megszorításokat tehetünk (és tennünk is kell) arra, hogy milyen típusokra lehessen azt használni, hogyan lehessen a sablont példányosítani.

Jó példa erre az Ada nyelv, ahol a sablon specifikációja egy "szerződés" a sablon törzse és a példányosítás között:

- A sablon törzse nem használhat mást, csak amit a sablon specifikációja megenged neki. (A törzset nem feltétlenül kell, hogy ismerjük példányosításkor.)

```
generic
    type Element_T is private;
    with function "*" (X, Y: Element_T) return Element_T is <>;
function Square (X : Element_T) return Element_T;
```

Itt a `with function` kezdetű sor végén az `is <>` azt jelenti, hogy ha az adott típusra már létezik * művelet (pl. egész számokra), akkor nem kell külön megadni példányosításkor, a program automatikusan azt használja.

A törzs:

```
function Square (X: Element_T) return Element_T is
begin
    return X * X;   -- The formal operator "*".
end Square;
```

- A példányosításnak biztosítania kell mindent, amit a sablon specifikációja megkövetel tőle. A következő példában a négyzetre emelő függvényt mátrixokra alkalmazzuk, feltéve, hogy definiáltuk a mátrixszorzást.

```
with Square;
with Matrices;
procedure Matrix_Example is
    function Square_Matrix is new Square
        (Element_T => Matrices.Matrix_T, "*" => Matrices.Product);
    A : Matrices.Matrix_T := Matrices.Identity;
begin
    A := Square_Matrix (A);
end Matrix_Example;
```

Például a C++-ban a sablonszerződés nem így működik. Ott a sablon specifikációja az egész definíció (emiatt sablonosztályokat csak teljes egészében header fájlokban definiálhatunk). Példányosításkor ezt is ismerni kell, hogy tudjuk, hogyan példányosíthatunk. Az információelrejtés elve tehát sérül.

Más nyelvekben (pl. Java, funkcionális nyelvek) nem kell a sablonokat példányosítani – mindig ugyanaz a megírt kód hajtódik végre, csak épp az aktuális paraméterekkel. Pl. Java-ban a típusparaméter fordításkor "elveszik", csak futási időben derül ki.

5 Hatókör/láthatóság

1. Hatókör: Deklarációkor a programozó összekapcsol egy entitást (például egy változót vagy függvényt) egy névvel. A hatókör alatt a forrásszöveg azt a részét értjük, amíg ez az összekapcsolás érvényben van. Ez általában annak a blokknak a végéig tart, amely tartalmazza az adott deklarációt.

2. A láthatóság a hatókör részhalmaza, a programszöveg azon része, ahol a deklarált névhez a megadott entitás tartozik. Mivel az egymásba ágyazott blokkokban egy korábban már bevezetett nevet más entitáshoz kapcsolhatunk, ezért ilyenkor a külső blokkban deklarált entitás a nevével már nem elérhető. Ezt nevezzük a láthatóság elfedésének.

   Egyes nyelvekben (például C++) bizonyos esetekben (például osztályszintű adattagok) a külső blokkban deklarált entitáshoz minősített névvel hozzá lehet férni ekkor is.

6 Automatikus, statikus és dinamikus élettartam, szemétgyűjtés

Élettartam: A változók élettartama alatt a program végrehajtási idejének azt a szakaszát értjük, amíg a változó számára lefoglalt tárhely a változóé.

6.1 Automatikus élettartam

A blokkokban deklarált lokális változók automatikus élettartamúak, ami azt jelenti, hogy a deklarációtól a tartalmazó blokk végéig tart, azaz egybeesik a hatókörrel. A helyfoglalás számukra a végrehajtási verem aktuális aktivációs rekordjában történik meg.

6.2 Statikus élettartam

A globális változók, illetve egyes nyelvekben a statikusként deklarált változók (például C/C++ esetén a `static` kulcsszóval) statikus élettartamúak. Az ilyen változók élettartama a program teljes végrehajtási idejére kiterjed, számukra a helyfoglalás már a fordítási időben megtörténhet.

6.3 Dinamikus élettartam

A dinamikus élettartamú változók esetén a programozó foglal helyet számukra a dinamikus tárterületen (heap), és a programozó feladata gondoskodni arról is, hogy ezt a tárterületet később felszabadítsa. Amennyiben utóbbiról megfeledkezik, azt nevezzük memóriaszivárgásnak (memory leak). Mint látjuk, a dinamikus élettartam esetén a hatókör semmilyen módon nem kapcsolódik össze az élettartammal, az élettartam szűkebb vagy tágabb is lehet a hatókörnél.

6.4 Szemétgyűjtő

A szemétgyűjtő másik neve a hulladékgyűjtő, az angol Garbage Collector név után pedig gyakran csak GC-nek rövidítik. Feladata a dinamikus memóriakezeléshez kapcsolódó tárhelyfelszabadítás automatizálása, és a felelősség levétele a programozó válláról, így csökkentve a hibalehetőséget.

A szemétgyűjtő figyeli, hogy mely változók kerültek ki a hatókörükből, és azokat felszabadíthatóvá nyilvánítja. A módszer hátránya a számításigényessége, illetve a nemdeterminisztikussága. A szemétgyűjtő ugyanis nem szabadítja fel egyből a hatókörükből kikerült változókat, és a felszabadítás sorrendje sem ugyanaz, amilyen sorrendben a változók felszabadíthatóvá váltak.

Azt, hogy a hulladékgyűjtő mikor és mely változót szabadítja fel, egy programozási nyelvenként egyedi, összetett algoritmus határozza meg, amelyben rendszerint szerepet játszik a rendelkezésre álló memória telítettsége, illetve a felszabadításhoz szükséges becsült idő. (Például ha egy objektum rendelkezik destruktorral, akkor általában a GC később szabadítja csak fel.)

Összességében a szemétgyűjtő csak annyit garantál, hogy előbb-utóbb (legkésőbb a program futásának végeztével) minden dinamikusan allokált változót felszabadít.

Szemétgyűjtést használó nyelvek pl. Java, C#, Ada. C/C++-ban nincs szemétgyűjtés, a programozónak kell gondoskodni a dinamikusan allokált memóriaterületek felszabadításáról.

7 Konstruktor, destruktor

7.1 Konstruktor

A konstruktor az objektumok inicializáló eljárása, akkor fut le, ha egy osztályból új objektumot példányosítunk. Alapértelmezett konstruktor alatt a paraméter nélküli konstruktort értjük, a legtöbb programozási nyelv esetén ezt a fordítóprogram automatikusan generálja üres törzzsel, amennyiben nem lett megadva egy konstruktor sem egy osztályban. Többek között a konstruktorban szokás gondoskodni arról, hogy az objektum dinamikus élettartamú változói számára tárhelyet foglaljunk.

7.2 Destruktor

A destruktor a konstruktor ellentétes párja, ez az eljárás az objektumok felszabadításakor fut le. Meghívása automatikusan megtörténik, attól függetlenül, hogy az objektum felszabadítása automatikusan történik a hatókör végeztével (lokális objektumok esetén), manuálisan a programozó által (dinamikus élettartamú objektumok esetén) vagy a szemétgyűjtő által (szintén dinamikus élettartamú objektumok esetén).

A desktruktorban szokás többek között az objektum dinamikus helyfoglalású adattagjait és a lefoglalt erőforrásokat felszabadítani.

8 Objektumok másolása, összehasonlítása

Az objektumok másolása egy speciális konstruktorral, az úgynevezett másoló konstruktorral (copy constructor) történik. Ez paraméterül az adott osztály egy példányát kapja meg, és azt a programozó által megadott működési logika szerint lemásolja az éppen inicializált objektumba.

Több programozási nyelv (például C++) fordítóprogramja automatikusan elkészít egy másoló konstruktort, ha a programozó nem definiál sajátot. Ez az alapértelmezett másoló konstruktor lemásolja a forrásobjektum összes adattagjának értékét, ezt nevezzük sekély másolatnak (shallow copy). Ha az objektum dinamikus foglalású adattagokat is tartalmaz, akkor azoknak nem az értéke, hanem csak a hivatkozása lesz lemásolva, ami általában nem a kívánt működés. Ez esetben saját másoló konstruktor írása szükséges, ami mély másolatot (deep copy) készít.

9 Alprogramok, paraméterátadás, túlterhelés

9.1 Alprogramok

Alprogramoknak a függvényeket, eljárásokat és műveleteket nevezzük. Segítségükkel a program feladatonként tagolható, a főprogramból az önálló feladatok kiszervezhetőek.

9.2 Paraméterátadás

Az alprogramoknak szüksége lehet bemenő adatokra és vissza is adhat értékeket. Az alprogramokat általános írjuk meg, saját változónevekkel, ezek a formális paraméterek. Az alprogram meghívásakor az átadott aktuális paraméterek alapján a formális paraméterek értéket kapnak. Az, hogy a formális paraméterek értéke mi lesz, a paraméterátadás módjától függ.

9.2.1 Szövegszerű paraméterátadás

A makrókban használatosak mind a mai napig. A makró törzsében a formális paraméter helyére beíródik az aktuális paraméter szövege.

9.2.2 Név szerinti paraméterátadás

Az aktuális paraméter kifejezést újra és újra kiértékeljük, ahányszor hivatkozás történik a formálisra. A paramétert a törzs kontextusában értékeljük ki, így a formális paraméter különböző előfordulásai mást és mást jelenthetnek az alprogramon belül. Alkalmazása archaikus (például: Algol 60, Simula 67).

9.2.3 Érték szerinti paraméterátadás

Az egyik legelterjedtebb paraméterátadási mód (például: C, C++, Pascal, Ada, Java), bemeneti szemantikájú. A formális paraméter az alprogram lokális változója, híváskor a vermen készül egy másolat az aktuális paraméterről, ez lesz a formális. Az alprogram végén a formális paraméter megszűnik

9.2.4 Cím szerinti paraméterátadás

A másik legelterjedtebb paraméterátadási mód (például: Pascal, C++, C#), be- és kimeneti szemantikájú. A híváskor az aktuális paraméter címe adódik át, azaz a formális és az aktuális paraméter ugyanazt az objektumot jelentik, egy alias jön létre.

Megjegyzés: A Java-ban nincs cím szerinti paraméterátadás, csak érték szerinti. A Java ugyanis a primitív típusoknak az értékét tárolja, objektumok esetén pedig egy referenciát az adott objektumra (mint C++-ban a referencia típus). Objektum átadásakor ez a referencia másolódik le, azaz a referencia adódik át érték szerint.

Például:

```
public static void BadSwap(Object x, Object y)
{
    Object tmp = x;
    x = y;
```

```
    y = tmp;
}
```

A fenti függvény nem cseréli ki az x-et és y-t, csak lokálisan (a függvénytörzsön belül), viszont a hívás helyén x és y is helyben maradnak.

9.2.5 Eredmény szerinti paraméterátadás

Kimeneti szemantikájú paraméterátadási mód. A formális paraméter az alprogram lokális változója, az alprogram végén a formális paraméter értéke bemásolódik az aktuálisba. Azonban az alprogram meghívásakor az aktuális értéke nem másolódik be a formálisba. (Használja például az Ada.)

9.2.6 Érték/eredmény szerinti paraméterátadás

Az érték és eredmény szerinti paraméterátadás összekombinálása, így egy be- és kimeneti szemantikájú paraméterátadási módot kapunk. (Használja például az Algol-W vagy az Ada.)

9.2.7 Megosztás szerinti paraméterátadás

Objektumorientált programozási nyelvek (például: CLU, Eiffel) paraméterátadási módja. Lényege, hogy ha a formális paraméter megváltoztatható és az aktuális paraméter egy megváltoztatható változó, akkor cím szerinti paraméterátadás történik, egyébként pedig érték szerinti. A paraméterátadás módját külön megadni nem lehet. Megváltoztatható (mutable) objektum alatt azt értjük, hogy tulajdonságai, ezáltal állapota megváltoztatható.

9.2.8 Igény szerinti paraméterátadás

Ezt a paraméterátadási módot a lusta kiértékelésű funkcionális nyelvek (például: Clean, Haskell, Miranda) alkalmazzák. Az aktuális paramétert nem híváskor értékeli ki, hanem akkor, amikor először szüksége van rá a számításokhoz.

9.3 Túlterhelés

A túlterhelés (overloading) segítségével azonos nevű alprogramokat hozhatunk létre eltérő szignatúrával. A szignatúra a legtöbb programozási nyelvben az alprogram nevét és a formális paraméterek számát és típusát jelenti, de egyes nyelvekben (például Ada) a visszatérési érték típusa is beletartozik. A túlterhelés elsődleges felhasználási területe, hogy ugyanazt a tevékenységet különböző paraméterezéssel is elvégezhessük.

A fordító az alprogramhívásból el tudja dönteni, hogy a túlterhelt változatok közül melyiket kell meghívni. Ha egyik sem illeszkedik vagy több is illeszkedik, akkor fordítási hiba lép fel.

Chapter 11

11

Záróvizsga tételsor
11. Formális nyelvek

Dobreff András

Formális nyelvek
Formális nyelvtanok és a Chomsky-féle nyelvosztályok. Automaták: véges automata, veremautomata. Reguláris nyelvek tulajdonságai és alkalmazásai. Környezetfüggetlen nyelvek tulajdonságai és elemzésük.

1 Formális nyelvtanok és a Chomsky-féle nyelvosztályok

1.1 Alapfogalmak

Ábécé, szó

Szimbólumok véges nemüres halmazát ábécének nevezzük.

Egy V ábécé elemeiből képzett véges sorozatokat V feletti szavaknak vagy sztringeknek nevezzük. A 0 hosszúságú sorozatot üres szónak nevezzük és ε-nal jelöljük.

A V ábécé feletti szavak halmazát (beleértve az üres szót is) V^*-gal, a nemüres szavak halmazát V^+-szal jelöljük.

Konkatenáció

Az $x = uv$ szót az $u, v \in V$ szavak konkatenációjának nevezzük.

A konkatenáció asszociatív, de (általában) nem kommutatív.

V^* zárt a konkatenációra. Azaz:

$$u, v \in V^* \Rightarrow uv \in V^*$$

Továbbá ε egységelemnek tekinthető V^*-gal és a konkatenációval. Azaz:

$$u \in V^* \Rightarrow u\varepsilon \in V^*, \text{ és } \varepsilon u \in V^*$$

Egyéb definíciók

- $u, v \in V$. Az u szót a v részszavának nevezzük, ha $v = xuy$, $(x, y \in V)$ teljesül. Ha még $xy \neq \varepsilon$, akkor u valódi részszó.
- $v = xuy$ $(x, y, u, v \in V)$. Ekkor:
  - Ha $x = \varepsilon$, akkor u-t a v szó prefixének nevezzük
  - Ha $y = \varepsilon$, akkor u-t a v szó szufixének nevezzük
- Egy $u \in V$ szó tükörképe alatt a szimbólumai fordított sorrendben való felírását értjük. (Jele: u^{-1})

1.2 Formális nyelvtanok

Nyelv

Ha $L \subset V^*$, akkor L-et V feletti nyelvnek tekintjük.

Az üres nyelv (egy szót sem tartalmaz) jele: $\emptyset$

L nyelv véges nyelv, ha véges számú szót tartalmaz, különben végtelen nyelv.

Nyelvekre vonatkozó műveletek:

- $L_1 \cup L_2 = \{u |\ u \in L_1 \text{ vagy } u \in L_2\}$: az L_1 és L_2 nyelv uniója

- $L_1 \cap L_2 = \{u|\ u \in L_1 \text{ és } u \in L_2\}$: az L_1 és L_2 nyelv metszete
- $L_1 - L_2 = \{u|\ u \in L_1 \text{ és } u \notin L_2\}$: az L_1 és L_2 nyelv különbsége
- $\overline{L} = V^* - L$: az $L \subseteq V^*$ komplementere
- $L_1L_2 = \{u_1u_2|\ u_1 \in L_1, u_2 \in L_2\}$: az L_1 és L_2 nyelv konkatenációja
  Minden nyelvre fennáll: $\emptyset L = L\emptyset = \emptyset$ illetve $\{\varepsilon\}L = L\{\varepsilon\} = L$
- L^i : az L nyelv i-edik ($i \geq 1$) iterációja (a konkatenációra nézve), és $L^0 = \{\varepsilon\}$
- $L^* = \bigcup_{i \geq 1} L^i$: az L nyelv iteratív lezártja

Az unió, konkatenáció és iteráció lezárása műveleteket reguláris műveleteknek nevezzük.

Grammatika

Nyelvek sokféle módon előállíthatók. A produkciós rendszerekkel való előállítás egyik módja a nyelvek generálása grammatikával.

A G generatív grammatikán egy (N, T, P, S) négyest értünk, ahol:

- N és T diszjunkt ábécék, a nemterminális (N) és terminális (T) szimbólumok ábécéi.
- $S \in N$ a kezdőszimbólum.
- P az (x, y) rendezett párok halmaza, ahol $x, y \in (N \cup T)^*$ és x legalább egy nemterminális szimbólumot tartalmaz.
  A P halmaz elemeit átírási szabályoknak nevezzük.

Az (x, y) jelölés helyett az $x \to y$ jelölést alkalmazzuk. (Ha $\to \notin (N \cup T)$)

Levezetés

$G = (N, T, P, S)$, és $u, v \in (N \cup T)^*$. A v szó közvelenül (egy lépésben) levezethető u szóból G-ben, ha

$$u = u_1xu_2,\ v = u_1yu_2 \text{ és } x \to y \in P \quad (u_1, u_2 \in (N \cup T)^*)$$

Ezt $u \Longrightarrow_G v$ jelöljük.

Azt mondjuk, hogy a v szó k (≥ 1) lépésben levezethető az u szóból G-ben, ha

$$\exists u_1, ..., u_{k+1} \in (N \cup T)^* : u = u_1, v = u_{k+1} \text{ és } u_i \Longrightarrow_G u_{i+1} \quad (1 \leq i \leq k)$$

A v szó levezethető az u szóból G-ben, ha $u = v$ vagy $\exists k \geq 1$ szám, hogy v k lépésben levezethető u-ból.
Másképp: A v szó levezethető az u szóból G-ben (jele: $u \Longrightarrow_G^* v$), ha $u = v$ vagy $\exists z \in (N \cup T)^*$ szó, hogy $u \Longrightarrow_G^* z$ és $z \Longrightarrow_G v$

Generált nyelv

$L(G)$ a $G = (N, T, P, S)$ grammatika által generált nyelv, ha:

$$L(G) = \{w|\ S \Longrightarrow_G^* w, w \in T^*\}$$

1.3 Chomsky-féle hierarchia

$G = (N, T, P, S)$ generatív grammatika i-típusú ($i = 0, 1, 2, 3$), ha P szabályhalmazára a következők teljesülnek:

Mondatszerkezetű grammatika (i=0)

Nincs korlátozás

Környezetfüggő grammatika (i=1)

P minden szabálya $u_1Au_2 \to u_1vu_2$ alakú, ahol, $A \in N$ és $v \neq \varepsilon$, $(u_1, u_2, v \in (N \cup T)^*)$. Kivéve az $S \to \varepsilon$ szabályt (ha létezik). Ekkor S nem fordul elő egyetlen szabály jobboldalán sem.

Környezetfüggetlen grammatika (i=2)

P minden szabálya $A \to v$ alakú, ahol, $A \in N, v \in (N \cup T)^*$.

Reguláris grammatika (i=3)

P minden szabálya $A \to vB$ vagy $A \to v$ alakú, ahol, $A, B \in N, v \in T^*$.

L nyelv i-típusú, ha i-típusú grammatikával generálható. Az i-típusú nyelvek osztályát $\mathcal{L}_i$ jelöljük.
Az i-típusú nyelvosztályok a következő tulajdonságokkal rendelkeznek:

- $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$
- Az $\mathcal{L}_i$ $(i = 0, 1, 2, 3)$ nyelvosztályok mindegyike zárt a reguláris műveletekre.

2 Automaták

Formális nyelvek megadása nemcsak generatív, hanem felismerő eszközökkel is lehetséges, azaz olyan számítási eszközök segítségével, amelyek szavak feldolgozására és azonosítására alkalmasak. Ilyen eszköz például az automata, amely egy szó, mint input hatására kétféleképpen viselkedhet: vagy elfogadja, vagy elutasítja.

2.1 Véges Automata

Definíció

A véges automata egy rendezett ötös,

$$A = (Q, T, \delta, q_0, F)$$

ahol:

- Q - állapotok véges nemüres halmaza
- T - input szimbólumok ábécéje
- $\delta : Q \times T \to Q$ - állapot-átementi függvény
- $q_0 \in Q$ - kezdőállapot
- $F \subseteq Q$ - elfogadó állapotok halmaza

Működés:
A véges automata diszkrét időintervallumokban végrehajtott lépések sorozata által működik. Minden egyes lépés során az automata elolvassa a következő input szimbólumot és átmegy egy olyan állapotba, amelyet az állapotátmeneti függvény meghatároz (az aktuális állapot és input szimbólum alapján).

Kezdetben az A véges automata a q_0 kezdőállapotban van és az olvasófej az input szalagon levő $u \in T^*$ szó első betűjét dolgozza fel. Ezután a véges automata lépések sorozatát végrehajtva elolvassa az input u szót; betűről betűre haladva olvas és új állapotba kerül.

Miután az u input szó utolsó betűjét is elolvasta a véges automata, vagy $q \in F$, azaz elfogadó állapotba kerül, és akkor az u szót az automata elfogadja, vagy az új állapot nem lesz eleme F-nek, és ekkor az automata a szót nem fogadja el.

VDA - Véges determinisztikus automata

A δ függvény egyértékű, ezért minden egyes (q, a) párra, ahol $(q, a) \in Q \times T$ egyetlen olyan s állapot létezik, amelyre $\delta(q, a) = s$ teljesül. Ezért ezt a véges automatát determinisztikusnak nevezzük.

VNDA - Véges nemdeterminisztikus automata

Ha többértékű állapot-átmeneti függvényt is megengedünk, azaz $\delta : Q \times T \to 2^Q$, akkor nemdeterminisztikus véges automatáról beszélünk. (Ebben az esetben aktuális állapotnak egy állapothalmaz valamely elemét, mintsem egyetlen állapotot tekinthetünk.)

Ez azt jelenti, hogy a kezdeti állapot helyettesíthető egy $Q_0 \subseteq Q$ kezdőállapot halmazzal. (És az is előfordulhat, hogy egy a input szimbólum esetén $\delta(q, a)$ üres az aktuális állapotok mindegyikére.)

Tulajdonságok

Az állapot-átmeneteket

$$qa \to p$$

alakú szabályok formájában is írhatjuk $p \in \delta(q, a)$ esetén. Jelöljük M_δ-val az $A = (Q, T, \delta, Q_0, F)$ nemdeterminisztikus véges automata δ állapot-átmenet függvénye által az előbbi módon származó szabályok halmazát.

Ha minden egyes (q, a) párra egyetlen $qa \to p$ szabály van M_δ-ban, akkor a véges automata determinisztikus, egyébként nemdeterminisztikus.

- Közvetlen redukció:
  Legyen $A = (Q, T, \delta, q_0, F)$ egy véges automata és legyenek $u, v \in QT^*$ szavak. Azt mondjuk, hogy az A automata az u szót a v szóra redukálja egy lépésben/közvetlenül. Ha van olyan $qa \to p$ szabály M_δ-ban, és van olyan $w \in T^*$ szó, amelyre $u = qaw$ és $v = pw$ teljesül.
- Redukció:
  Az $A = (Q, T, \delta, q_0, F)$ véges automata az $u \in QT^*$ szót a $v \in QT^*$ szóra redukálja ($u \Longrightarrow_A^* v$), ha $u = v$, vagy $\exists z \in QT^*$, amelyre $u \Longrightarrow_A^* z$ és $z \Longrightarrow_A v$ teljesül.
- Az automata által elfogadott nyelv:
  Az $A = (Q, T, \delta, q_0, F)$ véges automata által elfogadott/felismert nyelv alatt az

  $$L(A) = \{u \in T^* |\ q_0 u \Longrightarrow_A^* p, \quad q_0 \in Q_0 \text{ és } p \in F\}$$

  szavak halmazát értjük. (Az üres szó, akkor és csak akkor van benne az automata által elfogadott $L(A)$ nyelvben, ha $Q_0 \cap F \neq \emptyset$).
- Tétel:
  Minden A nemdeterminisztikus véges automatához meg tudunk adni egy 3-típusú G grammatikát úgy, hogy $L(G) = L(A)$ teljesül.
- Tétel:
  Minden 3-típusú G grammatikához meg tudunk adni egy A véges automatát úgy, hogy $L(A) = L(G)$ teljesül.

  Ezek után fenáll a kérdés: Létezik-e olyan reguláris nyelv, amely VNDA-val felismerhető, de nem ismerhető fel VDA-val?
  Válasz: Nincs.
- Tétel:
  Minden $A = (Q, T, \delta, Q_0, F)$ VNDA-hoz meg tudunk konstruálni egy $A' = (Q', T, \delta', q_0', F')$ VDA-t úgy, hogy $L(A) = L(A')$ teljesül.

2.2 Veremautomata

Definíció

A veremautomata egy rendezett hetes

$$A = (Z, Q, T, \delta, z_0, q_0, F)$$

ahol

- Z - a veremszimbólumok véges halmaza,
- Q - az állapotok véges halmaza,
- T - az inputszimbólumok véges halmaza,
- $\delta : Z \times Q \times (T \cup \{\varepsilon\}) \to 2^{Z^* \times Q}$ - átmeneti függvény
- $z_0 \in Z$ - a kezdeti veremszimbólum,
- $q_0 \in Q$ - a kezdeti állapot,
- $F \subseteq Q$ - az elfogadó állapotok halmaza

A veremautomata konfigurációja alatt egy uq alakú szót értünk, ahol $u \in Z^*$ a verem aktuális tartalma és $q \in Q$ az aktuális állapot.
A kezdeti konfiguráció $z_0 q_0$.

Működés:
Tegyük fel, hogy az A veremautomata olvasófeje az a inputszimbólumon áll, a veremautomata q

állapotban van, valamint a verem tetején levő szimbólum z. Legyen $\delta(z,q,a) = (u_1,r_1),...,(u_n,r_n)$, ahol $u_i \in Z^*$ és $r_i \in Q, 1 \leq i \leq n$. Ekkor A következő állapota valamely r_i lesz és egyidejűleg z-t helyettesíti az u_i szóval, továbbá az olvasófej egy cellával jobbra lép az input szalagon.
Ha $\delta(z,q,\varepsilon)$ nem üres, akkor ún. ε-átmenet hajtható végre.

Ha az input szalag a $w \in T^*$ szót tartalmazza és a z_0q_0 kezdeti konfigurációból kiindulva a lépések sorozatát végrehajtva az A veremautomata egy up konfigurációba ér, ahol p elfogadó állapot, akkor azt mondjuk, hogy A elfogadta a w szó.

Tulajdonságok

- Közvetlen redukció:
  $\alpha, \beta \in Z^*QT^*$
  Az A veremautomata az α szót a β szóra redukálja egy lépésben ($\alpha \Longrightarrow_A \beta$), ha:

  $$\exists z \in Z,\ p,q \in Q,\ a \in T \cup \{\varepsilon\},\ r,u \in Z^* \text{ és } w \in T^*$$

  hogy:

  $$(u,p) \in \delta(z,q,a) \text{ és } \alpha = rzqaw \text{ és } \beta = rupw$$

- Redukció:
  Az A veremautomata az α szót a β szóra redukálja ($\alpha \Longrightarrow_A^* \beta$), ha vagy $\alpha = \beta$, vagy $\exists \gamma_1,...,\gamma_n \in Z^*QT^*$ szavakból álló véges sorozat, hogy $\alpha = \gamma_1,\ \beta = \gamma_n,$ és
  $\gamma_i \Longrightarrow_A \gamma_{i+1} \quad (i = 1,...,n-1)$
- A veremautomata által elfogadott nyelv:
  Az A veremautomata által (elfogadó állapottal) elfogadott nyelv:

  $$L(A) = \{w \in T^* |\ z_0q_0w \Longrightarrow_A^* up, \text{ ahol } u \in Z^*, p \in F\}$$

- Determinizmus:
  A δ leképezést szabályok formájában is megadhatjuk. Az így nyert szabályhalmazt M_δ-val jelöljük. Tehát
  1. $zqa \to up \in M_\delta$ ha $(u,p) \in \delta(z,q,a)$
  2. $zq \to up \in M_\delta$ ha $(u,p) \in \delta(z,q,\varepsilon)$

  Az $A = (Z,Q,T,\delta,z_0,q_0,F)$ veremautomatát determinisztikusnak mondjuk, ha minden $(z,q) \in Z \times Q$ pár esetén:
  1. $\forall a \in T : |\delta(z,q,a)| = 1$ és $\delta(z,q,\varepsilon) = \emptyset$
     vagy
  2. $|(z,q,\varepsilon)| = 1$ és $\forall a \in T : \delta(z,q,a) = \emptyset$
- Üres veremmel elfogadott nyelv:
  Az $N(A)$ nyelvet az A veremautomata üres veremmel fogadja el, ha

  $$N(A) = \{w \in T^* |\ z_0q_0w \Longrightarrow_A^* p, \text{ ahol } p \in Q\}$$

- Tétel:
  Bármely G környezetfüggetlen grammatikához meg tudunk adni egy olyan A veremautomatát, amelyre $L(A) = L(G)$ teljesül.
- Tétel:
  Minden A veremautomatához meg tudunk adni egy környezetfüggetlen G grammatikát úgy, hogy $L(G) = N(A)$ teljesül.

3 Reguláris nyelvek tulajdonságai és alkalmazásai

3.1 3-típusú grammatikák normálformája

Minden 3-típusú, azaz reguláris nyelv generálható egy olyan grammatikával, amelynek szabályai:

- $X \to aY$, ahol $X,Y \in N$ és $a \in T$
- $X \to \varepsilon$, ahol $X \in N$

3.2 Reguláris kifejzések

Motiváció

Ismeretes, hogy minden véges nyelv reguláris. Tudjuk továbbá, hogy az $\mathcal{L}_3$ nyelvosztály (a reguláris nyelvek osztálya) zárt az unió, a konkatenáció és az iteráció lezártja műveletekre nézve.

Következésképpen, kiindulva véges számú véges nyelvből és az előzőekben felsorolt, ún. reguláris műveleteket véges sokszor alkalmazva reguláris nyelvet kapunk.

Kérdés az, hogy vajon ezzel az eljárással minden reguláris nyelvet elő tudunk-e állítani, azaz, ez a módszer elégséges-e az $\mathcal{L}_3$ nyelvosztály leírására?

Definíció

Legyenek V és $V' = \{\varepsilon, \cdot, +, *, (,)\}$ diszjunkt ábécék. A V ábécé feletti reguláris kifejezéseket rekurzív módon a következőképpen definiáljuk:

1. ε reguláris kifejezés V felett.
2. Minden $a \in V$ reguláris kifejezés V felett.
3. Ha R reguláris kifejezés V felett, akkor $(R)^*$ is reguláris kifejezés V felett. [iteratív lezárás]
4. Ha Q és R reguláris kifejezések V felett, akkor $(Q) \cdot (R)$ [konkatenáció] és $(Q) + (R)$ [unió] is reguláris kifejezés V felett.

Megjegyzés: Minden reguláris kifejezés jelöl (meghatároz) valamely reguláris nyelvet. (Pl.: ε a $\{\varepsilon\}$ nyelvet, $a + b$ az $\{a\} \cup \{b\} = \{a, b\}$ és $a \cdot b$ az $\{a\}\{b\} = \{ab\}$ nyelvet.) A reguláris kifejezés a szintaxis, az, hogy hogyan értelmezzük, a szemantika.

Axiómák

P, Q, R reguláris kifejezések. Ekkor fennállnak a következő tulajdonságok:

- Asszociativitás:
$$P + (Q + R) = (P + Q) + R$$
$$P \cdot (Q \cdot R) = (P \cdot Q) \cdot R$$
- Kommutativitás:
$$P + Q = Q + P$$
- Disztributivitás:
$$P \cdot (Q + R) = P \cdot Q + P \cdot R$$
$$(P + Q) \cdot R = P \cdot R + Q \cdot R$$
- Egységelem:
$$\varepsilon \cdot P = P \cdot \varepsilon = P$$
$$P^* = \varepsilon + P \cdot P^*$$
$$P^* = (\varepsilon + P)^*$$

A fenti axiómák azonban még önmagukban nem elegendőek az összes reguláris kifejezés előállítására (helyettesítés segítségével). Szükség van még az alábbi inferencia szabályra:

$$P = R + P \cdot Q \quad \wedge \quad \varepsilon \notin Q \quad \Longrightarrow \quad P = R \cdot Q^*$$

Vegyük még hozzá az $\emptyset$ szimbólumot a reguláris kifejezések halmazához, amely az üres nyelvet jelöli. (Ebben az esetben nincs szükségünk az ε szimbólumra, mivel $\emptyset^* = \{\varepsilon\}$). Így, a definícióban helyettesíthetjük az ε szimbólumot az $\emptyset$ szimbólummal. Ekkor helyettesítjük ε-t a megelőző axióma rendszerben $(\emptyset)^*$-gal és még egy további axiómát tekintünk:

$$\emptyset \cdot P = P \cdot \emptyset = \emptyset$$

A fenti szabályok elégségesek ahhoz, hogy levezessünk minden érvényes egyenlőséget reguláris kifejezések között.

Reguláris kifejezések és reguláris nyelvek

Minden reguláris kifejezés egy reguláris (3-típusú) nyelvet jelöl, és megfordítva, minden reguláris nyelvhez megadható egy, ezen nyelvet jelölő reguláris kifejezés.
(Ezzel választ adtunk a motivációban feltett kérdésre.)

3.3 Lineáris grammatikák és nyelvek

Definíció

Egy $G = (N, T, P, S)$ környezetfüggetlen grammatikát lineárisnak nevezünk, ha minden szabálya:

1. $A \to u, \quad A \in N, u \in T^*$
2. $A \to u_1 B u_2, \quad A, B \in N, u_1, u_2 \in T^*$

Továbbá G-t bal-lineárisnak, illetve jobb-lineárisnak mondjuk, ha $u_1 = \varepsilon$, illetve $u_2 = \varepsilon$ minden 2. alakú szabályra.

Egy L nyelvet lineárisnak, bal-lineárisnak, illetve jobb-lineárisnak mondunk, ha van olyan G lineáris, bal-lineáris, illetve jobb-lineáris grammatika, amelyre $L = L(G)$ teljesül.

Lineáris és reguláris grammatikák, nyelvek

A jobb-lineáris grammatikák azonosak a reguláris grammatikákkal (3-típusúak).

Tétel:
Minden bal-lineáris grammatika reguláris (3-típusú) nyelvet generál.

4 Környezetfüggetlen nyelvek tulajdonságai és elemzésük

4.1 Környezetfüggetlen grammatikák normálformái

Környezetfüggetlen grammatikák normálformái olyan grammatikai transzformációval előállított grammatikák, melyek:

- bizonyos szintaktikai feltételeknek/tulajdonságoknak tesznek eleget
- (általában) valamilyen szempontból egyszerűbbek, mint az eredeti grammatikák
- ugyanazt a nyelvet generálják (így ugyanazon típusba tartoznak)

Tétel:
Minden $G = (N, T, P, S)$ környezetfüggetlen grammatikához meg tudunk konstruálni egy vele ekvivalens $G' = (N', T, P', S')$ környezetfüggetlen grammatikát úgy, hogy: G' minden szabályának jobboldala nemüres szó [kivéve azt az esetet, mikor $\varepsilon \in L(G)$, ekkor $S' \to \varepsilon$ az egyetlen szabály, melynek jobboldala az üres szó és ekkor S' nem fordul elő G' egyetlen szabályának jobboldalán sem.]

ε-mentes grammatika

A G grammatika ε-mentes, ha egyetlen szabályának jobboldala sem az üres szó.

Tétel:
Minden környezetfüggetlen G grammatikához meg tudunk konstruálni egy G' ε-mentes környezetfüggetlen grammatikát, amelyre $L(G') = L(G) - \{\varepsilon\}$ teljesül.

Chomsky normálforma

A $G = (N, T, P, S)$ környezetfüggetlen grammatikát Chomsky-normálformájúnak mondjuk, ha minden egyes szabálya:

- $X \to a$, ahol $X \in N, a \in T$
- $X \to YZ$, ahol $X, Y, Z \in N$

Tétel:
Minden ε-mentes $G = (N, T, P, S)$ környezetfüggetlen grammatikához meg tudunk konstruálni egy vele ekvivalens $G' = (N', T, P', S)$ Chomsky-normálformájú környezetfüggetlen grammatikát.

Tétel (az előző következménye):
Minden G környezetfüggetlen grammatika esetében eldönthető, hogy egy u szó benne van-e G grammatika által generált nyelvben.

Redukált grammatika

- A környezetfüggetlen grammatika egy nemterminálisát inaktívnak/nem aktívnak nevezzük, ha nem vezethető le belőle terminális szó.

- A környezetfüggetlen grammatika egy nemterminálisát nem elérhetőnek nevezzük, ha nem fordul elő egyetlen olyan szóban sem, amely a kezdőszimbólumból levezethető.
- Egy nemterminálist nem hasznosnak mondunk, ha vagy inaktív, és/vagy nem elérhető.
- Az, hogy egy A nemterminális elérhető-e vagy aktív-e, az eldönthető.
- Egy környezetfüggetlen grammatika redukált, ha minden nemterminálisa aktív és elérhető.

Tétel:
Minden környezetfüggetlen grammatikához meg tudunk konstruálni egy vele ekvivalens redukált környezetfüggetlen grammatikát.

4.2 Levezetési fa

A környezetfüggetlen grammatikák levezetéseit fákkal is jellemezhetjük. A levezetési fa egy szó előállításának lehetőségeiről ad információkat. A levezetési fa egy irányított gráf, amely speciális tulajdonságoknak tesz eleget:

- A gyökér cimkéje: S
- A többi csúcs cimkéje ($N \cup T$) valamely eleme

A levezetési fa nem minden esetben adja meg a levezetés során alkalmazott szabályok sorrendjét. Két levezetés lényegében azonos, ha csak a szabályok alkalmazásának sorrendjében különbözik.

Egy környezetfüggetlen grammatika minden levezetési fája egy egyértelmű (egyetlen) legbaloldalibb levezetést határoz meg. A legbaloldalibb levezetés során minden levezetési lépésben a legbaloldalibb nemterminálist kell helyettesítenünk.

Tétel:
Minden környezetfüggetlen grammatikáról eldönthető, hogy az általa generált nyelv az üres nyelv-e vagy sem.

4.3 Bar-Hillel Lemma

Minden L környezetfüggetlen nyelvhez meg tudunk adni két p és q természetes számot úgy, hogy minden olyan szó L-ben, amely hosszabb, mint p

$$uxwyz$$

alakú, ahol $|xwy| \leq q$, $xy \neq \varepsilon$, és minden

$$ux^iwy^iv$$

szó is benne van az L nyelvben minden $i \geq 0$ egész számra ($u, x, w, y, v \in T^*$).

Tétel:
Eldönthető, hogy egy környezetfüggetlen grammatika végtelen nyelvet generál-e vagy sem.

Chapter 12

12

Záróvizsga tételsor
12. Logika és számításelmélet

Ancsin Ádám

Logika és számításelmélet

Ítéletkalkulus és elsőrendű predikátumkalkulus: szintaxis, szemantika, ekvivalens átalakítások, a szemantikus következmény fogalma, rezolúció. – A kiszámíthatóság fogalma és a Church-Turing tézis. A Turing-gép. Rekurzív és rekurzívan felsorolható nyelvek. Eldönthetetlen problémák. Nevezetes idő- és tárbonyolultsági osztályok: P, NP, PSPACE. NP-teljes problémák.

1 Logika

1.1 Alapfogalmak

A logika tárgya az emberi gondolkodási folyamat vizsgálata és helyes gondolkodási formák keresése, illetve létrehozása.

Fogalmak:

1. **Állítás**: Olyan kijelentés, melynek logikai értéke (igaz volta) eldönthető, tetszőleges kontextusban igaz vagy hamis. Azt mondjuk, hogy egy állítás igaz, ha információtartalma megfelel a valóságnak (a tényeknek), és hamis az ellenkező esetben.

   A mindennapi beszédben használt kijelentő mondatok legtöbbször nem állítások, mivel a mondat tartalmába a kontextus is beleszámít: időpont, környezet állapota, általános műveltség bizonyos szintje, stb. (pl. nem állítás az, hogy "ma reggel 8-kor sütött a nap", de állítás pl. az, hogy "minden páros szám osztható 2-vel").

2. **Igazságérték**: Az igazságértékek halmaza $\mathbb{L} = \{igaz, hamis\}$.

3. **Gondolkodási forma**: Gondolkodási forma alatt egy olyan (F, A) párt értünk, ahol A állítás, $F = \{A_1, A_2, ..., A_n\}$ pedig állítások egy halmaza.

   A gondolkodásforma helyes, ha minden esetben, amikor F minden állítása igaz, akkor A is igaz.

1.2 Ítéletkalkulus

1.2.1 Az ítéletlogika szintaxisa

Az ítéletlogika ábécéje

Az ítéletlogika ábécéje $V_0 = V_v \cup \{(,)\} \cup \{\neg, \wedge, \vee, \supset\}$, ahol V_v az ítéletváltozók halmaza. Tehát V_0 az ítéletváltozókat, a zárójeleket, és a logikai műveletek jeleit tartalmazza.

Az ítéletlogika nyelve

Az ítéletlogika nyelve ($\mathcal{L}_0$) ítéletlogikai formulákból áll, amelyek a következőképpen állnak elő:

1. Minden ítéletváltozó ítéletlogikai formula. Ezek az úgynevezett prímformulák (vagy atomi formulák).

2. Ha A ítéletlogikai formula, akkor $\neg A$ is az.

3. Ha A és B ítéletlogikai formulák, akkor $(A \wedge B)$, $(A \vee B)$ és $(A \supset B)$ is ítéletlogikai formulák.

4. Minden ítéletlogikai formula az 1-3. szabályok véges sokszori alkalmazásával áll elő.

Literál: Ha X ítéletváltozó, akkor az X és $\neg X$ formulák literálok, amelyek alapja X.

Közvetlen részformula:

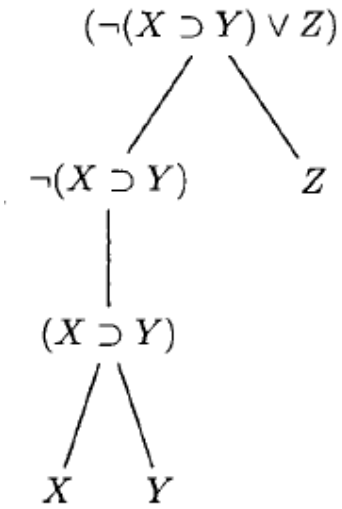
1. Prímformulának nincs közvetlen részformulája.
2. $\neg A$ közvetlen részformulája A.
3. $A \circ B$ ($\circ$ a $\wedge, \vee, \supset$ binér összekötőjelek egyike) közvetlen részformulái A (bal oldali) és B (jobb oldali).

Részformula: Legyen $A \in \mathcal{L}_0$ egy ítéletlogikai formula. Ekkor A részformuláinak halmaza a legszűkebb olyan halmaz, melynek

1. eleme az A, és
2. ha a C formula eleme, akkor C közvetlen részformulái is elemei.

Szerkezeti fa: Egy C formula szerkezeti fája egy olyan véges rendezett fa, melynek csúcsai formulák,

1. gyökere C,
2. a $\neg A$ csúcsának pontosan egy gyermeke van, az A,
3. a $A \circ B$ csúcsának pontosan két gyermeke van, rendre az A és B formulák,
4. levelei prímformulák.



ábra 1: Példa szerkezeti fára.

Logikai összetettség: Egy formula logikai összetettsége a benne található logikai összekötőjelek száma.

Művelet hatásköre: Egy művelet hatásköre a formula részformulái közül az a legkisebb logikai összetettségű részformula, melyben az adott művelet előfordul.

Fő logikai összekötőjel: Egy formula fő logikai összekötőjele az az összekötőjel, amelynek hatásköre maga a formula.

Precedencia: A logikai összekötőjelek precedenciája csökkenő sorrendben a következő: $\neg, \wedge, \vee, \supset$.

A definíciók alapján egyértelmű, hogy egy *teljesen zárójelezett formulában* mi a logikai összekötőjelek hatásköre és mi a fő logikai összekötőjel. Most megmutatjuk, hogy egy formulában milyen esetekben és mely részformulákat határoló zárójelek hagyhatóak el úgy, hogy a logikai összekötőjelek hatásköre ne változzon. A részformulák közül a prímformuláknak és a negációs formuláknak nincs külső zárójelpárja,

ezért csak az $(A \circ B)$ alakú részformulákról kell eldöntenünk, hogy írható-e helyettük $A \circ B$. A zárójelek elhagyását mindig a formula külső zárójelpárjának (ha van ilyen) elhagyásával kezdjük. Majd ha egy részformulában már megvizsgáltuk a külső zárójelelhagyás kérdését, utána ezen részformula közvetlen részformuláinak külső zárójeleivel foglalkozunk. Két eset lehetséges:

1. A részformula egy negációs formula, melyben az $(A \circ B)$ alakú közvetlen részformula külső zárójelei nem hagyhatók el.

2. A részformula egy $(A \bullet B)$ vagy $A \bullet B$ alakú formula, melynek A és B közvetlen részformuláiban kell dönteni a külső zárójelek sorsáról. Ha az A formula $A_1 \circ A_2$ alakú, akkor A külső zárójelpárja akkor hagyható el, ha $\circ$ nagyobb precedenciájú, mint $\bullet$. Ha a B formula $B_1 \circ B_2$ alakú, akkor B külső zárójelpárja akkor hagyható el, ha $\circ$ nagyobb vagy egyenlő precedenciájú, mint $\bullet$.

3. Ha egy $(A \wedge B)$ vagy $A \wedge B$ alakú formula valamely közvetlen részformulája szintén konjunkció, illetve egy $(A \vee B)$ vagy $A \vee B$ alakú formula valamely közvetlen részformulája szintén diszjunkció, akkor az ilyen részformulákból a külső zárójelpár elhagyható.

Formulaláncok: A zárójelek elhagyására vonatkozó megállapodásokat figyelembe véve úgynevezett konjunkciós, diszjunkciós, illetve implikációs formulaláncokat is nyerhetünk. Ezek alakja $A_1 \wedge ... \wedge A_n$, $A_1 \vee ... \vee A_n$, illetve $A_1 \supset ... \supset A_n$ Ezeknek a láncformuláknak a fő logikai összekötőjelét a következő zárójelezési megállapodással fogjuk meghatározni: $(A_1 \wedge (A_2 \wedge ... \wedge (A_{n-1} \wedge A_n)...))$, $(A_1 \vee (A_2 \vee ... \vee (A_{n-1} \vee A_n)...))$, illetve $(A_1 \supset (A_2 \supset ... \supset (A_{n-1} \supset A_n)...))$

1.2.2 Az ítéletlogika szemantikája

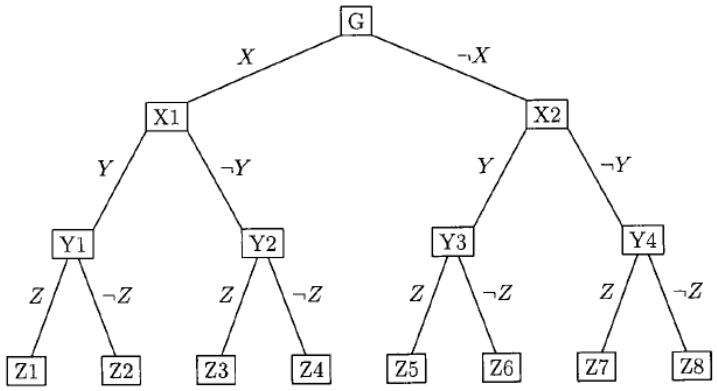
Interpretáció: $\mathcal{L}_0$ interpretációján egy $\mathcal{I} : V_v \to \mathbb{L}$ függvényt értünk, mely minden ítéletváltozóhoz egyértelműen hozzárendel egy igazságértéket.

Boole-értékelés: $\mathcal{L}_0$-beli formulák $\mathcal{I}$ *interpretációbeli Boole-értékelése* a következő $\mathcal{B}_\mathcal{I} : \mathcal{L}_0 \to \mathbb{L}$ függvény:

1. ha A prímformula, akkor $\mathcal{B}_\mathcal{I}(A) = \mathcal{I}(A)$,
2. $\mathcal{B}_\mathcal{I}(\neg A)$ legyen $\neg\mathcal{B}_\mathcal{I}(A)$,
3. $\mathcal{B}_\mathcal{I}(A \wedge B)$ legyen $\mathcal{B}_\mathcal{I}(A) \wedge \mathcal{B}_\mathcal{I}(B)$,
4. $\mathcal{B}_\mathcal{I}(A \vee B)$ legyen $\mathcal{B}_\mathcal{I}(A) \vee \mathcal{B}_\mathcal{I}(B)$,
5. $\mathcal{B}_\mathcal{I}(A \supset B)$ legyen $\mathcal{B}_\mathcal{I}(A) \supset \mathcal{B}_\mathcal{I}(B)$,

Bázis: A formula ítéletváltozóinak egy rögzített sorrendje.

Szemantikus fa: Egy formula különböző interpretációit szemantikus fa segítségével szemléltethetjük. A szemantikus fa egy olyan bináris fa, amelynek i. szintje ($i >= 1$) a bázis i. ítéletváltozójához tartozik, és minden csúcsából két él indul, az egyik a szinthez rendelt ítéletváltozóval, a másik annak negáltjával címkézve. Az X ítéletváltozó esetén az X címke jelentse azt, hogy az X *igaz* az adott interpretációban, a $\neg X$ címke pedig azt, hogy *hamis* az adott interpretációban. A szemantikus fa minden ága egy-egy lehetséges interpretációt reprezentál. Egy n változós formula esetén minden ág n hosszú, és a fának 2^n ága van és az összes lehetséges interpretációt tartalmazza.



ábra 2: Az X,Y,Z ítéletváltozókat tartalmazó formula szemantikus fája.

Igazságtábla: Egy n változós formula igazságtáblája egy $n+1$ oszlopból és 2^n sorból álló táblázat. A táblázat fejlécében az i. oszlophoz ($1 <= i <= n$) a formula bázisának i. ítéletváltozója, az $n+1$. oszlophoz maga a formula van hozzárendelve. Az első n oszlopban az egyes sorokhoz megadjuk rendre a formula különböző interpretációit, majd a formula oszlopába minden sorba beírjuk a formula - a sorhoz tartozó interpretációbeli Boole-értékeléssel kapott - igazságértékét.

A logikai műveletek igazságtáblája:

X	Y	$\neg X$	$X \wedge Y$	$X \vee Y$	$X \supset Y$
i	i	h	i	i	i
i	h	h	h	i	h
h	i	i	h	i	i
h	h	i	h	h	i

Igazhalmaz, hamishalmaz: Egy A formula igazhalmaza (A^i) azon interpretációk halmaza, melyen a formula igazságértékelése igaz. Az A formula hamishalmaza (A^h) pedig azon interpretációk halmaza, melyekre a formula igazságértékelése hamis.

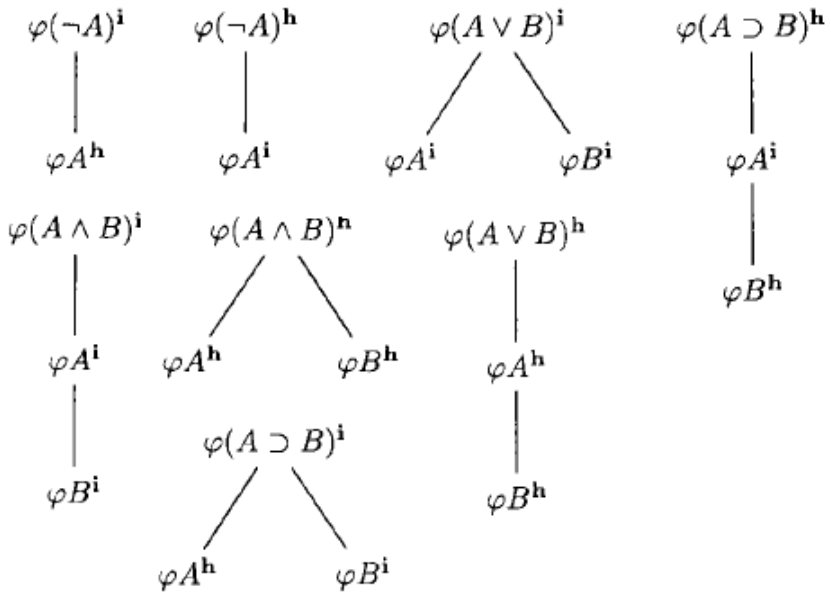
Igazságértékelés függvény: Olyan függvény, amely minden formulához hozzárendeli az igazhalmazát (φA^i) vagy a hamishalmazát (φA^h).

Legyen A egy tetszőleges ítéletlogikai formula. Határozzuk meg A-hoz az interpretációira vonatkozó φA^i, illetve φA^h feltételeket a következőképpen:

1. Ha A prímformula, a φA^i feltételt pontosan azok az $\mathcal{I}$ interpretációk elégítik ki, melyekre $\mathcal{I}(A) = igaz$, a φA^h feltételt pedig pontosan azok melyekre $\mathcal{I}(A) = hamis$.
2. A $\varphi(\neg A)^i$ feltételek pontosan akkor teljesülnek, ha teljesülnek a φA^h feltételek.
3. A $\varphi(A \wedge B)^i$ feltételek pontosan akkor teljesülnek, ha a φA^i és a φB^i feltételek egyszerre teljesülnek.
4. A $\varphi(A \vee B)^i$ feltételek pontosan akkor teljesülnek, ha a φA^i vagy a φB^i feltételek teljesülnek.
5. A $\varphi(A \supset B)^i$ feltételek pontosan akkor teljesülnek, ha a φA^h vagy a φB^i feltételek teljesülnek.

Tétel: Tetszőleges A ítéletlogikai formula esetén a φA^i feltételeket pontosan az A^i-beli interpretációk teljesítik.

Igazságértékelés-fa: Egy A formula φA^i, illetve φA^h feltételeket kielégítő interpretációit az igazságértékelés-fa segítségével szemléltethetjük. Az igazságértékelés-fát a formula szerkezeti fájának felhasználásával állítjuk elő. A gyökérhez hozzárendeljük, hogy A melyik igazságértékre való igazságértékelés-feltételeit keressük, majd a gyökér alá A közvetlen részformulái kerülnek a megfelelő feltétel-előírással, az alábbiak szerint:



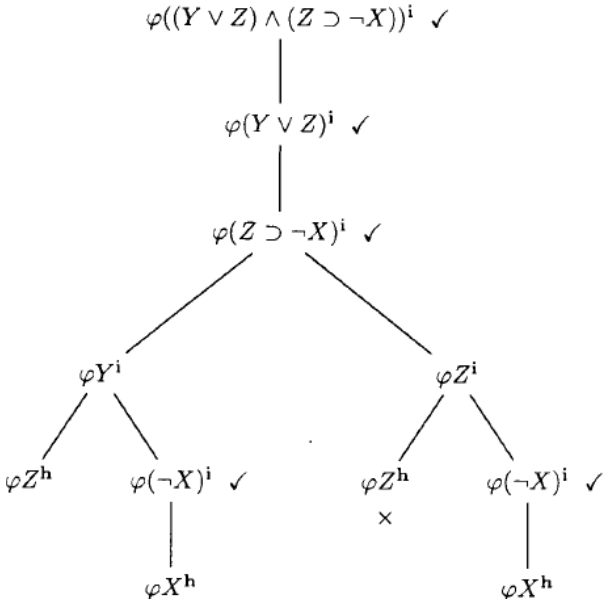
ábra 3: Igazságértékelés-fa feltétel-előírásai.

Ezután a gyökérhez a ✓(feldolgozott) jelet rendeljük. Az eljárást rekurzívan folytatjuk, amíg egy ágon a fel nem dolgozott formulák

- (a) mind ítéletváltozók nem lesznek, vagy
- (b) ugyanarra a formulára egymásnak ellentmondó előírás nem jelenik meg.

Az (a) esetben az ágon előforduló ítéletváltozóknak az ágon rögzített igazságértékeit tartalmazó n-esek mind elemei φA^i gyökér esetén a formula igazhalmazának, φA^h gyökér esetén a formula hamishalmazának.

A (b) esetben nem áll elő ilyen igazságérték n-es.



ábra 4: Az $(Y \vee Z) \wedge (Z \supset \neg X)$ formula igazságértékelés-fája.

A fenti példában a formula igazhalmaza az igazságértékelés-fa alapján: $\{(i,i,h),(h,i,i),(h,i,h),(h,h,i)\}$

Kiterjesztett igazságtábla: Egy igazságtáblában a formula igazságértéke kiszámításának megkönnyítésére vezették be a kiterjesztett igazságtáblát. A kiterjesztett igazságtáblában az ítéletváltozókhoz és a formulához rendelt oszlopokon kívül rendre a formula részformuláihoz tartozó oszlopok is megjelennek. Tulajdonképpen a szerkezeti fában megjelenő részformulák vannak felsorolva.

X	Y	Z	$Y \vee Z$	$\neg X$	$Z \supset \neg X$	$(Y \vee Z) \wedge (Z \supset \neg X)$
i	i	i	i	h	h	h
i	i	h	i	h	i	i
i	h	i	i	h	h	h
i	h	h	h	h	i	h
h	i	i	i	i	i	i
h	i	h	i	i	i	i
h	h	i	i	i	i	i
h	h	h	h	i	i	h

ábra 5: Az $(Y \vee Z) \wedge (Z \supset \neg X)$ formula kiterjesztett igazságtáblája.

Formula kielégíthetősége, modellje: Egy A ítéletlogikai formula *kielégíthető*, ha létezik olyan $\mathcal{I}$ interpretáció, melyre $\mathcal{I} \models_0 A$, azaz a $\mathcal{B}_\mathcal{I}$ Boole-értékelés A-hoz igaz értéket rendel. Egy ilyen interpretációt A *modelljének* nevezünk. Ha A-nak nincs modellje, akkor azt mondjuk, hogy *kielégíthetetlen.*

Ha A igazságtáblájában van olyan sor, amelyben a formula oszlopában igaz érték szerepel, akkor a formula kielégíthető, különben kielégíthetetlen. Ugyanígy, ha φA^i nem üres, akkor kielégíthető, különben kielégíthetetlen.

Ítéletlogikai törvény, tautológia: Egy A ítéletlogikai formula *ítéletlogikai törvény* vagy másképpen *tautológia*, ha $\mathcal{L}_0$ minden interpretációja modellje A-nak. (jelölés: $\models_0 A$)

Eldöntésprobléma: Eldöntésproblémának nevezzük a következő feladatokat:

1. Döntsük el tetszőleges formuláról, hogy tautológia-e!
2. Döntsük el tetszőleges formuláról, hogy kielégíthetetlen-e!

Tautologikusan ekvivalens formulák: Az A és B ítéletlogikai formulák *tautologikusan ekvivalensek* (jelölés: $A \sim_0 B$), ha $\mathcal{L}_0$ minden $\mathcal{I}$ interpretációjában $\mathcal{B}_\mathcal{I}(A) = \mathcal{B}_\mathcal{I}(B)$.

Formulahalmaz kielégíthetősége, modellje: $\mathcal{L}_0$ formuláinak egy tetszőleges Γ halmaza kielégíthető, ha van $\mathcal{L}_0$-nak olyan $\mathcal{I}$ interpretációja, melyre: $\forall A \in \Gamma : \mathcal{I} \models_0 A$. Egy ilyen $\mathcal{I}$ interpretáció modellje Γ-nak. Ha Γ-nak nincs modellje, akkor Γ kielégíthetetlen.

Lemma: Egy $\{A_1, A_2, ..., A_n\}$ formulahalmaznak pontosan azok az $\mathcal{I}$ interpretációk a modelljei, amelyek a $A_1 \wedge A_2 \wedge ... \wedge A_n$ formulának. Következésképpen $\{A_1, A_2, ..., A_n\}$ pontosan akkor kielégíthetetlen, ha az $A_1 \wedge A_2 \wedge ... \wedge A_n$ formula kielégíthetlen.

Szemantikus következmény: Legyen Γ ítéletlogikai formulák tetszőleges halmaza, B egy tetszőleges formula. Azt mondjuk, hogy a B formula *tautologikus következménye* a Γ formulahalmaznak (jelölés: $\Gamma \models_0 B$), ha minden olyan interpretáció, amely modellje Γ-nak, modellje B-nek is. A Γ-beli formulákat feltételformuláknak, vagy premisszáknak, a B formulát következményformulának (konklúziónak) hívjuk.

Tétel: Legyen Γ ítéletlogikai formulák tetszőleges halmaza, A,B,C tetszőleges ítéletlogikai formulák. Ha $\Gamma \models_0 A$, $\Gamma \models_0 B$ és $\{A, B\} \models_0 C$, akkor $\Gamma \models_0 C$.

Tétel: Legyenek $A_1, A_2, ..., A_n, B$ tetszőleges ítéletlogikai formulák. $\{A_1, A_2, ..., A_n\} \models_0 B$ pontosan akkor, ha a $\{A_1, A_2, ..., A_n, \neg B\}$ formulahalmaz kielégíthetetlen, azaz a $A_1 \wedge A_2 \wedge ... \wedge A_n \wedge \neg B$ formula kielégíthetetlen.

Tétel: Legyenek $A_1, A_2, ..., A_n, B$ tetszőleges ítéletlogikai formulák. $\{A_1, A_2, ..., A_n\} \models_0 B$ pontosan akkor, ha $\models_0 A_1 \wedge A_2 \wedge ... \wedge A_n \supset B$.

Ekvivalens átalakítások

Fogalmak:

1. Egy prímformulát (ítéletváltozót), vagy annak a negáltját közös néven *literálnak* nevezünk. A prímformula a *literál alapja*. Egy literált bizonyos esetekben *egységkonjunkciónak* vagy *egységdiszjunkciónak* (*egységklóznak*) is hívunk.

2. *Elemi konjunkció* az egységkonjunkció, illetve a különböző alapú literálok konjunkciója ($\wedge$ kapcsolat a literálok között). *Elemi diszjunkció* vagy *klóz* az egységdiszjunkció és a különböző alapú literálok diszjunkciója ($\vee$ kapcsolat a literálok között). Egy elemi konjunkció, illetve elemi diszjunkció *teljes* egy n-változós logikai műveletre nézve, ha mind az n ítéletváltozó alapja valamely literáljának.

3. *Diszjunktív normálformának* (DNF) nevezzük az elemi konjunkciók diszjunkcióját. *Konjunktív normálformának* (KNF) nevezzük az elemi diszjunkciók konjunkcióját. *Kitüntetett* diszjunktív, illetve konjunktív normálformákról (KDNF, ileltve KKNF) beszélünk, ha a bennük szereplő elemi konjunkciók, illetve elemi diszjunkciók teljesek.

Tetszőleges logikai műveletet leíró KDNF, KKNF előállítása: Legyen $b : \mathbb{L}^n \to \mathbb{L}$ egy n-változós logikai művelet. Adjuk meg b műveletábláját. Az első n oszlop fejlécébe az X_1, X_2, ... X_n ítéletváltozókat írjuk.

A b-t leíró KDNF előállítása:

1. Válasszuk ki azokat a sorokat a műveletáblában, ahol az adott igazságérték n-eshez b *igaz* értéket rendel hozzá. Legyenek ezek a sorok rendre $s_1, s_2, ..s_r$. Minden ilyen sorhoz rendeljünk hozzá egy $X'_1 \wedge X'_2 \wedge ... \wedge X'_n$ teljes elemi konjunkciót úgy, hogy az X'_j literál X_j vagy $\neg X_j$ legyen aszerint, hogy ebben a sorban X_j *igaz* vagy *hamis* igazságérték szerepel. Az így nyert teljes elemi konjunkciók legyenek rendre $k_{s_1}, k_{s_2}, ..k_{s_r}$.

2. Az így kapott teljes elemi konjunkciókból készítsünk egy diszjunkciós láncformulát: $k_{s_1} \vee k_{s_2} \vee ... \vee k_{s_r}$. Ez a formula lesz a b művelet kitüntetett diszjunktív normálformája (KDNF).

X	Y	Z	b		a teljes elemi konjunkciók
i	i	i	h		
i	i	h	i	$\star$	$X \wedge Y \wedge \neg Z$
i	h	i	h		
i	h	h	i	$\star$	$X \wedge \neg Y \wedge \neg Z$
h	i	i	i	$\star$	$\neg X \wedge Y \wedge Z$
h	i	h	h		
h	h	i	i	$\star$	$\neg X \wedge \neg Y \wedge Z$
h	h	h	i	$\star$	$\neg X \wedge \neg Y \wedge \neg Z$

ábra 6: Egy háromváltozós b logikai művelet műveletáblája és az előállított teljes elemi konjunkciók.

A fenti példa b műveletének kitüntetett diszjunktív normálformája a következő formula:
$(X \wedge Y \wedge \neg Z) \vee (X \wedge \neg Y \wedge \neg Z) \vee (\neg X \wedge Y \wedge Z) \vee (\neg X \wedge \neg Y \wedge Z) \vee (\neg X \wedge \neg Y \wedge \neg Z)$.

A b-t leíró KKNF előállítása:

1. Válasszuk ki azokat a sorokat a műveletáblában, ahol az adott igazságérték n-eshez b *hamis* értéket rendel hozzá. Legyenek ezek a sorok rendre $s_1, s_2, ..s_r$. Minden ilyen sorhoz rendeljünk hozzá egy $X'_1 \vee X'_2 \vee ... \vee X'_n$ teljes elemi diszjunkciót úgy, hogy az X'_j literál X_j vagy $\neg X_j$ legyen aszerint, hogy ebben a sorban X_j *hamis* vagy *igaz* igazságérték szerepel. Az így nyert teljes elemi diszjunkciók legyenek rendre $d_{s_1}, d_{s_2}, ..d_{s_r}$.

2. Az így kapott teljes elemi diszjunkciókból készítsünk egy konjunkciós láncformulát: $d_{s_1} \wedge d_{s_2} \wedge ... \wedge d_{s_r}$. Ez a formula lesz a b művelet kitüntetett konjunktív normálformája (KKNF).

X	Y	Z	b		a teljes elemi diszjunkciók
i	i	i	h	$\star$	$\neg X \vee \neg Y \vee \neg Z$
i	i	h	i		
i	h	i	h	$\star$	$\neg X \vee Y \vee \neg Z$
i	h	h	i		
h	i	i	i		
h	i	h	i		
h	h	i	h	$\star$	$X \vee Y \vee \neg Z$
h	h	h	i		

ábra 7: Egy háromváltozós b logikai művelet műveletáblája és az előállított teljes elemi diszjunkciók.

A fenti példa b műveletének kitüntetett konjunktív normálformája a következő formula:
$(\neg X \vee \neg Y \vee \neg Z) \wedge (\neg X \vee Y \vee \neg Z) \wedge (X \vee Y \vee \neg Z)$.

KNF, DNF egyszerűsítése: Egy ítéletlogikai formula logikai összetettségén a formulában szereplő logikai összekötőjelek számát értettük. Ugyanazt a logikai műveletet leíró formulák közül azt tekintjük egyszerűbbnek, amelynek kisebb a logikai összetettsége (azaz kevesebb logikai összekötőjelet tartalmaz).

Legyen X egy ítéletváltozó k egy az X-et nem tartalmazó elemi konjunkció, d egy X-et nem tartalmazó elemi diszjunkció. Ekkor az

- (a) $(X \wedge k) \vee (\neg X \wedge k) \sim_0 k$ és
- (b) $(X \vee d) \wedge (\neg X \vee d) \sim_0 d$

egyszerűsítési szabályok alkalmazásával konjunktív és diszjunktív normálformákat írhatunk át egyszerűbb alakba.

Klasszikus Quine–McCluskey-féle algoritmus KDNF egyszerűsítésére:

1. Soroljuk fel a KDNF-ben szereplő összes teljes elemi konjunkciót az L_0 listában, $j := 0$.
2. Megvizsgáljuk az L_j-ben szereplő összes lehetséges elemi konjunkciópárt, hogy alkalmazható-e rájuk az (a) egyszerűsítési szabály. Ha igen, akkor a két kiválasztott konjunkciót ✓-val megjelöljük, és az eredmény konjunkciót beírjuk a L_{j+1} listába. Azok az elemi konjunkciók, amelyek az L_j vizsgálata során nem lesznek megjelölve, nem voltak egyszerűsíthetők, tehát bekerülnek az egyszerűsített diszjunktív normálformába.
3. Ha az L_{j+1} konjunkciólista nem üres, akkor $j := j + 1$. Hajtsuk végre újból a 2. lépést.
4. Az algoritmus során kapott, de meg nem jelölt elemi konjunkciókból készítsünk egy diszjunkciós láncformulát. Így az eredeti KDNF-el logikailag ekvivalens, egyszerűsített DNF-et kapunk.

Rezolúció

Legyenek $A_1, A_2, ..., A_n, B$ tetszőleges ítéletlogikai formulák. Azt szeretnénk bebizonyítani, hogy $\{A_1, A_2, ..., A_n\} \models_0 B$, ami ekvivalens azzal, hogy $\{A_1, A_2, ..., A_n, \neg B\}$ kielégíthetetlen. Írjuk át ez utóbbi formulahalmaz formuláit KNF alakba! Ekkor a $\{KNF_{A_1}, KNF_{A_2}, ..., KNF_{A_n}, KNF_{\neg B}\}$ formulahalmazt kapjuk, ami pontosan akkor kielégíthetetlen, ha a halmaz formuláiban szereplő klózok halmaza kielégíthetetlen.

A klózokra vonatkozó egyszerűsítési szabály szerint ha X ítéletváltozó, C pedig X-et nem tartalmazó klóz, akkor $(X \vee C) \wedge (\neg X \vee C) \sim_0 C$. Az X és a $\neg X$ egységklózok (azt mondjuk, hogy X és $\neg X$ komplemens literálpár) konjunkciójával ekvivalens egyszerűbb, egyetlen literált sem tartalmazó klóz az üres klóz, melyet a $\square$ jellel jelölünk és definíció szerint minden interpretációban hamis igazságértékű.

Legyenek most C_1 és C_2 olyan klózok, melyek pontosan egy komplemens literálpárt tartalmaznak, azaz $C_1 = C_1' \vee L_1$ és $C_2 = C_2' \vee L_2$, ahol L_1 és L_2 az egyetlen komplemens literálpár (C_1' és C_2' üres klózok is lehetnek). Világos, hogy ha a két klózban a komplemens literálpáron kívül is vannak literálok, és ezek nem mind azonosak, az egyszerűsítési szabály alkalmazhatósági feltétele nem áll fenn.

Tétel: Ha $C_1 = C_1' \vee L_1$ és $C_2 = C_2' \vee L_2$, ahol L_1 és L_2 komplemens literálpár, akkor $\{C_1, C_2\} \models_0 C_1' \vee C_2'$

Rezolvens: Legyenek C_1 és C_2 olyan klózok, melyek pontosan egy komplemens literálpárt tartalmaznak, azaz $C_1 = C_1' \vee L_1$ és $C_2 = C_2' \vee L_2$, ahol L_1 és L_2 a komplemens literálpár, a $C_1' \vee C_2'$ klózt a (C_1, C_2) klózpár (vagy a $C_1 \vee C_2$ formula) *rezolvensének* nevezzük. Ha $C_1 = L_1$ és $C_2 = L_2$ (azaz C_1' és C_2' üres klózok), rezolvensük az üres klóz ($\square$). Az a tevékenység, melynek eredménye a rezolvens, a *rezolválás.*

	klózpár	rezolvens
(a)	$(X \vee Y,\ \neg Y \vee Z)$	$X \vee Z$
(b)	$(X \vee \neg Y,\ \neg Y \vee Z)$	nincs: mindkét azonos alapú literál negált
(c)	$(X \vee \neg Y,\ Z \vee \neg V)$	nincs: nincs azonos alapú literál
(d)	$(\neg X \vee \neg Y,\ X \vee Y \vee Z)$	nincs: két komplemens literálpár van
(e)	$(X,\ \neg X)$	$\square$

ábra 8: Példák klózpárok rezolválhatóságára, rezolvensére.

Tétel: Ha a C klóz a (C_1, C_2) klózpár rezolvense, akkor azon $\mathcal{I}$ interpretációk a $\{C_1, C_2\}$ klózhalmazt nem elégíthetik ki, amelyekben C igazságértéke hamis, azaz $\mathcal{B}_{\mathcal{I}}(C) = hamis$.

Rezolúciós levezetés: Egy S klózhalmazból a C klóz rezolúciós levezetése egy olyan véges $k_1, k_2, ..., k_m (m \geq 1)$ klózsorozat, ahol minden $j = 1, 2, ..., m$-re

1. vagy $k_j \in S$,
2. vagy van olyan $1 \leq s, t \leq j$, hogy k_j a (k_s, k_t) klózpár rezolvense,

és a klózsorozat utolsó tagja, k_m, éppen a C klóz.

Megállapodásunk szerint a rezolúciós kalkulus eldöntésproblémája az, hogy levezethető-e S-ből $\square$. A rezolúciós levezetés célja tehát $\square$ levezetése S-ből. Azt, hogy $\square$ levezethető S-ből, úgy is ki lehet fejezni, hogy létezik S-nek rezolúciós cáfolata.

Példa: Próbáljuk meg levezetni $\square$-t az $S = \{\neg X \vee Y, \neg Y \vee Z, X \vee Z, \neg V \vee Y \vee Z, \neg Z\}$ klózhalmazból. A levezetés bármelyik S-beli klózból indítható.

1.	$\neg V \vee Y \vee Z$	[$\in S$]
2.	$\neg Z$	[$\in S$]
3.	$\neg V \vee Y$	[1, 2 rezolvense]
4.	$\neg Y \vee Z$	[$\in S$]
5.	$\neg Y$	[2, 4 rezolvense]
6.	$\neg V$	[3, 5 rezolvense]
7.	$X \vee V$	[$\in S$]
8.	X	[6, 7 rezolvense]
9.	$\neg X \vee Y$	[$\in S$]
10.	Y	[8, 9 rezolvense]
11.	$\square$	[5, 10 rezolvense]

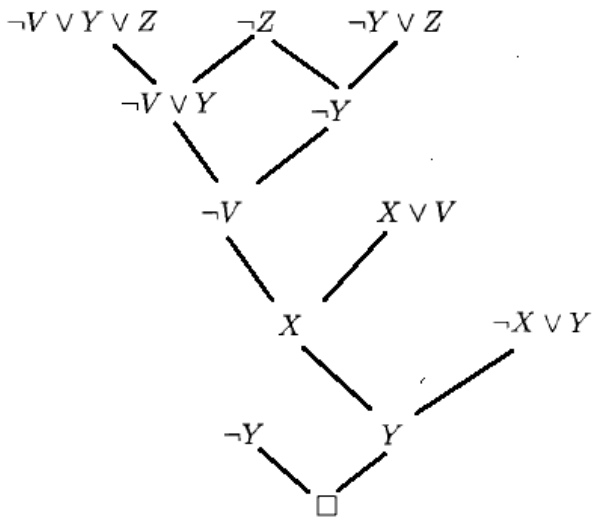
ábra 9: $\square$ rezolúciós levezetése S-ből.

Lemma: Legyen S tetszőleges klózhalmaz. S-ből történő rezolúciós levezetés esetén bármely S-ből levezetett klóz tautologikus következménye S-nek.

A rezolúciós kalkulus helyessége: A rezolúciós kalkulus *helyes*, azaz tetszőleges S klózhalmaz esetén amennyiben S-ből levezethető $\square$, akkor S *kielégíthetetlen.*

A rezolúciós kalkulus teljessége: A rezolúciós kalkulus *teljes*, azaz bármely véges, kielégíthetetlen S klózhalmaz esetén S-ből levezethető $\square$.

Levezetési fa: Egy rezolúciós levezetés szerkezetét *levezetési fa* segítségével szemléltethetjük. A levezetési fa csúcsai klózok. Két csúcsból pontosan akkor vezet él egy harmadik, közös csúcsba, ha az a két klóz rezolvense.



ábra 10: Az előző példa levezetési fája.

Rezolúciós stratégiák:

- *Lineáris rezolúció*: Egy S klózhalmazból való lineáris rezolúciós levezetés egy olyan $k_1, l_1, k_2, l_2, ..., k_{m-1}, l_{m-1}, k_m$ rezolúciós levezetés, amelyben minden $j = 2, 3, ..., m$-re k_j a (k_{j-1}, l_{j-1}) klózpár rezolvense. A k_j klózokat centrális klózoknak, az l_j klózokat mellékklózoknak nevezzük.

  Tetszőleges rezolúciós levezetés átírható lineárissá, azaz a lineáris rezolúciós kalkulus teljes.

- *Lineáris inputrezolúció*: Egy S klózhalmazból való lineáris inputrezolúciós levezetés egy olyan $k_1, l_1, k_2, l_2, ..., k_{m-1}, l_{m-1}, k_m$ lineáris rezolúciós levezetés, amelyben minden $j = 1, 2, ..., m-1$-re $l_j \in S$, azaz a lineáris inputrezolúciós levezetésben a mellékklózok S elemei.

  A lineáris inputrezolúciós stratégia nem teljes, de megadható olyan formulaosztály, melyre az. A legfeljebb egy negált literált tartalmazó klózokat Horn-klózoknak nevezzük, a Horn-formulák pedig azok a formulák, melyek konjunktív normálformája Horn-klózok konjunkciója. A lineáris inputrezolúciós stratégia Horn-formulák esetén teljes.

1.3 Predikátumkalkulus

1.3.1 Elsőrendű logikai nyelvek szintaxisa

Egy elsőrendű logikai nyelv ábécéje logikai és logikán kívüli szimbólumokat, továbbá elválasztójeleket tartalmaz. A logikán kívüli szimbólumhalmaz megadható $< Srt, Pr, Fn, Cnst >$ alakban, ahol:

1. Srt nemüres halmaz, elemei fajtákat szimbolizálnak,
2. Pr nemüres halmaz, elemei predikátumszimbólumok,
3. az Fn halmaz elemei függvényszimbólumok,

4. $Cnst$ pedig a függvényszimbólumok halmaza.

Az $< Srt, Pr, Fn, Cnst >$ ábécé szignatúrája egy $< \nu_1, \nu_2, \nu_3 >$ hármas, ahol

1. minden $P \in Pr$ predikátumszimbólumhoz ν_1 a predikátumszimbólum alakját, azaz a $(\pi_1, \pi_2, ..., \pi_k)$ fajtasorozatot,

2. minden $f \in Fn$ függvényszimbólumhoz ν_2 a függvényszimbólum alakját, azaz a $(\pi_1, \pi_2, ..., \pi_k, \pi)$ fajtasorozatot és

3. minden $c \in Cnst$ konstansszimbólumhoz ν_3 a konstansszimbólumhoz alakját, azaz (π)-t

rendel ($k > 0$ és $\pi_1, \pi_2, ..., \pi_k, \pi \in Srt$).

Logikai jelek az ítéletlogikában is használt logikai összekötőjelek, valamint az univerzális ($\forall$) és egzisztenciális ($\exists$) kvantorok és a különböző fajtájú individuumváltozók. Egy elsőrendű nyelv ábécéjében minden $\pi \in Srt$ fajtához szimbólumoknak megszámlálhatóan végtelen $v_1^\pi, v_2^\pi, ...$ rendszere tartozik, ezeket a szimbólumokat nevezzük π fajtájú változóknak. Elválasztójel a nyitó és csukó zárójelek, és a vessző.

Az elsőrendű logikai nyelvekben az elválasztójelek és a logikai jelek mindig ugyanazok, viszont a logikán kívüli jelek halmaza, illetve ezek szignatúrája nyelvről nyelvre lényegesen különbözhet. Ezért mindig megadjuk a $< Srt, Pr, Fn, Cnst >$ négyest és ennek $< \nu_1, \nu_2, \nu_3 >$ szignatúráját, amikor egy elsőrendű logikai nyelv ábécéjére hivatkozunk. Jelölése $V[V_\nu]$, ahol V_ν adja meg a $< \nu_1, \nu_2, \nu_3 >$ szignatúrájú $< Srt, Pr, Fn, Cnst >$ négyest.

Termek: A $V[V_\nu]$ ábécé feletti termek halmaza $\mathcal{L}_t[V_\nu]$, ami a következő tulajdonságokkal bír:

1. Minden $\pi \in Srt$ fajtájú változó és konstans π fajtájú term.

2. Ha az $f \in Fn$ függvényszimbólum $(\pi_1, \pi_2, ..., \pi_k, \pi)$ alakú és $t_1, t_2, ..., t_k$ – rendre $\pi_1, \pi_2, ..., \pi_k$ fajtájú – termek, akkor az $f(s_1, s_2, ..., s_k)$ egy π fajtájú term.

3. Minden term az 1-2. szabályok véges sokszori alkalmazásával áll elő.

Formulák: A $V[V_\nu]$ ábécé feletti elsőrendű formulák halmaza $\mathcal{L}_f[V_\nu]$, ami a következő tulajdonságokkal bír:

1. Ha a $P \in Pr$ predikátumszimbólum $(\pi_1, \pi_2, ..., \pi_k)$ alakú és az $t_1, t_2, ..., t_k$ – rendre $\pi_1, \pi_2, ..., \pi_k$ fajtájú – termek, akkor a $P(t_1, t_2, ..., t_k)$ szó egy elsőrendű formula. Az így nyert formulákat atomi formuláknak nevezzük.

2. Ha S elsőrendű formula, akkor $\neg S$ is az.

3. Ha S és T elsőrendű formulák és $\circ$ binér logikai összekötőjel, akkor $(S \circ T)$ is elsőrendű formula.

4. Ha S eleme elsőrendű formula, Q kvantor ($\forall$ vagy $\exists$) és x tetszőleges változó, akkor QxS is elsőrendű formula. Az így nyert formulákat kvantált formuláknak nevezzük, a $\forall xS$ alakú formulák univerzálisan kvantált formulák, a $\exists xS$ alakú formulák pedig egzisztenciálisan kvantált formulák. A kvantált formulákban Qx a formula prefixe, S pedig a magja.

5. Minden elsőrendű formula az 1-4. szabályok véges sokszori alkalmazásával áll elő.

A $V[V_\nu]$ ábécé feletti elsőrendű logikai nyelv $\mathcal{L}[V_\nu] = \mathcal{L}_t[V_\nu] \cup \mathcal{L}_f[V_\nu]$, azaz $\mathcal{L}[V_\nu]$ minden szava vagy term, vagy formula.

A negációs, konjunkciós, diszjunkciós, implikációs (ezek jelentése ua., mint nulladrendben) és kvantált formulák összetett formulák.

Az elsőrendű logikai nyelv prímformulái az atomi formulák és a kvantált formulák.

Változóelőfordulás fajtái: Egy formula x változójának egy előfordulása:

- szabad, ha nem esik x-re vonatkozó kvantor hatáskörébe,

- kötött, ha x-re vonatkozó kvantor hatáskörébe esik.

Változó fajtái: Egy formula x változója:

- szabad, ha minden előfordulása szabad,
- kötött, ha minden előfordulása kötött, és
- vegyes, ha van szabad és kötött előfordulása is.

Formula zártsága, nyíltsága: Egy formula:

- zárt, ha minden változója kötött,
- nyílt, ha legalább egy változójának van szabad előfordulása és
- kvantormentes, ha nincs benne kvantor

Megjegyzés: a zárt formulák elsőrendű állításokat szimbolizálnak (egy elsőrendű állítás nem más, mint elemek egy halmazára megfogalmazott kijelentő mondat).

1.3.2 Az elsőrendű logika szemantikája

Matematikai struktúra: Matematikai struktúrán egy $< U, R, M, K >$ négyest értünk, ahol:

1. $U = \bigcup_\pi U_\pi$ nem üres alaphalmaz (univerzum),
2. R az U-n értelmezett logikai függvények (relációk) halmaza,
3. M az U-n értelmezett matematikai függvények (alapműveletek) halmaza,
4. K az U kijelölt elemeinek (konstansainak) halmaza (lehet üres).

Interpretáció: Az interpretáció egy $< U, R, M, K >$ matematikai struktúra és $\mathcal{I} =< \mathcal{I}_{Srt}, \mathcal{I}_{Pr}, \mathcal{I}_{Fn}, \mathcal{I}_{Cnst} >$ függvénynégyes, ahol:

- az $\mathcal{I}_{Srt} : \pi \mapsto U_\pi$ függvény megad minden egyes $\pi \in Srt$ fajtához egy U_π nemüres halmazt, a π fajtájú individuumok halmazát,
- az $\mathcal{I}_{Pr} : P \mapsto P^{\mathcal{I}}$ függvény megad minden $(\pi_1, \pi_2, ..., \pi_k)$ alakú $P \in Pr$ predikátumszimbólumhoz egy $P^{\mathcal{I}} : U_{\pi_1} \times U_{\pi_2} \times ... \times U_{\pi_k} \to \mathbb{L}$ logikai függvényt (relációt),
- az $\mathcal{I}_{Fn} : f \mapsto f^{\mathcal{I}}$ függvény hozzárendel minden $(\pi_1, \pi_2, ..., \pi_k, \pi)$ alakú $f \in Fn$ függvényszimbólumhoz egy $P^{\mathcal{I}} : U_{\pi_1} \times U_{\pi_2} \times ... \times U_{\pi_k} \to U_\pi$ matematikai függvényt (műveletet),
- az $\mathcal{I}_{Cnst} : c \mapsto ct^{\mathcal{I}}$ pedig minden π fajtájú $c \in Cnst$ konstansszimbólumhoz az U_π individuumtartománynak egy individuumát rendeli, azaz $c^{\mathcal{I}} \in U_\pi$.

Változókiértékelés: Legyen az $\mathcal{L}[V_\nu]$ nyelvnek $\mathcal{I}$ egy interpretációja, az interpretáció univerzuma legyen U és jelölje V a nyelv változóinak halmazát. Egy olyan $\kappa : V \to U$ leképezést, ahol ha x π fajtájú változó, akkor $\kappa(x) \in U_\pi$, $\mathcal{I}$-beli változókiértékelésnek nevezünk.

$\mathcal{L}_t[V_\nu]$ szemantikája: Legyen az $\mathcal{L}[V_\nu]$ nyelvnek $\mathcal{I}$ egy interpretációja és κ egy $\mathcal{I}$-beli változókiértékelés. Az $\mathcal{L}[V_\nu]$ nyelv egy π fajtájú t termjének értéke $\mathcal{I}$-ben a κ változókiértékelés mellett az alábbi – $|t|^{\mathcal{I},\kappa}$-val jelölt – U_π-beli individuum:

1. ha $c \in Cnst$ π fajtájú konstansszimbólum, akkor $|c|^{\mathcal{I},\kappa}$ az U_π-beli $c^{\mathcal{I}}$ individuum,
2. ha x π fajtájú változó, akkor $|x|^{\mathcal{I},\kappa}$ az U_π-beli $\kappa(x)$ individuum,
3. ha $t_1, t_2, ..., t_k$ rendre $\pi_1, \pi_2, ..., \pi_k$ fajtájú termek és ezek értékei a κ változókiértékelés mellett rendre az U_{π_1}-beli $|t_1|^{\mathcal{I},\kappa}$, az U_{π_2}-beli $|t_2|^{\mathcal{I},\kappa}$ ... és az U_{π_k}-beli $|t_k|^{\mathcal{I},\kappa}$ individuumok, akkor egy $(\pi_1, \pi_2, ..., \pi_k, \pi)$ alakú $f \in Fn$ függvényszimbólum esetén $|f(t_1, t_2, ..., t_k)|^{\mathcal{I},\kappa}$ az U_π-beli $f^{\mathcal{I}}(|t_1|^{\mathcal{I},\kappa}, |t_2|^{\mathcal{I},\kappa}, ..., |t_k|^{\mathcal{I},\kappa})$ individuum.

Változókiértékelés x-variánsa: Legyen x egy változó. A κ^* változókiértékelés a κ változókiértékelés x-variánsa, ha $\kappa^*(y) = y$ minden x-től különböző y változó esetén.

Elsőrendű logikai formula logikai értéke: Legyen az $\mathcal{L}[V_\nu]$ nyelvnek $\mathcal{I}$ egy interpretációja és κ egy $\mathcal{I}$-beli változókiértékelés. Az $\mathcal{L}[V_\nu]$ nyelv egy C formulájához $\mathcal{I}$-ben a κ változókiértékelés mellett az alábbi – $|C|^{\mathcal{I},\kappa}$-val jelölt – igazságértéket rendeljük:

1. $|P(t_1, t_2, ..., t_k)|^{\mathcal{I},\kappa} = \left\{ \begin{array}{ll} igaz & : P^{\mathcal{I}}(|t_1|^{\mathcal{I},\kappa}, |t_2|^{\mathcal{I},\kappa}, ..., |t_k|^{\mathcal{I},\kappa}) = igaz \\ hamis & : kulonben \end{array} \right\}$
2. $|\neg A|^{\mathcal{I},\kappa}$ legyen $\neg|A|^{\mathcal{I},\kappa}$
3. $|A \wedge B|^{\mathcal{I},\kappa}$ legyen $|A|^{\mathcal{I},\kappa} \wedge |B|^{\mathcal{I},\kappa}$
4. $|A \vee B|^{\mathcal{I},\kappa}$ legyen $|A|^{\mathcal{I},\kappa} \vee |B|^{\mathcal{I},\kappa}$
5. $|A \supset B|^{\mathcal{I},\kappa}$ legyen $|A|^{\mathcal{I},\kappa} \supset |B|^{\mathcal{I},\kappa}$
6. $|\forall x A|^{\mathcal{I},\kappa} = \left\{ \begin{array}{ll} igaz & : |A|^{\mathcal{I},\kappa^*} = igaz\ \kappa\ minden\ \kappa^*\ x - variansara \\ hamis & : kulonben \end{array} \right\}$
7. $|\exists x A|^{\mathcal{I},\kappa} = \left\{ \begin{array}{ll} igaz & : |A|^{\mathcal{I},\kappa^*} = igaz\ \kappa\ valamely\ \kappa^*\ x - variansara \\ hamis & : kulonben \end{array} \right\}$

Elsőrendű formula kielégíthetősége: Egy A elsőrendű formula kielégíthető, ha van olyan $\mathcal{I}$ interpretáció és κ változókiértékelés, amelyre $|A|^{\mathcal{I},\kappa} = igaz$ (ekkor azt mondjuk, hogy az $\mathcal{I}$ interpretáció és κ változókiértékelés kielégíti A-t), különben kielégíthetetlen.

Amennyiben az A formula zárt, igazságértékét egyedül az interpretáció határozza meg. Ha $|A|^{\mathcal{I}} = igaz$, azt mondjuk, hogy az $\mathcal{I}$ kielégíti A-t vagy másképpen: $\mathcal{I}$ modellje A-nak ($\mathcal{I} \models A$).

Logikailag igaz elsőrendű formula: Egy A elsőrendű logikai formula logikailag igaz, ha minden $\mathcal{I}$ interpretációban és $\mathcal{I}$ minden κ változókiértékelése mellett $|A|^{\mathcal{I},\kappa} = igaz$. Jelölése: $\models A$.

Szemantikus következmény: Azt mondjuk, hogy a G formula *szemantikus következménye* az $\mathcal{F}$ formulahalmaznak, ha minden olyan $\mathcal{I}$ interpretációra, amelyre $\mathcal{I} \models \mathcal{F}$ fennáll, $\mathcal{I} \models G$ is igaz (jelölés: $\mathcal{F} \models G$).

Tétel: Legyenek $A_1, A_2, ..., A_n, B$ ($n \geq 1$) tetszőleges, ugyanabból az elsőrendű logikai nyelvből való formulák. Ekkor $\{A_1, A_2, ..., A_n\} \models B$ akkor és csak akkor, ha $A_1 \wedge A_2 \wedge ... \wedge A_n \wedge \neg B$ kielégíthetetlen.

Rezolúció: Elsőrendű predikátumkalkulusban is végezhető rezolúció, ráadásul a módszer helyes és teljes is. Nehézséget a klózok kialakítása okozhat, amelyek zárt, univerzálisan kvantált literálok konjunkciójából állnak. Ehhez eszközeink a prenex-, illetve skolem-formák.

2 Számításelmélet

2.1 Kiszámíthatóság

2.1.1 Algoritmusmodellek

- **Gödel**: rekurzív függvények (primitív rekurzív függvények 1931-ben, majd általánosabb 1934-ben)
- **Church**: λ-kalkulus, λ-definiálható függvények: ekvivalensek a rekurzív függvényekkel (bizonyított)
- **Turing**: Turing-gép (1936), a λ-definiálható és a Turing-géppel kiszámítható függvények megegyeznek (bizonyított)

Church-Turing tézis: A kiszámíthatóság különböző matematikai modelljei mind az effektíven kiszámítható függvények osztályát definiálják.

2.1.2 Fogalmak

Kiszámítási problémának nevezünk egy olyan, a matematika nyelvén megfogalmazott kérdést, amire egy algoritmussal szeretnénk megadni a választ. A gyakorlati élet szinte minden problémájához rendelhető, megfelelő absztrakciót használva, egy kiszámítási probléma.

Egy problémát a hozzá tartozó konkrét bementettel együtt a probléma egy példányának nevezzük.

Speciális kiszámítási probléma az eldöntési probléma. Ilyenkor a problémával kapcsolatos kérdés egy eldöntendő kérdés, tehát a probléma egy példányára a válasz "igen" vagy "nem" lesz.

Egy kiszámítási probléma reprezentálható egy $f : A \to B$ függvénnyel. Az A halmaz tartalmazza a probléma egyes bemeneteit, jellemzően egy megfelelő ábécé feletti szóban elkódolva, míg a B halmaz tartalmazza a bemenetekre adott válaszokat, szintén valamely alkalmas ábécé feletti szóban elkódolva. Értelemszerűen, ha eldöntési problémáról van szó, akkor az f értékkészlete, vagyis a B egy két elemű halmaz: $\{igen, nem\}$, $\{1, 0\}$, stb.

Kiszámítható függvény: Egy $f : A \to B$ függvényt *kiszámíthatónak* nevezünk, ha minden $x \in A$ elemre az $f(x) \in B$ függvényérték kiszámítható valamilyen algoritmikus modellel.

Megoldható, eldönthető probléma: Egy kiszámítási probléma *megoldható* (eldöntési probléma esetén azt mondjuk, hogy *eldönthető*), ha az általa meghatározott függvény kiszámítható.

Algoritmusok időigénye: Legyenek $f, g : \mathbb{N} \to \mathbb{N}$ függvények, ahol $\mathbb{N}$ a természetes számok halmaza. Azt mondjuk, hogy f legfeljebb olyan gyorsan nő, mint g (jelölése: $f(n) = \mathcal{O}(g(n))$), ha $\exists c > 0$ és $n_0 \in \mathbb{N}$, hogy $f(n) \leq c * g(n)\ \forall n \geq n_0$. Az $f(n) = \Omega(g(n))$ jelöli azt, hogy $g(n) = \mathcal{O}(f(n))$ teljesül és $f(n) = \Theta(g(n))$ jelöli azt, hogy $f(n) = \mathcal{O}(g(n))$ és $f(n) = \Omega(g(n))$ is teljesül.

Példa: $3n^3 + 5n^2 + 6 = \mathcal{O}(n^3)$, $n^k = \mathcal{O}(2^n)\ \forall k \geq 0$, stb.

Tétel: Minden polinomiális függvény lassabban nő, mint bármely exponenciális függvény, azaz minden $p(n)$ polinomhoz és $c > 0$-hoz $\exists n_0$ egész szám, hogy $\forall n \geq n_0$ esetén $p(n) \leq 2^{cn}$

Kiszámítási probléma megfeleltetése eldöntési problémának: Tekintsünk egy P kiszámítási problémát és legyen $f : A \to B$ a P által meghatározott függvény. Ekkor megadható P-hez egy P' eldöntési probléma úgy, hogy P' pontosan akkor eldönthető, ha P kiszámítható. Állítsuk párba ugyanis minden $a \in A$ elemre az a és $f(a)$ elemeket, és kódoljuk el az így kapott párokat egy-egy szóban. Ezek után legyen P' az így kapott szavakból képzett formális nyelv. Nyilvánvaló, hogy ha minden $a \in A$ és $b \in B$ elemre az $(a, b) \in P'$ tartalmazás eldönthető (azaz P' eldönthető), akkor P kiszámítható és fordítva. E megfeleltetés miatt a továbbiakban jellemzően eldöntési problémákkal foglalkozunk.

2.2 Turing-gépek

Hasonlóan a véges automatához vagy a veremautomatához, a Turing-gép is egy véges sok állapottal rendelkező eszköz. A Turing-gép egy két irányban végtelen szalagon dolgozik. A szalag cellákra van osztva, tulajdonképpen ez a gép (korlátlan) memóriája. Kezdetben a szalagon csak a bemenő szó van, minden cellán egy betű. A szalag többi cellája egy úgynevezett blank vagy szóköz ($\sqcup$) szimbólumokkal van feltöltve. Kezdetben a gép úgynevezett író-olvasó feje a bemenő szó első betűjén áll és a gép a kezdőállapotában van. A gép az író-olvasó fejet tetszőlegesen képes mozgatni a szalagon. Képes továbbá a fej pozíciójában a szalag tartalmát kiolvasni és átírni. A gépnek van két kitüntetett állapota, a q_i és a q_n állapotok. Ha ezekbe az állapotokba kerül, akkor rendre elfogadja illetve elutasítja a bemenő szót. Formálisan a Turing-gépet a következő módon definiáljuk.

A Turing-gép formális definíciója: A Turing-gép egy olyan $M = (Q, \Sigma, \Gamma, \delta, q_0, q_i, q_n)$ rendszer, ahol:

- Q az állapotok véges, nem üres halmaza,
- $q_0, q_i, q_n \in Q$, q_0 a kezdőállapot, q_i az elfogadó állapot, q_n pedig az elutasító állapot,

- Σ és Γ ábécék, a bemenő jelek és a szalagszimbólumok ábécéje úgy, hogy $\Sigma \subseteq \Gamma$ és $\Gamma - \Sigma$ tartalmaz egy speciális $\sqcup$ szimbólumot,
- $\delta : (Q - \{q_i, q_n\}) \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$ az átmenetfüggvény.

Úgy mint a veremautomaták esetében, egy M Turing-gép működésének fázisait is konfigurációkkal írhatjuk le.

Turing-gép konfigurációja: Az M Turing-gép konfigurációja egy olyan uqv szó, ahol $q \in Q$ és $u, v \in \Gamma^*$, $v \neq \varepsilon$. Ez a konfiguráció az M azon állapotát tükrözi amikor a szalag tartalma uv (uv előtt és után a szalagon már csak $\sqcup$ van), a gép a q állapotban van, és az író-olvasó fej a v első betűjére mutat. M összes konfigurációjának halmazát $\mathcal{C}_M$-el jelöljük.

Turing-gép kezdőkonfigurációja: M kezdőkonfigurációja egy olyan $q_0u\sqcup$ szó, ahol u csak Σ-beli betűket tartalmaz.

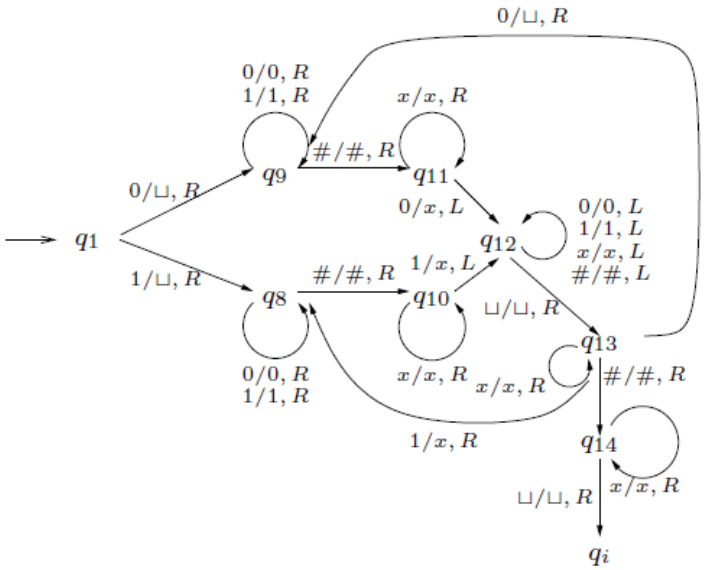
Turing-gép konfigurációátmenete: M konfigurációátmenete egy olyan $\vdash \subseteq \mathcal{C}_M \times \mathcal{C}_M$ reláció, amit a következőképpen definiálunk. Legyen $uqav$ egy konfiguráció, ahol $a \in \Gamma$ és $u, v \in \Gamma^*$. A következő három esetet különböztetjük meg:

1. Ha $\delta(q, a) = (r, b, S)$, akkor $uqav \vdash urbv$.
2. Ha $\delta(q, a) = (r, b, R)$, akkor $uqav \vdash ubrv'$, ahol $v' = v$, ha $v \neq \varepsilon$, különben $v' = \sqcup$.
3. Ha $\delta(q, a) = (r, b, L)$, akkor $uqav \vdash u'rcbv$, ahol $u'c = u$ valamely $u' \in \Gamma^*$-ra és $c \in \Gamma$-ra, ha $u \neq \varepsilon$, egyébként pedig $u' = \varepsilon$, $c = \sqcup$.

Azt mondjuk, hogy M véges sok lépésben eljut a C konfigurációból a C' konfigurációba (jele $C \vdash^* C'$), ha létezik olyan $n \geq 0$ és $C_1, ...C_n$ konfigurációsorozat, hogy $C_1 = C$, $C_n = C'$ és minden $1 \leq i < n$-re $C_i \vdash C_{i+1}$.

Ha $q \in \{q_i, q_n\}$, akkor azt mondjuk, hogy az uqv konfiguráció egy megállási konfiguráció. Továbbá, $q = q_i$ esetében elfogadó, míg $q = q_n$ esetében elutasító konfigurációról beszélünk.

Turing-gép által felismert nyelv: Az M Turing-gép által felismert nyelv (jelölése $L(M)$) azoknak az $u \in \Sigma^*$ szavaknak a halmaza, melyekre igaz, hogy $q_0u\sqcup \vdash^* xq_iy$ valamely $x, y \in \Gamma^*$, $y \neq \varepsilon$ szavakra.



ábra 11: Egy, az $L = \left\{u\#u \mid u \in \{0,1\}^+\right\}$ felismerő Turing-gép.

Turing-gépek ekvivalenciája: Két Turing-gépet ekvivalensnek nevezünk, ha ugyanazt a nyelvet ismerik fel.

Turing-felismerhető nyelv, rekurzívan felismerhető nyelvek osztálya: Egy $L \subseteq \Sigma^*$ nyelv Turing-felismerhető, ha $L = L(M)$ valamely M Turing-gépre. A Turing-felismerhető nyelveket szokás *rekurzívan felsorolhatónak* is nevezni. A rekurzívan felsorolható nyelvek osztályát RE-vel jelöljük.

Turing-eldönthető nyelv, rekurzív nyelvek osztálya: Egy $L \subseteq \Sigma^*$ nyelv Turing-eldönthető, ha létezik olyan Turing-gép, amely minden bemeneten megállási konfigurációba jut és felismeri L-et. A Turing-felismerhető nyelveket szokás *rekurzívnak* is nevezni. A rekurzív nyelvek osztályát R-rel jelöljük.

Turing-gép futási ideje, időigénye: Tekintsünk egy $M = (Q, \Sigma, \Gamma, \delta, q_0, q_i, q_n)$ Turing-gépet és annak egy $u \in \Sigma^*$ bemenő szavát. Azt mondjuk, hogy M futási ideje (időigénye) az u szón n ($n \geq 0$), ha M a $q_0 u \sqcup$ kezdőkonfigurációból n lépésben el tud jutni egy megállási konfigurációba. Ha nincs ilyen szám, akkor M futási ideje az u szón végtelen.

Legyen $f : \mathbb{N} \to \mathbb{N}$ egy függvény. Azt mondjuk, hogy M időigénye $f(n)$ (vagy azt, hogy M egy $f(n)$ időkorlátos gép), ha minden $u \in \Sigma^*$ input szóra M időigénye az u szón legfeljebb $f(l(u))$.

2.2.1 Többszalagos Turing-gépek

A többszalagos Turing-gépek, értelemszerűen, egynél több szalaggal rendelkeznek. Mindegyik szalaghoz tartozik egy-egy író-olvasó fej, melyek egymástól függetlenül képesek mozogni a szalagon.

Többszalagos Turing-gép definíciója: Legyen $k > 1$. Egy k-szalagos Turing-gép egy olyan $M = (Q, \Sigma, \Gamma, \delta, q_0, q_i, q_n)$ rendszer, ahol a komponensek a δ kivételével megegyeznek az egyszalagos Turing-gép komponenseivel, δ pedig a következőképpen adódik. $\delta : (Q - \{q_i, q_n\}) \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$. Legyenek $q, p \in Q$, $a_1, a_2, ..., a_k, b_1, b_2, ..., b_k \in \Gamma$ és $D_1, D_2, ..., D_k \in \{L, R, S\}$. Ha $\delta(q, a_1, a_2, ..., a_k) = (p, b_1, b_2, ..., b_k, D_1, D_2, ..., D_k)$, akkor a gép akkor a gép a q állapotból, ha a szalagjain rendre az $a_1, a_2, ..., a_k$ betűket olvassa, át tud menni a p állapotba, miközben az $a_1, a_2, ..., a_k$ betűket átírja a $b_1, b_2, ..., b_k$ betűkre és a szalagokon a fejeket $D_1, D_2, ..., D_k$ irányokba mozgatja.

A többszalagos Turing-gép konfigurációja, a konfigurációátmenet valamint a felismert illetve eldöntött nyelv definíciója az egyszalagos eset értelemszerű általánosítása. A többszalagos Turing-gép időigényét is az egyszalagoshoz hasonlóan definiáljuk.

Többszalagos és egyszalagos gépek ekvivalenciája: Minden k-szalagos, $f(n)$ időkorlátos Turing-géphez van vele ekvivalens $\mathcal{O}(n * f(n))$ időkorlátos egyszalagos Turing-gép.

2.2.2 Nemdeterminisztikus Turing-gépek

Egy M nemdeterminisztikus Turing-gép állapotfüggvénye $\delta : (Q - \{q_i, q_n\}) \times \mathcal{P}(\Gamma \to Q \times \Gamma \times \{L, R\})$ alakú. Tehát M minden konfigurációjából néhány (esetleg nulla) különböző konfigurációba mehet át. Ily módon M számítási sorozatai egy u szón egy fával reprezentálhatók. A fa csúcsa M kezdőkonfigurációja, a szögpontjai pedig M konfigurációi. A fa minden levele megfelel M egy számítási sorozatának az u-n. M akkor fogadja el u-t, ha a fa valamelyik levele elfogadó konfiguráció. Nevezzük ezt a most leírt fát az M nemdeterminisztikus számítási fájának az u-n. Az M által felismert nyelv a determinisztikus esethez hasonlóan definiálható, a gép által eldöntött nyelv pedig a következőképpen.

Nemdeterminisztikus Turing-gép által eldöntött nyelv: Azt mondjuk, hogy egy nemdeterminisztikus M Turing-gép eldönt egy $L \subseteq \Gamma^*$ nyelvet, ha felismeri, és minden $u \in \Sigma*$ szóra M számítási sorozatai végesek és elfogadási vagy elutasítási konfigurációba vezetnek.

Nemdeterminisztikus Turing-gép időigénye: Legyen $f : \mathbb{N} \to \mathbb{N}$ függvény, M egy nemdeterminisztikus Turing-gép. Az M időigénye $f(n)$, ha egy n hosszú u bemeneten nincsenek M-nek $f(n)$-nél hosszabb számítási sorozatai, azaz az M számítási fája az u-n legfeljebb $f(n)$ magas.

Determinisztikus és nemdeterminisztikus Turing-gépek ekvivalenciája: Minden M nemdeterminisztikus Turing-géphez megadható egy ekvivalens M' determinisztikus Turing-gép. Továbbá, ha M $f(n)$ időigényű valamely $f : \mathbb{N} \to \mathbb{N}$ függvényre, akkor M' $2^{\mathcal{O}(f(n))}$ időigényű.

2.3 Eldönthetetlen problémák

Ebben a fejezetben megmutatjuk, hogy bár a Turing-gép a lehető legáltalánosabb algoritmus modell, mégis vannak olyan problémák, melyek nem számíthatók ki Turing-géppel.

Emlékeztető: A rekurzívan felsorolható (Turing-felismerhető) nyelvek osztályát RE-vel, a rekurzív (Turing-eldönthető) nyelvek osztályát R-rel jelöljük.

Világos, hogy $R \subseteq RE$. A célunk az, hogy megmutassuk: az R valódi részhalmaza az RE-nek, azaz van olyan nyelv (probléma) ami Turing-felismerhető, de nem eldönthető.

Csak olyan Turing-gépeket fogunk vizsgálni, melyek bemenő ábécéje a $\{0,1\}$ halmaz. Ez nem jelenti az általánosság megszorítását, hiszen ha találunk egy olyan $\{0,1\}$ feletti nyelvet, melyet nem lehet eldönteni ilyen Turing-géppel, akkor ezt a nyelvet egyáltalán nem lehet eldönteni.

2.3.1 Turing-gépek kódolása

A $\{0,1\}$ feletti szavak felsorolhatóak (vagyis megszámlálhatóak). Valóban, tekintsük azt a felsorolást, amelyben a szavak a hosszuk szerint követik egymást, és két egyforma hosszú szó közül pedig az van előbb, amelyik az alfabetikus rendezés szerint megelőzi a másikat. Ily módon a $\{0,1\}^*$ halmaz elemeinek egy felsorolása a következőképpen alakul: $w_1 = \varepsilon$, $w_2 = 0$, $w_3 = 1$, $w_4 = 00$, $w_5 = 01$ és így tovább. Ebben a fejezetben tehát a w_i szóval a $\{0,1\}^*$ i. elemét jelöljük.

Legyen továbbá M egy $\{0,1\}$ inputábécé feletti Turing-gép. Van olyan $k > 0$ szám, hogy Q-t felírhatjuk $Q = \{p_1, ...p_k\}$ alakban, ahol $p_1 = q_0$, $p_{k-1} = q_i$, $p_k = q_n$. Továbbá, van olyan $m > 0$ szám, hogy Γ-t felírhatjuk $\Gamma = \{X_1, ...X_m\}$ alakban, ahol $X_1 = 0$, $X_2 = 1$, $X_3 = \sqcup$, és $X_4, ...X_m$ az M további szalagszimbólumai. Nevezzük végül az L, R, S szimbólumokat (amelyek irányokat jelölnek) rendre D_1, D_2 és D_3-nak. Ezek után M egy $\delta(p_i, X_j) = (p_r, X_s, D_t)$ $(0 \leq i, r \leq k,\ 1 \leq j, s \leq m$ és $1 \leq t \leq 3)$ átmenete elkódolható a $0^i10^j10^r10^s10^t$ szóval. Mivel minden 0-s blokk hossza legalább 1, az átmenetet kódoló szóban nem szerepel az 11 részszó. Tehát az M összes átmenetét kódoló szavakat összefűzhetjük egy olyan szóvá, melyben az átmeneteket az 11 részszó választja el egymástól. Az így kapott szó pedig magát M-et kódolja.

A továbbiakban M_i-vel jelöljük azt a Turing-gépet, amelyet a w_i szó kódol $(i \geq 1)$. Amennyiben w_i nem a fent leírt kódolása egy Turing-gépnek, akkor tekintsük M_i-t olyannak, ami minden input esetén azonnal a q_n állapotba megy, azaz $L(M_i) = \emptyset$.

A későbbiekben szükségünk lesz arra, hogy elkódoljunk egy (M, w) Turing-gép és bemenet párost egy $\{0,1\}$ feletti szóban. Mivel a Turing-gépek kódolása nem tartalmazhat 111-et, ezért (M, w) kódja a következő: M kódja után írunk 111-et, majd utána w-t.

2.3.2 Egy nem rekurzívan felsorolható nyelv

Az $L_{átló}$ nyelv: Az $L_{átló}$ nyelv azon $\{0,1\}$ feletti Turing-gépek bináris kódjait tartalmazza, melyek nem fogadják el önmaguk kódját, mint bemenő szót, azaz $L_{átló} = \{w_i \mid i \geq 1, w_i \notin L(M_i)\}$

Tétel: $L_{átló} \notin RE$.

2.3.3 Egy rekurzívan felsorolható, de nem eldönthető nyelv

Az L_u nyelv: Tekintsük azon (M, w) párok halmazát (egy megfelelő bináris szóban elkódolva), ahol M egy $\{0,1\}$ bemenő ábécé feletti Turing-gép, w pedig egy $\{0,1\}$ feletti szó úgy, hogy $w \in L(M)$, azaz M elfogadja w-t. Ezt a nyelvet jelöljük L_u-val. $L_u = \{\langle w_i, w_j \rangle \mid i, j \geq 1, w_j \in L(M_i)\}$

Tétel: $L_u \in RE$.

Tétel: $L_u \notin R$.

2.3.4 További tételek

1. Legyen L egy nyelv. Ha $L, \bar{L} \in RE$, akkor $L \in R$. Következmény: a rekurzívan felsorolható nyelvek nem zártak a komplementerképzésre.
2. Ha $L \in R$, akkor $\bar{L} \in R$, azaz a rekurzív nyelvek zártak a komplementerképzésre.

2.3.5 További eldönthetetlen problémák

Kiszámítható függvény: Legyen Σ és Δ két ábécé és f Σ^*ból Δ^*-ba képző függvény. Azt mondjuk, hogy f kiszámítható, ha van olyan M Turing-gép, hogy M-et egy $w \in \Sigma^*$ szóval a bemenetén elindítva, M úgy áll meg, hogy a szalagján a $f(w) \in \Delta^*$ szó van.

Eldöntési problémák visszavezetése: Legyen $L_1 \subseteq \Sigma^*$ és $L_2 \subseteq \Delta^*$ két eldöntési probléma. L_1 visszavezethető L_2-re ($L_1 \leq L_2$), ha van olyan $f : \Sigma^* \to \Delta^*$ kiszámítható függvény, hogy minden $w \in \Sigma^*$ szóra $w \in L_1$ pontosan akkor teljesül, ha $f(w) \in L_2$ is teljesül.

Tétel: Legyen $L_1 \subseteq \Sigma^*$ és $L_2 \subseteq \Delta^*$ két eldöntési probléma és tegyük fel, hogy L_1 visszavezethető L_2-re. Ekkor igazak a következő állítások:

1. Ha L_1 eldönthetetlen, akkor L_2 is.
2. Ha $L_1 \notin RE$, akkor $L_2 \notin RE$.

A megállási probléma: Legyen $L_h = \{\langle M, w\rangle \mid M \textit{ megáll a } w \textit{ bemeneten}\}$, azaz L_h azon $\langle M, w\rangle$ Turing-gép és bemenet párosokat tartalmazza elkódolva, melyekre M megáll a w bemeneten. L_h eldönthetetlen (L_u visszavezethető L_h-ra), viszont $L_h \in RE$.

Az $L_{üres}$ probléma: Legyen $L_{üres} = \{\langle M\rangle \mid L(M) = \emptyset\}$. $L_{üres}$ eldönthetetlen (L_u visszavezethető $L_{üres}$-re), valamint $L_{üres} \notin RE$.

Rekurzívan felsorolható nyelvek (nem triviális) tulajdonsága: Ha $\mathcal{P}$ a rekurzívan felsorolható nyelvek egy halmaza, akkor $\mathcal{P}$ a rekurzívan felsorolható nyelvek egy tulajdonsága. Ha $\mathcal{P} \neq \emptyset$ és $\mathcal{P} \neq RE$, akkor $\mathcal{P}$ nem triviális tulajdonsága a rekurzívan felsorolható nyelveknek.

Rice tétele: Adott $\mathcal{P}$ tulajdonságra jelöljük $L_{\mathcal{P}}$-vel azon Turing-gépek kódjainak halmazát, amelyek $\mathcal{P}$-beli nyelvet ismernek fel. Ha $\mathcal{P}$ a rekurzívan felsorolható nyelvek egy nem triviális tulajdonsága, akkor $L_{\mathcal{P}}$ eldönthetetlen.

Post Megfelelkezési Probléma (röviden PMP): A PMP problémát a következőképpen definiáljuk. Legyen Σ egy legalább két betűt tartalmazó ábécé és legyen $D = \left\{[\frac{u_1}{v_1}], ..., [\frac{u_n}{v_n}]\right\}$ egy dominóhalmaz, melyben $n \geq 1$ és $u_1, ..., u_n, v_1, ..., v_n \in \Sigma^+$. A kérdés az, hogy van-e egy olyan $1 \leq i_1, ..., i_m \leq m$ ($m \geq 1$) indexsorozat, melyre teljesül, hogy a $[\frac{u_{i_1}}{v_{i_1}}], ..., [\frac{u_{i_m}}{v_{i_m}}]$ dominókat egymás mellé írva alul és felül ugyanaz a szó adódik, azaz $u_{i_1}...u_{i_m} = v_{i_1}...v_{i_m}$. Ebben az esetben a fenti dominósorozatot a D egy megoldásának nevezzük.

Formális nyelvként a következőképpen definiálhatjuk a PMP-t: PMP $= \{\langle D\rangle \mid D - \textit{nek van megoldása}\}$. PMP eldönthetetlen.

2.4 Bonyolultságelmélet

A bonyolultságelmélet célja a megoldható (és ezen belül az eldönthető) problémák osztályozása a megoldáshoz szükséges erőforrások (jellemzően az idő és a tár) mennyisége szerint.

2.4.1 Időbonyolultsági fogalmak

TIME: Legyen $f : \mathbb{N} \to \mathbb{N}$ függvény. $\mathbf{TIME}(f(n)) = \{L \mid L \textit{ eldönthetõ } \mathcal{O}(f(n)) \textit{ idõigényû Turing} - \textit{géppel}\}$

$\mathbf{P} = \bigcup_{k\geq 1} \mathbf{TIME}(n^k)$. Tehát **P** azon nyelveket tartalmazza, melyek eldönthetőek polinom időkorlátos determinisztikus Turing-géppel. Ilyen például a jól ismert ELÉRHETŐSÉG probléma, melynek bemenete egy G gráf és annak két kitüntetett csúcsa (s és t). A kérdés az, hogy van-e a G-ben út s-ből t-be. Ha

az ELÉRHETŐSÉG problémára nyelvként tekintünk, akkor írhatjuk azt, hogy

$$\text{ELÉRHETŐSÉG} = \{\langle G, s, t\rangle \mid G - ben\ van\ út\ s - bõl\ t - be\}.$$

Könnyen megadható az ELÉRHETŐSÉG problémáját polinom időben eldöntő determinisztikus Turing-gép, tehát ELÉRHETŐSÉG $\in \mathbf{P}$.

NTIME: Legyen $f : \mathbb{N} \to \mathbb{N}$ függvény.
$\mathbf{NTIME}(f(n)) = \{L \mid L\ eldönthetõ\ \mathcal{O}(f(n))\ idõigényû\ nemdeterminisztikus\ Turing - géppel\}$

$\mathbf{NP} = \bigcup_{k\geq 1} \mathbf{NTIME}(n^k)$. Az **NP**-beli problémák rendelkeznek egy közös tulajdonsággal az alábbi értelemben. Ha tekintjük egy **NP**-beli probléma egy példányát és egy lehetséges "bizonyítékot" arra nézve, hogy ez a példány "igen" példánya az adott problémának, akkor ezen bizonyíték helyességének leellenőrzése polinom időben elvégezhető. Ennek megfelelően egy **NP**-beli problémát eldöntő nemdeterminisztikus Turing-gép általában úgy működik, hogy "megsejti" a probléma bemenetének egy lehetséges megoldását, és polinom időben leellenőrzi, hogy a megoldás helyes-e.

Tekintsük a SAT problémát, amit a következőképpen definiálunk. Adott egy ϕ ítéletlogikai KNF. A kérdés az, hogy kielégíthető-e. Annak a bizonyítéka, hogy a ϕ kielégíthető, egy olyan változó-hozzárendelés, ami mellett kiértékelve a ϕ-t igaz értéket kapunk. Egy tetszőleges változó-hozzárendelés tehát a ϕ kielégíthetőségének egy lehetséges bizonyítéka .Annak leellenőrzése pedig, hogy ez a hozzárendelés tényleg igazzá teszi-e ϕ-t, polinom időben elvégezhető. A SAT **NP**-beli probléma.

Az a definíciókból következik, hogy fennáll a $\mathbf{P} \subseteq \mathbf{NP}$ tartalmazás.

2.4.2 NP-teljes problémák

Polinom időben kiszámítható függvény: Legyen Σ és Δ két ábécé és f Σ^*ból Δ^*-ba képző függvény. Azt mondjuk, hogy f polinom időben kiszámítható, ha kiszámítható egy polinom időigényű Turing-géppel.

Eldöntési problémák polinom idejű visszavezetése: Legyen $L_1 \subseteq \Sigma^*$ és $L_2 \subseteq \Delta^*$ két eldöntési probléma. L_1 polinom időben visszavezethető L_2-re ($L_1 \leq_p L_2$), ha $L_1 \leq L_2$ és a visszavezetésben használt f függvény polinom időben kiszámítható.

Tétel: Legyen L_1 és L_2 két probléma úgy, hogy $L_1 \leq_p L_2$. Ha L_2

1. **P**-beli, akkor L_1 is **P**-beli.
2. **NP**-beli, akkor L_1 is **NP**-beli.

NP-teljes probléma: Legyen L egy probléma. Azt mondjuk, hogy L **NP**-teljes, ha

1. **NP**-beli, és
2. minden további **NP**-beli probléma polinom időben visszavezethető L-re.

Tétel: Legyen L egy **NP**-teljes probléma. Ha $L \in \mathbf{P}$, akkor $\mathbf{P} = \mathbf{NP}$.

Megjegyzés: Jelenleg **NEM** tudunk **P**-beli **NP**-teljes problémáról!!!

Tétel: Legyen L_1 egy **NP**-teljes, L_2 pedig **NP**-beli probléma. Ha $L_1 \leq_p L_2$, akkor L_2 is **NP**-teljes.

Cooke tétele: SAT **NP**-teljes.

Legyen $k \geq 1$. kSAT $= \{\langle\phi\rangle \mid \phi\ minden\ tagjában\ k\ literál\ van.\}$

Tétel: 3SAT **NP**-teljes, ugyanis SAT $\leq_p$ 3SAT.

TELJES RÉSZGRÁF $= \{\langle G, k\rangle \mid G\ véges\ gráf, k \geq 1,\ G - nek\ \exists\ k\ csúcsú\ részgráfja\}$. Tehát a TELJES RÉSZGRÁF azon G és k párokat tartalmazza, megfelelő ábécé feletti szavakban elkódolva, melyekre igaz,

hogy G-ben van k csúcsú teljes részgráf, azaz olyan részgráf, melyben bármely két csúcs között van él.

TELJES RÉSZGRÁF= $\{\langle G,k\rangle \mid G\ véges\ gráf, k \geq 1,\ G-nek\ \exists\ k\ csúcsú\ részgráfja\}$. Tehát a TELJES RÉSZGRÁF azon G és k párokat tartalmazza, megfelelő ábécé feletti szavakban elkódolva, melyekre igaz, hogy G-ben van k csúcsú teljes részgráf, azaz olyan részgráf, melyben bármely két csúcs között van él.

FÜGGETLEN CSÚCSHALMAZ = $\{\langle G,k\rangle \mid G\ véges\ gráf, k \geq 1,\ G-nek\ \exists\ k\ elemű\ független\ csúcshalmaza\}$. Vagyis a FÜGGETLEN CSÚCSHALMAZ azon G és k párokat tartalmazza, melyekre igaz, hogy G-ben van k olyan csúcs, melyek közül egyik sincs összekötve a másikkal.

CSÚCSLEFEDÉS =
$$\left\{\langle G,k\rangle \mid \begin{array}{l} G\ véges\ gráf, k \geq 1,\ G-nek\ van\ olyan\ k\ elemű\ csúcshalmaza, \\ mely\ tartalmazza\ G\ minden\ élének\ legalább\ 1\ végpontját. \end{array}\right\}.$$

TELJES RÉSZGRÁF, FÜGGETLEN CSÚCSHALMAZ és CSÚCSLEFEDÉS **NP**-teljesek (TELJES RÉSZGRÁF $\leq_p$ FÜGGETLEN CSÚCSHALMAZ $\leq_p$ CSÚCSLEFEDÉS).

UTAZÓÜGYNÖK =
$$\left\{\langle G,k\rangle \mid \begin{array}{l} G\ véges\ irányítatlan\ gráf,\ az\ éleken\ egy-egy\ pozitív\ egész\ súllyal\ és \\ van\ G-ben\ legfeljebb\ k\ összsúlyú\ Hamilton\ kör \end{array}\right\}.$$

Tétel: Az UTAZÓÜGYNÖK probléma **NP**-teljes.

2.4.3 Tárbonyolultság

A tárbonyolultságot egy speciális, úgynevezett offline Turing-gépen vizsgáljuk.

Off-line Turing-gép: Offline Turing-gépnek nevezünk egy olyan többszalagos Turing-gépet, mely a bemenetet tartalmazó szalagot csak olvashatja, a többi, ún. munkaszalagokra pedig írhat is. Az offline Turing-gép tárigényébe csak a munkaszalagokon felhasznált terület számít be.

A továbbiakban Turing-gép alatt minidig offline Turing-gépet értünk. Most definiáljuk a tárbonyolultsággal kapcsolatos nyelvosztályokat.

SPACE(f(n)) = $\{L|L\ eldönthetõ\ \mathcal{O}(f(n))\ tárigényû\ determinisztikus\ Turing-géppel\}$

NSPACE(f(n)) = $\{L|L\ eldönthetõ\ \mathcal{O}(f(n))\ tárigényû\ nemdeterminisztikus\ Turing-géppel\}$

PSPACE = $\bigcup_{k>0}$ **SPACE**(n^k)

NPSPACE = $\bigcup_{k>0}$ **NSPACE**(n^k)

L = **SPACE**$(\log_2 n)$

NL = **NSPACE**$(\log_2 n)$

Savitch tétele: Ha $f(n) \geq \log n$, akkor **NSPACE**$(f(n)) \subseteq$ **SPACE**$(f^2(n))$.

Chapter 13

13

Záróvizsga tételsor

13. Alapvető algoritmusok és adatszerkezetek

Fekete Dóra

Alapvető algoritmusok és adatszerkezetek

Egyszerű adattípusok ábrázolásai, műveletei és fontosabb alkalmazásai. A hatékony adattárolás és visszakeresés néhány megvalósítása (bináris keresőfa, AVL-fa, 2-3-fa és B-fa, hasítás („hash-elés")). Összehasonlító rendező algoritmusok (buborék és beszúró rendezés, ill. verseny, kupac, gyors és összefésülő rendezés); a műveletigény alsó korlátja.

1 Egyszerű adattípusok ábrázolásai, műveletei és fontosabb alkalmazásai

1.1 Adattípus

Adatszerkezet: $\sim$ struktúra.
Adattípus: adatszerkezet és a hozzá tartozó műveletek.
Adatszerkezetek:

- *Tömb*: azonos típusú elemek sorozata, fix méretű.
- *Verem*: Mindig a verem tetejére rakjuk a következő elemet, csak a legfelsőt kérdezhetjük le, és vehetjük ki.

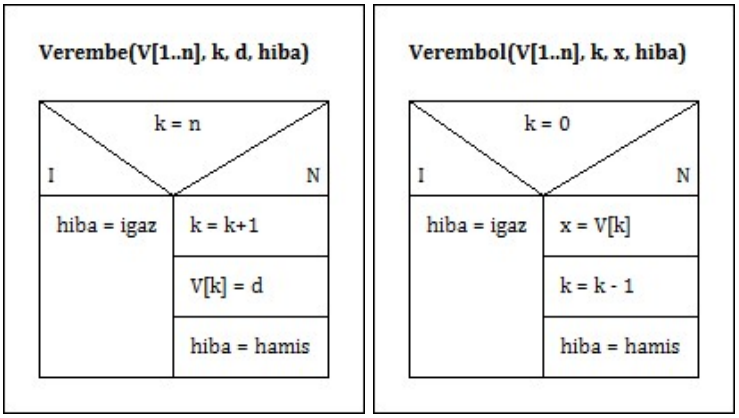


Figure 1: Verem műveletei

- *Sor*: Egyszerű, elsőbbségi és kétvégű. A prioritásos sornál az elemekhez tartozik egy érték, ami alapján rendezhetjük őket.

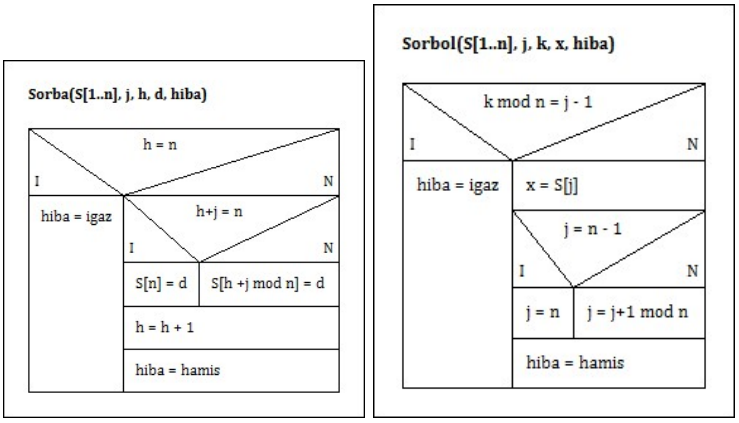


Figure 2: Sor műveletei

- *Lista*: Láncolt ábrázolással reprezentáljuk. 3 szempont szerint különböztethetjük meg a listákat: fejelem van/nincs, láncolás iránya egy/kettő, ciklusosság van/nincs. Ha fejelemes a listánk, akkor a fejelem akkor is létezik, ha üres a lista.
  A lista node-okból áll, minden node-nak van egy, a következőre mutató pointere, illetve lehet az előzőre is, ha kétirányú. Ezen kívül van egy első és egy aktuális node-ra mutató pointer is, és az utolsó elem mutatója NIL. A listát megvalósíthatjuk úgy, hogy tetszőleges helyre lehessen elemet beszúrni, illetve törölni.

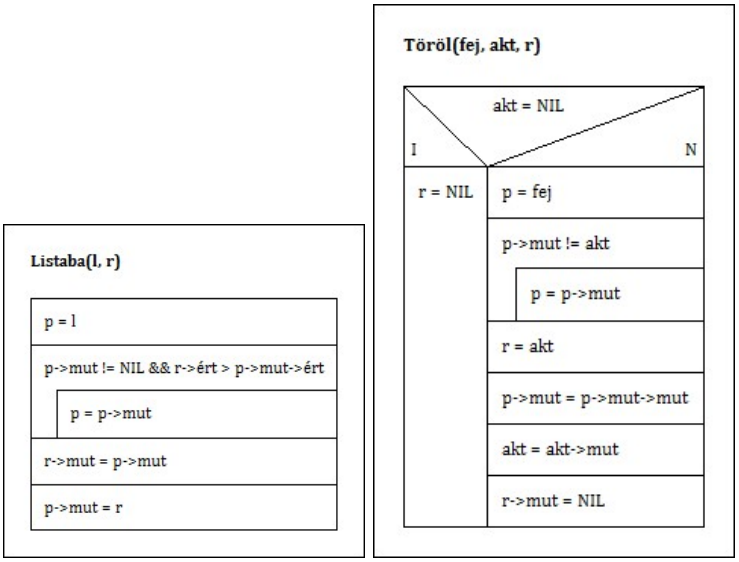


Figure 3: Lista műveletei

- *Fa*: Egyszerű, bináris és speciális (kupac, bináris keresőfa, AVL-fa). A bináris fát rekurzívan definiáljuk: $t \in T(E)$ [bin. fák típusérték halmaza(alaptípus)] $\iff$ t üres fa (jele: Ω), vagy t-nek van gyökéreleme, $bal(t)$, $jobb(t)$ részfája. Láncoltan ábrázoljuk, tömbösen csak teljes fák, illetve kupac esetén.

- *Kupac*: Olyan bináris fa, melynek alakja majdnem teljes és balra rendezett. Tömbösen ábrázoljuk, mert pointeresen a bonyolult lépkedést nem teszi lehetővé, tömbösen indexösszefüggésekkel könnyen megoldható.

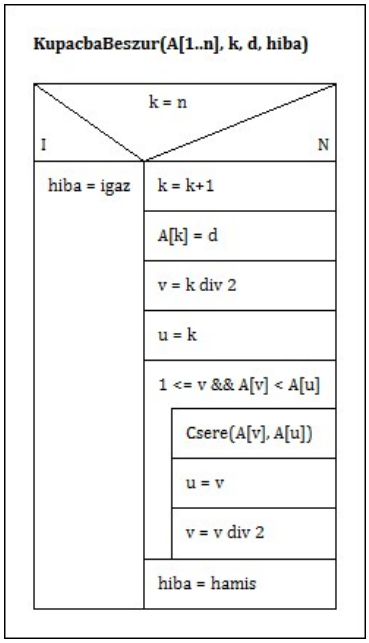


Figure 4: Kupac műveletei

- *Hasítótábla*
- *Gráf* [Nem egyszerű adattípus.]

1.2 Ábrázolásai

Absztrakciós szintek:

1. *absztrakt adattípus (ADT)*: specifikáció szintje, itt nincs szerkezeti összefüggés, csak matematikai fogalmak, műveletek logikai axiómákkal vagy előfeltételekkel.

   - algebrai (axiomatikus) specifikáció, példa: $Verembe : V \times E \to V$. Axióma, példa: $Felso(Verembe(v, e)) = e$
   - funkcionális (elő- és utófeltételes) specifikáció, példa: (elsőbbségi) sor $S(E), s \in S$ egy konkrét sor, $s = \{(e_1, t_1), ..., (e_n, t_n)\}$, $n \geq 0$. Ha $n = 0$, akkor a sor üres.
     $\forall i, j \in [1..n] : i \neq j \to t_i \neq t_j$.
     $Sorbol : S \to S \times E$, $\mathcal{D}_{Sorbol} = S \backslash \{ures\}$. Előfeltétel: $Q = (s = s' \wedge s' \neq \emptyset)$, utófeltétel: $R = (s = s' \backslash \{(e, t)\} \wedge (e, t) \in s' \wedge \forall i (e_i, t_i) \in s' : t \leq t_i)$.

2. *absztrakt adatszerkezet (ADS)*: kognitív pszichológia szintje, ábrák. Az alapvető szerkezeti tulajdonságokat tartalmazza (nem mindet). Ennek a szintnek is része a műveletek halmaza. Példák: az ábra egy irányított gráf, művelet magyarázata, adatszerkezet definiálása.

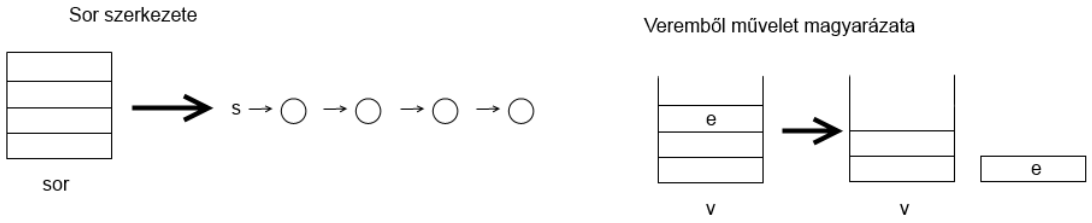


Figure 5: ADS

3. *ábrázolás/reprezentáció*: döntések (tömbös vagy pointeres ábrázolás), a nyitva hagyott szerkezeti kérdések. Egy adatszerkezetet többféle reprezentációval is meg lehet valósítani (pl. prioritásos sor lehet rendezetlen tömb, rendezett tömb, kupac).
    - *tömbös ábrázolás*: takarékos ábrázolás, elhelyezése, tetszőleges rákövetkezések, bejárások, de ezeket meg kell adni.
    - *pointeres ábrázolás*: minden pointer egy összetett rekord elejére mutat.
4. *implementálás*
5. *fizikai szint*

1.3 Műveletei

- Üres adatszerkezet létrehozása
- Annak lekérdezése, hogy üres-e az adatszerkezet
- Elem berakása, itt ellenőrizni kell, hogy nem telt-e még meg
- Elem kivétele vagy törlése, itt ellenőrizni kell, hogy nem üres-e
- Adott tulajdonságú elem (például maximum, veremben a felső) lekérdezése, itt is ellenőrizni kell, hogy üres-e az adatszerkezet
- Bejárások (preorder, inorder, postorder, szintfolytonos), listáknál az első, előző vagy következő elemre lépés
- Elem módosítása bizonyos adatszerkezeteknél (pl. listák)

1.4 Fontosabb alkalmazásai

Prioritásos sor: nagygépes programfuttatásnál az erőforrásokat a prioritás arányában osszuk el, adott pillanatban a maximális prioritásút válasszuk. Sürgősségi ügyeleten, gráfalgoritmusoknál is alkalmazható.
B-fa: ipari méretekben adatbázisokban használják.

2 A hatékony adattárolás és visszakeresés néhány megvalósítása (bináris keresőfa, AVL-fa, 2-3-fa és B-fa, hasítás („hash-elés”))

2.1 Bináris keresőfa

Nincsenek benne azonos kulcsok, a követendő elv: „kisebb balra, nagyobb jobbra". Inorder bejárással növekvő kulcssorozatot kapunk.
Műveletigénye fa magasságú nagyságrendű.
Az a cél, hogy a bináris keresőfa ne nyúljon meg láncszerűen, erre jó az AVL-fa és a 2-3-fa.

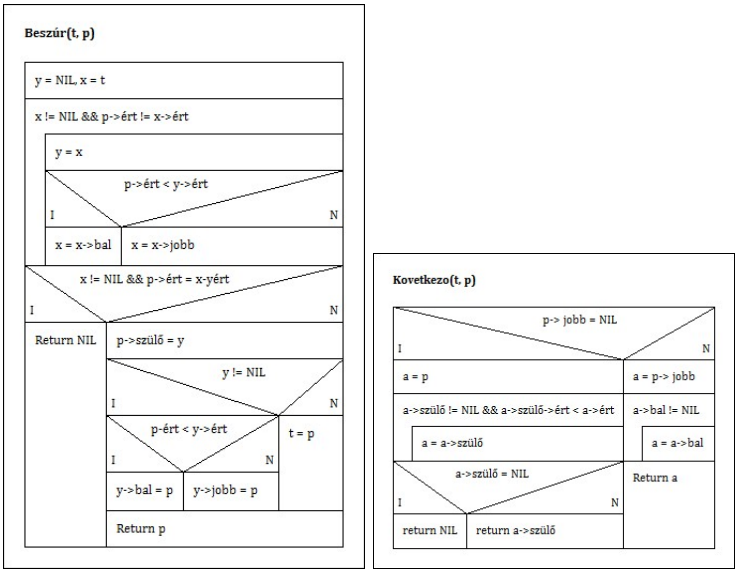


Figure 6: Bináris keresőfa műveletei

2.2 AVL-fa

Cél: a t bináris keresőfa magasságának $\log_2(n)$ közelében tartása, azaz $h(t) \leq c \cdot \log_2(n)$, ahol c elfogadhatóan kicsi. Az ilyen fát kiegyensúlyozottnak nevezzük.
AVL: Adelszon-Velszkij, Landisz 1962-ben alkották meg.
A t bináris keresőfát egyúttal AVL-fának nevezzük $\iff$ t minden x csúcsára $|h(bal(x)) - h(jobb(x))| \leq 1$.
Minden csúcsnak van egy címkéje $+, -, =$ (gyerekek magasságának különbsége). A beszúrás helyétől felfelé ellenőrizzük ezeket, és ha kell, akkor módosítjuk. Ha valahol $++$ vagy $--$ alakul ki, akkor ott elromlik az AVL-tulajdonság, egy vagy több forgatással vagy átkötéssel konstans műveletigénnyel helyre lehet hozni.
Többféle séma is van: $(++, +), (++, -), (++, =)$ és a tükörképeik.

2.3 2-3-fa és B-fa

2-3-fa kis méretben az elmélet számára jó, a B-fa a gyakorlati változat adatbázisban.
t 2-3-fa $\iff$ minden belső csúcsnak 2 vagy 3 gyereke van, a levelek azonos szinten helyezkednek el, adatrekordok csak a levelekben vannak, belső pontokban kulcsok és mutatók, levelekben a kulcsok balról jobbra nőnek.
Ha 4 gyerek lenne a beszúrás után, akkor csúcsot kell vágni. Ha törlésnél 1 gyerek lenne valahol, akkor csúcsösszevonásokat és gyerekátadást alkalmazunk.
B-fa nagyobb méretű, itt két határ között mozog a gyerekszám: $\lceil \frac{r}{2} \rceil$ és r, ahol $50 \leq r \leq 1000$.

2.4 Hasítás

Kulcsos rekordokat tárol.

- *Hasítás láncolással*: a kulcsütközést láncolással oldja fel. Van egy hasítófüggvény: $h : U \to [0..m-1]$, elvárás vele kapcsolatban, hogy gyorsan számolható és egyenletes legyen. m-et úgy választjuk meg n nagyságrendjének ismeretében, hogy $\alpha = \frac{n}{m}$ lesz a várható listahossz, ha egyenletes hasítást feltételezünk.
  Például kétirányú listát használhatunk a hasításhoz. Műveletek: beszúrás, keresés, törlés.
  Gyakorlatban érdemes m-et úgy megválasztani, hogy olyan prímszám legyen, ami nem esik 2-hatvány közelébe.

- *Hasítás nyitott/nyílt címzéssel*: A kulcsokat lehessen egészként értelmezni, ekkor vannak jó hasítófüggvények.
  Próbálkozás általános képlete: $h(k) + h_i(k) \ (mod\ M)$, $0 \leq i \leq M-1$. Egész addig alkalmazza, amíg üres helyet nem talál.

1. *Lineáris próba*: $h_i(k) = -i \ (mod \ M)$, egyesével balra lépegetve keressük az üres helyet. Hátránya az elsődleges csomósodás, ez jelentős lassulást okoz beszúrásnál és keresésnél.
2. *Négyzetes próba*: $h_i(k) = (-1)^i(\lceil \frac{i}{2} \rceil)^2 \ (mod \ M)$, a négyzetszámokkal lépegetünk balra-jobbra, ezek az eltolások kiadják $\{0, 1, ..., M-1\}$-et. Hátrány: másodlagos csomósodás.
3. *Kettős hash-elés*: $h_i(k) = -ih'(k) \ (mod \ M)$, $h'(k)$ a k-hoz tartozó egyedi lépésköz, $(h'(k), M) = 1$ relatív prímek. Ha az M elég nagy, akkor nincs csomósodás.

- *Hasítófüggvények*: Leggyakoribb: k egész, kongruencia reláció. Általánosan: $h(k) = (ak + b \ (mod \ p)) \ (mod \ M)$, az univerzális hasítás családja. Tapasztalat: k egyenletesen hasít.

3 Összehasonlító rendező algoritmusok (buborék és beszúró rendezés, ill. verseny, kupac, gyors és összefésülő rendezés)

Buborék- és beszúró rendezés klasszikusak, n^2-es műveletigényűek, a többi hatékony, $n \log(n)$-es idejűek.

3.1 Buborékrendezés

A legnagyobb értéket cserékkel a végéig felbuborékozza, ezt minden ciklus végén elhagyjuk. A gyakorlatban nem használják.

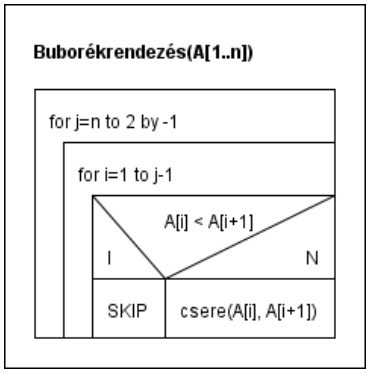


Figure 7: Buborékrendezés

3.2 Beszúró rendezés

Kis n-re (kb 30) ez a rendezés a legjobb.
Itt az elemmozgatás mindig 1 értékadás (buborékrendezésnél a csere 3 értékadás). Listára is implementálni lehet, ez esetben a pointereket állítjuk át, az elemek helyben maradnak.
$A[1..j]$ rendezett, $j = 1..n$.

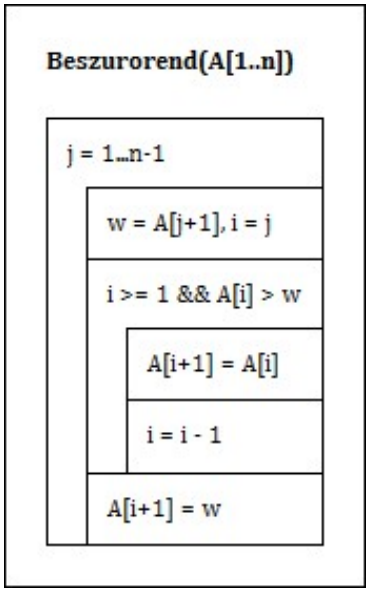


Figure 8: Beszúró rendezés

3.3 Versenyrendezés

Gyakorlatban nem használják.
Teljes bináris fa az alapja, egy versenyfa. Szintfolytonosan ábrázoljuk tömbösen.

1. A versenyfa kitöltése (a verseny lejátszása). Maximum a gyökérben, ennek kiírása az outputra.
2. $(n-1)$-szer
   a) gyökérben szereplő maximális elem helyének megkeresése a levélszinten és $-\infty$ írása a helyére
   b) az egészet újrajátsszuk (azt az ágat, ahol volt) $\rightarrow$ 2. legjobb feljut a gyökérbe

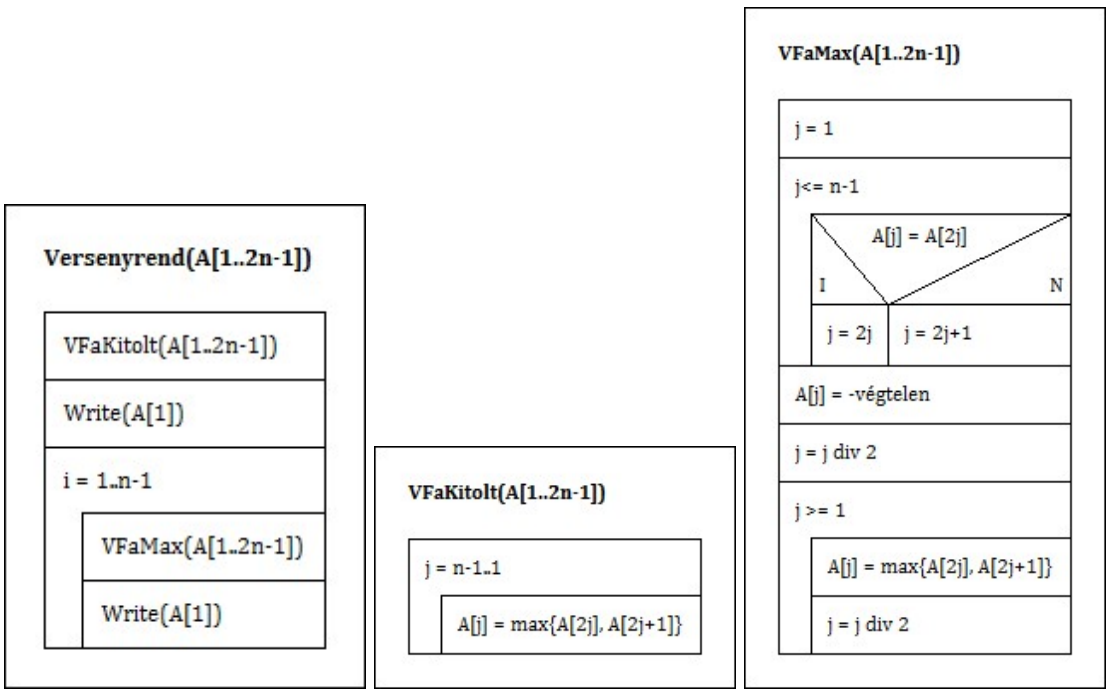


Figure 9: Versenyrendezés

3.4 Kupacrendezés

1. Kezdő kupac kialakítása. Rendezetlen input tömbből tartalmi invariánst készítünk, ami már kupac struktúrájú. Elv: cserékkel lesüllyesztjük az elemet a nagyobb gyerek irányába, ha kisebb a nagyobbik gyereknél. A süllyesztés eljuthat ahhoz a csúcshoz, amelynek nincs jobb gyereke.

2. $(n-1)$-szer

   a) gyökérelem és az alsó szint jobb szélső (=utolsó) aktív elemének cseréje, és a csere után lekerült elem inaktívvá tétele

   b) a gyökérbe került elem süllyesztése az aktív kupacon

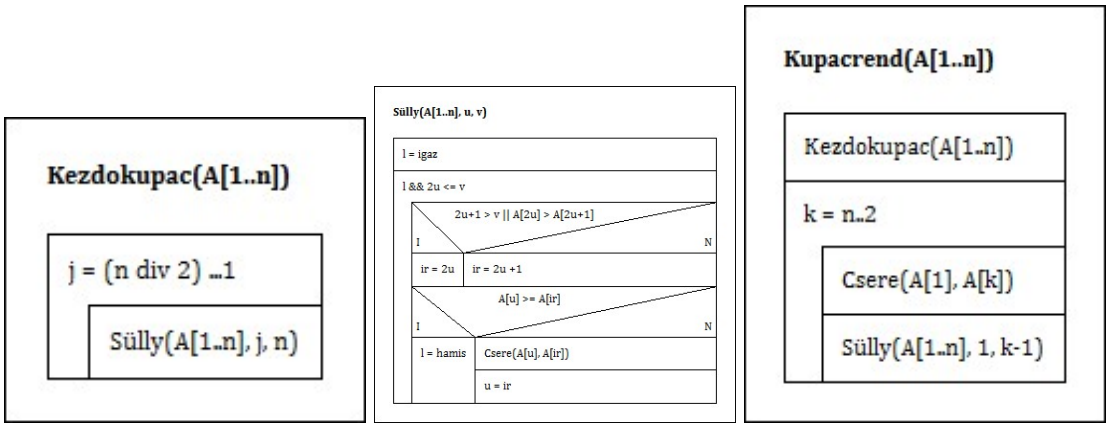


Figure 10: Kupacrendezés

A kezdőkupac kialakításánál, és a ciklus közben a süllyesztés módja kicsit különbözik, hiszen az első esetben a változó elem süllyed le a teljes kupacon, a másodikban a gyökér süllyed az aktív kupacon. A képen látható algoritmus mindkét műveletet teljesíti.

3.5 Gyorsrendezés

Elve: véletlenül választunk egy elemet. A nála kisebb elemeket tőle balra, a nagyobbakat jobbra rakjuk, az elemet berakjuk a két rész közé. Rekurzív algoritmus.

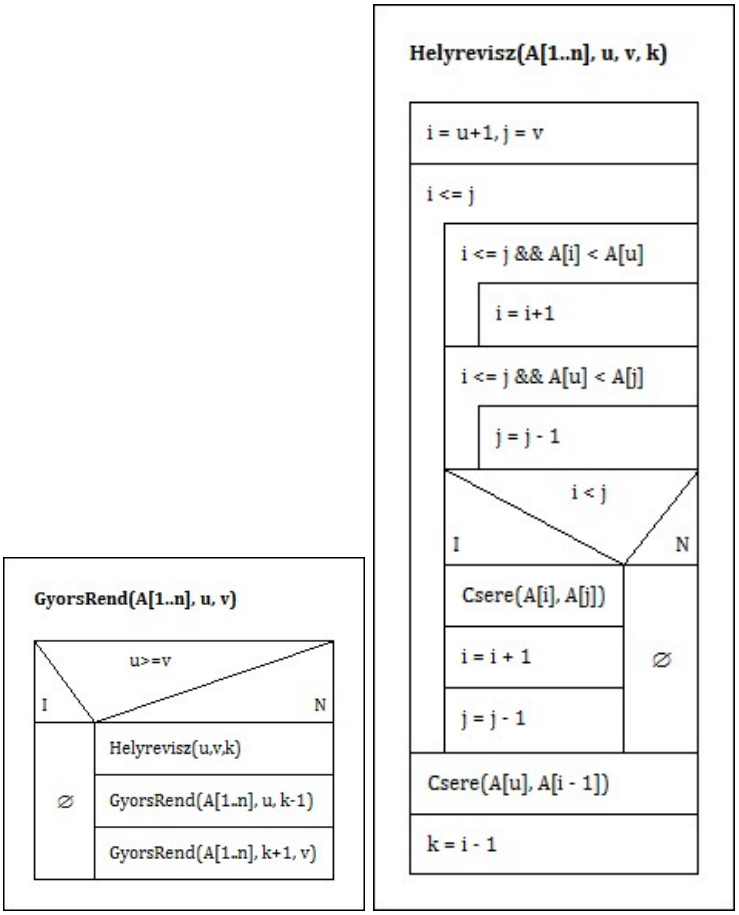


Figure 11: Gyorsrendezés

3.6 Összefésülő rendezés

Alapja: 2 rendezett sorozat összefésülése. Ezt alkalmazhatjuk felülről lefelé (rekurzív) vagy alulról felfelé (iteratív), ez utóbbit szekvenciális fájloknál.

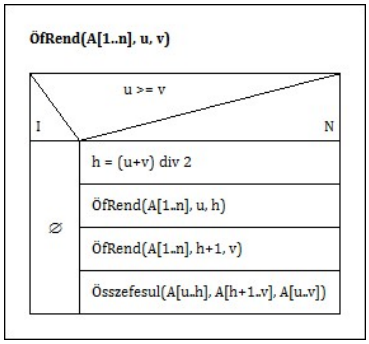


Figure 12: Összefésülő rendezés

4 A műveletigény alsó korlátja összehasonlító rendezésekre

4.1 Műveletigény

Kijelöljük a domináns műveleteket, és az n inputméret függvényében hányszor hajtódnak végre, ezt nézzük. Jelölés általánosan $T(n)$, de lehet konkrétan is, pl $Cs(n)$ [csere]. $mT(n)$ a minimális műveletigény, $MT(n)$ a maximális és $AT(n)$ az átlagos.

Θ : nagyságrendileg azonos, két konstans közé beszorítható

$\mathcal{O}$: nagyságrendi felső becslés, o: nincs megengedve az egyenlőség

Ω : nagyságrendi alsó becslés, ω: nincs megengedve az egyenlőség

4.2 Alsókorlát

Például: n elem maximumkiválasztása legalább $(n-1)$ összehasonlítást igényel. Bizonyítása: Ha ennél kevesebb összehasonlítás lenne, akkor legalább 1 elem kimaradt, és ezzel ellentmondásba kerülhetünk.
Döntési fa: Algoritmus n méretű inputra. Kiegyenesednek a ciklusok véges hosszú lánccá, a végrehajtás nyoma egy fa struktúrát ad. Tökéletes fa: minden belső pontnak 2 gyereke van. Ennél az algoritmusnál nincs jobb, mert $2^{h(t)} \geq n!$, összehasonlító rendezés esetén, $n!$ input.

4.3 Alsókorlát legrosszabb esetben

Tétel: $MO_R(n) = \Omega(n \log n)$ A legkedvezőtlenebb permutációra legalább $n \log n$ összehasonlítás. Bizonyítás: $\log_2 n! \leq n \log_2 n = \Omega(n \log n)$, és $MO_R(n) = h(t) \geq \log_2 n!$ (lemma miatt) $\Rightarrow MO_R(n) = \Omega(n \log n)$.

4.4 Alsókorlát átlagos esetben

Legyen minden input egyformán valószínű ($\frac{1}{n!}$).
$AO_R(n) = \frac{1}{n!} \sum_{p \in Perm(n)} O_R(p)$, és könnyű belátni, hogy $\sum_p O_R(p) = lhsum(h(t_R(n)))$ [levél-magasság-összeg].
Lemma: Az $n!$ levelet tartalmazó tökéletes fák közül azokra a legkisebb az $lhsum(h(t_R(n)))$ érték, amelyek majdnem teljesek.
Tétel: $AO_R(n) = \Omega(n \log n)$.

Chapter 14

14

Záróvizsga tételsor
14. Haladó algoritmusok

Dobreff András

Haladó algoritmusok
Gráfalgoritmusok: gráfok ábrázolás, szélességi bejárás, minimális költségű utak keresése, minimális költségű feszítőfa keresése, mélységi bejárás, DAG topologikus rendezése. Adattömörítések (Huffman- és LZW-algoritmus). Mintaillesztés módszerei.

1 Gráfalgoritmusok

1.1 Gráf ábrázolás

Láncolt listás ábrázolás
A gráf csúcsait helyezzük el egy tömbben (vagy láncolt listában). Minden elemhez rendeljünk hozzá egy láncolt listát, melyben az adott csúcs szomszédjait (az esetleges élsúlyokkal) soroljuk fel.

Mátrixos ábrázolás
Legyen a csúcsok elemszáma n. Ekkor egy $A^{n\times n}$ mátrixban jelöljük, hogy mely csúcsok vannak összekötve. Ekkor mind a sorokban, mind az oszlopokban a csúcsok szerepelnek, és az a_{ij} cellában a i csúcsból j csúcsba vezető él súlya szerepel, ha nincs él a két csúcs között, akkor $-\infty$ (súlyozatlan esetben 1 és 0)

Amennyiben a gráf irányítatlan nyilván $a_{ij} = a_{ji}$

1.2 Szélességi bejárás

G gráf (irányított/irányítatlan) s startcsúcsából a távolság sorrendjében érjük el a csúcsokat. A legrövidebb utak feszítőfáját adja meg, így csak a távolság számít, a súly nem.

A nyilvántartott csúcsokat egy sor adatszerkezetben tároljuk, az aktuális csúcs gyerekeit a sor-ba tesszük. A következő csúcs pedig a sor legelső eleme lesz.

A csúcsok távolságát egy d, szüleiket egy π tömbbe írjuk, és ∞ illetve 0 értékekkel inicializáljuk.

Az algoritmus:

1. Az s startcsúcsot betesszük a sorba
2. A következő lépéseket addig ismételjük, míg a sor üres nem lesz
3. Kivesszük a sor legelső (u) elemét
4. Azokat a gyerekcsúcsokat, melyeknek a távolsága nem ∞ figyelmen kívül hagyjuk (ezeken már jártunk)
5. A többi gyerekre (v): beállítjuk a szülőjét ($\pi[v] = u$), és a távolságát ($d[v] = d[u] + 1$). Majd berakjuk a sorba.

1.3 Minimális költségű utak keresése

Dijkstra algoritmus
Egy G irányított, pozitív élsúlyokkal rendelkező gráfban keres s startcsúcsból minimális költségű utakat minden csúcshoz.

Az algoritmus a szélességi bejárás módosított változata. Mivel itt egy hosszabb útnak lehet kisebb a költsége, mint egy rövidebbnek, egy már megtalált csúcsot nem szabad figyelmen kívül hagyni. Ezért minden csúcs rendelkezik három állapottal (nem elért, elért, kész). A d és π tömböket a szélességi bejáráshoz hasonlóan kezeljük.

A még nem kész csúcsokat egy prioritásos sorba helyezzük, vagy minden esetben minimumkeresést alkalmazunk.

Az algoritmus:

1. Az s startcsúcs súlyát 0-ra állítjuk eltároljuk
2. A következő lépéseket addig ismételjük, míg a konténerünk üres nem lesz
3. Kivesszük a sor legjobb (u) elemét, és "kész"-re állítjuk
4. Ha egy gyerekcsúcs (v) nem kész, és a jelenleg hozzávezető út súlya kisebb, mint az eddigi, akkor: a szülőjét u-ra állítjuk ($\pi[v] = u$), és a súlyát frissítjük ($d[v] = d[u] + d(u, v)$).
5. A többi csúcsot kihagyjuk.

Bellman-Ford algoritmus

Egy G élsúlyozott (akár negatív) irányított gráf s startcsúcsából keres minden élhez minimális költségű utakat, illetve felismeri, ha negatív költségű kör van a gráfban. A d és π tömböket az előzőekhez hasonlóan kezeljük.

Az algoritmus:

1. A startcsúcs súlyát állítsuk be 0-ra.
2. $n-1$ iterációban menjünk végig az összes csúcson, és minden csúcsot (u) vessünk össze minden csúccsal (v). Ha olcsóbb utat találtunk akkor v-be felülírjuk a súlyát ($d[v] = d[u] + d(u, v)$), és a szülőjét ($\pi[v] = u$).
3. Ha az n-edik iterációban is történt módosítás, negatív kör van a gráfban

1.4 Minimális költség feszítőfa keresése

A Prim algoritmus egy irányítatlan élsúlyozott (akár negatív) gráf s startcsúcsából keres minimális költségű feszítőfát. A d és π tömböket az előzőekhez hasonlóan kezeljük. Az algoritmus egy prioritásos sorba helyezi a csúcsokat.

Az algoritmus:

1. A startcsúcs súlyát állítsuk be 0-ra.
2. A csúcsokat behelyezzük a prioritásos sorba.
3. A következő lépéseket addig végezzük, míg a prioritásos sor ki nem ürül.
4. Kiveszünk egy csúcsot (u) a sorból.
5. Minden gyerekére (v), amely még a sorban és a nyilvántartott v-be vezető él súlya nagyobb, mint a most megtalált: A v szülőjét u-ra változtatjuk, a nyilvántartott súlyt felülírjuk $d[v] = d(u, v)$. Majd felülírjuk a v állapotát a prioritásos sorban.
6. Azokkal a gyerekekkel, melyek nincsenek a sorban, vagy a súlyukon nem tudunk javítani, nem változtatunk.

1.5 Mélységi bejárás

G irányított (nem feltétlenül összefüggő) gráf mélységi bejárásával egy mélységi fát (erdőt) kapunk. Az algoritmus a következő:

- Az élsúlyok nem játszanak szerepet
- Nincs startcsúcs, a gráf minden csúcsára elindítjuk az algoritmust. (Természetesen ekkor, ha már olyan csúcsot választunk, amin már voltunk, az algoritmus nem indul el.)
- A csúcsokat mohón választjuk, azaz minden csúcs gyerekei közül az elsőt választva haladunk előre, amíg csak lehet. (Olyan csúcsot találunk, amelynek nincs gyereke, vagy minden gyerekén jártunk már.)
- Ha már nem lehet előre haladni visszalépünk.

- Minden csúcshoz hozzárendelünk két értéket. Az egyik a mélységi sorszám, mely azt jelöli, hogy hanyadiknak értük el. A másik a befejezési szám, mely azt jelzi, hogy hanyadiknak léptünk vissza belőle.

A gráf éleit a mélységi bejárás közben osztályozhatjuk. (Inicializáláskor minden értéket 0-ra állítottunk)

- Faél: A következő csúcs mélységi száma 0
- Visszaél: A következő csúcs mélységi száma nagyobb, mint 0, és befejezési száma 0 (Tehát az aktuális út egy előző csúcsára kanyarodunk vissza)
- Keresztél: A következő csúcs mélységi száma nagyobb, mint 0, és befejezési száma is nagyobb, mint 0, továbbá az aktuális csúcs mélységi száma nagyobb, mint a következő csúcs mélységi száma. (Ekkor egy az aktuális csúcsot megelőző csúcsból induló, már megtalált útba mutató éllel van dolgunk)
- Előreél: A következő csúcs mélységi száma nagyobb, mint 0, és befejezési száma is nagyobb, mint 0, továbbá az aktuális csúcs mélységi száma kisebb, mint a következő csúcs mélységi száma. (Ekkor egy az aktuális csúcsból induló, már megtalált útba mutató éllel van dolgunk)

1.6 DAG Topologikus rendezése

Alapfogalmak

- Topologikus rendezés:
  Egy $G(V, E)$ gráf topologikus rendezése a csúcsok olyan sorrendje, melyben $\forall(u \to v) \in E$ élre u előbb van a sorrendben , mint v
- DAG - Directed Acyclic Graph:
  Irányított körmentes gráf.
  Legtöbbször munkafolyamatok irányítására illetve függőségek analizálására használják.
  Tulajdonságok:
  - Ha G gráfra a mélységi bejárás visszaélt talál (Azaz kört talált) $\implies$ G nem DAG
  - Ha G nem DAG (van benne kör) $\implies$ Bármely mélységi bejárás talál visszaélt
  - Ha G-nek van topologikus rendezések $\implies$ G DAG
  - Minden DAG topologikusan rendezhető.

DAG topologikus rendezése

Egy G gráf mélységi bejárása során tegyük verembe azokat a csúcsokat, melyekből visszaléptünk. Az algoritmus után a verem tartalmát kiírva megkapjuk a gráf egy topologikus rendezését.

2 Adattömörítések

2.1 Huffman-algoritmus

A Huffman-algoritmussal való tömörítés lényege, hogy a gyakrabban előforduló elemeket (karaktereket) rövidebb, míg a ritkábban előfordulókar hosszabb kódszavakkal kódoljuk.

Ehhez tisztában kell lennünk az egyes karakterek gyakoriságával (vagy relatív gyakoriságával). Ezek alapján egy ún. Huffman-fát építünk, melyben az éleket a kód betűivel címkézzük, a fa levelein a kódolandó betűk helyezkednek el, a gyökérből a levelekig vezető út címkéi alapján rajuk össze a kódszavakat.

Az algoritmus (spec. bináris Huffman fára):

1. A kódolandó szimbólumokat gyakoriságaik alapján sorba rendezzük.
2. A következő redukciós lépéseket addig hajtjuk végre, míg egy csoportunk marad.
3. Kiválasztjuk az utolsó két elemet (legritkább), összevonjuk őket egy új csoportba, és ennek a csoportnak a gyakorisága a gyakoriságok összege lesz.
4. A csoportot visszahelyezzük a rendezett sorba (gyakoriság alapján rendezve).

5. A csoportból új csúcsot képezünk, mely csúcs az őt alkotó két elem szülője lesz.

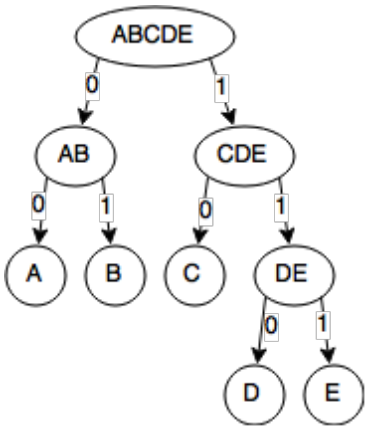
Példa:
Legyen a következő 5 betű, mely a megadott gyakorisággal fordul elő:

A	B	C	D	E
5	4	3	2	1

Ekkor a redukciós lépések a következők:

A	B	C	D, E
5	4	3	3
C, D, E	A	B
6	5	4
A , B	C, D, E
9	6
A , B , C, D, E
15

A huffman-fa a XY. ábrán látható.



ábra 1: Huffman-fa példa

Tehát a kódszavak:

A	B	C	D	E
00	01	10	110	111

2.2 LZW-algoritmus

Az LZW (Lempel-Ziv-Welch) tömörítésnek a lényege, hogy egy szótárat bővítünk folyamatosan, és az egyes kódolandó szavakhoz szótárindexeket rendelünk.

Kódolás

A kódolás algoritmusa a következő lépésekből áll:

1. A szótárt inicializáljuk az összes 1 hosszú szóval
2. Kikeressük a szótárból a leghosszabb, jelenlegi inputtal összeillő W sztringet
3. W szótárindexét kiadjuk, és W-t eltávolítjuk az inputról
4. A W szó és az input következő szimbólumának konkatenációját felvesszük a szótárba
5. A 2. lépéstől ismételjük

Dekódolás

A dekódolás során is építenünk kell a szótárat. Ezt már azonban csak a dekódolt szöveg(rész) segítségével tudjuk megtenni, mivel egy megkapott kód dekódolt szava és az utána lévő szó első karakteréből áll össze a szótár következő eleme.

Tehát a dekódolás lépései:

1. Kikeressük a kapott kódhoz tartozó szót a szótárból (u), az output-ra rakjuk
2. Kikeressük a következő szót (v) a szótárból, az első szimbólumát u-hoz konkatenálva a szótárba rakjuk a következő indexszel.
3. Amennyiben már nincs következő szó, dekódolunk, de nem írunk a szótárba.

Megtörténhet az az eset, hogy mégis kapunk olyan kódszót, mely még nincs benne a szótárban. Ez akkor fordulhat elő, ha a kódolásnál az aktuálisan szótárba írt szó következik.

Példa:

Szöveg: AAA
Szótár: A - 1

Ekkor a kódolásnál vesszük az első karaktert, a szótárbeli indexe 1, ezt kiküldjük az outputra. A következő karakter A, így AA-t beírjuk a szótárba 2-es indexszel. Az első karaktert töröljük az inputról. Addig olvasunk, míg szótárbeli egyezést találunk, így AA-t olvassuk (amit pont az előbb raktunk be), ennek indexe 2, tehát ezt küldjük az outputra. AA-t töröljük az inputról, és ezzel végeztünk is. Az output: 1,2

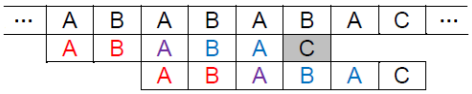
Dekódoljuk az 1,2 inputot! Jelenleg a szótárban csak A van 1-es indexszel. Vegyük az input első karakterét, az 1-et, ennek szótárbeli megfelelője A. Ezt tegyük az outputra. A következő index a 2, de ilyen bejegyzés még nem szerepel a szótárban.

Ebben az esetben a dekódolásnál, egy trükköt vetünk be. A szótárba írás pillanatában még nem ismert a beírandó szó utolsó karaktere (A példában A-t találtuk, de nem volt 2-es bejegyzés). Ekkor ?-et írunk a szótárba írandó szó utolsó karakterének helyére. (Tehát A? - 2 kerül a szótárba). De mostmár tudni lehet az új bejegyzés első betűjét (A? - 2 az új bejegyzés, ennek első betűje A). Cseréljük le a ?-et erre a betűre. (Tehát AA - 2 lesz a szótárban).

3 Mintaillesztés

3.1 Knuth-Morris-Pratt algoritmus

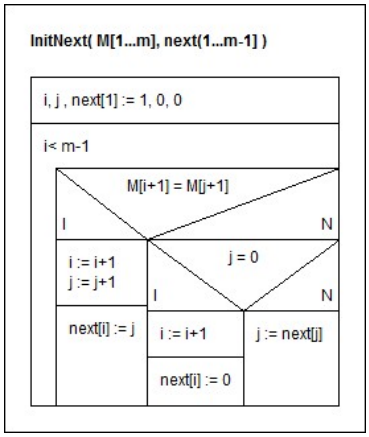
A Knuth-Morris-Pratt eljárásnak a Brute-Force (hasonlítsuk össze, toljunk egyet, stb..) módszerrel szemben az az előnye, hogy egyes esetekben, ha a mintában vannak ismétlődő elemek, akkor egy tolásnál akár több karakternyit is ugorhatunk.



ábra 2: KMP algoritmus több karakter tolás estén

Az ugrás megállapítását a következőképp tesszük: Az eddig megvizsgált egyező mintarész elején (prefix) és végén (suffix) olyan kartersorozatot keresünk, melyek megegyeznek. Ha találunk ilyet, akkor a mintát annyival tolhatjuk, hogy az elején lévő része ráilleszkedjen a végén levőre.

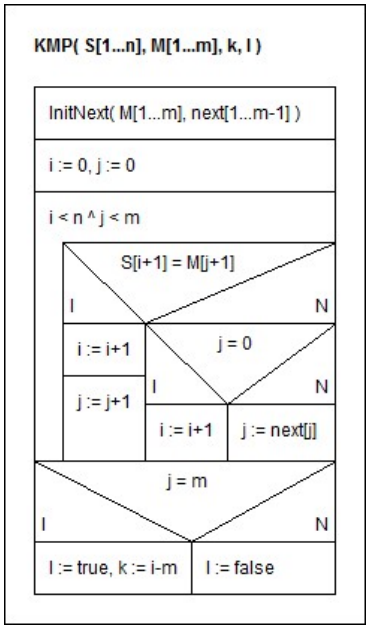
Azt, hogy ez egyes esetekben mekkorát tolhatunk nem kell minden elromlás alkalmával vizsgálni. Ha a mintára önmagával lefuttatjuk az algoritmus egy módosított változatát (3. ábra), kitölthetünk egy tömböt, mely alapján a tolásokat végezni fogjuk.



ábra 3: KMP tolásokat szabályzó tömb kitöltése

Az algoritmus (ld 4. ábra):

- Két indexet i és j futtatunk a szövegen illetve a mintán.
- Ha az $i + 1$-edik és $j + 1$-edik karakterek megegyeznek, akkor léptetjük mind a kettőt.
- Ha nem egyeznek meg, akkor:
  - Ha a minta első elemét vizsgáltuk, akkor egyet tolunk a mintán, magyarul a minta indexe marad az első betűn, és a szövegben lévő indexet növeljük eggyel ($i = i + 1$)
  - Ha nem a minta első elemét vizsgáltuk, akkor annyit tolunk, amennyit szabad. Ez azt jelenti, hogy csak a mintán lévő indexet helyezzük egy kisebb helyre ($j = next[j]$)
- Addig megyünk, míg vagy a minta, vagy a szöveg végére nem érünk. Ha a minta végére értünk, akkor megtaláltuk a mintát a szövegben, ha a szöveg végére értünk, akkor pedig nem.

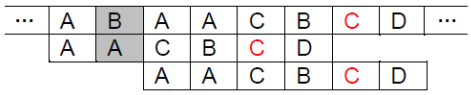


ábra 4: KMP algoritmus

3.2 Boyer-Moore — Quick search algoritmus

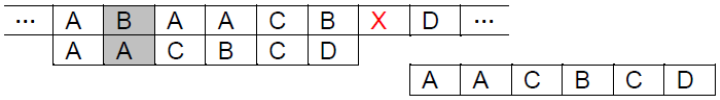
Míg a KMP algoritmus az elromlás helye előtti rész alapján döntött a tolásról, addig a QS a minta utáni karakter alapján. Tehát elromlás esetén:

- Ha a minta utáni karakter benne van a mintában, akkor jobbról az első előfordulására illesztjük. (5. ábra)



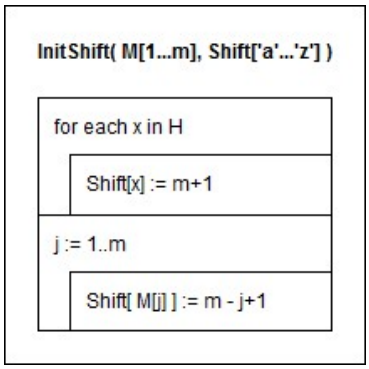
ábra 5: QS - eltolás ha a minta utáni karakter benne van a mintában

- Ha a minta utáni karakter nincs benne a mintában, akkor a mintát ezen karakter után illesztjük. (6. ábra)



ábra 6: QS - eltolás ha a minta utáni karakter nincs benne a mintában

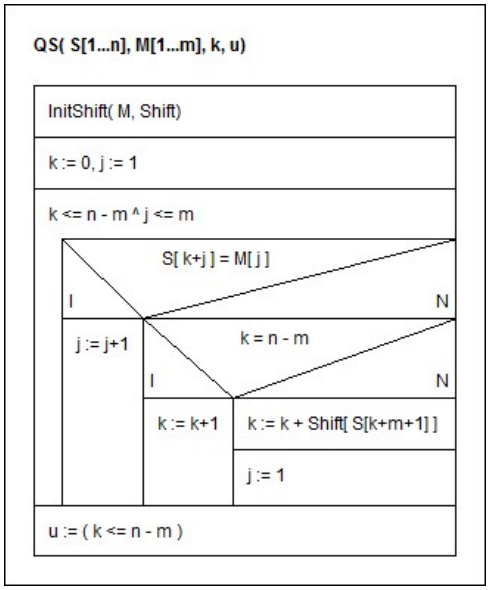
Az eltolás kiszámítását megint elő lehet segíteni egy tömbbel, most azonban, mivel nem a minta az érdekes, és nem tudjuk pontosan mely karakterek szerepelnek a szövegben, így a tömbbe az egész abc-t fel kell vennünk (7. ábra)



ábra 7: QS - Az eltolást elősegítő tömb ($Shift['a'...'z']$) konstruálása

Az algoritmus (ld. 8. ábra):

- Két indexet k és j futtatunk a szövegen illetve a mintán.
- Ha a szöveg $k+j$-edik eleme megegyezik a minta j-edik karakterével, akkor léptetjük j-t (mivel a szövegben $k+j$-edik elemet nézzük, így elég j-t növelni).
- Ha nem egyeznek meg, akkor:
  - Ha a minta már a szöveg végén van ($k = n - m$), akkor csak növeljük k-t eggyel, ami hamissá teszi a ciklus feltételt.
  - Ha még nem vagyunk a szöveg végén k-t toljuk annyival, amennyivel lehet (ezt az előre beállított $Shift$ tömb határozza meg). És a j-t visszaállítjuk 1-re.
- Addig megyünk, míg vagy a minta végére érünk j-vel, vagy a mintát továbbtoltuk a szöveg végénél. Előbbi esetben egyezést találtunk, míg az utóbbiban nem.



ábra 8: QS

3.3 Rabin-Karp algoritmus

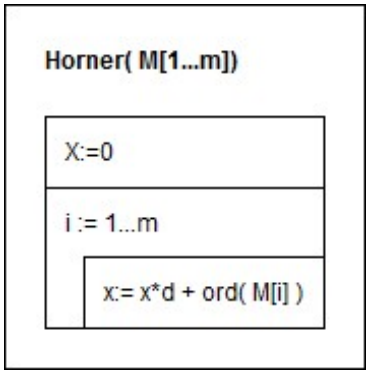
A Rabin-Karp algoritmus lényege, hogy minden betűhöz az ábécéből egy számjegyet rendelünk, és a keresést számok összehasonlításával végezzük. Világos, hogy ehhez egy ábécé méretnek megfelelő számrendszerre lesz szükségünk. A szövegből mindig a minta hosszával egyező részeket szelünk ki, és ezeket hasonlítjuk össze.

Példa:
Minta: BBAC $\rightarrow$ 1102
Szöveg: DACABBAC $\rightarrow$ 30201102, amiből a következő számokat állítjuk elő: 3020, 0201, 2011, 0110, 1102

A fent látható szeletek lesznek az s_i-k.

Az algoritmus működéséhez azonban számos apró ötletet alkalmazunk:

1. A minta számokká alakítását Horner-módszer segítségével végezzük.



ábra 9: RK - Horner-módszer

   Az *ord*() függvény az egyes betűknek megfelelő számot adja vissza. A d a számrendszer alapszáma.

2. A szöveg mintával megegyező hosszú szeleteinek (s_i) előállítása:
   s_0-t a Horner-módszerrel ki tudjuk számolni. Ezek után s_{i+1} a következőképp számolandó:

$$s_{i+1} = (s_i - ord(S[i]) \cdot d^{m-1}) \cdot d + ord(S[i+1])$$

Magyarázat: s_i elejéről levágjuk az első számjegyet ($s_i - ord(S[i]) \cdot d^{m-1}$), majd a maradékot eltoljuk egy helyiértékkel (szorzás d-vel), végül az utolsó helyiértékre beírjuk a következő betűnek megfelelő számjegyet ($+ord(S[i+1])$)

Példa:

Az előző példa szövegével és mintájával ($d = 10$ elemű ábécé és $m = 4$ hosszú minta):
$s_0 = 3020$, ekkor: $s_{0+1} = s_1 = (3020 - ord(D) \cdot 10^3) \cdot 10 + ord(B) = (3020 - 3000) \cdot 10 + 1 = 0201$

3. Felmerülhet a kérdés, hogy az ilyen magas alapszámú számrendszerek nem okoznak-e gondot az ábrázolásnál? A kérdés jogos. Vegyük a következő életszerű példát:

   4 bájton ábrázoljuk a számainkat (2^{32}). Az abc legyen 32 elemű ($d = 32$), a minta 8 hosszú ($m = 8$). Ekkor a d^{m-1} kiszámítása: $32^7 = (2^5)^7 = 2^{35}$, ami már nem ábrázolható 4 bájton.

   Ennek kiküszöbölésére vezessünk be egy nagy p prímet, melyre $d \cdot p$ még ábrázolható. És a műveleteket számoljuk $\mod p$. Ekkor természetesen a kongruencia miatt lesz olyan eset, amikor az algoritmus egyezést mutat, mikor valójában nincs. Ez nem okoz gondot, mivel ilyen esetben karakterenkénti egyezést vizsgálva ezt a problémát kezelni tudjuk. (Fordított eset nem fordul elő tehát nem lesz olyan eset, mikor karakterenkénti egyezés van, de numerikus nincs). [Ha p kellően nagy, a jelenség nagyon ritkán fordul elő.]

4. A $\mod p$ számítás egy másik problémát is felvet. Ugyanis a kivonás alkalmával negatív számokat is kaphatunk.

   Például: Legyen $p = 7$, ekkor, ha $ord(S[i]) = 9$, akkor előző számítás után $s_i = 2...$, de ebből $ord(S[i]) \cdot d^{m-1} = 9 \cdot 10^3 = 9000$-et vonunk ki negatív számot kapunk.

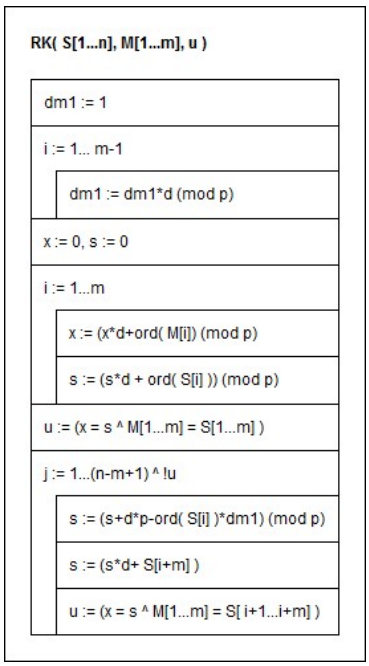
   Megoldásként s_{i+1}-et két lépésben számoljuk:

$$s := (s_i + d \cdot p - ord(S[i]) \cdot d^{m-1}) \mod p$$

$$s_{i+1} := (s \cdot d + ord(S[i+1])) \mod p$$

A fentiek alapján az algoritmus a következő (ld. 10. ábra)

1. Kiszámoljuk d^{m-1}-et ($dm1$)
2. Egy iterációban meghatározzuk Horner-módszerrel a minta számait (x) és s_0-t
3. Ellenőrizzük, hogy egyeznek-e
4. Addig számolgatjuk s_i értékét míg a minta nem egyezik s_i-vel, vagy a minta a szöveg végére nem ért.



ábra 10: RK

Chapter 15

15

Záróvizsga tételsor
15. Operációs rendszerek

Fekete Dóra

Operációs rendszerek
Folyamatok megvalósítása, ütemező algoritmusaik. Párhuzamosság, kritikus szekciók, kölcsönös kizárás megvalósítása. Szemaforok, osztott memória, üzenetküldés. Be- és kimeneti eszközök ütemezési lehetőségei, holtpontok. Memóriakezelés, virtuális memória fogalma. Lapozás és szegmentálás. Lapcserélési algoritmusok. Lemezterület-szervezés, redundáns tömbök, fájlrendszerek szolgáltatásai és jellemző megvalósításaik.

Operációs rendszer: olyan programrendszer, amely a számítógépes rendszerben a programok végrehajtását vezérli: így például ütemezi a programok végrehajtását, elosztja az erőforrásokat, biztosítja a felhasználó és a számítógépes rendszer közötti kommunikációt. Olyan program, ami egyszerű felhasználói felületet nyújt, eltakarva a számítógép(rendszer) eszközeit.

1 Folyamatok megvalósítása, ütemező algoritmusaik

1.1 Folyamatok megvalósítása

Program: fájlrendszerben egy bájthalmaz.
Folyamat(processz): futó program a memóriában (kód + I/O adatok + állapot).
Egyszerre hány folyamat működik?
Valódi multitask esetén egymástól teljesen függetlenül mennek a folyamatok. Operációs rendszerek egy szekvenciális modellt követnek, ami álmultitasking. Itt egyidejűleg a memóriához csak egy folyamat fér hozzá, gyorsan váltogat a dolgok között (multiprogramozás).
Az operációs rendszernek ahhoz, hogy ezeket a folyamatokat helyesen tudja kezelni, felügyelnie kell a folyamatokat. Az operációs rendszerünk így minden egyes folyamatot nyilvántart, és az operációs rendszer lelkének is nevezett ütemező (scheduler) segítségével szépen sorban minden egyes folyamatnak ad egy kis processzor- (CPU) időszeletet, amíg az adott folyamat dolgozik, azaz a processzorra kerülhet.
Rendszer modell: 1 processzor + 1 rendszer memória + 1 I/O eszköz = 1 feladat-végrehajtás
Interaktív (ablakos) rendszerekben több program, processz fut.

- Környezetváltásos rendszer: csak az előtérben lévő alkalmazás fut
- Kooperatív rendszer: az aktuális processz bizonyos időközönként, vagy időkritikus műveletnél önként lemond a CPU-ról (Win 3.1)
- Preemptív rendszer: az aktuális processztől a kernel bizonyos idő után elveszi a vezérlést, és a következő várakozó folyamatnak adja.
- Real time rendszer: egy operációs rendszer nyújtson lehetőséget az időfaktor figyelembe vételéhez. A mai OR-ek kezdenek ilyen tulajdonságokkal is kiegészülni.

Folyamatok létrehozása: ma tipikusan preemptív rendszereket használunk. Létrehozás oka lehet: rendszer inicializálás, folyamatot eredményező rendszerhívás (másolat az eredetiről [fork], az eredeti cseréje [execve]), felhasználói kérés (parancs&), nagy rendszerek kötegelt feladatai.
A folyamatok futhatnak az előtérben, illetve a háttérben. Ez utóbbiakat hívjuk démonoknak.
Folyamatok kapcsolata: Szülő-gyermek kapcsolat, folyamatfa: egy folyamatnak egy szülője van, egy folyamatnak viszont lehet több gyereke is, vannak összetartozó folyamatcsoportok.
Reinkarnációs szerver: meghajtó programok, kiszolgálók elindítója, ha elhal az egyik, akkor azt újraszüli, reinkarnálja.
Folyamatok befejezése: Egy folyamat az elindulása után a megadott időkeretben (el)végzi a feladatát. A befejezés lehet önkéntes vagy önkéntelen.

- Önkéntes befejezések: Szabályos kilépés (exit, return stb.), Kilépés valamilyen hiba miatt, amit a program felfedez (szintén pl. return utasítással).
- Önkéntelen befejezések: Illegális utasítás, végzetes hiba (0-val osztás, nem létező memória használat, stb), Külső segítséggel: Másik processz, netán mi „lőjük" ki az adott folyamatot.

Folyamatok állapota: A folyamat önálló programegység, saját utasításszámlálóval, veremmel stb. Általában nem függetlenek a folyamatok, egyik-másik eredményétől függ a tevékenység. Egy folyamat három állapotban lehet:

- Futó
- Futásra kész, ideiglenesen leállították, arra vár, hogy az ütemező CPU időt adjon a folyamatnak
- Blokkolt, ha logikailag nem lehet folytatni a tevékenységet, mert pl. egy másik eredményére vár. (catFradi.txt|grepFradi|sort, grep és sort blokkolt az elején)

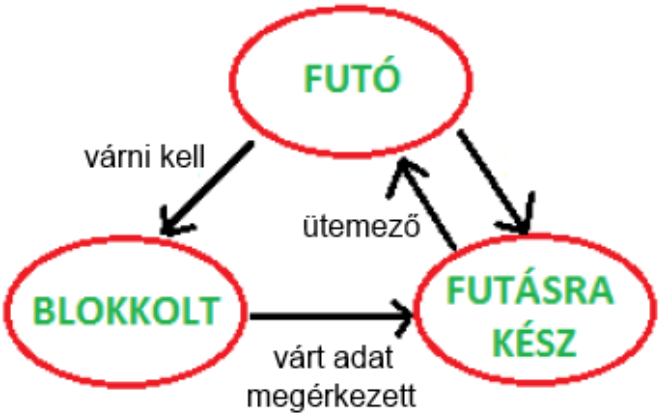


Figure 1: Állapotátmenetek

További állapotok: alvó, megállított, zombi (ha egy gyermek folyamat befejeződik, de szülője nem hív wait(&st) hívást, akkor a gyermek bent marad a processztáblában, init folyamat törli ezeket).
Folyamatok megvalósítása: A processzor csak végrehajtja az aktuális utasításokat. Egyszerre egy folyamat aktív. Folyamatokról nem tud a processzor. Ha lecseréljük az aktív folyamatot a következőre, akkor mindent meg kell őrizni, hogy visszatérhessünk a folytatáshoz. A „minden": utasításszámláló, regiszterek, lefoglalt memória állapot, nyitott fájl infók, stb. Ezeket az adatokat az úgynevezett folyamatleíró táblában tároljuk (processz tábla, processz vezérlő blokk). Van egy I/O megszakításvektor is.
A folyamatokat úgy is osztályozhatjuk, hogy számításigényes vagy input/output-igényes (avagy beviteli/kiviteli, röviden B/K vagy I/O) folyamatról van-e szó. Könnyen belátható, hogy a számításigényes folyamatoknak az a legjobb, ha az ütemező általában hosszú időperiódusra adja meg nekik a processzort, míg az input/output-igényes folyamatoknak a rövidebb periódusidő a megfelelőbb.

1.2 Ütemező algoritmusok

Folyamatok váltása: Kezdeményezheti időzítő, megszakítás, esemény, rendszerhívás kezdeményezés. Az ütemező elmenti az aktuális folyamat jellemzőket a folyamatleíró táblába. Betölti a következő folyamat állapotát, a processzor folytatja a munkát. Nem lehet menteni a gyorsítótárakat. Gyakori váltás többleterőforrást igényel. A folyamatváltási idő „jó" megadása nem egyértelmű.
Folyamatleíró táblázat(Process Control Block - PCB): A rendszer inicializálásakor létrejön; 1 elem, rendszerindító már bent van, mikor a rendszer elindul. Tömbszerű szerkezet (PID alapon), de egy-egy elem egy összetett processzus adatokat tartalmazó struktúra. Egy folyamat fontosabb adatai: Azonosítója (ID), neve (programnév); Tulajdonos, csoport azonosító; Memória, regiszter adatok stb.
Szálak: Önállóan működő programegységek (thread), egy folyamaton belüli különálló utasítássor. Általában egy folyamat = egy utasítássorozat = egy szál. Néha szükséges lehet, hogy egy folyamaton

belül „több utasítássorozat" legyen. Gyakran „lightweight process"-nek nevezik a több utasítássorozatos folyamatot. Egy folyamaton belül több egymástól „független" végrehajtási sor lehet. Lehet egy folyamaton belül egy szál vagy egy folyamaton belül több szál. Ez utóbbi esetben, ha egy szál blokkolódik, a folyamat is blokkolva lesz. Van külön száltáblázat.
Folyamatnak önálló címtartománya van, globális változói, megnyitott fájlleírói, gyermekfolyamatai, szignálkezelői, ébresztői stb., szálnak ezek nincsenek; viszont mindkettőnek vannak utasításszámlálói, regiszterei, van verme.

Egyszerre csak 1 folyamat tud futni. Az ütemező hozza meg a döntést. Ütemezési algoritmus alapján dönti el, hogy melyik fusson.
Folyamatváltás biztosan van: ha befejeződik egy folyamat, vagy ha egy folyamat blokkolt állapotba kerül (I/O vagy szemafor miatt). Általában van váltás: ha új folyamat jön létre, I/O megszakítás bekövetkezés (I/O megszakítás után jellemzően egy blokkolt folyamat, ami erre várt, folytathatja futását), időzítő megszakítás (nem megszakítható ütemezés, megszakítható ütemezés).
Ütemezések csoportosítása:

- Minden rendszerre jellemző a pártatlanság, hogy mindenki hozzáférhet a CPU-hoz, ugyanazok az elvek érvényesek mindenkire, és „azonos" terhelést kapnak.
- Kötegelt rendszerek: Áteresztőképesség, áthaladási idő, CPU kihasználtság
- Interaktív rendszerek: Válaszidő, megfelelés a felhasználói igényeknek
- Valós idejű rendszerek: Határidők betartása, adatvesztés, minőségromlás elkerülése

Ütemezés kötegelt rendszerekben:

- Sorrendi ütemezés, nem megszakítható
  - First Come First Served - (FCFS)
  - Egy folyamat addig fut, amíg nem végez, vagy nem blokkolódik.
  - Ha blokkolódik, a sor végére kerül.
  - Pártatlan, egyszerű, láncolt listában tartjuk a folyamatokat.
  - Hátránya: I/O igényes folyamatok nagyon lassan végeznek.
- Legrövidebb feladat először, nem megszakítható ez se (shortest job first – SJB)
  - Kell előre ismerni a futási időket
  - Akkor optimális, ha kezdetben mindenki elérhető
- Legrövidebb maradék futási idejű következzen
  - Megszakítható, minden új belépéskor vizsgálat.
- Háromszintű ütemezés
  - Bebocsátó ütemező: A feladatokat válogatva engedi be a memóriába.
  - Lemez ütemező: Ha a bebocsátó sok folyamatot enged be és elfogy a memória, akkor lemezre kell írni valamennyit, meg vissza. Ez ritkán fut.
  - CPU ütemező: A korábban említett algoritmusok közül választhatunk.

Ütemezés interaktív rendszerekben:

- Körben járó ütemezés – Round Robin
  - Mindenkinek időszelet, aminek végén, vagy blokkolás esetén jön a következő folyamat
  - Időszelet végén a körkörös listában következő lesz az aktuális folyamat
  - Pártatlan, egyszerű
  - Egy listában tárolhatjuk a folyamatokat (jellemzőit), és ezen megyünk körbe-körbe.

  - Időszelet mérete lehet probléma, mert a processz átkapcsolás időigényes. Ha kicsi az idő, akkor sok CPU megy el a kapcsolgatásra, ha pedig túl nagy, akkor interaktív felhasználóknak lassúnak tűnhet pl a billentyűkezelés.

- Prioritásos ütemezés
  - Fontosság, prioritás bevezetése. Unix: 0-49 nem megszakítható (kernel) prioritás, 50-127 user prioritás
  - Legmagasabb prioritású futhat. Dinamikus prioritás módosítás, különben éhenhalás
  - Prioritási osztályok használata. Egy osztályon belül Round Robin. Ki kell igazítani a folyamatok prioritását, különben az alacsonyak nagyon ritkán jutnak CPU-hoz. Tipikusan minden 100 időszeletnél a prioritásokat újraértékeli és ilyenkor jellemzően a magas prioritások alacsonyabbra kerülnek, majd ezen a soron megy RR. A végén újra felállnak az eredeti osztályok.

- Többszörös sorok
  - Szintén prioritásos és RR
  - Legmagasabb szinten minden folyamat 1 időszeletet kap
  - Következő 2-t, majd 4-et, 8, 16,32,64-et.
  - Ha elhasználta a legmagasabb szintű folyamat az idejét egy szinttel lejjebb kerül.

- Legrövidebb folyamat előbb
  - Bár nem tudjuk a hátralévő időt, de becsüljük meg az előzőekből.
  - Öregedés, súlyozott átlag az időszeletre: T0, T0/2+T1/2, T0/4+T1/4+T2/2, T0/8+T1/8+T2/4+T3/2

- Garantált ütemezés
  - Minden aktív folyamat arányos CPU időt kap.
  - Nyilván kell tartani, hogy egy folyamat már mennyi időt kapott, ha valaki arányosan kevesebb időt kapott, az kerül előbbre.

- Sorsjáték ütemezés
  - Mint az előző, csak a folyamatok között „sorsjegyeket" osztunk szét, az kapja a vezérlést, akinél a kihúzott jegy van
  - Arányos CPU időt könnyű biztosítani, hasznos pl. video szervereknél

- Arányos ütemezés
  - Vegyük figyelembe a felhasználókat is. Olyan, mint a garantált, csak a felhasználókra vonatkoztatva.

Ütemezés valós idejű rendszerekben
A valós idejű rendszerben az idő kulcsszereplő. Garantálni kell adott határidőre a tevékenység, válasz megadását.

- Hard Real Time (szigorú), abszolút, nem módosíthatóhatáridők.
- Soft Real Time (toleráns), léteznek a határidők, de ezek kis mértékű elmulasztása tolerálható.

A programokat több kisebb folyamatra bontják. Külső esemény észlelésekor, adott határidőre válasz kell.
Ütemezhető: ha egységnyi időre eső n esemény CPU válaszidő összege <=1.
Gyakori a gyermek folyamatok jelenléte a rendszerben. A szülőnek nem biztos, hogy minden gyermekével azonos prioritásra van szüksége. Tipikusan a kernel *prioritásos ütemezést* használ (+RR):

- Biztosít egy rendszerhívást, amivel a szülő a gyermek prioritását adhatja meg
- Kernel ütemez – felhasználói folyamat szabja meg az elvet, prioritást.

Szálütemezés:

- Felhasználói szintű szálak

  - Kernel nem tud róluk, a folyamat kap időszeletet, ezen belül a szálütemező dönt ki fusson
  - Gyors váltás a szálak között
  - Alkalmazásfüggő szálütemezés lehetséges
- Kernel szintű szálak
  - Kernel ismeri a szálakat, kernel dönt melyik folyamat melyik szála következzen
  - Lassú váltás, két szál váltása között teljes környezetátkapcsolás kell
  - Ezt figyelembe is veszik.

2 Párhuzamosság, kritikus szekciók, kölcsönös kizárás megvalósítása

2.1 Párhuzamosság és megvalósítása

Ütemező a folyamatok gyors váltogatásával „teremt" párhuzamos végrehajtás érzetet.
Többprocesszoros rendszerekben több processzor van egy gépben, nagyobb a teljesítmény, de a megbízhatóságot általában nem növeli.
Klaszterek: megbízhatóság növelése elsősorban a cél.

2.2 Kritikus szekciók

Azokat az utasításokat, azt a programrészt, amelyben a programunk a közös adatokat használja, kritikus területnek, kritikus szekciónak vagy kritikus blokknak nevezzük.
Kulcskérdés: a közös erőforrások használata, amikor két folyamat ugyanazt a memóriát használja. Kritikus programterület, szekció, az a rész, mikor a közös erőforrást (memóriát) használjuk.
Versenyhelyzet: két vagy több folyamat közös memóriát ír vagy olvas, a végeredmény a futási időpillanattól függ. Nehezen felderíthető hibát okoz.
Megoldás: Módszer, ami biztosítja, hogy a közös adatokat egyszerre csak egy folyamat tudja használni.

2.3 Kölcsönös kizárás és megvalósítása

Kölcsönös kizárásnak nevezzük azt a módszert, ami biztosítja, hogy ha egy folyamat használ valamilyen megosztott, közös adatot, akkor más folyamatok ebben az időben ne tudják azt elérni.
A jó kölcsönös kizárás az alábbi feltételeknek felel meg:

- Nincs két folyamat egyszerre a kritikus szekciójában.
- Nincs sebesség, CPU paraméter függőség.
- Egyetlen kritikus szekción kívül levő folyamat sem blokkolhat másik folyamatot.
- Egy folyamat sem vár örökké, hogy a kritikus szekcióba tudjon belépni.

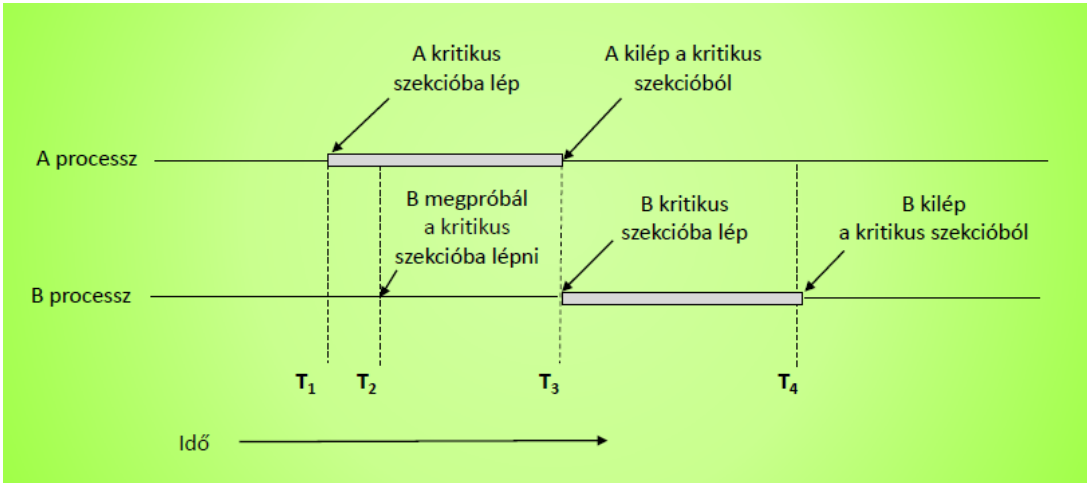


Figure 2: A megkívánt kölcsönös kizárás viselkedése

2.3.1 Megvalósítások

- Megszakítások tiltása (összes): Belépéskor az összes megszakítás tiltása, Kilépéskor azok engedélyezése. Ez nem igazán jó, mivel a felhasználói folyamatok kezében lenne a megszakítások tiltása, persze a kernel használja.

- Osztott, ún. zárolás változó használata: 0 (senki) és 1 (valaki) kritikus szekcióban van. Két folyamat is kritikus szekcióba tud kerülni! Egyik folyamat belép a kritikus szekcióba, de éppen az 1-re állítás előtt a másik folyamat kerül ütemezésre.

- Szigorú váltogatás: Több folyamatra is általánosítható. A kölcsönös kizárás feltételeit teljesíti a 3. kivételével, ugyanis ha pl 1 folyamat a lassú, nem kritikus szekcióban van, és a 0 folyamat gyorsan belép a kritikus szekcióba, majd befejezi a nem kritikus szekciót is, akkor ez a folyamat blokkolódik mert a kovetkezo=1 lesz!(Saját magát blokkolja!)

◦ 0. folyamat

```
while(1)
 {
  while(kovetkezo!=0) ;
  kritikus_szekcio();
  kovetkezo=1;
  nem_kritikus_szekcio();
 }
```

1.folyamat

```
while(1)
 {
  while(kovetkezo!=1) ;
  kritikus_szekcio();
  kovetkezo=0;
  nem_kritikus_szekcio();
 }
```

Figure 3: Szigorú váltogatás

- G. L. Peterson javította a szigorú váltogatást. A kritikus szekció előtt minden folyamat meghívja a belépés, majd utána kilépés függvényt.

```
#define N  2
int kovetkezo;
int akarja[N];
/* a módosított folyamat*/
while(1)
 {
  belepes(processz);
  kritikus_szekcio();
  kilepes(processz);
  nem_kritikus_szekcio();
 }
```

```
void belepes(int proc)
{
  int masik;
  masik=1-proc; //mivel N=2...
  // masik=(proc+1) % N;
  akarja[proc]=1; //processz futni akar
  kovetkezo=proc;
  while(  kovetkezo==proc &&
          akarja[masik]);
}
void kilepes(int proc)
{
  akarja[proc]=0;   //hamis
}
```

```
void belepes(int proc)
{
  int masik;
  masik=1-proc; //mivel N=2...
  // masik=(proc+1) % N;
 kovetkezo=proc; //két sor csere
//* itt van ütemező váltás
 akarja[proc]=1; //proc futni akar
  while(  kovetkezo==proc &&
          akarja[masik]);
}
void kilepes(int proc)
{
  akarja[proc]=0;   //hamis
}
```

Figure 4: Peterson javítása és a benne rejlő hiba

Ez a javítás viszont nagy hibát okozhat. Tegyük fel proc=0. A jelölt ütemezés váltásnál a proc=1 belépése jön. Mivel akarja[0] értéke 0, ezért az 1-es process belép a kritikus szakaszba! Ekkor újra váltson az ütemező, akarja[1]=1, a következő értéke szintén 1, így a következo==proc hamis, azaz a 0. proc is belép a kritikus szakaszba!

- Tevékeny várakozás gépi kódban: TSL utasítás, Test and Set Lock. Ez atomi művelet, vagyis megszakíthatatlan.

- Tevékeny várakozás: A korábbi Peterson megoldás is, a TSL használata is jó, csak ciklusban várakozunk. A korábbi megoldásokat, tevékeny várakozással (aktív várakozás) megoldottnak hívjuk, mert a CPU-t „üres" ciklusban járatjuk a várakozás során.De ez a CPU időt pazarolja. Helyette jobb lenne az, ha a kritikus szekcióba lépéskor blokkolna a folyamat, ha nem szabad belépnie. Az

aktív várakozás nem igazán hatékony. Megoldás: blokkoljuk(alvás) várakozás helyett a folyamatot, majd ha megengedett ébresszük fel. Különböző paraméter megadással is implementálhatók. Tipikus probléma: Gyártó-Fogyasztó probléma vagy másképp korlátos tároló probléma. PL: Pék-pékség-Vásárló háromszög:

- A pék süti a kenyeret, amíg a pékség polcain van hely.
- Vásárló tud venni, ha a pékség polcain van kenyér.
- Ha tele van kenyérrel a pékség, akkor „a pék elmegy pihenni".
- Ha üres a pékség, akkor a vásárló várakozik a kenyérre.

Pék folyamat

```
#define N 100
int hely=0;
void pék()
{
  int kenyér;
  while(1)
   {
          kenyér=új_kenyér()
          if (hely==N) alvás();
          polcra(kenyér);
          hely++;
          if (hely==1)
              ébresztő(vásárló);
   }
}
```

Vásárló folyamat

```
 void vásárló()
{
  int kenyér;
  while(1)
   {
          if (hely==0) alvás();
          kenyér=kenyeret();
          hely--;
          if (hely==N-1)
              ébresztő(pék);
          megesszük(kenyér);
   }
}
```

Figure 5: Gyártó-fogyasztó probléma egy megvalósítása

Ennél a megvalósításnál probléma lehet, hogy A „hely" változó elérése nem korlátozott, így ez okozhat versenyhelyzetet.

- Vásárló látja, hogy a hely 0 és ekkor az ütemező átadja a vezérlést a péknek, aki süt egy kenyeret. Majd látja, hogy a hely 1, ébresztőt küld a vásárlónak. Ez elveszik, mert még a vásárló nem alszik.
- Vásárló visszakapja az ütemezést, a helyet korábban beolvasta, az 0, megy aludni.
- A pék az első után megsüti a maradék N-1 kenyeret és ő is aludni megy

Lehet ébresztő bittel javítani, de több folyamatnál a probléma nem változik.

3 Szemaforok, osztott memória, üzenetküldés

3.1 Szemaforok

E.W. Dijkstra(1965) javasolta ezen új változótípus bevezetését.

- Ez valójában egy egész változó.
- A szemafor tilosat mutat, ha értéke 0. Ekkor a folyamat elalszik, megáll a tilos jelzés előtt.
- Ha a szemafor >0, szabad a pálya, beléphetünk a kritikus szakaszra. Két művelet tartozik hozzá: ha beléptünk, csökkentjük szemafor értékét (down); ha kilépünk, növeljük a szemafor értékét (up). Ezeket Dijkstra P és V műveletnek nevezte.

Elemi művelet: a szemafor változó ellenőrzése, módosítása, esetleges elalvás oszthatatlan művelet, nem lehet megszakítani. Ez garantálja, hogy ne alakuljon ki versenyhelyzet.
Ha a szemafor tipikus vasutas helyzetet jelöl, azaz 1 vonat mehet át csak a jelzőn, a szemafor értéke ekkor 0 vagy 1 lehet. Ez a *bináris szemafor*, vagy más néven MUTEX (Mutual Exclusion), és kölcsönös kizárásra használjuk.
Rendszerhívással, felhasználói szinten nem biztosítható a műveletek atomiságának megvalósítása. Művelet elején például letiltunk minden megszakítást. Ha több CPU van, akkor az ilyen szemafort védeni tudjuk a TSL utasítással. Viszont ezek a szemafor műveletek kernel szintű, rendszerhívás műveletek. A fejlesztői környezetek biztosítják.

Gyártó (pék) függvénye

```
typedef int szemafor;
szemafor szabad=1;  /*Bináris szemafor,1 mehet tovább, szabad a jelzés*/
szemafor üres=N, tele=0;  /* üres a polc, ez szabad jelzést mutat*/
void pék()                /* N értéke a „kenyerespolc" mérete */
{
  int kenyér;
  while (1)
   {
     kenyér=pék_süt();
     down(&üres);          /* üres csökken, ha előtte>0, mehet tovább*/
     down(&szabad);        /* Piszkálhatjuk-e a pékség polcát? */
     kenyér_polcra(kenyér); /* Igen, betesszük a kenyeret. */
     up(&szabad);          /* Elengedjük a pékség polcát. */
     up(&tele);            /* Jelezzük vásárlónak, van kenyér. */
   }
}
```

Fogyasztó (Vásárló) függvénye.

```
void vásárló()            /* vásárló szemaforja a tele */
{
  int kenyér;
  while (1)
   {
     down(&tele);          /*tele csökken, ha előtte>0, mehet tovább*/
     down(&szabad);        /*Piszkálhatjuk-e a pékség polcát? */
     kenyér=kenyér_polcról(); /* Igen, levesszük a kenyeret. */
     up(&szabad);          /* Elengedjük a pékség polcát. */
     up(&üres);            /* Jelezzük péknek, van hely, lehet sütni. */
     kenyér_elfogyasztása(kenyér);
   }
}
```

Figure 6: Gyártó-fogyasztó probléma megoldása szemaforokkal

Szabad: kenyér polcot (boltot) védi, hogy egy időben csak egy folyamat tudja használni (vagy a pék, vagy a vásárló): Kölcsönös kizárás, Elemi műveletek (up, down).
Tele, üres szemafor: szinkronizációs szemaforok, a gyártó álljon meg ha a tároló tele van, illetve a fogyasztó is várjon ha a tároló üres.

Szemafornál két utasítás felcserélése is gondot okozhat.
Monitor: magasabb szintű, nyelvű konstrukció. Eljárások, adatszerkezetek lehetnek benne. Egy időben csak egy folyamat lehet aktív a monitoron belül.

3.2 Osztott memória

Elosztott közös memória: Hálózatban futó folyamatok közti memóriamegosztás.
Lehetőségünk van egy programon belüli különböző folyamatok, szálak által használt memóriaterületek közössé tételére, vagyis használhatjuk ugyanazt a memóriarészt, „időosztásos" üzemmódban. Különböző, egymással valamilyen módon „összekapcsolt" programrészekhez, folyamokhoz, szálakhoz ugyanazt a memóriaterület kapcsoljuk.

3.3 Üzenetküldés

A folyamatok jellemzően két primitívet használnak: *Send(célfolyamat, üzenet)*, *Receive(forrás, üzenet)*. Ezek rendszerhívások, nem nyelvi konstrukciók.
Ha küldő-fogadó nem azonos gépen van, szükséges ún. nyugtázó üzenet. Így ha küldő nem kapja meg a nyugtát, ismét elküldi az üzenetet. Ha a nyugta veszik el, a küldő újra küld. Ismételt üzenetek megkülönböztetése sorszám segítségével történik.
Összegzés: Ideiglenes tároló helyek (levelesláda) létrehozása mindkét helyen. El lehet hagyni, ekkor ha send előtt van receive, a küldő blokkolódik, illetve fordítva. Ezt hívják randevú stratégiának. Például a Minix 3 is randevút használ, rögzített méretű üzenetekkel.
Adatcső kommunikáció hasonló, csak az adatcsőben nincsenek üzenethatárok, ott csak bájtsorozat van.
Az üzenetküldés a párhuzamos rendszerek általános technikája. Pl. MPI
Klasszikus IPC (inter-process communication) problémák:

- Étkező filozófusok esete: körben felváltva 5 villa, tányér. 2 villa kell a spagetti evéshez. A tányér melletti villákra pályáznak. Esznek-gondolkoznak. A legegyszerűbb megoldás (végtelen ciklusban gondolkodik, felveszi a két villáját egymás után, eszik, leteszi a villákat) pár hibát rejt magában, hogy pl. holtpont lehet, ha egyszerre megszerzik a bal villát és mind várnak a jobbra. Illetve ha

leteszi a bal villát és újra próbálkozik, még az se az igazi, hiszen folyamatosan felveszik a bal villát, majd leteszik. (Éhezés)

```
Int N=5;
Szemafor villa[]={1,1,1,1,1}; //mind
szabad
Szemafor max=4; //max 4 villa használt
                        //egyszerre
Void filozófus(int i)
{
 while(1)
  {
        gondolkodom();
        down(max);
        down(villa[i]);   // bal villa
        down(villa[(i+1)%N];   //jobb
        eszem();
        up(villa[i]);
        up(villa[(i+1)%N]);
        up(max);
  }
}
```

Figure 7: Étkező filozófusok, javított megoldás: 5 villa szemafor van, és egy maximum. Korlátozott erőforrás megszerzésre példa

- Író-olvasó probléma: Adatbázist egyszerre többen olvashatják, de csak 1 folyamat írhatja.

```
// író folyamat
Szemafor database=1;
Szemafor mutex=1;
int rc=0;
Void író()
{
 while(1)
 {
    csinál_valamit();
    down(database);   // kritikus
    írunk_adatbázisba();
    up(database);
 }
}
```

```
Void olvasó()
{
 while(1)
 {
    down(mutex);
    rc++;
    if (rc==1) down(database);
    up(mutex);
    olvas_adatbázisból();
    down(mutex);   // kritikus
    rc--;
    if (rc==0) up(database);
    up(mutex);
    adatot_feldolgozunk();
 }
}
```

Figure 8: Író-olvasó probléma megvalósítása

4 Be- és kimeneti eszközök ütemezési lehetőségei, holtpontok

4.1 Be- és kimeneti eszközök ütemezési lehetőségei

Input-Output eszközök:

- *Blokkos eszközök.* Adott méretű blokkban tároljuk az információt. Blokkméret 512 byte - 32768 byte között. Egymástól függetlenül írhatók vagy olvashatók. Blokkonként címezhetőek. Ilyen eszköz: HDD, szalagos egység
- **Karakteres eszközök**. Nem címezhető, csak jönnek-mennek sorban a „karakterek" (bájtok)
- **Időzítő**: kivétel, nem blokkos és nem karakteres

Megszakítások: Általában az eszközöknek van állapotbitjük, jelezve, hogy az adat készen van.
Megszakítás használat (IRQ):

1. CPU tevékenység megszakítása
2. a kért sorszámú kiszolgáló végrehajtása

3. a kívánt adat beolvasása, a szorosan hozzátartozó tevékenység elvégzése
4. visszatérés a megszakítás előtti állapothoz.

Közvetlen memória elérés(DMA): Direct Memory Access, tartalmaz memória cím regisztert, átvitel irány jelzésre, mennyiségre regisztert. Ezeket szabályos in, out portokon lehet elérni.
Működés lépései:

1. CPU beállítja a DMA vezérlőt. (Regisztereket.)
2. A DMA a lemezvezérlőt kéri a megadott műveletre.
3. Miután a lemezvezérlő beolvasta a pufferébe, a rendszersínen keresztül a memóriába(ból) írja, olvassa az adatot.
4. Lemezvezérlő nyugtázza, hogy kész a kérés teljesítése.
5. DMA megszakítással jelzi, befejezte a műveletet.

4.2 Holtpontok (deadlock)

Két vagy több folyamat egy erőforrás megszerzése során olyan helyzetbe kerül, hogy egymást blokkolják a további végrehajtásban. Pontos definíció: Folyamatokból álló halmaz holtpontban van, ha minden folyamat olyan eseményre vár, amit csak a halmaz egy másik folyamata okozhat. Nem csak az I/O eszközökhöz kötődik, pl párhuzamos rendszerek, adatbázisok, stb.
Coffman E.G. szerint 4 feltétel szükséges a holtpont kialakulásához:

1. *Kölcsönös kizárás feltétel.* Minden erőforrás hozzá van rendelve 1 folyamathoz vagy szabad.
2. *Birtoklás és várakozás feltétel.* Korábban kapott erőforrást birtokló folyamat kérhet újabbat.
3. *Megszakíthatatlanság feltétel.* Nem lehet egy folyamattól elvenni az erőforrást, csak a folyamat engedheti el.
4. *Ciklikus várakozás feltétel.* Két vagy több folyamatlánc kialakulása, amiben minden folyamat olyan erőforrásra vár, amit egy másik tart fogva.

Irányított gráffal lehet modellezni a folyamatokat és erőforrásokat. Ha van kör, akkor az holtpontot jelent.

Stratégiék holtpont esetén:

1. A probléma figyelmen kívül hagyása.
   - Nem törődünk vele, nagy valószínűséggel Ő sem talál meg bennünket.
   - Ezt a módszert gyakran strucc algoritmus néven is ismerjük.
   - Vizsgálatok szerint a holtpont probléma és az egyéb (fordító, op.rendszer, hw, sw hiba) összeomlások aránya 1:250.
   - A Unix, Windows világ is ezt a „módszert” használja. De túl nagy az ár a várható haszonért cserébe.
2. Felismerés és helyreállítás.
   - Engedjük a holtpontot megjelenni (kör), ezt észrevesszük és cselekszünk.
   - Folyamatosan figyeljük az erőforrás igényeket, elengedéseket.
   - Kezeljük az erőforrás gráfot folyamatosan. Ha kör keletkezik, akkor egy körbeli folyamatot megszüntetünk.
   - Másik módszer, nem foglalkozunk az erőforrás gráffal, ha x (fél óra?) ideje blokkolt egy folyamat, egyszerűen megszüntetjük. Ez nagygépes rendszereknél ismert módszer.
3. Megelőzés.
   - A 4 szükséges feltétel egyikének meghiúsítása.
   - A Coffman féle 4 feltétel valamelyikére mindig él egy megszorítás.

- Kölcsönös kizárás. Ha egyetlen erőforrás soha nincs kizárólag 1 folyamathoz rendelve, akkor nincs holtpont se. De ez nehézkes, míg pl. nyomtató használatnál a nyomtató démon megoldja a problémát, de ugyanitt a nyomtató puffer egy lemezterület, itt már kialakulhat holtpont.
- Ha nem lehet olyan helyzet, hogy erőforrásokat birtokló folyamat további erőforrásra várjon, akkor szintén nincs holtpont. Ezt kétféle módon érhetjük el: Előre kell tudni egy folyamat összes erőforrásigényét vagy ha erőforrást akar egy folyamat, először engedje el az összes birtokoltat.
- A Coffman féle harmadik feltétel a megszakíthatatlanság. Ennek elkerülése eléggé nehéz. Pl nyomtatás közben nem szerencsés a nyomtatót másnak adni.
- A negyedik feltétel, a ciklikus várakozás már könnyebben megszüntethető. Egyszerű mód: Minden folyamat egyszerre csak 1 erőforrást birtokolhat. Másik módszer: Sorszámozzuk az erőforrásokat, és a folyamatok csak ezen sorrendben kérhetik az erőforrásokat. Ez jó elkerülési mód, csak megfelelő sorrend nincs.

4. Dinamikus elkerülés.

   - Erőforrások foglalása csak „óvatosan".
   - Bankár algoritmus (Dijkstra, 1965) Mint a kisvárosi bankár hitelezési gyakorlata. A bankár algoritmus minden kérés megjelenésekor azt nézi, hogy a kérés teljesítése biztonságos állapothoz vezet-e. Ha igen, jóváhagyja, ha nem, a kérést elhalasztja. Eredetileg 1 erőforrásra tervezett. A korábbi megelőzés is, meg ez az elkerülés is olyan információt kér (az erőforrás pontos igényeket, a folyamatok számát előre), ami nehezen megadható. (Folyamatok dinamikusan jönnek létre, erőforrások dinamikusan módosulnak.) Ezért a gyakorlatban kevesen alkalmazzák.
   - Biztonságos állapotok, olyan helyzetek, melyekből létezik olyan kezdődő állapotsorozat, melynek eredményeként mindegyik folyamat megkapja a kívánt erőforrásokat és befejeződik.

5 Memóriakezelés, virtuális memória fogalma

Monoprogramozás: A legegyszerűbb memóriakezelési módszer, időben csak egyetlen programot futtatunk.
Multiprogramozás: multiprogramozás rögzített partíciókkal: Ekkor a rendelkezésre álló memóriát felosztják több, általában nem egyforma hosszúságú részre. Működéséhez minden partíciónak szüksége van egy úgynevezett várakozási sorra. Ha beérkezik egy igény, az operációs rendszer berakja annak a legkisebb méretű, partíciónak nevezett rész várakozási sorába, ahová még befér. Ilyenkor elvész a partíciónak az a része – nem használható –, amit az éppen futó processz nem használ. Ha az adott szeletben befejeződik a munka, a legrégebben várakozó megkapja a területet, és elkezd működni.

5.1 Memóriakezelés

A memóriakezelő az operációs rendszer része, gyakran a kernelben. Feladata: a memória nyilvántartása, melyek szabadok, foglaltak; memóriát foglaljon folyamatok számára; memóriát felszabadítson; csere vezérlése a RAM és a (Merev)Lemez között.
Kétféle algoritmus csoport:

- Szükséges a folyamatok mozgatása, cseréje a memória és a lemez között. (swap)
- Nincs erre szükség, ha elegendő memória van.

Multiprogramozás rögzített memóriaszeletekkel: Osszuk fel a memóriát n (nem egyenlő) szeletre. (Fix szeletek) Pl. rendszerindításnál ez megtehető. Egy közös várakozási sor van, minden szeletre külön-külön várakozási sor. Kötegelt rendszerek tipikus megoldása.
Relokáció: Nem tudjuk hova kerül egy folyamat, így a memória hivatkozások nem fordíthatók fix értékekre.
Védelem: Nem kívánatos, ha egy program a másik memóriáját „éri el".
Másik megoldás: Bázis+határregiszter használata, ezeket a programok nem módosíthatják, de minden címhivatkozásnál ellenőrzés: lassú.
Multiprogramozás memóriacsere használattal: A korábbi kötegelt rendszerek tipikus megoldása a rögzített

memória szeletek használata (IBM OS/MFT). Időosztásos, grafikus felületek esetén ez nem az igazi. Itt a teljes folyamatot mozgatjuk a memória-lemez között. Nincs rögzített memória partíció, mindegyik dinamikusan változik, ahogy az op. rendszer oda-vissza rakosgatja a folyamatokat. Dinamikus, jobb memória kihasználtságú lesz a rendszer, de a sok csere lyukakat hoz létre, ezért memória tömörítést kell végezni. (Sok esetben ez az időveszteség nem megengedhető.)
Dinamikus memóriafoglalás: Általában nem ismert, hogy egy programnak mennyi dinamikus adatra, veremterületre van szüksége. A program „kód" része fix szeletet kap, míg az adat és verem része változót. Ezek tudnak nőni (csökkenni). Ha elfogy a memória, akkor a folyamat leáll, vár a folytatásra, vagy kikerül a lemezre, hogy a többi még futó folyamat memóriához jusson. Ha van a memóriában már várakozó folyamat, az is cserére kerülhet.
A „dinamikus" memória nyilvántartása:

- Allokációs egység definiálása. Ennek mérete kérdés. Ha kicsi, akkor kevésbé lyukasodik a memória, viszont nagy a nyilvántartási „erőforrás (memória) igény". Ha nagy az egység, akkor túl sok lesz az egységen belüli maradékokból adódó memóriaveszteség.
- A nyilvántartás megvalósítása: Bittérkép használattal vagy Láncolt lista használattal. Ha egy folyamat befejeződik, akkor szükség lehet az egymás melletti memória lyukak egyesítésére. Külön lista a lyukak és folyamatok listája.

Memóriafoglalási stratégiák: Új vagy swap partícióról behozott folyamat számára, több memória elhelyezési algoritmus ismert (hasonlóak a lemezhez):

- First Fit (első helyre, ahova befér, leggyorsabb, legegyszerűbb)
- Next Fit (nem az elejéről, hanem az előző befejezési pontjából indul a keresés, kevésbé hatékony, mint a first fit)
- Best Fit (lassú, sok kis lyukat produkál)
- Worst Fit (nem lesz sok kis lyuk, de nem hatékony)
- Quick Fit (méretek szerinti lyuklista, a lyukak összevonása költséges)

5.2 Virtuális memória fogalma

Multiprogramozás virtuális memóriahasználattal: Egy program használhat több memóriát, mint a rendelkezésre álló fizikai méret. Az operációs rendszer csak a „szükséges részt" tartja a fizikai memóriában. Egy program a „virtuálismemória-térben" tartózkodik. Az elv akár a monoprogramozás környezetben is használható.
Memory Management Unit (MMU): A virtuális címtér „lapokra" van osztva. (Ezt laptáblának nevezzük). Van jelenlét/hiány bit. Virtuális-fizikai lapokat összerendeljük. Ha az MMU látja, hogy egy lap nincs a memóriában, laphibát okoz, op.rendszer kitesz egy lapkeretet, majd behozza a szükséges lapot.

6 Lapozás és szegmentálás

6.1 Lapozás

Lapozástervezési szempontok:

- Munkahalmaz modell
  - A szükséges lapok betöltése. (Induláskor-előlapozás) A folyamat azon lapjainak fizikai memóriában tartása, melyeket használ. Ez az idővel változik.
  - Nyilván kell tartani a lapokat. Ha egy lapra az utolsó N időegységben nem hivatkoznak, laphiba esetén kidobjuk.
  - Óra algoritmus javítása: Vizsgáljuk meg, hogy a lap eleme a munkahalmaznak? (WSClock algoritmus)
- Lokális, globális helyfoglalás

  - Egy laphibánál ha az összes folyamatot (globális), vagy csak a folyamathoz tartozó (lokális) lapokat vizsgáljuk.
  - Globális algoritmus esetén minden folyamatot elláthatunk méretéhez megfelelő lappal, amit aztán dinamikusan változtatunk.
  - Page Fault Frequency (PFF) algoritmus, laphiba/másodperc arány, ha sok a laphiba, növeljük a folyamat memóriában lévő lapjainak a számát. Ha sok a folyamat, akár teljes folyamatot lemezre vihetünk.(terhelosztás)

- Helyes lapméret meghatározása.
  - Kicsi lapméret: A „lapveszteség" kicsi, viszont nagy laptábla kell a nyilvántartáshoz.
  - Nagy lapméret: Fordítva, „lapveszteség" nagy, kicsi laptábla.
  - Jellemzően: n x 512 bájt a lapméret, XP, Linuxok 4KB a lapméret. 8KB is használt (szerverek).

- Közös memória
  - Foglalhatunk memóriaterületet, amit több folyamat használhat.
  - Elosztott közös memória: Hálózatban futó folyamatok közti memóriamegosztás.

6.2 Szegmentálás

Virtuális memória: egy dimenziós címtér, 0-tól a maximum címig (4,8,16 GB, ...).
Több programnak van dinamikus területe, melyek növekedhetnek, bizonytalan mérettel. Hozzunk létre egymástól független címtereket, ezeket szegmensnek nevezzük.
Ebben a világban egy cím 2 részből áll: szegmens szám, és ezen belüli cím. (eltolás)
Szegmentálás lehetővé teszi osztott könyvtárak „egyszerű" megvalósítását. Logikailag szét lehet szedni a programot, adat szegmens, kód szegmens stb. Védelmi szintet is megadhatunk egy szegmensre. Lapok fix méretűek, a szegmensek nem. Szegmens töredezettség megjelenése. Ez töredezettség-összevonással javítható.

7 Lapcserélési algoritmusok

Ha nincs egy virtuális című lap a memóriában, akkor egy lapot ki kell dobni, berakni ezt az új lapot. A processzor gyorsító tár (cache) memória használatnál, vagy a böngésző helyi gyorsítótáránál is hasonló a helyzet. Optimális lapcserélés: Címkézzünk meg minden lapot azzal a számmal, ahány CPU utasítás végrehajtódik, mielőtt hivatkozunk rá. Dobjuk ki azt a lapot, amelyikben legkisebb ez a szám. Egy baj van, nem lehet megvalósítani, viszont kétszeres futásnál tesztelési célokat szolgálhat.

- NRU (Not Recently Used) algoritmus:
  - Használjuk a laptábla bejegyzés módosítás (Modify) és hivatkozás (Reference) bitjét. A hivatkozás bitet időnként (óramegszakításnál, kb. 0.02 sec) állítsuk 0-ra, ezzel azt jelezzük, hogy az „utóbbi időben" volt-e használva, hivatkozva.
    * 0.osztály: nem hivatkozott, nem módosított
    * 1.osztály: nem hivatkozott, módosított. Ide akkor lehet kerülni, ha az óramegszakítás állítja 0-ra a hivatkozás bitet.
    * 2.osztály: hivatkozott, nem módosított
    * 3.osztály: hivatkozott, módosított
  - Válasszunk véletlenszerűen egy lapot a legkisebb nem üres osztályból. Egyszerű, nem igazán hatékony implementálni, megfelelő eredményt ad.

- FIFO lapcserélés. Egyszerű FIFO, más területekről is ismert, ha szükség van egy új lapra akkor a legrégebbi lapot dobjuk ki. Listában az érkezés sorrendjében a lapok, egy lap a lista elejére érkezik, és a végéről távozik. Ennek javítása a Második Lehetőség lapcserélő algoritmus.

- Második Lehetőség lapcserélő algoritmus. Olyan, mint a FIFO, csak ha a lista végén lévő lapnak a hivatkozás bitje 1, akkor kap egy második esélyt, a lista elejére kerül és a hivatkozás bitet 0-ra állítjuk.

- Óra algoritmus: olyan, mint a második lehetőség, csak ne a lapokat mozgassuk körbe egy listában, hanem rakjuk körbe őket és egy mutatóval körbe járunk. A mutató a legrégebbi lapra mutat. Laphibánál, ha a mutatott lap hivatkozás bitje 1, nullázzuk azt, és a következő lapot vizsgáljuk. Ha vizsgált lap hivatkozás bitje 0, akkor kitesszük.
- LRU (Least Recently Used) algoritmus: Legkevésbé (legrégebben) használt lap kidobása. HW vagy SW megvalósítás.
  - HW1: Vegyünk egy számlálót, ami minden memória hivatkozásnál 1-gyel nő. Minden laptáblában tudjuk ezt a számlálót tárolni. Minden memóriahivatkozásnál ezt a számlálót beírjuk a lapba. Laphibánál megkeressük a legkisebb számlálóértékű lapot.
  - HW2: LRU bitmátrix használattal, n lap, n x n bitmátrix. Egy k. lapkeret hivatkozásnál állítsuk a mátrix k. sorát 1-re, míg a k. oszlopát 0-ra. Laphibánál a legkisebb értékű sor a legrégebbi.
- NFU (Not Frequently Used) algoritmus:
  - Minden laphoz tegyünk egy számlálót. Minden óramegszakításnál ehhez adjuk hozzá a lap hivatkozás (R) bitjét.
  - Laphibánál a legkisebb számlálóértékű lapot dobjuk ki. (A leginkább nem használt lap)
  - Hiba, hogy az NFU nem felejt, egy program elején gyakran használt lapok megőrzik nagy értéküket.
  - Módosítsuk: Minden óramegszakításnál csináljunk jobbra egy biteltolást a számlálón, balról pedig hivatkozás bitet tegyük be (shr).(Öregítő algoritmus)
  - Ez jól közelíti az LRU algoritmust.
  - Ez a lapszámláló véges bitszámú (n), így n időegység előtti eseményeket biztosan nem tud megkülönböztetni.

8 Lemezterület-szervezés, redundáns tömbök, fájlrendszerek szolgáltatásai és jellemző megvalósításaik

8.1 Lemezterület-szervezés

8.2 Redundáns tömbök

RAID – Redundant Array of Inexpensive Disks. Ha operációs rendszer nyújtja, gyakran Soft Raidnek nevezik. Ha intelligens (külső) vezérlőegység nyújtja, gyakran Hardver Raid-nek, vagy csak Raid diszkrendszernek nevezik. Bár nevében olcsó (Inexpensive), valójában inkább nem az. Több lemezt fog össze, és egy logikai egységként látja az operációs rendszer. Többféle „összefogási" elv létezik: RAID 0-6.

- RAID 0(striping)
  - Ez az a Raid, ami nem is redundáns
  - Több lemez logikai összefűzésével egy meghajtót kapunk.
  - A lemezkapacitások összege adja az új meghajtó kapacitását.
  - A logikai meghajtó blokkjait szétrakja a lemezekre (striping), ezáltal egy fájl írása több lemezre kerül.
  - Gyorsabb I/O műveletek.
  - Nincs meghibásodás elleni védelem.
- RAID 1(tükrözés)
  - Két független lemezből készít egy logikai egységet.
  - Minden adatot párhuzamosan kiír mindkét lemezre.(Tükrözés, mirror)
  - Tárolókapacitás felére csökken.
  - Drága megoldás.

  - Jelentős hibatűrő képesség. Mindkét lemez egyszerre történő meghibásodása okoz adatvesztést.

- RAID 1+0, RAID 0+1
  - RAID 1+0: Tükrös diszkekből vonjunk össze többet.
  - RAID 0+1: RAID 0 összevont lemezcsoportból vegyünk kettőt.
  - A vezérlők gyakran nyújtják egyiket, másikat, mivel így is, úgy is tükrözés van, azaz drága, így ritkán használt.
- RAID 2: Adatbitek mellett hibajavító biteket is tartalmaz. (ECC: Error Correction Code) Pl. 4 diszkhez 3 javító diszk
- RAID 3: Elég egy plusz „paritásdiszk", n+1 diszk, $\sum n$ a kapacitás
- RAID 4: RAID 0 kiegészítése paritásdiszkkel.
- Ma ezen megoldások (RAID 2,3,4) nem gyakran használatosak.
- RAID 5
  - Nincs paritásdiszk, ez el van osztva a tömb összes elemére. (stripe set)
  - Adatok is elosztva kerülnek tárolásra.
  - Intenzív CPU igény (vezérlő CPU!!!)
  - Redundáns tárolás, 1 lemez meghibásodása nem okoz adatvesztést. A paritásbitből meg a többiből az egy eltűnt kiszámítható. 2 lemez egyidejű meghibásodása már okoz adatvesztést.
  - N lemez RAID 5 tömbben (N>=3), n-1 lemez méretű logikai meghajtót ad.
- RAID 6
  - A RAID 5 paritásblokkhoz, hibajavító kód kerül tárolásra.(+1 diszk)
  - Még intenzívebb CPU igény.
  - Két diszk egyidejű kiesése sem okoz adatvesztést.
  - Relatív drága
  - N diszk RAID 6-os tömbjének kapacitása, N-2 diszk kapacitással azonos.
  - Elvileg általánosítható a módszer (3 diszk kiesése)

Ma leggyakrabban a RAID 1,5 verziókat használják. A RAID 6 vezérlők az utóbbi 1-2 évben jelentek meg. Bár olcsó diszkekről szól a RAID, de valójában ezek nem mindig olcsók. RAID 6-nl már 2 lemez kiesik, így ez még inkább drága.
Hot-Swap(forró csere) RAID vezérlő: működés közben a meghibásodott lemezt egyszerűen kicseréljük.

8.3 Fájlrendszerek szolgáltatásai

Fájl: adatok egy logikai csoportja, névvel egyéb paraméterekkel ellátva. A fájl az információtárolás egysége, névvel hivatkozunk rá. Jellemzően egy lemezen helyezkedik el, de általánosan az adathalmaz, adatfolyam akár képernyőhöz, billentyűzethez is köthető.
A lemezen általában 3 féle fájl, állomány található:

- Rendes felhasználói állomány.
- Ideiglenes állomány
- Adminisztratív állomány. Ez a működéshez szükséges, általában rejtett.

Könyvtár: fájlok (könyvtárak) logikai csoportosítása.
Fájlrendszer: módszer, a fizikai lemezünkön, kötetünkön a fájlok és könyvtárak elhelyezés rendszerének kialakítására.
Fájlok elhelyezése: A partíció elején, az ún. Szuperblokk (pl. FAT esetén a 0. blokk) leírja a rendszer jellemzőit. Általában következik a helynyilvántartás (FAT, láncolt listás nyilvántartás). Ezután a könyvtárszerkezet (inode), a könyvtár bejegyzésekkel, fájl adatokkal. (FAT16-nál a könyvtár előbb van, majd utána a fájl adatok.)
Elhelyezési stratégiák:

- Folytonos tárkiosztás: First Fit, Best Fit, Worst Fit (olyan memória szakaszba tesszük, hogy a lehető legnagyobb rész maradjon szabadon). Veszteséges lemezkihasználás.
- Láncolt elhelyezkedés: Nincs veszteség (csak a blokkméretből adódóan). Fájl adatok (blokkokra bontva) láncolt lista tábla. Az n. blokk olvasása lassú lesz. Szabad-foglalt szektorok: File Allocation Table,FAT. Ez nagy lehet, a FAT-nak a memóriában kell lenni fájl műveletnél.
- Indextáblás elhelyezés: Katalógus tartalmazza a fájlhoz tartozó kis tábla (inode) címét. Egy inode címből elérhető a fájl.

Fájlrendszer típusok:

- Merevlemezen alkalmazott fájlrendszer: FAT, NTFS, EXT2FS, XFS, stb
- Szalagos rendszereken (elsősorban backup) alkalmazott fájlrendszer: Tartalomjegyzék, majd a tartalom szekvenciálisan
- CD, DVD, Magneto-opto Disc fájlrendszer: CDFS, UDF (Universal Disc Format), kompatibilitás
- RAM lemezek (ma már kevésbé használtak)
- FLASH memória meghajtó (FAT32)
- Hálózati meghajtó: NFS
- Egyéb pszeudó fájlrendszerek: Zip, tar.gz, ISO

Naplózott fájlrendszerek: A fájlrendszer sérülés, áramszünet stb. esetén inkonzisztens állapotba kerülhet. Gyakran nevezik: LFS-nek (Log-structured File System) vagy JFS-nek(Journaled). Adatbáziskezelők mintájára: művelet + log naplózódik. Tranzakciós alapra épül. Leállás, hiba esetén a log alapján helyre lehet állítani. Célszerűen a log másik lemez (másik partíció). Nagyobb erőforrás igényű, de nagyobb a megbízhatóság.
A mai operációs rendszerek „rengeteg" típust támogatnak, pl: Linux 2.6 kernel több mint 50-et. A fájlrendszert csatolni többféleképpen is lehet:

- Mount, eredményeképpen a fájlrendszer állományok elérhetők lesznek.
- Automatikus csatolás (pl. USB drive)
- Kézi csatolás (Linux, mount parancs)

Külön névtérbeli elérhetőség a Windowsnál az A, B, C stb lemezek. Egységes névtér a UNIX-nál van.

8.4 Fájlrendszerek szolgáltatásainak jellemző megvalósításai

- FAT
  - File Allocation Table. Talán a legrégebbi, ma is élő fájlrendszer.
  - A FAT tábla a lemez foglaltsági térképe, annyi eleme van, ahány blokk a lemezen. Pl: Fat12, FDD, Cluster méret 12 bites. Ha értéke 0, szabad, ha nem, foglalt. Biztonság kedvéért 2 tábla van.
  - Láncolt elhelyezés. A katalógusban a file adatok (név stb) mellett csak az első fájl blokk sorszáma van megadva. A FAT blokk azonosító mutatja a következő blokk címét. Ha nincs tovább, FFF az érték.
  - Rögzített bejegyzés méret, 32 bájt (max. 8.3 név)
  - System, Hidden, Archive, Read only, könyvtár attribútumok
  - A fájl utolsó módosítás ideje is tárolva van.
  - *FAT16*, 16 bites cluster leíró, 4 bájt (2x2) írja le a fájl kezdőblokkját. Max. 4 GB partíciós méret (64kb blokk méretnél), jellemzően 2 GB. Fájlméret maximum is a 4 (2) GB. Külön könyvtári terület (FDD-nez a 0. sáv). FDD-n 512 könyvtári bejegyzés. HDD-n 32736 könyvtári bejegyzés (16 bit előjelesen)

- *FAT32* (1996-tól elérhető): 28 bites clusterleíró, 2 TB partíciós méret (alap szektor mérettel), 32 MB-ig 1 blokk = 1 szektor(512bájt). 64 MB: 1 blokk=1KB (2 szektor), 128MB: 1 blokk=2KB. 1 blokk max. 64 KB lehet.
- Támogatták már a hosszú fájl neveket is. Többszörös 8.3 részre fenntartott bejegyzésekkel.
- Töredezettségmentesítés szükséges.

- NTFS
  - New Technology File System
  - FAT-NTFS hatékonysági határ: kb. 400 MB.
  - 255 karakteres fájl név, 8+3 másodlagos név
  - Kifinomult biztonsági beállítások
  - Ahogy a FAT esetén, itt is szükséges a töredezettségmentesítés.
  - Titkosított fájlrendszer támogatása, naplózás
  - POSIX támogatás. Hardlink (fsutil parancs), időbélyegek, kis-nagybetűk különböznek
  - Tömörített fájl, mappa, felhasználói kvóta kezelés
  - Az NTFS csak klasztereket tart nyilván, szektort (512bájt) nem

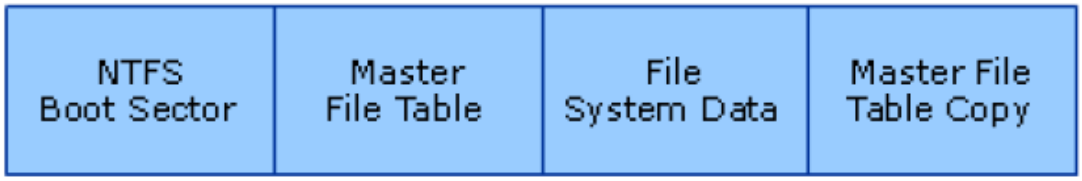


Figure 9: NTFS partíció. A Master File Table és a File System Data egy-egy táblázat

  - MFT: NTFS partíció az MFT (Master File Table) táblázattal kezdődik. 16 attribútum ad egy fájl bejegyzést. Minden attribútum max. 1kb. Ha ez nem elég, akkor egy attribútum mutat a folytatásra. Az adat is egyfajta attribútum, így egy bejegyzés több adatsort tartalmazhat. (PL: Betekintő kép) Elvi fájlméret 2^64 bájt lehet. Ha a fájl $<$ 1kb, belefér az attribútumba, közvetlen fájl. Nincs fájlméret maximum.

Figure 10: Az NTFS partíció felépítése

- ext, az ext2 és az ext3
  - Az „ext” kifejezés a fájlrendszerek neveiben az extended (magyarul kiterjesztett) kifejezést takarja. Az extended fájlrendszer volt az első kifejezetten a UNIX-szerű GNU/LINUX operációs rendszerekhez készített fájlrendszer, amely örökölte az UFS (UNIX File System) fájlrendszer metaadat-szerkezetét, és arra készült, hogy a Minix operációs rendszer fájlrendszerének a hibáit kiküszöbölje. A hibák kiküszöbölése többek között a Minix operációs rendszer fájlrendszer-határainak kiterjesztése.
  - Az ext2 fájlrendszer, amely a GNU/LINUX operációs rendszereken kívül más rendszereken is megjelent, több Linux disztribúció alapértelmezett fájlrendszere volt, amíg az utódja, az „ext3” fájlrendszer (third extended filesystem – harmadik kiterjesztett fájlrendszer) el nem készült.
  - Az ext3 fájlrendszer (third extended filesystem – harmadik kiterjesztett fájlrendszer) az ext2 fájlrendszer utódja, amely már az ext2 fájlrendszerhez képest naplózást is tartalmaz. Ez a naplózás elsősorban a biztonságot növeli, és lehetővé teszi azt, hogy szabálytalan leállás bekövetkezése után ne kelljen az egész fájlrendszert újra ellenőrizni.
- ReiserFS: A ReiserFS fájlrendszer lehetővé teszi egy blokkos eszközön (block device) változó méretű fájlok tárolását és könyvtárstruktúrába rendezését. A kezdeti UNIX és UNIX-szerű operációs rendszerek (így pl. a GNU/LINUX operációs rendszer is) csak egyfajta fájlrendszert támogattak, a saját formátumukat. A modern operációs rendszerek viszont többféle fájlrendszert is támogatnak, és vannak olyan fájlrendszerek is, amelyeket több operációs rendszer is támogat. A ReiserFS fájlrendszer egyáltalán nem ilyen. A ReiserFS fájlrendszer egy olyan fájlrendszer, amely csak és kizárólag a GNU/LINUX operációs rendszer alatt használható jelenleg korlátozás nélkül.

Chapter 16

16

Záróvizsga tételsor

16. Számítógépes hálózatok és Internet eszközök

Dobreff András

Számítógépes hálózatok és Internet eszközök

Fizikai réteg, adatkapcsolati réteg, hálózati réteg, szállítói réteg – feladatok, módszerek, protokollok

1 Hálózatok modelljei

TCP/IP modell

Transmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol. Röviden TCP/IP. A TCP/IP modell 1982-ben lett az amerikai hadászati célú számítógépes hálózatok standardja. 1985-től népszerűsítették kereskedelmi használatra.

4 Réteget különböztet meg:

1. Kapcsolati réteg
2. Hálózati réteg
3. Szállítói réteg
4. Alkalmazási réteg

OSI modell

Open **S**ystem **I**nterconnection Reference Model. Röviden OSI referencia modell. Standard koncepcionális modellt definiál kommunikációs hálózatok belső funkcionalitásához.

7 Réteget különböztet meg:

1. Fizikai réteg
2. Adatkapcsolati réteg
3. Hálózati réteg
4. Szállítási réteg
5. Munkamenet réteg
6. Megjelenítési réteg
7. Alkalmazási réteg

2 Fizikai réteg

Definíció

A fizikai réteg feladata a bitek továbbítása a kommunikációs csatornán keresztül. Azaz a korrekt bit átvitel biztosítása, a kapcsolat kezelése és az átvitelhez szükséges idő és egyéb részletek tisztázása. Tehát a tervezési szempontok az interfész mechanikai, elektromos és eljárási kérdéseire, illetve az átviteli közegre vonatkoznak.

Adatátvitel

Vezetékes

Adatátvitel vezeték esetén valamilyen fizikai jellemző változtatásával lehetséges (pl.: feszültség, áramerősség). Ezt egy $g(t)$ periodikus függvénnyel jellemezhetjük.

$$g(t) = \frac{1}{2}c + \sum_{n=1}^{\infty} a_n sin(2\pi nft) + \sum_{n=1}^{\infty} b_n cos(2\pi nft)$$

ahol $f = \frac{1}{T}$ az alapfrekvencia, a_n és b_n pedig az n-edik harmonikus szinuszos illetve koszinuszos amplitúdók

Vezetékes nélküli

Vezeték nélküli adatátvitelre sok helyen használnak elektormágneses hullámokat. A hullámoknak van frekvenciája és hullámhossza.

- Frekvencia: A hullám másodpercenkénti rezgésszáma. Jele: f, mértékegysége: Hz (Hertz)
- Hullámhossz: két egymást követő hullámcsúcs (v. hullámvölgy) közötti távolság. Jele: λ

$$\lambda f = c$$

ahol c a fénysebesség, azaz az elektromágneses hullámok terjedési sebessége vákuumban.

Tartomány neve	Hullámhossz (*centiméter*)	Frekvencia (*Hertz*)
Rádió	10	$< 3 * 10^9$
Mikrohullám	10 - 0.01	$3 * 10^9$ - $3 * 10^{12}$
Infravörös	0.01 - 7×10^{-5}	3×10^{12} - 4.3×10^{14}
Látható	7×10^{-5} - 4×10^{-5}	$4.3 * 10^{14}$ - $7.5 * 10^{14}$
Ultraibolya	4×10^{-5} - 10^{-7}	$7.5 * 10^{14}$ - $3 * 10^{17}$
Röntgen sugarak	10^{-7} - 10^{-9}	$3 * 10^{17}$ - $3 * 10^{19}$
Gamma sugarak	$< 10^{-9}$	$> 3 * 10^{19}$

ábra 1: Elektromágneses spektrum

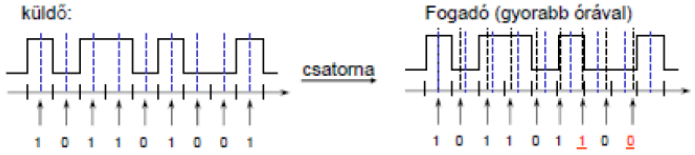
Szimbólumok

Bitek helyett szimbólumokat küldünk át. (Pl. 4 szimbólum: A - 00, B - 01, C - 10, D - 11)
Baud: szimbólum/másodperc
Adatráta: bit/másodperc

Szinkronizáció

Kérdés: Mikor kell szignálokat mérni, illetve mikor kezdődik egy szimbólum? Ehhez szinkronizáció kell a felek között.

- Explicit órajel
  Párhuzamos átviteli csatornák használata, szinkronizált adatok, rövid átvitel esetén alkalmas.
- Kritikus időpontok
  Szinkronizáljunk például egy szimbólum vagy blokk kezdetén, a kritikus időpontokon kívül szabadon futnak az órák, feltesszük, hogy az órák rövid ideig szinkronban futnak.
- Szimbólum kódok
  Önütemező jel–külön órajel szinkronizáció nélkül dekódolható jel, a szignál tartalmazza a szinkronizáláshoz szükséges információt.



ábra 2: Szinkronizáció szükségessége

Átviteli közegek

Vezetékes

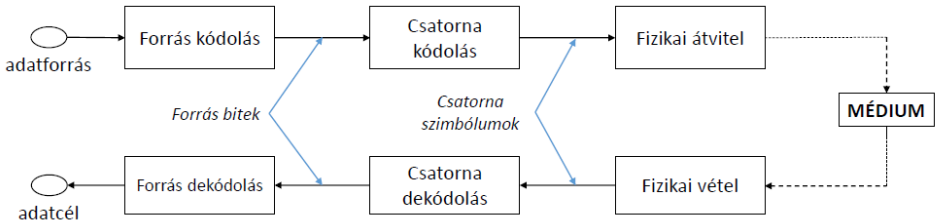
- mágneses adathordozók
  sávszélesség jó, késleltetés nagy
- sodort érpár
  Főként távbeszélőrendszerekben használatos; dupla rézhuzal; analóg és digitális jelátvitel
- Koaxális kábel
  Nagyobb sebesség és távolság érhető el, mint a sodorttal; analóg és digitális jelátvitel
- Fényvezető szálak
  Fényforrás, átviteli közeg és detektor; fényimpulzus 1-es bit, nincs fényimpulzus 0-s bit

Vezetékes nélküli

- Rádiófrekvenciás átvitel
  egyszerűen előállíthatóak; nagy távolság; kültéri és beltéri alkalmazhatóság; frekvenciafüggő terjedési jellemzők
- Mikrohullámú átvitel
  egyenes vonal mentén terjed; elhalkulás problémája; nem drága
- Infravörös és milliméteres hullámú átvitel
  kistávolságú átvitel esetén; szilárd tárgyakon nem hatol át
- Látható fényhullámú átvitel
  lézerforrás + fényérzékelő; nagy sávszélesség, olcsó, nem engedélyköteles; időjárás erősen befolyásolhatja

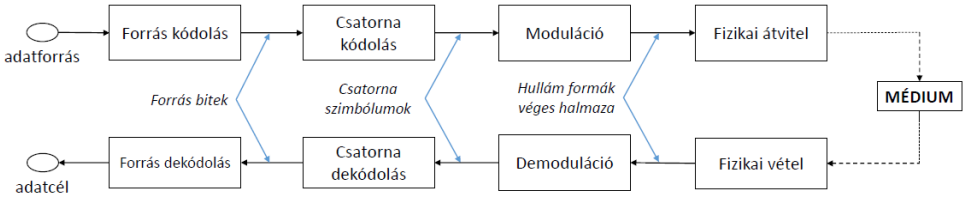
Jelátvitel

- Alapsáv
  A digitális jel direkt árammá vagy feszültséggé alakul. A jel minden frekvencián átvitelre kerül. Átviteli korlátok



ábra 3: Digitális alapsávú átvitel struktúrája

- Szélessáv
  Széles frekvencia tartományban történik az átvitel. A jel modulálására az alábbi lehetőségeket használhatjuk.



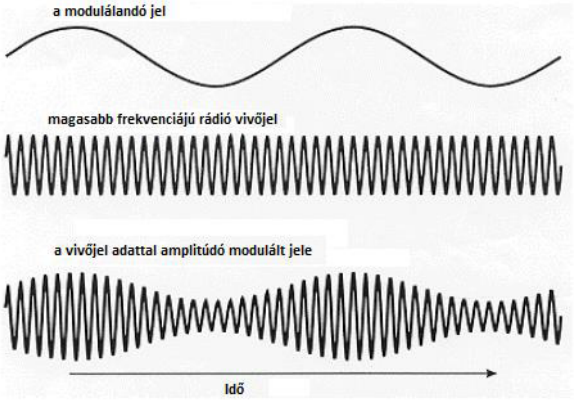
ábra 4: Digitális szélessávú átvitel struktúrája

Modulációk:
Egy szinuszos rezgés ábrázolása T periódus idejű függvényre $g(t) = Asin(2\pi ft + \varphi)$, ahol A az amplitúdó, $f = \frac{1}{T}$ a frekvencia és φ a fáziseltolás

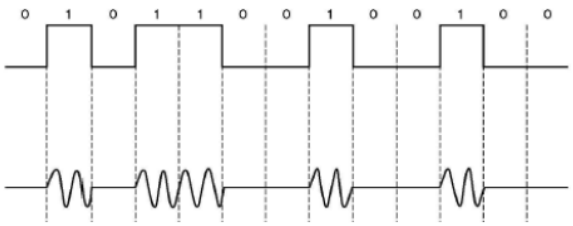
- Amplitúdó moduláció
  Az $s(t)$ szignált a szinusz görbe amplitúdójaként kódoljuk, azaz:

$$f_A(t) = s(t) \cdot sin(2\pi ft + \varphi)$$

Analóg szignál: amplitúdó moduláció
Digitális szignál: amplitúdó keying(szignál erőssége egy diszkrét halmaz értékeinek megfelelően változik)



ábra 5: Amplitúdó moduláció

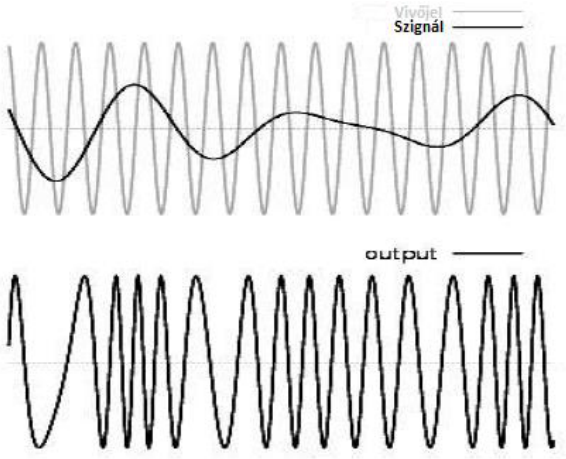


ábra 6: Amplitúdó keying

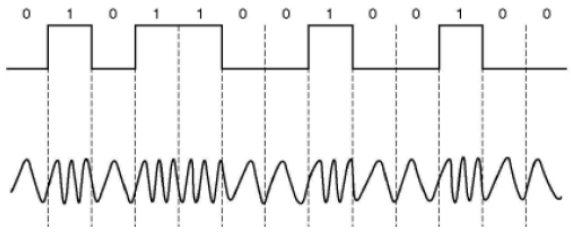
– Frekvencia moduláció
Az $s(t)$ szignált a szinusz görbe frekvenciájában kódoljuk, azaz:

$$f_F(t) = a \cdot sin(2\pi s(t)t + \varphi)$$

Analóg szignál: frekvencia moduláció
Digitális szignál: frekvencia-eltolás keying(például egy diszkrét halmaz szimbólumaihoz különböző frekvenciák hozzárendelésével)



ábra 7: Frekvencia moduláció

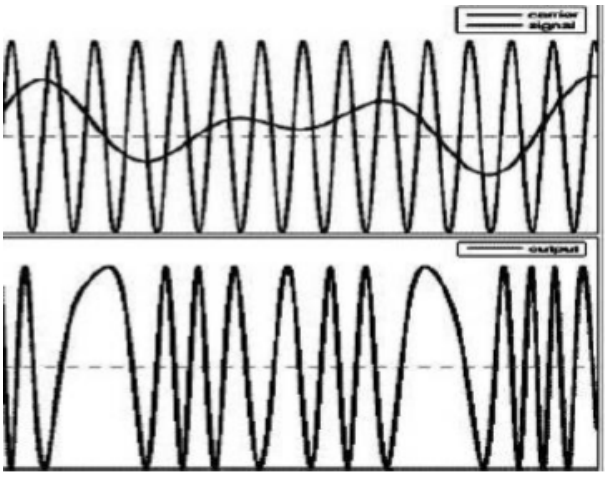


ábra 8: Frekvencia keying

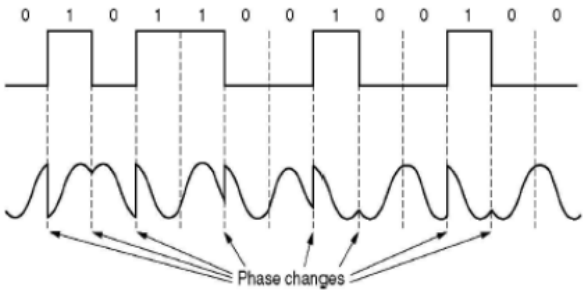
– Fázis moduláció
Az $s(t)$ szignált a szinusz görbe fázisában kódoljuk, azaz:

$$f_P(t) = a \cdot sin(2\pi ft + s(t))$$

Analóg szignál: fázis moduláció (nem igazán használják)
Digitális szignál: fázis-eltolás keying (például egy diszkrét halmaz szimbólumaihoz különböző fázisok hozzárendelésével)



ábra 9: Fázis moduláció



ábra 10: Fázis-eltolás keying

Digitális és analóg jelek összehasonlítása:
Digitális átvitel: Diszkrét szignálok véges halmazát használja (például feszültség vagy áramerősség értékek).
Analóg átvitel: Szignálok folytonos halmazát használja (például feszültség vagy áramerősség a vezetékben)
Digitális esetében lehetőség van a vételpontosság helyreállítására illetve az eredeti jel helyreállítására, míg az analógnál a fellépő hibák önmagukat erősíthetik.

3 Adatkapcsolati réteg

Definíció

Az adatkapcsolati réteg feladata jól definiált szolgálati interfész biztosítása a hálózati rétegnek, melynek három fázisa van:

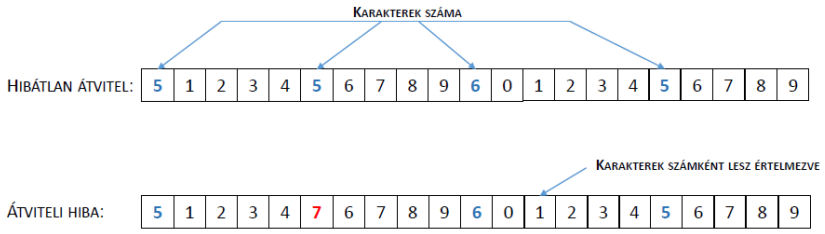
- nyugtázatlan összeköttetés alapú szolgálat
- nyugtázott összeköttetés nélküli szolgálat
- nyugtázott összeköttetés alapú szolgálat

Továbbá az átviteli hibák kezelése és az adatforgalom szabályozása (elárasztás elkerülése).

Keretképzés

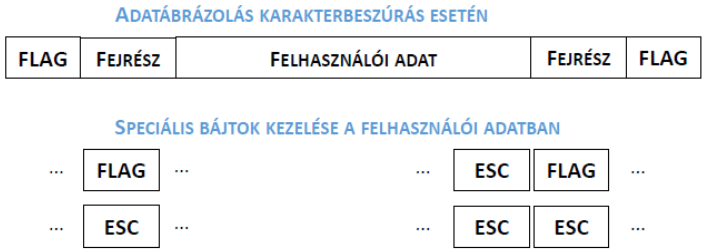
A fizikai réteg nem garantál hibamentességet, az adatkapcsolati réteg feladata a hibajelzés illetve a szükség szerint javítás. Erre megoldás: keretekre tördelése a bitfolyamnak, és ellenőrző összegek számítása. A keretezés nem egyszerű feladat, mivel megbízható időzítésre nem nagyon van lehetőség. Négy lehetséges módszer:

1. Karakterszámlálás
   A keretben lévő karakterek számát a keret fejlécében adjuk meg. Így a vevő adatkapcsolati rétege tudni fogja a keret végét. Probléma: nagyon érzékeny a hibára a módszer.



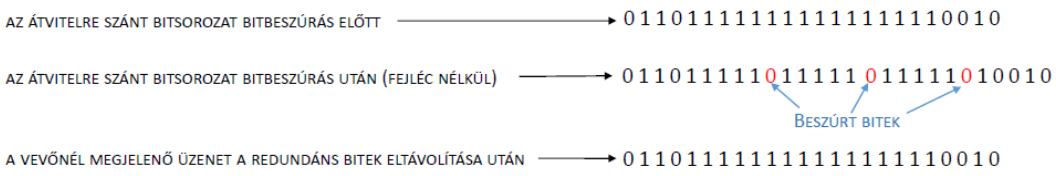
ábra 11: Karakterszámlálás

2. Kezdő és végkarakterek karakterbeszúrással
   Különleges bájtokat helyezünk el a keret elejének és végének jelzésére, aminek a neve jelző bájt(flagbyte). Az adatfolyamban szereplő speciális bájtokhoz ESC bájtot használnak.



ábra 12: Kezdő és végkarakterek karakterbeszúrással

3. Kezdő és végjelek bitbeszúrása
   Minden keret egy speciális bitmintával kezdődik (flagbájt, 01111110) és minden egymást követő 5 hosszú folytonos 1-es bit sorozat után beszúr egy 0-át.



ábra 13: Kezdő és végjelek bitbeszúrásal

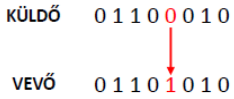
4. Fizikai rétegbeli kódolás-sértés
   Olyan hálózatokban használható, ahol a fizikai rétegbeli kódolás redundanciát tartalmaz.

Hibakezelés

Hibakezelés szempontjából a következő két esetet kell vizsgálnunk. A keretek megérkeztek-e a célállomás hálózati rétegéhez, illetve helyes sorrendben érkeztek-e meg. Ehhez valamilyen visszacsatolás szükséges a vevő és az adó között. (például nyugták). Időkorlátokat vezetünk be az egyes lépésekhez. Hiba estén a csomagot újraküldjük. Többszörös vétel lehet, amin segíthet a sorszámok használata. Az adatkapcsolati réteg feladata a hibakezelés szempontjából, hogy az időzítőket és számlálókat úgy kezelje, hogy biztosítani tudja a keretek pontosan egyszeri (nem több és nem kevesebb) megérkezését a célállomás hálózati rétegéhez.

Bithibák:

- egyszerű bithiba
  Az adategység 1 bitje nulláról egyre avagy egyről nullára változik.



ábra 14: Egyszerű bithiba

- csoportos bithiba
  Egy olyan folytonos szimbólum sorozatot, amelynek az első és utolsó szimbóluma hibás, és nem létezik ezen két szimbólummal határolt részsorozatban olyan m hosszú részsorozat, amelyet helyesen fogadtunk, m hosszú csoportos bithibának nevezünk.

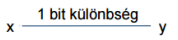
Hiba jelzés és javítás:
Kétféle hibakezelési stratégia létezik. Ezek a hibajelző (redundáns információ mellékelése) és hibajavító kódok (adatok közé iktatott redundancia). [Megbízható csatornákon a hibajelzés olcsóbb. (csomagot inkább újraküldjük). A kevésbé megbízható csatornákon a hibajavításos módszer célszerűbb]

- Hamming-távolság, Hamming-korlát
  Küldendő keret m bitet tartalmaz. Redundáns bitek száma r. Tehát az elküldött keret: $n = m + r$ bit.

  Hamming-távolság:
  Az olyan bitpozíciók számát, amelyeken a két kódszóban különböző bitek állnak, a két kódszó Hamming távolságának nevezzük. Jelölés: $d(x, y)$
  Legyen S az egyenlő hosszú bitszavak halmaza. S Hamming-távolsága:

$$d(S) := \min_{x,y,\in S \ \wedge \ x \neq y} d(x, y)$$

  d(S) = 1 esetén:
  Nincs hibafelismerés, ugyanis megengedett kódszóból 1 bit megváltoztatásával megengedett kódszó áll elő.

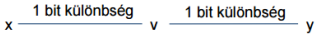


ábra 15: Kód Hamming-távolsága = 1

d(S) = 2 esetén:
Ha az x kódszóhoz létezik olyan v nem megengedett kódszó, amelyre $d(u, x) = 1$, akkor hiba történt. Ha x és y megengedett kódszavak (távolságuk minimális = 2), akkor a következő összefüggésnek teljesülnie kell:

$$2 = d(x, y) \leq d(x, v) + d(v, y)$$

Azaz egy bithiba felismerhető, de nem javítható.

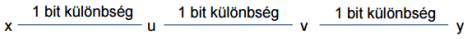


ábra 16: Kód Hamming-távolsága = 2

d(S) = 3 esetén:
Ekkor minden u, melyre $d(x, u) = 1$ és $d(u, y) > 1$ nem megengedett. Ekkor három lehetőség áll fent:

- x került átvitelre és 1 bit hibával érkezett
- y került átvitelre és 2 bit hibával érkezett
- valami más került átvitelre és legalább 2 bit hibával érkezett

De valószínűbb, hogy x került átvitelre, tehát ez egy 1 bit hiba javító, 2 bit hiba felismerő kód.



ábra 17: Kód Hamming-távolsága = 3

Hamming-korlát:
$C \subseteq \{0, 1\}^n$ és $d(C) = k$. Ekkor a kódszavak $\frac{k-1}{2}$ sugarú környezeteiben található bitszavak egymással diszjunkt halmazainak uniója legfeljebb az n-hosszú bitszavak halmazát adhatja ki. Vagyis formálisan:

$$|C| \sum_{i=0}^{\lfloor \frac{k-1}{2} \rfloor} \binom{n}{i} \leq 2^n$$

- Hibafelismerés:
  d bit hiba felismeréséhez a keretek halmazában legalább d+1 Hamming távolság szükséges.
- Hibajavítás:
  d bit hiba javításához a megengedett keretek halmazában legalább $2d + 1$ Hamming távolság szükséges.
- Kód rátája:
  $R_S = \frac{log_2|S|}{n}$ a kód rátája ($S \subseteq \{0, 1\}^n$) - hatékonyságot karakterizálja
- Kód távolsága:
  $\delta_S = \frac{d(S)}{n}$ a kód távolsága ($S \subseteq \{0, 1\}^n$) - hibakezelést karakterizálja

A jó kódnak a rátája és a távolsága is nagy.

- Paritásbit
  A paritásbit olyan bit, melyet a kódszóban lévő egyesek száma alapján választunk.
  - Odd Parity - ha az egyesek száma páratlan, akkor 0 befűzése; egyébként 1-es befűzése
  - Even Parity - ha az egyesek száma páros, akkor 0 befűzése; egyébként 1-es befűzése

  Egy paritást használó módszer az ún. Hamming módszer:
  A kódszó bitjeit számozzuk meg (1-gyel kezdődően). A 2 hatványú pozíciókon az ellenőrző bitek kapnak helyet, a maradék helyekre az üzenet bitjei kerülnek. Mindegyik ellenőrző bit a bitek egy csoportjának (beleértve önmagát is) a paritását állítja be párosra (vagy páratlanra). A csoportok a következőképp alakulnak:
  - 1. bit: Minden első egyhosszú bitsorozat az első bittől kezdve (tehát: 1,3,5,7,...)

– 2. bit: Minden első kéthosszú bitsorozat a második bittől kezdve (tehát: 2-3,6-7,10-11)
– 4. bit: Minden első négyhosszú bitsorozat a negyedik bittől kezdve (tehát: 4-7,12-15)
– stb.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		...
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

ábra 18: Paritásbitek csoportjai

Példa:
Legyen az átküldendő üzenet: 1000101
Ekkor a kódszó a következőképp alakul:
♡♡1♡000♡101
A 8. bit a 8-11 bitsorozat paritását állítja be párosra:
♡♡1♡0000101
A 4. bit a 4-7 bitsorozat paritását állítja be párosra:
♡♡100000101
A 2. bit a 2-3, 6-7, 10-11 bitsorozat paritását állítja be párosra:
♡0100000101
Az 1. bit az 1,3,5,7,9,11 bitsorozat paritását állítja be párosra:
10100000101

Tehát a elküldendő bitsorozat: 10100000101

- CRC - Polinom-kód, azaz ciklikus redundancia
  A bitsorozatokat egy $\mathbb{Z}_2$ feletti polinom ($M(x)$) együtthatóinak tekintjük. Definiálunk egy $G(x)$ r fokú generátorpolinomot, melyet a vevő és küldő egyaránt ismer.
  1. Fűzzünk r darab 0 bitet a keret alacsony helyi értékű végéhez. Azaz vegyük az $x^r M(x)$ polinomot (ez már $m + r$ fokú)
  2. Osszuk el $x^r M(x)$-et $G(x)$-szel (mod 2).
  3. A maradékot (mely mindig r vagy kevesebb bitet tartalmaz) vonjuk ki $x^r M(x)$-ből (mod 2). Így az eredeti keret végére egy r hosszú ellenőrző összeg kerül. Legyen ez a polinom $T(x)$.
  4. A vevő egy $T(x) + E(x)$-nek megfelelő polinomot kap (ahol $E(x)$ a hiba polinom). Ezt elosztva a generátorpolinommal egy $R(x)$ polinomot kapunk. Ha ez a polinom nem nulla, akkor hiba történt.

  A $G(x)$ többszöröseinek megfelelő bithibákat nem ismerjük fel.

Protokollok

Elemi adatkapcsolati protokollok

- Korlátozás nélküli szimplex protokoll
  Környezet:
  – Adó, vevő: mindig kész.
  – Nincs feldolgozási idő
  – Végtelen puffer
  – Nincs keret rontás, vesztés
  Protokoll:
  – Nincs sorszám/nyugta
  – Küldő végtelen ciklusban küldi kifele a kereteket folyamatosan

  - A vevő kezdetben várakozik az első keret megérkezésére, keret érkezésekor a hardver puffer tartalmát változóba teszi és az adatrészt továbbküldi a hálózati rétegnek.
- Szimplex megáll és vár protokoll
  Környezet:
  - Adó, vevő hálózati rétegei: mindig kész.
  - A vevőnek δt időre van szüksége a bejövő keret feldolgozására.
  - Nincs pufferelés és sorban állás sem
  - Nincs keret rontás, vesztés

  Protokoll:
  - Nincs sorszám/nyugta
  - Küldő egyesével küld, következőt csak a nyugtát követően.
  - A vevő kezdetben várakozik az első keret megérkezésére, keret érkezésekor a hardver puffer tartalmát változóba teszi és az adatrészt továbbküldi a hálózati rétegnek, végül nyugtázza a keretet.
- Szimplex protokoll zajos csatornához
  Környezet:
  - Adó, vevő hálózati rétegei: mindig kész.
  - A vevőnek δt időre van szüksége a bejövő keret feldolgozására.
  - Nincs pufferelés és sorban állás sem
  - Keret sérülhet, elveszhet

  Protokoll:
  - Nincs sorszám/nyugta
  - Küldő egyesével küld, addig nem küld újat, míg határidőn belül nyugtát nem kap. Határidő után újraküldi a keretet.
  - A vevő kezdetben várakozik az első keret megérkezésére, keret érkezésekor a hardver puffer tartalmát változóba teszi, leellenőrzi a kontroll összeget:
    * Nincs hiba: az adatrészt továbbküldi a hálózati rétegnek, végül nyugtázza a keretet.
    * Van hiba: eldobja a keretet és nem nyugtáz

Csúszóablakos protokoll

Egy adott időpontban egyszerre több keret is átviteli állapotban lehet. A fogadó n keretnek megfelelő méretű puffert allokál. A küldőnek legfeljebb n, azaz ablak méretnyi, nyugtázatlan keretet küldése engedélyezett. A keret sorozatbeli pozíciója adja a keret címkéjét. (sorozatszám). A fogadónak a hibás, illetve a nem megengedett sorozatszámmal érkező kereteket el kell dobnia. A küldő nyilvántartja a küldhető sorozatszámok halmazát (adási ablak). A fogadó nyilvántartja a fogadható sorozatszámok halmazát (vételi ablak). Az adási ablak minden küldéssel szűkül, illetve nő egy nyugta érkezésével.

Mi van ha egy hosszú folyam közepén történik egy keret hiba?

1. "visszalépés N-nel" stratégia
   Az összes hibás keret utáni keretet eldobja és nyugtát sem küld róluk. Mikor az adónak lejár az időzítője, akkor újraküldi az összes nyugtázatlan keretet, kezdve a sérült vagy elveszett kerettel. Hátrány: Nagy sávszélességet pazarolhat el, ha nagy a hibaarány.
2. "szelektív ismétlés" stratégia
   A hibás kereteket eldobja, de a jó kereteket a hibás után puffereli. Mikor az adónak lejár az időzítője, akkor a legrégebbi nyugtázatlan keretet küldi el újra. Hátrány: Nagy memória igény nagy vételi ablak esetén.

Példák adatkapcsolati protokollokra

- HDLC - High-level Data Link Control
  A HDLC protokoll 3 bites csúszó-ablak protokollt alkalmaz a sorszámozáshoz. Három típusú keretet használ:
  - információs
  - felügyelő
    * nyugtakeret (RECIEVE READY)
    * negatív nyugta keret (REJECT)

* vételre nem kész (RECIEVE NOT READY) - nyugtáz minden keretet a következőig
* szelektív elutasítás (SELECTIVE REJECT) - egy gy adott keret újraküldésére szólít fel

– Számozatlan

Általános keretfelépítése:

– FLAG bájt a keret határok jelzésére
– *cím* mező - több vonallal rendelkező terminálok esetén van jelentősége
– *vezérlés* mező - sorszámozás, nyugtázás és egyéb feladatok ellátására
– *adat* mező - tetszőleges hosszú adat lehet
– *ellenőrző összeg* mező - CRC kontrollösszeg (CRC-CCITT generátor polinom felhasználásával)

8 bit	8 bit	8 bit	< 0 bit	16 bit	8 bit
01111110	CÍM	VEZÉRLÉS	ADAT	ELLENŐRZŐ ÖSSZEG	01111110

ábra 19: HDLC keret felépítése

- PPP - Point-to Point Protocol
  A PPP protokoll három dolgot biztosít:
  – Keretezési módszert (egyértelmű kerethatárok)
  – Kapcsolatvezérlő protokollt (a vonalak felélesztésére, tesztelésére, az opció egyeztetésére és a vonalak elengedésére.)
  – Olyan módot a hálózati réteg-opciók megbeszélésére, amely független az alkalmazott hálózati réteg-protokolltól.

  Bájt alapú keretszerkezet használ (azaz a legkisebb adategység a bájt). Vezérlő mező alapértéke a számozatlan keretet jelzi. Protokoll mezőben protokoll kód lehet az LCP, NCP, IP, IPX, AppleTalkvagy más protokollhoz.

1	1	1	1 vagy 2	változó	2 vagy 4	1
Jelző 01111110	Cím 1111111	Vezérlő 00000011	Protokoll	Adatmező	Ellenőrző összeg	Jelző 01111110

ábra 20: PPP keret felépítése

MAC - Media Access Control

Eddigi tárgyalásaink során pont-pont összeköttetést feltételeztünk. Most az adatszóró csatornát használó hálózatok tárgykörével foglalkozunk majd. A csatorna kiosztás történhet statikus vagy dinamikus módon.

Statikus esetben vagy Frekvenciaosztásos nyalábolást vagy időosztásos nyalábolást használnak. Frekvenciaosztásos esetben a sávszélességet osztják N részre, és mindegyik felhasználó egy sávot kap. Időosztásos esetben az időegységet osztják N részre és ezeket adják a felhasználóknak. Mind a két módszer löketszerű forgalom esetén nem tud hatékony lenni.

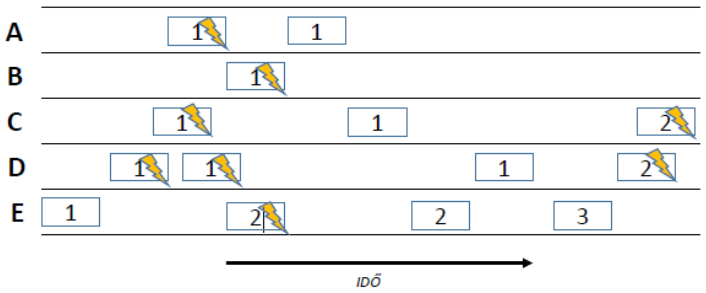
A továbbiakban a dinamikus csatorna kiosztási módszereket vizsgáljuk.

- Verseny protokollok
  N független állomás van, amelyeken egy program vagy egy felhasználó továbbítandó kereteket generál. Ha egy állomás generált egy keretet, akkor blokkolt állapotban marad mindaddig, amíg a keretet sikeresen nem továbbította. Egyetlen csatorna van, melyen mindenféle kommunikáció zajlik. Minden állomás tud adatot küldeni és fogadni ezen a csatornán. Ha két keret egy időben kerül átvitelre, akkor átlapolódnak, és az eredményül kapott jel értelmezhetetlenné válik. Ezt nevezzük ütközésnek. Ez minden állomás számára felismerhető. Az ütközésben érintett kereteket később újra kell küldeni. (Ezen a hibán kívül egyéb hiba nem történhet.)
  Kétféle időmodellt különböztetünk meg:

1. Folytonos – Mindegyik állomás tetszőleges időpontban megkezdheti a küldésre kész keretének sugárzását.
2. Diszkrét – Az időt diszkrét résekre osztjuk. Keret továbbítás csak időrés elején lehetséges. Az időrés lehet üres, sikeres vagy ütközéses

Az egyes állomások vagy rendelkeznek vivőjel érzékeléssel vagy nem. Ha nem, akkor az állomások nem tudják megvizsgálni a közös csatorna állapotát, ezért egyszerűen elkezdenek küldeni, ha van rá lehetőségük. Ha igen, akkor az állomások meg tudják vizsgálni a közös csatorna állapotát a küldés előtt. A csatorna lehet: foglalt vagy szabad. Ha a foglalt a csatorna, akkor nem próbálják használni az állomások, amíg fel nem szabadul

– Egyszerű ALOHA
  A felhasználó akkor vihet át adatot, amikor csak szeretne. Ütközés esetén véletlen ideig várakozik az állomás, majd újra próbálkozik.
  Keret idő–egy szabványos, fix hosszúságú keret átviteléhez szükséges idő
  Egy keret akkor nem szenved ütközést, ha elküldésének első pillanatától kezdve egy keretideig nem próbálkozik más állomás keretküldéssel.

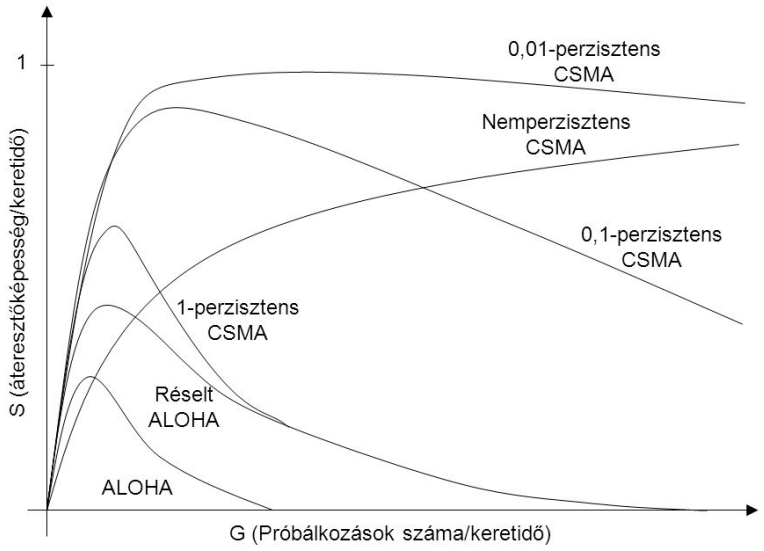


ábra 21: Egyszerű ALOHA keret ütközések

– Réselt ALOHA
  Az idő diszkrét, keretidőhöz igazodó időszeletek-re osztásával az ALOHA rendszer kapacitása megduplázható. A csatorna terhelésének kis növekedése is drasztikusan csökkentheti a médium teljesítményét.
– 1-prezisztens CSMA
  Vivőjel érzékelés van, azaz minden állomás belehallgathat a csatornába. Folytonos időmodellt használ a protokoll.
  Algoritmus:
  1. Keret leadása előtt belehallgat a csatornába:
     (a) Ha foglalt, akkor addig vár, amíg fel nem szabadul. Szabad csatorna esetén azonnal küld. (perzisztens)
     (b) Ha szabad, akkor küld.
  2. Ha ütközés történik, akkor az állomás véletlen hosszú ideig vár, majd újrakezdi a keret leadását.
– Nem-prezisztens CSMA
  Vivőjel érzékelés van, azaz minden állomás belehallgathat a csatornába. Folytonos időmodellt használ a protokoll. Mohóságot kerüli, azaz nem küld azonnal, ha foglalt.
  Algoritmus:
  1. Keret leadása előtt belehallgat a csatornába:
     (a) Ha foglalt, akkor véletlen ideig vár (nem figyeli a forgalmat), majd kezdi előről a küldési algoritmust. (nem-perzisztens)
     (b) Ha szabad, akkor küld.
  2. Ha ütközés történik, akkor az állomás véletlen hosszú ideig vár, majd újrakezdi a keret leadását.
– p-prezisztens CSMA
  Vivőjel érzékelés van, azaz minden állomás belehallgathat a csatornába. Diszkrét időmodellt használ a protokoll.
  Algoritmus:

1. Adás kész állapotban az állomás belehallgat a csatornába:
   (a) Ha foglalt, akkor vár a következő időrésig, majd megismétli az algoritmust.
   (b) Ha szabad, akkor p valószínűséggel küld, illetve $1 - p$ valószínűséggel visszalép a szándékától a következő időrésig. Várakozás esetén a következő időrésben megismétli az algoritmust. Ez addig folytatódik, amíg el nem küldi a keretet, vagy amíg egy másik állomás el nem kezd küldeni, mert ilyenkor úgy viselkedik, mintha ütközés történt volna.
2. Ha ütközés történik, akkor az állomás véletlen hosszú ideig vár, majd újrakezdi a keret leadását.

– CSMA/CD
Ütközés érzékelés esetén meg lehessen szakítani az adást. Minden állomás küldés közben megfigyeli a csatornát, ha ütközést tapasztalna, akkor megszakítja az adást, és véletlen ideig várakozik, majd újra elkezdi leadni a keretét

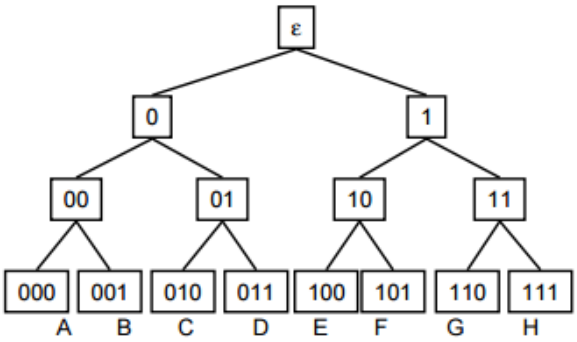


ábra 22: ALOHA és CSMA protokollok összehasonlítása

- Verseny mentes protokollok
Motiváció: Az ütközések hátrányosan hatnak a rendszer teljesítményére, és a CSMA/CD nem mindenhol alkalmazható.
N állomás van. Az állomások 0-ától N-ig egyértelműen sorszámozva vannak. Réselt időmodellt feltételezünk.
  – Egy helyfoglalásos protokoll
  Ha az i-edikállomás küldeni szeretne, akkor a i-edikversengési időrésben egy 1-es bit elküldésével jelezheti. Így a versengési időszak végére minden állomás ismeri a küldőket. A küldés a sorszámok szerinti sorrendben történik meg.
  – Bináris visszaszámlálás protokoll
  Minden állomás azonos hosszú bináris azonosítóval rendelkezik. A forgalmazni kívánó állomás elkezdi a bináris címét bitenként elküldeni a legnagyobb helyi értékű bittel kezdve. Az azonos pozíciójú bitek logikai VAGY kapcsolatba lépnek ütközés esetén. Ha az állomás nullát küld, de egyet hall vissza, akkor feladja a küldési szándékát, mert van nála nagyobb azonosítóval rendelkező küldő.
- Korlátozott verseny protokollok
Olyan protokoll, amely kis terhelés esetén versenyhelyzetes technikát használ a kis késleltetés érdekében, illetve nagy terhelés mellett ütközésmentes technikát alkalmaz a csatorna jó kihasználása érdekében.

– Adaptív fabejárás
Minden állomást egy egyértelmű, bináris ID reprezentál. Az ID-k egy (bináris) fa leveleinek felelnek meg. Az időrések a fa egyes csomópontjaihoz vannak rendelve Minden időrésben megvizsgáljuk az adott csomópont alatti részfát. A fa egy u csomópontjánál 3 esetet különböztethetünk meg:
* Egy állomás sem küld az u részfában.
* Pontosan egy állomás küld az u részfában.
* Több állomás küld az u részfában. Ezt nevezzük kollíziónak.

Kollízió esetén hajtsuk végre az ellenőrzést u bal, és jobb oldali gyerekére egyaránt. Ezzel a módszerrel könnyen megállapítható, hogy melyik állomás küldhet az adott időszeletben.



ábra 23: Adaptív fabejárás protokoll bináris fája

4 Hálózati réteg

Definíció

A hálózati réteg fő feladata a csomagok továbbítása a forrás és a cél között. Ez a legalacsonyabb olyan réteg, amely két végpont közötti átvitellel foglalkozik. Ismernie kell a kommunikációs alhálózat topológiáját. Ügyelni kell, hogy ne terheljen túl se bizonyos kommunikációs útvonalakat, se bizonyos routereket úgy, hogy mások tétlen maradnak.

A szállítási réteg felé nyújtott szolgálatok:

- Függetlenek az alhálózatok kialakításától
- Eltakarják a jelen lévő alhálózatok számát, típusát és topológiáját
- A szállítási réteg számára rendelkezésre bocsájtott hálózati címek egységes számozási rendszert kell alkotnak

Forgalom irányítás típusai

- Hierarchikus forgalomirányítás
Routhereket tartományokra osztjuk. A saját tartományát az összes router ismeri, de a többi belső szerkezetéről nincs tudomása. Nagy hálózatok esetén többszintű hierarchia lehet szükséges.
- Adatszóró forgalomirányítás
egy csomag mindenhová történő egyidejű küldése.
- Többküldéses forgalomirányítás
Egy csomag meghatározott csoporthoz történő egyidejű küldése.

Forgalom irányító algoritmusok

A hálózati réteg szoftverének azon része, amely azért a döntésért felelős, hogy a bejövő csomag melyik kimeneti vonalon kerüljön továbbításra. A folyamat két lépésre bontható:

1. Forgalomirányító táblázatok feltöltése és karbantartása.

2. Továbbítás

A forgalomirányító algoritmusok osztályai:

1. Adaptív algoritmusok
   (a) távolság alapú
   (b) kapcsolat alapú

   A topológia és rendszerint a forgalom is befolyásolhatja a döntést.
2. Nem-adaptív algoritmusok
   Offline meghatározás, betöltés a router-ekbe induláskor

Dijkstra algoritmus

A Dijkstra algoritmus egy statikus algoritmus, melynek célja két csomópont közötti legrövidebb út meghatározása.

Minden csomópontot felcímkézünk a kezdőpontból az addig megtalált legrövidebb út hosszával. Az algoritmus működése során a címkék változhatnak az utak megtalálásával. Két fajta címkét különböztetünk meg: ideiglenes és állandó. Kezdetben minden címke ideiglenes. A legrövidebb út megtalálásakor a címke állandó címkévé válik, és továbbá nem változik.

Elárasztás algoritmus

Elárasztás algoritmusa egy statikus algoritmus.

Minden bejövő csomagot minden kimenő vonalon továbbítunk kivéve azon, amin érkezett. Így azonban nagyon sok duplikátum keletkezne. Ezért

- Ugrásszámlálót vezetünk be, melyet minden állomás eggyel csökkent. Ha 0-ra csönnen, eldobják.
- Az állomások nyilvántartják a már kiküldött csomagokat. Így egy csomagot nem küldenek ki többször.

Elosztott Bellman-Ford algoritmus

Az Elosztott Bellman-Ford algoritmus adaptív, távolság alapú forgalomirányító algoritmus. Minden csomópont csak a közvetlen szomszédjaival kommunikálhat. Minden állomásnak van saját távolság vektora. Ezt periodikusan elküldi a direkt szomszédoknak. Minden router ismeri a közvetlen szomszédjaihoz a költséget. A kapott távolság vektorok alapján minden csomópont aktualizálja a saját vektorát.

Kapcsolatállapot alapú forgalomirányítás

A kapcsolatállapot alapú forgalomirányító algoritmusnak a motivációja, hogy a távolság alapú algoritmusok lassan konvergáltak, illetve az eltérő sávszélek figyelembevétele.
A kapcsolatállapot alapú forgalomirányító algoritmus lépései:

1. Szomszédok felkutatása, és hálózati címeik meghatározása
2. Megmérni a késleltetést vagy költséget minden szomszédhoz
3. Egy csomag összeállítása a megismert információkból
4. Csomag elküldése az összes többi router-nek
5. Kiszámítani a legrövidebb utat az összes többi router-hez.

Hálózat réteg az Interneten

A hálózati réteg szintjén az internet autonóm rendszerek összekapcsolt együttesének tekinthető. Nincs igazi szerkezete, de számos főbb gerinchálózata létezik. A gerinchálózatokhoz csatlakoznak a területi illetve regionális hálózatok. A regionális és területi hálózatokhoz csatlakoznak az egyetemeken, vállalatoknál és az internet szolgáltatóknál lévő LAN-ok.

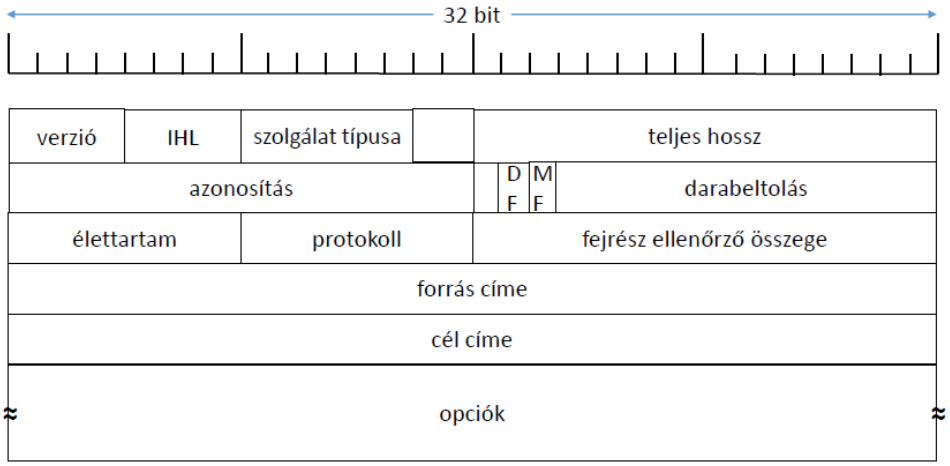
Az internet protokollja, az IP.

Az Interneten a kommunikáció az alábbi módon működik:

1. A szállítási réteg viszi az adatfolyamokat és datagramokra tördeli azokat.
2. Minden datagram átvitelre kerül az Interneten, esetleg menet közben kisebb egységekre darabolva.
3. A célgép hálózati rétege összeállítja az eredeti datagramot, majd átadja a szállítási rétegének.
4. A célgép szállítási rétege beilleszti a datagramot a vételi folyamat bemeneti adatfolyamába.

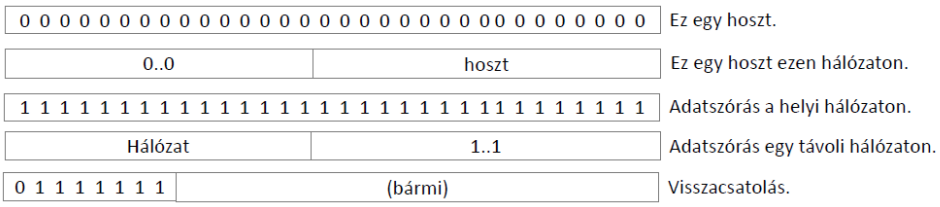
Internet Protokoll - IP

- Az IP fejrésze:
  - verzió:
    IP melyik verzióját használja
  - IHL:
    a fejléc hosszát határozza meg
  - szolgálat típusa:
    szolgálati osztályt jelöl
  - teljes hossz:
    fejléc és adatrész együttes hossza bájtokban
  - azonosítás:
    egy datagram minden darabja ugyanazt az azonosításértéket hordozza.
  - DF:
    ”ne darabold” flag a router-eknek
  - MF:
    ”több darab” flag minden darabban be kell legyen állítva, kivéve az utolsót.
  - darabeltolás:
    a darab helyét mutatja a datagramon belül.
  - élettartam:
    másodpercenként kellene csökkenteni a mező értékét, minden ugrásnál csökkentik eggyel az értékét
  - protokoll:
    szállítási réteg protokolljának azonosítóját tartalmazza
  - ellenőrző összeg:
    a router-eken belüli rossz memóriaszavak által előállított hibák kezelésére használt ellenőrző összeg a fejrészre, amelyet minden ugrásnál újra kell számolni
  - forrás cím és cél cím:
    IP cím
  - opciók:
    következő verzió bővíthetősége miatt hagyták benne.



ábra 24: IPv4 fejléce

- IP cím
  Minden hoszt és minden router az Interneten rendelkezik egy IP-címmel, amely a hálózat számát és a hoszt számát kódolja. 4 bájton ábrázolják az IP-címet. Az IP-t pontokkal elválasztott decimális rendszerben írják. (Például: 192.168.0.1) Van pár speciális cím (ábra 25).

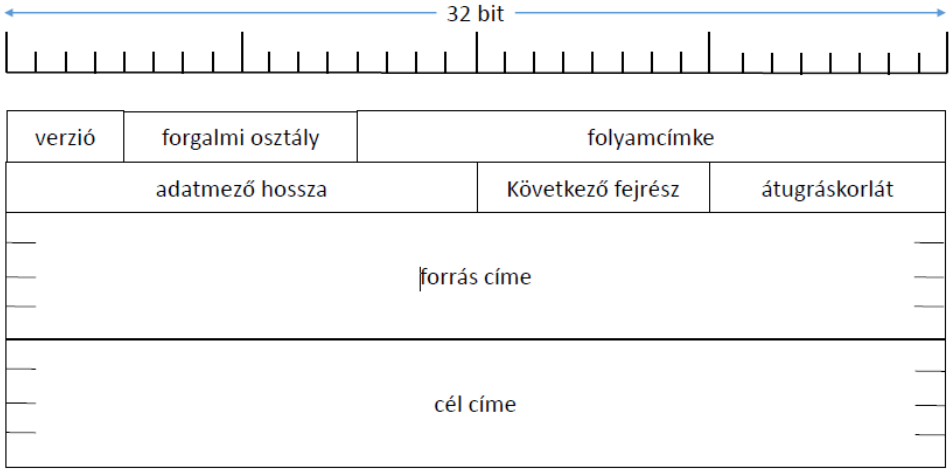


ábra 25: Speciális IP címek

Alhálózatok:
Az azonos hálózatban lévő hosztok ugyanazzal a hálózatszámmal rendelkeznek. Egy hálózat belső felhasználás szempontjából több részre osztódhat, de a külvilág számára egyetlen hálózatként jelenik meg. Azonosításnál az alhálózati maszk ismerete kell a routernek. A forgalomirányító táblázatba a router-eknél (hálózat,0) és (saját hálózat, hoszt) alakú bejegyzések. Ha nincs találat, akkor az alapértelmezett router felé továbbítják a csomagot.

IP címek fogyása:
Az IP címek gyorsan fogytak. Megoldás: osztályok nélküli környezetek közötti forgalomirányítás (CIDR). A forgalomirányítás megbonyolódik: Minden bejegyzés egy 32-bites maszkkal egészül ki. Egy bejegyzés innentől egy hármassal jellemezhető: (ip-cím, alhálózati maszk, kimeneti vonal). Új csomag esetén a cél címből kimaszkolják az alhálózati címet, és találat esetén a leghosszabb illeszkedés felé továbbítják.
Másik módszer a NAT, ami gyors javítás az IP címek elfogyásának problémájára. Az internet forgalomhoz minden cégnek egy vagy legalábbis kevés IP-címet adnak, míg vállalaton belül minden számítógéphez egyedi IP-címet használnak a belső forgalomirányításra:
10.0.0.0 – 10.255.255.255 : 16 777 216 egyedi cím
172.16.0.0 – 172.31.255.255 : 1 048 576 egyedi cím
192.168.0.0 – 192.168.255.255 : 65 536 egyedi cím

IPv6:
Az IPv4-gyel szemben 16 bájt hosszú címeket használ; 8 darab, egyenként négy-négy hexadecimális számjegyből álló csoportként írjuk le. (Például: 8000:0000:0000:0000:0123:4567:89AB:CDEF)
Az IP fejléc egyszerűsödött, amely lehetővé teszi a router-eknek a gyorsabb feldolgozást. A biztonság irányába jelentős lépés történt.



ábra 26: IPv6 fejléce

Protokollok

- Internet Control Message Protocol - ICMP
  Feladata a váratlan események jelentése. Többféle ICMP-üzenetet definiáltak:

  - Elérhetetlen cél
  - Időtúllépés
  - Paraméterprobléma
  - Forráslefojtás
  - Visszhang kérés
  - Visszhang válasz
  - etc.
- Address Resolution Protocol - ARP
  Feladata az IP cím megfeleltetése egy fizikai címnek. Egy "Kié a 192.60.34.12-es IP-cím?" csomagot küld ki az Ethernet-re adatszórással az alhálózaton. Minden egyes host ellenőrzi,hogy övé-e a kérdéses IP-cím. Ha egyezik az IP a hoszt saját IP-jével, akkor a saját Ethernet címével válaszol.
- Reverse Address Resolution Protocol - RARP
  Feladatat a fizikai cím megfeleltetése egy IP címnek. Az újonnan indított állomás adatszórással csomagot küld ki az Ethernetre: "A 48-bites Ethernet-címem 14.04.05.18.01.25. Tudja valaki az IP címemet?" Az RARP-szerver pedig válaszol a megfelelő IP címmel, mikor meglátja a kérést.
- Open Shortest Path First - OSPF
  Az OSPF az AS-eken (Autonomus System) belüli forgalomirányításért felel. A hálózat topológiáját térképezi fel, és érzékeli a változásokat. A topológiát egy súlyozott irányított gráffal reprezentálja, melyben legolcsóbb utakat keres.
- Border Gateway Protocol - BGP
  Feladata hogy a politikai szempontok szerepet játsszanak az AS-ek közötti forgalomirányítási döntésekben (Pl. Az IBM-nél kezdődő illetve végződő forgalom ne menjen át a Microsoft-on vagy Csak akkor haladjunk át Albánián, ha nincs más út a célhoz.)
  A BGP router szempontjából a világ AS-ekből és a közöttük átmenő vonalakból áll. (Két AS összekötött, ha van köztük a BGP-router-eiket összekötő vonal.) Az átmenő forgalom szempontjából 3 féle hálózat van:
  - Csonka hálózatok, amelyeknek csak egyetlen összeköttetésük van a BGP gráffal
  - Többszörösen bekötött hálózatok, amelyeket használhatna az átmenő forgalom, de ezek ezt megtagadják
  - Tranzit hálózatok, amelyek némi megkötéssel, illetve általában fizetség ellenében, készek kezelni harmadik fél csomagjait

5 Szállítói réteg

Definíció

A szállítási réteg biztosítja, hogy a felhasználók közötti adatátvitel transzparens (átlátszó) legyen. A réteg biztosítja, és ellenőrzi egy adott kapcsolat megbízhatóságát. Az alkalmazási rétegtől kapott adat elejére egy úgynevezett fejlécet csatol, mely jelzi, hogy melyik szállítási rétegbeli protokollal küldik az adatot. Néhány protokoll kapcsolat orientált. Ez azt jelenti, hogy a réteg nyomon követi az adatcsomagokat, és hiba esetén gondoskodik a csomag vagy csomagok újraküldéséről.

Kapcsolat nélküli és kapcsolatorientált

A kapcsolatorientált protokoll elfedi az alkalmazások előtt az átvitel esetleges hibáit, nem kell törődnünk az elveszett, vagy duplán megérkezett, illetve sérült csomagokkal, és azzal sem, hogy milyen sorrendben érkeztek meg. Viszont ez rontja a teljesítményét.

Kapcsolat nélküli esetben nincs szükség az adat keretekre bontására, és nincs csomagújraküldés.

Megbízhatóság

A megbízhatóság ismérvei:

- Minden csomag megérkezése nyugtázásra kerül.
- A nem nyugtázott adatcsomagokat újraküldik.
- A fejléchez és a csomaghoz ellenőrzőösszeg van rendelve.

- A csomagokat számozza, és a fogadónál sorba rendezésre kerülnek a csomagok a sorszámaik alapján.
- Duplikátumokat törli.

Torlódásfelügyelet

Minden hálózaton korlátos az átviteli sávszélessége. Ha több adatot vezetünk a hálózatba, akkor az torlódáshoz (congestion) vezet, vagy akár a hálózat összeomlásához (congestive collapse). Következmény: Az adatcsomagok nem érkeznek meg.

Lavina jelenség:
A hálózat túlterhelése csomagok elvesztését okozza, ami csomag újraküldését eredményezi. Az újraküldés tovább növeli a hálózat terhelését így még nagyobb lesz csomagvesztés. Ez még több újraküldött csomagot eredményez. ...
A torlódás felügyelet feladata a lavina jelenség elkerülése.

Követelmények a torlódásfelügyelettel szemben:

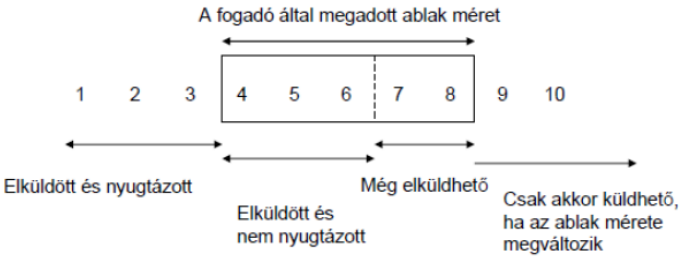
- Hatékonyság:
  Az átvitel nagy, míg a késés kicsi.
- Fairness:
  Minden folyamat megközelítőleg azonos részt kap a sávszélességből. (Priorizálás lehetősége fennáll)

A torlódásfelügyelet eszközei:

- Kapacitásnövelés
- Erőforrás foglalás és hozzáférés szabályzás
- Terhelés csökkentése és szabályzása

Stratégiák:

- Csúszóablak
  Adatráta szabályozása ablak segítségével. A fogadó határozza meg az ablak (wnd) méretét. Ha a fogadási puffere tele van, lecsökkenti 0-ra, egyébként >0-t küld. A küldő nem küld több csomagot, ha az elküldött még nem nyugtázott csomagok száma elérte az ablak méretét.



ábra 27: Csúszóablak

- Slow-start
  A küldőnek nem szabad a fogadó által küldött ablakméretet azonnal kihasználni. Meghatároz egy másik ablakot (cwnd - Congestion Window), melyet ő választ. Ezután végül amiben küld: min{wnd, cwnd}. Kezdetben cwnd = MSS (Maximum Segment Size). Minden csomagnál a megkapott nyugta után növeli: cwnd = cwnd + MSS (azaz minden RTT után duplázódik). Ez addig megy, míg a nyugta egyszer kimarad.
- TCP-Nagle
  Biztosítani kell, hogy a kis csomagok időben egymáshoz közel kerüljenek kiszállításkor, illetve hogy sok adat esetén a nagy csomagok részesüljenek előnyben.
  Ehhez: A kis csomagok nem kerülnek kiküldésre, míg nyugták hiányoznak (egy csomag kicsi, ha az adathossz <MSS). Ha a korábban küldött csomag nyugtája megérkezik, küldi a következőt.

- TCP Tahoe és Reno A TCP csúszóablakot és a Slow-start mechanizmusát is használja. Habár a kezdő ráta kicsi, az ablak mérete rohamosan nő. Amikor a cwnd eléri az ssthresh (slow start threshold) értéket átvált torlódás elkerülési állapotba. A TCP Tahoe és Reno torlódás elkerülési algoritmusok. A két algoritmus abban különbözik, hogy hogyan detektálják és kezelik a csomag vesztést.
  TCP Tahoe: A torlódás detektálására egy időzítőt állít a várt nyugta megérkezésére.
  - Kapcsolatfelvételkor: cwnd = MSS, ssthresh = 2^{16}
  - Csomagvesztésnél : Multiplicative decrease
    cwns = MSS, ssthresh = max{2MSS, $\frac{min\{cwnd,wnd\}}{2}$}
  - cwnd $\leq$ ssthresh : Slow-start
    cwnd = cwnd + MSS
  - cwnd >ssthresh : Additive Increase
    cwnd = cwnd + MSS$fracMSScwnd$

  TCP Reno: A torlódás detektálásához időzítőt és gyors újraadást is használ. [Gyors újraadás: ugyanazon csomaghoz 3 nyugta duplikátum érkezik (4 azonos nyugta), akkor újraküldi a csomagot és Slow-start fázisba lép.]
  Gyors újraadás után: ssthresh = max{$\frac{min\{wnd,cwnd\}}{2}$, 2MSS}, cwnd = sstresh + 3MSS.
  Gyors visszaállítás a gyors újraadás után minden további nyugta után növeli a rátát : cwnd = cwnd + MSS.

Hatékonyság és Fairness:
Az átvitel maximális, ha a terhelés a hálózat kapacitását majdnem eléri. Ha a terhelés tovább nő, túlcsordulnak a pufferek, csomagok vesznek el, újra kell küldeni, drasztikusan nő a válaszidő. Ezt a torlódásnak nevezzük. Ezért a maximális terhelés helyett, ajánlatos a hálózat terhelését a könyök közelében beállítani. Itt a válaszidő csak lassan emelkedik, míg az adatátvitel már a maximum közelében van

Egy jó torlódáselkerülési (angolul congestion avoidance) stratégia a hálózat terhelését a könyök közelében tartja: *hatékonyság.* Emellett fontos, hogy minden résztvevőt egyforma rátával szolgáljunk ki: *fairness*

Jelölje az i-edik résztvevő adatrátáját a t időpontban $x_i(t)$. Minden résztvevő aktualizálja az adatrátáját a $t+1$-ik fordulóban:

$$x_i(t+1) = f_0(t) \quad ha \ \sum_{i=1}^{n} x_i(t) \leq K$$
$$x_i(t+1) = f_1(t) \quad ha \ \sum_{i=1}^{n} x_i(t) > K$$

ahol $f_0(x) = a_I + b_I x$ a növelési, $f_1(x) = a_D + b_D x$ a csökkentési stratégia.

Speciálsi esetek:

- **M**ultiplcative **I**ncrease **M**ultiplcative **D**ecrease - MIMD:

$$f_0(x) = b_I x \quad (b_I > 1)$$
$$f_1(x) = b_D x \quad (b_D < 1)$$

- **A**dditive **I**ncrease **A**dditive **D**ecrease - AIAD:

$$f_0(x) = a_I + x \quad (a_I > 0)$$
$$f_1(x) = a_D + x \quad (a_D < 0)$$

- **A**dditive **I**ncrease **M**ultiplcative **D**ecrease - AIMD:

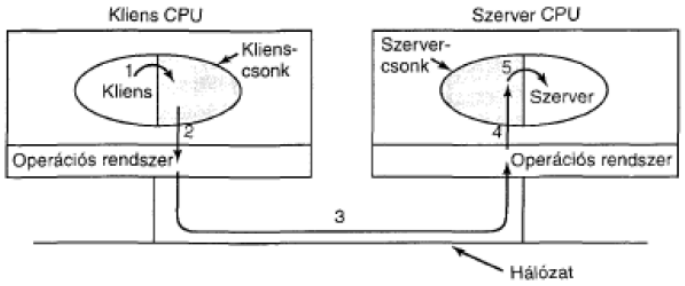
$$f_0(x) = a_I + x \quad (a_I > 0)$$
$$f_1(x) = b_D x \quad (b_D < 1)$$

Multiplexálás, demultiplexálás

Multiplexelés alatt a telekommunikációban azt az eljárást értik, amikor két vagy több csatornát összefognak egy csatornába úgy, hogy az inverz multiplexelés művelettel, vagy demultiplexeléssel, vagy demuxálással elő tudják állítani az eredeti csatornákat. Az eredeti csatornák egy úgynevezett kódolási sémával azonosíthatóak.

Interakciós modellek

- Kétirányú bájtfolyam
  Az adatok két egymással ellentétes irányú bájt-sorozatként kerülnek átvitelre. A tartalom nem interpretálódik Az adatcsomagok időbeli viselkedése megváltozhat: átvitel sebessége növekedhet, csökkenhet, más késés, más sorrendben is megérkezhetnek. Megpróbálja az adatcsomagokat időben egymáshoz közel kiszállítani. Megpróbálja az átviteli közeget hatékonyan használni.
- RPC
  A távoli gépen futtatandó eljárás eléréséhez hálózati kommunikációra van szükség, ezt az eljáráshívási mechanizmust az RPC (Remote Procedure Call) fedi el.
  A hívás lépései:
  1. A kliensfolyamat lokálisan meghívja a klienscsonkot.
  2. Az becsomagolja az eljárás azonosítóját és paramétereit, meghívja az OS-t.
  3. Az átküldi az üzenetet a távoli OS-nek.
  4. Az átadja az üzenetet a szervercsonknak.
  5. Az kicsomagolja a paramétereket, átadja a szervernek.
  6. A szerver lokálisan meghívja az eljárást, megkapja a visszatérési értéket.
  7. Ennek visszaküldése a klienshez hasonlóan zajlik, fordított irányban.



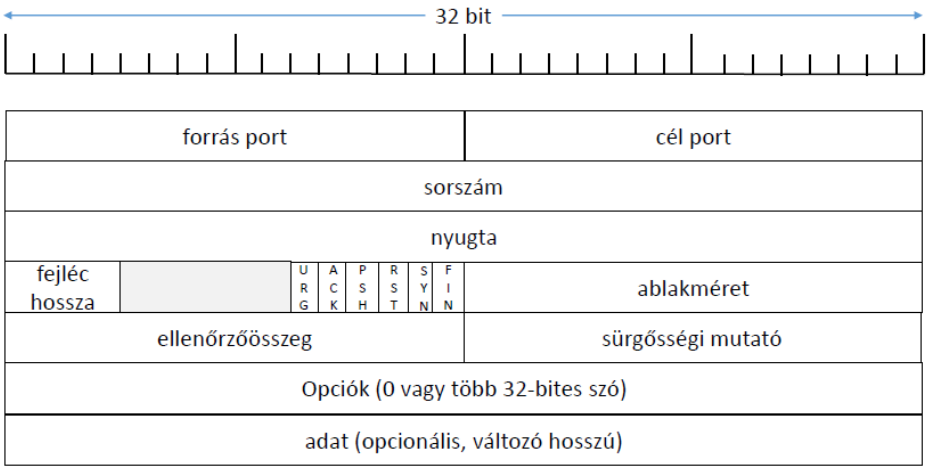
ábra 28: RPC

Protokollok

- TCP
  - Megbízható adatfolyam létrehozása két végpont között
  - Az alkalmazási réteg adatáramát osztja csomagokra
  - A másik végpont a csomagok fogadásról nyugtát küld

  A TCP fejléc tartalma:
  - küldő port(16 bit)
    A küldő folyamatot azonosítja
  - cél port(16 bit)
    A címzett folyamat azonosítója
  - sorszám(32 bit)
    Az első adatbájt sorszáma az aktuális szegmensen belül. Ha a SYN jelzőbit értéke 1, akkor ez a sorszám a kezdeti sorszám, azaz az első adatbájt sorszáma a kezdeti sorszám + 1 lesz.
  - nyugtaszám(32 bit)
    Ha az ACK jelzőbit értéke 1, akkor a fogadó által következőnek fogadni kívánt sorszámot tartalmazza. Minden kapcsolat felépítés esetén elküldésre kerül.

- fejléc hossza (4 bit)
  A TCP fejléc hossza 32-bites egységekben.
- Ablak(16 bit)
  A nyugtázott bájttal kezdődően hány bájtot lehet elküldeni. (A 0 érték is érvényes.)
- Ellenőrzőösszeg(16 bit)
  Az adat-, fej-, és pszeudofejrész ellenőrzésére.
- Opciók(0-40 bájt)
  A szabványos fejlécen kívüli lehetőségekre tervezték. Legfontosabb ilyen lehetőség az MSS, azaz a legnagyobb szegmens méret megadása. További opciók: MD5-aláírás, TCP-AO, "usertimeout", stb.
- sürgősségi mutató(16 bit)
  A sürgős adat bájtban mért helyét jelzi a jelenlegi bájtsorszámhoz viszonyítva.
- Jelző bitek (6)
  1. URG – Sürgős jelzőbit.
  2. ACK – nyugta jelzés.
  3. PSH – Az jelzi, hogy gyors adattovábbítás kell a felhasználói rétegnek.
  4. RST – Kapcsolat egyoldalú bontását jelzi.
  5. SYN – Sorszám szinkronizációtjelez.
  6. FIN – Adatfolyam végét jelzi.



ábra 29: TCP Fejléc

TCP jellemzői:

- Kapcsolatorientált
- Megbízható
- Kétirányú bájtfolyam

- UDP
  - Egyszerű, nem megbízható szolgáltatás csomagok küldésére
  - Az alkalmazási réteg határozza meg a csomag méretét
  - Az inputot egy datagrammá alakítja

  Összeköttetés nélküli protokoll. Olyan szegmenseket használ az átvitelhez, amelyek egy 8 bájtos fejrészből, valamint a felhasználói adatokból állnak.
  A Fejrész tartalmaz:
  - egy forrásportot(2 bájt);
  - egy célportot(2 bájt);
  - egy UDP szegmens hossz értéket (2 bájt);
  - egy UDP ellenőrzőösszeget (2 bájt)

Az UDP nem végez forgalomszabályozást, hibakezelést vagy újraküldést egy rossz szegmens fogadása után. Kliens-szerver alkalmazások esetén kifejezetten hasznos lehet az UDP a rövid üzenetek miatt.

Chapter 17

17

Záróvizsga tételsor
17. Osztott rendszerek

Ancsin Ádám

Osztott rendszerek
Folyamat fogalma, elosztott rendszerek tulajdonságai és felépítése, elnevezési rendszerek, kommunikáció, szinkronizáció, konzisztencia.

1 Folyamatok, szálak

Szál: A szál (thread) a processzor egyfajta szoftveres megfelelője, minimális kontextussal. Ha a szálat megállítjuk, a kontextus elmenthető és továbbfuttatáshoz visszatölthető.

Folyamat: A folyamat (process vagy task) egy vagy több szálat összefogó nagyobb egység. Egy folyamat szálai közös memóriaterületen (címtartományon) dolgoznak, azonban különböző folyamatok nem látják egymás memóriaterületét.

Kontextusváltás: A másik folyamatnak/szálnak történő vezérlésátadás, így egy processzor több szálat/folyamatot is végre tud hajtani.

Szál vs. folyamat: A szálak közötti váltáshoz nem kell igénybe venni az oprendszer szolgáltatásait, míg a folyamatok közötti váltásnál ahhoz, hogy a régi és új folyamat memóriaterülete elkülönüljön a memóriavezérlő (MMU) tartalmának jó részét át kell írni, amihez csak a kernel szintnek van joga. A folyamatok létrehozása, törlése és a kontextusváltás közöttük sokkal költségesebb a szálakénál.

2 Elosztott rendszerek tulajdonságai és felépítése

Elosztott rendszer fogalma: Az elosztott rendszer önálló számítógépek olyan összessége, amely kezelői számára egyetlen koherens rendszernek tűnik.

2.1 Az elosztott rendszer céljai, tulajdonságai

Az elosztott rendszer céljai a következők:

- Távoli erőforrások elérhetővé tétele
- Átlátszóság (transparency)
- Nyitottság (openness)
- Skálázhatóság

2.1.1 Átlátszóság

Az átlátszóság nem más, mint az erőforrásokkal kapcsolatos különböző információk elrejtése a felhasználó elől. Az alapján, hogy mit rejtünk el, többféle fajtája létezik:

Fajta	Angolul	Mit rejt el az erőforrással kapcsolatban?
Hozzáférési/elérési	Access	Adatábrázolás; elérés technikai részletei
Elhelyezési	Location	Fizikai elhelyezkedés
Áthelyezési	Migration	Elhelyezési + a hely meg is változhat
Mozgatási	Relocation	Áthelyezési + használat közben is történhet az áthelyezés
Többszörözési	Replication	Az erőforrásnak több másolata is lehet a rendszerben
Egyidejűségi	Concurrency	Több versenyhelyzetű felhasználó is elérheti egyszerre
Meghibásodási	Failure	Meghibásodhat és újra üzembe állhat

ábra 1: Az átlátszóság különböző típusai.

2.1.2 Nyitottság

A rendszer képes más nyitott rendszerek számára szolgáltatásokat nyújtani, és azok szolgáltatásait igénybe venni:

- A rendszerek jól definiált interfészekkel rendelkeznek.
- Az alkalmazások hordozhatóságát (portability) minél inkább támogatják.
- Könnyen elérhető a rendszerek együttműködése.

A nyitott elosztott rendszer legyen könnyen alkalmazható heterogén környezetben, azaz különböző hardvereken, platformokon, programozási nyelveken.

Implementálása:

- Fontos, hogy a rendszer könnyen cserélhető elemekből álljon.
- Belső interfészek használata, nem egyetlen monolitikus rendszer.
- A rendszernek minél jobban paraméterezhetőnek kell lennie.
- Egyetlen komponens megváltoztatása/cseréje lehetőleg minél kevésbé hasson a rendszer más részeire.

2.1.3 Skálázhatóság

Többféle jelentése van, 3 fontos dimenzió:

1. méret szerinti skálázhatóság: a felhasználók és/vagy folyamatok száma
2. földrajzi skálázhatóság: a csúcsok közötti legnagyobb távolság
3. adminisztrációs skálázhatóság: az adminisztrációs tartományok száma

Ezek közül a legtöbb rendszer a méret szerinti skálázhatóságot kezeli, ennek egy lehetséges megvalósítási módja erősebb szerverek használata. A másik kettőt nehezebb kezelni.

Technikák a skálázhatóság megvalósítására:

- A kommunikációs késleltetés elfedése azzal, hogy a válaszra várás közben más tevékenységet végzünk. Ehhez aszinkron kommunikáció szükséges.
- Elosztás: az adatokat és számításokat több számítógép tárolja/végzi (pl. amit lehet, a klienssel számoltatunk ki, elosztott elnevezési rendszerek használata, stb.)
- Replikáció/cache-elés: Több számítógép tárolja egy adat másolatait

A skálázhatóságnak ára van. Több másolat fenntartása inkonzisztenciához vezethet (ha módosítjuk az egyiket, az eltérhet a többitől). Ez globális szinkronizációval kikerülhető (minden egyes változtatás után az összes másolatot frissítjük), viszont a globális szinkronizáció rosszul skálázódik. Emiatt sok esetben fel kell hagynunk a globális szinkronizációval, ez viszont bizonyos mértékű inkonzisztenciát eredményez. Rendszerfüggő, hogy ez milyen mértékben megengedett. A cél az, hogy az inkonzisztencia mértéke a megengedett szint alatt maradjon.

2.2 Elosztott rendszerek típusai

Főbb típusok:

- Elosztott számítási rendszerek:
- Elosztott információs rendszerek
- Elosztott átható rendszerek

2.2.1 Elosztott számítási rendszerek

Célja számítások végzése nagy teljesítménnyel.

Cluster (fürt): Lokális hálózatra kapcsolt számítógépek összessége. Homogén rendszer (ugyanaz az oprendszer, hardveresen hasonlóak), központosított vezérléssel (általában egy gépre).

Grid (rács) Nagyméretű hálózatokra is kiterjedhet, akár több szervezeti egységen is átívelhet. Heterogén architektúra jellemzi.

Cloud(felhő): Többrétegű architektúra: hardver, infrastruktúra, platform, alkalmazás.

2.2.2 Elosztott információs rendszerek

Az elsődleges cél általában adatok kezelése, illetve más információs rendszerek elérése. Például tranzakciókezelő rendszerek.

A tranzakció adatok összességén (pl. egy adatbázison, adatbázis objektumon, stb.) végzett művelet (lehetnek részműveletei). A tranzakciókkal szemben az alábbi követelményeket szokás támasztani (ACID):

- Oszthatatlan, elemi (atomicity): Vagy a teljes tranzakció végbemegy minden részműveletével, vagy az adattárház egyáltalán nem változik.
- Konzisztens (consistency): Az adattárra akkor mondjuk, hogy érvényes, ha bizonyos, az adott adattárra megfogalmazott feltételek teljesülnek. Egy tranzakció konzisztens, ha érvényes állapotot állít elő a tranzakció végén.
- Elkülöníthető, sorosítható (isolation): Egyszerre zajló tranzakciók olyan eredményt adnak, mintha egymás után hajtódtak volna végre.
- Tartósság (durability): Végrehajtás után az eredményt tartós adattárolóra mentjük, így az összeomlás esetén visszaállítható.

2.3 Elosztott rendszerek felépítése

Alapötlet: A rendszer elemeit szervezzük logikai szerepük szerint különböző komponensekbe, és ezeket osszuk el a rendszer gépein.

2.3.1 Központosított architektúrák

Kliens-szerver modell: Egyes folyamatok (szerverek) szolgáltatásokat ajánlanak, míg más folyamatok (kliensek) ezeket a szolgáltatásokat szeretnék használni. A kliens kérést küld a szervernek, amire a szerver válaszol, így veszi igénybe a szolgáltatást. A kliens és szerver folyamatok különböző gépeken lehetnek.

2.3.2 Többrétegű architektúrák

Az elosztott információs rendszerek gyakran három logikai rétegre (layer vagy tier) vannak tagolva:

- Megjelenítés: az alkalmazás felhasználói felületét alkotó komponensekből áll.
- Üzleti logika: az alkalmazás működését írja le konkrét adatok nélkül
- Perzisztencia: az adatok tartós tárolása

2.3.3 Decentralizált architektúrák

Peer-to-peer (P2P): A csúcsok (peer-ek) között többnyire nincsenek kitüntetett szerepűek.

Overlay hálózat: A gráfban szomszédos csúcsok fizikailag lehetnek távol egymástól, a rendszer elfedi, hogy a köztük lévő kommunikáció több gépen keresztül zajlik. A legtöbb P2P rendszer overlay hálózatra épül.

P2P rendszerek fajtái:

- Strukturált P2P: A csúcsok által kiadott gráfszerkezet rögzített. A csúcsokat valamilyen struktúra szerint overlay hálózatba szervezzük és a csúcsoktól az azonosítójuk alapján lehet szolgáltatásokat igénybe venni. Pl.: elosztott hasítótábla (DHT).
- Struktúrálatlan P2P: Az ilyen rendszerek igyekeznek véletlen gráfstruktúrát fenntartani. Mindegyik csúcsnak csak részleges nézete van a gráfról. Minden P csúcs időnként véletlenszerűen kiválaszt egy Q szomszédot. P és Q információt cserélnek és elküldik egymásnak az általuk ismert csúcsokat.
- Hibrid P2P: néhány csúcsnak speciális szerepe van

Superpeer: Olyan csúcs, aminek külön feladata van, pl. kereséshez index fenntartása, a hálózat állapotának felügyelete, csúcsok közötti kapcsolatok létrehozása.

3 Elnevezési rendszerek

Az elosztott rendszerek entitásai a kapcsolódási pontjaikon (access point) keresztül érhetőek el. Ezeket távolról a címük azonosítja, amely megnevezi az adott pontot.

Célszerű lehet az entitást a kapcsolódási pontjaitól függetlenül is elnevezni. Az ilyen nevek helyfüggetlenek (location independent).

Egyszerű név: Nincs szerkezete, tartalmaz véletlen szöveg. Csak összehasonlításra használható.

Azonosító: Egy név azonosító, ha egy-egy kapcsolatban áll a megnevezett entitással, és ez a hozzárendelés maradandó, azaz a név később nem hivatkozhat más egyedre.

3.1 Strukturálatlan nevek

3.1.1 Egyszerű megoldások

Broadcasting: Kihirdetjük az azonosítót a hálózaton. Az egyed visszaküldi jelenlegi címét. Hátrányai:

- Lokális hálózatokon túl nem skálázódik.
- A hálózaton minden gépnek figyelnie kell a beérkező kérésre.

Továbbítómutató: Amikor az egyed elköltözik, egy mutató marad utána az új helyére.

- A kliens elől el van fedve, hogy a szoftver továbbítómutató-láncot old fel.
- A megtalált címet vissza lehet küldeni a klienshez, így a további feloldások gyorsabban mennek.
- Földrajzi skálázási problémák:

– A hosszú láncok nem hibatűrőek.
– A feloldás hosszú időbe telik.
– Külön mechanizmus szükséges a láncok rövidítésére.

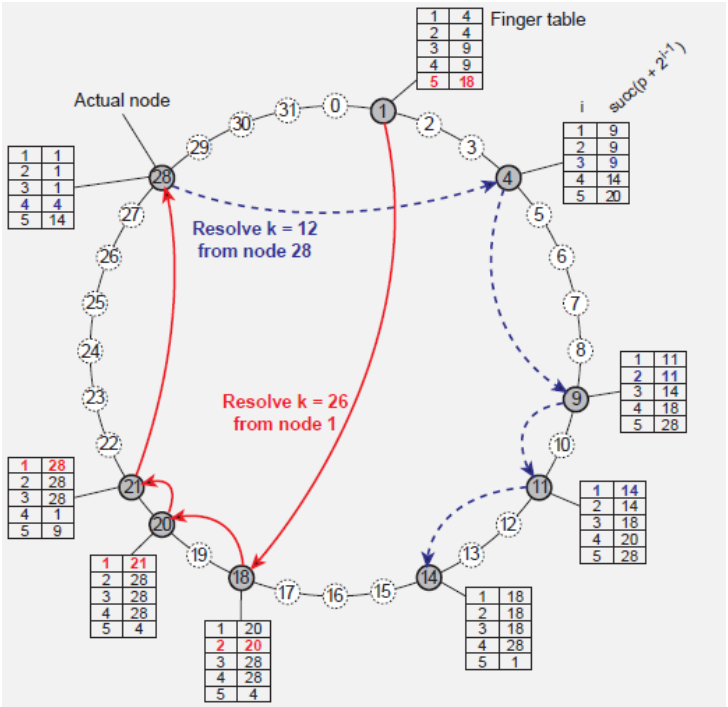
3.1.2 Otthon alapú megoldások

Egyrétegű rendszer: Az egyedhez tartozik egy otthon, ez tartja számon az egyed jelenlegi címét. Az egyed otthoni címe (home address - HA) be van jegyezve egy névszolgáltatásba. Az otthon számon tartja a jelenlegi címet (foreign address - FA). A kliens az otthonhoz kapcsolódik, onnan kapja meg a címet.

Kétrétegű rendszer: Az egyes környékeken feljegyezzük, hogy mely egyedek tartózkodnak a közelben. A névfeloldás először ezt a jegyzéket vizsgálja meg és ha az egyed nincs a környéken, akkor kell az otthonhoz fordulni.

3.1.3 Elosztott hasítótábla

Elosztott hasítótáblát (DHT) készítünk, ebben csúcsok tárolnak egyedeket. Az N csúcs gyűrű overlay szerkezetbe van szervezve. Minden csúcshoz hozzárendelünk egy m bites azonosítót, és mindegyik entitáshoz egy m bites kulcsot ($N \leq 2^m$). A k kulcsú egyed felelőse az az id azonosítójú csúcs, amelyre $k \leq id$, és nincs köztük másik csúcs. Ezt a csúcsot a kulcs rákövetkezőjének is szokás nevezni: $succ(k)$. Mindegyik p csúcs egy FT_p finger table-t tárol m bejegyzéssel: $FT_p[i] = succ(p+2^{i-1})$. Bináris (jellegű) keresést szeretnénk elérni, ezért minden lépés felezi a keresési tartományt. A k kulcsú egyed kikereséséhez (ha nem a jelenlegi csúcs tartalmazza) a kérést továbbítjuk ahhoz a j indexű csúcshoz, melyre $FT_p[j] \leq k < FT_p[j+1]$, illetve, ha $p < k < FT_p[1]$, akkor is $FT_p[1]$-hez irányítjuk a kérést.



ábra 2: Példa DHT-re finger table-el.

3.1.4 Hierarchikus módszerek

Hierarchical Location Services(HLS): A hálózatot osszuk fel tartományokra, és mindegyik tartományhoz tartozzon katalógus. Építsünk hierarchiát a katalógusokból.

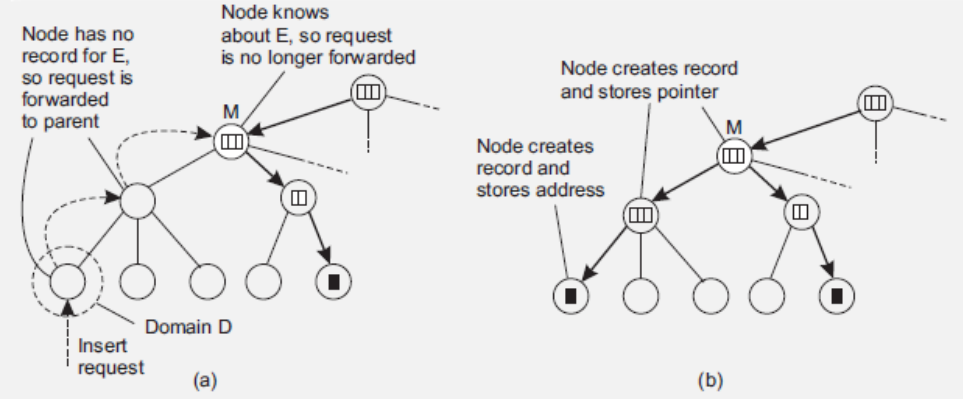
A csúcsokban tárolt adatok:

- Az E egyed címe egy levélben található.

- A gyökértől az E leveléig vezető úton minden belső csúcsban van egy mutató a lefelé következő csúcsra az úton.
- Mivel a gyökér minden út kiindulópontja, minden egyedről van információja.

Keresés a fában: A kliens tartományából indul a keresés. Felmegyünk addig a fában, amíg olyan csúcshoz nem érünk, amelyik tud E-ről, majd követjük a mutatókat a levélig, amely tudja E címét. Mivel a gyökér minden egyedet ismer, a terminálás garantált.

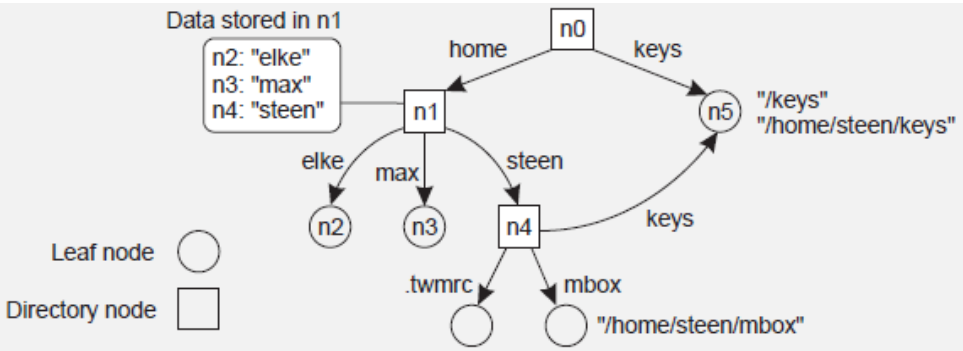
Beszúrás a fában: Ugyanaddig megyünk felfelé a fában, mint keresésnél, majd a belső csúcsokban mutatókat helyezünk el.



ábra 3: Beszúrás a fában HLS-nél.

3.2 Strukturált nevek

Névtér: Gyökeres, irányított, élcímkézett gráf, a levelek tartalmazzák a megnevezett egyedeket, a belső csúcsokat katalógusoknak vagy könyvtáraknak nevezzük. Az egyedhez vezető út címkéit összeolvasva kapjuk az egyed egy nevét. A bejárt út, ha a gyökértől indul, abszolút útvonalnév, ha belső csúcsból indul, relatív útvonalnév. Mivel egy egyedhez több út is vezethet, több neve is lehet.



ábra 4: Példa névtérre.

A névtér csúcsaiban (akár levélben, akár belső csúcsban) különféle attribútumokat is eltárolhatunk, pl. az egyed típusát, azonosítóját, helyét/címét, más neveit, stb.

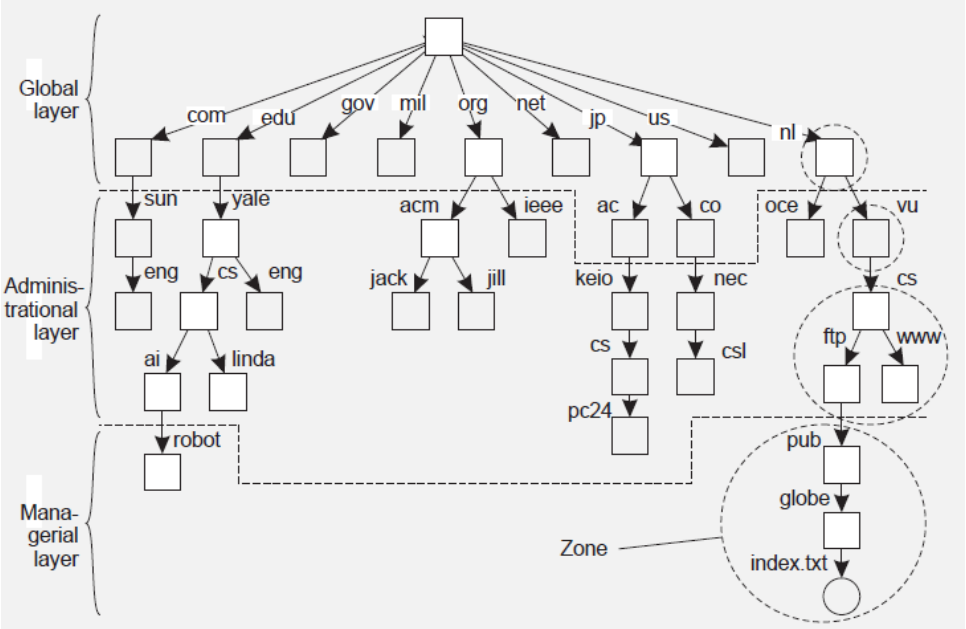
Névfeloldás: Kiinduló csúcsra van szükség a névfeloldás megkezdéséhez. A gyökér elérhetőségét a név jellegétől függő környezet biztosítja, pl.:

- www.inf.elte.hu : egy DNS névszerver
- /home/steen/mbox : a lokális NFS fájlszerver
- 0031204447784 : a telefonos hálózat

- 157.181.161.79 : a www.inf.elte.hu webszerverhez vezető út

Névtér implementációja - DNS: Ha nagy névterünk van, el kell osztani a gráfot a gépek között, hogy hatékonnyá tegyük a névfeloldást és a névtér kezelését. Ilyen nagy névtér a DNS (Domain Name System).
A DNS névtérnek alapvetően 3 szintjét különböztetjük meg:

- Globális szint: Ide tartozik a gyökér és a felsőbb csúcsok (TLD-k, pl. országokhoz tartozó csúcsok - .hu, .uk, stb.). A szervezetek ezt közösen kezelik.
- Szervezeti szint: Egy-egy szervezet által kezelt csúcsok szintje (pl. elte.hu, stb.).
- Kezelői szint: Egy adott szervezeten belül kezelt csúcsok (pl. elte.hu-n belüli csúcsok)



ábra 5: A DNS névtér egy része.

A névfeloldás különöző megközelítései: DNS névtér esetén alapvetően két különböző névfeloldási megközelítést alkalmazunk:

- Rekurzív névfeloldás: A rekurzív névfeloldás során a névszerverek egymás között kommunikálva oldják fel a neveket, a kliensoldali névfeloldóhoz rögtön a válasz érkezik.
- Iteratív névfeloldás: A névfeloldást a gyökér névszerverek egyikétől indítjuk. Az iteratív névfeloldás során a névnek mindig csak egy komponensét oldjuk fel, a megszólított névszerver az ehhez tartozó névszerver címét adja vissza (ha a kliensoldali névfeloldó megkapja ezt a címet, a következő komponens feloldását ettől a névszervertől kéri - ez addig megy, míg teljesen fel nem oldjuk a nevet).

Skálázhatóság: Mivel sok kérést kell kezelni rövid idő alatt, ezért a globális szint névszerverei nagy terhelést kapnának. Mivel a felső szinteken a gráf ritkán változik, ezért az ezeken a szinteken található csúcsok adatairól több szerveren is tarthatunk másolatot, így a keresést közelebbről indíthatjuk (pl. van több gyökér névszerver, a hozzánk legközelebbihez fordulunk).

3.2.1 Attribútumalapú nevek

Az egyedeket sokszor kényelmes lehet tulajdonságaik (attribútumaik) alapján keresni, viszont ha bármilyen kombinációban megadhatunk attribútumértékeket, akkor a kereséshez az összes egyedet érintenünk kell, ami nem hatékony.

X.500, LDAP: A katalógusszolgáltatásokban az attribútumokra megkötések érvényesek (X.500 szabvány), amelyet az LDAP protokollon keresztül szokás elérni. Az elnevezési rendszer fastruktúrájú, élei attribútum-érték párokkal címzettek. Az egyedekre az útjuk jellemzői vonatkoznak, és további párokat is tartalmazhatnak.

4 Kommunikáció

4.1 Köztesréteg

A köztesrétegbe (middleware) olyan szolgáltatásokat és protokollokat szokás sorolni, amelyek sokfajta alkalmazáshoz lehetnek hasznosak és alapvetően a rendszer egyedei közötti összekötő kapocsként szolgálnak.

- Kommunikációs protokollok
- Sorosítás (szerializáció, marshalling), adatok reprezentációjának átalakítása
- Elnevezési protokollok az erőforrások megosztásának könnyítésére
- Biztonsági protokollok a kommunikáció biztonságosabbá tételére
- Skálázási mechanizmusok adatok replikációjára és gyorsítótárazására

4.2 A kommunikáció fajtái

A kommunikáció lehet:

- időleges (transient) vagy megtartó (persistent):
  - időleges: a kommunikációs rendszer elveti az üzenetet, ha az nem kézbesíthető
  - megtartó: a kommunikációs rendszer hajlandó huzamosabb ideig tárolni az üzenetet
- szinkron vagy aszinkron
  - szinkron: a küldő vár a válaszra, addig blokkolódik
  - aszinkron: a küldő nem vár a válaszra, hanem más tevékenységet folytat

4.2.1 Kliens-szerver modell

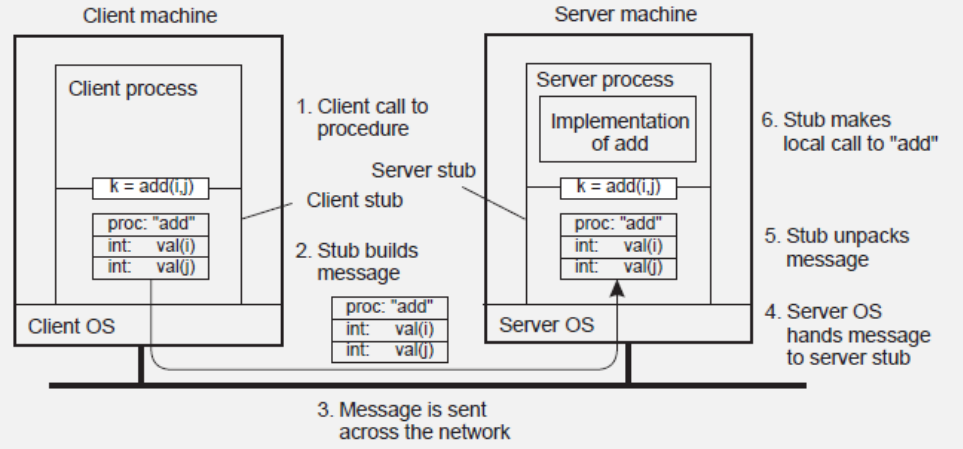
A kliens-szerver modell jellemzően időleges, szinkron kommunikációt végez, ahol a kliensnek és a szervernek egyidejűleg kell aktívnak lenni. A kliens a kérés küldése után blokkolódik, vár a szerver válaszára. A szerver csak a kliensek fogadásával és a kérések feldolgozásával foglalkozik.

4.2.2 Távoli eljáráshívás (RPC)

A távoli eljáráshívásnál egy távoli gépen szeretnénk futtatni egy alprogramot. Ehhez hálózati kommunikáció szükséges, amit elfedünk egy eljáráshívással.

A hívás lépései:

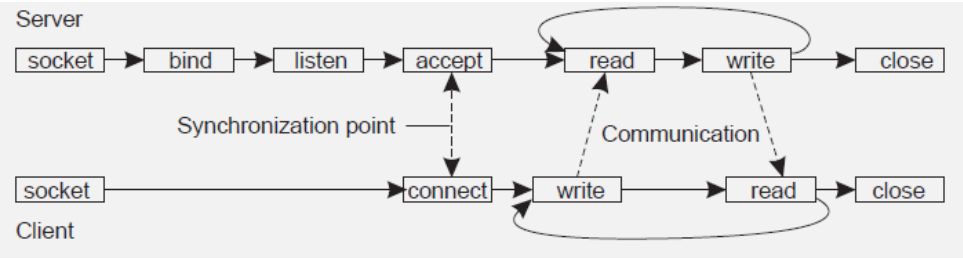
1. A kliensfolyamat lokálisan meghívja a klienscsonkot (client stub).
2. A klienscsonk becsomagolja az eljárás azonosítóját és paramétereit. Meghívja az oprendszert.
3. A lokális gép oprendszere elküldi a csomagot a távoli gép oprendszerének.
4. Az átadja az üzenetet a szervercsonknak (server stub).
5. A szervercsonk kicsomagolja az azonosítót és a paramétereket, amiket átad a szerverfolyamatnak.
6. A szerverfolyamat lokálisan meghívja az eljárást, megkapja a visszatérési értéket.
7. A visszatérési érték visszaküldése a kliensfolyamatnak hasonlóan történik, fordított irányban.



ábra 6: A távoli eljáráshívás lépései.

4.2.3 Socket

Az időleges kommunikáció egy módja.



ábra 7: Kommunikáció socket-el.

4.2.4 Üzenetorientált köztesréteg (MOM)

Az üzenetorientált köztesréteg (MOM - message-oriented middleware) egy megtartó, aszinkron kommunikációs architektúra. Segítségével a folyamatok üzeneteket küldhetnek egymásnak. A küldő félnek nem kell a válaszra várnia, addig foglalkozhat mással.

A MOM várakozási sorokat tart fenn a rendszer gépein. A kliensek az alábbi műveleteket használhatják a várakozási sorokra:

- PUT: Üzenetet tesz a sor végére.
- GET: Blokkol, amíg a sor üres, majd kiveszi az első üzenetet
- POLL: Lekérdezi, hogy van-e üzenet. Ha van, leveszi az elsőt. Ha nincs, nem blokkol, folytatja a tevékenységét.
- NOTIFY: Kezelőrutint telepít a várakozási sorhoz, amely minden beérkező üzenetre meghívódik.

Az üzenetsorkezelő rendszerek feltételezik, hogy a rendszer minden eleme közös protokollt használ, azaz az üzenetek szerkezete és adatábrázolása megegyezik. A kérdés: mi van akkor, ha heterogén a rendszerünk? Erre szolgál az üzenetközvetítő (message broker), amely heterogén rendszerben gondoskodik a megfelelő konverziókról, azaz átalakítja az üzenetet a fogadó által használt formátumra. Általában proxy-ként is működik, azaz a közvetítés mellett más funkciókat is nyújt, pl. biztonsági funkciókat.

4.2.5 Folyam (stream)

Az eddig tárgyalt kommunikációfajtákban közös, hogy az adategységek közötti időbeli kapcsolat nem befolyásolja azok jelentését, folyamatos médiánál (pl. audio, videó, szenzoradatok) viszont az adatok időfüggőek, ezért a kommunikáció időbeliségével kapcsolatban izokrón megkötést teszünk, ami felső és

alsó korlátot is ad a csomagok átvitelének idejére.

Folyam: Ilyen izokrón adatátvitelt lehetővé tevő kommunikációs forma a folyam. Főbb jellemzői:

- Egyirányú
- Legtöbbször egy forrástól irányul egy vagy több nyelő felé
- A forrás és/vagy nyelő gyakran közvetlenül kapcsolódik olyan hardverelemekhez, mint pl. egy kamera, képernyő, mikrofon, stb.

Főbb típusai:

- Egyszerű folyam: egyfajta adatot továbbít, pl. egyetlen audiocsatornát, vagy csak videót.
- Összetett folyam: Többfajta adatot továbbít egyszerre, pl. videót többcsatornájú audióval (sztereó, 5.1, stb.). Az összetett folyam esetében biztosítani kell, hogy az alfolyamok a nyelőnél időben ne csússzanak el egymáshoz képest. Ennek egyik módja a szinkronizáció. Egy másik lehetséges módszer a multiplexálás és demultiplexálás. Ekkor a forrás egyetlen folyamot készít (multiplexálás). Itt az alfolyamok garantáltan szinkronban vannak egymással. A nyelőnél kell szétbontani a folyamot alfolyamokra (demultiplexálás).

QoS: A folyamokkal kapcsolatban sokfajta követelmény írható elő, ezeket összefoglaló néven a szolgáltás minőségének (QoS - Quality of Service) nevezzük. Ilyen jellemzők például a következők:

- Az átviteli sebesség, azaz a bitráta.
- A folyam elindításának legnagyobb megengedett késleltetése.
- A folyam adategységeinek megadott idő alatt el kell jutniuk a forrástól a nyelőig.
- Remegés (jitter): az adategységek beérkezési idejének egyenetlensége. Ennek csökkentésének egy módja a pufferelés.

5 Szinkronizáció

5.1 Órák szinkronizálása

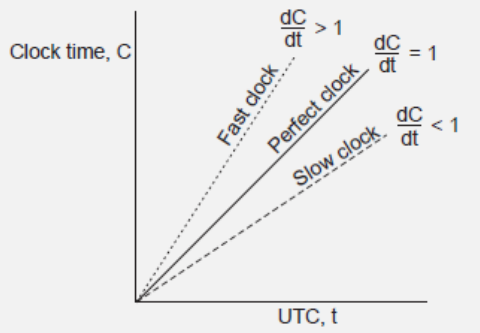
Néha a pontos időt szeretnénk megtudni, néha elég, hogy ha két időpont közül megállapítható, hogy melyik volt korábban. A világidő: UTC.

5.1.1 Fizikai órák

A fizikai idő elterjesztése: Ha a rendszerünkben van UTC-vevő, az megkapja a pontos időt. Ezt a következők figyelembevételével terjeszthetjük el a rendszeren belül.

- A p gép saját órája szerint az idő a t UTC-időpillanatban $C_p(t)$
- Ideális esetben az óra mindig pontos, azaz $C_p(t) = t$ minden t UTC-időpillanatra. Másképpen fogalmazva az óra sebessége mindig 1, azaz $dC/dt = 1$.
- A valóságban p órája vagy túl gyors, vagy túl lassú, de viszonylag pontos:

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$



ábra 8: Az óra sebessége.

Cristian algoritmusa: Csak megadott δ eltérést akarunk megengedni az óra sebességében. Mindegyik gép egy központi időszerverről kéri le a pontos időt legfeljebb $\frac{\delta}{2\rho}$ másodpercenként (ekkor tudunk δ eltérésen belül maradni). Az órát nem a megkapott időpontra kell állítani: bele kell számolni, hogy a szerver kezelte a kérést és a válasznak vissza kellett érkeznie a hálózaton.

Berkeley algoritmusa: Nem a pontos idő beállítása a cél, csak az, hogy a rendszeren belül minden gép ideje azonos legyen. Az időszerver időnként minden gép idejét bekéri, amiből átlagot von, majd mindenkit értesít, hogy a saját óráját mennyivel kell átállítania. Az idő egyik gépnél sem folyhat visszafelé, ezért ha valamelyik órát vissza kellene állítani, akkor ehelyett lelassítja az óráját addig, amíg a kívánt idő be nem áll.

5.1.2 Logikai órák

Az előbb-történt reláció: Az előbb-történt (happened-before) reláció az alábbi tulajdonságokkal bíró reláció. Annak jelölése, hogy a előbb történt, mint b: $a \to b$.

- Ha ugyanabban a folyamatban a előbb következett be, mint b, akkor $a \to b$.
- Ha a esemény egy üzenet küldése, b pedig ennek az üzenetnek a fogadása, akkor $a \to b$.
- Tranzitív: Ha $a \to b$ és $b \to c$, akkor $a \to c$.

Az idő és az előbb-történt reláció: Minden e eseményhez időbélyeget rendelünk, ami egy egész szám. Jelölése: $C(e)$, és megköveteljük az alábbi tulajdonságokat:

- Ha $a \to b$ egy folyamat eseményeire, akkor $C(a) < C(b)$
- Ha a esemény egy üzenet küldése, b pedig ennek az üzenetnek a fogadása, akkor $C(a) < C(b)$.

Ha van globális óra, akkor az időbélyeg elkészíthető. A továbbiakban azzal foglalkozunk, hogy mi van akkor, ha nincs globális óra.

Lampert-féle időbélyeg: Minden P_i folyamat egy C_i számlálót tart nyilván az alábbiak szerint:

- P_i minden eseménye eggyel növeli C_i-t.
- Az elküldött m üzenetre ráírjuk az időbélyeget: $ts(m) = C_i$.
- Ha az m üzenet beérkezik P_j folyamathoz, ott a számláló új értéke $C_j = \max\{C_j, ts(m)\} + 1$ lesz
- P_i és P_j egybeeső időbélyegjei közül tekintsük a P_i-belit elsőnek, ha $i < j$.

Pontosan sorbarendezett csoportcímzés: A P_i folyamat minden műveletet időbélyeggel ellátott üzenetben küld el. P_i egyúttal beteszi a küldött üzenetet a saját $queue_i$ prioritásos sorába. A P_j folyamat a beérkező üzeneteket az ő $queue_j$ prioritásos sorába teszi be az időbélyegnek megfelelő prioritással. Az üzenet érkezéséről mindegyik folyamatot értesíti. P_j akkor adja át a msg_i üzenet feldolgozásra, ha:

- msg_i a $queue_j$ elején található, azaz az ő időbélyege a legkisebb

- a $queue_j$ sorban minden $P_k, k \neq i$ folyamatnak megtalálható legalább egy üzenete, amelynek msg_i-nél későbbi az időbélyege

Időbélyeg-vektor:

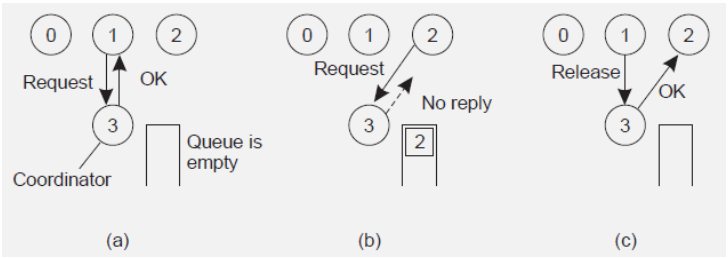
- P_i most már az összes folyamat idejét is számon tartja egy $VC_i[1..n]$ tömbben, ahol $VC_i[j]$ azon P_j-ben bekövetkezett események száma, amiről P_i tud.
- Az m üzenet elküldése során P_i megnöveli eggyel $VC_i[i]$ értékét és a teljes VC_i időbélyeg-vektort ráírja az üzenetre.
- Amikor az m üzenet megérkezik P_j-hez, amelyen a $ts(m)$ időbélyeg van, akkor
  1. $VC_j[k] := \max\{VC_j[k], ts_m[k]\}$
  2. $VC_j[j]$ megnő eggyel

5.2 Kölcsönös kizárás

Több folyamat egyszerre szeretne hozzáférni egy adott erőforráshoz. Ezt egyszerre csak egynek engedhetjük meg közülük, különben az erőforrás helytelen állapotba kerülhet.

5.2.1 Kölcsönös kizárás központi szerver használatával

Egy központi szerver a koordinátor, ő szabályozza az erőforráshoz való hozzáférést. Van egy várakozási sora. Ha az erőforrás szabad, akkor ha kérés érkezik rá, a szerver megadja a hozzáférést és foglalttá teszi. Ezután ha valaki más hozzá akar férni az erőforráshoz, akkor bekerül a várakozási sorba. Miután az első kliens elengedte az erőforrást, az ahhoz kerül, aki a sor elején van. Ha kiürült a sor és az utolsó kliens is elengedte az erőforrást, az újra szabaddá válik.



ábra 9: Példa központosított kölcsönös kizárásra.

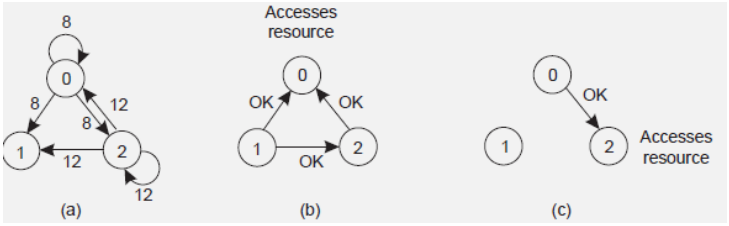
5.2.2 Decentralizált kölcsönös kizárás

Tegyük fel, hogy az erőforrás n-szeresen többszörözött, és minden replikátumhoz tartozik egy azt kezelő koordinátor. A hozzáférésről többségi szavazás dönt: legalább m koordinátor szükséges, ahol $m > \frac{n}{2}$. Feltesszük, hogy egy esetleges összeomlás után a koordinátor felépül, de a kiadott engedélyeket elfelejti.

5.2.3 Elosztott kölcsönös kizárás

Többszörözött az erőforrás. Amikor a kliens hozzá szeretne férni az erőforráshoz, kérést küld a koordinátornak időbélyeggel ellátva. Választ (hozzáférési engedélyt) akkor kap, ha:

- A koordinátor nem igényli az erőforrást, vagy
- a koordinátor is igényli az erőforrást, de kisebb az időbélyege.
- Különben a koordinátor átmenetileg nem válaszol.



ábra 10: Példa elosztott kölcsönös kizárásra.

5.2.4 Kölcsönös kizárás token ring-gel

A folyamatokat egy logikai gyűrűbe szervezzük. Egy tokent küldünk körbe. Amelyik folyamat birtokolja a tokent, az férhet hozzá az erőforráshoz.

5.3 Vezetőválasztás

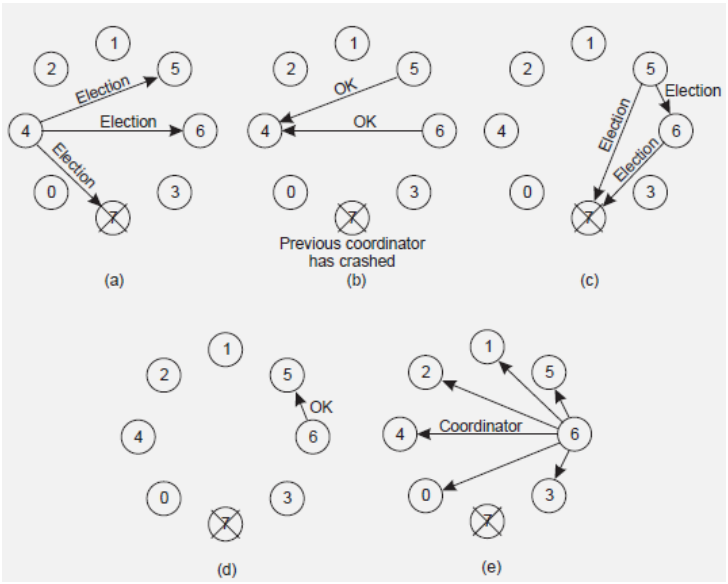
Sok algoritmusnak szüksége van arra, hogy kijelöljön egy folyamatot, amely a további lépéseket koordinálja.

5.3.1 Zsarnok-algoritmus

A folyamatoknak sorszámot adunk, melyek közül a legnagyobb sorszámút szeretnénk vezetőnek választani.

A zsarnok-algoritmus lépései:

1. A vezetőválasztás kezdeményezése. Bármelyik folyamat kezdeményezheti. Mindegyik olyan folyamatnak, amelyről nem tudja, hogy kisebb lenne az övénél a sorszáma, elküld egy üzenetet.
2. Ha a nagyobb sorszámú folyamat üzenetet kap egy kisebb sorszámútól, akkor visszaküld neki egy olyan üzenetet, amivel kiveszi a kisebb sorszámút a választásból.
3. Amelyik folyamat nem kap letiltó üzenet egy bizonyos időn belül, akkor ő lesz a vezető. Erről értesíti a többi folyamatot egy-egy üzenettel.



ábra 11: Példa a zsarnok-algoritmus működésére.

5.3.2 Vezetőválasztás gyűrűben

Logikai gyűrűnk van, a folyamatoknak vannak sorszámai. A legnagyobb sorszámú folyamatot szeretnénk vezetőnek választani. Bármelyik folyamat kezdeményezhet vezetőválasztást: elindít egy üzenetet a gyűrűn körbe, amelyre mindenki ráírja a a sorszámát. Ha egy folyamat összeomlott, az kimarad az üzenetküldésből. Amikor az üzenet visszajut a kezdeményezőhöz, minden aktív folyamat sorszáma szerepel rajta. Ezek közül a legnagyobb sorszámú lesz a vezető. Ezt egy másik üzenet körbeküldése tudatja mindenkivel.

Ha több folyamat kezdeményez egyszerre választást, az nem probléma, ugyanaz az eredmény adódik. Ha az üzenetek elvesznének, akkor újra lehet kezdeni a választást.

5.3.3 Superpeer-választás

A superpeer-eket úgy szeretnénk megválasztani, hogy teljesüljön rájuk:

- A többi csúcs alacsony késleltetéssel éri el őket.
- Egyenletesen vannak elosztva a hálózaton.
- A csúcsok megadott hányadát választjuk superpeer-nek.
- Egy superpeer korlátozott számú peer-t szolgál ki.

Megvalósítás DHT esetén: Ha m-bites azonosítókat használunk, és S superpeer-re van szükség, akkor a $k = \lceil \log_2 S \rceil$ felső bitet foglaljuk le a superpeer-ek számára. Így N csúcs esetén kb. $2^{k-m}N$ superpeer lesz.

A p kulcshoz tartozó superpeer a p AND $\underbrace{11...11}_{k}\underbrace{00..00}_{m-k}$ kulcs felelőse lesz.

6 Konzisztencia

Konfliktusos műveletek: A replikátumok konzisztensen tartásához biztosítani kell, hogy az egymással konfliktusba kerülhető műveletek minden replikátumon egyforma sorrendben futnak le. Írás-olvasás és írás-írás konfliktusok fordulhatnak elő.

Konzisztenciamodell: A konzisztenciamodell megszabja, milyen módokon használhatják a folyamatok az adatbázist. Ha a feltételek teljesülnek, az adattárat érvényesnek tekintjük.

Konzisztencia mértéke: A konzisztencia többféle módon is sérülhet: eltérhet a replikátumok számértéke, frissessége, meg nem történt frissítési műveletek száma.

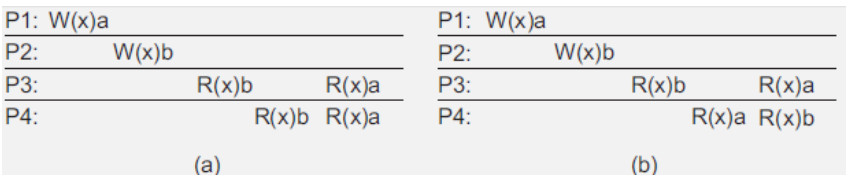
Conit: Az olyan adategység, amelyre közös feltételrendszer vonatkozik, a conit (consistency unit).

6.1 Soros konzisztencia

A feltételeket nem számértékekre, hanem írások/olvasások tényére alapozzuk. Jelölések:

- W(x) : x változót írta a folyamat
- R(x) : x változót olvasta a folyamat

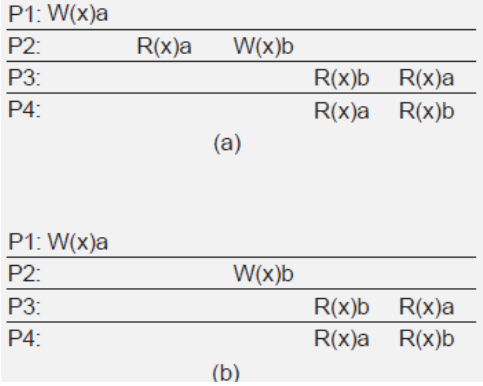
Soros konzisztencia esetén azt várjuk el, hogy a végrehajtás eredménye olyan legyen, mintha az összes folyamat összes művelete egy meghatározott sorrendben történt volna meg, megőrizve bármely adott folyamat saját műveletinek sorrendjét.



ábra 12: Példa: az (a) teljesíti, (b) nem a soros konzisztencia követelményeit.

6.2 Okozati konzisztencia

A potenciálisan okozati összefüggésben álló műveleteket kell mindegyik folyamatnak azonos sorrendben látnia. A konkurens írásokat a különböző folyamatok különböző sorrendben láthatják.



ábra 13: Példa: a (b) teljesíti, (a) nem az okozati konzisztencia követelményeit.

6.3 Kliensközpontú konzisztencia

Azt helyezzük most előtérbe, hogy a szervereken tárolt adatok hogyan látszanak egy adott kliens számára. A kliens mozog: különböző szerverekhez csatlakozik, és írási/olvasási műveleteket hajt végre.

Az A szerver után a B szerverhez csatlakozva különböző problémák léphetnek fel:

- Az A-ra feltöltött frissítések lehet, hogy nem jutottak még el B-hez.
- B-n lehet, hogy újabb adatok találhatóak, mint A-n.
- A B-re feltöltött frissítések ütközhetnek az A-ra feltöltöttekkel.

A cél az, hogy a kliens azokat az adatokat, amiket az A szerveren kezelt, ugyanolyan állapotban lássa B-n is. Ekkor az adatbázis konzisztensnek látszik a kliens számára.

6.3.1 Monoton olvasás

Ha egyszer a kliens kiolvasott egy értéket x-ből, minden ezután következő olvasás ezt adja, vagy ennél frissebb értéket.

Például levelezőkliens esetén minden korábban letöltött levelünknek meg kell lennie az új szerveren is.

6.3.2 Monoton írás

A kliens akkor írhatja x-et, ha kliens korábbi írásai x-re már befejeződtek.

Például verziókezelésnél minden korábbi verziónak meg kell lennie a szerveren, ha új verziót akarunk feltölteni.

6.3.3 Olvasd az írásodat

Ha kliens olvassa x-et, a saját legutolsó írásának eredményét kapja, vagy frissebbet.

Például a kliens a honlapját szerkeszti, majd megnézi az eredményt. Ahelyett, hogy a böngésző gyorsítótárából egy régebbi változat kerülne elő, a legfrissebbet szeretné látni.

6.3.4 Írás olvasás után

Ha a kliens kiolvasott egy értéket x-ből, minden ezután kiadott frissítési művelete x-nek legalább ennyire friss értékét módosítja.

Például egy fórumon a kliens csak olyan hozzászólásra tud válaszolni, amit már látott.

6.4 Tartalom replikálása

Különböző jellegű folyamatok tárolhatják a másolatokat:

- Tartós másolat: eredetszerver (origin server)
- Szerver által kezdeményezett másolat: replikátum kihelyezése egy szerverre, amikor az igényli az adatot
- Kliens által kezdeményezett másolat: kliensoldali gyorsítótár

6.4.1 Frissítés terjesztése

Megváltozott tartalmat több különféle módon lehet kliens-szerver architektúrában átadni:

- Kizárólag a frissítésről szóló értesítés/érvénytelenítés elterjesztése.
- Passzív replikáció: adatok átvitele egyik másolatról a másikra
- Aktív replikáció: frissítési művelet átvitele

A frissítést kezdeményezheti a szerver (küldésalapú frissítés), ekkor a szerver a kliens kérése nélkül elküldi a frissítést a kliensnek, vagy kezdeményezheti a kliens, aki kérvényezi a frissítést a szervertől (rendelésalapú frissítés).

Haszonbérlet (lease): A szerver ígéretet tesz a kliensnek, hogy átküldi a frissítést, amíg a haszonbérlet aktív.

Chapter 18

18

Záróvizsga tételsor

18. Adatbázisok tervezése és lekérdezése

Fekete Dóra

Adatbázisok tervezése és lekérdezése

Relációs adatmodell, egyed-kapcsolat modell és átalakítása relációs adatmodellbe. Relációs algebra, SQL. Az SQL procedurális kiterjesztése (PL/SQL vagy PSM). Relációs adatbázis-sémák tervezése, normálformák, dekompozíciók.

1 Relációs adatmodell, egyed-kapcsolat modell és átalakítása relációs adatmodellbe

1.1 Relációs adatmodell

Relációs adatmodell: adatok gyűjteményét kezeli.
Reláció: a gyűjtemény megadása (tábla).
Adatmodell: a valóság fogalmainak, kapcsolatainak, tevékenységeinek magasabb szintű ábrázolása, számítógép és felhasználó számára is megadja, hogy hogyan néznek ki az adatok. Adatok leírására szolgáló jelölés.
Részei:

1. az adat struktúrája
2. az adaton végezhető műveletek
3. az adatokra tett megszorítások

Egyéb fogalmak:

- Relációséma: *relációnév(sortípus)*, $R(A_1, ...A_n)$
- Előfordulás: példány, a sortípusnak megfelelő *véges sok* sor, $\{t_1, ...t_m\}$, ahol $t_i =< v_i 1...v_i n >$
- Attribútumok: adattípus : sortípus, <attr.név$_1$: értéktípus$_1$,...>
- Kulcsok: Egyszerű kulcs 1 attribútumból áll, összetett többből, nem lehet a relációban két különböző sor, aminek azonos a kulcsa.
- Külső kulcs: Idegen kulcs. $R(A_1, ...A_m)$ reláció, $X = \{A_{i_1}, ...A_{i_k}\}$ kulcs. $S(B_1, ...B_n)$ reláció, $Y = \{B_{j_1}, ...B_{j_k}\}$ idegen kulcs, ami az X-re hivatkozik a megadott attribútum sorrendben: B_{j_1} az A_{i_1}-re stb.
- Hivatkozási épség: megszorítás a két tábla együttes előfordulására. Ha $s \in S$ sor, akkor $\exists t \in R$ sor: $s[B_{j_1}, ...B_{j_k}] = t[A_{i_1}, ...A_{i_k}]$.

A relációs adatmodell több szempontból is előnyös, amik miatt elterjedt és kifinomult. Az adatmodell egy egyszerű és könnyen megérthető strukturális részt tartalmaz. A természetes táblázatos formát nem kell magyarázni, és jobban alkalmazható. A relációs modellben a fogalmi-logikai-fizikai szint teljesen szétválik, nagyfokú logikai és fizikai adatfüggetlenség. A felhasználó magas szinten, hozzá közel álló fogalmakkal dolgozik (implementáció rejtve). Elméleti megalapozottság, több absztrakt kezelő nyelv létezik, például relációs algebra (ezen alapul az SQL automatikus és hatékony lekérdezés optimalizálása). Műveleti része egyszerű kezelői felület, szabvány SQL.

Relációs adatbázis felépítése: Az adatbázis tulajdonképpen relációk halmaza. A megfelelő relációsémák halmaza adja az adatbázissémát (jelölése dupla szárú $\mathbb{R}$), $\mathbb{R} = \{R_1, ..., R_k\}$. A hozzá tartozó előfordulások

az adatbázis-előfordulás. Előfordulás tartalma: egyes relációk előfordulásai. Ez a koncepcionális szint, vagyis a fogalmi modell. Fizikai modell: a táblát valamilyen állományszerkezetben jeleníti meg (például szeriális állományban). A relációs adatbázis-kezelők indexelnek, indexelési mód: pl. B+ fa.

1.2 Egyed-kapcsolat modell

Elemei:

- Egyedhalmazok: hasonló egyedek összessége
- Attribútumok: megfigyelhető tulajdonságok, megfigyelt értékek, egyedek tulajdonságai
- Kapcsolatok: más egyedhalmazokkal való kapcsolatok
- Séma: $E(A_1, ...A_n)$ egyedhalmaz séma, E név, A_i tulajdonság, $DOM(A_i)$ a lehetséges értékek halmaza, pl *tanár(név, tanszék)*
- Előfordulás: Konkrét egyedek (entitások), $E = \{e_1, ...e_m\}, e_i(k) \in DOM(A_k)$ az egyedek halmaza. Minden attribútumban nem egyezhetnek meg $\rightarrow$ (vagyis az összes tulajdonság szuperkulcsot alkot), minimális szuperkulcs = kulcs.
- $K(E_1, E_2)$ bináris kapcsolat, $K(E_1, ...E_p)$ a kapcsolat sémája. K a kapcsolat neve, E_i az egyedhalmazok sémái, többágú kapcsolat, ha p¿2. pl *tanít(tanár, tárgy)*. Többágú kapcsolat átírható megfelelő számú binér kapcsolatra, 3-ágú 3-ra stb.
- $K(E_1, ...E_p)$ sémájú kapcsolat előfordulása, $K = \{(e_1, ...e_p)\}$ egyed p-esek halmaza, ahol $e_i \in E_i$. A kapcsolat előfordulásaira tett megszorítások határozzák meg a kapcsolat típusát.

Diagram: egyedhalmazok, kapcsolatok típusok, egyenek ábrázolása.
Szerepek: egyedhalmaz önmagával kapcsolódik
Kapcsolat attribútum: két egyedhalmaz együttes függvénye, de egyiké sem külön
Egyedhalmaz: az elsődleges kulcshoz tartozó tulajdonságokat aláhúzzuk

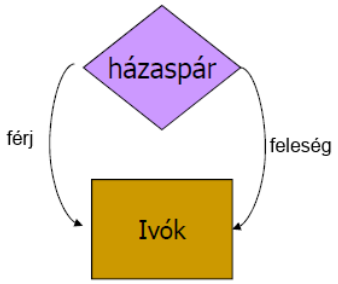


Figure 1: Szerepek

Kapcsolatok típusai (vegyünk hozzá egy $K(E_1, E_2)$ bináris kapcsolatot alapul):

- sok-egy: K előfordulásaiban minden E_1-beli egyedhez legfeljebb 1 E_2-beli tartozhat, pl *született(név,ország)*
- sok-sok: nincs megszorítása, minden E_1-beli egyedhez több E_2-beli egyed tartozhat, és fordítva, minden E_2-beli egyedhez több E_1-beli egyed tartozhat, pl *tanul(diák,nyelv)*
- egy-egy: sok-egy és egy-sok, vagyis minden E_1-beli egyedhez legfeljebb egy E_2-beli egyed tartozhat, és fordítva, minden E_2-beli egyedhez legfeljebb egy E_1-beli egyed tartozhat, pl *házaspár(férfi,nő)*

Lehet több kapcsolat is két egyedhalmaz között.
Alosztály: ,,isa" = ,,az-egy", öröklődés, speciális egy-egy kapcsolat. Összes E_1-belihez van egy E_2-beli, pl *isa(főnök, dolgozó)*

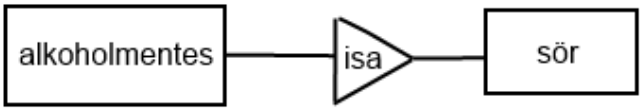


Figure 2: Alosztály

Kulcs: aláhúzással jelölik, összetett kulcsnál több attribútum van aláhúzva
Szuperkulcs: Az egyedhalmaz szuperkulcsa egy azonosító, vagyis olyan tulajdonság-halmaz, amelyről feltehető, hogy az egyedhalmaz előfordulásaiban nem szerepel két különböző egyed, amelyek ezeken a tulajdonságokon megegyeznek. Az összes tulajdonság mindig szuperkulcs.
Hivatkozási épség: kerek végződéssel jelölik, megszorítás

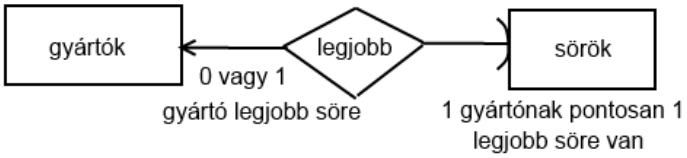


Figure 3: Hivatkozási épség

Gyenge egyedhalmaz: Téglalap dupla kontúrral. Önmagában nem azonosítható egyértelműen.

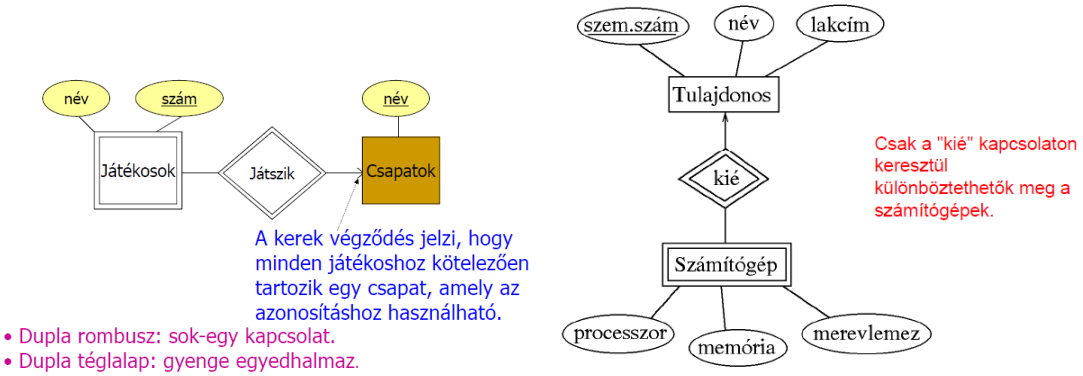


Figure 4: Gyenge egyedhalmaz

Tervezési alapelvek:

- valósághű modellezés: megfelelő tulajdonságok tartozzanak az egyedosztályokhoz, például a tanár neve ne a diák tulajdonságai közé tartozzon
- redundancia elkerülése: az *index(etr-kód,lakcím,tárgy,dátum,jegy)* rossz séma, mert a lakcím annyiszor ismétlődik, ahány vizsgajegye van a diáknak, helyette 2 sémát érdemes felvenni: *hallgató(etr-kód,lakcím)*, *vizsga(etr-kód,tárgy,dátum,jegy)*.
- egyszerűség: fölöslegesen ne vegyünk fel egyedosztályokat, például a *naptár(év,hónap,nap)* helyett a megfelelő helyen inkább dátum tulajdonságot használjunk
- tulajdonság vagy egyedosztály: például a vizsgajegy osztály helyett jegy tulajdonságot használjunk.

1.3 Egyed-kapcsolat modell átalakítása relációs adatmodellbe

KÖNYV (könyvszám, szerző, cím)
OLVASÓ (olvasószám, név, lakcím)
KÖLCSÖN (könyvszám, olvasószám, kivétel, visszahozás)

Figure 5: Példa: könyvtár adatmodellje

Összetett attribútumok leképezése: például ha a lakcímet (helység, utca, házszám) struktúrában akarjuk kezelni, akkor fel kell venni a sémába mindet attribútumként. *Többértékű attribútumok leképezése*:

- Megadás egyértékűként. Példa: többszerzős könyvnél egy mezőben soroljuk fel az összeset. Nem túl jó, mert nem lehet a szerzőket külön kezelni, és esetleg nem is fér el mind a mezőben.
- Megadás többértékűként.
  - Sorok többszörözése. Könyves példánál maradva, felveszünk annyi sort, ahány szerző van. Ez redundanciához vezet.
  - Új tábla felvétele. A *könyv(könyvszám, szerző, cím)* sémát az alábbi két sémával helyettesítjük: *könyv(könyvszám, cím)*, *szerző(könyvszám, szerző)*
  - Sorszámozás. Ha nem mindegy a szerzők sorrendje, akkor az előző megoldásban (új tábla) ki kell egészíteni a szerző táblát egy sorszám mezővel.

Kapcsolatok leképezése:

- egy-egy: beolvasztás az azonos kulcsú sémába. Pl egy könyvet lehet kölcsönözni: KÖLCSÖN(könyvszám, olvasószám, **kivétel**) kapcsolatsémából KÖNYV (könyvszám, szerző, cím, olvasószám, **kivétel**), OLVASÓ (olvasószám, név, lakcím) lesz. Itt a kapcsolatsémában az olvasószám is kulcs, és felvehetnénk úgy is, hogy az legyen a kulcs. Ez esetben az OLVASÓ-ba kell beolvasztani.
- sok-egy: beolvasztás a „sok”-ba. Tehát a példát követve, ha több könyvet lehet kölcsönözni, akkor a könyvszámnak kell a kulcsnak lennie, és csak a KÖNYV-be olvaszthatjuk be.
- sok-sok: új tábla. Ha a korábbi kölcsönzések is el vannak tárolva, akkor a kulcsban benne van vagy a kivétel, vagy a visszahozás. Ekkor egyik táblába sem lehet beolvasztani, új táblát kell létrehozni. A séma ez lehet: KÖNYV (könyvszám, szerző, cím), OLVASÓ (olvasószám, név, lakcím), KÖLCSÖN (könyvszám, olvasószám, kivétel, visszahozás).

Átalakítás E/K modell → relációs adatmodell:

- egyedhalmaz séma → relációséma
- tulajdonságok → attribútumok
- (szuper)kulcs → (szuper)kulcs
- egyedhalmaz előfordulása → reláció

- egyed $\rightarrow e(A_1)...e(A_n)$ sor
- $R(E_1,...E_p,A_1,...A_q)$ kapcsolati séma (E_i egyedhalmaz, A_j tulajdonság) $\rightarrow R(K_1,...K_p,A_1,...A_q)$ relációséma (K_i az E_i (szuper)kulcsa)

Átnevezhetjük, hogy ne legyen kétszer ugyanaz.
isa esetén a speciális osztályhoz hozzávesszük az általános osztály (szuper)kulcsát. Gyenge entitáshoz a meghatározó kapcsolatok kulcsait vesszük hozzá.
Összevonás akkor lehet, ha az egyikben idegenkulcs van a másikra. Illetve akkor, ha sok-egy kapcsolatnak felel meg az egyik reláció, a másik pedig a sok oldalon álló egyedhalmaz reláció. Pl Ivók(név, cím) és Kedvenc(ivó,sör) összevonható, és kapjuk az Ivó1(név,cím,kedvencSöre) sémát.

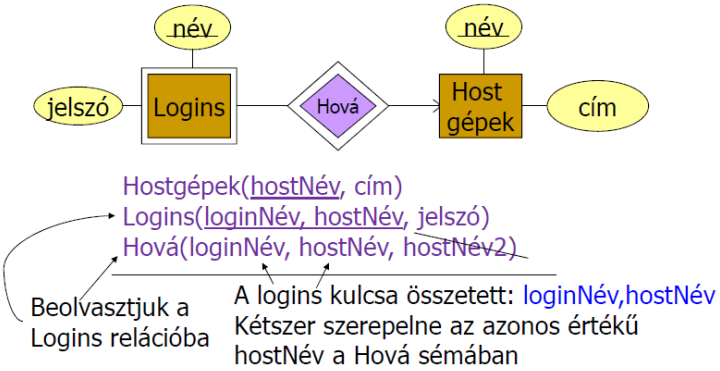


Figure 6: Gyenge egyedhalmaz átírása

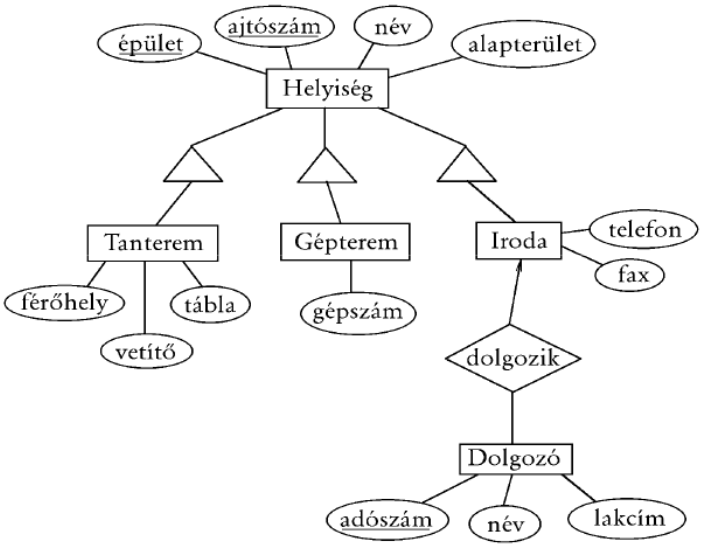


Figure 7: Példa alosztály átírására relációkká

Alosztály átírása:

- Egyed-kapcsolatos: modellben legyen 1 reláció minden alosztályra, de az általánosból csak a kulcsokat vesszük hozzá a saját attribútumokhoz.
  Minden altípushoz külön tábla, egy egyed több táblában is szerepelhet. Főtípus táblájában minden egyed szerepel, plusz annyi altípusában, amennyiben szerepel. Hátrány: több táblában keresés.

(E/K stílusú reprezentálás.)

HELYISÉG (épület, ajtószám, név, alapterület)
TANTEREM (*épület*, *ajtószám*, férőhely, tábla, vetítő)
GÉPTEREM (*épület*, *ajtószám*, gépszám)
IRODA (*épület*, *ajtószám*, telefon, fax)
DOLGOZÓ (adószám, név, lakcím, *épület, ajtószám*)

Figure 8: E/K stílusú

- Nullértékes: 1 reláció van összesen, ha nincs a speciális tulajdonság, akkor NULL-t írunk a helyére. Attribútumok uniója szerepel a táblában. Hátrány, hogy sok NULL van a táblában, típusinformációt is elveszíthetünk (például ha a gépteremnél a gépszám nem ismert, és ezért NULL, akkor a gépterem lényegében az egyéb helyiségek kategóriájába kerül).

(Reprezentálás nullértékekkel)
HELYISÉG (épület, ajtószám, név, alapterület, férőhely, tábla,
vetítő, gépszám, telefon, fax)
DOLGOZÓ (adószám, név, lakcím, *épület, ajtószám*)

Figure 9: NULL értékes

- Objektumorientált: 1 reláció minden alosztályra, összes tulajdonság felsorolva az öröklötből is. Minden altípushoz külön tábla, egy egyed csak 1 táblában szerepel. Hátrányok: kombinált altípushoz új altípus felvétele szükséges, keresés gyakran több táblán keresztül.

(Objektumorientált stílusú reprezentálás)
HELYISÉG (épület, ajtószám, név, alapterület)
TANTEREM (épület, ajtószám, név, alapterület, férőhely,
tábla, vetítő)
GÉPTEREM (épület, ajtószám, név, alapterület, gépszám)
IRODA (épület, ajtószám, név, alapterület, telefon, fax)
DOLGOZÓ (adószám, név, lakcím, *épület, ajtószám*)

Figure 10: Objektumorientált

2 Relációs algebra, SQL

2.1 Relációs algebra

Algebra műveleteket és atomi operandusokat tartalmaz.
Relációs algebra: az atomi operandusokon és az algebrai kifejezéseken végzett műveletek alkalmazásával kapott relációkon műveleteket adunk meg, kifejezéseket építünk (a kifejezés felel meg a kérdés szintaktikai alakjának). Fontos tehát, hogy minden művelet végeredménye reláció, amelyen további műveletek adhatók meg. A relációs algebra atomi operandusai a következők: a relációkhoz tartozó változók; konstansok, amelyek véges relációt fejeznek ki. *Műveletek*:

- Halmazműveletek: Reláció előfordulás véges sok sorból álló halmaz. Így értelmezhetők a szokásos halmazműveletek: az unió (az eredmény halmaz, csak egyszer szerepel egy sor), értelmezhető a metszet és a különbség.
R, S azonos típusú, R $\cup$ S és R $-$ S típusa ugyanez
Az alapműveletekhez az unió és különbség tartozik, metszet műveletet származtatjuk: $R \cap S = R - (R - S)$

- Vetítés (projekció): Adott relációt vetít le az alsó indexben szereplő attribútumokra (attribútumok számát csökkentik). $\Pi_{lista}(R)$, ahol $\{A_{i_1}, ...A_{i_k}\}$ R sémájában levő attribútumok egy részhalmazának felsorolása. Reláció soraiból kiválasztja az attribútumoknak megfelelő $A_{i_1}, ...A_{i_k}$-n előforduló értékeket, ha többször előfordul, akkor a duplikátumokat kiszűrjük (hogy halmazt kapjunk).

- Kiválasztás (szűrés): Kiválasztja az argumentumban szereplő reláció azon sorait, amelyek eleget tesznek az alsó indexben szereplő feltételnek. $\sigma_F(R) = \{t|t \in R$ és t kielégíti az F feltételt$\}$
  A feltétel lehet elemi vagy összetett. Elemi: $A_i \Theta A_j$, $A_i \Theta c$, ahol c konstans, Θ pedig $=, \neq, <, >, \geq, \leq$. Összetett: ha B_1, B_2 feltétel, akkor $\not B_1, B_1 \cap B_2, B_1 \cup B_2$ és a zárójelezések is feltétel.
- Természetes összekapcsolás: Szorzásjellegű műveletek közül csak ez az alapművelet. Nő az attribútumok száma. A közös attribútumnevekre épül. $R \bowtie S$ azon sorpárokat tartalmazza R-ből illetve S-ből, amelyek R és S azonos attribútumain megegyeznek. $R \bowtie S$ típusa a két attribútumhalmaz uniója.
- Átnevezés: Reláció önmagával vett szorzatát ki tudjuk fejezni vele. $\rho_{T(B_1,...B_k)}(R(A_1,...A_k))$, ha az attribútumokat nem akarjuk átnevezni, akkor $\rho_T(R)$

A relációs algebrában a fent felsorolt 6 alapművelet van. Ez egy *minimális készlet*, vagyis bármelyiket elhagyva az a többivel nem fejezhető ki.
Szorzásjellegű műveletnél tekinthetjük a *direkt-szorzatot* alapműveletnek, de a természetes összekapcsolást használják. $R \times S$: az R és S minden sora párban összefűződik, az első tábla minden sorához hozzáfűzzük a második tábla minden sorát. A direkt-szorzat (vagy szorzat, Descartes-szorzat) esetén természetesen nem fontos az attribútumok egyenlősége. A két vagy több reláció azonos nevű attribútumait azonban meg kell különböztetni egymástól (átnevezéssel).
Monotonitás: monoton, nem csökkenő kifejezés esetén bővebb relációra alkalmazva az eredmény is bővebb.
A kivonás kivételnek számít az alapműveletek között, mert nem monoton. Következmény ebből, hogy a kivonás nem fejezhető ki a többi alapművelettel. A kivonás nélkül szokás monoton relációs algebrának is nevezni.
Osztás: maradékos osztás mintájára. R és S sémája $R(A_1, ..., A_n, B_1, ..., B_m)$, illetve $S(B_1, ..., B_m)$, $Q = R \div S$ sémája $Q(A_1, ..., A_n)$. $R \div S$ a legnagyobb (legtöbb sort tartalmazó) reláció, amelyre $(R \div S) \times S \subseteq R$. Relációs algebrában: $R(A, B) \div S(B) = \Pi_{A_1,...,A_n}(R)$–$\Pi_{A_1,...,A_n}(\Pi_{A_1,...,A_n}(R) \times S$–$R)$
Relációs algebrai kifejezés, mint lekérdező nyelv (L-nyelv). Az alapkifejezések az elemi kifejezések, az összetettek pedig a rajtuk végzett alapműveletek.
Kifejezés kiértékelése: összetett kifejezést kívülről befelé haladva átírjuk kiértékelő fává, levelek: elemi kifejezések.

Legyen R, S az R(A, B, C), S(C, D, E) séma feletti reláció
$\Pi_{B,D}\ \sigma_{A = 'c' \text{ and } E = 2}\ (R \bowtie S)$
Ehhez a kiértékelő fa: (kiértékelése alulról felfelé történik)

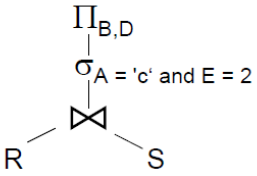


Figure 11: Kifejezésfa

2.1.1 Relációs algebra kiterjesztése

- Ismétlődések megszüntetése (δ), select distinct. Multihalmazból halmazt csinál.
- Összesítő műveletek és csoportosítás (γ_{lista}), group by. Összesítő függvények: sum, count, min, max, avg. Itt a *lista* valamennyi eleme a következők egyike:
  - a reláció egy attribútuma: ez az attribútum egyike a csoportosító attribútumoknak (a GROUP BY után jelenik meg).
  - a reláció egyik attribútumára (ez az összesítési attribútumra) alkalmazott összesítő operátor, ha az összesítés eredményére névvel szeretnénk hivatkozni, akkor nyilat és új nevet használunk.

  R sorait csoportokba osszuk. Egy csoport azokat a sorokat tartalmazza, amelyek a listán szereplő csoportosítási attribútumokhoz tartozó értékei megegyeznek, vagyis ezen attribútumok minden egyes különböző értéke egy csoportot alkot. Minden egyes csoporthoz számoljuk ki a lista összesítési attribútumaira vonatkozó összesítéseket. Az eredmény minden egyes csoportra egy sor:

1. a csoportosítási attribútumok és
2. Az összesítési attribútumra vonatkozó összesítése (az adott csoport összes sorára)

- Vetítési művelet kiterjesztése (Π_{lista}), select kif [as onev]. A *lista* tartalmazhatja:
  - R egy attribútumát,
  - x → y, ahol x, y attribútumnevek, s itt x-et y-ra nevezzük át,
  - E → y, ahol E R attribútumait, konstansokat, aritmetikai operátorokat és karakterlánc operátorokat tartalmazhat, például: A + 5, C || 'nevű emberek'.
- Rendezési művelet (τ_{lista}), order by. $lista = (A_1, ...A_n)$ Először A_1 attribútum szerint rendezzük R sorait. Majd azokat a sorokat, amelyek értéke megegyezik az A_1 attribútumon, A_2 szerint, és így tovább.
  Ez az egyetlen olyan művelet, amelynek az eredménye se nem halmaz, se nem multihalmaz.
- Külső összekapcsolások ($\overset{\circ}{\bowtie}$), [left | right | full] outer join. Ez nem relációs algebrai művelet, ugyanis kilép a modellből. $R \bowtie S$ relációt kiegészítjük az R és S soraival, a hiányzó helyekre NULL értéket írva megőrzi a ,,lógó sorokat".

SQL-ben, és a kiterjesztésben is multihalmazok vannak, vagyis egy sor többször is előfordulhat.
R és S uniójánál n+m előfordulás lesz, metszeténél min[n,m], különbségnél max[0, n-m]. A többi műveletnél nem küszöböljük ki az ismétlődéseket.

2.2 SQL

Fő komponensei:

- Adatleíró nyelv, DDL (Data Definition Language): CREATE, ALTER, DROP
- Adatkezelő nyelv, DML (Data Manipulation Language): INSERT, UPDATE, DELETE, SELECT
  Az SQL elsődlegesen lekérdező nyelv (Query Language): SELECT utasítás (az adatbázisból információhoz jussunk)
- Adatvezérlő nyelv, DCL (Data Control Language): GRANT, REVOKE
- Tranzakció-kezelés: COMMIT, ROLLBACK, SAVEPOINT
- Procedurális kiterjesztések: Oracle PL/SQL (Ada alapján), SQL/PSM (PL/SQL alapján)

Reláció itt tábla, alapvetően 3-féle: TABLE (alaptábla, permanens), VIEW (nézettábla), WITH utasítással (átmeneti munkatábla).
Séma megadása CREATE utasítással, típus SQL konkrét megvalósítása alapján. Kiegészítő lehetőségek is vannak, pl PRIMARY KEY. Csak egyetlen PRIMARY KEY lehet a relációban, viszont UNIQUE több is lehet, PRIMARY KEY egyik attribútuma sem lehet NULL érték egyik sorban sem. Viszont UNIQUE-nak deklarált attribútum lehet NULL értékű, vagyis a táblának lehet olyan sora, ahol a UNIQUE attribútum értéke NULL, vagyis hiányzó érték.

Relációs algebrai kifejezések felírása SELECT-tel:

- SELECT lista FROM táblák szorzata WHERE felt. = $\Pi_{lista}(\sigma_{felt.}$(táblák szorzata))
- halmazműveletek: UNION, EXCEPT/MINUS, INTERSECT. Multihalmazból halmaz lesz. ALL kulcsszóval megmarad a multihalmaz.
- átnevezés: oszlopnév után szóközzel odaírjuk az új nevet.

Kiterjesztett műveletek:

- rendezés: ORDER BY, minden más záradék után következik, csökkenő, növekvő sorrend.
- ismétlődések megszüntetése: select DISTINCT ...

- összesítések (aggregálás): SUM, COUNT, MIN, MAX, AVG a SELECT záradékban. COUNT(*) az eredmény sorainak száma. Ha összesítés is szerepel a lekérdezésben, a SELECT-ben felsorolt attribútumok vagy egy összesítő függvény paramétereként szerepelnek, vagy a GROUP BY attribútumlistájában is megjelennek.
- csoportosítás: GROUP BY.
- csoportok szűrése: HAVING. Csoportokat szűr, nem egy-egy sort. Az alkérdésre nincs megszorítás. Viszont az alkérdésen kívül csak olyan attribútumok szerepelhetnek, amelyek: vagy csoportosító attribútumok, vagy összesített attribútomok. (Azaz ugyanazok a szabályok érvényesek, mint a SELECT záradéknál).

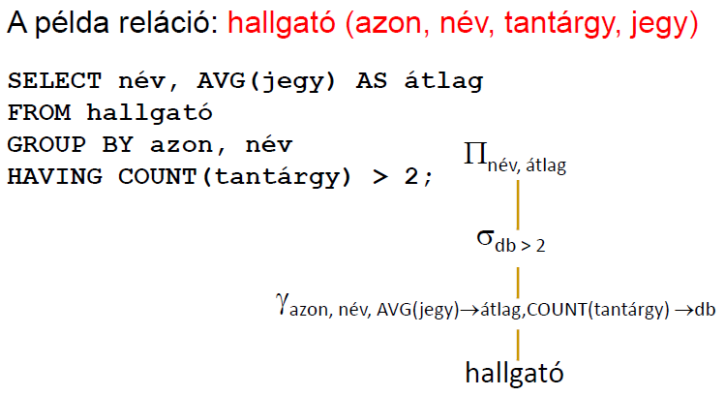


Figure 12: Példa csoportosításra

WHERE záradéknál SQL-es specialitások:

- Átírható relációs algebrába: BETWEEN ... AND ..., IN(értékhalmaz)
- Nem írható át: LIKE karakterláncok összehasonlításánál, IS NULL (ismeretlen vagy nem definiált érték) [emiatt 3-értékű logika SQL-ben: true, false, unknown]

SELECT utasítás több részből áll, a részeket záradékoknak nevezzük. Három alapvető záradék van:

- SELECT lista (3.) – milyen típusú sort szeretnénk az eredményben látni? * jelentése: minden attribútum.
- FROM Relációnév (1.) – a tábla neve vagy relációk (táblák) összekapcsolása, illetve szorzata
- WHERE feltétel (2.) – milyen feltételeknek eleget tevő sorokat kiválasztani?

Több séma összekapcsolása esetén ha egy attribútumnév több sémában is előfordul, akkor kell hozzá a séma is a hivatkozásnál: R.A (R reláció A attribútuma).
Alkérdéseket is használhatunk SQL-ben, ezt zárójelbe tevéssel érjük el.

- FROM záradékban ilyen módon ideiglenes táblát is létrehozhatunk, ekkor többnyire a sorváltozó nevét is meg kell adni hozzá.
- WHERE záradékban az alkérdés eredménye lehet:
  - egy skalárérték, vagyis mintha egy konstans lenne
  - skalár értékekből álló multihalmaz, logikai kifejezésekben használható: EXISTS, [NOT] IN, ANY/ALL (pl x ¿ ANY (alkérdés)).
  - teljes többdimenziós tábla, használható: EXISTS, [NOT] IN

Az alkérdés nem korrelált, ha önállóan kiértékelhető, külső kérdés közben nem változik. Korrelált, ha többször kerül kiértékelésre, alkérdésen kívüli sorváltozóból származó értékadással.

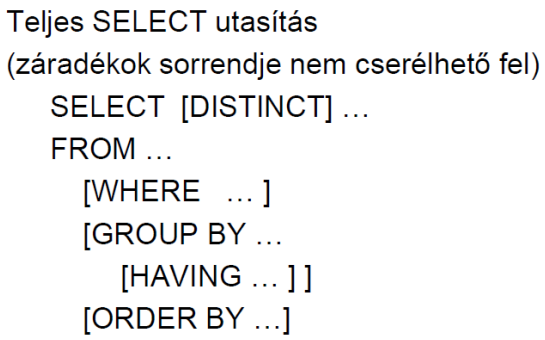


Figure 13: SQL záradékok

Összekapcsolások:

- Descartes-szorzat: R CROSS JOIN S, vagy ,,R,S"
- Természetes összekapcsolás: R NATURAL JOIN S
- Théta-összekapcsolás: R JOIN S ON ¡feltétel¿
- Külső összekapcsolás: R {LEFT | RIGHT | FULL} OUTER JOIN S. LEFT az R lógó sorait őrzi meg, RIGHT az S-ét, FULL az összeset.

2.2.1 DML

A módosító utasítások nem adnak vissza eredményt, mint a lekérdezések, hanem az adatbázis tartalmát változtatják meg. 3-féle módosító utasítás létezik:

- INSERT - sorok beillesztése, beszúrása. Egyetlen sor: INSERT INTO R VALUES (<attr. lista>). A tábla létrehozásánál ha megadtunk default értékeket attribútumoknál, akkor ha nem adunk meg értéket hozzá, akkor a default lesz, egyébként NULL. Több sor: INSERT INTO R (<alkérdés>).
- DELETE – sorok törlése. DELETE FROM R [WHERE <feltétel>].
- UPDATE – sorok komponensei értékeinek módosítása. UPDATE R SET <attribútum értékadások listája> WHERE <sorokra vonatkozó feltétel>

Tranzakciók: Az adatbázisrendszereket általában több felhasználó és folyamat használja egy időben. Tranzakció = olyan folyamat, ami adatbázis lekérdezéseket, módosításokat tartalmaz. Az utasítások egy ,,értelmes egészt" alkotnak.
ACID: atomicity (vagy az összes vagy egyik utasítás sem hajtódik végre), consistency (konzisztencia, a megszorítások megőrződnek), isolation (elkülönítés, felhasználó számára úgy tűnik, hogy egymás után futnak a tranzakciók), durability (tartósság, befejeződött tranzakció módosításai nem vesznek el).
A COMMIT SQL utasítás végrehajtása után a tranzakció véglegesnek tekinthető. A tranzakció módosításai véglegesítődnek. A ROLLBACK SQL utasítás esetén a tranzakció abortál, azaz az összes utasítás visszagörgetésre kerül.
Többféle elkülönítési szint közül lehet választani, hogy az egy időben történő tranzakciók esetén milyen interakciók engedélyezettek.

1. SERIALIZABLE: ACID tulajdonságok teljesülnek rá. Más tranzakciója közbeni állapotot nem láthat a felhasználó.
2. REPEATABLE READ: Hasonló a read-commited megszorításhoz. Itt, ha az adatot újra beolvassuk, akkor amit először láttunk, másodszor is látni fogjuk. De második és az azt követő beolvasások után akár több sort is láthatunk.
3. READ COMMITTED: csak kommitálás utáni adatot láthat, de nem feltétlenül mindig ugyanazt az adatot.
4. READ UNCOMMITTED

2.2.2 DDL

Adatleíró részt is tartalmaz az SQL.

- CREATE: létrehozás
- DROP: eldobja a teljes leírást, és minden, ami ehhez kapcsolódott, elérhetetlen lesz.
- ALTER: leírás módosítása

A *megszorítás* adatelemek közötti kapcsolat, amelyet az AB rendszernek fent kell tartania. Lehet kulcs megszorítás, értékekre, sorokra vonatkozó. Megszorítás módosítása CONSTRAINTS kulcsszóval, önálló megszorítások: CREATE ASSERTION <név> CHECK (<feltétel>). Itt alapvetően az adatbázis bármely módosítása előtt ellenőrizni kell. Egy okos rendszer felismeri, hogy mely változtatások, mely megszorításokat érinthetnek.
Idegen kulcs (R-ről S-re) megszorítást meg kell őrizni, ez kétféleképpen sérülhet: 1. Egy R-be történő beszúrásnál vagy R-ben történő módosításnál S-ben nem szereplő értéket adunk meg. 2. Egy S-beli törlés vagy módosítás ,,lógó" sorokat eredményez R-ben. Védeni többféleképpen lehet: alapértelmezetten nem hajtja végre, továbbgyűrűzésnél igazítjuk a tábla értékeit a változáshoz, set NULL-nál pedig az érintett sorokat NULL-ra állítjuk.
Sor megszorításnál a CREATE utasításon belül a végére tehetünk egy CHECK (<feltétel>) utasítást.

Triggerek olyankor hajtódnak végre, amikor valamilyen megadott esemény történik, mint például sorok beszúrása egy táblába. Az önálló megszorításokkal (assertions) sok mindent le tudunk írni, az ellenőrzésük azonban gondot jelenthet. Az attribútumokra és sorokra vonatkozó megszorítások ellenőrzése egyszerűbb (tudjuk mikor történik), ám ezekkel nem tudunk minden kifejezni. A triggerek esetén a felhasználó mondja meg, hogy egy megszorítás mikor kerüljön ellenőrzésre. A triggereket esetenként ECA szabályoknak (event-condition-action) is nevezik.

- Esemény: általában valamilyen módosítás a adatbázisban, INSERT, DELETE, UPDATE. Mikor?: BEFORE, AFTER, INSTEAD, Mit?: OLD ROW, NEW ROW, FOR EACH ROW, OLD/NEW TABLE, FOR EACH STATEMENT
- WHEN feltétel : bármilyen SQL igaz-hamis-(ismeretlen) feltétel.
- Tevékenység : SQL utasítás, BEGIN..END, PSM tárolt eljárás

A triggerek az eddigi megszorításoktól három dologban térnek el:

- A triggereket a rendszer csak akkor ellenőrzi, ha bizonyos események bekövetkeznek. A megengedett események általában egy adott relációra vonatkozó beszúrás, törlés, módosítás, vagy a tranzakció befejeződése.
- A kiváltó esemény azonnali megakadályozása helyett a trigger először egy feltételt vizsgál meg.
- Ha a trigger feltétele teljesül, akkor a rendszer végrehajtja a triggerhez tartozó tevékenységet. Ez a művelet ezután megakadályozhatja a kiváltó esemény megtörténtét, vagy meg nem történtté teheti azt.

Nézettáblák: A nézettábla olyan reláció, amit tárolt táblák (alaptáblák) és más nézettáblák felhasználásával definiálunk. Kétféle létezik: 1. virtuális = nem tárolódik az adatbázisban; csak a relációt megadó lekérdezés. 2. Materializált = kiszámítódik, majd tárolásra kerül. CREATE [MATERIALIZED] VIEW <név> AS <lekérdezés>. A nézettáblák ugyanúgy kérdezhetők le, mint az alaptáblák. A nézettáblákon keresztül az alaptáblák néhány esetben módosíthatóak is, ha a rendszer a módosításokat át tudja vezetni. Virtuális nézetet nem lehet módosítani, mert nem létezik, de egy INSTEAD OF triggerrel mégis végrehajtathatjuk a változtatásokat.

3 Az SQL procedurális kiterjesztése (PL/SQL vagy PSM)

3.1 PSM

Amikor az SQL utasításokat egy alkalmazás részeként, programban használjuk, a következő problémák léphetnek fel:

- Osztott változók használata: közös változók a nyelv és az SQL utasítás között (ott használható SQL utasításban, ahol kifejezés használható).
- A típuseltérés problémája: Az SQL magját a relációs adatmodell képezi. Tábla – gyűjtemény, sorok multihalmaza, mint adattípus nem fordul elő a magasszintű nyelvekben. A lekérdezés eredménye hogyan használható fel? Három esetet különböztetünk meg attól függően, hogy a SELECT FROM [WHERE stb] lekérdezés eredménye skalárértékkel, egyetlen sorral vagy egy listával (multihalmazzal) tér-e vissza. Utóbbinál kurzor használata, az eredmény soronkénti bejárása. Egyetlen sornál SELECT $e_1, ...e_n$ INTO vált$_1$,...vált$_n$

Háromféleképpen is megközelíthetjük programozási szempontból:

1. SQL kiterjesztése procedurális eszközökkel, az adatbázis séma részeként tárolt kódrészekkel, tárolt modulokkal (pl. PSM = Persistent Stored Modules, Oracle PL/SQL).
2. Beágyazott SQL (sajátos előzetes beágyazás EXEC SQL. - Előfordító alakítja át a befogadó gazdanyelvre/host language, pl. C)
3. Hívásszintű felület: hagyományos nyelvben programozunk, függvénykönyvtárat használunk az adatbázishoz való hozzáféréshez (pl. CLI = call-level interface, JDBC, PHP/DB)

PSM: Persistent Stored Procedures. SQL utasítások és konvencionális elemek (if, while stb) keverékéből áll. Olyan dolgokat is meg lehet csinálni, amit önmagában az SQL-ben nem.

```
CREATE PROCEDURE eljárás-név (
    paraméter-lista )
[DECLARE … deklarációk]
BEGIN
    az eljárás utasításai;
END;
```

```
CREATE FUNCTION függvény-név (
    paraméter-lista) RETURNS értéktípus
[DECLARE … deklarációk]
BEGIN
    utasítások …
END;
```

Figure 14: PSM tárolt eljárás és függvény szerkezete

A tárolt eljárásnak adhatunk paramétereket is, ezek IN, OUT és INOUT módúak lehetnek. Ezek mellé még a paraméter nevét és típusát is meg kell adni, mód-név-típus hármas. Tárolt függvények esetén csak IN módú paramétereink lehetnek.
Néhány fontosabb utasítás:

- Eljárás meghívása a CALL <eljárás neve> (<argumentumlista>) utasítással történik.
- Függvénynél a RETURN utasítás határozza meg a visszatérési érték típusát. Fontos, hogy ezen utasítás hatására nem terminál a függvény.
- Változót deklarálásához ezt használjuk: DECLARE <név> <típus>.
- értékadás: SET <változó> = <kifejezés>.
- BEGIN...END között vannak az utasítások, pontosvesszővel választjuk el őket.
- címke: utasításnak lehet adni, név és kettőspont elé írásával.
- SQL utasítások, DML, MERGE.
- ciklusból kilépés: LEAVE <ciklus címkéje>
- WHILE-DO cikluson kívül REPEAT-UNTIL

Kivételek: 5 jegyű SQLSTATE karakterlánccal jelzi. Nem mindig hiba, hanem lehet a normálistól eltérő viselkedés is. DECLARE <hova menjen> HANDLER FOR <feltétel lista> <utasítás>. A <hova menjen> lehet CONTINUE, EXIT, UNDO.
Kurzorok: Ha a SELECT eredménye több sorral tér vissza, akkor valamilyen ciklussal járjuk be az

eredmény sorait. A kurzor alapvetően egy tuple-változó, ami végigmegy egy lekérdezés eredményének minden tuple-én. DECLARE <sormutató> CURSOR FOR (<lekérdezés>); A kurzor használatához szükség van az OPEN <sormutató>; utasításra, aminek hatására a rendszer a lekérdezést kiértékeli, és hozzáférhető lesz a lekérdezés eredménye, ehhez a bejáráshoz egy ciklust kell indítani, és a sormutató az eredmény első sorára mutat. Amikor végeztünk, a CLOSE <sormutató>; utasítással bezárjuk a kurzort.

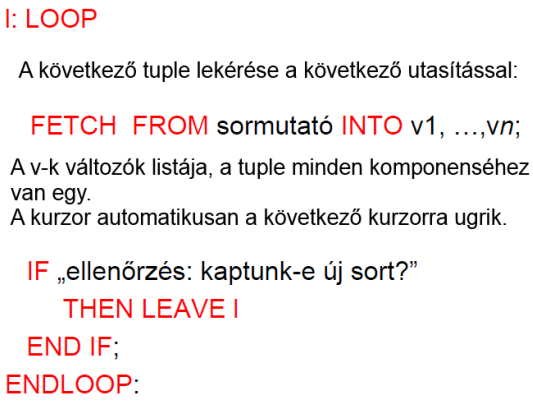


Figure 15: PSM kurzor ciklusának szerkezete

A ciklusból való kilépés trükkös pontja a kurzornak. A megoldás a következő: deklarálunk egy boolean feltételt, ami akkor igaz, ha az SQLSTATE egy meghatározott értéket vesz fel (02000, nem talált következő tuple-t).

3.2 PL/SQL

PL/SQL-ben nem csak tárolni lehet eljárásokat és függvényeket, hanem futtatni is lehet a ,,generic query interface”-ből (sqlplus), mint bármely SQL utasítást. A triggerek a PL/SQL része.
A deklarációs rész külön válik a törzstől és opcionális, és a DECLARE kulcsszó csak egyszer szerepel a rész elején, nincs minden változó előtt. Értékadás := jellel. További különbség a PSM-hez képest, hogy paramétereknél név-mód-típus sorrendben kell megadni, és az IN OUT mód két szóban írandó. Több új típus is van, pl NUMBER lehet INT van REAL. Egy attribútum típusára lehet hivatkozni is: R.x%TYPE. Létezik R%ROWTYPE is, ami egy tuple-t ad vissza. Az x tuple komponensének értékét így kapjuk meg: x.a. ELSEIF (PSM-ben) helyett ELSIF. LEAVE ciklus helyett EXIT WHEN <feltétel>.

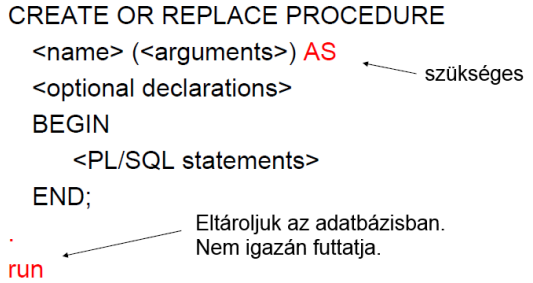


Figure 16: PL/SQL eljárás szerkezete

Kurzor: CURSOR <név> IS <lekérdezés>. A ciklusban a kurzor tuple-jének lekérése: FETCH <kurzor neve> INTO <változó(k)>. PL/SQL-nél a kurzor ciklusát így hagyjuk el: EXIT WHEN <kurzornév>%NOTFOUND.

4 Relációs adatbázis-sémák tervezése, normálformák, dekompozíciók

4.1 Relációs adatbázis-sémák tervezése

Függőségek: funkcionális, többértékű, ezeket a tervezésnél használják (az adatbázisrendszerek nem támogatják, ott megszorítások vannak).
Normalizálás: jó sémákra való felbontás, funkcionális függőségek → (1,2,)3NF, BCNF; többértékű függőségek → 4NF.

4.1.1 Funkcionális függőségek

X → Y az R relációra vonatkozó megszorítás, miszerint ha két sor megegyezik X összes attribútumán, Y attribútumain is meg kell, hogy egyezzenek.
Jelölés: X, Y, Z... attribútum halmazokat; A, B, C... attribútumokat jelöl. A,B,C attribútumhalmaz helyett ABC-t írunk.
Definíció. Legyen R(U) egy relációséma, továbbá X és Y az U attribútumhalmaz részhalmazai. X-től *funkcionálisan függ* Y (jelölésben X → Y), ha bármely R feletti T tábla esetén valahányszor két sor megegyezik X-en, akkor megegyezik Y-on is, $\forall t_1, t_2 \in T$ esetén $(t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y])$. Ez lényegében azt jelenti, hogy az X-beli attribútumok értéke egyértelműen meghatározza az Y-beli attribútumok értékét. Jelölés: R $\models$ X → Y, vagyis R kielégíti X → Y függőséget.
Jobboldalak szétvágása: $X \to A_1A_2...A_n$ akkor és csak akkor teljesül R relációra, ha $X \to A_1$, $X \to A_2$,..., $X \to A_n$ is teljesül R-en. Példa: A→BC ekvivalens A→B és A→C függőségek kettősével. Baloldalak szétvágására nincs szabály, ezért elég nézni, ha a FF-k jobboldalán egyetlen attribútum szerepel.
Példa funkcionális függőségekre: Sörivók(név, cím, kedveltSörök, gyártó, kedvencSör) táblában név → cím kedvencSör (egy ua., mint név → cím és név → kedvencSör), kedveltSörök → gyártó.

Kulcs, szuperkulcs Funkcionális függőség X → Y speciális esetben, ha Y = U, ez a kulcsfüggőség. R(U) relációséma esetén az U attribútumhalmaz egy K részhalmaza akkor és csak akkor szuperkulcs, ha a K → U FF teljesül. A kulcsot tehát a függőség fogalma alapján is lehet definiálni: olyan K attribútumhalmazt nevezünk kulcsnak, amelytől az összes többi attribútum függ, de K-ból bármely attribútumot elhagyva ez már nem teljesül (vagyis minimális szuperkulcs). Példa az előző alapján: {név, kedveltSörök} szuperkulcs, ez a két attr. meghatározza funkcionálisan a maradék attr-kat.

Függőségek implikációja: F implikálja $X \to Y$-t, ha minden olyan táblában, amelyben F összes függősége teljesül, $X \to Y$ is teljesül. Jelölés: $F \models X \to Y$, ha F implikálja $X \to Y$–et.
Legyenek $X_1 \to A_1$, $X_1 \to A_2$,..., $X_1 \to A_n$ adott FF-k, szeretnénk tudni, hogy $Y \to B$ teljesül-e olyan relációkra, amire az előbbi FF-k teljesülnek. Példa: $A \to B$ és $B \to C$ teljesülése esetén $A \to C$ biztosan teljesül. $Y \to B$ teljesülésének ellenőrzésekor vegyünk két sort, amelyek megegyeznek az összes Y-beli attribútumon. Használjuk a megadott FF-ket annak igazolására, hogy az előbbi két sor más attribútumokon is meg kell, hogy egyezzen. Ha B egy ilyen attribútum, akkor $Y \to B$ teljesül. Egyébként az előbbi két sor olyan előfordulást ad majd, ami az összes előírt egyenlőséget teljesíti, viszont $Y \to B$ mégsem teljesül, azaz $Y \to B$ nem következménye a megadott FF-eknek.

Armstrong-axiómák: Legyen R(U) relációséma és $X, Y \subseteq U$, és jelölje XY az X és Y attribútumhalmazok egyesítését. F legyen funkcionális függőségek tetsz. halmaza.

- FD1 (reflexivitás): $Y \subseteq X$ esetén $X \to Y$.
- FD2 (bővíthetőség): $X \to Y$ és tetszőleges Z esetén $XZ \to YZ$.
- FD3 (tranzitivitás): $X \to Y$ és $Y \to Z$ esetén $X \to Z$.

Az Armstrong-axiómarendszer helyes és teljes, azaz minden levezethető függőség implikálódik is, illetve azok a függőségek, amelyeket F implikál azok le is vezethetők F-ből. $F \vdash X \to Y \iff F \models X \to Y$

Levezetés: $X \to Y$ levezethető F-ből, ha van olyan $X_1 \to Y_1$, ..., $X_k \to Y_k$,..., $X \to Y$ véges levezetés, hogy $\forall k$-ra $X_k \to Y_k \in F$ vagy $X_k \to Y_k$ az FD1, FD2, FD3 axiómák alapján kapható a levezetésben előtte szereplő függőségekből. Jelölés: $F \vdash X \to Y$, ha $X \to Y$ levezethető F-ből.
További levezethető szabályok:

- Összevonhatósági szabály: $F \vdash X \to Y$ és $F \vdash X \to Z$ esetén $F \vdash X \to YZ$.
- Pszeudotranzitivitás: $F \vdash X \to Y$ és $F \vdash WY \to Z$ esetén $F \vdash XW \to Z$.
- Szétvághatósági szabály: $F \vdash X \to Y$ és $Z \subseteq Y$ esetén $F \vdash X \to Z$.

Attribútumhalmaz lezártja: Adott R séma és F funkcionális függőségek halmaza mellett, X^+ az összes olyan A attribútum halmaza, amire X → A következik F-ből. (R,F) séma esetén legyen $X \subseteq R$. Definíció: $X^{+(F)} := \{A | F \vdash X \to A\}$ az X attribútumhalmaz lezárása F-re nézve. Lemma: $F \vdash X \to Y \iff Y \subseteq X^+$. Következménye: az implikációs probléma megoldásához elég az X^+-t hatékonyan kiszámolni.
Az implikációt lezárással is el lehet dönteni. Kiindulás: $Y^+ = Y$. Indukció: Olyan FF-ket keresünk, melyeknek a baloldala már benne van Y^+-ban. Ha $X \to A$ ilyen, A-t hozzáadjuk Y^+-hoz.

FF-ek vetítése: Motiváció: „normalizálás", melynek során egy reláció sémát több sémára bonthatunk szét. Példa: ABCD $F = \{AB \to C, C \to D, D \to A\}$. Bontsuk fel ABC és AD-re. Milyen FF-k teljesülnek ABC –n? Nem csak $AB \to C$, de $C \to A$ is! Vetület kiszámítása: Induljunk ki a megadott FF-ekből és keressük meg az összes nem triviális FF-t, ami a megadott FF-ekből következik. (Nem triviális = a jobboldalt nem tartalmazza a bal.) Csak azokkal az FF-kel foglalkozzunk, amelyekben a projektált séma attribútumai szerepelnek. Függőségek vetülete: Adott (R,F), és $R_i \subseteq R$ esetén: $\Pi_{R_i}(F) := \{X \to Y | F \vdash X \to Y, XY \subseteq R_i\}$.

A többértékű függőség (TÉF): az R reláció fölött $X \to\to Y$ teljesül: ha bármely két sorra, amelyek megegyeznek az X minden attribútumán, az Y attribútumaihoz tartozó értékek felcserélhetőek, azaz a keletkező két új sor R-beli lesz. Más szavakkal: X minden értéke esetén az Y-hoz tartozó értékek függetlenek az R-X-Y értékeitől. Definíció: $X, Y \subseteq R, Z := R - XY$ esetén $X \to\to Y$ többértékű függőség. A függőség akkor teljesül egy táblában, ha bizonyos mintájú sorok létezése garantálja más sorok létezését.
Formális definíció: Egy R sémájú r reláció kielégíti az $X \to\to Y$ függőséget, ha $t, s \in r$ és t[X]=s[X] esetén létezik olyan $u, v \in r$, amelyre u[X]=v[X]=t[X]=s[X], u[Y]=t[Y], u[Z]=s[Z], v[Y]=s[Y], v[Z]=t[Z]. Állítás: Elég az u,v közül csak az egyik létezését megkövetelni. Példa: Sörivók(név, cím, tel, kedveltSörök) tábla. A sörivók telefonszámai függetlenek az általuk kedvelt söröktől: név→→tel és név→→kedveltSörök. Így egy-egy sörivó minden telefonszáma minden általa kedvelt sörrel kombinációban áll. Ez a jelenség független a funkcionális függőségektől.
TÉF szabályok:

- Minden FF TÉF. Ha $X \to Y$ és két sor megegyezik X-en, Y-on is megegyezik, emiatt ha ezeket felcseréljük, az eredeti sorokat kapjuk vissza, azaz: $X \to\to Y$.
- Komplementálás : Ha $X \to\to Y$ és Z jelöli az összes többi attribútum halmazát, akkor $X \to\to Z$.
- Nem lehet darabolni.
- Állítás: $X \to\to Y$-ből nem következik, hogy $X \to\to A$, ha $A \in Y$. (A jobb oldalak nem szedhetők szét!)
- Nem tranzitív.

A veszteségmentesség, függőségőrzés definíciójában most F funkcionális függőségi halmaz helyett D függőségi halmaz többértékű függőségeket is tartalmazhat.
Tétel: A $d = (R_1, R_2)$ akkor és csak akkor veszteségmentes dekompozíciója R-nek, ha $D \vdash R_1 \cap R_2 \to\to R_1 - R_2$.

4.2 Normálformák

- *Boyce-Codd normálforma*: R reláció BCNF-ben van, ha minden X → Y nemtriviális FF-re R-ben X szuperkulcs. Nemtriviális: Y nem része X-nek. Szuperkulcs: tartalmaz kulcsot (ő maga is lehet kulcs). Ha van olyan következmény FF F-ben, ami sérti a BCNF-t, akkor egy F-beli FF is sérti. Kiszámítjuk X^+-t: Ha itt nem szerepel az összes attribútum, X nem szuperkulcs.
- Bizonyos FF halmazok esetén a felbontáskor elveszíthetünk függőségeket. *3. normálformában* (3NF) úgy módosul a BCNF feltétel, hogy az előbbi esetben nem kell dekomponálnunk. Egy attribútum elsődleges attribútum (prím), ha legalább egy kulcsnak eleme. X → A megsérti 3NF-t akkor és csak akkor, ha X nem szuperkulcs és A nem prím.

- A *4.normálforma* hasonlít a BCNF-re, azaz minden nem triviális többértékű függőség bal oldala szuperkulcs. A TÉF-ek okozta redundanciát a BCNF nem szünteti meg. A megoldás: a negyedik normálforma. A negyedik normálformában (4NF), amikor dekomponálunk, a TÉF-eket úgy kezeljük, mint az FF-eket, a kulcsok megtalálásánál azonban nem számítanak.
  Egy R reláció 4NF -ben van ha: minden $X \rightarrow\rightarrow Y$ nemtriviális TÉF esetén X szuperkulcs.
  Nemtriviális TÉF: Y nem részhalmaza X-nek, és X és Y együtt nem adják ki az összes attribútumot.
  A szuperkulcs definíciója ugyanaz marad, azaz csak az FF-ektől függ. Definíció: R 4NF-ben van D-re nézve, ha $XY \neq R$, $Y \not\subset X$, és $D \vdash X \rightarrow\rightarrow Y$ esetén $D \vdash X \rightarrow R$.
  Definíció: $d = \{R_1, ..., R_k\}$ dekompozíció 4NF-ben van D-re nézve, ha minden R_i 4NF-ben van $\Pi_{R_i}(D)$-re nézve.
  Állítás: Ha R 4NF-ben van, akkor BCNF-ben is van. Következmény: Nincs mindig függőségőrző és veszteségmentes 4NF dekompozíció.
  Veszteségmentes 4NF dekompozíciót mindig tudunk készíteni a naiv BCNF dekomponáló algoritmushoz hasonlóan.
  Ha R 4NF-ben van, akkor BCNF-ben is.

Tételek: Mindig van VM BCNF-ra és VM FŐ 3NF-ra való felbontás.

4.3 Dekompozíciók

A rosszul tervezettség anomáliákat is eredményez. A jó tervezésnél cél az anomáliák és a redundancia megszüntetése.

név	cím	kedveltSörök	gyártó	kedvencS
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	???	WickedAle	Pete's	???
Spock	Enterprise	Bud	???	Bud

Figure 17: Sörivó(név, cím, kedveltSörök, gyártó, kedvencSör) tábla

- Módosítási anomália: ha Janeway-t Jane-re módosítjuk, megtesszük-e ezt minden sornál? Egy adat egy előfordulását megváltoztatjuk, más előfordulásait azonban nem.
- Törlési anomália: Ha senki sem szereti a Bud sört, azzal töröljük azt az infót is, hogy ki gyártotta. Törléskor olyan adatot is elveszítünk, amit nem szeretnénk.
- Beillesztési anomália: és felvinni ilyen gyártót? Megszorítás, trigger kell, hogy ellenőrizni tudjuk (pl. a kulcsfüggőséget).

Dekomponálás (felbontás): A fenti problémáktól dekomponálással (felbontással) tudunk megszabadulni!
Definíció: $d = \{R_1, ..., R_k\}$ az (R,F) dekompozíciója, ha nem marad ki attribútum, azaz $R_1 \cup ... \cup R_k = R$.(Az adattábla felbontását projekcióval végezzük).
Elvárások:

1. Veszteségmentes legyen a felbontás, vagyis ne legyen információvesztés. A fenti jelölésekkel: ha $r = \Pi_{R_1}(r) \bowtie ... \bowtie \Pi_{R_k}(r)$ teljesül, akkor az előbbi összekapcsolásra azt mondjuk, hogy veszteségmentes. Itt r egy R sémájú relációt jelöl. Chase-teszt a veszteségmentességhez: Készítünk egy felbontást. A felbontás eleminek összekapcsolásából veszünk egy sort. Az algoritmussal bebizonyítjuk, hogy ez a sor az eredeti relációnak is sora.
2. A vetületek legyenek jó tulajdonságúak, és a vetületi függőségi rendszere egyszerű legyen (normálformák: BCNF, 3NF def. később)
3. Függőségek megőrzése a vetületekben (FŐ) A dekompozíciókban érvényes függőségekből következzen az eredeti sémára kirótt összes függőség. Adott (R,F) esetén $d = \{R_1, ..., R_k\}$ függőségőrző dekompozíció akkor és csak akkor, ha minden F-beli függőség levezethető a vetületi függőségekből: minden $x \rightarrow Y \in F$ esetén $\Pi_{R_1}(F) \cup ... \cup \Pi_{R_k}(F) \vdash X \rightarrow Y$.

A függőségőrzésből nem következik a veszteségmentesség és a veszteségmentességből nem következik a függőségőrzés.

Chapter 19

19

Záróvizsga tételsor

19. Adatbázisok optimalizálása és konkurencia kezelése

Ancsin Ádám

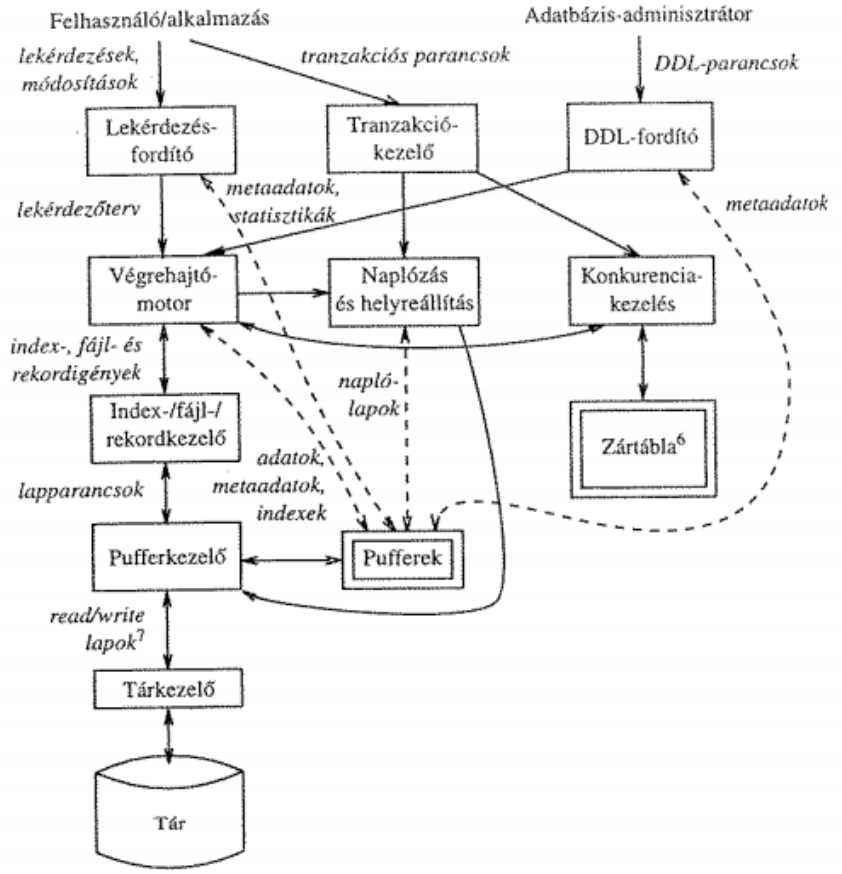
Adatbázisok optimalizálása és konkurencia kezelése

Az adatbázis-kezelő rendszerek feladata, részei. Indexstruktúrák, lekérdezések végrehajtása, optimalizálási stratégiák. Tranzakciók feldolgozása, naplózás és helyreállítás, konkurencia-kezelés.

1 Az adatbázis-kezelő rendszerek feladata, részei

Adatbázis-kezelő rendszer alatt olyan számítógépprogramot értünk, mely megvalósítja nagy tömegű adat biztonságos tárolását, gyors lekérdezhetőségét és módosíthatóságát, tipikusan egyszerre több felhasználó számára.

Az adatbázis-kezelési tevékenységeket két csoportra szokás osztani: adatmanipulációra (lekérdezés), illetve definiálásra (adatszerkezetek kialakítása, módosítása). Az adatok manipulációjára szolgáló nyelveket összefoglalóan Data Manipulation Language-nek (DML), míg a definíciós eszközökkel rendelkező nyelveket Data Definition Language-nek (DDL) szokás nevezni.



ábra 1: Az adatbázis-kezelő rendszer alkotórészei.

1.1 Az egyes alkotórészek rövid jellemzése

Lekérdezésfordító: A lekérdezésfordító elemzi és optimalizálja a lekérdezést, ami alapján elkészíti a lekérdezés-végrehajtási tervet (lekérdezéstervet).

Végrehajtómotor: A végrehajtómotor a lekérdezésfordítótól megkapja a lekérdezéstervet, majd kisebb adatdarabokra (tipikusan rekordokra, egy reláció soraira) vonatkozó kérések sorozatát adja át az erőforrás-kezelőnek.

Erőforrás-kezelő: Az erőforrás-kezelő ismeri a relációkat tartalmazó adatfájlokat, a fájlok rekordjainak formátumát, méretét, valamint az indexfájlokat. Az adatkéréseket az erőforrás-kezelő lefordítja lapokra, amit átad a pufferkezelőnek.

Pufferkezelő: Feladata, hogy a másodlagos adattárolóról (lemez, stb.) az adatok megfelelő részét olvassa be a központi memória puffereibe. A pufferkezelő információkat cserél a tárkezelővel, hogy megkapja az adatokat a lemezről.

Tárkezelő: Adatokat ír-olvas a másodlagos adattárolóról. Előfordulhat, hogy igénybe veszi az oprendszer parancsait is, de sokszor közvetlenül a lemezkezelőhöz intézi a parancsait.

Tranzakciókezelő: A lekérdezéseket és más tevékenységeket tranzakciókba szervezzük. A tranzakciók olyan egységek, amelyeket atomosan és elkülöníthetően kell végrehajtani, valamint a végrehajtásnak tartósnak kell lennie, illetve a tranzakció végrehajtása nem állíthat elő érvénytelen adatbázis-állapotot (azaz konzisztens). Ezeket a követelményeket az ACID mozaikszóval szokás összefoglalni. A tranzakciókezelő hajtatja végre a tranzakciókat és gondoskodik a naplózásról és helyreállításról, valamint a konkurenciakezelésről.

2 Indexstruktúrák

Az indexek keresést gyorsító segédstruktúrák. Több mezőre is lehet indexet készíteni. Nem csak a főfájlt, hanem az indexet is karban kell tartani, ami plusz költséget jelent. Ha a keresési mező egyik indexmezővel sem esik egybe akkor kupac szervezést jelent. Az indexrekordok szerkezete (a,p) , ahol "a" egy érték az indexelt oszlopban, "p" egy blokkmutató, arra a blokkra mutat, amelyben az A=a értékű rekordot tároljuk. Az index mindig rendezett az indexértékek szerint.

2.1 Elsődleges index

Elsődleges index esetén a főfájl is rendezett (az indexmező szerint), így emiatt csak egy elsődleges indexet lehet megadni. Elég a főfájl minden blokkjának legkisebb rekordjához készíteni indexrekordot, így azok száma: $T(I) = B$ (ritka index). Indexrekordból sokkal több fér egy blokkba, mint a főfájl rekordjaiból: $bf(I) >> bf$, azaz az indexfájl sokkal kisebb rendezett fájl, mint a főfájl: $B(I) = \frac{B}{bf(I)} << B = \frac{T}{bf}$.

Keresésnél, mivel az indexfájlban nem szerepel minden érték, ezért csak fedő értéket kereshetünk, a legnagyobb olyan indexértéket, amely a keresett értéknél kisebb vagy egyenlő. Fedő érték keresése az index rendezettsége miatt bináris kereséssel történik: $\log_2(B(I))$. A fedő indexrekordban szereplő blokkmutatónak megfelelő blokkot még be kell olvasni. Így a költség $1+\log_2(B(I)) << \log_2(B)$(rendezett eset). Módosításnál a rendezett fájlba kell beszúrni. Ha az első rekord változik a blokkban, akkor az indexfájlba is be kell szúrni, ami szintén rendezett. A megoldás az, hogy üres helyeket hagyunk a főfájl, és az indexfájl blokkjaiban is. Ezzel a tárméret duplázódhat, de a beszúrás legfeljebb egy főrekord, és egy indexrekord visszaírását jelenti.

2.2 Másodlagos index

Másodlagos index esetén a főfájl rendezetlen (az indexfájl mindig rendezett). Másodlagos indexből többet is meg lehet adni. A főfájl minden rekordjához kell készíteni indexrekordot, így az indexrekordok száma: $T(I) = T$ (sűrű index). Indexrekordból sokkal több fér egy blokkba, mint a főfájl rekordjaiból: $bf(I) >> bf$, azaz az indexfájl sokkal kisebb fájl, mint a főfájl: $B(I) = \frac{T}{bf(I)} << B = \frac{T}{bf}$. Az indexben a keresés az index rendezettsége miatt bináris kereséssel történik: $\log_2(B(I))$. A talált indexrekordban

szereplő blokkmutatónak megfelelő blokkot még be kell olvasni. Így a költség $1 + \log_2(B(I)) << \log_2(B)$ (rendezett eset). Az elsődleges indexnél rosszabb a keresési idő, mert több az indexrekord. A főfájl kupac szervezésű. Rendezett fájlba kell beszúrni. Ha az első rekord változik a blokkban, akkor az indexfájlba is be kell szúrni, ami szintén rendezett. Megoldás: üres helyeket hagyunk a főfájl, és az indexfájl blokkjaiban is. Ezzel a tárméret duplázódhat, de a beszúrás legfeljebb egy főrekord és egy indexrekord visszaírását jelenti.

2.3 Bitmap index

A bitmap indexeket az oszlopok adott értékeihez szokták hozzárendelni, az alábbi módon:

- Ha az oszlopban az $i.$ sor értéke megegyezik az adott értékkel, akkor a bitmap index $i.$ tagja egy 1-es.
- Ha az oszlopban az $i.$ sor értéke viszont nem egyezik meg az adott értékkel, akkor a bitmap index $i.$ tagja egy 0.

Így egy lekérdezésnél csak megfelelően össze kell AND-elni, illetve OR-olni a bitmap indexeket, és az így kapott számsorozatban megkeresni, hol van 1-es. A bináris értékeket szokás szakaszhossz kódolással tömöríteni a hatékonyabb tárolás érdekében.

2.4 Többszintű indexek

Az indexfájl (1. indexszint) is fájl, ráadásul rendezett, így ezt is meg lehet indexelni, elsődleges indexszel. A főfájl lehet rendezett vagy rendezetlen (az indexfájl mindig rendezett) . A $t.$ szintű index: az indexszinteket is indexeljük, összesen t szintig. A $t.$ szinten ($I(t)$) bináris kereséssel keressük meg a fedő indexrekordot. Követjük a mutatót, minden szinten, és végül a főfájlban: $\log_2(B(I(t))) + t$ blokkolvasás. Ha a legfelső szint 1 blokkból áll, akkor $t + 1$ blokkolvasást jelent. Minden szint blokkolási faktora megegyezik, mert egyforma hosszúak az indexrekordok.

A $t.$ szinten 1 blokk: $1 = \frac{B}{bf(I)^t}$. Azaz $t = \log_{bf(I)}(B) < \log_2(B)$, tehát jobb a rendezett fájlszervezésnél. A $log_{bf(I)}(B) < \log_2(B(I))$ is teljesül általában, így az egyszintű indexeknél is gyorsabb.

2.5 B-fa index

Logikailag az index egy rendezett lista. Fizikailag a rendezett sorrendet táblába rendezett mutatók biztosítják. A fa struktúrájú indexek B-fákkal ábrázolhatóak. A B-fák megoldják a bináris fák kiegyenlítetlenségi problémáját, mivel "alulról" töltjük fel őket. A B-fa egy csomópontjához több kulcsérték tartozhat. A mutatók más csomópontokra mutatnak, és így az összes kulcsértékre az adott csomóponton.

Mivel a B-fák kiegyenlítettek (minden ág egyenlő hosszú, vagyis ugyanazon a szinten fejeződik be), kiküszöbölik a változó elérési időket, amik a bináris fákban megfigyelhetőek. Bár a kulcsértékek és a hozzájuk kapcsolódó címek még mindig a fa minden szintjén megtalálhatók, és ennek eredménye: egyenlőtlen elérési utak, és egyenlőtlen elérési idő, valamint komplex fakeresési algoritmus az adatfájl logikailag soros olvasására. Ez kiküszöbölhető, ha nem engedjük meg az adatfájl címek tárolását levélszint felett. Ebből következően: minden elérés ugyanolyan hosszú utat vesz igénybe, aminek egyenlő elérési idő az eredménye, és egy logikailag soros olvasása az adatfájlnak a levélszint elérésével megoldható. Nincs szükség komplex fakeresési algoritmusra.

A B^+-fa egy olyan B-fa, mely legalább 50%-ban telített. A szerkezeten kívül a telítettséget biztosító karbantartó algoritmusokat is beleértjük.

A B^*-fa egy olyan B-fa, mely legalább 66%-ban telített.

2.6 Hasító index

A rekordok edényekbe (bucket) particionálva helyezkednek el, melyekre a h hasítómezőn értelmezett függvény osztja szét őket. Hatékony egyenlőségi keresés, beszúrás, és törlés jellemzi, viszont nem támogatja az intervallumos keresést.

A rekordokat blokkláncokba soroljuk, és a blokklánc utolsó blokkjának első üres helyére tesszük a rekordokat a beérkezés sorrendjében. A blokkláncok száma lehet előre adott (statikus hasítás, ekkor a számot K-val jelöljük), vagy a tárolt adatok alapján változhat (dinamikus hasítás). A besorolás az indexmező értékei alapján történik. Egy $h(x) \in \{1, .., K\}$ hasító függvény értéke mondja meg, hogy melyik

kosárba tartozik a rekord, ha x volt az indexmező értéke a rekordban. A hasító függvény általában maradékos osztáson alapul (például $mod(K)$). Akkor jó egy hasító függvény, ha nagyjából egyforma hosszú blokkláncok keletkeznek, azaz egyenletesen sorolja be a rekordokat. Ekkor a blokklánc $\frac{B}{K}$ blokkból áll.

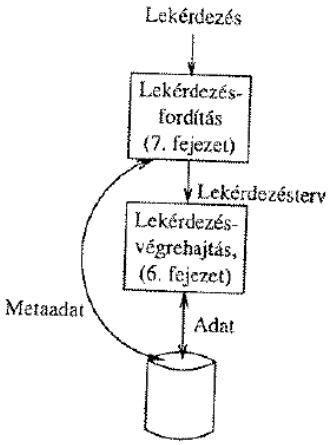
A keresés költsége:

- Ha az indexmező és keresési mező eltér, akkor kupac szervezést jelent.
- Ha az indexmező és keresési mező megegyezik, akkor csak elég a $h(a)$ sorszámú kosarat végignézni, amely $\frac{B}{K}$blokkból álló kupacnak felel meg, azaz $\frac{B}{K}$ legrosszabb esetben. A keresés így K-szorosára gyorsul.

A tárméret B, ha minden blokk nagyjából tele van. Nagy K esetén azonban sok olyan blokklánc lehet, amely egy blokkból fog állni, és a blokkban is csak 1 rekord lesz. Ekkor a keresési idő: 1 blokkbeolvasás, de B helyett T számú blokkban tároljuk az adatokat. Módosításnál $\frac{B}{K}$ blokkból álló kupac szervezésű kosarat kell módosítani.

3 Lekérdezések végrehajtása, optimalizálása

A lekérdezésfeldolgozó egy relációs adatbázis-kezelő komponenseinek azon csoportja, amelyik a felhasználó lekérdezéseit, valamint adatmódosító utasításait lefordítja adatbázis-műveletekre, amiket végre is hajt.

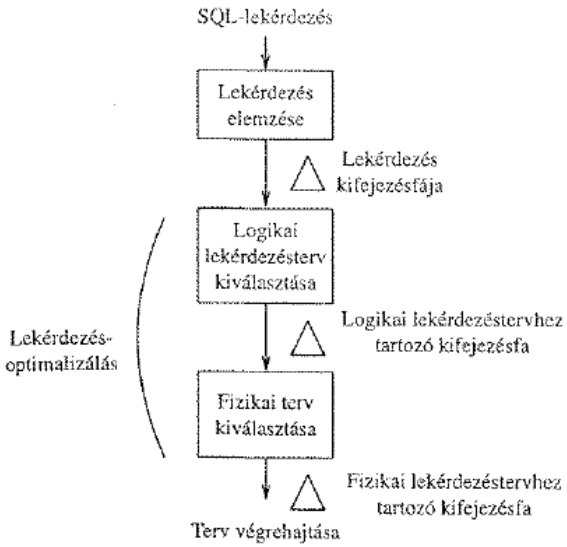


ábra 2: A lekérdezésfeldolgozás lépései.

A lekérdezés végrehajtása tulajdonképpen az adatbázist manipuláló algoritmusok összessége, azonban a végrehajtás előtt szükség van a lekérdezések fordítására.

A lekérdezés-fordítás lépései:

1. Elemzés: egy elemző fát építünk fel, amely a lekérdezést és annak szerkezetét jellemzi
2. Lekérdezésátírás: Az elemző fából egy kezdeti lekérdezéstervet készítünk, amelyet átalakítunk egy ezzel ekvivalens tervvé, aminek végrehajtási ideje várhatóan kisebb lesz. Elkészül a logikai lekérdezésterv, amely relációs algebrai kifejezéseket tartalmaz.
3. A logikai tervet átalakítjuk fizikai tervvé úgy, hogy a logikai terv operátoraihoz kiválasztunk egy algoritmust, valamint meghatározzuk az operátorok végrehajtási sorrendjét. A fizikai terv olyan részleteket is tartalmaz, hogy pl. kell-e rendezni, hogyan férünk hozzá az adatokhoz.



ábra 3: A lekérdezésfordítás lépései.

3.1 Algebrai optimalizálás

A relációs algebrai kifejezéseket minél gyorsabban akarjuk kiszámolni. A kiszámítás költsége arányos a relációs algebrai kifejezés részkifejezéseinek megfelelő relációk tárolási méreteinek összegével. A módszer az, hogy műveleti tulajdonságokon alapuló ekvivalens átalakításokat alkalmazunk, azért, hogy várhatóan kisebb méretű relációk keletkezzenek. Az eljárás heurisztikus, tehát nem az argumentum relációk valódi méretével számol. Az eredmény nem egyértelmű, ugyanis az átalakítások sorrendje nem determinisztikus, így más sorrendben végrehajtva az átalakításokat más végeredményt kaphatunk, de mindegyik általában jobb költségű, mint amiből kiindultunk.

Az optimalizáló algoritmus a következő heurisztikus elveken alapul:

- Minél hamarabb szelektáljunk, hogy a részkifejezések várhatóan kisebb relációk legyenek.
- A szorzás utáni kiválasztásokból próbáljunk természetes összekapcsolásokat képezni, mert az összekapcsolás hatékonyabban kiszámolható, mint a szorzatból történő kiválasztás.
- Vonjuk össze az egymás utáni unáris műveleteket (kiválasztásokat és vetítéseket), és ezekből lehetőleg egy kiválasztást, vagy vetítést, vagy kiválasztás utáni vetítést képezzünk. Így csökken a műveletek száma, és általában a kiválasztás kisebb relációt eredményez, mint a vetítés.
- Keressünk közös részkifejezéseket, amiket így elég csak egyszer kiszámolni a kifejezés kiértékelése során.

3.2 Relációs algebrai műveletek megvalósítása

3.2.1 Kiválasztás

A kiválasztás (σ) lehetséges megvalósításai:

- Lineáris keresés: Olvassunk be minden lapot és keressük az egyezéseket (egyenlőségvizsgálat esetén). Az átlagos költség a lapok száma, ha a mező nem kulcs, illetve a lapok számának fele, ha a mező kulcs.
- Bináris (logaritmikus) keresés: Csak rendezett mező esetén használható.
- Elsődleges index használata.
- Másodlagos index használata.

Összetett kiválasztás: Előfordulhat, hogy több feltétel van, amelyek és/vagy kapcsolatban vannak egymással. Megvalósítása:

- Konjunkciós kiválasztás esetén ($\sigma_{\theta_1 \wedge ... \wedge \theta_n}$):
  - Válasszuk ki a legkisebb költségű σ_{θ_i}-t, és azt végezzük el (lásd fent), majd az eredményt szűrjük a többi feltételre. A költség az egyszerű kiválasztás költsége lesz a kiválasztott σ_{θ_i}-re.
  - Ha mindegyik θ_i mezőjére van indexünk, akkor keressük az indexekben és adjuk vissza a megfelelő sorok rowid-jeit. Végül vegyük ezek metszetét. A költség az indexekben való keresés összköltsége + a rekordok beolvasása.
- Diszjunkciós kiválasztás esetén ($\sigma_{\theta_1 \vee ... \vee \theta_n}$):
  - Lineáris keresés.
  - Ha mindegyik θ_i mezőjére van indexünk, akkor keressük az indexekben és adjuk vissza a megfelelő sorok rowid-jeit. Végül vegyük ezek unióját. A költség az indexekben való keresés összköltsége + a rekordok beolvasása.

3.2.2 Vetítés és halmazműveletek

Vetítésnél és halmazműveleteknél a duplikátumokat ki kell szűrni.

Vetítés (π) megvalósítása:

1. Kezdeti átnézés: eldobjuk a felesleges mezőket.
2. Duplikátumok törlése: ehhez az eredményt rendezzük az összes mező szerint, így a duplikáltak szomszédosak lesznek. Ezeket kell eldobni.

A költség a kezdeti átnézés, a rendezés és a duplikátumok törlésének összköltsége lesz.

3.2.3 Összekapcsolások

Az összekapcsolásokat (join) többféle módon is el lehet végezni:

- **Beágyazott ciklusú összekapcsolás (nested loop join)**: Bármekkora méretű relációra használható, nem szükséges, hogy az egyik reláció elférjen a memóriában. Két fajtája van, a sor és a blokk alapú.
  - **Sor alapú beágyazott ciklusú összekapcsolás**: Legyen R és S a két összekapcsolandó reláció, ezek közül jelölje S a kisebb méretűt, ez lesz a belső reláció, R pedig a külső. Az algoritmus a következő: menjünk végig R (a külső reláció) összes során és R minden egyes soránál menjünk végig a belső reláció, S összes során. Ha az aktuálisan vizsgált 2 sor összekapcsolható, akkor írjuk bele az eredménybe. Jó esetben S belefér a memóriába, ekkor csak egyszer kell beolvasni S-t, majd mindvégig a memóriában tarthatjuk. Ebben az esetben mindkét reláció lapjait egyszer kell beolvasni. Legrosszabb esetben mindkét relációból csak egy-egy lap fér bele a memóriába. Ekkor R minden egyes soránál végig kell olvasni S-t.
  - **Blokk alapú beágyazott ciklusú összekapcsolás**: Az előbbihez képest annyi a különbség, hogy nem soronként megyünk végig a relációkon. Legyen M a memóriába férő lapok száma. Ekkor minden egyes iterációban R-nek M-1 lapját olvassuk be, majd szervezzük ezeket valamilyen keresési struktúrába, ahol a kulcs az R és S közös attribútumai. Ezután menjünk végig laponként S-en, és S memóriában lévő soraiból keressük meg azokat, amelyek összekapcsolhatóak valamely R-ből beolvasott sorral, majd írjuk bele ezeket az eredményrelációba.
- **Összefésüléses rendező összekapcsolás (merge join)**: A relációk az összekapcsolási mezők szerint rendezettek. Egyesítjük a rendezett relációkat: mutatók az első rekordra mindkét relációban. Beolvasunk S-ből egy rekordcsoportot, ahol az összekapcsolási attribútum értéke megegyezik. Beolvasunk rekordokat R-ből és feldolgozzuk. A rendezett relációkat csak egyszer kell végigolvasni, ezért az összekapcsolás költsége: rendezés költsége + a 2 reláció lapjainak száma.
- **Hasításos összekapcsolás (hash join)**: Az összekapcsolási attribútumot használjunk hasítókulcsként, ez alapján particionáljuk R és S sorait hasítással. Ha mindegyik táblából n kosarat szeretnénk, akkor az eredmény: $R_0, R_1, ..R_{n-1}, S_0, S_1, .., S_{n-1}$ kosarak. Ezután az egymáshoz rendelt kosárpárokat összekapcsoljuk blokk alapú beágyazott ciklusú összekapcsolással, hasítófüggvény alapú indexet használva.

Több tábla összekapcsolása: Az összekapcsolások kommutatívak és asszociatívak, ezért az eredmény szempontjából mindegy, hogy milyen sorrendben kapcsoljuk őket össze. A sorrend viszont befolyásolhatja a hatékonyságot, ugyanis rossz választás esetén a köztes eredmények nagy méretűek lesznek.

A legjobb összekapcsolási fa megtalálása n reláció egy halmazához:

- Hogy megtaláljuk a legjobb összekapcsolási fát n reláció egy S halmazához, vegyük az összes lehetséges tervet mely így néz ki: $S_1 \bowtie (S - S_1)$, ahol S_1 az S tetszőleges nem üres részhalmaza.
- Rekurzívan számítsuk ki S részhalmazainak összekapcsolásának költségeit, hogy meghatározzuk minden egyes terv költségét. Válasszuk a legolcsóbbat.
- Mikor bármely részhalmaz terve kiszámításra került, az újbóli kiszámítás helyett tároljuk el és hasznosítsuk újra amikor ismét szükség lesz rá.

4 Tranzakciókezelés

Konzisztens adatbázis: Az adatbázisokra különböző megszorítások adhatóak meg. Az adatbázis konzisztens állapotban van, ha kielégíti az összes ilyen megszorítást. Konzisztens adatbázis egy olyan adatbázis, amely konzisztens állapotban van.

A konzisztencia sérülhet a következő esetekben:

- Tranzakcióhiba: hibásan megírt, rosszul ütemezett, félbehagyott tranzakciók.
- Adatbázis–kezelési hiba: az adatbázis-kezelő valamelyik komponense nem, vagy rosszul hajtja végre a feladatát.
- Hardverhiba: elvész egy adat, vagy megváltozik az értéke.
- Adatmegosztásból származó hiba.

Tranzakció: Konzisztenciát tartó adatbázis-műveletek sorozata. Ezek után mindig feltesszük, hogy ha a T tranzakció indulásakor az adatbázis konzisztens állapotban van, akkor ha T egyedül fut le, az adatbázis konzisztens állapotban lesz a futás végén (közben kialakulhat inkonzisztens állapot).
Helyesség feltétele:

- Ha leáll egy vagy több tranzakció (abort, vagy hiba miatt), akkor is konzisztens adatbázist kapunk.
- Minden egyes tranzakció induláskor konzisztens adatbázist lát.

A tranzakcióktól a következő tulajdonságokat szoktuk elvárni (ACID):

- Atomosság (A, azaz Atomicity): a tranzakció "mindent vagy semmit" jellegű végrehajtása (vagy teljesen végrehajtjuk, vagy egyáltalán nem hajtjuk végre).
- Konzisztencia (C, azaz Consistency): az a feltétel, hogy a tranzakció megőrizze az adatbázis konzisztenciáját, azaz a tranzakció végrehajtása után is teljesüljenek az adatbázisban előírt konzisztenciamegszorítások.
- Elkülönítés (I, azaz Isolation): az a tény, hogy minden tranzakciónak látszólag úgy kell lefutnia, mintha ez alatt az idő alatt semmilyen másik tranzakciót sem hajtanánk végre.
- Tartósság (D, azaz Durability): az a feltétel, hogy ha egyszer egy tranzakció befejeződött, akkor már soha többé nem veszhet el a tranzakciónak az adatbázison kifejtett hatása.

A konzisztenciát mindig adottnak tekintjük. A másik három tulajdonságot viszont az adatbázis-kezelő rendszernek kell biztosítania, de ettől időnként eltekintünk. Feltesszük, hogy az adatbázis adategységekből, elemekből áll. Az adatbáziselem a fizikai adatbázisban tárolt adatok egyfajta funkcionális egysége, amelynek értékét tranzakciókkal lehet elérni (kiolvasni) vagy módosítani (kiírni). Adatbáziselem pl. a reláció, relációsor, lap.

4.1 A tranzakciók alaptevékenységei

A tranzakció és az adatbázis kölcsönhatásának 3 fontos helyszíne van:

- Az adatbázis elemeit tartalmazó lemezblokkok területe.
- A pufferkezelő által használt virtuális vagy valós memóriaterület.
- A tranzakció memóriaterülete.

Ahhoz, hogy a tranzakció egy adatbáziselemet beolvashasson, azt előbb memóriapuffer(ek)be kell hozni, ha még nincs ott. Ezt követően tudja a puffer(ek) tartalmát a tranzakció saját memóriaterületére beolvasni. Az adatbáziselem új értékének kiírás fordítva történik: előbb a tranzakció kialakítja az új értéket saját memóriaterületén, majd ez az érték másolódik át a megfelelő puffer(ek)be.

A pufferek tartalmának lemezre írásáról a pufferkezelő dönt. Vagy azonnal lemezre írja a változásokat, vagy nem.

A naplózási algoritmusokn és más tranzakciókezelő algoritmusok tanulmányozása során különböző jelölésekre lesz szükség, melyekkel a különböző területek közötti adatmozgásokat írhatjuk le. A következő alapműveletek használjuk:

- INPUT(X): Az X adatbáziselemet tartalmazó lemezblokk másolása a pufferbe.
- READ(X,t): Az X adatbáziselem bemásolása a tranzakció t lokális változójába. Ha az X-et tartalmazó blokk még nincs a memóriában, akkor INPUT(X)-et is beleértjük.
- WRITE(X,t): A t lokális változó tartalmaz az X adatbáziselem memóriapufferbeli tartalmába másolódik. Ha az X-et tartalmazó blokk még nincs a pufferben, akkor előbb INPUT(X) is végrehajtódik.
- OUTPUT(X): Az X adatbáziselemet tartalmazó blokk kiírása lemezre.

A továbbiakban feltételezzük, hogy egy adatbáziselem nem nagyobb egy blokknál.

4.2 Naplózás és helyreállítás

Az adatokat meg kell védeni a rendszerhibáktól, ezért szükség van az adatok helyreállíthatóságára. Erre az elsődleges technika a naplózás, amely valamilyen biztonságos módszerrel rögzíti az adatbázisban végrehajtott módosítások történetét.

A napló (log) nem más, mint naplóbejegyzések (log records) sorozata, melyek arról tartalmaznak információt, hogy mit tett egy tranzakció. Rendszerhiba esetén a napló segítségével rekonstruálható, hogy mit tett a tranzakció a hiba fellépéséig.

4.2.1 Naplóbejegyzések

Úgy kell tekintenünk, hogy a napló, mint fájl kizárólag bővítésre van megnyitva. Tranzakció végrehajtásakor a naplókezelőé a feladat, hogy minden fontos eseményt rögzítsen a naplóban.

Az összes naplózási módszer által használt naplóbejegyzések:

- $\langle START\ T \rangle$: Ez a bejegyzés jelzi a T tranzakció végrehajtásának kezdetét.
- $\langle COMMIT\ T \rangle$: A T tranzakció rendben befejeződött, már nem akar további módosításokat végrehajtani.
- $\langle ABORT\ T \rangle$: A T tranzakció abortált, nem tudott sikeresen befejeződni. Az általa tett változtatásokat nem kell a lemezre másolni, vagy ha a lemezre másolódtak, akkor vissza kell állítani.

4.2.2 Semmisségi (undo) naplózás

A semmisségi naplózás lényege, hogy ha nem biztos, hogy egy tranzakció műveletei rendben befejeződtek és minden változtatás lemezre íródott, akkor a tranzakció hatását vissza kell vonni, azaz az adatbázist olyan állapotba kell visszaállítani, mintha a tranzakció el se kezdődött volna.

A semmisségi naplózásnál szükség van még egy fajta naplóbejegyzésre, a módosítási bejegyzésre, amely egy $\langle T, X, v \rangle$ hármas, és azt jelenti, hogy a T tranzakció az X adatbáziselemet módosította, és a

módosítás előtti értéke X-nek v volt.

A semmisségi naplózás szabályai:

- Ha a T tranzakció módosítja az X adatbáziselemet, akkor a $\langle T, X, v\rangle$ naplóbejegyzést az előtt kell a lemezre írni, hogy az új értéket a lemezre írná a rendszer.
- Ha a tranzakció hibamentesen befejeződött, akkor a COMMIT bejegyzést csak azután szabad lemezre írni, hogy a tranzakció által végrehajtott összes módosítás lemezre íródott.

Helyreállítás a semmisségi naplózás használatával

Tegyük fel, hogy rendszerhiba történt. Ekkor előfordulhat, hogy egy tranzakció nem atomosan hajtódott végre, azaz bizonyos módosításai már lemezre íródtak, de mások még nem. Ekkor az adatbázis inkonzisztens állapotba kerülhet. Ezért rendszerhiba esetén gondoskodni kell az adatbázis konzisztenciájának visszaállításáról. Semmisségi naplózás esetén ez a be nem fejeződött tranzakciók által végrehajtott módosítások semmissé tételét jelenti.

Visszaállítás ellenőrzőpont nélkül: A legegyszerűbb módszer. Ekkor a teljes naplót látjuk. Az első feladat a tranzakciók felosztása sikeresen befejezett és befejezetlen tranzakciókra. Egy T tranzakció sikeresen befejeződött, ha van a naplóban $\langle COMMIT\ T\rangle$ bejegyzés. Ekkor T önmagában nem hagyhatta inkonzisztens állapotban az adatbázist. Amennyiben találunk a naplóban $\langle START\ T\rangle$ bejegyzést, de $\langle COMMIT\ T\rangle$ bejegyzést nem, akkor feltételezhetjük, hogy T végrehajtott olyan módosítást az adatbázisban, amely még nem íródott ki lemezre. Ekkor T nem komplett tranzakció, hatását semmissé kell tenni.
Az algoritmus a következő: A helyreállítás-kezelő elkezdi vizsgálni a naplóbejegyzéseket az utolsótól kezdve, visszafelé haladva, közben feljegyzi azokat a T tranzakciókat, melyre $\langle COMMIT\ T\rangle$ vagy $\langle ABORT\ T\rangle$ bejegyzést talált. Visszafelé haladva, amikor $\langle T, X, v\rangle$ naplóbejegyzést lát:

1. Ha T-re találkozott már COMMIT bejegyzéssel, akkor nem tesz semmit.
2. Más esetben T nem teljes vagy abortált. Ekkor a helyreállítás-kezelő az X adatbáziselem értékét v-re változtatja.

A fenti változtatások végrehajtás után minden nem teljes T tranzakcióra $\langle ABORT\ T\rangle$-t ír a napló végére és kiváltja a napló lemezre írását. Ezt követően az adatbázis normál használata folytatódhat.

4.2.3 Ellenőrzőpont-képzés

A helyreállítás elvben a teljes napló átvizsgálását igényelné. Ha undo naplózást használunk, akkor ha egy T tranzakcióra van COMMIT bejegyzés a naplóban, akkor a T tranzakcióra vonatkozó bejegyzések nem szükségesek a helyreállításhoz, viszont nem feltétlenül igaz az, hogy törölhetjük a T tranzakcióra vonatkozó COMMIT előtti bejegyzéseket. A legegyszerűbb megoldás időnként ellenőrzőpontokat készíteni.

Az egyszerű ellenőrzőpont képzése

1. Új tranzakció indítására vonatkozó kérések leállítása.
2. A még aktív tranzakciók befejeződésének és a COMMIT/ABORT bejegyzés naplóba írásának kivárása.
3. A napló lemezre írása.
4. A $\langle CKPT\rangle$ naplóbejegyzés képzése, naplóba írása, majd a napló lemezre írása.
5. Tranzakcióindítási kérések kiszolgálásának újraindítása.

Az ellenőrzőpont előtt végrehajtott tranzakciók befejeződtek, módosításaik lemezre kerültek. Ezért elég az utolsó ellenőrzőpont utáni részét elemezni a naplónak helyreállításnál.

Ellőrzőpont létrehozása a rendszer működése közben

Az egyszerű ellenőrzőpont-képzéssel az a probléma, hogy nem engedi új tranzakciók elindítását, amíg az aktív tranzakciók be nem fejeződnek. Ez viszont még sok időt igénybe vehet, a felhasználó számára pedig leállítottnak tűnik a rendszer, hiszen nem tud új tranzakciót indítani. Ezt nem engedhetjük meg. Egy bonyolultabb módszer azonban lehetővé teszi ellenőrzőpont képzését anélkül, hogy az új tranzakciók indítását fel kellene függeszteni.

E módszer lépései:

1. $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ bejegyzés készítése és a napló lemezre írása. $T_1, ..T_k$ az éppen aktív, befejezetlen tranzakciók.
2. Meg kell várni a $T_1, ..T_k$ tranzakciók befejeződését. Eközben indíthatóak új tranzakciók.
3. Ha az ellenőrzőpont-képzés kezdetén még aktív $T_1, ..T_k$ tranzakciók mindegyike befejeződött, akkor $\langle END\ CKPT\rangle$ naplóbejegyzés készítése és lemezre írása.

Helyreállítás: Visszafelé elemezve megtaláljuk a be nem fejezett tranzakciókat, az ezen tranzakciók által módosított adatbáziselemek tartalmát a régi értékre állítjuk vissza. Két eset fordulhat elő, vagy $\langle END\ CKPT\rangle$, vagy $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ naplóbejegyzéssel találkozunk előbb.

- Ha előbb $\langle END\ CKPT\rangle$ bejegyzéssel találkozunk, akkor az összes be nem fejezett tranzakcióra vonatkozó bejegyzés megtalálható a legközelebbi $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ bejegyzésig. Az ennél korábbiakkal nem kell foglalkoznunk.
- Ha $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ bejegyzéssel találkozunk előbb, akkor a hiba ellenőrzőpont-képzés közben történt. Ekkor a $T_1, ..T_k$ tranzakciók közül a legkorábban elindítottnak a START bejegyzéséig kell visszamenni, ami viszont biztosan az ezt megelőző START CKPT bejegyzés után található.

Általános szabályként, ha END CKPT-ot írunk a lemezre, akkor az azt megelőző START CKPT bejegyzést megelőző bejegyzésekre nincs szükség a helyreállítás szempontjából.

4.2.4 Helyrehozó (redo) naplózás

Redo vs. undo naplózás:

- A helyrehozó naplózás a semmisségi naplózással szemben helyreállításnál figyelmen kívül hagyja a befejezetlen tranzakciókat és befejezi a normálisan befejezettek által végrehajtott változtatásokat.
- Undo naplózás esetén a COMMIT naplóba írása előtt megköveteljük a módosítások lemezre írását. Ezzel szemben redo naplózás esetén csak akkor írjuk lemezre a tranzakció által végrehajtott módosításokat, ha a COMMIT bejegyzés a naplóba íródott és lemezre került.
- Undo naplózásnál a módosított elemek régi értékére van szükség helyreállításnál, redo-nál pedig az újra.

Helyrehozó naplózás szabályai: Itt a $\langle T, X, v\rangle$ bejegyzés jelentse azt, hogy a T tranzakció az X adatbáziselemet értékét v-re változtatta. Annak sorrendjét, hogy az adat- és naplóbejegyzések hogyan kell, hogy lemezre kerüljenek, az alábbi, ún. "írj korábban" naplózási szabály határozza meg: Mielőtt az adatbázis bármely X elemét a lemezen módosítanánk, szükséges, hogy a $\langle T, X, v\rangle$ és $\langle COMMIT\ T\rangle$ naplóbejegyzések lemezre kerüljenek.

Helyreállítás helyrehozó naplózás használatával:
Ha egy T tranzakció esetén nincs $\langle COMMIT\ T\rangle$ bejegyzés a naplóban, akkor tudjuk, hogy T módosításai nem kerültek lemezre, így ezekkel nem kell foglalkozni. Ha viszont T befejeződött, azaz van $\langle COMMIT\ T\rangle$ bejegyzés, akkor vagy lemezre kerültek a módosításai, vagy nem. Ezért meg kell ismételni T módosításait. Szerencsére a naplóbejegyzések az új értékeket tartalmazzák.

Katasztrófa esetén a helyreállítás lépései:

1. Meghatározni azon tranzakciókat, amelyre van COMMIT bejegyzés a naplóban.

2. Elemezni a naplót az elejéről kezdve. Ha $\langle T, X, v\rangle$ bejegyzést találunk, akkor ha T befejezett tranzakció, akkor v értékét kell X-be írni. Ha T befejezetlen, nem teszünk semmit.

3. Ha végigértünk a naplón, akkor minden be nem fejezett T tranzakcióra $\langle ABORT\ T\rangle$ naplóbejegyzést írunk a naplóba és a naplót lemezre írjuk.

Helyrehozó naplózás ellenőrzőpont-képzéssel

Helyrehozó naplózásnál a működés közbeni ellenőrzőpont-képzés a következő lépésekből áll:

1. $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ naplóbejegyzés készítése és lemezre írása, ahol $T_1, ...T_k$ az aktív tranzakciók.

2. Az összes olyan adatbáziselem kiírása lemezre, amelyeket olyan tranzakciók írtak pufferbe, amelyek $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ előtt befejeződtek (COMMIT), de puffereik még nem kerültek lemezre.

3. $\langle END\ CKPT\rangle$ naplóbejegyzés készítése és lemezre írása.

Visszaállítás ellenőrzőponttal kiegészített redo naplózásnál: Két eset fordulhat elő: az utolsó ellenőrzőponttal kapcsolatos naplóbejegyzés vagy START CKPT, vagy END CKPT.

- Tegyük fel, hogy az utolsó ellenőrzőpont-bejegyzés a naplóban END CKPT. Ekkor a $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ előtt befejeződött tranzakciók módosításai már biztosan lemezre kerültek, ezekkel nem kell foglalkoznunk, viszont a T_i-kel és a START CKPT bejegyzés után indított tranzakciókkal foglalkoznunk kell. Ekkor olyan visszaállítás kell tennünk, mint a sima helyrehozó naplózásnál, annyi különbséggel, hogy csak a T_i-ket és a START CKPT után indított tranzakciókat kell figyelembe venni. A keresésnél a legkorábbi $\langle START\ T_i\rangle$ bejegyzésig kell visszamenni, ahol T_i a $T_1, ...T_k$ valamelyike.

- Tegyük fel, hogy az utolsó ellenőrzőpont-bejegyzés a naplóban $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$. Nem lehetünk biztosak benne, hogy az ez előtt befejeződött tranzakciók módosításai lemezre íródtak, ezért az ezt megelőző END CKPT előtti $\langle START\ CKPT(S_1, S_2, ...T_m)\rangle$ kell visszamenni. Vissza kell állítani azoknak a tranzakcióknak a módosításait, amik S_i-k közül valóak, vagy $\langle START\ CKPT(S_1, S_2, ...T_m)\rangle$ után indultak és befejeződtek.

4.2.5 Semmisségi/helyrehozó (undo/redo) naplózás

Undo/redo naplózásnál a módosítást jelző naplóbejegyzések $\langle T, X, v, w\rangle$ alakúak, ami azt jelenti, hogy a T tranzakció az X adatbáziselem értékét v-ről w-re változtatta.

Undo/redo naplózás esetén a következő előírást kell betartani: Mielőtt az adatbázis bármely X elemének értékét módosítanánk a lemezen, a $\langle T, X, v, w\rangle$ naplóbejegyzésnek a lemezre kell kerülnie.

Speciálisan a $\langle COMMIT\ T\rangle$ bejegyzés megelőzheti és követheti is a módosítások lemezre írását.

Helyreállítás undo/redo naplózásnál

Az alapelvek:

1. A legkorábbitól kezdve állítsuk helyre minden befejezett tranzakció hatását.

2. A legutolsótól kezdve tegyük semmissé a be nem fejezett tranzakciók hatását.

Undo/redo naplózás ellenőrzőpont-képzéssel:

1. Írjunk a naplóba $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ bejegyzést, ahol $T_1, ..T_k$ az aktív tranzakciók, majd írjuk lemezre a naplót.

2. Írjuk lemezre azokat a puffereket, amelyek módosított adatbáziselemeket tartalmaznak (piszkos pufferek). Redo naplózással ellentétben itt minden piszkos puffert lemezre írunk, nem csak a befejezettekét.

3. Írjunk $\langle END\ CKPT\rangle$ naplóbejegyzést a naplóba és írjuk ki lemezre.

4.3 Konkurenciakezelés

A tranzakciók közötti egymásra hatás az adatbázis inkonzisztenssé válását okozhatja, még akkor is, amikor a tranzakciók külön-külön megőrzik a konzisztenciát és rendszerhiba sem történt. Ezért valamiképpen szabályoznunk kell, hogy a különböző tranzakciók egyes lépései milyen sorrendben következzenek egymás után. Ezt az ütemező végzi, magát a folyamatot pedig konkurenciavezérlésnek hívjuk.

Az alapfeltevésünk (helyességi elv), hogy ha minden egyes tranzakciót külön hajtunk végre, akkor azok megőrzik a konzisztens adatbázis-állapotot. A gyakorlatban viszont a tranzakciók általában konkurensen futnak, ezért a helyességi elv nem alkalmazható közvetlenül. Így olyan ütemezéseket kell tekintenünk, amelyek biztosítják, hogy ugyanazt az eredményt állítják elő, mintha a tranzakciókat egymás után, egyesével hajtottuk volna végre.

4.3.1 Ütemezések

Ütemezés: Az ütemezés egy vagy több tranzakció által végrehajtott lényeges műveletek időrendben vett sorozata. Az ütemezéseknél csak az írási és olvasási műveletekkel foglalkozunk.

Soros ütemezés: Egy ütemezés soros, ha bármely T és T' tranzakcióra, ha T-nek van olyan művelete, amely megelőzi T' valamely műveletét, akkor T minden művelete megelőzi T' minden műveletét. A soros ütemezést a tranzakciók felsorolásával adjuk meg, pl. (T_1, T_2).

Sorbarendezhetőség: Egy ütemezés sorba rendezhető, ha ugyanolyan hatással van az adatbázis állapotára, mint valamelyik soros ütemezés, függetlenül az adatbázis kezdeti állapotától.

Jelölések:

- $w_i(x)$ azt jelenti, hogy a T_i tranzakció írja az x adatbáziselemet.
- $r_i(x)$ azt jelenti, hogy a T_i tranzakció olvassa az x adatbáziselemet.

Konfliktus: Konfliktus akkor van, ha van olyan egymást követő műveletpár az ütemezésben, amelynek ha a sorrendjét felcseréljük, akkor legalább az egyik tranzakció viselkedése megváltozik. Tegyük fel, hogy T_i és T_j különböző tranzakciók. Ekkor nincs konfliktus, ha a pár:

- $r_i(X)$ és $r_j(Y)$, még akkor sem, ha $X = Y$. Azaz 2 különböző tranzakció által végrehajtott olvasási művelet sosem áll konfliktusban egymással, még akkor sem, ha ugyanarra az adatbáziselemre vonatkoznak.
- $r_i(X)$ és $w_j(Y)$, ha $X \neq Y$
- $w_i(X)$ és $r_j(Y)$, ha $X \neq Y$
- $w_i(X)$ és $w_j(Y)$, ha $X \neq Y$

Konfliktus van, ha:

- Ugyanannak a tranzakciónak bármely két művelete konfliktusban van, hiszen ezek nem cserélhetőek fel.
- Ugyanazt az adatbáziselemet két különböző tranzakció éri el, és ezek közül legalább az egyik írási művelet.

Tehát különböző tranzakciók műveletei nincsenek konfliktusban egymással, azaz felcserélhetőek, hacsak nem

1. ugyanarra az adatbáziselemre vonatkoznak, és
2. legalább az egyik művelet írás

Konfliktusekvivalens ütemezések: Két ütemezés konfliktusekvivalens, ha szomszédos műveletek nem konfliktusos cseréjével egymásba vihetők.

Konfliktus-sorbarendezhető ütemezések: Egy ütemezés konfliktus-sorbarendezhető, ha konfliktusekvivalens valamely soros ütemezéssel. A konfliktus-sorbarendezhetőség elégséges, de nem szükséges

feltétele a sorbarendezhetőségnek. Piaci rendszerekben a konfliktus-sorbarendezhetőséget ellenőrzik.

Megelőzési gráf: Adott a T_1 és T_2 tranzakcióknak, esetleg további tranzakcióknak is, egy S ütemezése. T_1 megelőzi T_2-t S-ben, ha van a T_1-ben olyan A_1 művelet és T_2-ben olyan A_2 művelet, melyekre:

1. A_1 megelőzi A_2-t S-ben,
2. A_1 és A_2 ugyanarra az adatbáziselemre vonatkoznak, és
3. legalább az egyik írási művelet

Ezek pont azok a feltételek, amikor A_1 és A_2 konfliktusban vannak, nem cserélhetőek fel. Ezeket a megelőzéseket megelőzési gráffal szemléltethetjük. A megelőzési gráf csomópontjai S-beli tranzakciók. Ha a tranzakciókat T_i-vel jelöljük, legyen i a T_i-hez tartozó csomópont a gráfban. Az i csomópontból j csomópontba megy irányított él, ha T_i megelőzi T_j-t.

Megelőzési gráf és a konfliktus-sorbarendezhetőség kapcsolata: Egy S ütemezés konfliktus-sorbarendezhető akkor és csak akkor, ha megelőzési gráfja körmentes. Ekkor a megelőzési gráf csúcsainak bármely topologikus rendezése megad egy konfliktus-ekvivalens soros ütemezést.

4.3.2 Zárak

Zárak használatával is elérhető a konfliktus-sorbarendezhetőség. Ha az ütemező zárakat használ, akkor a tranzakcióknak zárakat kell kérniük és feloldaniuk az adatbáziselemek olvasásán és írásán felül. A zárak használatának két értelemben is helyesnek kell lennie:

- Tranzakciók konzisztenciája: A műveletek és a zárak az alábbi elvárások szerint kapcsolódnak egymáshoz:
  1. A tranzakció csak akkor olvashat vagy írhat egy elemet, ha már korábban zárolta azt, és még nem oldotta fel a zárat.
  2. Ha egy tranzakció zárol egy elemet, akkor azt később fel kell szabadítania.
- Az ütemezések jogszerűsége: A zárak értelme feleljen meg a szándék szerinti elvárásnak: nem zárolhatja két tranzakció ugyanazt az elemet, csak úgy, ha az egyik előbb már feloldotta a zárat.

Jelölések:

- $l_i(x)$: A T_i tranzakció zárat kér az x adatbáziselemre.
- $u_i(x)$: A T_i tranzakció az x adatbáziselem zárolását feloldja.

Zárolási ütemező: A zárolási ütemező feladata, hogy akkor és csak akkor engedélyezze a kéréseket, ha az jogszerű ütemezést eredményez. Ebben segít a zártábla, amely minden adatbáziselemhez megadja azt a tranzakciót, feltéve, hogy van ilyen, amelyik éppen zárolja az adott elemet.

Kétfázisú zárolás: Kétfázisú zárolásról (2FZ) beszélünk, ha minden tranzakcióban minden zárolási művelet megelőz minden feloldási műveletet. Azok a tranzakciók, amelyek eleget tesznek 2FZ-nek, 2FZ tranzakcióknak nevezzük. Konzisztens, 2FZ tranzakciók jogszerű ütemezése konfliktus-sorbarendezhető.

Holtpont: Holtpontról beszélünk, ha az ütemező arra kényszerít egy tranzakciót, hogy örökké várjon egy olyan zárra, amelyet egy másik tranzakció tart zárolva. Tipikus példa holtpont kialakulására, ha 2 tranzakció egymás által zárolt elemeket akar zárolni. A kétfázisú zárolás nem tudja megakadályozni holtpontok kialakulását.

		Kért zár	
		S	X
Érvényes zár ebben a módban	S	Igen	Nem
	X	Nem	Nem

ábra 4: Példa holtpontra.

Különböző zármódú zárolási rendszerek

Osztott és kizárolagos zárak: Tetszőleges adatbáziselemet vagy egyszer lehet zárolni kizárólagosan (írásra), vagy többször osztottan (olvasásra). Kizárólagosan zárolt elemet nem lehet osztottan zárolni, illetve osztottan zárolt elemet nem lehet kizárólagosan zárolni. Ha X-et írni akarjuk, akkor X-re kizárólagos zárral kell rendelkeznünk, ha olvasni, akkor elég az osztott is, de kizárólagos zárral is tudjuk olvasni.

Kompatibilitási mátrix: A kompatibilitási mátrix minden egyes zármódhoz rendelkezik egy sorral és egy oszloppal. A sorok egy másik tranzakció által az X elemre már érvényes záraknak felelnek meg, az oszlopok pedig az X-re kért zármódoknak felelnek meg. A használat szabálya: C módú zárat akkor és csak akkor engedélyezhetünk, ha táblázat minden olyan R sorára, amelyre más tranzakció már zárolta X-et R módban, a C oszlopban "Igen" szerepel.

		Kért zár	
		S	*X*
Érvényes zár ebben a módban	*S*	Igen	Nem
	X	Nem	Nem

ábra 5: Osztott és kizárólagos zárak kompatibilitási mátrixa.