22. fejezet

Alapvető algoritmusok

Az adattárolás és -visszakeresés néhány megvalósítása (bináris keresőfa, AVL-fa, 2-3-fa és B-fa, hasítás láncolással és nyílt címzéssel. Rendezési módszerek és hatékonyságuk (buborék, beszúró és maximum-kiválasztó, ill. verseny, kupac, gyors és összefésülő rendezés). Rendezés lineáris időben: edényrendezések.

22.1. Adattárolás és -visszakeresés

22.1.1. Bináris keresőfa

A bináris keresőfa olyan bináris fa, melyben a csúcsok a kulcsok szerint rendezetten helyezkednek el. A bináris fa részletes leírását ld. az adatszerkezetekről szóló, 21. fejezetben.

22.1.2. AVL-fa

A bináris keresőfával, ha a bemeneti adatok „rossz" sorrendben érkeznek, előfordulhat, hogy kiegyensúlyozatlanná válik, azaz néhány hosszú ágon tárolja az adatok nagy részét. Ekkor a keresés hatékonysága erősen lecsökkenhet, szélsőséges esetben nem lesz jobb, mintha egyszerű listában tárolnánk az adatokat.

Az AVL-fa a bináris keresőfák kiegyensúlyozottsági problémáját igyekszik kiküszöbölni azáltal, hogy a beszúrások, illetve törlések közben folyamatosan alacsony szinten tartja a fa kiegyensúlyozatlanságát. Nevét *G. M. Adelson-Velsky*-ről és *E. M. Landis*-ról kapta.

22.1.1. Definíció (AVL-fa invariánsa). *Jelölhe $h(t_i)$ a t_i facsúcs alatti részfa magasságát. Ekkor az AVL-fa megköveteli, hogy a fa minden t_i csúcsára teljesüljön a*

$$|h(bal(t_i)) - h(jobb(t_i))| \leq 1$$

korlát.

22.1.2. Tétel (Az AVL-fa legnagyobb magassága).

- *Az AVL-fa magassága nem nagyobb, mint $1,44 \cdot \log_2 n$, ahol n a tartalmazott csúcsok száma.*
- *Az AVL-fa műveleteinek időigényére is vonatkozik a fenti korlát.*
- *Az AVL-tulajdonság ellenőrzése $\mathcal{O}(\log n)$ idejű.*
- *Beszúrást követően az AVL-tulajdonság helyreállítása mindig konstans idejű.*
- *Törlés után a fa helyreállítása legfeljebb $c \cdot \log_2 n$ műveletigényű.*

Beszúrás utáni helyreállítás

A beszúrás (és a törlés) a bináris keresőfában leírtakhoz hasonlóan történik, és ha ezzel a kiegyensúlyozottság elveszett, helyre kell állítani.

A helyreállítás első lépése, hogy felismerjük, melyik az a legmélyebben fekvő csúcs, melynél az invariáns nem áll fenn (*„hibás csúcs”*).

Fekete István megfelelő anyagában szép ábrákkal illusztrálva megtalálható a beszúrás és a törlés utáni helyreállítás.

Példa: (++,+) szabály.

Legyen a beszúrás előtt a majdani hibás csúcs alatti fák magassága h (pl. bal részfa), illetve $h + 1$ (pl. jobb részfa). Ekkor a hibás csúcs jobb gyereke (pl. a) alatti részfák magassága nyilván h.

Tekintsük azt az esetet, mikor a hibás csúcs jobb oldali részfájába szúrtunk be egy elemet, és az AVL-tulajdonság elromlik, mivel a egyik (mondjuk jobb) részfája $h + 1$ magas lett. Ekkor a következő forgatást tesszük:

Helyezzük a hibás csúcs helyébe a-t, és legyen a bal gyereke a hibás csúcs, melynek bal részfája maradjon meg, jobb részfája pedig legyen a korábbi bal részfája. a jobb részfáját szintén helyben hagyhatjuk.

Ezeket a szabályokat *forgatás*oknak is nevezzük.

22.1.3. 2-3 fa, B-fa

A 2-3 fák és a B-fák a bináris fákhoz hasonló, de nem bináris kereső fastruktúrák.

22.1.3. Definíció (2-3 fa).

- *A fa minden belső csúcsának 2 vagy 3 gyereke van,*
- *a levelek mind azonos szinten vannak,*
- *a belső csúcsokban csak keresést segítő kulcsokat tárolunk, adatok csak a levélszinten vannak,*
- *a kulcsok balról jobbra rendezett sort alkotnak.*

A belső csúcsok segítő kulcsainak kritériuma: minden kulcs a tőle jobbra eső részfa minimális kulcsértéke.

Hatékonyság. A fa mélysége legfeljebb $\log_2 n$, de ennél jóval kisebb is lehet. A beszúrás és a törlés hasonló műveletigényű.

Beszúrás. A beszúráshoz először megkeressük azt a levélpozíciót (a segítő kulcsok segítségével), ahová be kellene szúrni az új elemet. Ha ez alapján a leendő szülő túl sok (4) levelet tartalmazna, akkor *csúcsvágás*t hajtunk végre rajta – a vágás felgyűrűzhet akár a gyökérig is.

Esetleg, ha szükséges, helyreállítjuk a kulcsértékeket a fában felfelé haladva.

Törlés. Ha a törléssel egy csúcsnak túl kevés (1) gyereke lenne, akkor:

- megnézzük, hogy van-e „testvére”, melynek három gyereke van: ha igen, akkor egy megfelelőt átadunk a törölt csúcs helyére, és helyreállítjuk a kulcsokat,
- ha a szülőnek nincs három gyerekes testvére, akkor csúcsösszevonásokat eszközölünk, melyek felgyűrűzhetnek (a gyökérig is).

Különben egyszerűen törlünk, és esetleg helyreállítjuk a kulcsértékeket a fában felfelé haladva.

B-fa

A B-fa a 2-3 fa általánosítása több ágú csúcsokra.

22.1.4. Definíció (B-fa). *Egy B_r-fa olyan, a 2-3 fához hasonló fa, ahol a csúcsok minimális gyerekszáma $\lceil \frac{r}{2} \rceil$, maximális gyerekszáma pedig r.*

A műveletek a 2-3 fával teljesen analóg módon történnek.

Felhasználások. B-fákat és változataikat gyakran használják adatbáziskezelő rendszerekben tárolási struktúraként, az r érték a gyakorlatban 50 és 1000 között mozog.

22.1.4. Hasítótáblák, hasítás

A hasítás szintén az adattárolás és -keresés hatékonyságát növelő struktúrák. Alapjukat a *hasító függvény (hash-függvény)* képezi.

22.1.5. Definíció (Hasító függvény). *A h hasító függvény egy tetszőleges alaphalmazból (például adatrekordok kulcsérték-halmazából) $[0..M-1]$ intervallumba képező függvény, melyre jó választás esetén teljesül, hogy:*

- *az alaphalmaz számossága M-nél jelentősen nagyobb,*
- *a hasító függvény gyorsan kiszámítható és*
- *az alaphalmazt egyenletesen képezi le az intervallumra.*

A hasítótáblázatok két változatát vizsgáljuk: a *láncolt* és a *nyílt címzéses* hasítótáblát.

Hasítás láncolással

Ennél a módszernél az adatok tárolására egy mutatókat tartalmazó, M hosszú tömböt használunk ($t[0..M-1]$). A mutatók (általában fejelem nélküli) láncolt listák kezdőcímeit tartalmazzák.

Beszúrás.

1. A beszúrandó rekord kulcsát leképezzük $[0..M-1]$-re a hasító függvény segítségével ($m := h(k)$).
2. A $t[m]$ listába – rendezett vagy rendezetlen módon – beszúrjuk az elemet.

Ha nem engedünk meg két azonos kulcsú elemet, akkor először meg kell vizsgálnunk, hogy $t[m]$ lista tartalmazza-e már az adott kulcsot, és ekkor hibát kell jeleznünk.

Keresés.

1. A beszúrandó rekord kulcsát leképezzük $[0..M-1]$-re a hasító függvény segítségével ($m := h(k)$).
2. A $t[m]$ listában keressük az elemet a lista szerkezetétől függő módszerrel.

Törlés. Megkeressük, majd a listából töröljük a kérdéses elemet.

Hasítás nyílt címzéssel

Ennél a módszernél a teljes hasító táblázat M méretű, és az adatokat (vagy legalábbis a kulcsokat és az adatok mutatóit) közvetlenül tároljuk egy $t[0..M-1]$ tömbben.

Beszúrás. Ha beszúráskor kulcsütközés lép fel, akkor valamilyen h_i függvénnyel eltoljuk a beszúrás helyét, azaz megpróbáljuk elhelyezni az elemet a $(h(k) + h_1(k) \mathrm{mod} M$ helyen. Ha ez sem sikerül, tovább próbálkozunk ugyanezzel a módszerrel $(h(k) + h_1(k) + h_2(k))$.

Ha betelt a táblázat (ez nyilvántartható külön vagy figyelhetjük a próbák számát), akkor például a táblázat növelésével és a hasító függvény cseréjével készíthetünk nagyobb hasító táblát.

Keresés. Ha a hasító érték alapján nem a keresett kulcsot találjuk meg (azaz másik kulcs van ott, vagy a cella *törölt*, ld. alább), akkor a beszúrásnál használt h_i függvény segítségével próbálkozunk a következő lehetséges helyen.

Ha üres cellát találunk a keresés közben, akkor a kulcs nem található.

Törlés. Törléskor előbb megkeressük a kulcsot, majd a cellát egyszerű törlés helyett speciális *törölt* státusszal kell ellátnunk, hogy a keresés ne akadjon el, ha esetleg egy „próbálkozási lánc" közepén található a törölt elem.

A következőkben három módszer következik a h_i megválasztására.

Lineáris próbálás. Ekkor $h_i \equiv -1$ (lehetne $+1$ is – a -1 választás hatékonysági előnyt jelentett egyes korai számítógép-architektúrákon).

Ennek a módszernek a problémáját *lineáris csomósodás*nak szokás nevezni: ha sok azonos hasító értékű adat érkezik, akkor a beszúrások és keresések meghosszabbodhatnak, mivel minden beszúrt elemen végig kell haladni.

Négyzetes próbálás.

22.1.6. Tétel (A négyzetes próbálás alaptétele). *Ha $M = 4k+3$ alakú és prímszám, akkor a*

$$0^2, 1^2, -1^2, 2^2, -2^2, 3^2, \ldots, \left(\frac{M-1}{2}\right)^2, -\left(\frac{M-1}{2}\right)^2 \pmod{M}$$

sorozat minden értéket előállít a $[0..M-1]$ intervallumon.

A fenti tétel alapján, ha a h_i függvény értékeit a fenti sorozatból vesszük, akkor a táblázat biztosan teljesen kitölthető.

A módszer megszünteti az elsődleges csomósodást, de mivel az elemek fix nyomvonalra kerülnek, másodlagos csomósodás lép fel.

Kettős hash-elés. Ez a módszer kiküszöböli a másodlagos csomósodást is: válasszunk h_i-nek egy újabb hasító függvényt, és ennek értékével lépjünk mindig balra, ameddig szükséges. A teljes kitöltéshez ki kell kötni, hogy $h'(k)$ *és M legyenek relatív prímek.* Választható például a $h'(k) = k \bmod (M-1)+1$-nek.

A nyílt címzés kritikája. Mivel ez a módszer megkötést tesz a hasító tábla méretére, a gyakorlatban ritkábban használják.

Hasító függvény választása

Osztó-módszer. Legyen $h(k) = k \bmod M$! *Knuth* szerint ha M olyan prím, amely nem esik kettő hatvány közelébe, akkor ez a módszer egyenletesen hasít (pl. $M = 701$).

Szorzó-módszer. Válasszuk a függvényt így: $h(k) = \lfloor \{k \cdot A\} \cdot M \} (A \in \mathbb{R})$, ahol $\{x\}$ az x törtrészét jelöli.

Ez a módszer M értékére nem, csak A-ra érzékeny: egy javaslat az aranymetszés arányszáma: $A = \frac{\sqrt{5}-1}{2}$.

Hasító táblák értékelése

A keresőfákkal összehasonlításban mondhatjuk, hogy általános esetben a keresőfák jobbak, mivel van elméleti felső korlát a keresési időre, illetve olyan lehetőségeket kínálnak (pl. minimális elem keresése), melyek hasító táblázattal nem lesznek hatékonyabbak.

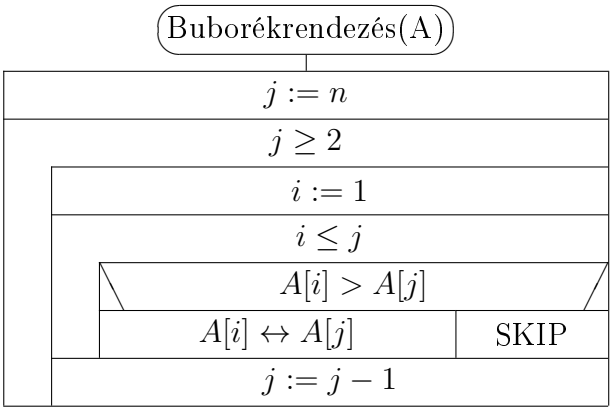
A hasító táblák alkalmazása nem széles körű, de néhány speciális területen gyakori.

22.2. Rendezési módszerek

Ebben a szakaszban különféle adatok rendezésére szolgáló algoritmusokkal foglalkozunk. Mindig feltételezünk az adatok között egy rendezési relációt, illetve az egyszerűség kedvéért úgy kezeljük az adatokat, mintha egy $A[1..n]$ tömbben helyezkednének el, teljesen kitöltve azt.

22.2.1. Buborékrendezés

Az alapötlet szerint a tömbön többször végighaladva a szomszédos elemeket cserélgetjük, ezáltal csökkentjük az inverziószámot. Egy menetben a tömbnek valamelyik végére kikerül az egyik szélsőérték, ezt a következő menetben már elhagyhatjuk.

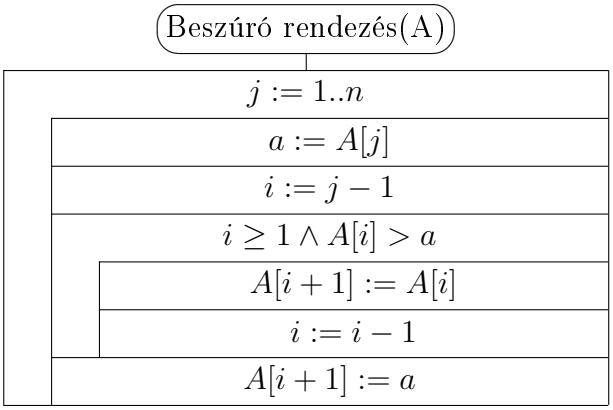


Műveletigény. Az összehasonlítások száma $\Theta(n^2)$ nagyságrendű. A cserék száma:

- legjobb esetben nyilván 0,
- átlagos esetben $\frac{n(n-1)}{4}$,
- legrosszabb esetben $\frac{n(n-1)}{2}$.

22.2.2. Beszúró rendezés

A beszúró rendezés jobbról balra haladva minden elem helyét megkeresi a rendezett sorban, oda beszúrja és a jobbra eső részt eltolja eggyel.



Műveletigény. Műveletigénye általában $\Theta(n^2)$. Megvalósítása láncolt listákra sokkal hatékonyabb.

22.2.3. Maximumkiválasztó rendezés

A módszer minden menetben a tömbből kiválasztja a maximumot, amelyet kicserél a tömb végén lévő elemmel, majd a maradék elemekkel folytatja ugyanezt.

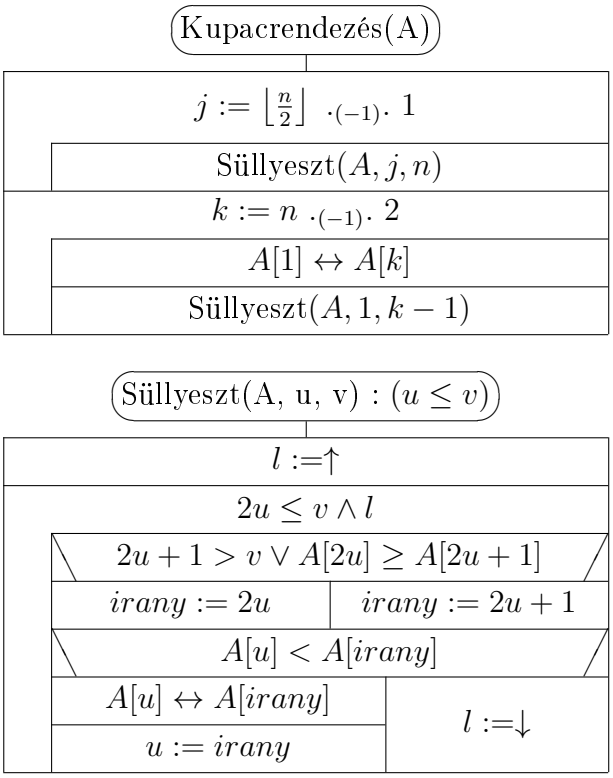
22.2.4. Versenyrendezés

A versenyrendezés a maximumkiválasztó rendezés egy speciális változata, ahol a legnagyobb elem kiválasztása egy kieséses verseny lebonyolításához hasonló fával választja ki az aktuális maximumot.

22.2.5. Kupacrendezés

A kupacrendezés a versenyrendezés helyben rendező változata. Rekurzív algoritmusként is megvalósítható, ekkor a tömböt egy kupac adatszerkezet reprezentációjának tekintve rekurzívan végrehajtjuk a rendezést a kezdőcsúcstól kezdve mindkét gyerekre, majd a két gyerek szülőjét lesüllyesztjük a kupacban a helyére (a *Süllyeszt* eljárás első paramétere egy a tömb, a második a süllyesztendő elem, a harmadik a résztömb felső határa, amelyben süllyeszt).

Iteratív megvalósítása tömbre:

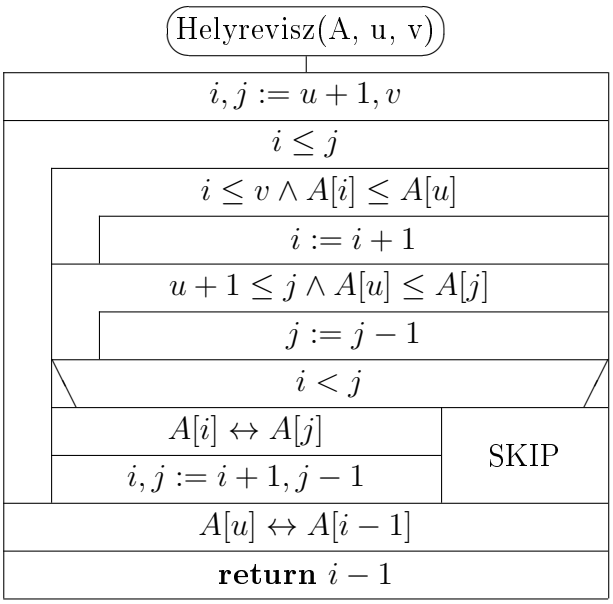


Műveletigény. A kupacrendezés műveletigénye $\Theta(n \log n)$.

22.2.6. Gyorsrendezés (quick sort)

A gyorsrendezés az egy elemet helyrerakó rendezések közé tartozik. Minden lépésben kiválaszt egy elemet, majd a sorozat többi tagját úgy helyezzük el, hogy a kisebbek a kiválasztott elemtől balra, a nagyobbak jobbra legyenek. Ezután a jobb és bal oldal résztömbökön rekurzívan végrehajtjuk a rendezést.

A kiválasztott (általában valamely szélső, legyen ez most a bal) elem helyre vitelét olyan módszerrel végezzük, ahol két index fut szembe egymással a tömbön. Az u a résztömb alsó, v a felső korlátja, a visszatérési érték megadja, hogy hányadik helyre került a kiválasztott elem.



Műveletigény. Legrosszabb esetben (ha vágáskor az egyik résztömb mindig üres) a gyorsrendezés műveletigénye $\Theta(n^2)$. Ez azonban ritkán fordul elő. Átlagos esetben (feltéve, hogy mindenféle elrendezés azonos valószínűséggel fordul elő) a műveletigény $\approx 1,39 \cdot n \log_2 n$.

22.2.7. Összefésülő rendezés (merge sort)

Ez az algoritmus is rekurzív, és a rendezendő sorozat kettéosztásán alapul. Minden lépésben félbevágja a sorozatot, a két részt rendezi, majd a részeket rendezetten összefuttatja. Kiküszöböli a gyorsrendezés bizonytalanságát, bár a helyben rendezés nehézkesebb.

Műveletigény. Legrosszabb esetben az összehasonlítások száma $((n-1) \log_2 n)$.

22.2.8. Lineáris idejű rendezések

A lineáris idejű rendezések vagy *edényrendezések* olyan algoritmusok, melyek a rendezendő adatok speciális szerkezetét használják ki a hatékonyság növeléséhez. Több változatuk ismert:

Leszámoló rendezés

A leszámoló rendezés esetén a rendezési kulcsok osztatlanok, és feltesszük, hogy egy viszonylag szűk $[1..k]$ intervallumba esnek.

Ekkor kiszámíthatjuk az egyes értékek gyakoriságát (esetleg eloszlásfüggvényét is), melyből a tömb rendezett változata könnyen generálható.

A leszámoló rendezés kétszer halad végig a tömbön, műveletigénye tehát $2n$.

Edényrendezés (bucket sort)

Itt a feladatunk n darab $[0, 1)$-beli, egyenletesen elosztott szám rendezése.

Itt lényegében hasító táblába rendezzük az adatokat (mondjuk úgy, hogy tizedenként osztjuk a kulcsértékeket a hasítótáblába – fontos, hogy a hasítás során a rendezettség megmaradjon). A hasító tábla listáiban beszúró rendezést használunk (ez láncolt listák építésére lineáris idejű). Végül a tábla egyes listáit egymás után láncolva kapjuk a rendezett sorozatot.

Többmezős kulcsok rendezése előre

Feltételezzük, hogy a rendezendő adatok kulcsa több mezőből áll, melyek egy véges intervallumból veszik értékeiket.

A kulcsmezők alapján fát építünk, amelynek minden i. szintjén az i. kulcsmező összes lehetséges értéke jelenik meg minden előző szintbeli csúcs gyermekeként, rendezett sorrendben. A levelekre helyezzük az adatokat úgy, hogy a fa csúcsáig vezető úton lévő kulcsmező-értékek adják ki az elem kulcsát. Ezután a fa frontját összeolvasva kapjuk a rendezett adatsort.

Az eljárást ilyen formában nem alkalmazzák a fa túl nagy mérete miatt, de a *Radix-rendezés* az alapötletet felhasználja bináris kulcsra.

Többmezős kulcsok rendezése visszafelé

Az elv az edényrendezéshez hasonló: kulcsmezőnként hátulról előre haladva rendezzük az adatokat a mező értékkészlete szerinti edényekbe, majd fűzzük össze a listát a kulcsmező szerint és folytassuk ezen a listán a rendezést a következő kulccsal.