ELTE IK - Programtervező Informatikus BSc

Záróvizsga tételek

15. Adatszerkezetek és adattípusok

Adatszerkezetek és adattípusok

Tömb, verem, sor, láncolt listák; bináris fa, általános fa, bejárások, ábrázolások; bináris kupac, prioritásos sor; bináris kereső fa és műveletei, AVL fa, B+ fa; hasító táblák, hasító függvények, kulcsütközés és feloldásai: láncolással, nyílt címzéssel, próbasorozat; gráfok ábrázolásai.

1 Egyszerű adattípusok

1.1 Adattípus

Adatszerkezet: ∼ 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.

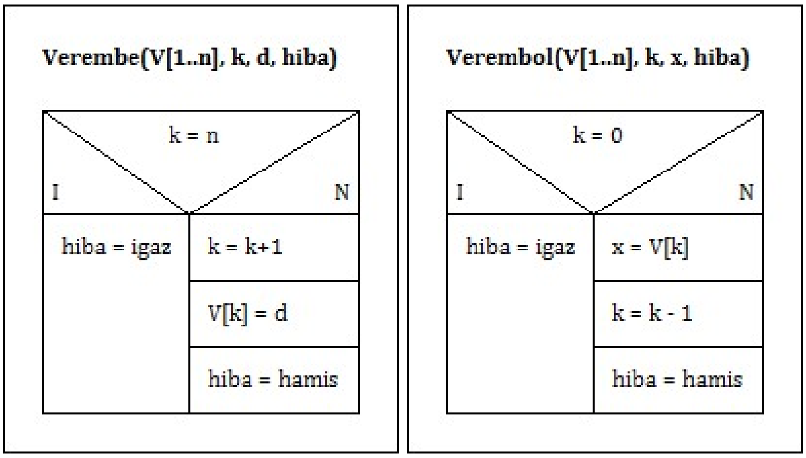


Figure 1: Verem műveletei

- *Sor*: Egyszerű, elsőbbségi és kétvégű. A prioritásos sornál az elemekhez tartozik egy érték, ami alapján rendezhetjük őket.

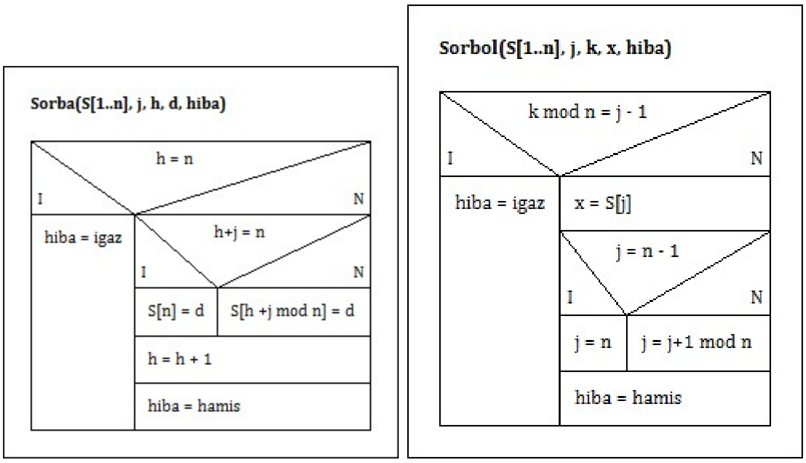


Figure 2: Sor műveletei

- *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.

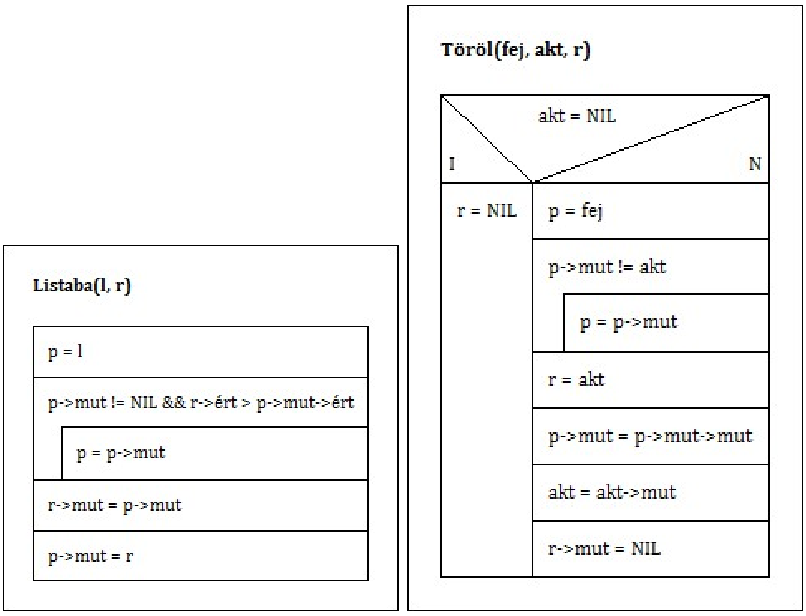


Figure 3: Lista műveletei

2 Fák és bejárásaik, ábrázolásaik

2.1 Bináris fa

A fákat nagy méretű adathalmazok és multihalmazok ábrázolására, de egyéb adatreprezentációs célokra is gyakran használjuk. A (bináris) fák esetében minden adatelemnek vagy szokásos nevén csúcsnak legfeljebb kettő rákövetkezője van: egy bal és/vagy egy jobb rákövetkezője.

Fogalmak:

- *gyerek*: a csúcs bal vagy jobb rákövetkezője
- *szülő*: a gyerekek csúcsa

- *testvérek*: azonos csúcs gyerekei
- *levél*: gyerek nélküli szülő
- *gyökércsúcs*: nincs szülője
- *belső csúcs*: nem-levél csúcs
- *leszármazottak*: egy csúcs gyerekei és annak leszármazottai
- *ősök*: egy csúcs szülője és annak ősei

2.2 Általános fa

A bináris fa fogalma általánosítható. Ha a fában egy tetszőleges csúcsnak legfeljebb r rákövetkezője van, r-áris fáról beszélünk. Egy csúcs gyerekeit és a hozzájuk tartozó részfákat ilyenkor [0..r)-beli szelektorokkal szokás sorszámozni. Ha egy csúcsnak nincs i-edik gyereke (i ϵ [0..r)), akkor az i-edik részfa üres. Így tehát a bináris fa és a 2-áris fa lényegében ugyanazt jelenti, azzal, hogy itt a left $\sim$ 0 és a right $\sim$ 1 szelektor-megfeleltetést alkalmazzuk.

A fa szintjeit a következők képpen határozzuk meg. A gyökér van a nulladik szinten. Az i-edik szintű csúcsok gyerekeit az (i + 1)-edik szinten találjuk. A fa magassága egyenlő a legmélyebben fekvő levelei szintszámával. Az üres fa magassága h($\oslash$) = -1.

Az itt tárgyalt fákat gyökeres fáknak is nevezik, mert tekinthetők olyan irányított gráfoknak, amiknek az élei a gyökércsúcstól a levelek felé vannak irányítva, a gyökérből minden csúcs pontosan egy úton érhető el.

2.3 Bejárásaik

A fákkal dolgozó programok gyakran kapcsolódnak a négy klasszikus bejárás némelyikéhez, amelyek adott sorrend szerint bejárják a fa csúcsait, és minden csúcsra ugyanazt a műveletet hívják meg, amivel kapcsolatban megköveteljük, hogy futási ideje Θ (1) legyen (ami ettől még persze összetett művelet is lehet). A *f csúcs feldolgozása lehet például f $\rightarrow$ key kiíratása. Üres fára mindegyik bejárás az üres program. *Nemüres r-áris fákra*

- *Preorder*: először a fa gyökerét dolgozza fel, majd sorban bejárja a 0..r - 1-edik részfákat;

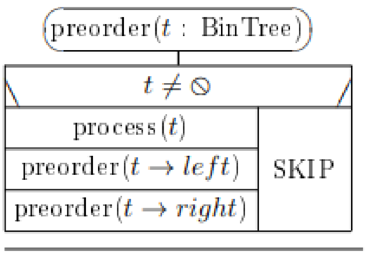


Figure 4: Preorder bejárás

- *Inorder*: először bejárja a nulladik részfát, ezután a fa gyökerét dolgozza fel, majd sorban bejárja az 1..r - 1-edik részfákat;

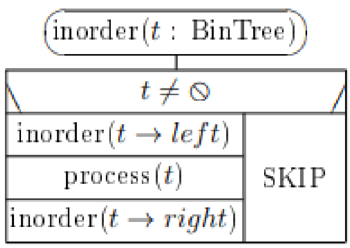


Figure 5: Inorder bejárás

- *Postorder*: előbb sorban bejárja a 0..r - 1-edik részfákat, és a fa gyökerét csak a részfák után dolgozza fel

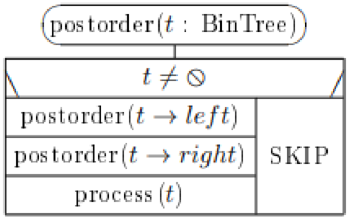


Figure 6: Postorder bejárás

- *LevelOrder*: a csúcsokat a gyökértől kezdve szintenként, minden szintet balról jobbra bejárva dolgozza fel.

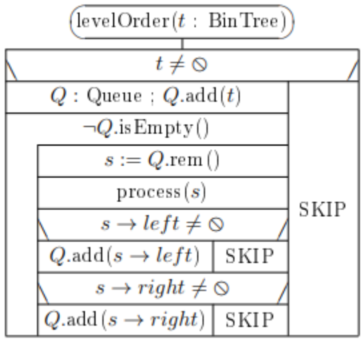


Figure 7: LevelOrder bejárás

Az első három bejárás tehát nagyon hasonlít egymásra. Nevük megsúgja, hogy a gyökércsúcsot a részfákhoz képest mikor dolgozzák fel.

2.4 Ábrázolásaik

2.4.1 Grafikus

A legtermészetesebb és az egyik leggyakrabban használt a bináris fák láncolt ábrázolása:

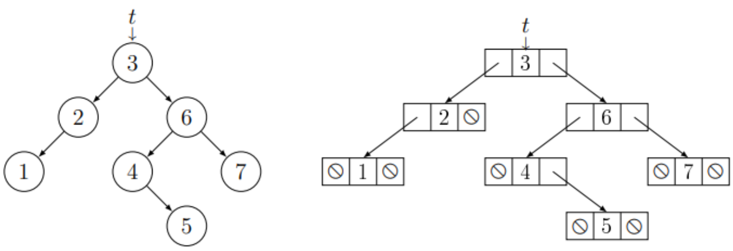


Figure 8: Ugyanaz a bináris fa grafikus és láncolt ábrázolással.

2.4.2 Absztrakt

Az üres fa reprezentációja a $\oslash$ pointer, jelölése tehát ugyanaz, mint az absztrakt fáknál. A bináris fa csúcsait pl. az alábbi osztály objektumaiként ábrázolhatjuk, ahol a BinTree absztrakt típus reprezentációja egyszerűen a Node*.

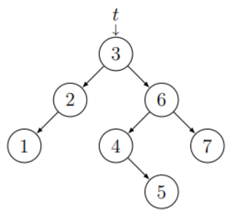
Node
+ *key* : $\mathcal{T}$ // $\mathcal{T}$ valamilyen ismert típus
+ *left*, *right* : Node*
+ Node() { *left* := *right* := ⦸ } // egycsúcsú fát képez belőle
+ Node(x : $\mathcal{T}$) { *left* := *right* := ⦸ ; *key* := x }

Néha hasznos, ha a csúcsokban van egy parent szülő pointer is, mert a fában így felfelé is tudunk haladni.

Node3
+ *key* : $\mathcal{T}$ // $\mathcal{T}$ valamilyen ismert típus
+ *left*, *right*, *parent* : Node3*
+ Node3(p:Node3*) { *left* := *right* := ⦸ ; *parent* := p }
+ Node3(x : $\mathcal{T}$, p:Node3*) { *left* := *right* := ⦸ ; *parent* := p ; *key* := x }

2.4.3 Zárójelezett

Tetszőleges nemüres bináris fa zárójeles, azaz szöveges alakja: (balRészFa Gyökér jobbRészFa) Az üres fát az üres string reprezentálja. A könnyebb olvashatóság kedvéért többféle zárójelpárt is használhatunk. A zárójeles ábrázolás lexikai elemei: nyitózárójel, csukó zárójel és a csúcsok címkéi.



A balra látható t bináris fa

- egyszerű zárójeles alakja:
  (((1) 2) 3 ((4 (5)) 6 (7)))
- elegáns zárójeles alakja:
  { [(1) 2] 3 [(4 ⟨5⟩) 6 (7)] }

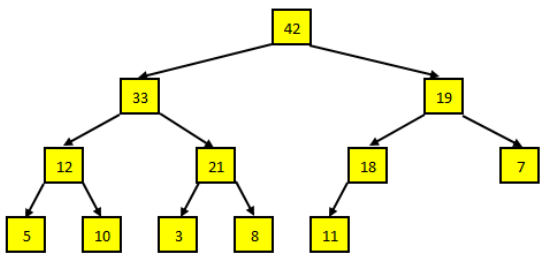
Figure 9: Ugyanaz a bináris fa grafikus és szöveges ábrázolással.

3 Kupac és sor

A bináris kupac egy konkrét adatszerkezet, amelyet a prioritásos sor implementálására használnak. A prioritásos sor általánosabb fogalom, amely többféle adatszerkezetet vagy algoritmust foglalhat magában a prioritás alapján történő rendezés és az elemek elérése szempontjából.

3.1 Bináris kupac

Egy teljes bináris fa, amelyben az elemek prioritása a csúcsokban található kulcsok alapján van meghatározva. A bináris kupac hatékonyan támogatja a prioritásos sor műveleteket, például az elem beszúrását és eltávolítását a legmagasabb prioritással.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	42	33	19	12	21	18	7	5	10	3	8	11			

Figure 10: Példa

3.2 Metódusai

3.2.1 Hozzáadás

Új elemet hozzáadunk a kupachoz a levélszintre (a legközelebbi szabad pozícióba). Ezután a kupacban felfelé haladva összehasonlítjuk az új elemet az ő szülőjével. Ha az új elem prioritása nagyobb, akkor felcseréljük az elemeket, és folytatjuk a felfelé mozgást a egyel magasabb szintre. Ezt addig ismételjük, amíg az új elem prioritása nem megfelelő a bináris kupac tulajdonságainak.

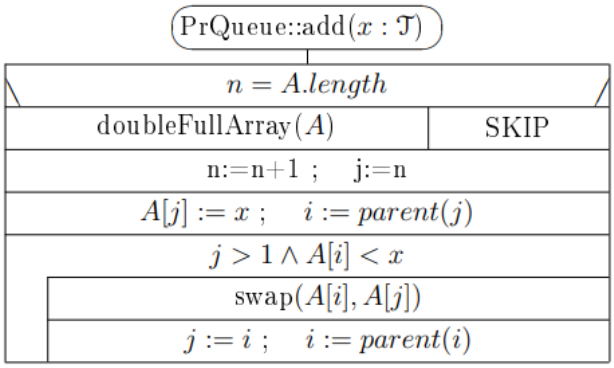


Figure 11: Az add(x) algoritmusa

3.2.2 Törlés

A legmagasabb prioritással rendelkező elem mindig a kupac gyökerén található. Eltávolítjuk(remMax) ezt az elemet, amely a legfelső elem a kupacban. Ilyenkor a kupac üres lesz, vagy helyettesíteni kell a gyökérlemet az aljában található legutolsó elemmel. Ha a kupac nem üres, a helyettesítő elemet lefelé mozgatjuk a kupacban, hogy helyreállítsuk a kupac tulajdonságait. Az elemet összehasonlítjuk a gyermekével, majd a kisebb prioritású gyermekkel cseréljük(sink), ha az nagyobb. Ezt a folyamatot addig ismételjük, amíg az elem megfelelő helyre nem kerül a kupacban.

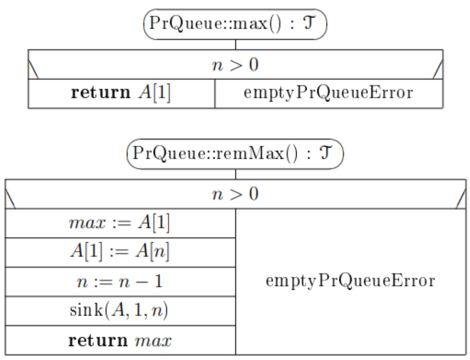


Figure 12: A remMax() algoritmusa

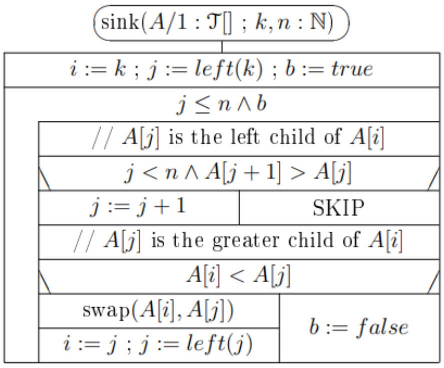


Figure 13: A sink(x) algoritmusa

4 Különleges fák

4.1 Bináris kereső fa

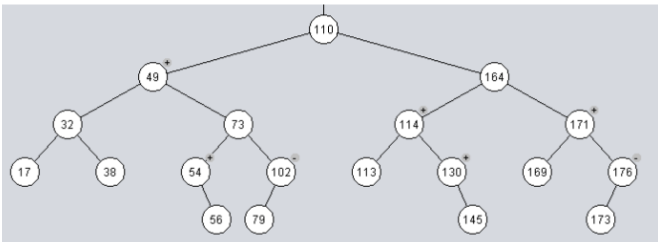


Figure 14: Példa bináris kereső fára

A bináris kereső fa egy olyan adatszerkezet, amelyet a rendezett adatok hatékony tárolására és lekérdezésére használnak. *Tulajdonságai*

- *Rendezettség*: Minden csúcsnak van egy kulcsa, amelyek alapján eldönthető, hogy a bal részfában (kisebb mint a csúcs) vagy a jobb részfában (nagyobb mint a csúcs) helyezkedik el.
- *Gyors keresés*: Lehetővé teszi a gyors keresést a rendezettségnek köszönhetően. Tehát, ha egy elemet keresünk és a csúcstól indulunk egyből tudjuk, hogy jobbra vagy balra van a keresett elem.

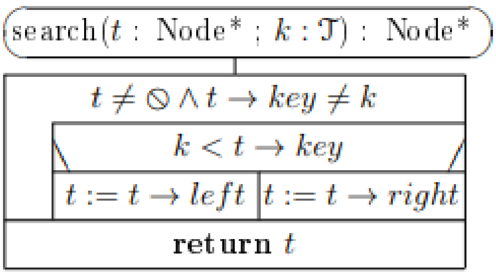


Figure 15: Keresés bináris kereső fában

- *Beszúrás és törlés*: Beszúrásnál az új elemet a megfelelő helyre illesztjük, a rendezettséget figyelembe véve. Törlésnél pedig a struktúrát kell megváltoztatni, hogy a rendezettség megmaradjon.

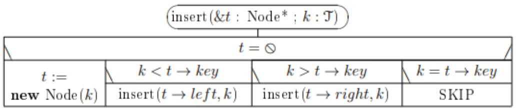


Figure 16: Beszúrás bináris kereső fába

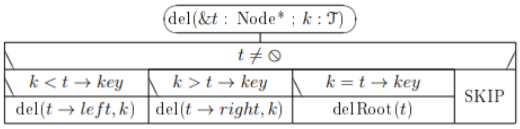


Figure 17: Törlés bináris kereső fából

- *In-order bejárás*: Ezt a bejárást használva, megkapjuk a bináris kereső fa elemeit rendezett sorrendben.

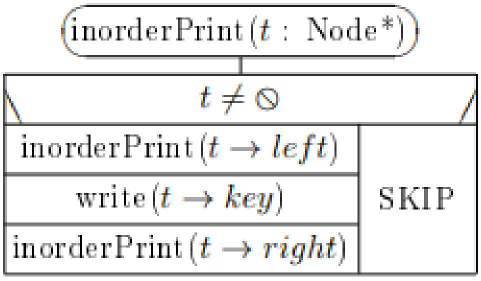


Figure 18: In-order bejárás bináris kereső fában

4.2 AVL fa

Az AVL fa egy speciális bináris kereső fa. Célja az, hogy fenntartsa az egyensúlyt a fa minden csomópontjában, hogy a keresési, beszúrási és törlési műveletek hatékonyan végezhetők legyenek. Tehát a bináris kereső fa tulajdonságain felül van még kettő: a magasság és az önkiegyensúlyozás.

4.2.1 Magasságtulajdonság

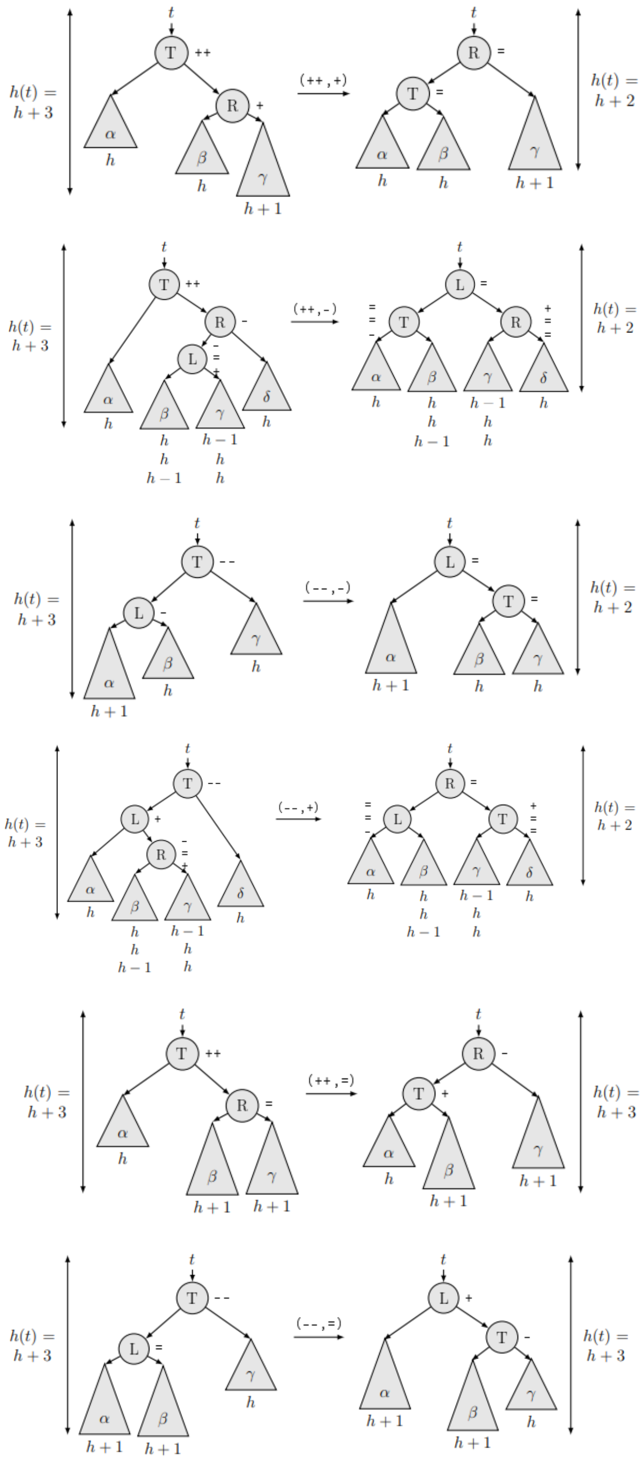
Az AVL fa az egyensúly fenntartása érdekében használja a magasságtulajdonságot. Minden csomópont rendelkezik egy magasságértékkel, amely a csomóponttól a legtávolabbi levél magasságát jelenti. Az AVL fában az összes csomópont magasságkülönbsége legfeljebb 1 lehet, vagyis az AVL fa egyensúlyban van.

4.2.2 Önkiegyensúlyozás

Ha egy beszúrási vagy törlési művelet után az AVL fa egyensúlyát megsértik, akkor a fa kiegyensúlyozására szolgáló rotációs műveleteket végeznek. A rotációk célja, hogy a magasságkülönbséget korrigálják és visszaállítsák az AVL fa egyensúlyát.

4.2.3 Rotációk

Annak függvényében, hogy melyik oldal változott és mennyire az alábbi rotációkat kell használni. Ha a bal oldali részfa süllyed egy szintet akkor - jelölést kap, ha a jobb akkor $+$ jelölést. A $++/-$ azt jelenti, hogy az egyik oldalon a részfa magassága kettővel nagyobb mint a másikon.



4.3 B+ fa

A B+ fa, amiben minden csúcs legfeljebb d mutatót ($4 \leq$ d), és legfeljebb d-1 kulcsot tartalmaz,ahol d a fára jellemző állandó, a B+ fa fokszáma. Úgy tekintjük, hogy a belső csúcsokban mindegyikreferencia két kulcs "között" van, azaz egy olyan részfa gyökerére mutat, amiben minden érték a kétkulcs között található.

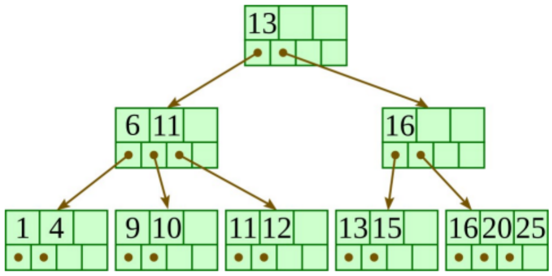


Figure 19: 4-es fokszámú B+ fa

Tetszőleges d-ed fokú B+ fa a következő invariánsokat teljesíti, ahol $4 \leq$ d állandó:

- Minden levélben legfeljebb d-1 kulcs, és ugyanennyi, a megfelelő adatrekordrahivatkozó mutató található.
- A gyökértől mindegyik levél ugyanolyan távol található.
- Minden belső csúcsban eggyel több mutató van, mint kulcs, ahol d a felső határ a mutatók számára.
- Minden Cs belső csúcsra, ahol k a Cs csúcsban a kulcsok száma: az első gyerekhez tartozórészfában minden kulcs kisebb, mint a Cs első kulcsa; az utolsó gyerekhez tartozó részfábanminden kulcs nagyobb-egyenlő, mint a Cs utolsó kulcsa; és az i-edik gyerekhez tartozó részfában($2 \leq i \leq k$) lévő tetszőleges r kulcsra Cs.kulcs[i-1] $\leq r <$ Cs.kulcs[i].
- A gyökércsúcsnak legalább két gyereke van. Kivéve, ha ez a fa egyetlen csúcsa.
- Minden, a gyökértől különböző belső csúcsnak legalább d/2 alsó egész-rész gyereke van.
- Minden levél legalább d/2 alsó egész-rész kulcsot tartalmaz. A B+ fa által reprezentált adathalmaz minden kulcsa megjelenik valamelyik levélben, balról jobbraszigorúan monoton növekvő sorrendben.

4.3.1 Beszúrás

Ha a fa üres, hozzunk létre egy új levélcsúcsot, és a beszúrandókulcs/mutató pár a tartalma! Különben keressük meg a kulcsnak megfelelő levelet! Ha a levélben márszerepel a kulcs, a beszúrás sikertelen. Egyébként:

- Ha a csúcsban van üres hely, szúrjuk be a megfelelő kulcs/mutató párt kulcs szerint rendezetten ebbe a csúcsba!
- Ha a csúcs már tele van, vágjuk szét két csúccsá, középen és osszuk el a d darab kulcsot egyenlően a kétcsúcs között! Ha a létre jött csúcs egy levél, vegyük a létrejött második csúcs legkisebb értékének másolatát, és ismételjük meg ezt a beszúró algoritmust, hogy beszúrjuk azt a szülő csúcsba! Ha a csúcs nemlevél, vegyük ki a középső értéket a kulcsok elosztása során, és ismételjük meg ezt a beszúróalgoritmust, hogy beszúrjuk ezt a középső értéket a szülő csúcsba! (Ha kell, a szülő csúcsot előbblétrehozzuk. Ekkor a B+ fa magassága nő.)

4.3.2 Törlés

Keressük meg a törlendő kulcsot tartalmazó levelet! Ha ilyen nincs, a törlés meghiúsul.

- Ha keresés során megtalált levélcsúcs egyben a gyökércsúcs is:
  - Töröljük a megfelelő kulcsot és a hozzá tartozó mutatót a csúcsból!
  - Ha a csúcs tartalmaz még kulcsot, kész vagyunk.
  - Különben töröljük a fa egyetlen csúcsát, és üres fát kapunk.
- A keresés során megtalált levélcsúcs nem a gyökércsúcs:

  - Töröljük a megfelelő kulcsot és a hozzá tartozó mutatót a levélcsúcsból!
  - Ha a levélcsúcs még tartalmaz elég kulcsot és mutatót, hogy teljesítse az invariánsokat, készvagyunk.
  - Ha a levélcsúcsban már túl kevés kulcs van ahhoz, hogy teljesítse az invariánsokat, de a következő,vagy a megelőző testvérének több van, mint amennyi szükséges, osszuk el a kulcsokat egyenlően közte és a megfelelő testvére között! Írjuk át a két testvér közös szülőjében a két testvérhez tartozóhasító kulcsot a két testvér közül a második minimumára!
  - Ha a levélcsúcsban már túl kevés kulcs van ahhoz, hogy teljesítse az invariánst, és a következő,valamint a megelőző testvére is a minimumon van, hogy teljesítse az invariánst, akkor egyesítsükegy vele szomszédos testvérével! Ennek során a két testvér közül a (balról jobbra sorrend szerinti)másodikból a kulcsokat és a hozzájuk tartozó mutatókat sorban átmásoljuk az elsőbe, annak eredetikulcsai és mutatói után, majd a második testvért töröljük. Ezután meg kell ismételnünk a törlőalgoritmust a szülőre, hogy eltávolítsuk a szülőből a hasító kulcsot (ami eddig elválasztotta a mostegyesített levélcsúcsokat), a most törölt második testvérre hivatkozó mutatóval együtt.

- Belső — a gyökértől különböző — csúcsból való törlés:
  - Töröljük a belső csúcs éppen most egyesített két gyereke közti hasító kulcsot és az egyesítés sorántörölt gyerekére hivatkozó mutatót a belső csúcsból!
  - Ha a belső csúcsnak van még floor(d/2) gyereke, (hogy teljesítse az invariánsokat) kész vagyunk.
  - Ha a belső csúcsnak már túl kevés gyereke van ahhoz, hogy teljesítse az invariánsokat, de a következő, vagy a megelőző testvérének több van, mint amennyi szükséges, osszuk el a gyerekeketés a köztük levő hasító kulcsokat egyenlően közte és a megfelelő testvére között, a hasító kulcsokközé a testvérek közti (a közös szülőjükben lévő) hasító kulcsot is beleértve! A gyerekek és ahasító kulcsok újraelosztása során, a középső hasító kulcs a testvérek közös szülőjében a kéttestvérhez tartozó régi hasító kulcs helyére kerül úgy, hogy megfelelően reprezentálja a köztükmegváltozott vágási pontot! (Ha a két testvérben a gyerekek összlétszáma páratlan, akkor azújraelosztás után is annak a testvérnek legyen több gyereke, akinek előtte is több volt!)
  - Ha a belső csúcsnak már túl kevés gyereke van ahhoz, hogy teljesítse az invariánst, és a következő,valamint a megelőző testvére is a minimumon van, hogy teljesítse az invariánst, akkor egyesítsükegy vele szomszédos testvérével! Az egyesített csúcsot a két testvér közül a (balról jobbra sorrendszerinti) elsőből hozzuk létre. Gyerekei és hasító kulcsai először a saját gyerekei és hasító kulcsaiaz eredeti sorrendben, amiket a két testvér közti (a közös szülőjükben lévő) hasító kulcs követ, ésvégül a második testvér gyerekei és hasító kulcsai jönnek, szintén az eredeti sorrendben. Ezutántöröljük a második testvért. A két testvér egyesítése után meg kell ismételnünk a törlő algoritmusta közös szülőjükre, hogy eltávolítsuk a szülőből a hasító kulcsot (ami eddig elválasztotta a mostegyesített testvéreket), a most törölt második testvérre hivatkozó mutatóval együtt.

- A gyökércsúcsból való törlés, ha az nem levélcsúcs:
  - Töröljük a gyökércsúcs éppen most egyesített két gyereke közti hasító kulcsot és az egyesítés sorántörölt gyerekére hivatkozó mutatót a gyökércsúcsból!
  - Ha a gyökércsúcsnak van még 2 gyereke, kész vagyunk.
  - Ha a gyökércsúcsnak csak 1 gyereke maradt, akkor töröljük a gyökércsúcsot, és a megmaradtegyetlen gyereke legyen az új gyökércsúcs! (Ekkor a B+ fa magassága csökken.)

5 Hasító táblák

A hasító tábla (hash table), más néven hasítótábla vagy hash map, egy hatékony adatszerkezet, amely kulcs-érték párokat tárol. A hasító tábla kulcsokat használ a tárolt értékekhez való gyors hozzáférésre. A hasító tábla alapvetően egy tömb, amely az adatokat ún. hash funkció segítségével tárolja és keresi.

5.1 Hasító függvények

Amikor egy kulcshoz tartozó értéket hozzá szeretnénk adni a hasító táblához, először egy hash funkciót alkalmazunk a kulcsra. A hash funkció egy olyan algoritmus, amely egyedi hash kódot generál a kulcs alapján. A hash kód egy indexet határoz meg a tömbben, ahol az értéket tárolni fogjuk.

- *Tárolás:* Az előző lépésben generált hash kód alapján elhelyezzük az értéket a hasító táblában lévő tömb megfelelő indexénél. Ha két vagy több kulcsnak véletlenül ugyanaz a hash kódja, akkor azt hash ütközésnek nevezzük.
- *Hash ütközések kezelése:* A hash ütközések kezelése kritikus szerepet játszik a hasító tábla hatékonyságában.
  - *Láncolás:* Minden hash kódhoz egy "lánc" tartozik, amely az ütköző kulcsokat tárolja egy adatszerkezetben (általában egy láncolt lista vagy tömb). Ha egy új kulcsnak azonos hash kódja van, a megfelelő láncra kerül, és az érték hozzáadódik a láncolt listához vagy tömbhöz.
  - *Nyílt címzés:* Az ütköző kulcsokat közvetlenül a tömbben tároljuk, anélkül, hogy külön adatszerkezetet használnánk. Ha egy új kulcsnak ütközik a hash kódja, különböző üres helyeket próbálunk meg keresni a tömbben, amíg üres helyet találunk a tároláshoz.
- *Keresés:* A keresés műveletekor ismét alkalmazzuk a hash funkciót a keresett kulcsra, és meghatározzuk az érték tárolásának helyét a hasító táblában. Ha láncolást használunk, akkor végigmegyünk a láncolt listán, hogy a kulcshoz tartozó értéket keresük a hasító táblában, alkalmazzuk a hash funkciót a keresett kulcsra, és meghatározzuk az érték tárolásának helyét a hasító táblában. Ha láncolást használunk, akkor végigmegyünk a láncolt listán vagy a tömbben, és összehasonlítjuk a kulcsokat, hogy megtaláljuk a megfelelő értéket.
- *Törlés:* A törlés művelete hasonló a kereséshez. Először alkalmazzuk a hash funkciót a kulcsra, és meghatározzuk az érték tárolásának helyét a hasító táblában. Ha láncolást használunk, akkor végigmegyünk a láncolt listán vagy a tömbben, megtaláljuk a megfelelő értéket, és töröljük azt.

A hasító tábla előnyei közé tartozik a gyors adatelérés. Ha a hash funkció jól tervezett és a hash ütközéseket hatékonyan kezeljük, a keresési, beszúrási és törlési műveletek átlagos időkomplexitása O(1) közelíthető, ami nagyon hatékony. Azonban a hasító tábla néhány korlátot is felvet. Először is, a hash funkció nem mindig garantálja a teljesen egyedi hash kódokat, így lehetőség van a hash ütközések előfordulására. Ezt a megfelelő ütközéskezeléssel kell kezelni. Másodsorban, a hasító tábla fogyasztja a memóriát, különösen, ha nagy méretű tömböt kell tartalmaznia. Az adatmennyiség és a hash funkció hatékonysága közötti egyensúly megtalálása fontos szempont a hatékonyság szempontjából.

5.2 Próbasorozat

A próbasorozat egy adatszerkezetekben, például hasító táblákban vagy hasábokban alkalmazott módszer, amelyet a kulcsok egyedi helyét meghatározására használnak. Amikor egy kulcsot beszúrunk vagy keresünk egy hasító táblában vagy hasábokban, a próbasorozatot használjuk annak meghatározására, hogy hol található a kulcs tárolási helye a táblában. A próbasorozat olyan sorozat vagy sorrend, amelyet a hash kódhoz vagy a kulcs alapján generálunk. A leggyakoribb próbasorozatok közé tartoznak:

- *Lineáris próbasorozat:* A kiválasztott hash kód vagy index már foglalt a táblában, az algoritmus egymás után következő indexeket próbál meg, amíg üres helyet nem talál. Például, ha az eredeti hash kód 5, és az 5-ös index már foglalt, az algoritmus a 6-os, majd a 7-es indexet próbálja meg, és így tovább.
- *Négyzetes próbasorozat:* Az algoritmus négyzet alakban növeli az indexet az ütközés esetén. Például, ha az eredeti hash kód 5, és az 5-ös index már foglalt, az algoritmus a $(5 + 1^2 = 6)$, majd a $(5 + 2^2 = 9)$ indexet próbálja meg, és így tovább.
- *Dupla hasításos próbasorozat:* Az algoritmus egy második hash funkciót használ az ütközés esetén a következő index meghatározására. A második hash funkció különböző hash kódot generál, amely segít elkerülni az összeomlást és a ciklusokat az indexek között.

6 Gráfok ábrázolásai

- *Szomszédsági mátrix:* A szomszédsági mátrix egy n x n méretű mátrix, ahol n a gráf csúcsainak száma. Az i. sor és j. oszlop eleme az i. és j. csúcs közötti élek jelenlétét vagy súlyát jelzi. Ha az élek irányítottak, akkor a mátrix nem szimmetrikus. Ez a reprezentáció hatékonyan tárolja a gráf szerkezetét, de erőforrásigénye négyzetes arányban nő a csúcsok számával.
- *Éllista:* Az éllista egy olyan lista, amely az összes él adatait tartalmazza. Minden élhez tartozik a kezdőcsúcs, a végcsúcs és esetleges további tulajdonságok, például súly. Ez a reprezentáció kevés memóriát igényel, de a gráf szerkezetének lekérdezésekor időigényesebb lehet.
- *Szomszédsági lista:* A szomszédsági lista minden csúcshoz tartozó listát tartalmaz, amelyben felsorolják a csúcs közvetlenül szomszédos csúcsait vagy éleit. Ez a reprezentáció általában hatékonyabb a ritka gráfoknál, mivel csak a ténylegesen szomszédos csúcsokat tárolja. Azonban a gráf szerkezetének lekérdezésekor lineáris időigényű lehet.
- *Incidencia mátrix:* Az incidencia mátrix egy n x m méretű mátrix, ahol n a csúcsok száma, m pedig az élek száma. Az i. sor és j. oszlop eleme 1, ha az i. csúcs érintett az j. élben, különben 0 vagy más jelölés lehet. Ez a reprezentáció hatékonyan tárolja a gráf szerkezetét és az élek attribútumait, de az erőforrásigénye arányos a csúcsok és élek számával.