8. ELSŐBBSÉGI SOR

Az *elsőbbségi sor* köznapi előfordulásának megfelel minden olyan tároló, amelyből mindig a *legnagyobb* (vagy a *legkisebb*) elemet vesszük ki következő lépésben. Amíg például a háziorvosi rendelőben a páciensek várakozási sort alkotnak, addig egy sürgősségi ügyeleten a betegek ellátása az állapotuk súlyossága szerint történik; azt is mondhatjuk, hogy ebben a helyzetben egy elsőbbségi sor működik. Másik példa lehet a nagygépes operációs rendszerekben a prioritás-értékekkel rendelkező *job*-ok feldolgozása. Az elsőbbségi sor, amelyet prioritásos sornak nevezünk, egy maximum vagy minimum kiválasztó struktúra.

Keressük azt az *adatszerkezetet*, amelyben az elsőbbségi sor tárolása, műveleteinek megvalósítása hatékony módon lehetséges. Olyan struktúrát szeretnénk találni, amelyben az az elemek és a hozzájuk tartozó prioritások elhelyezése, illetve kivétele minél rövidebb idő alatt végezhető el. Nem zárjuk ki, hogy céljainknak több adatszerkezet is megfelel.

Előre vetítve megfontolásaink eredményét, három adatszerkezetet vizsgálunk meg és közülük egyet találunk igazán alkalmasnak az elsőbbségi sor ábrázolására. (Vizsgálataink eredményét általánosítva akár további hatékony adatszerkezeteket is feltételezhetünk, mint ahogy azok valóban rendelkezésünkre is állnak, a haladó tankönyvek és kurzusok tanulsága szerint.)

Ha *eltekintünk* maguktól az *objektumoktól*, amelyekhez az elsőbbségi értékeket rendeljük, és pusztán a *prioritások* halmazán végezzük a beszúrás és a kivétel (törlés) műveleteit, akkor a célunk úgy is értelmezhető, mint egy *hatékony maximum kiválasztó struktúra* megtalálása. Ennek jelentőségét az adja, hogy számos algoritmus működésének a lényegét egy iterált maximum- vagy minimum-kiválasztás jelenti, így a hatékonyság meghatározó tényezője a legnagyobb vagy legkisebb érték gyors kiválasztása. A legtöbb *gráfos algoritmusra* érvényes ez a megjegyzés.

8.1. Az elsőbbségi sor és az absztrakciós szintek

Célszerű, ha először *szerkezeti elképzelések nélkül* specifikáljuk az elsőbbségi sor fogalmát. Annál is inkább az, mivel egy hatékony reprezentáció megtalálása a célunk és könnyen lehet, hogy csak az ADT szint az, ami a különböző megvalósításokban közös.

Kézenfekvő lehetőség az, hogy a prioritásokat egy tömbben tároljuk, rendezetlen módon. Ha végig gondoljuk, hogy az elsőbbségi sor – alább bevezetésre kerülő – műveletei hogyan működnek, akkor kínálkozik az az ötlet, hogy próbálkozzunk rendezett tömbös tárolással is. Ebben a két esetben azonos a reprezentáló adatszerkezet, így absztrakt szinten sem különböznek. Látni fogjuk azonban, hogy célszerű megpróbálkozni egy sajátos adatszerkezettel, a kupaccal is, amelyet absztrakt szinten bináris faként szemlélünk.

Ezek szerint az ADS szint esetünkben nem mutat egyértelműen a megfelelő ábrázolás irányába. Az ADS szintet, mivel ott még nem tudunk egyértelmű absztrakt szerkezetet kialakítani, lényegében átlépjük. Mégis, *különleges szerepe* és jelentősége lesz az ADS szintnek a kupac adatszerkezet tanulmányozásában. *A kupacot mindig tömbben tároljuk, de egy fastruktúrában gondoljuk el az elemeit!* Nem lenne könnyű a kupac műveleteinek működését szigorúan a tömbön megtervezni vagy megérteni. A fa adatszerkezet azonban roppant szemléletes hátteret ad ehhez.

Ezek előre bocsátása után térjünk vissza az ADT szinthez, éppen csak annyira, hogy nem formálisan specifikáljuk az elsőbbségi sort. Az algebrai specifikációnak nincs elvi akadálya, de a logikai axiómák ezúttal, az elsőbbségi sor definiálásánál, nehézkes eszközzé válnak.

A funkcionális specifikációhoz tartozott az adattípus matematikai reprezentálása. Ez most egyszerűen a *prioritások halmaza*, vagyis a p elsőbbségi sor absztrakt szinten az alábbi halmazzal definiáljuk:

$$p = \{k_1, k_2, \ldots, k_n\}$$

Feltesszük továbbá, hogy az elsőbbségi értékek természetes számok:

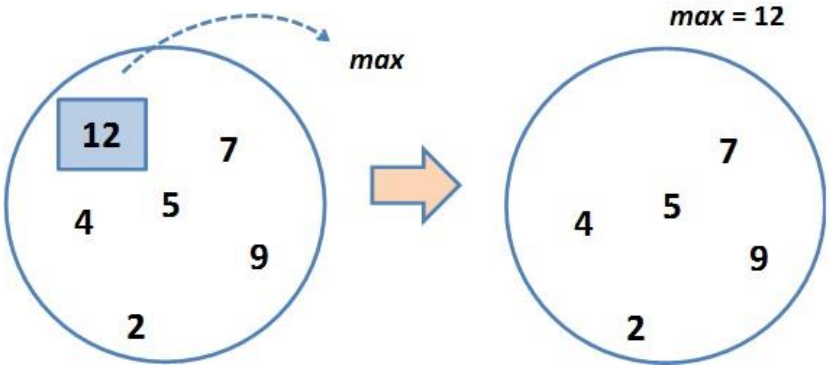
$$\forall j\ (1 \leq j \leq n)\colon k_j \in \mathbb{N}$$

Az *üres sort* az $n = 0$ érték jelzi. Ha felidézzük a verem és a sor matematikai reprezentációit, akkor azok is halmazok voltak, mégpedig (*elem, időpont*) rendezett párokból alkotott halmazok. Az időpont az elem beillesztésének idejét jelzi. Most, az elsőbbségi sornál, az időpont helyére a prioritás lép. Így (*elem, prioritás*) rendezett párokból kellene halmazt formálnunk, azonban az elemet magát – a modell egyszerűsítése jegyében – *elhagyjuk*.

Bevezetjük a $p \in P$ prioritásos sorok műveleteit, ahol P jelöli az összes olyan p prioritásos sor halmazát, amely eleget tesz a fenti definíciónak. Megismételjük, hogy az egyszerűség kedvéért az elsőbbségi sorokba csak az egyes elemek prioritását tesszük be, az magukat az elemeket nem. Az elő-, utófeltételes specifikáció formalizmusát most mellőzzük, a műveletekre informális meghatározást adunk.

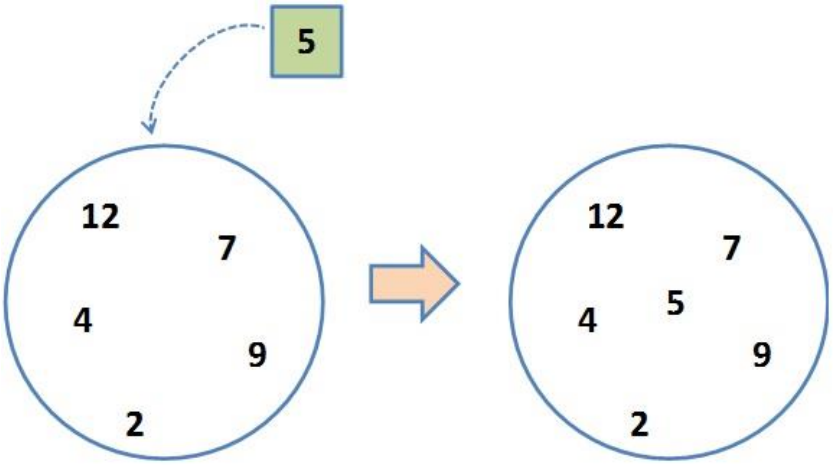
- Az *Üres* (*p*) művelet létrehoz egy üres prioritásos sort, amelyre a *p* azonosítóval hivatkozhatunk.
- Az *Üres-e* (*p*) művelet a logikai igaz vagy hamis értéket adja vissza, annak megfelelően, hogy *p* üres, illetve tartalmaz elemet.
- A *PrSorból* (*p*, *x*) művelet kiválasztja a *p* halmaz maximális elemét (az egyiket, ha több van) az eltávolítja onnan; az értéket az *x* változó kapja.
- *PrSorba* (*p*, *k*) művelet elhelyezi *p*-ben a megadott *k* prioritást.
- A *MaxPrior* (*p*) művelet lekérdezi a legnagyobb *p*-beli értéket és azt úgy adja vissza, hogy *p* változatlan marad.

Két művelet értelmezési tartományára kell megszorítást adnunk. A *PrSoról* és a *MaxPrior* művelet csak a *nem-üres* elsőbbségi sorra értelmezhetők. A *p* halmaz tehát nem lehet üres. A *PrSorba* művelet esetén a *p* elsőbbségi sor befogadóképességére nem adunk meg felső korlátot.



8.1. ábra. Elem kivétele az elsőbbségi sorból (ADT)

A *PrSorból* műveletet a 8.1. ábra szemlélteti, míg a 8.2. ábra a *PrSorba* művelethez nyújt illusztrációt.



8.2. ábra. Elem beszúrása az elsőbbségi sorba (ADT)

Az absztrakt adattípus informális leírása után az ADS szintre térünk át. Amint jeleztük, ezt a szintet most célszerű átlépni és a reprezentáció szintjén folytatni a meggondolásainkat. A fentiek szerint az ADS szint most nem elegendő támpontot megfelelő reprezentáció megtalálásához.

Három adatszerkezetet tanulmányozunk, ezek közül az első kettő tömb, míg a harmadik, a kupac, ADS-szinten fastruktúrájú. A kupacos reprezentáció vizsgálatakor különleges szerepet nyer az ADS szint, ugyanis *a kupacot* mindig *tömbben tároljuk,* de az *elemeit absztrakt szinten egy fastruktúrában* helyezzük el. A kupac műveleteit így szemléletesen értelmezhetjük, és utána már csak vissza kell képezni a tömb szintjére.

8.2. Az elsőbbségi sor tömbös megvalósításai

A prioritásokat kézenfekvő módon egy tömbben tároljuk. Először a rendezetlen tömbbel próbálkozunk, utána megvizsgáljuk, hogy a rendezett állapotban tartott tömbbel jobb lehetőséghez jutunk-e.

8.2.1. Rendezetlen tömb

Tekintsük először azt a kézenfekvő lehetőséget, hogy az elsőbbségi értékeket egy $A[1..n]$ tömbben tároljuk, rendezetlen módon, például annak megfelelően, ahogyan a beírás sorrendjében érkeznek az elemek. A tömbben tárolt k számú prioritás folyamatosan helyezkedik el az 1,... k indexű helyeken. Az üres, illetve a tele sort $k = 0$, illetve $k = n$ jelzi.

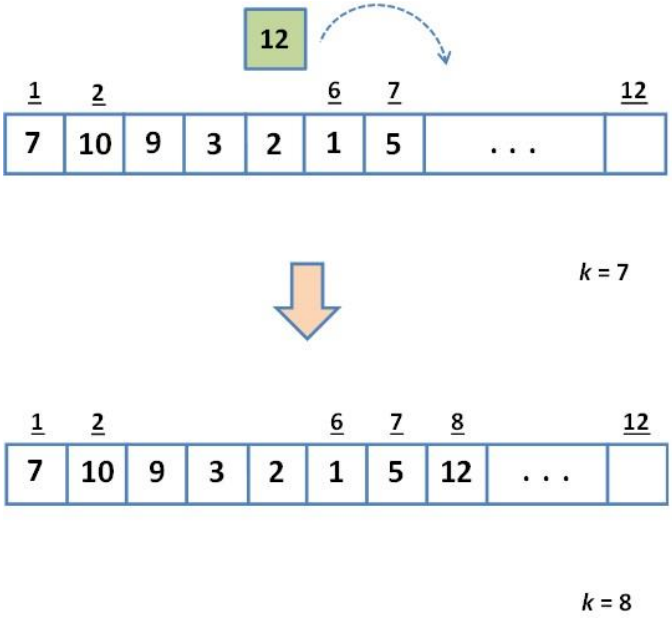
Az új prioritás beszúrását végző *PrSorba* művelet ezek szerint elhelyezi az új értéket a tömbben tárolt elemek utáni első szabad helyre. A 8.3. ábra szemlélteti ezt a lépést. A művelet egyszerű algoritmusa a 8.4. ábrán látható. *Hiba* abban az esetben lép fel, ha a sor betelt és a tömbben már nincs hely új elem befogadására. A beillesztés *konstans időben* elvégezhető:

$$T_{PrSorba}(k) = \Theta(1)$$

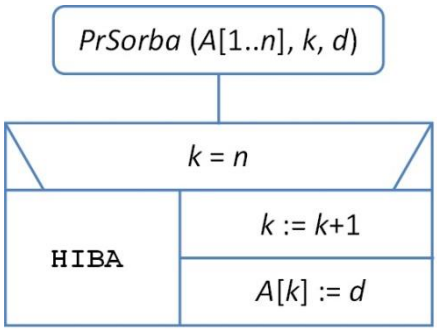
A *legnagyobb prioritás* megkeresését és kivételét végző *PrSorból* művelet először végrehajtja a sokszor alkalmazott *maximum-kiválasztást*. A megtalált maximális elem kivétele után annak helyre teszi a „jobb szélső” elsőbbségi értéket, így folytonos marad a tárolt elemek sora. A 8.5. ábra illusztrálja ez elsőbbségi sorból való kivételt. Az algoritmus a 8.6. ábrán látható. *Hiba* akkor lép fel, ha üres sorból próbálunk elemet kivenni.

A műveletigény a maximum-kiválasztás által végrehajtott k-1 összehasonlítás miatt időben *lineáris* lesz:

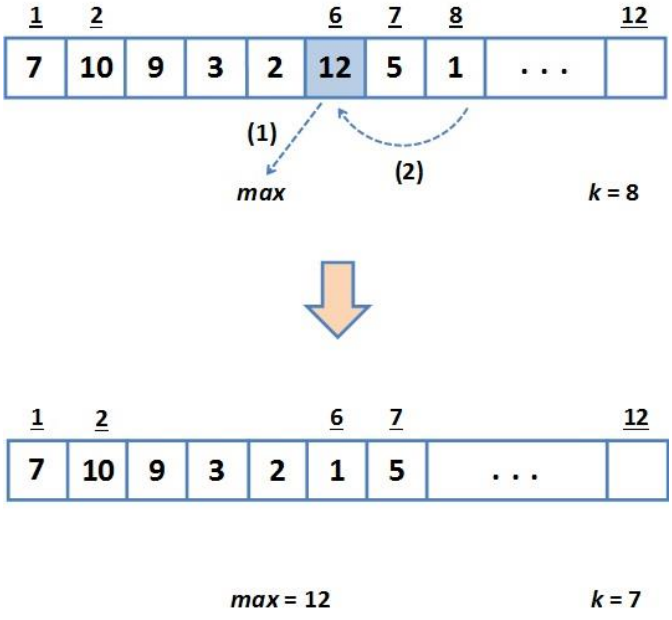
$$T_{PrSorból}(k) = \Theta(k)$$



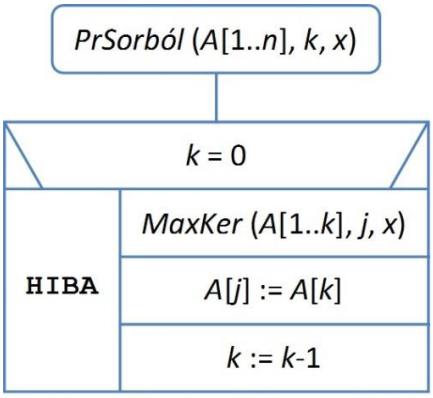
8.3. ábra. Elem beszúrása az elsőbbségi sorba (rendezetlen tömb)



8.4. ábra. Elem beszúrása az elsőbbségi sorba, algoritmus (rendezetlen tömb)



8.5. ábra. Elem kivétele az elsőbbségi sorból (rendezetlen tömb)

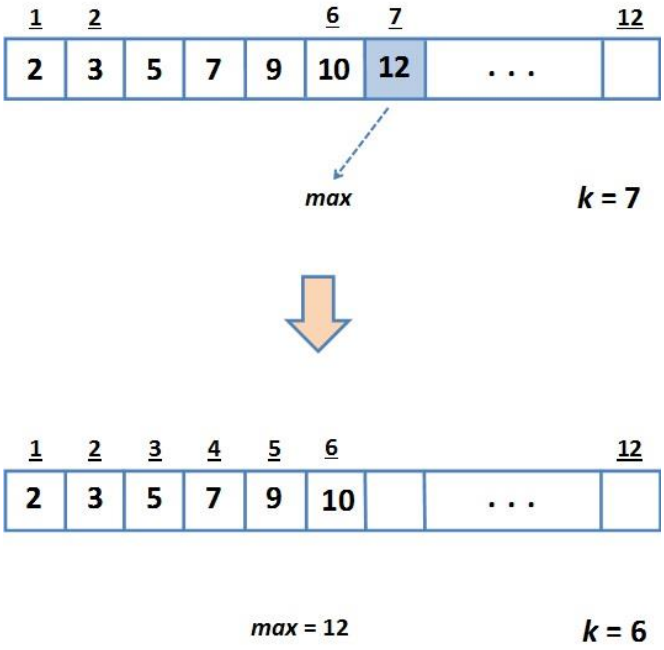


8.6. ábra. Elem kivétele elsőbbségi sorból, algoritmus (rendezetlen tömb)

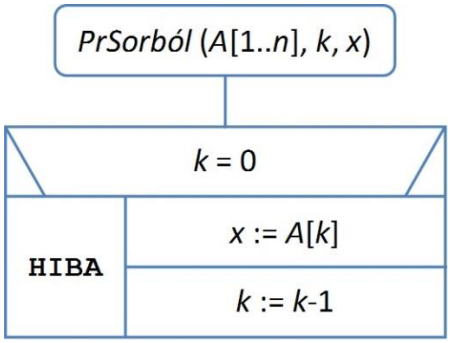
Összefoglalva azt mondhatjuk, hogy a rendezetlen tömbös megvalósítás esetén a két meghatározó művelet egyike *konstans*, míg a másik *lineáris idejű*.

8.2.2. Rendezett tömb

Próbálkozzunk most azzal, hogy a prioritásokat *rendezett* állapotban tarjuk a *tömbben*. Ha a legnagyobb elsőbbségi érték a sorban az utolsó, akkor közvetlenül elérhető.



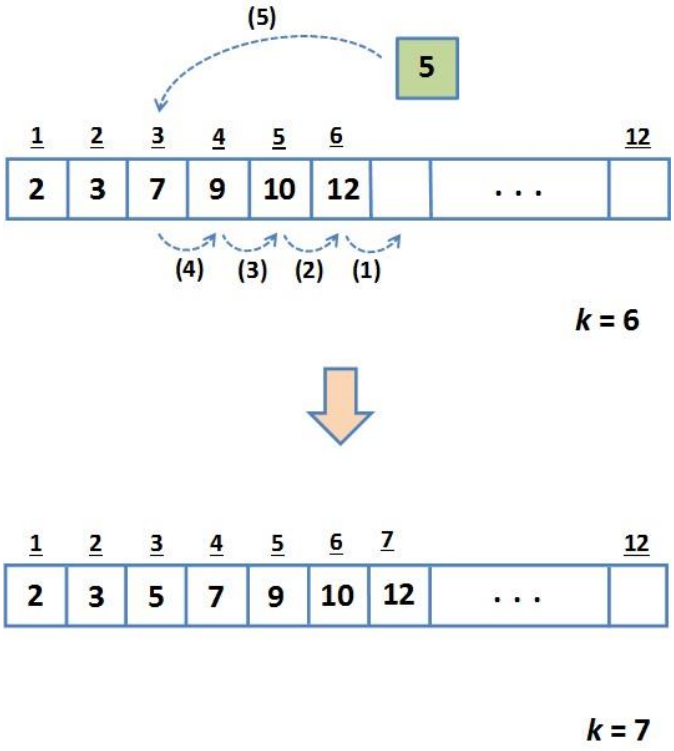
8.7. ábra. Elem kivétele az elsőbbségi sorból (rendezett tömb)



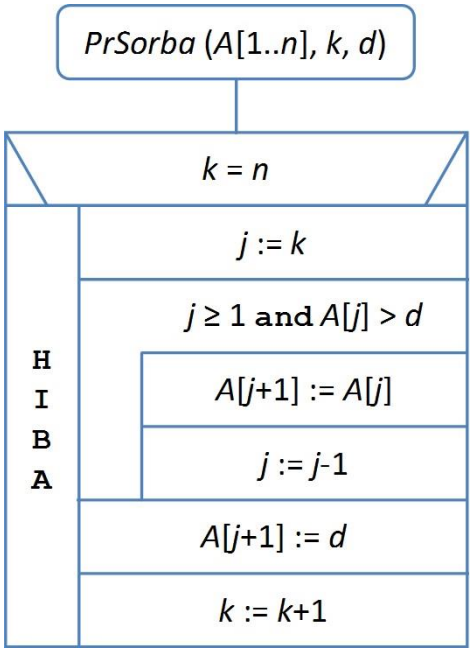
8.8. ábra. Elem kivétele elsőbbségi sorból, algoritmus (rendezett tömb)

A *PrSorból* művelet végrehajtását a 8.7. ábra mutatja be. A 8.8 ábrán az a művelet algoritmusa olvasható. *Hiba* abban az esetben lép fel, ha az üres sorból próbálunk elemet kivenni. Az elem kivétele a közvetlen elérhetőség miatt *konstans időben* végrehajtható:

$$T_{PrSorból}(k) = \Theta(1)$$



8.9. ábra. Elem beszúrása az elsőbbségi sorba (rendezett tömb)



8.10. ábra. Elem beszúrása az elsőbbségi sorba, algoritmus (rendezett tömb)

A tömb kitöltött részének rendezettségét a PrSorba művelet biztosítja. Az új elsőbbségi értéket a nagyság szerint helyére kell beszúrni. Ezt a nagyobb elemek jobbra történő léptetésével érhetjük el, ahogyan a 8.9. ábrán láthatjuk. Az algoritmus megfogalmazását a 8.10. ábra tartalmazza.

Az elemek rendezett sorába történő beillesztéshez szükséges összehasonlítások száma 1 és *k* közötti érték, így a műveletigény kifejezésében az „ordót” kell használnunk:

$$T_{PrSorba}(k) = O(k),$$

illetve felírhatjuk a maximális műveletigényt:

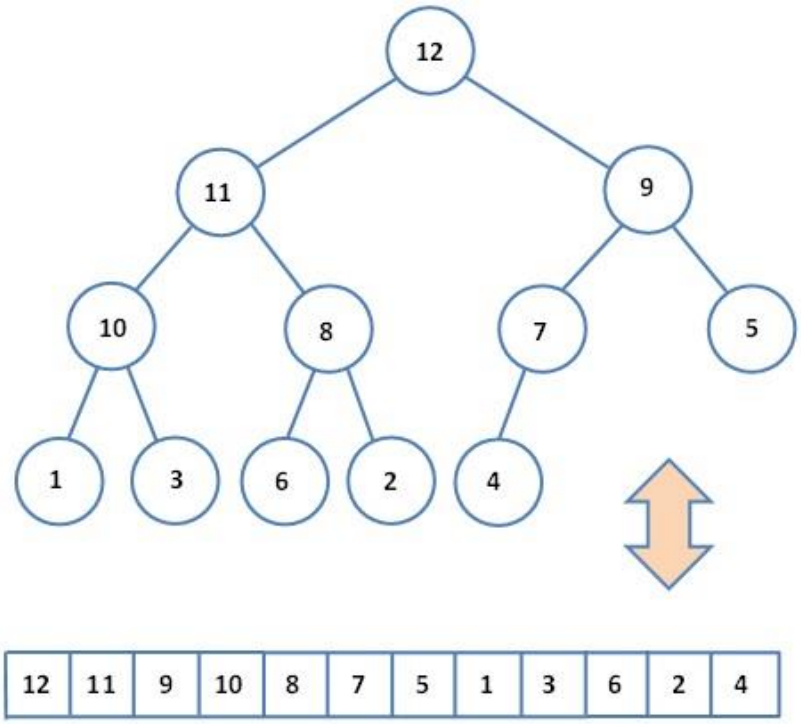
$$MT_{PrSorba}(k) = \Theta(k)$$

Összefoglalva azt mondhatjuk, hogy a rendezett tömbös reprezentáció esetén – a rendezetlen tömbös tároláshoz hasonlóan – azt kaptuk, hogy a két meghatározó művelet egyike konstans, míg a másik (legfeljebb) lineáris idejű (az átlagos műveletigényt intuitív módon szintén lineárisnak gondoljuk).

A rendezett tömbös megvalósítást halványan jobbnak érezhetjük, mert ha nem is nagyságrendben, de kevesebb lépést végez. Ráadásul a lineáris idejű művelet nem a kiválasztásra, hanem a beillesztésre fordítódik, ami valós idejű feladatoknál előny lehet. A hatékonyság egészen kis mértékben tovább javítható azzal, ha a maximális prioritás helyét megjegyezzük. Az elsőbbségi sor nagyságrendben hatékonyabb megvalósításához a kupacos reprezentáció vezet.

8.3. A kupac adatszerkezet

Bevezetjük a kulcsfontosságú *kupac* (*heap*) adatszerkezetet. Ezt ADS szinten tesszük, aminek itt komoly szerep jut, mivel a kupacot absztrakt szinten bináris fának látjuk, viszont mindig tömbben tároljuk. A kupac absztrakt adatszerkezet speciális bináris fa, amelynek az alakjára és a csúcsokban tárolt értékek közötti összefüggésekre invariáns állításokat mondunk ki.



8.11. ábra. Kupac absztrakt képe (bináris fa) és tárolása tömbben

A *kupac* absztrakt adatszerkezet egy olyan speciális *bináris faként* definiáljuk, amely

(i) *majdnem teljes*, azaz, legfeljebb a levelek szintjén hiányozhat csúcs;
(ii) *balra tömörített*, ami a levélszint csúcsaira vonatkozik és
(iii) minden belső csúcsokban tárolt érték *nagyobb vagy egyenlő,* mint a *gyerekeinek* az értékei.

A 8.11. ábra egy kupacot szemléltet. Az absztrakt ábrázolású kupac bináris fára valóban teljesül az (i) és a (ii) előírás, a csúcsokban tárolt értékek között pedig fennáll a (iii) összefüggés. A kupac bináris fájában – a balra tömörítés miatt – egyetlen olyan belső csúcs lehet, közvetlenül a levelek szintje fölött, amelynek csak bal gyereke van.

Egy n csúcsot tartalmazó majdnem teljes bináris fa *magassága* már nem csökkenthető tovább; értéke

$$h(t_n) = \lfloor log_2 n \rfloor$$

A fában bármely olyan útvonalon, amely a gyökértől indul, és valamelyik levélben végződik, az elemek *monoton csökkenő* (illetve nem növekedő) sorozatot alkotnak.

Az ábrán a kupac elemeinek a *tömbös* elhelyezése is látható. A tömbben látható sorrend ugyanaz, mint amelyet a bináris fa szintfolytonos bejárásával kapunk. A tömbben a *szülő-gyerek* és a *gyerek-szülő* kapcsolatok az előző fejezetben megismert, alábbi képletek írják le:

$$ind(bal(c)) = 2*ind(c) \text{ és } ind(jobb(c)) = 2*ind(c) + 1$$

$$ind(szülő(c)) = \lfloor ind(c) \rfloor$$

Az első két összefüggés értelemszerűen *nem-levél* csúcsokra alkalmazható, míg a harmadik képlet a *gyökértől különböző* csúcsokra érvényes.

A fában említett rendezett útvonalak, amelyek a gyökértől valamely levélig tartanak, a tömbben is megtalálhatók; ezeket nevezhetjük *leszálló rendezett láncoknak*. A teljes tömb nem rendezett ugyan, de az elemeit tartalmazó speciális láncok azok. Ezzel a *részleges rendezettséggel* a kupac egy „kompromisszumos" középutat valósít meg a rendezetlen és a rendezett tömb között.

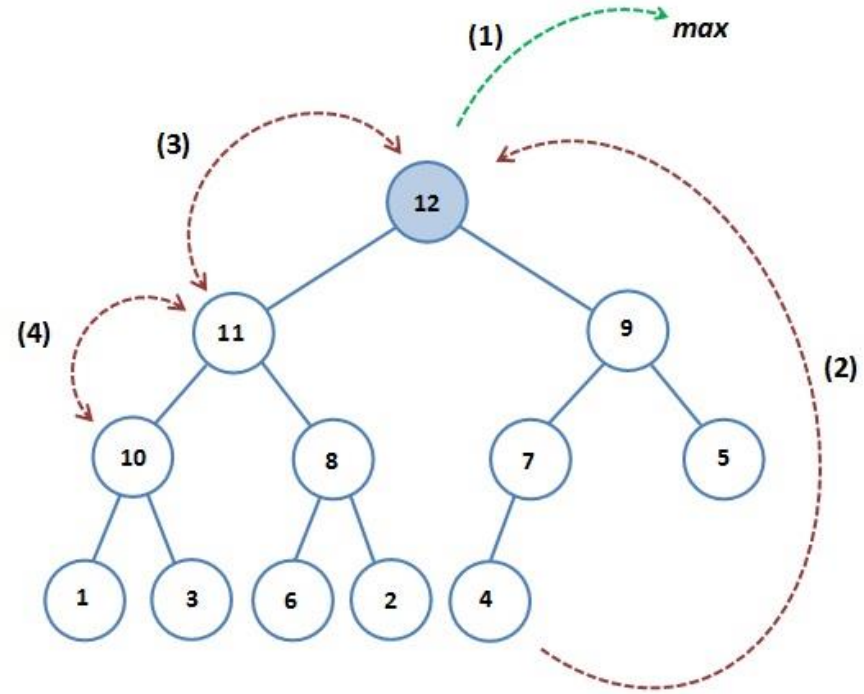
8.4. Az elsőbbségi sor kupacos megvalósítása

Nézzük most az elsőbbségi sornak azt a reprezentációját, amikor a prioritásokat egy *kupac* adatszerkezetben tároljuk. A két meghatározó tevékenységet, a *PrSorból* és a *PrSorba* műveleteket szemléletes módon, a bináris fán értelmezzük, anélkül, hogy a tömbös ábrázolás szintjén megírnánk a precíz algoritmusaikat. (Megjegyezzük, hogy a kupacos rendezésről szóló fejezetben nem állunk meg az ADS szint vizualitásánál, hanem tömbös algoritmusokat tervezünk, amelyek közül az egyik éppen az itteni *PrSorból* művelet lesz.)

A *PrSorból* művelet a legnagyobb értéket kiveszi a kupacból; lásd a 8.12. ábrán az (1)-gyel jelölt lépést. A maximális elemet a fa gyökéreleme tartalmazza, ami közvetlenül elérhető, mint a tömbben első (1-es indexű) eleme. Ha ezt az értéket eltávolítjuk, akkor sérül a *kupac tulajdonság*, amit két nagyobb lépésben *állítunk helyre*.

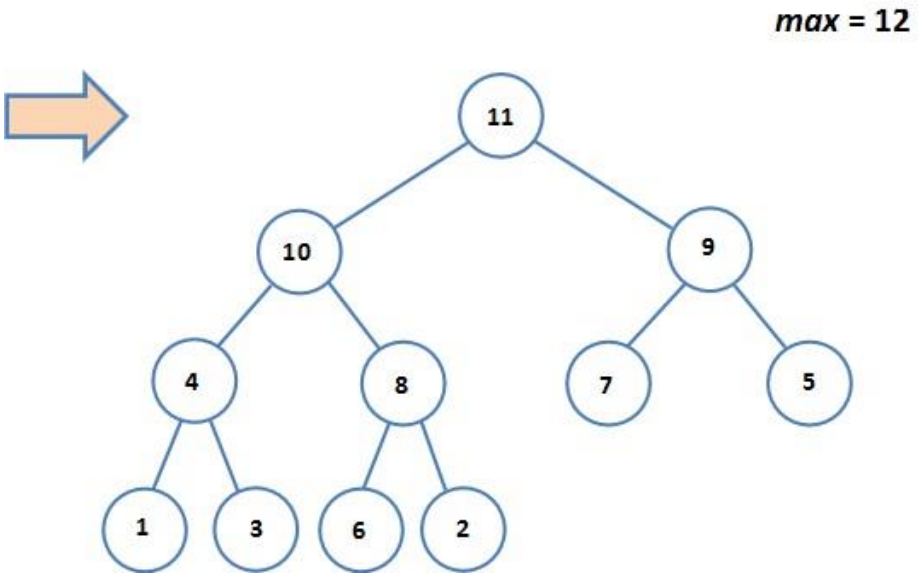
Először a *fa alakjára* vonatkozó fenti (i) és (ii) feltételeket juttatjuk érvényre azzal, hogy az alsó szint jobb szélső elemét felvisszük a gyökérbe, amint azt az ábra (2)-es lépése illusztrálja. A felvitt érték a tömb utolsó eleme, így a tárolt adatok folytonossága megmarad.

A gyökérelembe került *érték* aligha lehet a legnagyobb új prioritás, ezért a *nagyság szerint helyére* „*süllyesztjük*" a fában. Ahogyan az ábrán a (3)-as és (4)-es lépés mutatja, a gyökeret addig cseréljük a nagyobb értékű gyerekkel, amíg helyre nem áll a fenti (iii) tulajdonság.



8.12. ábra. Elem kivétele elsőbbségi sorból; az eljárás (kupac, ADS)

A kupac új állapota, amely a maximális prioritás kivétele és a kupac tulajdonság helyreállítása után kialakult, a 8.13. ábrán látható.



8.13. ábra. Elem kivétele utáni új elsőbbségi sor (kupac, ADS)

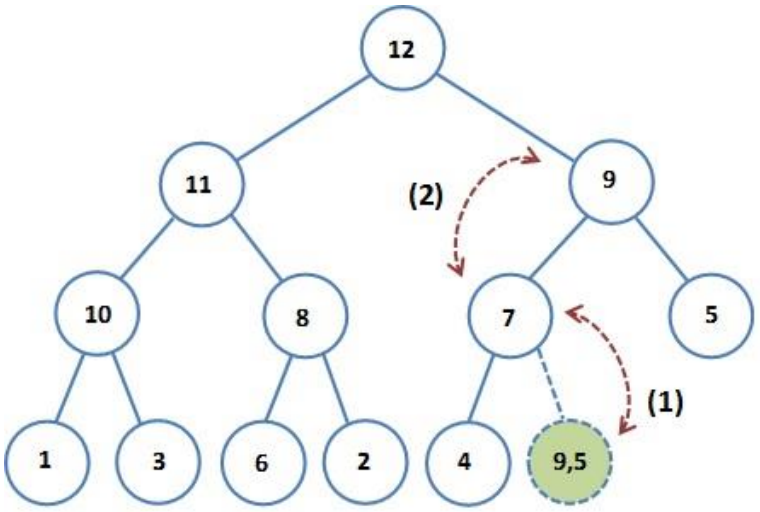
A *PrSorból* művelet lépésszámát nagyságrendben felülről becsüli bináris fa magassága, ugyanis az első két értékadás után legfeljebb annyi cserét hajtunk végre, amennyivel a gyökérelem a levelek szintjére jut. A cseréket megelőzi annak a vizsgálata, hogy kell-e cserélni, illetve, ha igen, akkor melyik gyerekkel. Ezzel együtt nyilvánvalóan helyesek a

$$T(n) = O(\log n) \text{ és } MT(n) = \Theta(\log n)$$

nagyságrendi becslések a lépésszámra az egyedi, illetve a legrosszabb esetben.

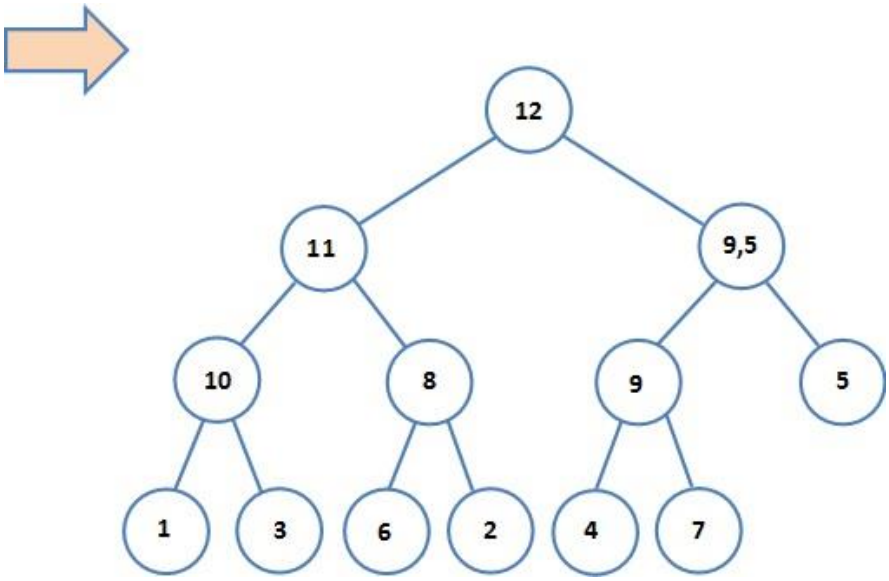
A *PrSorba* művelet egy új értéket helyez az elsőbbségi sorba. A kupac tulajdonságokat éppen úgy két nagyobb lépésben juttatjuk érvényre, mint ahogyan azt a *PrSorból* műveletnél láttuk.

Az új érték helye az alsó szint jobb szélén lesz, ami a tömbös ábrázolásban hasonlóan a jobb szélső elemre következő cella hozzá vételét jelenti a sorhoz. Az új elemet – a szülő csúccsal esetleg többször megcserélve – addig „*szivárogtatjuk fel*”, amíg a *nagyság szerinti helyére* nem kerül.



8.14. ábra. Elem beszúrása elsőbbségi sorba; az eljárás (kupac, ADS)

Az új érték beszúrásának lépéseit a 8.14. ábra mutatja be. Az így kialakult kupac a következő, 8.15. ábrán látható.



8.15. ábra. Elem beszúrása utáni új elsőbbségi sor (kupac, ADS)

A *PrSorba* művelet lépésszámát szintén felülről becsüli nagyságrendben bináris fa magassága, hiszen a levelek szintjéről legfeljebb a gyökér pozícióig lehet emelni az új elemet. A cseréket még egy vizsgálat is megelőzi. A művelet lépésszáma nyilvánvalóan

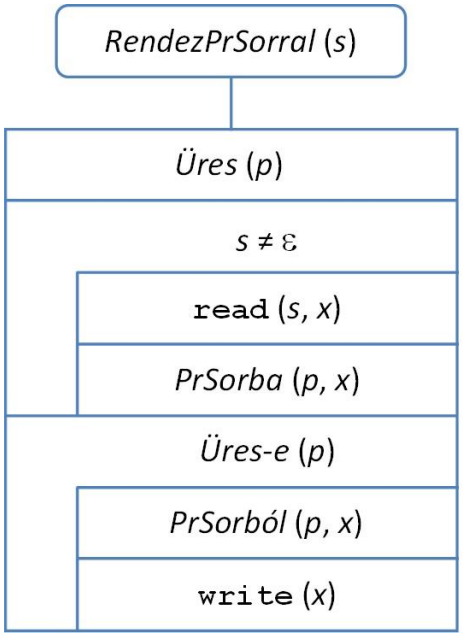
$$T(n) = O(\log n) \text{ és } MT(n) = \Theta(\log n)$$

nagyságrendű az általános, illetve a legrosszabb esetben.

Azt kaptuk, hogy a *kupacos reprezentáció* esetén *mindkét* vizsgált műveletnek egyaránt $\log n$-es a futási ideje. Azt gondolhatjuk, hogy a kupaccal ábrázolt elsőbbségi sor nem csak kiegyensúlyozottabb, hanem összességében *hatékonyabb* is, mint a két tömbös megvalósítás. Ezt akkor látjuk igazán, ha egy olyan művelet sorozatot hajtunk végre, amelyben vegyesen szerepel a prioritási értékek beszúrás és kivétele.

8.5. Rendezés elsőbbségi sorral

Találtunk egy olyan egyszerű algoritmust, amely a vizsgált műveletet egyenlő számban hajtja végre működése során. Az eljárás, amint a 8.17. ábráról leolvasható, az s szekvenciális inputról beolvasott értékeket rendre beteszi egy (kezdetben üres) elsőbbségi sorba, azután pedig sorban kiveszi és kiírja onnan az elemeket. A kiírt sorozat nyilvánvalóan rendezett lesz!



8.17. ábra. Rendezés elsőbbségi sorral, algoritmus (ADT)

Meglepőnek tűnhet, hogy ez a rendező eljárás nem tartalmaz *egymásba ágyazott ciklust*. Ennek az a magyarázata, hogy az ADT szintű algoritmus leírása *elrejti* előlünk a felhasznált adatszerkezet megvalósításának módját.

Érdekes kérdés az, hogy ha az elsőbbségi sor három reprezentációját is figyelembe vesszük, akkor *milyen rendezési eljárásokat* kapunk ebből a közös absztrakt eljárásból származtatva. Azt javasoljuk, hogy a jegyzetnek a rendezésekről szóló fejezeteit áttanulmányozva, térjen vissza az Olvasó ehhez a kérdéshez! Akkor már biztosan maga is ellenőrizni tudja az alábbi megállapítások helyességét.

Ha *rendezetlen tömbben* tároljuk az elsőbbségi értékeket, akkor lényegét tekintve a *maximum kiválasztó rendezéshez* jutunk. A *rendezett tömbös* elsőbbségi sor a *beszúró rendezéshez* vezet. A *kupacos* megvalósítás lényegében a *kupacrendezést* eredményezi, csekély eltéréssel. (A beillesztett n elemből a kupac nem úgy alakul ki, hogy összevárjuk az elemeket és utána kupacot formálunk belőlük, hanem a bejövő adatok kezdettől kupacot alkotnak, amelybe minden bejövő prioritást beszúrunk. Míg az előző módszer időben lineáris eljárás, addig az utóbbi $n \log n$-es futási idejű. (Miért?))

Elemezzük most algoritmusunk *műveletigényét* mindhárom reprezentáció esetén! Az elemzés n elem beszúrására és kivételére terjed, mindannyiszor a *PrSorba* és a *PrSorból*

műveletek lépésszámát figyelembe véve, rendre a három reprezentációban. A fentiek alapján érthető, hogy az egységesség kedvéért meggondolásaink a *maximális lépésszámra* (legrosszabb eset) vonatkoznak. Az ábrázolási módtól függetlenül érvényes az alábbi összefüggés:

$$MT(n) = T_{\text{Üres}} + \sum_{k=0}^{n-1} MT_{PrSorba}(k) + \sum_{k=1}^{n} MT_{PrSorból}(k)$$

Számítsuk ki a fenti kifejezés nagyságrendjét az elsőbbségi sor három megvalósításában.

- Rendezetlen tömbös reprezentációban:

$$MT(n) = \Theta(1) + n\Theta(1) + n\Theta(n) = \Theta(n^2)$$

- Rendezett tömbös ábrázolás mellett:

$$MT(n) = \Theta(1) + n\Theta(n) + n\Theta(1) = \Theta(n^2)$$

- Kupacos megvalósítás esetén:

$$MT(n) = \Theta(1) + 2\Theta(\log_2 1 + \log_2 2 + \cdots + \log_2 n) = \Theta(n \log n)$$

A nagyságrendek elhanyagolják a konstans szorzókat, amelyek bizonyos esetekben (csekély n elemszám, vagy igen nagy konstans) nem hagyhatók figyelmen kívül. Most vehetjük úgy, hogy a nagyságrendi értékek megfelelően tájékoztatnak a reprezentáció hatékonyságáról.

Gondoljunk például arra, hogy ha *ezer* körüli rendezendő számunk van (n = 1000), akkor az első két tömbös esetben a rendezés *milliós* nagyságrendű lépést igényel, míg a kupacos reprezentációban ez a ráfordítás *tízezres* nagyságrendűre csökken. A ma már mindennaposnak számító *ezres* elemszám mellet *két nagyságrendet* nyerünk a kupac alkalmazásával.

Az elemzés alapján joggal mondhatjuk, hogy az elsőbbségi sornak *kupaccal* megvalósítása *hatékonyabb* a másik két ábrázolási módnál.

****

Megjegyzés. Az utóbbi a számításban az összegzés eredményének a nagyságrendjét még bizonyítani kell. Az egyes tagok felső becslése alapján, a jobb oldalon $O(n \log n)$ azonnal írható. Belátjuk azonban, hogy nagyságrendi egyenlőség áll fenn.

A bal oldalon a zárójelben szereplő összeget, mint a $\log_2 x$ függvény kívül írt téglalapjainak területösszegét, tekintsük integrál közelítő összegnek. Ekkor

$$\log_2 1 + \log_2 2 + \cdots + \log_2 n \geq \int_1^n \log_2 x \, dx = {}^{1}\!/_{\ln 2} \int_1^n \ln x \, dx =$$

$$= {}^{1}\!/_{\ln 2} \left[x \ln x - x \right]_1^n = {}^{1}\!/_{\ln 2} \left[n \ln n - n + 1 \right] \geq n \log_2 n$$

Ez a becslés másik iránya, amely alapján a nagyságrendi egyenlőség már valóban állítható.