

3/1. tétel:

Linearis adatszerkezetek és műveleteik

A gyűjtemények (collections) közé sorolhatók a halmaz (set), a csomag (bag, multiset) és a vector (sequence, list).

Gyűjtemények általánosan

Értelmezzük a következő adatokat, illetve eljárásokat: méret, elemszám, tartalmazás, üres-e, teli-e, létezik-e.

Halmaz

Az elemek sorrendje és többszöri előfordulásuk nem lényeges, másként megfogalmazva egy adott elem csak egyszer fordulhat elő egy halmazban. Mivel a halmaz a gyűjteményből öröklődik, ezért az előbb felsorolt adatok-eljárások itt is léteznek (értelemszerűen a halmaznak megfelelő módon megvalósítva). Ezen kívül halmazokra értelmezzük: unio, metszet (e kettőt másik halmazzal, illetve csomaggal is végrehajthatjuk), különbség, symmetricus különbség, csomaggá és vectorrá való átalakítás.

Csomag

Az elemek sorrendje itt sem lényeges, de esetleges többszöri előfordulásuk igen (vagyis itt egy adott elem többször is előfordulhat). Itt is érvényesek az ált. műveletek (csomagnak megfelelő módon megvalósítva), illetve értelmezzük még: unio, metszet (mindkettőt másik csomaggal, illetve halmazzal is), halmazzá és vectorrá való átalakítás.

Vector

Az elemek sorrendje fontos, de többszöri előfordulásuk nem (tehát egy elem többször is előfordulhat, de lényeges, hogy milyen sorrendben). A vectoroknál előforduló spec. műveletek: hozzáfűzés (append – a végére, prepend – az elejére), részvector (subsequence), adott indexű elem kiválasztása, az első/utolsó elem kiválasztása, halmazzá és csomaggá alakítás. Az átalakítások (ez mindhárom esetre érvényes) természetesen az adott cél-gyűjteménytípusra jellemző módon történnek.

Verem és sorok

Ezek olyan dynamicus halmazok (azaz pontosabban vectorok, hiszen egy elem többször is szerepelhet bennük), melyekben előre meghatározott az az elem, amelyet egy törlő művelettel eltávolíthatunk.

Verem (stack, "LIFO¹ verem") esetén a legutoljára betett elem törölhető ki elsőként. A sor (queue, "FIFO verem²") esetén pedig a legelsőként behelyezett elem törlődik legelőször. Egyik adatszerkezetnél sem értelmezzük sem a keresést, sem a rendezést.

Verem (stack, LIFO)

Shakások felhasználása pl. Assembly programozás esetén a subrutinhívás. Egyrészt a visszatérési címet is veremben tároljuk, másrészt a regiszterek állapotának megőrzésére is kiválóan alkalmas. Magasabb szintű programnyelvek esetén a parameterátadásban van fontos szerepe. Gyakorlatilag felfogható egy tömbként is. A veremmutató (SP³) mindig a legutoljára behelyezett elemre mutat (hiszen őt kell majd eltávolítani legelőször). Adatot behelyezni a PUSH, míg kivenni a POP utasítással szoktunk (Assembly). PUSH utasítás esetén SP értéke növekszik, értelemszerűen a behelyezett adat méretétől függően (ezért nem célszerű pl. az SP registert kézzel babrálni, jobb rábízni a procira). POP utasításnál (amennyiben a verem nem üres) kivesszük a tárolt adatot, majd az SP értéke a megfelelő mértékben csökken.⁴

¹ LIFO: last-in-first-out, FIFO: first-in-first-out

² A "FIFO verem" elnevezés rendkívül hibás, mert a FIFO elvű szerkezet nem verem... Ennek megfelelően a "LIFO verem" elnevezés pedig fölösleges szószaporítás, mert a LIFO elvű szerkezetet defaultból veremnek hívjuk... Hozzáértő volt a tételsor készítője... :-))

³ SP: stack pointer

⁴ Ismert olyan megvalósítás is, hogy a verem "felülről lefelé" terjeszkedik. Ennek az az előnye, hogy ha a memoria felső részére tesszük, ritkán fordul elő, hogy pl. az adat- vagy programterülettel összeakad. Ha viszont lejjebb raknánk, akkor egyrészt a programterület (adatterület) méretét korlátoznánk fölöslegesen – másrészt sokszor nehéz előre megjósolni, hogy mekkora veremterület szükséges (vagy túl kicsinek, vagy túl nagyknak választanánk).

A verem esetén kétféle túlszordulásról beszélhetünk:

1. túlszordulás (overflow): SP értéke túlnő azon a határon, amit a verem számára maximalis értéknek szántunk. Ez például eredményezhetné egyéb segmens (adat- vagy programsegmens) felülírását, ezért általában az OS kivédi.
2. alulszordulás (underflow): a verem üres, de megpróbálunk belőle olvasni.

Sor (queue, FIFO)

Elsőbbségi sornak is nevezik – úgy működik, mint pl. a pénztáraknál álló sor. Az elsőként odaálló személy lesz először kiszolgálva, az utolsóként odaálló pedig utoljára. Pl. ilyen szerkezetet használnak a printer spooler programok.

E szerkezetnél nem elég egy mutatót használnunk, hanem a sor elejét (fejét) és végét is nyilván kell tartanunk. A következő érkező elem a végére kerül beszúrásra, eközben a "vége" pointer értéke eggyel nő. Egy kivett elem esetén pedig a "fej" pointer értékét növeljük. Megjegyzendő, hogy amennyiben a "vége" értéke elérte a sor méretét, számolása ismét indulhat a sor elejétől (viszont ekkor már a "vége" pointer értéke mindenképpen kisebb kell maradjon, mint a "fej" pointer értéke, hiszen ellenkező esetben a sorba betett adat egy már eleve ott lévő adatot megsemmisítene). Ha a "fej" értéke is eléri a sor végét, akkor az is újratekint a sor elejétől.

A sor akkor üres, ha a két pointer értéke azonos (nem feltétlenül 0!). A sor pontosan akkor van tele, ha:

- A "fej" értéke 1^5 , a "vége" pedig a sor méretének megfelelő.
- A "fej" értéke pontosan eggyel nagyobb, mint a "vége" értéke.

Láncolt listák

Az előző szerkezeteknél külön pointer(ek) (mely(ek) felfogható(k) tömbindex(ek)ként is) határozták meg az egyes elemeket. A láncolt listákban viszont az egyes elemek külön-külön tartalmaznak pointer(e)ket, mellyel megvalósítható az elemek lineáris rendezettsége. A láncolt listák különböző változatai ismertek:

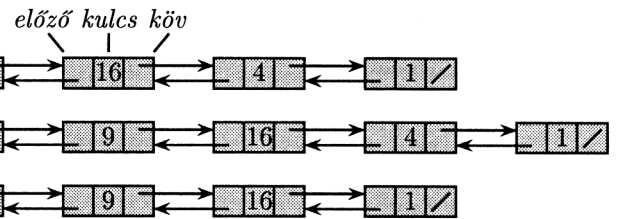
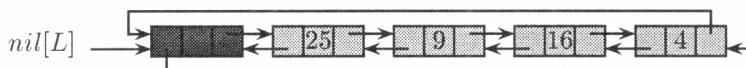
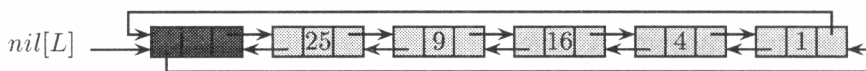
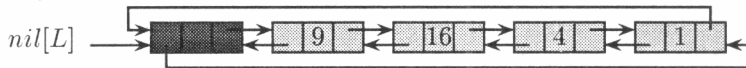
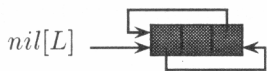
1. kétszeresen láncolt lista (lásd ábra):

elemei tartalmazzák egyrészt a tárolt adatot (kulcsmező), másrészt

két-két pointert. Ezek közül az egyik az előző, a másik pedig a következő elemre mutat. (Egyéb

mezők is lehetnek, de ezek minimalisan szükségesek.) Ha az "előző" mezőben NIL szerepel, akkor ő a lista feje; ha a "következő" mezőben van NIL, akkor pedig a vége. (NIL-t az ábrán slash (/) jelöli). A fej[L] pointer a lista elejére mutat – ha értéke NIL, akkor a lista üres.

2. egyszeresen láncolt lista: az előzőtől abban különbözik, hogy csak a "következő" nevű pointert tartalmazza.
3. cyclicus lista: a legelső elem "előző" pointere a lista utolsó elemére, a legutolsó elem "következő" pointere pedig a lista első elemére mutat.

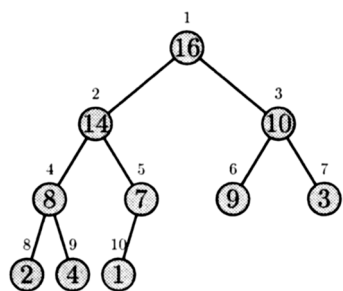


Rendezett listában a pointerok által meghatározott sorrend = az elemekben tárolt kulcsmezők növekvő sorrendjével – ellenkező esetben a lista rendezetlen. Új elem hozzáadása láncolt listához (ábra második sora) történhet a lista elején, illetve a végén is. Értelemszerűen a megfelelő pointerokat kell módosítani. Ugyanez igaz a törlésre (ábra harmadik sora) is. A másik ábrásor az előző kétszeresen láncolt listát mutatja, melyet egy speciális elem (sötétebb) segítségével cyclicus listává alakítottunk. A 3. sorban új elem hozzáadása, a 4. sorban pedig törlés látható.

⁵ A számozás indulhat 0-tól is, ez ízlés dolga – azaz inkább megvalósításfüggő...

Binaris kupac

Egy majdnem teljes binaris fának (vagyis minden csúcsnak max. 2 utódja lehet) fogható fel – akkor lenne teljes, ha minden nem levélcsúcsnak (pl. 5. számú) két leszármazottja lenne.



A fa minden csúcsa egy-egy tömbelem, lásd ábra. A fa minden szintje (kivéve a

| | | | | | | | | | |
|----|----|----|---|---|---|---|---|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

legalsó) teljesen kitöltött. Az A tömb tárolja az elemeket, e tömbnek két adata fontos: a $\text{hossz}[A]$ megmutatja a tömb teljes méretét, a $\text{kupacméret}[A]$ pedig azt, hogy ebből ténylegesen hány elem vesz

részt a kupac felépítésében (értelemszerűen $\text{hossz}[A] \geq \text{kupacméret}[A]$). A gyökérelemet $A[1]$ jelöli.

Mivel a fa binaris, rendkívül egyszerűen meghatározhatók a következő fontos adatok:

- egy adott indexű (i) elem szülőjének indexe: $\lfloor i/2 \rfloor$ ⁶
- egy adott indexű (i) elem bal oldali gyerekenek indexe: $2i$
- egy adott indexű (i) elem jobb oldali gyerekenek indexe: $2i+1$

A bal oldali gyerek meghatározásához egy register tartalmát úgy toljuk balra (arithmeticai SHIFT), hogy a legalacsonyabb helyiértékű bitre 0 lép be. A jobb gyereknél ugyanez, de 1 lép be. A szülő indexét pedig jobbra tolással kapjuk, a legmagasabb helyiértékre 0 lép be, a kilépő bittel pedig nem foglalkozunk. Ezeket a műveleteket C-ben is könnyen megvalósíthatjuk, a \gg és \ll operatorok segítségével.

Ezek a megvalósítások azért lényegesek, mert pl. a kupacok egy lehetséges felhasználási területe az elsőbbségi sorok, amelyek osztott működésű számítógépeken a feladatok ütemezéséhez használhatók.

Kupactulajdonság: $A[\text{szülő}(i)] \geq A[i]$, vagyis a kupac minden csúcsára igaz, hogy a szülőcsúcsban lévő érték nem kisebb, mint az adott csúcsban lévő érték (kivétel természetesen a gyökércsúcs, hiszen annak nincs szülője).

A fa egy csúcsának magassága egyenlő azon út éleinek számával, amely az adott elemből egy tőle legtávolabb lévő levél felé vezet. A fa magassága pedig a gyökérelem magasságával azonos. A kupacon végzett alapl műveletek végrehajtási ideje a fa magasságával, azaz $O(\log n)$ -nel arányos.

A kupacokra 3 algoritmus ismerete fontos: a kupactulajdonság fenntartása (KUPACOL), a kupac építése és rendezése.

KUPACOL (A, i)

Az eljárás argumentumként egy tömböt kap (mely a kupac elemeit tartalmazza), valamint egy indexet. Ez utóbbi

```
1  $l \leftarrow \text{BAL}(i)$ 
2  $r \leftarrow \text{JOBB}(i)$ 
3 if  $l \leq \text{kupac-méret}[A]$  és  $A[l] > A[i]$ 
4   then  $\text{legnagyobb} \leftarrow l$ 
5   else  $\text{legnagyobb} \leftarrow i$ 
6 if  $r \leq \text{kupac-méret}[A]$  és  $A[r] > A[\text{legnagyobb}]$ 
7   then  $\text{legnagyobb} \leftarrow r$ 
8 if  $\text{legnagyobb} \neq i$ 
9   then  $A[i] \leftrightarrow A[\text{legnagyobb}]$ 
10  KUPACOL( $A, \text{legnagyobb}$ )
```

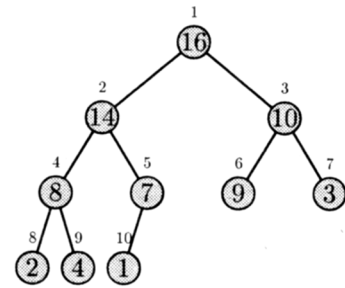
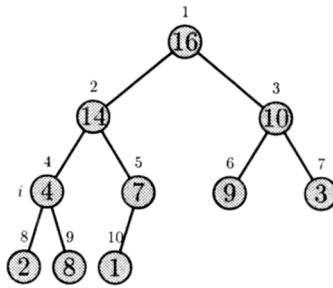
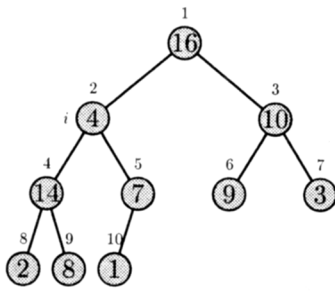
annak a csúcsnak az indexe, amelyikről feltételzzük, hogy mindkét oldali leszármazottja (azaz a részfa) kupac szerkezetűek, de maga $A[i]$ megsérti a kupactulajdonságot (aza kisebb lehet gyerekeinél). Az eljárás addig mozgatja lefelé az $A[i]$ értéket, amíg a i gyökerű részfa kupaccá nem alakul. Az alg.-ban először (1–2.) meghatározzuk az i csúcs bal és jobb leszármazottjának indexét (lásd előbb). Ha a bal oldali részfa csúcsa az indexe alapján benne van a kupacban és értéke az i -nél nagyobb (3.), akkor a "legnagyobb" értékben az i indexét tároljuk (4.). Ellenkező esetben az i indexe lesz a legnagyobb (5.)

– ez utóbbi eset azt jelenti, hogy vagy valóban az i indexű elem a nagyobb a kettő közül, vagy pedig nem létezik bal oldalsó elem (ekkor az i indexű elem nyilván levél).

Ugyanezt megvizsgáljuk a jobb oldali részfa csúcsára is (6–7.), de itt értelemszerűen a "legnagyobb"-bal hasonlítunk össze. Végül ha a "legnagyobb" nem azonos i -vel, akkor az $A[i]$ -t és az $A[\text{legnagyobb}]$ -at felcseréljük (9.). Végül recursiv módon meghívjuk az eljárást (10.) úgy, hogy az átadott index a legnagyobb elemhez tartozik (nyilván i fogja csak sérteni a kupactulajdonságot).

⁶ $A[x]$ jelölés (ún. floor) azt a legnagyobb egész számot jelenti, amely x -nél nem nagyobb. Párja $\lceil x \rceil$ (ceiling): az a legkisebb egész szám, amely x -nél nem kisebb.

⁷ $\log_2 x$: kettes alapú logaritmus (log. binaris), azaz $\log_2 x$



Az ábra mutatja az eljárást működés közben. Az első ábrán látható, hogy a 2. elem sérti a kupactulajdonságot. A lényeg, hogy ezt addig "görgetjük lefelé" a kupacban, amíg el nem éri végső helyét.

Fontos, hogy levélelemre nem lehet kupacolni (nincs jobb és bal oldali leszármazott, tehát a levélelem nem sértheti a kupactulajdonságot). Az alg. futási idejére azt mondhatjuk, hogy $O(\lg n)$ -nel arányos, ahol 'n' a csúcsponatok száma. Másként megfogalmazva $O(h)$, ahol 'h' jelöli a fa magasságát.

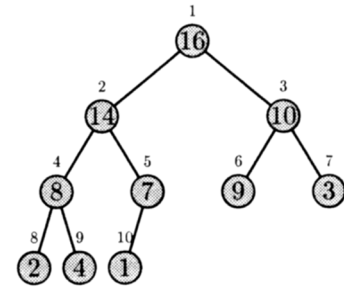
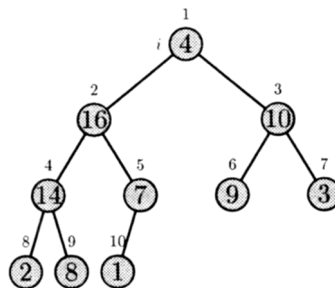
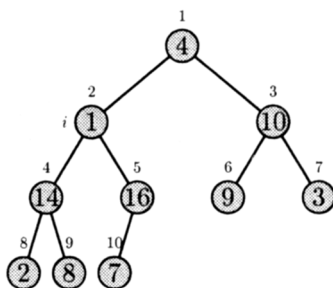
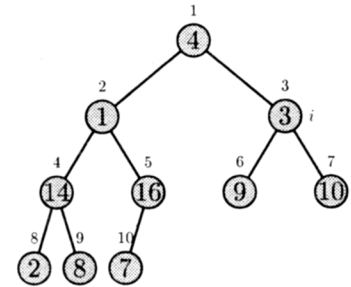
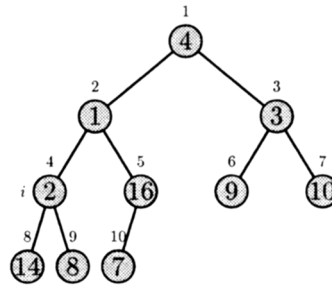
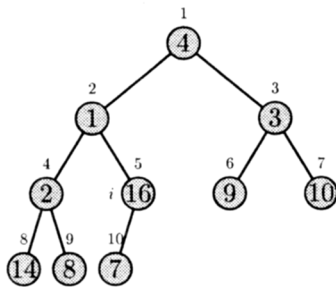
KUPACOT-ÉPÍT (A)

Az eljárás argumentumként szintén egy tömböt (vectort) kap, melyben a kupacba állítandó elemek találhatóak. Az alg. először meghatározza a tömb méretét, ez egyben a kupac méretét is jelenti (1). Ezután egy cyclus indul, amely a legutolsó levélelem szülőjétől – ezt adja meg a $\lfloor \text{hossz}[A] / 2 \rfloor$ kifejezés – a gyökerécsúcs felé haladva meghívja

- 1 $\text{kupac-méret}[A] \leftarrow \text{hossz}[A]$ a KUPACOL alg.-t. Az ábra mutatja az A jelű kiinduló tömböt, az ebből
- 2 **for** $i \leftarrow \lfloor \text{hossz}[A] / 2 \rfloor$ **downto** 1 először létrehozott kupacot, illetve a rendezés egyes lépéseit. Egy rendezetlen
- 3 **do** KUPACOL(A, i) tömb lineáris idő, tehát $O(n)$ alatt kupaccá alakítható.

A

| | | | | | | | | | |
|---|---|---|---|----|---|----|----|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|----|---|----|----|---|---|



KUPACRENDEZÉS (A)

A legelső ábrán látható, hogy egy kupactulajdonsággal rendelkező tömb elemei csökkenő sorban vannak. A

- 1 KUPACOT-ÉPÍT(A) kupacrendezés alapelve, hogy az adatokból először kupacot
 - 2 **for** $i \leftarrow \text{hossz}[A]$ **downto** 2 építünk (1.), majd egy olyan cyclust indítunk (2.), amely a
 - 3 **do** $\text{csere}[A[1] \leftrightarrow A[i]]$ legutolsó elemtől az utolsó előttiig tart. Kicseréljük a kupac első
 - 4 $\text{kupac-méret}[A] \leftarrow \text{kupac-méret}[A] - 1$ elemét a cyclusváltozó által mutatott elemmel (3.), majd eggyel
 - 5 KUPACOL(A, 1) csökkentjük a kupac méretét (4.) – hiszen a legutolsó elem már a
- helyére került. Ezután meghívjuk a KUPACOL eljárást, mely újra helyreállítja a kupactulajdonságot. Azért fut csak 2-ig a cyclus, mert a 2. elemre elvégezve a cserét a legutolsó elem már magától a helyére került.

Futási idő: $O(n * \lg n)$ – a kupac felépítése $O(n)$ ideig tart, a KUPACOL pedig $(n-1)$ alkalommal fut, egyenként $O(\lg n)$ ideig tart.

