Algoritmusok és adatszerkezetek I. előadásjegyzet

Ásványi Tibor – asvanyi@inf.elte.hu

2025. március 17.

Tartalomjegyzék

1. Bevezetés

Az itt következő előadásjegyzetekben, az előadásokon tárgyalt programok struktogramjait igyekeztem minden esetben megadni, a másolási hibák kiküszöbölése érdekében. A magyarázatok gyakran részletesebbek, mint az előadáson, esetleg a tárgyalt algoritmusokat, adatszerkezeteket más oldalról világítják meg.

Ezúton szeretnék köszönetet mondani *Umann Kristófnak* és *Varga Henrik Zoltánnak* az ebben a jegyzetben található szép, színvonalas szemléltető ábrák elkészítéséért, az ezekre szánt időért és szellemi ráfordításért!

A vizsgára való készülésben elsősorban az előadásokon és a gyakorlatokon készített jegyzeteikre támaszkodhatnak. További ajánlott források:

Hivatkozások

[1] Ásványi Tibor, Algoritmusok és adatszerkezetek I. előadásjegyzet (2024, Canvas Modulok)

[2] Fekete István, Algoritmusok jegyzet
http://ifekete.web.elte.hu/

[3] Rónyai Lajos – Ivanyos Gábor – Szabó Réka, Algoritmusok, *TypoTEX Kiadó*, 1999. ISBN 963 9132 16 0

[4] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
magyarul: Új Algoritmusok, *Scolar Kiadó*, Budapest, 2003.
ISBN: 963 9193 90 9
angolul: Introduction to Algorithms (Fourth Edititon),
The MIT Press, 2022.

[5] Wirth, N., Algorithms and Data Structures,
Prentice-Hall Inc., 1976, 1985, 2004.
magyarul: Algoritmusok + Adatstruktúrák = Programok, *Műszaki Könyvkiadó*, Budapest, 1982. ISBN 963 10 3858 0

[6] Weiss, Mark Allen, Data Structures and Algorithm Analysis, *Addison-Wesley*, 1995, 1997, 2007, 2012, 2013.

Saját jegyzeteiken kívül elsősorban az ebben a jegyzetben [1], illetve az itt hivatkozott helyeken [2, 4, 5] leírtakra támaszkodhatnak. A CLRS könyv [4], valamint Rónyai [3], Wirth [5] és Weiss [6] klasszikus munkáinak megfelelő fejezetei értékes segítséget nyújthatnak a mélyebb megértéshez. Ennek a jegyzetnek a „*”-gal jelölt alfejezetei szintén a mélyebb megértést szolgálják, azaz nem részei a vizsga anyagának.

Az angol nyelvű szakirodalom jelentős része letölthető a *Library Genesis* honlapról

A vizsgákon az elméleti kérdések egy-egy tétel bizonyos részleteire vonatkoznak. Lesznek még megoldandó feladatok, amelyekhez hasonlók az ebben a jegyzetben találhatókhoz.

2. Tematika

Minden tételhez: Egy algoritmus, program, művelet bemutatásának mindig része a műveletigény elemzése. Hivatkozások: például a „[4] 2, 7" jelentése: a [4] sorszámú szakirodalom adott fejezetei.

1. Az algoritmus fogalma, programok hatékonysága: Intuitív bevezetés. Példa: beszúró rendezés (insertion sort) ([1]; [2]; [4] 1-3.)

2. Az oszd meg és uralkodj elv, összefésülő (összefuttató) rendezés (merge sort), gyorsrendezés (quicksort) ([1]; [4] 2, 7; [2]).

3. Az adatszerkezet és az adattípus fogalma. Elemi adattárolók: vermek (gyakorlat), sorok ([1]; [2]; [4] 10.1), megvalósításuk tömbös és láncolt reprezentációk esetén (láncolt listás megvalósítás a gyakorlatokon). Vermek felhasználása.

4. Elemi, lineáris adatszerkezetek: tömbök, láncolt listák, láncolt listák típusai, listakezelés. ([1]; [2]; [4] 10; [5] 4.1 - 4.3)

5. Függvények aszimptotikus viselkedése ($O, o, \Omega, \omega, \Theta, \prec, \succ$) . Programok műveletigénye (futási idő nagyságrendje: $T(n), mT(n), AT(n), MT(n)$) ([1]; [2]; [4] 1-3.)

6. Fák, bináris fák, bejárások, láncolt reprezentáció, példák ([1]; [2]; [4] 10.4, 12.1).

7. Bináris keresőfák és műveleteik, bináris rendezőfák ([1]; [2]; [4] 12; [5] 4.4).

8. (Majdnem) teljes bináris fák, aritmetikai ábrázolás, prioritásos sorok, kupac, kupac műveletei, kupacrendezés (heap sort) ([1]; [2]; [4] 6).

9. A beszúró, összefésülő, kupac és gyors rendezés összehasonlítása. Az összehasonlító rendezések alsókorlát-elemzése ([1]; [4] 8.1; [2]).

10. Rendezés lineáris időben [1], [4] 8.2. A stabil rendezés fogalma. Leszámláló rendezés (Counting-Sort). Radix rendezés (Radix-Sort) tömbökre ([4] 8.3) és láncolt listákra ([1, 2]). Edényrendezés (bucket sort [1], [4] 8.4, [2]).

11. Hasító táblák [1], [4] 11. Direkt címzés (direct-address tables). Hasító táblák (hash tables). A hasító függvény fogalma (hash functions). Kulcsütközések (collisions).

Kulcsütközések feloldása láncolással (collision resolution by chaining); keresés, beszúrás, törlés (search and update operations); kitöltöttségi arány (load factor); egyszerű egyenletes hasítás (simple uniform hashing), műveletigények.

Jó hash függvények (good hash functions), egy egyszerű hash függvény (kulcsok a $[0;1)$ intervallumon), az osztó módszer (the division method), a szorzó módszer (the multiplication method).

Nyílt címzés (open addressing); próbasorozat (probe sequence); keresés, beszúrás, törlés (search and update operations); üres és törölt rések (empty and deleted slots); a lineáris próba, elsődleges csomósodás (linear probing, primary clustering); négyzetes próba, másodlagos csomósodás (quadratic probing, secondary clustering); kettős hash-elés (double hashing); az egyenletes hasítás (uniform hashing) fogalma; a keresés és a beszúrás próbasorozata várható hosszának felső becslései egyenletes hasítást feltételezve.

3. Néhány alapvető jelölés és ezek jelentése

$false = 0; true = 1$.
$\mathbb{B} = \{false; true\}$ a logikai (boolean) értékek halmaza.
$\mathbb{N} = \{0; 1; 2; 3; \dots\}$ a természetes számok halmaza.
$\mathbb{N}_+ = \{1; 2; 3; \dots\}$ a pozitív természetes számok halmaza.
$\mathbb{Z} = \{\cdots - 3; -2, -1; 0; 1; 2; 3; \dots\}$ az egész számok halmaza.
$\mathbb{R}$ a valós számok halmaza.
$\mathbb{P}$ a pozitív valós számok halmaza.
$\mathbb{P}_0$ a nemnegatív valós számok halmaza.

$$\log n = \begin{cases} \log_2 n & \text{ha} \quad n > 0 \\ 0 & \text{ha} \quad n = 0 \end{cases}$$

$fele(n) = \left\lfloor \frac{n}{2} \right\rfloor$, ahol $n \in \mathbb{N}$
$Fele(n) = \left\lceil \frac{n}{2} \right\rceil$, ahol $n \in \mathbb{N}$

$$\left.\begin{array}{l} [u\,..\,v] = \{i \in \mathbb{Z} : u \le i \le v\} \\ [u\,..\,v) = \{i \in \mathbb{Z} : u \le i < v\} \end{array}\right\} \text{ahol } u, v \in \mathbb{Z}.$$

Az $A[u\,..\,v]$ résztömb az $\langle A[u], A[u+1], \dots, A[v]\rangle$ sorozatot reprezentálja. Ha $u > v$, akkor a résztömb és a reprezentált sorozat üres.

Az $A[u\,..\,v)$ résztömb az $\langle A[u], A[u+1], \dots, A[v-1]\rangle$ sorozatot reprezentálja. Ha $u \ge v$, akkor a résztömb és a reprezentált sorozat üres.

A képletekben és a struktogramokban *alapértelmezésben* (tehát ha a környezetből nem következik más) az $i, j, k, l, m, n, I, J, K, M, N$ betűk egész számokat (illetve ilyen típusú változókat), míg a p, q, r, s, t, H, L betűk pointereket (azaz mutatókat, memóriacímeket, illetve ilyen típusú változókat) jelölnek.

A $\mathcal{T}$ *alapértelmezésben* olyan ismert (de közelebbről meg nem nevezett) típust jelöl, amelyen értékadás (pl. $x := y$) és általában teljes rendezés (az $=, \neq, <, >, \leq, \geq$ összehasonlításokkal) van értelmezve.

Az operátorok jelentése, prioritása és módja (prefix, szuffix, illetve infix és ezen belül jobbról balra vagy balról jobbra zárójelező) két kivétellel a C nyelvben megszokott. A kivételek: a C "=" operátorának itt a ":=", míg a C "==" operátorának itt az "=" operátor felel meg, a matematikában megszokott módon.

A skalár (elemi) típusú változókat (pl. egész, lebegőpontos, karakter, logikai, pointer) típusát általában a rájuk vonatkozó első értékadó utasítás határozza meg. A strukturált (tömb, tömbhivatkozás, objebtum) változókat minden esetben deklaráljuk.

A változók láthatósága és hatásköre is az őket tartalmazó struktogram, élettartamuk pedig az első, őket tartalmazó utasítás végrehajtásától a struktogram befejezéséig tart. Kivételt képeznek a globális változók, amelyek a program egész végrehajtása alatt élnek és láthatók is. A globális változókat minden esetben deklarálni kell, a **global** prefix segítségével. A változók lokálisan nem definiálhatók felül.

3.1. Tömb, tömbhivatkozás és tömbtároló (Array, Array reference and Array storage)

A mi modellünkben tetszőleges tömb (array) egy ún. tömbhivatkozásból (array reference) és egy tömbtárolóból (array storage) áll. A tömbhivatkozás két komponensű. Ezek egyike a tömb hossza (length), a tömb elemeinek a száma. A másik egy mutató (pointer), ami a tömbtárolóra hivatkozik. A tömbtároló a tömb elemeit tartalmazó memóriacelláknak a memóriában folytonos sorozata.

Az $A : \mathcal{T}[n]$ tömbdeklaráció kiértékelése létrehoz egy új, $\mathcal{T}$ elemtípusú, n elemű tömbtárolót ($n \in \mathbb{N}$), és az $A : \mathcal{T}[]$ típusú tömbhivatkozást. Ez utóbbi hosszmezőjének értéke n, pointer mezője pedig a tömbtárolóra mutat. A deklarációt tartalmazó programblokk befejeződésekor a tömbhivatkozással együtt a tömbtároló is automatikusan törlődik.

A tömb tehát „ismeri" a saját méretét. A fenti A tömb mérete pl. $A.length$, ami nem változtatható meg. (Erre az A tömbre így $A.length = n$.) Az $A.pointer$ mutató a tömb első elemére hivatkozik.

A tömböket alapértelmezésben 0-tól indexeljük. A fenti A tömbre tehát $*A.pointer = A[0]$, és $*(A.pointer + i) = A[i]$, ahol $i \in [0\,..\,n)$. (Itt a $[0\,..\,n)$ balról zárt, jobbról nyílt egész intervallum.) A tömb az $\langle A[0], \ldots A[n-1] \rangle$ sorozatot reprezentálja, ha $n > 0$, különben az üres, $\langle\rangle$ sorozatot.

Ha a tömböt nem nullától szeretnénk indexelni, azt pl. a $B/k : \mathcal{T}[n]$ deklarációval érhetjük el. Ekkor a B tömb hossza szintén n, de k-tól indexeljük, ahol $k \in \mathbb{Z}$. Erre a B tömbre így $B[k] = *B.pointer$, és $B[i] = *(B.pointer + i - k)$, ahol $i \in [k \,..\, k + n)$.

Ha csak egy $\mathcal{T}$ elemtípusú tömbhivatkozást szeretnénk deklarálni, ezt pl. a $P : \mathcal{T}[]$ vagy a $Q/k : \mathcal{T}[]$ deklarációs utasítással tehetjük meg.

Ezután a P hivatkozás inicializálható pl. a $P := A$ értékadó utasítással, miután $P.pointer$ is az $A.pointer$ által hivatkozott tömbtárolóra mutat, és $P.length = A.length$, valamint $A[0]$-nak $P[0]$, ..., $A[n-1]$-nek $P[n-1]$ felel meg.

Ezután a $Q := P$ értékadás hatására $Q.length = P.length = A.length$, valamint $Q[k] = P[0] = A[0]$, $Q[k+1] = P[1] = A[1]$, ..., $Q[k+n-1] = P[n-1] = A[n-1]$, sőt ezek a cellahármasok nem csak egyenlők, hanem azonosak is, így bármelyik cellahármas egyikét megváltoztatva a másik kettő is vele változik.

Pl. ha most végrehajtjuk a $P[1] := 5$ értékadó utasítást (ebben a bekezdésben feltéve, hogy $\mathcal{T} = \mathbb{Z}$ és $n > 1$), akkor $Q[k+1] = A[1] = 5$ is igaz lesz, hiszen $Q[k+1]$, $P[1]$ és $A[1]$ ugyanazt a memóriacellát jelölik.

3.1.1. Dinamikus tömb

Új tömböt dinamikusan pl. a **new** $\mathcal{T}[n]$ kifejezéssel hozhatunk létre, ami egy n elemű, $\mathcal{T}$ elemtípusú tömbtárolót hoz létre, majd a hosszát és a címét visszaadja. Az egyszerűség kevéért, alapértelmezésben feltesszük, hogy van elég szabad memória a **new** utasításaink számára.

A $P :=$ **new** $\mathcal{T}[n]$ utasítás hatására pl. a fent deklarált P tömbhivatkozás egy új tömbre fog hivatkozni, amelynek elemei sorban $P[0], \ldots P[n-1]$; mérete pedig $P.length = n$. Hasonlóan, a $Q :=$ **new** $\mathcal{T}[m]$ utasítás hatására a fenti Q tömbhivatkozás egy, az előzőtől különböző, új tömbre fog hivatkozni, amelynek elemei sorban $Q[k], \ldots Q[k+m-1]$; mérete pedig $Q.length = m$. Az $R : \mathcal{T}[] := P$, deklaráció+kezdeti értékadás hatására pedig R is a P által hivatkozott tömbre hivatkozik, így ugyanaz a tömb a P és az R tömbhivatkozásokon keresztül is olvasható és módosítható.

A mi (C/C++-hoz hasonló) modellünkben, a *memóriaszivárgás* elkerülése érdekében, a dinamikusan (**new** utasítással) létrehozott, és már feleslegessé vált objektumokat expliciten törölni kell. Ezzel ugyanis az objektum által lefoglalt memóriaterület újra felhasználhatóvá, míg ellentétes esetben az alkalmazás számára elérhetetlenné válik. Az ilyen memóriadarabok felhalmozódása jelentősen csökkentheti a programunk által használható memóriát, abban szemetet képez. (A JAVA, C#, Golang és más hasonló környezetek a memóriaszemét kezelésére automatikus eszközöket biztosítanak, de ennek a

hatékonyság oldaláról nézve súlyos ára van. Mivel a mi algoritmusaink egyik legfontosabb akalmazási területe a rendszerprogramozás, mi ilyen automatizmusokat nem feltételezünk.)

A **new** utasítással létrehozott objektumok törlésére a **delete** utasítás szolgál, pl.

delete P; **delete** Q;

amik törlik a P és a Q tömbhivatkozások által hivatkozott tömbtárolókat, de nem törlik a hivatkozásokat magukat. A fenti törlések hatására a P, az R és a Q hivatkozások nemdefiniáltakká válnak, és később újra érteket kaphatnak, de csak $\mathcal{T}$ elemtípusú tömbtárolókra hivatkozhatnak.

3.1.2. Résztömb és tömbhivatkozás

A tömbhivatkozások résztömböket is azonosíthatnak.

Tegyük fel például, hogy adott az $A : \mathbb{Z}[5]$ tömb, ami a $\langle 2, 3, 5, 7, 11\rangle$ sorozatot reprezentálja. Adott továbbá a $P : \mathbb{Z}[]$ tömbhivatkozás, és végrehajtjuk a $P := A[1\,..\,4)$ értékadást. Ezután a P az $A[1\,..\,3]$ résztömbre, azaz az A tömb középső három elemére hivatkozik, és a $\langle 3, 5, 7\rangle$ sorozatot reprezentálja.

Ha most végrehajtjuk a $P[1] := 19$ értékadást, ezzel $A[2] = 19$ is igaz lesz, hiszen $P[1]$ és $A[2]$ ugyanazt a memóriacellát jelölik. Ekkor A a $\langle 2, 3, 19, 7, 11\rangle$, P pedig a $\langle 3, 19, 7\rangle$ sorozatnak felel meg.

Ezután a $P := P[1\,..\,P.length)$ utasítással levágjuk a P által hivatkozott résztömb első elemét, azaz a P által reprezentált sorozat a $\langle 19, 7\rangle$ lesz. Ez az utasítás az A tömböt nem módosítja, csak a 3 értékű eleme kikerül a P tömbhivatkozás „látóköréből”.

3.2. Szögletes zárójelek közé írt utasítások

A struktogramokban néha szerepelnek szögletes zárójelek közé írt utasítások. Ez azt jelenti, hogy a bezárójelezett utasítás bizonyos programozási környezetekben szükséges lehet. Ha tehát a program megbízhatósága és módosíthatósága a legfontosabb szempont, ezek az utasítások is szükségesek. Ha a végletekig kívánunk optimalizálni, akkor bizonyos esetekben elhagyhatók.

3.3. A struktogramok paraméterlistái, érték szerinti és cím szerinti paraméterátadás

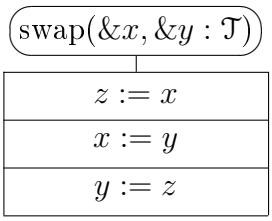
A továbbiakban az eljárásokat, függvényeket és az osztályok metódusait együtt *alprogramoknak* nevezzük. Az *eljárás* szó a C/Java/C# terminológia *void függvény* fogalmának felel meg, míg a *függvény* alatt sosem *eljárást* értünk.

A struktogramokhoz mindig tartozik egy alprogramnév és általában egy paraméterlista is (ami esetleg üres, de a „()” zárójelpárt ott is, és a megfelelő alprogramhívásban is kiírjuk). *Ha egy struktogramhoz csak név tartozik, akkor az úgy értendő, mintha a benne található kód a hívás helyén lenne.* A paraméterek típusát és – függvények esetén a visszatérési érték típusát – az UML dobozokban szokásos módon jelöljük.

Ha egy paraméterlistával ellátott alprogram struktogramjában olyan változónév szerepel, ami a paraméterlistán nem szerepel, és nem is az alprogram külső (azaz globális) változója, akkor ez a struktogrammal leírt alprogram lokális változója.

A skalár típusú[1] paramétereket *alapértelmezésben* érték szerint vesszük át.[2]

A skalár típusú paramétereket cím szerint is átvehetjük, de akkor ezt a formális paraméter listán a paraméter neve előtt egy & jellel jelölni kell. Pl. az alábbi eljárást a swap(a, b) utasítással meghíva, a és b értéke felcserélődik.



A cím szerinti paraméterátadás esetén ugyanis, az alprogramhíváskor az aktuális paraméter összekapcsolódik a megfelelő formális paraméterrel, egészen a hívott eljárás futásának végéig, ami azt jelenti, hogy bármelyik megváltozik, vele összhangban változik a másik is. Amikor tehát a formális paraméter értéket kap, az aktuális paraméter is ennek megfelelően változik. Ha az eljárásfej swap$(x, \&y : \mathcal{T})$ lenne, az eljáráshívás hatása „$b := a$” lenne, ha pedig az eljárásfej swap$(x, y : \mathcal{T})$ lenne, az eljáráshívás logikailag ekvivalens lenne a SKIP utasítással.

A formális paraméter listán a felesleges &-prefixek hibának tekintendők, mert a cím szerint átadott skalár paraméterek kezelése az eljárás futása során a legtöbb implementációban lassúbb, mint az érték szerint átadott paramétereké.

Az aktuális paraméter listán nem jelöljük külön a cím szerinti paraméterátadást, mert a formális paraméter listáról kiderül, hogy egy tetszőleges

[1] A skalár típusok az egyszerű típusok: a szám, a pointer és a felsorolás (pl. a logikai és a karakter) típusok.

[2] Az érték szerinti paraméterátadás esetén, az alprogramhíváskor az aktuális paraméter értékül adódik a formális paraméternek, ami a továbbiakban úgy viselkedik, mint egy lokális változó, és ha értéket kap, ennek nincs hatása az aktuális paraméterre.

paramétert érték vagy cím szerint kell-e átadni. (Összhangban a C++ jelölésekkel.) Pl. a swap eljárás egy lehetséges meghívása: swap($A[i], A[j]$).

Ha az alprogramokra szövegben hivatkozunk, a paraméterátadás módját – néhány kivételes esettől eltekintve – szintén nem jelöljük. (Pl.: „A swap(x, y) eljárás megcseréli az x és az y paraméterek értékét.")

A nem-skalár[3] típusú paraméterek csak cím szerint adhatók át. (Pl. a tömböket, rekordokat, objektumokat nem szeretnénk a paraméterátvételkor lemásolni.) A nem-skalár típusok esetén ezért általában nem jelöljük a paraméterátadás módját, hiszen az egyértelmű.

3.4. Tömb típusú paraméterek a struktogramokban

A tömb és a résztömb aktuális paramétereknek a formális paraméter listákon tömbhivatkozások felelnek meg. Az aktuális és a formális paraméterek elemtípusa azonos kell legyen.

Ha egy alprogram egy formális paramétere az F tömbhivatkozás, aminek az alprogramhívásban az A (rész)tömböt leíró kifejezés felel meg, akkor a paraméterátvételt a híváskor automatikusan végrehajtott $F := A$ értékadó utasítás valósítja meg. Így *F.length* az A (rész)tömb hossza lesz, míg *F.pointer* a megfelelő tömbtároló (megfelelő részének) címét tartalmazza.

Így az aktuális paraméter (rész)tömb végső soron cím szerint adódik át. Eszerint, ha a hívott alprogram futása során, a formális paraméter által hivatkozott (rész)tömböt megváltoztatjuk, ez az aktuális paraméter által hivatkozott (rész)tömbbel is azonnal megtörténik.

Az alprogram fejében az F formális paraméter tömbhivatkozás specifikálása során megadhatunk a szögletes zárójelek között egy azonosítót is, mint az alábbi példában. Ekkor ez az azonosító az *F.length* kifejezés szinonímája. Az alprogramban csak konstansként használhatjuk.

Pl. a linearSearch($F : \mathcal{T}[n]$; $x : \mathcal{T}$) : $\mathbb{N}$ függvényfej olyan függvényre utalhat, ami az F (rész)tömbben megkeresi az x első előfordulását, és visszaadja annak indexét; vagy n-et, ha $x \notin \{F[0], \ldots, F[n-1]\}$.

Ebben a példában a (rész)tömböt a függvényen belül nullától indexeljük, és n egy rövidítő jelölés a (rész)tömb *F.length* hosszára. A formális és az aktuális paraméter kezdőindexe természetesen különbözhet, de a formális paraméter által hivatkozott (rész)tömb ugyanaz, mint az aktuális paraméter által hivatkozott.

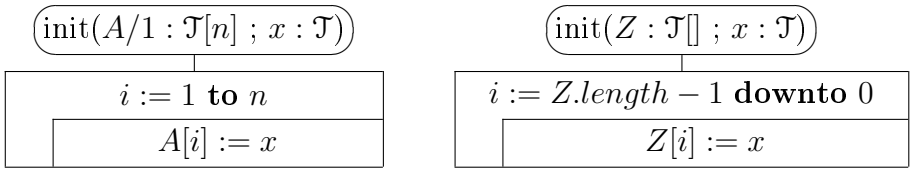
[3] A nem-skalár típusok az összetett típusok. Pl. a tömb, sztring, rekord, fájl, halmaz, zsák, sorozat, fa és gráf típusok, valamint a tipikusan `struct`, illetve `class` kulcsszavakkal definiált osztályok objektumai.

Hasonlóan, pl. a sort($B/1 : \mathcal{T}[]$) eljárásfej olyan eljárásra utalhat, ami (pl. monoton növekvően) rendezi a B tömböt. Ezen az eljáráson belül a tömböt egytől indexeljük.

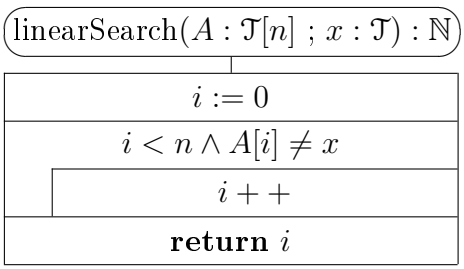
3.5. Eljárások, függvények, ciklusok, rekurzió

Először egy egyszerű eljárást nézünk meg két változatban, ami egy tetszőleges egy dimenziós tömb elemeit ugyanazzal az értékkel inicializálja. Mindkét esetben a Pascal programozási nyelvből esetleg már ismerős léptető ciklust alkalmazunk. Annyi a különbség, hogy az első esetben sorban haladunk az elemeken, míg a másodikban sorban visszafelé, és az A tömb 1-től, míg a Z nullától indexelődik.

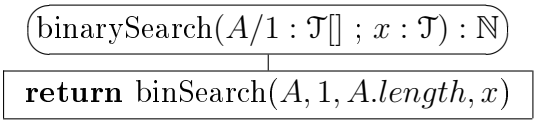
Vegyük észre, hogy az init eljárás két változata a tömbök és a paraméterátvétel tulajdonságai miatt ekvivalens.



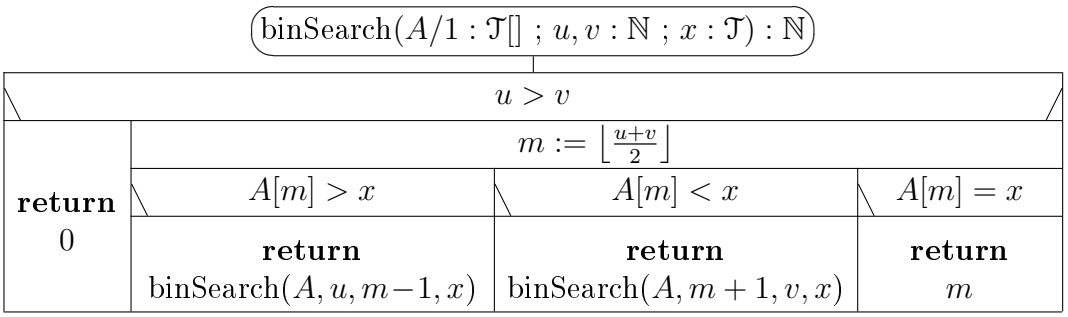
Ebben a jegyzetben megkülönböztetjük az eljárás és a függvény fogalmát. Az eljárások a környezetükkel csak a paramétereiken (és esetleg külső változókon) keresztül kommunikálnak, míg a függvényeknek visszatérési értékük is van, amit a szokásos módon használhatunk fel. (A mi *eljárás* fogalmunknak a C programozási nyelvben és leszármazottaiban a *void function* felel meg.) Alább láthatunk példákat függvényekre. A linearSearch(A, x) függvényhívás az A tömbben megkeresi az x első előfordulását, és visszaadja annak indexét; vagy n-et, ha $x \notin \{A[0], \ldots, A[n-1]\}$.



A binarySearch(A, x) függvényhívás az A monoton növekvően rendezett tömbben megkeresi az x egyik előfordulását, és visszaadja annak indexét; vagy nullát, ha $x \notin \{A[1], \ldots, A[A.length]\}$.



A fenti binarySearch(A, x) függvény, megfelelően paraméterezve meghívja az alábbi binSearch(A, u, v, x) függvényt, ami az $A[u\,..\,v]$ résztömbön keresi x-et (Ezt az állítást hamarosan igazoljuk.) A fenti paraméterezéssel tehát az egész tömbön keresi az x értéket. Így az alábbi függvény a fenti általánosítása.



Formailag a binSearch(A, u, v, x) rekurzív függvény, mivel van olyan programága, ahol önmagát hívja (amit rekurzív hívásnak nevezünk). A számítógép minden alprogram hívásra ugyanúgy, az alprogram hívások lokális adatait a *call stack*-ben tárolja, tehát a rekurzív hívásokat is ugyanúgy kezeli, mint a nemrekurzívakat: tetszőleges alprogram lokális adatainak akár több példánya is lehet a *call stack*-ben. Ez tehát önmagában nem okoz technikai nehézséget. A rekurzív alprogramoknál azonban gondoskodnunk kell a rekurzió megfelelő leállításáról, hiszen az alprogram elvileg a végtelenségig hívogathatja önmagát. Ezt szolgálják a rekurzív alprogramokban az alább ismertetetendő ún. *leálló ágak*, amiket a bemenő adatok közül az ún. *alapesetek* aktiválnak.

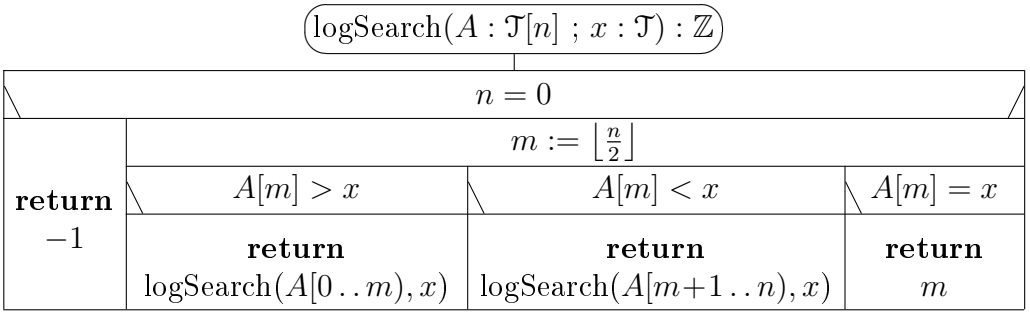
Most igazoljuk, hogy a binSearch(A, u, v, x) függvény visszaad egy $k \in u\,..\,v$ indexet, amelyre $A[k] = x$; vagy nullát, ha ilyen k index nem létezik. Működését tekintve, először megnézi, hogy az $u\,..\,v$ intervallum nem üres-e. Ha üres, akkor az $A[u\,..\,v]$ résztömb is az, tehát x-et nem tartalmazza, azaz nullát kell visszaadni. Ha az $u\,..\,v$ intervallum nemüres, m lesz az $A[u\,..\,v]$ résztömb középső elemének indexe. Ha $A[m] > x$, akkor az A tömb monoton növekvő rendezettsége miatt x csak az $A[u\,..\,(m-1)]$ résztömbben lehet, ha pedig $A[m] < x$, akkor x csak az $A[(m+1)\,..\,v]$ résztömbben lehet. Mindkét esetben egy lépésben feleztük a résztömb méretét, amin keresni kell, és a továbbiakban, rekurzívan, ugyanez történik. (Ha szerencsénk van, és $A[m] = x$,

akkor persze azonnal leállhatunk.) Így, könnyen belátható, hogy legfeljebb $\lceil \log n \rceil + 1$ lépésben elfogy az $u \,..\, v$ intervallum, és megáll az algoritmus, hacsak nem áll meg hamarabb az $A[m] = x$ feltételű programágon.

A binarySearch(A, x) függvény binSearch(A, u, v, x) rekurzív függvény számára interfészt biztosít. A programozási tapasztalatok szerint a rekurzív alprogramokhoz az esetek túlnyomó többségében szükség van egy ilyen interfész alprogramra. A rekurzív alprogram ugyanis az esetek többségében, mint a fenti példában is, az eredeti feladat egy általánosítását számítja ki, és gyakran több paramétere is van, mint az eredeti alprogramnak.

Figyeljük meg azt is, hogy a binSearch(A, u, v, x) függvénynek van két rekurzív és két nemrekurzív programága. A nemrekurzív ágakat *leálló ágaknak* nevezzük. **Tetszőleges rekurzív alprogramban kell lennie ilyen leálló ágnak**, hiszen ez szükséges (bár önmagában még nem elégséges) a rekurzió helyes megállásához. A leálló ágakon kezelt esetekeket *alapeseteknek* nevezzük. (Ebben a függvényben tehát két alapeset van. Az egyik az üres intervallum esete, amikor nincs megoldás. A másik az $A[m] = x$ eset, amikor megtaláltunk egy megoldást.) Ha egy rekurzív alprogramnak nincs leálló ága, akkor tuhatjuk, hogy vagy végtelen rekurzióba fog esni, vagy hibás működéssel fog megállni.

A bináris (más néven logaritmikus) keresésre fent adott megoldás nem használja ki, hogy alprogramhíváskor résztömböket is átadhatunk. Ha a programozási nyelvünk ezt támogatja, gyakran javíthatjuk a reurzív kód olvashatóságát ilyen módon. Az alábbi példában a (rész)tömböt nullától indexeltük, így az extremális index a „-1”.



3.6. Programok, alprogramok és hatékonyságuk

Emlékeztetünk rá, hogy az eljárásokat, függvényeket és az osztályok metódusait együtt *alprogramoknak* nevezzük, így az általunk vizsgált szekvenciális programok futása lényegében véve az alprogram hívások végrehajtásából áll.

A programok hatékonyságát általában a ciklusiterációk és az alprogram hívások számának összegével mérjük, és *műveletigénynek* nevezzük. A tapasztalatok, és bizonyos elméleti megfontolások alapján is, a program valóságos futási ideje a műveletigényével nagyjából egyenesen arányos. Mivel ennek az arányosságnak a szorzója elsősorban a számítógépes környezet sebességétől függ, így a műveletigény a programok hatékonyságáról jó, a programozási környezettől alapvetően független nagyságrendi információval szolgál. A legtöbb program esetében a nemrekurzív alprogram hívások számlálása a műveletigény nagyságrendje szempontjából elhanyagolható.

Most sorban megvizsgáljuk az előző alfejezetből ismerős alprogramok műveletigényeit, és ezzel kapcsolatban szemléletesen bevezetünk néhány műveletigény osztályt is. Az egyszerűség kedvéért a formális paraméterként adott tömb méretét mindegyik esetben n-nel jelöljük, és a műveletigényeket n függvényében adjuk meg. Általában is szokás a műveletigényt a bemenet méretének függvényében megadni.

Az init$(A/1 : \mathcal{T}[n]$ **;** $x : \mathcal{T})$ **eljárás** pontosan n iterációt végez. A műveletigényt $T(n)$-nel jelölve tehát azt mondhatjuk, hogy $T(n) = n + 1$.[4] Ha (mint most is) $T(n)$ az n pozitív együtthatós lineáris függvénye[5], azt szokás mondani, hogy $T(n) \in \Theta(n)$, ahol $\Theta(n)$ az előbbinél kicsit pontosabban azokat a függvényeket jelenti, amelyek aluról és felülről is az n pozitív együtthatós lineáris függvényeivel becsülhetők.(A $T(n) = n + 1$ függvény alsó és felső becslése is lehet önmaga.)

Ezt általánosítva azt mondhatjuk, hogy $\lim_{n\to\infty} g(n) = \infty$ esetén $\Theta(g(n))$ az a függvényosztály, aminek elemei alulról és felülről is a $g(n)$ pozitív együtthatós lineáris függvényeivel becsülhetők. (A $\Theta(g(n))$ függvényosztály szokásos definíciója a 8. fejezetben olvasható. Könnyen látható, hogy az előbbi meghatározás annak speciális esete.)

A linearSearch$(A : \mathcal{T}[n]$ **;** $x : \mathcal{T}) : \mathbb{N}$ **függvény** esetében nem tudunk ilyen általános, minden lehetséges inputra érvényes $T(n)$ műveletigényt megadni, hiszen előfordulhat, hogy azonnal megtaláljuk a keresett elemet, de az is, hogy végignézzük az egész tömböt, de így sem találjuk. Ezért itt megkülönböztetünk minimális műveletigényt [legjobb eset: $mT(n)$] és maximális műveletigényt [legrosszabb eset: $MT(n)$].

Világos, hogy most $mT(n) = 1$. Ez akkor áll elő, amikor x a tömb első eleme, és így egyet sem iterál a kereső ciklus. Ilyenkor, nagyságrendileg azt mondhatjuk, hogy $mT(n) \in \Theta(1)$, ahol $\Theta(1)$ azokat az $f(n)$ függvényeket

[4] Nyilván ugyanez érvényes az **init** eljárás másik változatára is.

[5] Itt ez a pozitív együttható az „egy”.

jelenti, amelyek két (n-től független) pozitív konstans közé szoríthatók, legalábbis nagy n értékekre. (Most minden n értékre $1 \leq mT(n) \leq 1$, azaz az előbbi követelmény teljesül.)

Továbbá $MT(n) = n + 1$. Ez az eset akkor áll elő, amikor a tömb nem tartalmazza x-et. Az **init** eljárásnál mondottak alapján tehát most $MT(n) \in \Theta(n)$.

A binarySearch$(A/1 : \mathcal{T}[]$ **;** $x : \mathcal{T}) : \mathbb{N}$ **függvény** esetében nyilván $mT(n) = 2$, ahonnét $mT(n) \in \Theta(1)$. (Ez az eset akkor realizálódik, amikor x a tömb $\left\lfloor \frac{n+1}{2} \right\rfloor$ sorszámú eleme.)

Azt mondhatjuk, hogy a bináris keresés műveletigénye a legrosszabb esetben körülbelül $\log n$-nel arányos, hiszen az aktuális résztömb minden rekurzív hívásnál feleződik. (A legrosszabb eset akkor realizálódik, amikor x nem eleme a tömbnek.) Ebből arra következtethetünk, hogy $MT(n) \in \Theta(\log n)$.

A lineáris és a bináris keresést összehasonlítva, a legjobb eset műveletigénye lényegében véve ugyanaz (bár a két keresés legjobb esete különbözik egymástól). A legrosszabb esetben a lineáris keresés $\Theta(n)$, míg a bináris keresés $\Theta(\log n)$ műveletigényű, viszont a bináris keresés rendezett input tömböt igényel. Ha tehát az input rendezett, és elég sok elemet tartalmaz, a maximális műveletigényt tekintve a bináris keresés lényegesen gyorsabb, és az előnye csak tovább nő, amikor még nagyobb tömbökre hívjuk meg, hiszen

$$\lim_{n \to \infty} \frac{\log n}{n} = 0$$

Ha például $n \approx 1000$ akkor $\log n \approx 10$, ha $n \approx 10^6$ akkor $\log n \approx 20$, és ha $n \approx 10^9$ akkor $\log n \approx 30$, ami azt mutatja, hogy a bináris keresés műveletigénye nagyon lassan nő; gyakorlati méretű rendezett tömbökre, kevesebb, mint 40 rekurzív hívás bőven elegendő, míg a lineáris keresés akár sok milliárd ciklusitarációt is igényelhet.

Szokás még az algoritmusok $AT(n)$ átlagos műveletigényéről is beszélni, ahol n az input mérete. Ezt általában a műveletigény várható értékeként határozzák meg, úgy hogy felteszik, minden lehetséges bemenetnek ugyanakkora a valószínűsége (ami nem mindig tükrözi a valóságot). Mindenféleképpen igaz kell legyen, hogy $mT(n) \leq AT(n) \leq MT(n)$. Részletes kiszámítását – megfelelő matematikai felkészültség híján – néhány kivételtől eltekintve mellőzni fogjuk.

4. Az „algoritmusok” témakör bevezetése a beszúró rendezésen keresztül

Az *algoritmus* egy jól definiált kiszámítási eljárás, amely valamely adatok (*bemenet* vagy *input*) felhasználásával újabbakat (*kimenet*, *eredmény* vagy *output*) állít elő [4]. (Gondoljunk pl. két egész szám legnagyobb közös osztójára [lnko$(x, y : \mathbb{Z}) : \mathbb{Z}$], a lineáris keresésre a maximum keresésre, az összegzésre stb.) Az algoritmus bemenete adott *előfeltételnek* kell eleget tegyen. (Az lnko(x, y) függvény esetén pl. x és y egész számok, és nem mindkettő nulla.) Ha az előfeltétel teljesül, a kimenet adott *utófeltételnek* kell eleget tegyen. Az utófeltétel az algoritmus bemenete és a kimenete közt elvárt kapcsolatot írja le. Maga az algoritmus számítási lépésekből áll, amiket általában szekvenciák, elágazások, ciklusok, eljárás- és függvényhívások segítségével, valamely pszeudo-kódot (pl. struktogramokat) felhasználva formálunk algoritmussá.

Szinte minden komolyabb számítógépes alkalmazásban szükséges, elsősorban a tárolt adatok hatékony visszakeresése céljából, azok rendezése. Így témánk egyik klasszikusa a *rendezési feladat*. Most megadjuk, a rendező algoritmusok bemenetét és kimenetét milyen elő- és utófeltétel páros, ún. *feladat specifikáció* írja le. Ehhez először megemlítjük, hogy *kulcs* alatt olyan adatot értünk, aminek típusán teljes rendezés definiált. (Kulcs lehet pl. egy szám vagy egy sztring.)

Bemenet: n darab kulcs $\langle a_1, a_2, \ldots, a_n \rangle$ sorozata.

Kimenet: A bemenet egy olyan $\langle a_{p_1}, a_{p_2}, \ldots, a_{p_n} \rangle$ permutációja, amelyre $a_{p_1} \leq a_{p_2} \leq \cdots \leq a_{p_n}$.

A fenti feladat nagyon egyszerű, ti. könnyen érthető, hatékony megoldására viszont kifinomult algoritmusokat (és hozzájuk kapcsolódó adatszerkezeteket) dolgoztak ki, így az algoritmusok témakörnek a szakirodalomban jól bevált bevezetése lett.

4.1. Tömb monoton növekvő rendezése

Rendezés alatt a továbbiakban, alapértelmezésben mindig monoton növekvő, pontosabban nem-csökkenő rendezést fogunk érteni, úgy, hogy a rendezés megfeleljen a fenti specifikációnak.

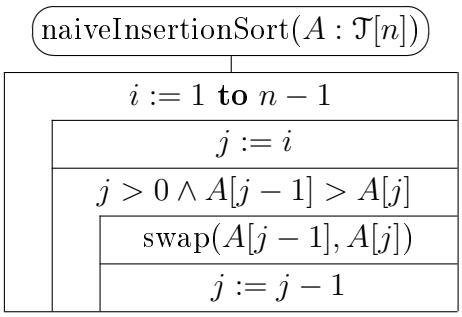
Egy sorozatot legegyszerűbben egy tömbben tárolhatunk, amit az egyes rendező eljárások paraméterlistáján általában „$A : \mathcal{T}[n]$”-nel vagy „$B/1 : \mathcal{T}[n]$”-nel fogunk jelölni. Emlékeztetünk, hogy az A és a B fizikailag tömbhivatkozások, amelyek *length* mezője a tömb hossza, *pointer* mezője pedig az úgynevezett tömbtárolóra hivatkozik, és a tömbhivatkozások így alkalmasak

a tömbök azonosítására. A tömbtároló a tömb elemeit ($A[0], \ldots, A[n-1]$, illetve $B[1], \ldots, B[n]$) tartalmazza. Ha $n = 0$, akkor a tömbnek nincs eleme. $B[k\,..\,u]$ az a résztömb, amelyben k az első elem indexe, u pedig az utolsó elem indexe. Ha $k > u$, akkor a $B[k\,..\,u]$ résztömb üres. A $B[1\,..\,n]$ résztömb a B tömb minden elemét tartalmazza. $A[k\,..\,u)$ az a résztömb, amelyben k az első elem indexe, $u-1$ pedig az utolsó elem indexe. Ha $k \geq u$, akkor az $A[k\,..\,u)$ résztömb üres. Az $A[0\,..\,n)$ résztömb az A tömb minden elemét tartalmazza. A tömb rendezéseknél feltesszük, hogy a tömb $\mathcal{T}$ elemtípusára teljes rendezés definiált, és az értékadó utasítás is értelmezve van.

4.1.1. Beszúró rendezés (Insertion sort)

Ha valaki semmit sem tud a rendezésekről, és megkapja azt a feladatot, hogy rakjon 10-30 dolgozatot nevek szerint sorba, jó eséllyel ösztönösen ezt az algoritmust fogja alkalmazni: Kiválaszt egy dolgozatot, a következőt ábécé rendben elé vagy mögé teszi, a harmadikat e kettő elé, közé, vagy mögé teszi a megfelelő helyre stb. Ha a rendezést egy számsorra kell alkalmaznunk, pl. az $\langle 5, 4, 2, 8, 3\rangle$-ra, először felosztjuk a sorozatot egy rendezett és egy ezt követő rendezetlen szakaszra, úgy, hogy kezdetben csak az első szám van a rendezett részben: $\langle 5 \mid 4, 2, 8, 3\rangle$. Ezután beszúrjuk a rendezetlen szakasz első elemét a rendezett részbe a megfelelő helyre, és ezt ismételgetjük, amíg a sorozat rendezetlen vége el nem fogy:
$\langle 5, 4, 2, 8, 3\rangle = \langle 5 \mid 4, 2, 8, 3\rangle \to \langle 4, 5 \mid 2, 8, 3\rangle \to$
$\to \langle 2, 4, 5 \mid 8, 3\rangle \to \langle 2, 4, 5, 8 \mid 3\rangle \to \langle 2, 3, 4, 5, 8 \mid \rangle = \langle 2, 3, 4, 5, 8\rangle$.

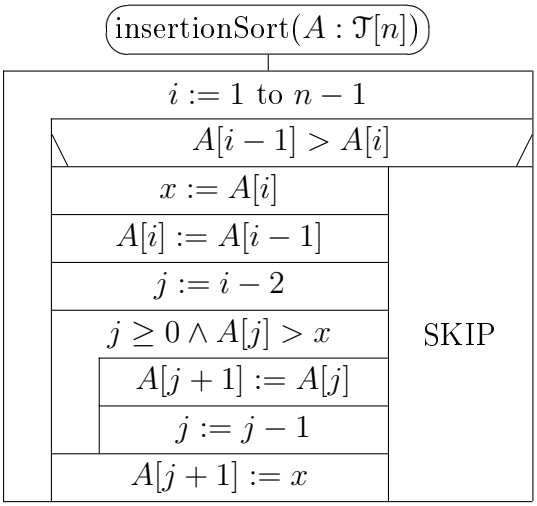
A rendezett beszúrás technikája attól függ, hogyan tároljuk a sorozatot. Ha egy tömbben, akkor az a legegyszerűbb megoldás, ha a beszúrandó elemet addig cserélgetjük a bal szomszédjával, amíg a helyére nem ér.



A fenti naiv megoldás azonban sok felesleges adatmozgatással jár, hiszen azt az elemet, amit a helyére szeretnénk vinni, újra és újra kivesszük a tömbből, majd visszatesszük bele. Nyilván hatékonyabb lenne, ha az elején kivennénk, majd amikor már megvan a helye, csak akkor tennénk vissza a tömbbe. (Ha persze már eleve a helyén van, akkor meg se mozdítjuk.)

Az előbbi megfontolás alapján a beszúró rendezés alapváltozatában a beszúrást úgy végezzük el, hogy a rendezetlen szakasz első elemét (legyen x) összehasonlítjuk a rendezett szakasz utolsó elemével (legyen u). Ha $u \leq x$, akkor x a helyén van, csak a rendezett szakasz felső határát kell eggyel növelni. Ha $u > x$, akkor x-et elmentjük egy temporális változóba, és u-t az x helyére csúsztatjuk. Úgy képzelhetjük, hogy u régi helyén most egy „lyuk" keletkezett. Az x a lyukba pontosan akkor illik bele, ha nincs bal szomszédja, vagy ez $\leq x$. Addig tehát, amíg a lyuknak van bal szomszédja, és ez nagyobb, mint x, a lyuk bal szomszédját mindig a lyukba tesszük, és így a lyuk balra mozog. Ha a lyuk a helyére ér, azaz x beleillik, akkor bele is tesszük. (Ld. az 1. ábrát és az alábbi struktogramot!)

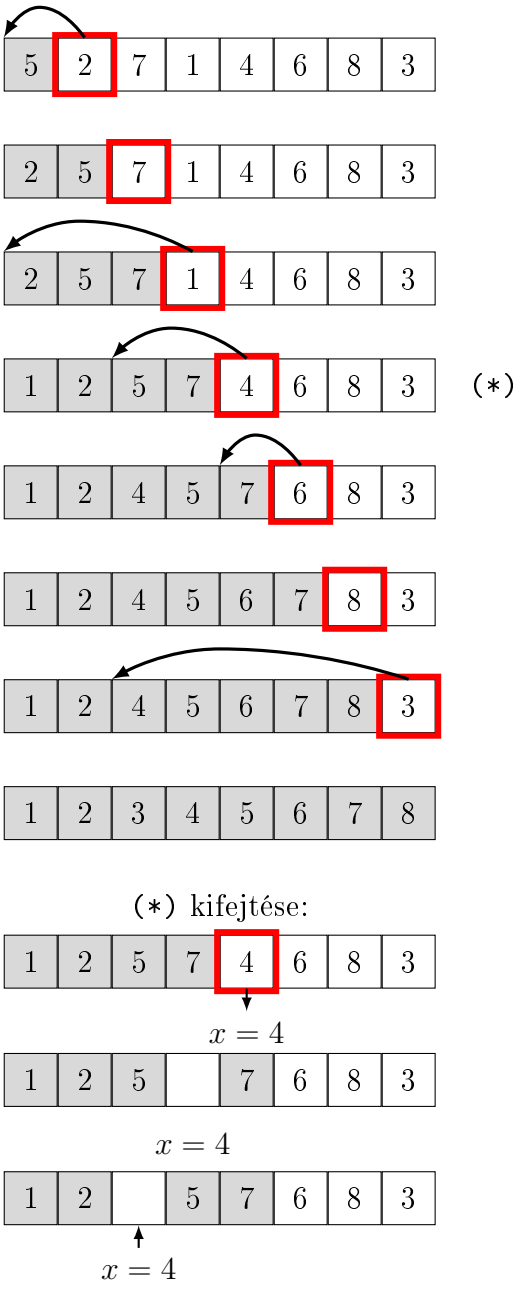
Tekintsük például a $\langle 2, 4, 5, 8, 3\rangle$ tömböt, ami a 8-ig rendezett, és már csak a 3-at kell rendezetten beszúrni. Először úgy találjuk, hogy $8 > 3$, így a 3-at kivesszük x-be, majd a lyukat (jelölje „_") a helyére mozgatjuk, és végül beletesszük a 3-at: $\langle 2, 4, 5, 8, 3\rangle \to \langle 2, 4, 5, 8, \_\rangle, x = 3 \to \langle 2, 4, 5, \_, 8\rangle, x = 3 \to \langle 2, 4, \_, 5, 8\rangle, x = 3 \to \langle 2, \_, 4, 5, 8\rangle, x = 3 \to \langle 2, 3, 4, 5, 8\rangle$.



A fenti eljárás az A tömböt rendezi monoton növekvően az előbb ismertetett egyszerű beszúró rendezéssel. A fő ciklus invariánsa:

$$1 \leq i \leq n \wedge A[0\,..\,n) \text{ az input tömb egy permutáltja,}$$
$$\text{aminek az } A[0\,..\,i) \text{ prefixe monoton növekvően rendezett.}$$

Összefoglalva a működést: Ha $n < 2$, akkor az A tömb üres, vagy egyelemű, ezért rendezett, és a program fő ciklusa egyszer sem fut le. Ha $n \geq 2$, a rendezés meghívásakor csak annyit tudhatunk, hogy $A[0\,..\,1)$ rendezett, tehát



1. ábra. A beszúró rendezés szemléltetése.

$i = 1$-re fennáll az invariáns. A fő ciklus magja ezután mindig $A[i]$-t szúrja be a tömb rendezett szakaszába, i-t eggyel növeli és tartja az invariánst. Mikor tehát i eléri az n értéket, már a teljes A tömb rendezett, és ekkor be is fejeződik az eljárás.

4.1. Feladat. *Az 1. ábrának megfelelő módon illusztrálja a beszúró rendezés (insertion sort) működését az alábbi tömbre! A második* 22 *beszúrását fejtse is ki!* $A = \langle 31; 41; 59; 22; 58; 7; 22; 91; 41\rangle$.

4.2. Példa. *Rendezze beszúró rendezéssel (insertionSort(A)) a következő tömböt!* $[4; 2; 7; 3; 1; 2']$ *(Az aposztróf a kulcs mellett a kulcs második előfordulását jelöli, az értékét nem módosítja.) Adja meg a kezdeti rendezett résztömböt, majd sorban mindegyik beszúrás után is annak pillanatnyi állapotát! Hány kulcs-összehasonlítást végez a rendezés az egyes beszúrások alatt, és hogy hány adatmozgatás történik az egyes beszúrások során?*

Megoldás: Kulcs-összehasonlítás: amikor a tömb elemeit egymáshoz vagy az x segédváltozóhoz hasonlítjuk. Adatmozgatás: értékadások a tömb elemei, ill. a tömb eleme és az x segédváltozó között.

A kezdeti rendezett résztömb a tömb $A[0\,..\,1) = [4]$ prefixe. (A rendezetlen rész a tömb [2;7;3;1;2'] maradéka.)

Az $A[0\,..\,1) = [4]$ rendezett résztömbbe beszúrjuk az $A[1] = 2$ kulcsot. Ehhez 1 kulcs-összehasonlítás ($A[0] > A[1]$) és 3 adatmozgatás ($x := A[1]$; $A[1] := A[0]$; $A[0] := x$) szükséges. (Mivel a belső ciklusnál rögtön az elején $j = -1$, az $A[j] > x$ kulcs-összehasonlítás nem hajtódik végre.) Eredmény, a tömb $[2; 4]$ rendezett prefixe. (A rendezetlen rész a tömb $[7; 3; 1; 2']$ maradéka.)

Az $A[0\,..\,2) = [2; 4]$ rendezett résztömbbe beszúrjuk az $A[2] = 7$ kulcsot. Ehhez 1 kulcs-összehasonlítás ($A[1] > A[2]$: hamis) és 0 adatmozgatás szükséges. Eredmény, a tömb $[2; 4; 7]$ rendezett prefixe. (A rendezetlen rész a tömb $[3; 1; 2']$ maradéka.)

Az $A[0\,..\,3) = [2; 4; 7]$ rendezett résztömbbe beszúrjuk az $A[3] = 3$ kulcsot. Ehhez 3 kulcs-összehasonlítás és 4 adatmozgatás szükséges. Eredmény, a tömb $[2; 3; 4; 7]$ rendezett prefixe. (A rendezetlen rész a tömb $[1; 2']$ maradéka.)

Az $A[0\,..\,4) = [2; 3; 4; 7]$ rendezett résztömbbe beszúrjuk az $A[4] = 1$ kulcsot. Ehhez 4 kulcs-összehasonlítás és 6 adatmozgatás szükséges. Eredmény, a tömb $[1; 2; 3; 4; 7]$ rendezett prefixe. (A rendezetlen rész a tömb $[2']$ maradéka.)

Az $A[0\,..\,5) = [1; 2; 3; 4; 7]$ rendezett résztömbbe beszúrjuk az $A[5] = 2'$ kulcsot. Ehhez 4 kulcs-összehasonlítás ($A[4] > A[5]$, $A[3] > x$, $A[2] > x$,

végül $A[1] > x$: hamis) és 5 adatmozgatás ($x := A[5]$; $A[5] := A[4]$; $A[4] := A[3]$; $A[3] := A[2]$; $A[2] := x$) szükséges. Eredmény, a tömb $[1; 2; 2'; 3; 4; 7]$ rendezett prefixe, ami most már az egész tömb. (A rendezetlen résztömb üres.)

4.1.2. Programok hatékonysága – és a beszúró rendezés

Fontos kérdés, hogy egy S program, például a fenti rendezés mennyire hatékony. *Hatékonyság alatt az eljárás erőforrás igényét, azaz futási idejét és tárigényét értjük.*[6] Az algoritmusok erőforrásigényét a bemenet mérete, rendező algoritmusoknál a rendezendő adatok száma (n) függvényében szokás megadni. Mivel ez az egyszerű rendezés a rendezendő tömbön kívül csak néhány segédváltozót igényel, extra tárigénye minimális, n-től független konstans, azaz $S_{IS}(n) \in \Theta(1)$ [ahol az S a tárigényre (space complexity) utal, az IS pedig a rendezés angol nevének (Insertion Sort) a rövidítése]. Így elsősorban a futási idejére lehetünk kíváncsiak. Mint már említettük (3.6), ezzel kapcsolatos nehézség, hogy nem ismerjük a leendő programozási környezetet: sem a programozási nyelvet, amiben kódolni fogják, sem a fordítóprogramot vagy interpretert, sem a leendő futtatási környezetet, sem a számítógépet, amin futni fog, így nyilván a futási idejét sem tudjuk meghatározni.

Meghatározhatjuk, vagy legalább becslés(eke)t adhatunk viszont arra, hogy adott n méretű input esetén valamely adott S algoritmus *hány alprogramhívást hajt végre + hányat iterálnak összesen kódban szereplő különböző ciklusok*. Emlékeztetünk rá, hogy megkülönböztetjük a legrosszabb vagy maximális $MT_S(n)$, a várható vagy átlagos $AT_S(n)$ és a legjobb vagy minimális $mT_S(n)$ eseteket (3.6). A valódi maximális, átlagos és minimális futási idők általában ezekkel arányosak lesznek. Ha $MT_S(n) = mT_S(n)$, akkor definíció szerint $T_S(n)$ a minden esetre vonatkozó műveletigény (tehát az eljáráshívások és a ciklusiterációk számának összege), azaz $T_S(n) = MT_S(n) = AT_S(n) = mT_S(n)$

A továbbiakban, a programok futási idejével kapcsolatos számításoknál, a *műveletigény* és a *futási idő*, valamint a *költség* kifejezéseket szinonimákként fogjuk használni, és ezek alatt az *alprogramhívások és a ciklusiterációk összegére* vonatkozó $MT_S(n)$, $AT_S(n)$, $mT_S(n)$ – és ha létezik, $T_S(n)$ – függvényeket értjük, ahol n az input mérete.

Tekintsük most példaként a fentebb tárgyalt beszúró rendezést (insertion sort)![7] A rendezés során egyetlen eljáráshívás hajtódik végre, és ez az

[6] Nem különböztetjük meg most a különféle hardver komponenseket, hiszen ezeket az algoritmus szintjén nem is ismerjük.

[7] A legtöbb program esetében a nemrekurzív alprogram hívások számlálása a műveletigény nagyságrendje szempontjából elhanyagolható. A gyakorlás kedvéért most mégis

insertionSort($A : \mathcal{T}[]$) eljárás hívása. Az eljárás fő ciklusa minden esetben pontosan $(n-1)$ -szer fut le.

Először adjunk becslést a beszúró rendezés minimális futási idejére, amit jelöljünk $mT_{IS}(n)$-nel, ahol n a rendezendő tömb mérete, általában a kérdéses kód által manipulált adatszerkezet mérete.[8] Lehet, hogy a belső ciklus egyet sem iterál, pl. ha a fő ciklus mindig a jobb oldali ágon fut le, mert $A[0\,..\,n)$ eleve monoton növekvően rendezett. Ezért

$$mT_{IS}(n) = 1 + (n-1) = n$$

(Egy eljáráshívás + a külső ciklus $(n-1)$ iterációja.)

Most adjunk becslést ($MT_{IS}(n)$) a beszúró rendezés maximális futási idejére! Világos, hogy az algoritmus ciklusai akkor iterálnak a legtöbbet, ha mindig a külső ciklus bal ágát hajtja végre, és a belső ciklus $j = -1$-ig fut. (Ez akkor áll elő, ha a tömb szigorúan monoton csökkenően rendezett.) Végrehajtódik tehát egy eljáráshívás + a külső ciklus $(n-1)$ iterációja, amihez a külső ciklus adott i értékkel való iterációjakor a belső ciklus maximum $(i-1)$-szer iterál. Mivel az i, 1-től $n-1$-ig fut, a belső ciklus összesen legfeljebb $\sum_{i=1}^{n-1}(i-1)$ iterációt hajt végre. Innét

$$MT_{IS}(n) = 1 + (n-1) + \sum_{i=1}^{n-1}(i-1) = n + \sum_{j=0}^{n-2} j = n + \frac{(n-1)*(n-2)}{2}$$

$$MT_{IS}(n) = \frac{1}{2}n^2 - \frac{1}{2}n + 1$$

Látható, hogy a minimális futási idő becslése az $A[0\,..\,n)$ input tömb méretének lineáris függvénye, míg a maximális, ugyanennek négyzetes függvénye, ahol a polinom fő együtthatója mindkét esetben pozitív. A továbbiakban ezeket a következőképpen fejezzük ki:

$$mT_{IS}(n) \in \Theta(n), \quad MT_{IS}(n) \in \Theta(n^2).$$

A $\Theta(n)$ ($Theta(n)$) függvényosztály ugyanis tartalmazza az n összes, pozitív főegyütthatós lineáris függvényét, $\Theta(n^2)$ pedig az n összes, pozitív főegyütthatós másodfokú függvényét. (Általában egy tetszőleges $g(n)$, a program hatékonyságának becslésével kapcsolatos függvényre a $\Theta(g(n))$ függvényosztály pontos definícióját a 8. fejezetben fogjuk megadni.)

Mint a maximális futási időre vonatkozó példából látható, a Θ jelölés szerepe, hogy elhanyagolja egy polinom jellegű függvényben (1) a kisebb nagyságrendű tagokat, valamint (2) a fő tag pozitív együtthatóját. Az előbbi azért

figyelembe vesszük őket.

[8] Az *IS* a rendezés angol nevének (Insertion Sort) a rövidítése.

jogos, mert a futási idő általában nagy méretű inputoknál igazán érdekes, hiszen tipikusan ilyenkor lassulhat le egy egyébként logikailag helyes program. Elég nagy n-ekre viszont pl. az $a * n^2 + b * n + c$ polinomban $a * n^2$ mellett $b * n$ és c elhanyagolható. A fő tag pozitív együtthatóját pedig egyrészt azért hanyagolhatjuk el, mert ez a programozási környezet, mint például a számítógép sebességének ismerete nélkül tulajdonképpen semmitmondó, másrészt pedig azért, mert ha az $a * f(n)$ alakú fő tag értéke n -et végtelenül növelve maga is a végtelenhez tart (ahogy az lenni szokott), elég nagy n-ekre az a konstans szorzó sokkal kevésbé befolyásolja a függvény értékét, mint az $f(n)$.

Látható, hogy a beszúró rendezés a legjobb esetben nagyon gyorsan rendez: Nagyságrendileg a lineáris műveletigénynél gyorsabb rendezés elvileg is lehetetlen, hiszen ehhez a rendezendő sorozat minden elemét el kell érnünk. A legrosszabb esetben viszont, ahogy n nő, a futási idő négyzetesen növekszik, ami, ha n milliós vagy még nagyobb nagyságrendű, már nagyon hosszú futási időket eredményez. Vegyünk példának egy olyan számítógépet, ami másodpercenként $2 * 10^9$ elemi műveletet tud elvégezni. Jelölje most $mT(n)$ az algoritmus által elvégzendő elemi műveletek minimális, míg $MT(n)$ a maximális számát! Vegyük figyelembe, hogy $mT_{IS}(n) = n$, ami közelítőleg a külső ciklus iterációinak száma, és a külső ciklus minden iterációja legalább 8 elemi műveletet jelent; továbbá, hogy $MT_{IS}(n) \approx (1/2) * n^2$, ami közelítőleg a belső ciklus iterációinak száma, és itt minden iteráció legalább 12 elemi műveletet jelent. Innét a $mT(n) \approx 8 * n$ és a $MT(n) \approx 6 * n^2$ képletekkel számolva a következő táblázathoz jutunk:

n	$mT_{IS}(n)$	in secs	$MT_{IS}(n)$	in time
1000	8000	$4 * 10^{-6}$	$6 * 10^6$	0.003 sec
10^6	$8 * 10^6$	0.004	$6 * 10^{12}$	50 min
10^7	$8 * 10^7$	0.04	$6 * 10^{14}$	$\approx$ 3.5 days
10^8	$8 * 10^8$	0.4	$6 * 10^{16}$	$\approx$ 347 days
10^9	$8 * 10^9$	4	$6 * 10^{18}$	$\approx$ 95 years

Világos, hogy már tízmillió rekord rendezésére is gyakorlatilag használhatatlan az algoritmusunk. (Az implementációs problémákat most figyelmen kívül hagytuk.) Látjuk azt is, hogy hatalmas a különbség a legjobb és a legrosszabb eset között.

Felmerülhet a kérdés, hogy átlagos esetben mennyire gyors az algoritmus. Itt az a gond, hogy nem ismerjük az input sorozatok eloszlását. Ha például az inputok monoton növekvően előrendezettek, ami alatt azt értjük, hogy az input sorozat elemeinek a rendezés utáni helyüktől való távolsága általában egy n -től független k konstanssal felülről becsülhető, azok száma pedig, amelyek a végső pozíciójuktól távolabb vannak, egy szintén n -től független

s konstanssal becsülhető felülről, az algoritmus műveletigénye lineáris, azaz $\Theta(n)$ marad, mivel a belső ciklus nem többször, mint $(k+s)*n$ -szer fut le. Ha viszont a bemenet monoton csökkenően előrendezett, az algoritmus műveletigénye is közel marad a legrosszabb esethez. (Bár ha ezt tudjuk, érdemes a tömböt a rendezés előtt $\Theta(n)$ időben megfordítani, és így monoton növekvően előrendezett tömböt kapunk.)

Véletlenített input sorozat esetén, egy-egy újabb elemnek a sorozat már rendezett kezdő szakaszába való beszúrásakor, átlagosan a rendezett szakaszban lévő elemek fele lesz nagyobb a beszúrandó elemnél. A rendezés várható műveletigénye ilyenkor tehát:

$$AT_{IS}(n) \approx 1 + (n-1) + \sum_{i=1}^{n-1}\left(\frac{i-1}{2}\right) = n + \frac{1}{2} * \sum_{j=0}^{n-2} j =$$

$$= n + \frac{1}{2} * \frac{(n-1)*(n-2)}{2} = \frac{1}{4}n^2 + \frac{1}{4}n + \frac{1}{2}$$

Nagy n -ekre tehát $AT_{IS}(n) \approx (1/4) * n^2$. Ez körülbelül a fele a maximális futási időnek, ami a rendezendő adatok milliós nagyságrendje esetén már így is nagyon hosszú futási időket jelent. Nagyságrenddel jobb műveletigényeket kapunk majd a *heap sort*, valamint az *oszd meg és uralkodj* elven alapuló rendezések (*merge sort, quicksort*) esetén. Összegezve az eredményeinket:

$$mT_{IS}(n) \in \Theta(n)$$

$$AT_{IS}(n), MT_{IS}(n) \in \Theta(n^2)$$

Ehhez hozzátehetjük, hogy előrendezett inputok esetén (ami a programozási gyakorlatban egyáltalán nem ritka) a beszúró rendezés segítségével lineáris időben tudunk rendezni, ami azt jelenti, hogy erre a feladatosztályra nagyságrendileg, és nem túl nagy k és s konstansok esetén valóságosan is az optimális megoldás a beszúró rendezés.

4.2. A futási időkre vonatkozó becslések magyarázata*

Jelölje most az insertionSort($A : \mathcal{T}[n]$) eljárás *tényleges* maximális és minimális futási idejét sorban $MrT(n)$ és $mrT(n)$!

Világos, hogy a rendezés akkor fut le a leggyorsabban, ha a fő ciklus minden elemet a végső helyén talál, azaz mindig a jobb oldali ágon fut le. (Ez akkor áll elő, ha a tömb már eleve monoton növekvően rendezett.) Legyen a a fő ciklus jobb oldali ága egyszeri végrehajtásának futási ideje, b pedig az eljárás meghívásával, a fő ciklus előkészítésével és befejezésével, valamint az eljárásból való visszatéréssel kapcsolatos futási idők összege! Ekkor a és

b nyilván pozitív konstansok, és $mrT(n) = a * (n-1) + b$. Legyen most $p = min(a,b)$ és $P = max(a,b)$; ekkor $0 < p \leq P$, és
$p*(n-1)+p \leq mrT(n) = a*(n-1)+b \leq P*(n-1)+P$, ahonnét
$p*n \leq mrT(n) \leq P*n$, azaz

$$p * mT_{IS}(n) \leq mrT(n) \leq P * mT_{IS}(n)$$

Most adjunk becslést a beszúró rendezés maximális futási idejére, $(MrT(n))$! Világos, hogy az algoritmus akkor dolgozik a legtöbbet, ha mindig a külső ciklus bal ágát hajtja végre, és a belső ciklus $j = 0$-ig fut. (Ez akkor áll elő, ha az input tömb szigorúan monoton csökkenően rendezett.) Legyen most a belső ciklus egy lefutásának a műveletigénye d; c pedig a külső ciklus bal ága egy lefutásához szükséges idő, eltekintve a belső ciklus lefutásaitól, de hozzászámolva a belső ciklusból való kilépés futási idejét (amibe beleértjük a belső ciklus feltétele utolsó kiértékelését, azaz a $j = -1$ esetet), ahol $c, d > 0$ állandók. Ezzel a jelöléssel:

$$MrT(n) = b + c*(n-1) + \sum_{i=1}^{n-1} d*(i-1) = b + c*(n-1) + d * \sum_{j=0}^{n-2} j =$$

$$= b + c*(n-1) + d * \frac{(n-1)*(n-2)}{2}$$

Legyen most $q = min(b,c,d)$ és $Q = max(b,c,d)$; ekkor $0 < q \leq Q$, és
$q + q*(n-1) + q*\frac{(n-1)*(n-2)}{2} \leq MrT(n) \leq Q + Q*(n-1) + Q*\frac{(n-1)*(n-2)}{2}$
$q*(n + \frac{(n-1)*(n-2)}{2}) \leq MrT(n) \leq Q*(n + \frac{(n-1)*(n-2)}{2})$, azaz

$$q * MT_{IS}(n) \leq MrT(n) \leq Q * MT_{IS}(n)$$

Mindkét esetben azt kaptuk tehát, hogy a valódi futási idő az *eljáráshívások és a ciklusiterációk számának összegével becsült futási idő* megfelelő pozitív konstansszorosaival alulról és felülről becsülhető, azaz, pozitív konstans szorzótól eltekintve ugyanolyan nagyságrendű. Könnyű meggondolni, hogy ez a megállapítás tetszőleges program minimális és maximális futási idejeire is általánosítható. (Ezt azonban már az Olvasóra bízzuk.)

4.3. Rendezések stabilitása

Egy rendezés akkor *stabil (stable)*, ha megtartja az egyenlő kulcsú elemek eredeti sorrendjét.

A beszúró rendezés (insertion sort) például – úgy, ahogy ebben a jegyzetben tárgyaljuk – stabil. A fenti tömbrendező algoritmusnál ez abból látható,

hogy a tömb rendezett szakaszába az újabb elemeket jobbról balra szúrjuk be, és a beszúrandóval egyenlő kulcsú elemeket már nem lépjük át.

Hasonlóképpen látni fogjuk, hogy a később ismertetendő összefésülő rendezés (merge sort) is stabil, míg a kupacrendezés (heap sort) és gyorsrendezés (quicksort) nem stabilak.

A stabilitás (stability) előnyös tulajdonság lehet, ha rekordokat rendezünk, és vannak azonos kulcsú rekordok. Tegyük fel például, hogy a rekordok emberek adatait tartalmazzák, és név szerint vannak rendezve. Ha most ugyanezeket a rekordokat stabil rendezéssel pl. születési év szerint rendezzük, akkor az azonos évben születettek névsorban maradnak.

A stabilitás nélkülözhetetlen tulajdonság lesz majd később a (lineáris műveletigényű) radix rendezésnél (ami nem kulcsösszehasonlításokkal dolgozik, eltérően a beszúró és a fent említett másik három rendezéstől).

4.4. Kiválasztó rendezések (selection sorts)

4.3. Feladat. *Tekintsük az $A[0\,..\,n)$ tömb rendezését a következő algoritmussal! Először megkeressük a tömb minimális elemét, majd megcseréljük $A[0]$-lal. Ezután megkeressük a második legkisebb elemét és megcseréljük $A[1]$-gyel. Folytassuk ezen a módon az $A[0\,..\,n)$ első $(n-1)$ elemére! (Itt tehát a tömböt egy rendezett és egy rendezetlen szakaszra bontjuk. A rendezett szakasz a tömb elején kezdetben üres. A minimumkeresés mindig a rendezetlen részen történik, és a csere után a rendezett szakasz mindig eggyel hosszabb lesz.) Pl.:*

$\langle 3;9;7;1;6;2\rangle \to \langle 1 \;/\; 9;7;3;6;2\rangle$
$\to \langle 1;2 \;/\; 7;3;6;9\rangle \to \langle 1;2;3 \;/\; 7;6;9\rangle$
$\to \langle 1;2;3;6 \;/\; 7;9\rangle \to \langle 1;2;3;6;7 \;/\; 9\rangle$
$\to \langle 1;2;3;6;7;9\rangle$

Írjunk struktogramot erre a – minimumkiválasztásos rendezés néven közismert – algoritmusra minKivRend($A : \mathcal{T}[n]$) néven! Mi lesz a fő ciklus invariánsa? Miért elég csak az első $(n-1)$ elemre lefuttatni? Adjuk meg az $MT(n)$ és $mT(n)$ függvényeket a minimumkiválasztásos rendezésre a szokásos Θ-jelöléssel!

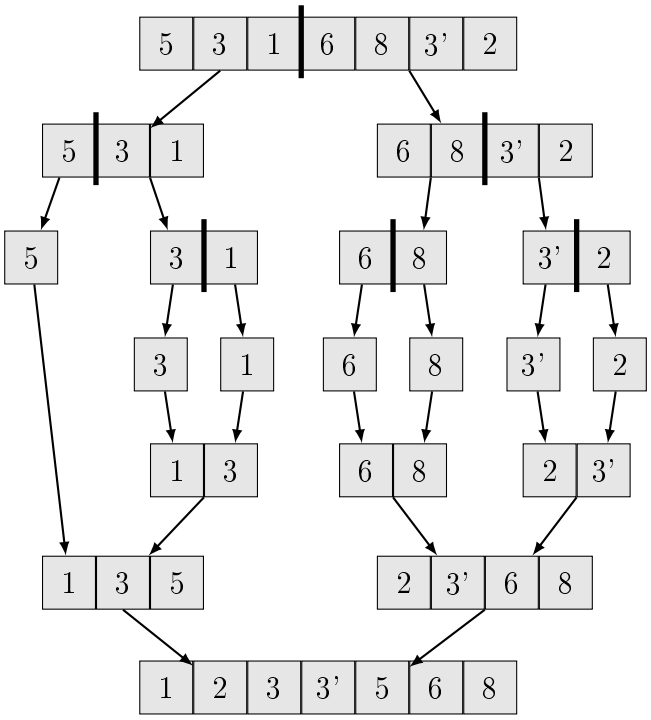
4.4. Feladat. *Tekintsük az $A[0\,..\,n)$ tömb rendezését a következő algoritmussal! Először megkeressük a tömb maximális elemét, majd megcseréljük $A[n-1]$-gyel. Ezután megkeressük a második legnagyobb elemét és megcseréljük $A[n-2]$-vel. Folytassuk ezen a módon az $A[0\,..\,n)$ utolsó $(n-1)$ elemére! Pl.:*

$\langle 3;1;9;2;7;6\rangle \to \langle 3;1;6;2;7 \;/\; 9\rangle$
$\to \langle 3;1;6;2 \;/\; 7;9\rangle \to \langle 3;1;2 \;/\; 6;7;9\rangle$

$\to \langle 2;1 \ / \ 3;6;7;9\rangle \to \langle 1 \ / \ 2;3;6;7;9\rangle$
$\to \langle 1;2;3;6;7;9\rangle$

Írjunk struktogramot erre a – maximumkiválasztásos rendezés néven közismert – algoritmusra MaxKivRend($A : \mathfrak{T}[n]$) néven! Mi lesz a fő ciklus invariánsa? Miért elég csak az utolsó $(n-1)$ elemre lefuttatni? Adjuk meg az $MT(n)$ és $mT(n)$ függvényeket a maximumkiválasztásos rendezésre a szokásos Θ-jelöléssel!

4.5. Feladat. *Stabilak-e a fenti kiválasztó rendezések? Miért?*



2. ábra. Az összefésülő rendezés szemléltetése. Az $\langle 5;3;1;6;8;3';2\rangle$ sorozatot rendezzük úgy, hogy elfelezzük, majd a fél-sorozatokat (előbb a bal oldalit, majd a jobb oldalit) az algoritmus rekurzívan rendezi, és végül a rendezett fél-sorozatokat rendezetten összefésüli. (Az aposztróf a kulcs mellett a kulcs második előfordulását jelöli, az értékét nem módosítja. Mint látható, a merge sort stabil, azaz az egyenlő kulcsú elemek sorrendjét nem változtatja meg.) Az *összefésülések során elvégzett kulcs-összehasonlítások eredményei* $\to$ *az összefésülések eredményei* sorban a következők: $3 > 1 \to 1;3$ $5 > 1;5 > 3 \to 1;3;5$ $6 \le 8 \to 6;8$ $3' > 2 \to 2;3'$ $6 > 2;6 > 3' \to 2;3';6;8$ $1 \le 2;3 > 2;3 \le 3';5 > 3';5 < 6 \to 1;2;3;3';5;6;8$

5. Az *oszd meg és uralkodj* elven alapuló gyors rendezések

Az *oszd meg és uralkodj (divide and conquer)* elv lényege, hogy az eredeti problémát (rekurzívan) két vagy több részproblémára bontjuk fel, kivéve, ha annyira egyszerű, hogy direkt módon is könnyedén megoldható. Az részproblémák ugyanolyan jellegűek, mint az eredeti, csak valamilyen értelemben kisebb méretűek, és ugyanazzal az algoritmussal oldjuk meg őket, mint az eredetit. A részproblémák megoldásaiból rakjuk össze az eredeti feladat megoldását.

A fent vázolt *oszd meg és uralkodj* technika sokféle probléma hatékony, algoritmikus megoldásának alapja. Ebben a fejezetben a gyorsrendezést (quicksort) és az összefésülő (összefuttató) rendezést (merge sort) hozzuk példának.

5.1. Összefésülő rendezés (merge sort)

Az *oszd meg és uralkodj* módszerrrel gyakran adhatunk optimális megoldást. Az adott problémát két (vagy több) az eredetihez hasonló, de egyszerűbb, azaz kisebb részfeladatra bontjuk, majd ezeket megoldva, a részeredményekből összerakjuk az felbontott probléma megoldását. Ha a megoldandó (rész)probléma elég egyszerű, akkor ezt már közvetlenül oldjuk meg.

Ha például adott egy rendezendő kulcssorozat, az általános elvnek megfelelően most is két esetet különböztetünk meg:

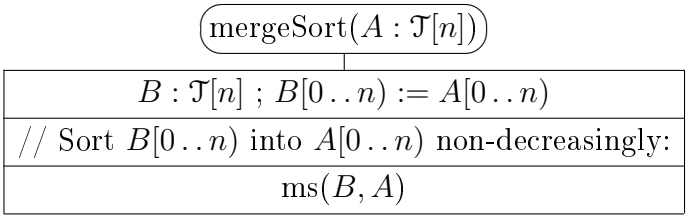
Az üres és az egyelemű sorozatok eleve rendezettek; a hosszabb sorozatokat pedig elfelezzük, a két fél-sorozatot ugyanezzel a módszerrel rendezzük, és a rendezett fél-sorozatokat rendezetten összefésüljük.

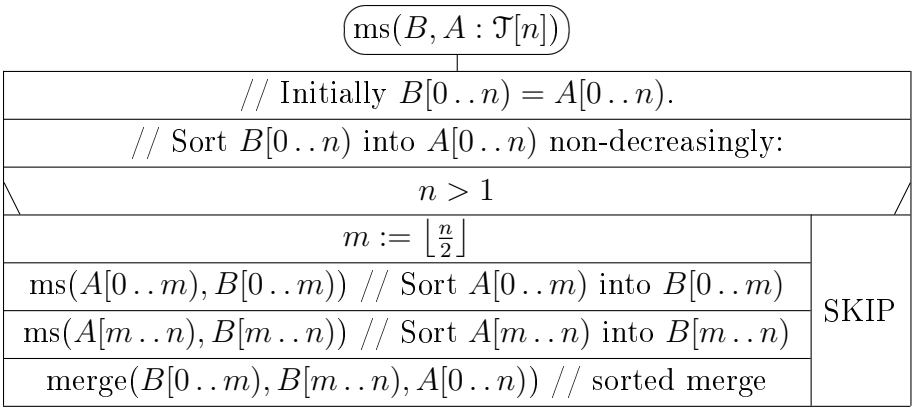
Ezt az eljárást hívjuk *összefésülő*, vagy más néven összefuttató *rendezés*nek (angolul *merge sort*, ld. a 2. ábrát).

Az összefésülő rendezés stabil (azaz megőrzi az egyenlő kulcsú elemek bemeneti sorrendjét). A legrosszabb esetének műveletigénye aszimptotikusan optimális az ún. összehasonító rendezések között.

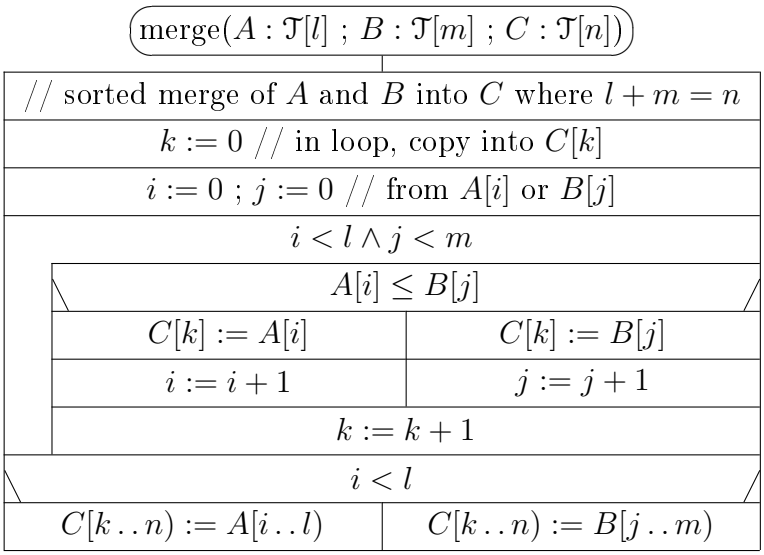
Az összefésülő rendezés (merge sort, rövidítve MS) nagy elemszámú sorozatokat is viszonylag gyorsan rendez. Ráadásul a legjobb és a legrosszabb eset között, az alprogramhívások és a ciklusiterációk számát tekintve sem mutatkozik eltérés. Első megközelítésben azt mondhatjuk, hogy a futási ideje is $n \log n$ -nel arányos. Ezt a szokásos Θ jelöléssel a következőképpen fejezzük ki.

$$MT_{MS}(n) = mT_{MS}(n) = T_{MS}(n) \in \Theta(n \log n)$$





Az $m := \lfloor \frac{n}{2} \rfloor$ értékadással elfeleztük az aktuális (rész)tömböt: $A[0\,..\,m)$ és $A[m\,..\,n)$ hossza ugyanaz, ha n páros szám; továbbá $A[0\,..\,m)$ eggyel rövidebb, mint $A[m\,..\,n)$, ha n páratlan szám.



A merge sort stabilitását a merge() eljárás explicit ciklusának elágazása biztosítja: $A[i] = B[j]$ esetén a $A[i]$-t másolja $C[k]$-ba, mert az A résztömb megelőzi a B résztömböt a mindkettőt tartalmazó (rész)tömbben.

A merge() eljárás pontosan n iterációt végez, mert n a C (rész)tömb hossza, és az eljárás során ennek mindegyik eleme egy-egy iterációval áll elő A-ból vagy B-ből. (Az explicit ciklus $C[0\,..\,k)$ -t tölti fel, k iterációval. Az implicit ciklusok a $C[k\,..\,n) := \ldots$ alakú utasításokba rejtve jelennek meg. Ezek közül csak az egyik fog végrehajtódni, $n-k$ iterációval. Összesen tehát $k + (n-k) = n$ iteráció hajtódik végre. A merge() eljárás törzsének (mb) a műveletigénye tehát:

$$T_{\mathrm{mb}}(n) = n \in \Theta(n)$$

5.1.1. A merge sort hatékonysága: szemléletes megközelítés

Műveletigény: Az 5.1. alfejezet elején megemlítettük, hogy az összefésülő rendezés (merge sort, rövidítve MS) nagy elemszámú sorozatokat is viszonylag gyorsan rendez. Ráadásul a legjobb és a legrosszabb eset között, az alprogramhívások és a ciklusiterációk számát tekintve sem mutatkozik eltérés. Első megközelítésben azt mondhatjuk, hogy *tömbökre* a maximális és a minimális futási ideje egyenlő és $n \log n$ -nel arányos. Ezt a szokásos Θ jelöléssel a következőképpen fejezzük ki.

$$T_{MS}(n) \in \Theta(n \log n)$$

A fenti összefüggést vázlatosan a következőképpen indokolhatjuk. Látható módon a műveletek túlnyomó részét a merge eljárás végzi el. Ezért az ez által végzett munkára adunk becslést, hogy az egész rendezés műveletigényének nagyságrendjét is megkapjuk. A merge(A, B, C) eljárás minden egyes meghívásának műveletigénye az 5.1. alfejezet szerint $\Theta(l)$, ahol l az aktuális résztömb, azaz C hossza. Mivel a rekurzív ms(B, A) ejárás minden hívásban felezi a rendezendő résztömb hosszát, ezért a rekurziónak kb. $\log n + 1$ szintje van, és az alsó egy vagy két szint kivételével minden rekurziós szinten igaz az, hogy a merge hívások résztömbjei együtt lefedik az egész A tömböt. Így egy tetszőleges szint összes merge hívásának műveletigényét összeadva $\Theta(n)$ nagyságrendű műveletigény adódik (az alsó két szintet leszámítva, ahol ez kevesebb is lehet). A szintenkénti műveletigényt a szintek számával szorozva aszimptotikusan $\Theta(n \log n)$ műveletigény adódik.

A fenti műveletigény matematikailag precíz kiszámítását az 5.1.2. alfejezetben fogjuk elvégezni.

Tárigény: Szükségünk van egy n méretű segédtömbre és néhány segédváltozóra. Ehhez hozzáadódik a rekurzív hívások adminisztrálásának tárigénye,

ami a rekurzió maximális mélysége miatt $\log n$ -nel arányos. Ezért *tömbökre* $S_{MS}(n) \in \Theta(n + \log n) = \Theta(n)$, röviden $S_{MS}(n) \in \Theta(n)$, mivel nagy n-ekre $\log n$ az n-hez képest elhanyagolható.

5.1. Feladat. *Mint láttuk, a* merge sort *fenti verziója a rendezendő tömbbel azonos méretű segédtömböt használ. Tegyük fel, hogy adott esetben a* merge sort *által használt extra memóriát szeretnénk csökkenteni, ezért egy n méretű rendezendő tömbhöz csak legfeljebb egy $\lfloor \frac{n}{2} \rfloor$ méretű segédtömböt szeretnénk létrehozni, amit ténylegesen csak a* merge() *eljárásban használunk.*

Rajzoljuk le a mergeSort($A : \mathcal{T}[n]$) *eljárás és szubrutinjai új struktogramjait! A műveletigény némi növekedése megengedett, ami azt jelenti hogy a legrosszabb esetben az adatmozgatások száma kb. a másfélszeresére nő. Alapvető, hogy az $MT(n) \in \Theta(n \log n)$ műveletigény az új* összefésülő rendezésre *is igaz maradjon.*

5.1.2. The time complexity of merge sort*

Merge sort is one of the fastest sorting algorithms, and there is not a big difference between its worst-case and best-case (i.e. maximal and minimal) running time. For our array sorting version, we state:

$$MT_{\text{mergeSort}}(n) = mT_{\text{mergeSort}}(n) = T_{\text{mergeSort}}(n) \in \Theta(n \log n)$$

Proof: Clearly $T_{\text{mergeSort}}(0) = 2$. We suppose that $n > 0$. First, we count all the loop iterations of the procedure merge. Next, we count the procedure calls of merge sort.

Loop iterations: We proved above that a single call of merge(A, B, C) makes $C.length$ iterations. Let us consider the levels of recursion of procedure ms(B, A). At level 0 of the recursion, ms is called for the whole array. Considering all the recursive calls and the corresponding subarrays at a given recursion depth, at level 1, this array is divided into two halves, at level 2 into 4 parts and so on. Let n_{ij} be the length of the jth subarray of the whole array at level i, and let $m = \lfloor \log n \rfloor$. We have:

At level 0: $2^m \leq n_{01} = n < 2^{m+1}$
At level 1: $2^{m-1} \leq n_{1j} \leq 2^{m+1-1}$ $(j \in [1\,..\,2^1])$
At level 2: $2^{m-2} \leq n_{2j} \leq 2^{m+1-2}$ $(j \in [1\,..\,2^2])$
...
At level i: $2^{m-i} \leq n_{ij} \leq 2^{m+1-i}$ $(i \in [1\,..\,m], j \in [1\,..\,2^i])$
...
At level $m-1$: $2 \leq n_{(m-1)j} \leq 4 = 2^2$ $(j \in [1\,..\,2^{m-1}])$

At level m: $1 \leq n_{mj} \leq 2 = 2^1$ $(j \in [1\,..\,2^m])$
At level $m+1$: $n_{(m+1)j} = 1$ $(j \in [1\,..\,(n-2^m)])$

Thus, these subarrays cover the whole array at levels $[0\,..\,m]$. At levels $[0\,..\,m)$, merge is called for each subarray, but at level m, it is called only for those subarrays with length 2, and the number of these subarrays is $n - 2^m$. Merge makes as many iterations as the length of the actual C (subarray). Consequently, at each level in $[0\,..\,m)$, merge makes n iterations in all the merge calls of the level altogether. At level m, the sum of the iterations is $2 * (n - 2^m)$, and there is no iteration at level $m+1$. Therefore, the total of all the iterations during the merge calls is

$$T_{mb[0\,..\,m]}(n) = n * m + 2 * (n - 2^m).$$

The number of procedure calls: The ms calls form a strictly binary tree. (See section 9.) The leaves of this tree correspond to the subarrays with length 1. Thus, this strictly binary tree has n leaves and $n-1$ internal nodes. Consequently, we have $2n-1$ calls of ms and $n-1$ calls of the merge. Adding to this the single call of mergeSort(), we receive $3n-1$ procedure calls altogether.

And there are n iterations hidden into the initial assignment $B[0\,..\,n) := A[0\,..\,n)$ in procedure mergeSort. Thus, the number of steps of mergeSort is

$$T(n) = (n+n*m+2*(n-2^m))+(3*n-1) = n*\lfloor \log n \rfloor + 2*(n-2^{\lfloor \log n \rfloor})+4*n-1$$

$$n * \log n + 2 * n \leq n * (\log n - 1) + 4 * n - 1 \leq T(n) < n * \log n + 6 * n.$$

Considering Theorem 8.29, we have

$$T_{MS}(n) \in \Theta(n \log n).$$

5.1.3. The space complexity of merge sort*

Considering the above merge sort code, we create the temporal array $B : \mathcal{T}[n]$ in the main procedure, and we need a temporal variable for the implicit for loop. We need $\Theta(n)$ working memory here. In addition, we have seen in the previous subsection (5.1.2) that the number of levels of recursion is $\lfloor \log n \rfloor + 1$ which is approximately $\log n$, and we have a constant amount of temporal variables at each level. Thus, we need $\Theta(\log n)$ working memory inside the call stack to control the recursion. And $\Theta(\log n) \prec \Theta(n)$. These measures do not depend on the content of the array to be sorted. For this reason, the space complexity of this array version of merge sort is $S(n) \in \Theta(n)$.

5.2. Gyorsrendezés (Quicksort)

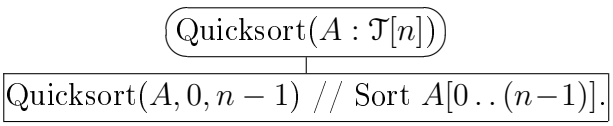
A *gyorsrendezés (quicksort)* az „oszd meg és uralkodj” elvet képviselő algoritmusok másik klasszikus példája. Tetszőleges, nagy méretű zsákból először kiválasztunk egy tengelyt (pivot), majd a maradékot két kisebb részre bontjuk: az egyik a tengelynél kisebb, a másik a nagyobb elemeket tartalmazza. (A tengellyel egyenlők bármelyik részbe kerülhetnek.) Ezután a *quicksort* rekurzívan rendezi a részeket.

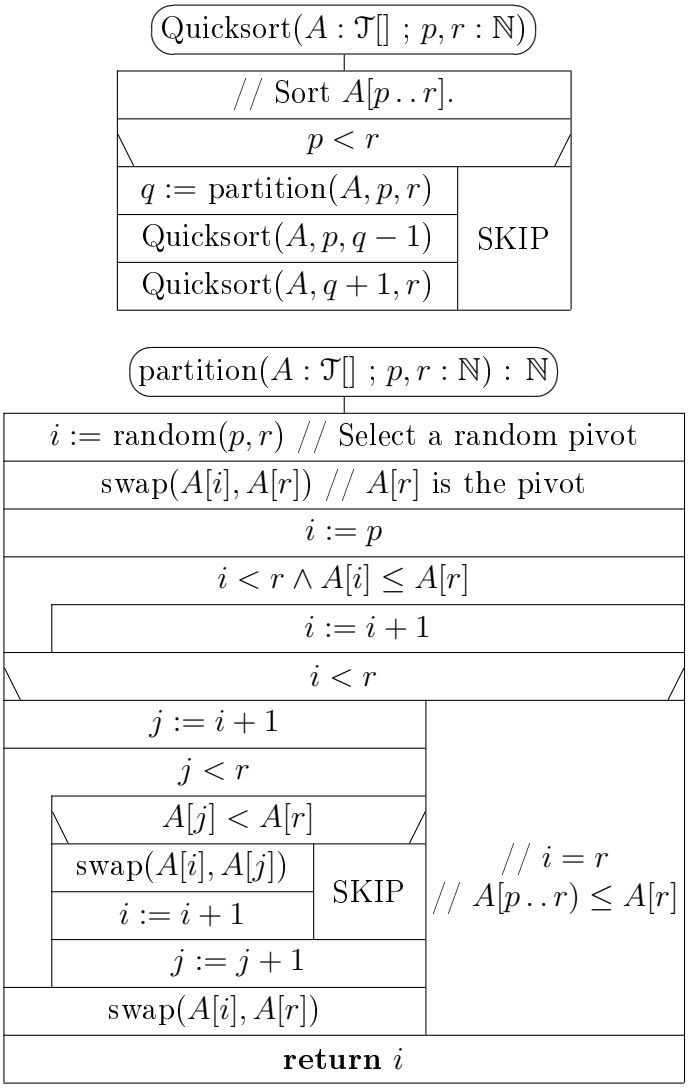
5.2. Megjegyzés. *Jelen idő szerint* nem ismerünk elfogadható hatékonyságú stabil quicksort tömbrendezőt. *Amennyiben* nagy hatékonyságú stabil rendezésre *van szükségünk, a* merge sort *és annak továbbfejlesztései állnak rendelkezésünkre. Amennyiben a* stabilitás nem szükséges, *nagyméretű tömböket gyakran mégis* quicksort*-tal, illetve annak valamelyik továbbfejlesztésével rendezünk, mivel nagy méretű* tömbökre *ennek a* legjobb az átlagos műveletigénye. *(A futtatási tapasztalatok szerint,* láncolt listákra általában jobb választás a merge sort.*)*

Az algoritmus lépései tömbökre, pontokba szedve:

- Válaszd ki a rendezendő (rész)tömb egy tetszőleges elemét! Ez lesz a *tengely* (angolul *pivot*).
- Részekre bontás (partitioning): Rendezd át úgy a tömböt, hogy minden, a tengelynél kisebb elem a tengely előtt, a nagyobbak pedig utána jöjjenek! (A tengellyel egyenlők bármelyik részbe kerülhetnek.) Ezzel az ún. particionálással (partition) a tengely már a végleges helyére került.
- Alkalmazd rekurzívan a fenti lépéseket, külön a tengelynél kisebb elemek résztömbjére, és külön a tengelynél nagyobb elemek résztömbjére!
- Az üres és az egyelemű résztömbök a rekurzió alapesetei. Ezek ui. már eleve készen vannak, így nem is kell őket rendezni.

A tengely kiválasztása és a részekre bontás lépései sokféleképpen elvégezhetők. A módszerek konkrét megválasztása erősen befolyásolja a rendezés hatékonyságát. Alapvető követelmény, hogy a „tengely kiválasztása és a részekre bontás” lépései együtt lineáris időben befejeződjenek.





A partition függvény működésének magyarázata és szemléltetése:
A partition fv szemléltetéséhez vezessük be a következő jelöléseket:

- $A[k\,..\,m] \leq x$ akkor és csak akkor, ha tetszőleges l-re, $k \leq l \leq m$ esetén $A[l] \leq x$
- $A[k\,..\,m] \geq x$ akkor és csak akkor, ha tetszőleges l-re, $k \leq l \leq m$ esetén $A[l] \geq x$

Feltesszük, hogy az alábbi $A[p\,..\,r]$ résztömböt bontjuk részekre, és a második 5-öst, azaz a résztömb p+3 indexű elemét választottuk tengelynek.

Az $i := \text{random}(p, r)$ utasítás utáni helyzet:

	p			i				r
$A:$	5	3	8	**5**	6	4	7	**1**

Az 1. ciklus előkészítése: A $\text{swap}(A[i], A[r])$ hívás után $A[r]$ a tengely (*pivot*).

	p							r	
$A:$	5	3	8	1	6	4	7	⑤	$pivot = A[r] = 5$

Az 1. ciklus megkeresi az első, a tengelynél nagyobb elemet, ha van ilyen. (Az i index előfordulásai, annak balról jobbra mozgását mutatják.)

	i=p	i	i					r
$A:$	5	3	**8**	1	6	4	7	⑤

Találtunk a tengelynél nagyobb elemet. A j változó a következő elemre áll.

	p		i	j				r
$A:$	5	3	**8**	1	6	4	7	⑤

Felbontottuk az $A[p\,..\,r]$ résztömböt négy szakaszra:

$$\underset{A[p\,..\,i)}{p} \underset{\leq pivot,}{\leq} \underset{A[i\,..\,j)}{i} \underset{\geq pivot,}{<} \underset{A[j\,..\,r)\,\text{(ismeretlen)},}{j} \underset{A[r]\,(\text{a } pivot).}{\leq} r$$

Ez a második ciklus invariáns tulajdonsága. (Vegyük észre, hogy a tengellyel egyenlő elemek az 1. és a 2. szakaszban is lehetnek.)

A továbbiakban az ismeretlen szakasz elemeit sorban a tengelynél $\leq$ vagy $\geq$ elemek szakaszához kapcsoljuk, míg az ismeretlen szakasz el nem fogy. Végül a tengelyt betesszük az első két szakasz közé. A második szakasz eltolása elfogadhatatlanul rossz hatékonyságot eredményezne, ezért ezt mind az első szakaszhoz csatolásnál mind a tengelynek a két szakasz közé mozgatásánál elkerüljük. A második szakasz eltolását úgy tudjuk elkerülni, hogy az aktuális elemet mindkét esetben megcseréljük a második szakasz első elemével. Ennek következménye, hogy *a quicksort algoritmusunk nem stabil.*

A második ciklus végrehajtását elkezdve, $A[j] = 1 < pivot$. Ezért meg kell cserélni $A[i]$-vel, hogy csatlakozhasson az $A[p\,..\,r]$ első, a tengelynél $\leq$ elemei szakaszához. Mivel az invariáns alapján $A[i] \geq x$, neki a 2. szakasz (a tengelynél $\geq$ elemek szakasza) végén is jó helye lesz. (Megvastagítottuk a megcserélendő elemeket.)

	p		i	j				r
$A:$	5	3	**8**	**1**	6	4	7	⑤

Megcseréljük az $A[i]$ és az $A[j]$ elemeket. Így az $A[p\,..\,r]$ első szakasza (a tengelynél $\leq$ elemek szakasza) eggyel hosszabb lett, a második szakasza (a

tengelynél $\geq$ elemek szakasza) pedig eggyel arrébb ment. Ezért az i és a j változókat is eggyel megnöveljük, hogy a ciklusinvariáns igaz maradjon.

	p			i	j			r
$A:$	5	3	1	**8**	**6**	4	7	⑤

Most $A[j] = 6 \geq pivot = 5$, ezért $A[j]$-t hozzávesszük az $A[p\,..\,r]$ 2. (a tengelynél $\geq$ elemek) szakaszához. Ehhez j-t megnöveljük eggyel.

	p			i		j		r
$A:$	5	3	1	**8**	6	**4**	7	⑤

Most viszont már $A[j] = 4 < pivot = 5$, ezért $A[j]$-ről kiderül, hogy meg kell cserélni $A[i]$-vel. (Most is megvastagítottuk a megcserélendő elemeket.)

Megcseréljük az $A[i]$ és az $A[j]$ elemeket. Így az $A[p\,..\,r]$ első szakasza (a tengelynél $\leq$ elemek szakasza) eggyel hosszabb lett, a második szakasza (a tengelynél $\geq$ elemek szakasza) pedig eggyel arrébb ment. Ezért az i és a j változókat is eggyel megnöveljük, hogy a ciklusinvariáns igaz maradjon.

	p				i		j	r
$A:$	5	3	1	4	**6**	8	**7**	⑤

Most $A[j] = 7 \geq pivot = 5$, ezért $A[j]$-t hozzávesszük az $A[p\,..\,r]$ 2. (a tengelynél $\geq$ elemek) szakaszához. Ehhez j-t megnöveljük eggyel.

Most viszont már $j = r$, ezért az $A[p\,..\,r]$ 3. szakasza elfogyott.

	p				i			j=r
$A:$	5	3	1	4	**6**	8	7	⑤

Most már az első 2 szakasz lefedi $A[p\,..\,(r-1)]$-et, és a 2. szakasz az $i < j$ invariáns miatt nemüres. Ezért a tengelyt berakhatjuk a nála $\leq$ és a nála $\geq$ elemek közé úgy, hogy a 2. szakasz első elemét megcseréljük a tengellyel. ("+" előjellel jelöljük, hogy a tengely (*pivot*) a végső helyére került, ui. tőle balra a tengelynél $\leq$, tőle jobbra pedig nála $\geq$ elemek vannak.)

	p				i			j=r
$A:$	5	3	1	4	$+$**5**	8	7	6

Ezzel az $A[p\,..\,r]$ résztömb részekre bontását befejeztük. Most még visszatérünk a tengely i indexével, hogy a Quicksort(A, p, r) rekurzív eljárás tudja, az $A[p\,..\,r]$ mely résztömbjeire kell meghívnia önmagát.

A partition eljárás másik esete az, amikor a tengely az $A[p\,..\,r]$ maximuma. Ez az eset triviális. Meggondolását az Olvasóra bízzuk.

Megjegyzés: Természetesen egyszerűbb lenne, ha a fenti partition függvényben a tengely, kezdettől fogva az $A[r]$ lenne. Ekkor azonban előrendezett inputokra a Quicksort lelassulna, mert a partition fv egyenetlenül vágna. Annak érdekében, hogy az ilyen „balszerencsés" bemenetek valószínűségét csökkentsük, és azért is, mert az előrendezett inputok rendezése a gyakorlatban egy fontos feladatosztály, érdemes a tengelyt az $A[p\,..\,r]$ résztömb egy véletlenszerű elemének választani.

A partition() fv helyességének ellenőrzéséséhez: Jelölje $A_0[p\,..\,r]$ az $A[p\,..\,r]$ tömböt a partition() fv meghívásának pillantában!

A partition fv előfeltétele: $0 \leq p < r < A.length$
(A p és r indexhatárok a függvényen belül természetesen konstansok.)

A partition fv második ciklusának invariánsa:
$A[p\,..\,r]$ egy permutációja az $A_0[p\,..\,r]$ résztömbnek $\wedge\ p \leq i < j \leq r\ \wedge$
$A[p\,..\,(i{-}1)] \leq A[r] \wedge A[i\,..\,(j{-}1)] \geq A[r]$

A partition fv utófeltétele:
$A[p\,..\,r]$ az $A_0[p\,..\,r]$ permutációja $\wedge\ p \leq i \leq r\ \wedge$
$A[p\,..\,(i{-}1)] \leq A[i] \wedge A[(i{+}1)\,..\,r] \geq A[i]$, ahol i a visszatérési érték.

5.3. Feladat. *Ellenőrizze a fenti invariánst! Ennek segítségével bizonyítsa be a partition() függvény helyességét a megadott elő- és utófeltétel szerint!*

5.4. Példa. *Rendezze gyorsrendezéssel az* [5,2,7,1,6,4,8,3] *tömböt! Feltesszük, hogy a particionálás minden esetben az aktuális résztömb első elemét választja tengelynek. Adja meg sorban a partition(A, p, r) segédfüggvény hívásai által kiszámolt résztömböket, az elemeik felsorolásával, a tengelyt „+" előjellel különböztetve meg!*

Megoldás (vázlat): Az első particionálás tehát a vektor első elemét (5) választja tengelynek, és az utolsó elemmel (3) cseréli meg. Utána a particionálás ugyanúgy folytatódik, ahogy a fenti példában is láttuk. Eredménye: [3,2,1,4,+5,7,8,6]. A második particionálás bemenete az első eredményének a tengelytől balra eső résztömbje, azaz [3,2,1,4], eredménye pedig [2,1,+3,4]. Harmadszorra a [2,1] résztömb particionálásával az [1,+2] résztömböt kapjuk. Mivel az egy elemű és az üres résztömböket nem particionáljuk (egyébként nem is lehet), az első particionálás eredményének a tengelyétől jobbra eső résztömbjével folytatjuk, ami a [7,8,6]. Végül még ezt is particionálva a [6,+7,8] résztömb adódik. A particionálások eredményei sorban (a zárójeleket elhagyva):

3,2,1,4,+5,7,8,6
2,1,+3,4
1,+2
6,+7,8

5.2.1. A gyorsrendezés (quicksort) műveletigénye

A fenti szétvágás (partition) műveletigénye nyilván lineáris, hiszen a két ciklus együtt $r - p - 1$ vagy $r - p$ iterációt végez.

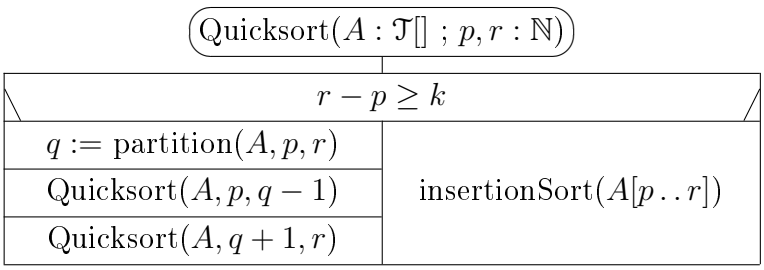
A quicksort műveletigényére ebből a következő adódik. (A részleteket ld. az MSc-n!) A várható vagy átlagos műveletigény aszimptotikusan a legjobb esethez esik közel, és a legrosszabb eset valószínűsége nagyon kicsi.

$mT(n), AT(n) \in \Theta(n \log n)$
$MT(n) \in \Theta(n^2)$

5.5. Feladat. *Lássuk be, hogy a gyorsrendezésre* $mT(n) \in O(n \log n)$ *és* $MT(n) \in \Omega(n^2)$*, ahol az* $O(g(n))$*, valamint az* $\Omega(g(n))$ *függvényosztályok definíciója a 8. fejezetben található.*

5.2.2. Vegyes gyorsrendezés (Mixed quicksort)

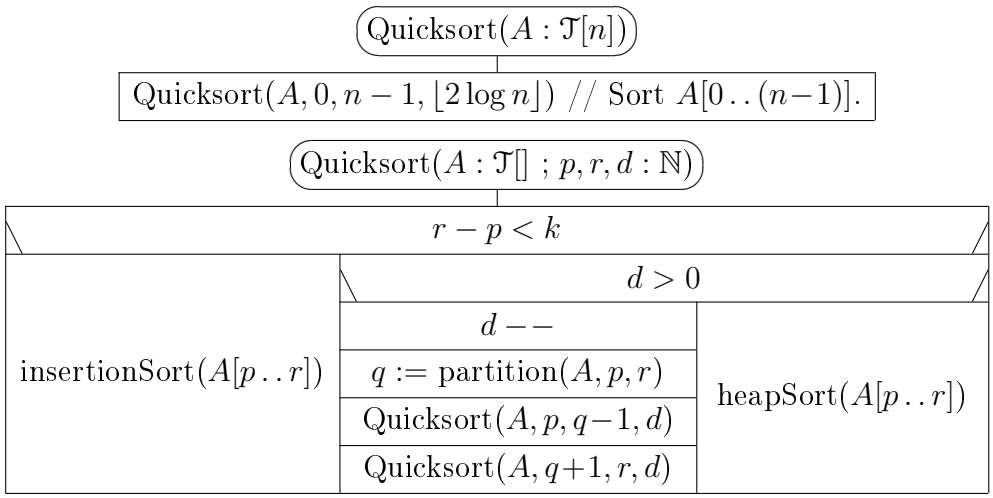
Ismert, hogy legfeljebb néhányszor tíz rendezendő elem esetén a beszúró rendezés hatékonyabb, mint a gyors rendezések (merge sort, heap sort, quicksort). Ezért pl. a Quicksort(A, p, r) eljárás jelentősen gyorsítható, ha a kis méretű résztömböknél áttérünk beszúró rendezésre. Mivel a szétvágások (partitions) során sok kicsi résztömb áll elő, így ezzel a program futása sok ponton gyorsítható:



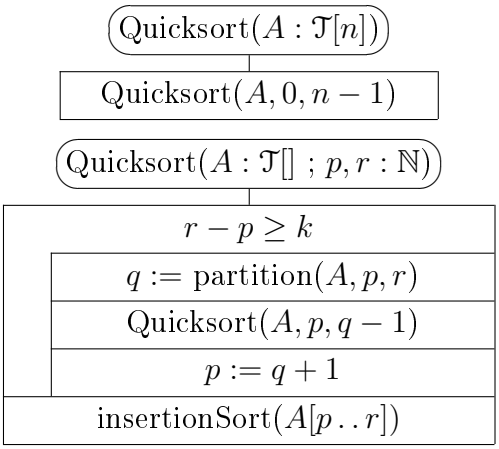
Itt $k \in \mathbb{N}$ konstans. Optimális értéke sok tényezőtől függ, de általában 25 és 50 között mozog.

5.6. Feladat. *Hogyan tudnánk az összefésülő rendezést hasonló módon gyorsítani? (Az így adódó vegyes összefésülő rendezés továbbfejlesztése a Timsort, ami ráadásul még az inputban előforduló monoton növekvő, illetve csökkenő szakaszokat is kihasználja. [Ld. Python, Java stb.])*

A Quicksort legrosszabb esetének $\Theta(n^2)$ műveletigénye kiküszöbölhető, azaz biztosítható az $MT(n) \in \Theta(n \log n)$ műveletigény, ha a vegyes gyorsrendezés rekurzív eljárásában figyeljük a rekurziós mélységet is, és pl. $\lfloor 2 \log n \rfloor$ mélység elérése esetén az aktuális résztömbre – ha még a beszúró rendezésre nem érdemes átváltani – áttérünk valamelyik olyan gyors rendezésre, ami tudja garantálni a $\Theta(n \log n)$ legrosszabb műveletigényt. Alkalmazhatunk itt segédeljárásként pl. kupacrendezést (ld. 9.11) vagy összefésülő rendezést is. Így társíthatjuk a gyorsrendezés átlagosan legjobb futási idejét valamelyik másik gyors rendező algoritmus tökéletes megbízhatóságával.



5.2.3. A gyorsrendezés végrekurzió-optimalizált változata*



5.7. Feladat. *Hogyan kellene módosítani a fenti ciklust, hogy a rekurzió mélysége* $\leq \log n$ *maradjon, és ezzel a Quicksort(n) eljárás tárigénye* $O(\log n)$ *legyen?*

5.8. Feladat. *Próbáljuk meg az előző feladat megoldásába a behozni a kupacrendezést is, ha a ciklusunk túl sokat iterálna!*

6. Elemi adatszerkezetek és adattípusok

Adatszerkezet (data structure) alatt adatok tárolásának és elrendezésének egy lehetséges módját értjük, ami lehetővé teszi a tárolt adatok elérését és módosítását, beleértve újabb adatok eltárolását és tárolt adatok törlését is. [4]

Nincs olyan adatszerkezet, ami univerzális adattároló lenne. A megfelelő adatszerkezetek kiválasztása vagy megalkotása legtöbbször a programozási feladat megoldásának alapvető része. A programok hatékonysága nagymértékben függ az alkalmazott adatszerkezetektől.

Az *adattípus (data type)* a mi értelmezésünkben egy adatszerkezet, a rajta értelmezett műveletekkel együtt.

Az *absztrakt adattípus (ADT)* esetében nem definiáljuk pontosan az adatszerkezetet, csak – informálisan – a műveleteket. Az ADT megvalósítása két részből áll:
– *reprezentálása* során megadjuk az adatszerkezetet,
– *implementálása* során pedig a műveletei kódját.

Az adattípusok megvalósítását gyakran UML jelöléssel, osztályok segítségével fogjuk leírni. A lehető legegyszerűbb nyelvi elemekre szorítkozunk. (Nem alkalmazunk sem öröklődést, sem template-eket, sem kivételkezelést.)

6.1. Verem (Stack)

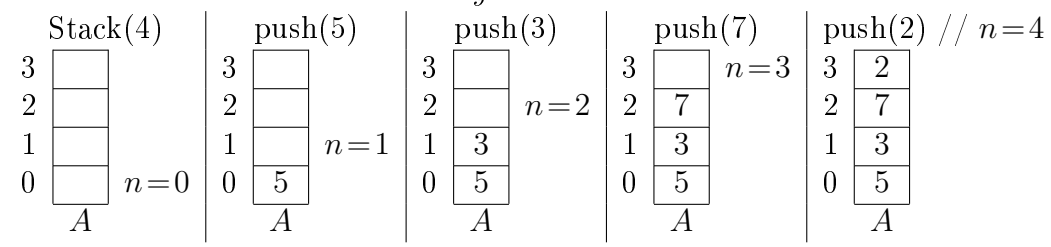
A *verem (stack)* adattípus LIFO (Last-In First-Out) adattároló, aminél tehát mindig csak az utoljára benne eltárolt, és még benne lévő adat érhető el, illetve törölhető. Tipikus műveleteit az alábbi megvalósítás mutatja.

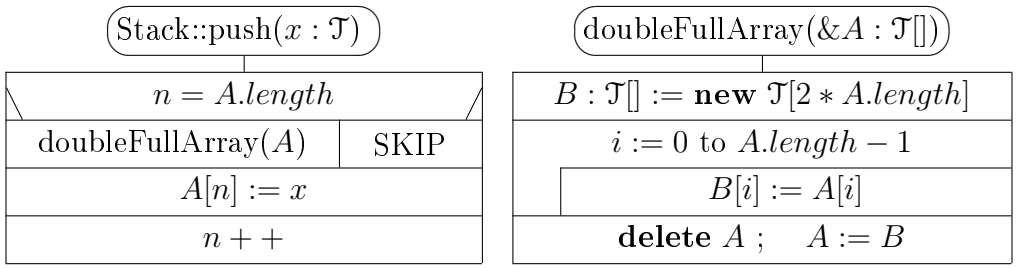
Stack
$-\ A : \mathcal{T}[]$ // $\mathcal{T}$ is some known type ; $A.length$ is the physical $-$ constant $m0 : \mathbb{N}_+ := 16$ // size of the stack, its default is m0. $-\ n : \mathbb{N}$ // $n \in 0\,..\,A.length$ is the actual size of the stack
+ Stack($m : \mathbb{N}_+ := m0$) $\{A :=$ **new** $\mathcal{T}[m]$; $n := 0\}$ // create empty stack + $\sim$ Stack() { **delete** A } + push($x : \mathcal{T}$) // push x onto the top of the stack + pop() : $\mathcal{T}$ // remove and return the top element of the stack + top() : $\mathcal{T}$ // return the top element of the stack + isEmpty() : $\mathbb{B}$ {**return** $n = 0$} + setEmpty() $\{n := 0\}$ // reinitialize the stack

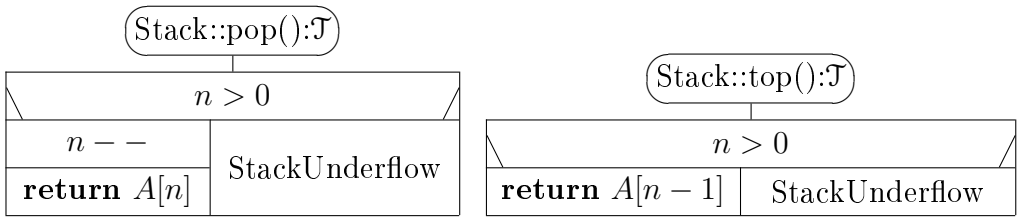
A vermet most dinamikus tömb ($A : \mathcal{T}[]$) segítségével reprezentáljuk, ahol $A.length$ a verem fizikai mérete, $\mathcal{T}$ a verem elemeinek típusa. Az egyszerűség

kevéért, alapértelmezésben feltesszük, hogy van elég szabad memória a **new** utasításaink számára.

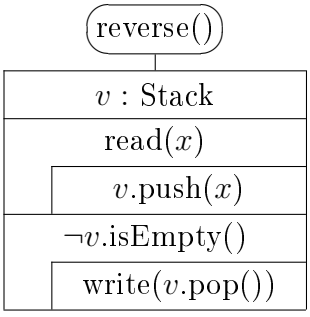
Néhány veremművelet







Példa a verem egyszerű használatára: Az input adatok kiírása fordított sorrendben. Feltesszük, hogy a read(&x:$\mathcal{T}$):$\mathbb{B}$ függvény a kurrens inputról olvas, ami akkor sikeres, és tér vissza igazzal, ha nincs még vége az inputnak. Ilyenkor beolvassa x-be a következő input adatot. A read(x) akkor sikertelen, és tér vissza hamissal, ha vége van az inputnak. Ekkor x értéke definiálatlan. A write(x) a kurrens outpura írja x értékét.



A vermek műveleteit – a push művelet kivételével – egyszerű, rekurziót és ciklust nem tartalmazó metódusokkal írtuk le. Ezért mindegyik műveletigénye $\Theta(1)$, ami – legalábbis együtt az összes elvégzett különféle művelet átlagos műveletigényét tekintve – alapkövetelmény minden verem megvalósítással kapcsolatban. A push műveletre nyilván $mT(n) \in \Theta(1)$ és $MT(n) \in \Theta(n)$. A 6.1. feladat szerint azonban a push műveletre is $AT(n) \in \Theta(1)$.

6.1. Feladat.* *Lássuk be, hogy a fenti Stack osztály* push *műveletére is teljesül* $AT(n) \in \Theta(1)$.

6.2. Sor (Queue)

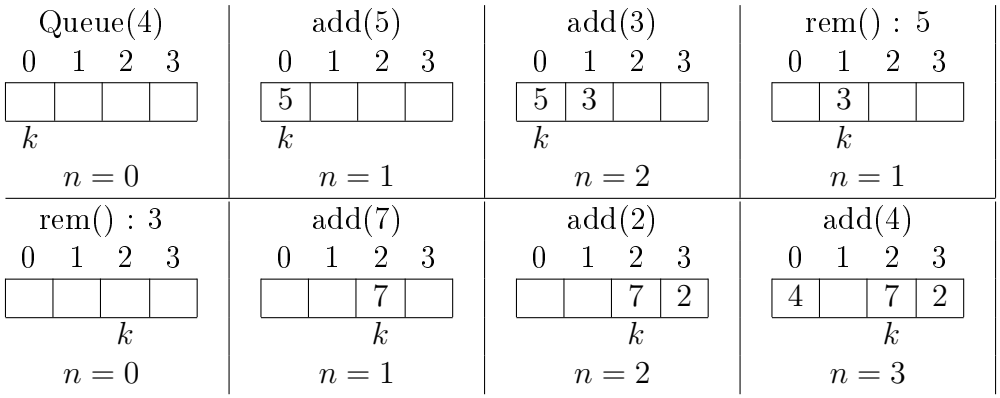
A *sor (queue)* adattípus FIFO (First-In First-Out) adattároló, aminél tehát a még benne lévő adatok közül adott pillanatban csak a legrégebben benne eltárolt érhető el, illetve törölhető. Tipikus műveleteit az alábbi megvalósítás mutatja.

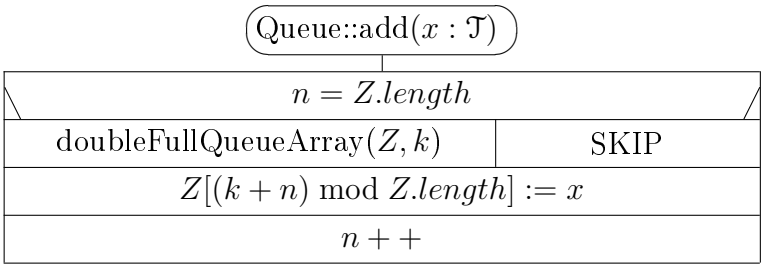
A sort nullától indexelt dinamikus tömb ($Z : \mathcal{T}[]$) segítségével reprezentáljuk, ahol az $Z.length$ a sor fizikai mérete, $\mathcal{T}$ a sor elemeinek típusa. Az egyszerűség kevéért, alapértelmezésben feltesszük, hogy van elég szabad memória a **new** utasításaink számára.

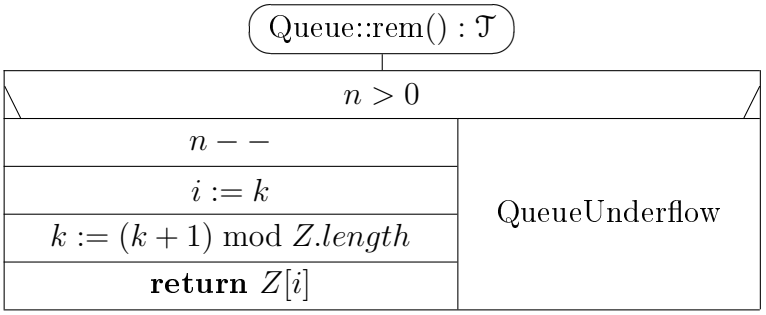
[A vermet és a sort természetesen ábrázolhatjuk láncolt listák segítségével is (ld. a 7. fejezetet), a verem esetében az egyszerű láncolt lista elejét a verem tetejének tekintve, a sor esetében pedig lista végéhez közvetlen hozzáférést biztosítva.]

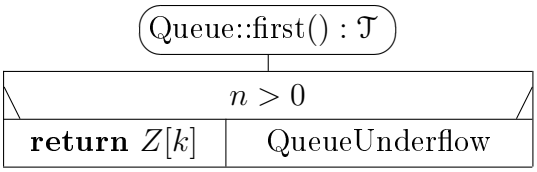
Queue
$-$ $Z : \mathcal{T}[]$ // $\mathcal{T}$ is some known type ; $Z.length$ is the physical
$-$ constant $m0 : \mathbb{N}_+ := 16$ // length of the queue, its default is m0.
$-$ $n : \mathbb{N}$ // $n \in 0\,..\,Z.length$ is the actual length of the queue
$-$ $k : \mathbb{N}$ // $k \in 0\,..\,(Z.length - 1)$: the starting position of the queue in Z
+ Queue($m : \mathbb{N}_+ := m0$){ $Z :=$ **new** $\mathcal{T}[m]$; $n := 0$; $k := 0$ }
// create an empty queue
+ add($x : \mathcal{T}$) // join x to the end of the queue
+ rem() : $\mathcal{T}$ // remove and return the first element of the queue
+ first() : $\mathcal{T}$ // return the first element of the queue
+ length() : $\mathbb{N}$ {**return** n}
+ isEmpty() : $\mathbb{B}$ {**return** $n = 0$}
+ $\sim$ Queue() { **delete** Z }
+ setEmpty() {$n := 0$} // reinitialize the queue

Egy sor néhány művelete









Az add művelet doubleFullQueueArray(Z, k) segédeljárásával kapcsolatban ld. a 6.2. feladatot!

A sorok műveleteit – az add művelet kivételével – egyszerű, rekurziót és ciklust nem tartalmazó metódusokkal írtuk le. Ezért mindegyik műveletigénye $\Theta(1)$, ami – legalábbis együtt az összes elvégzett különféle művelet átlagos műveletigényét tekintve – alapkövetelmény minden sor megvalósítással kapcsolatban. Az add műveletre nyilván $mT(n) \in \Theta(1)$ és $MT(n) \in \Theta(n)$. A 6.2. feladat szerint azonban az add műveletre is $AT(n) \in \Theta(1)$.

6.2. Feladat. *Írjuk meg a* Queue *osztály* add *műveletéhez tartozó* doubleFullQueueArray(Z, k) *segédeljárást! Ha az* add *művelet úgy találja, hogy már tele van a tömb, cserélje le nagyobbra, pontosan kétszer akkorára! Ügyeljünk a nagyságrendileg optimális átlagos futási időre, továbbá arra, hogy a* Z *tömbben a sor akár ciklikusan is elhelyezkedhet!*

Lássuk be, hogy így az add *műveletre* $mT(n) \in \Theta(1)$ *és* $MT(n) \in \Theta(n)$, *valamint* $AT(n) \in \Theta(1)$ *is teljesül!)*

6.3. Feladat. *Tegyük fel, hogy adott a* Stack *osztály, ami a* Stack(), ∼Stack(), push(x:$\mathcal{T}$), pop():$\mathcal{T}$, isEmpty():$\mathbb{B}$ *műveletekkel (elvileg) korlátlan méretű vermeket tud létrehozni és kezelni. Feltehető, hogy mindegyik művelet átlagos futási ideje* $\Theta(1)$.

Valósítsuk meg a Queue *osztályt két verem (és esetleg néhány egyszerű segédváltozó) segítségével, a következő műveletekkel:* Queue(), add(x:$\mathcal{T}$), rem():$\mathcal{T}$, length():$\mathbb{N}$. *Miért nincs szükség destruktorra? Mit tudunk mondani a műveletigényekről? Elérhető-e valamilyen értelemben a* $\Theta(1)$ *átlagos műveletigény?*

6.4. Feladat. *Tegyük fel, hogy adott a* Queue *osztály, ami a* Queue(), ∼Queue(), add(x:$\mathcal{T}$), rem():$\mathcal{T}$, length():$\mathbb{N}$ *műveletekkel (elvileg) korlátlan méretű sort tud létrehozni és kezelni. Feltehető, hogy mindegyik művelet átlagos futási ideje* $\Theta(1)$.

Valósítsuk meg a Stack *osztályt egy sor (és esetleg néhány egyszerű segédváltozó) segítségével, a következő műveletekkel:* Stack(), push(x:$\mathcal{T}$), pop():$\mathcal{T}$, isEmpty():$\mathbb{B}$. *Miért nincs szükség destruktorra? Mit tudunk mondani a műveletigényekről?*

7. Láncolt listák (Linked Lists)

A beszúró rendezésnél, a vermeknél, a soroknál és a rendezett prioritásos soroknál, egy dimenziós tömbökkel véges sorozatokat reprezentáltunk. Tegyük fel például, hogy adott az $A : \mathbb{Z}[100]$ tömb, aminek az $A[0\,..\,90)$ résztömbjében egy 90 elemű számsort tárolunk, és az $n = 90$ változó tartalmazza a tömb aktuálisan felhasznált prefixének hosszát. Ennek a módszernek az a fő előnye, hogy a sorozat bármely eleme közvetlenül, $\Theta(1)$ időben elérhető az $A[i]$ indexeléssel ($i \in [0\,..\,n)$). Hátránya, hogy ha pl. a $A[1]$ pozícióra be szeretnénk szúrni sorrendtartó módon az x számot, akkor az $A[1] := x$ értékadás előtt az $A[1\,..\,90)$ résztömb minden elemét eggyel jobbra kell csúsztatni. Hasonlóan, ha $A[1]$-et sorrendtartó módon szeretnénk törölni, akkor ehhez az $A[2\,..\,90)$ résztömb minden elemét eggyel balra kell csúsztatni. (Mindkét esetben az n értékét is megfelelően kell módosítani.) Látható, hogy minkét művelet költsége lineárisan arányos a tömb aktuális pozíciójától hátralévő elemek számával. Ha pedig az előbbi példában többszöri beszúrás után $n = 100$ lesz, további beszúrás már nem valósítható meg. (Bár ez utóbbi probléma – nem túl hatékonyan – megoldható pl. dinamikusan változtatható méretű tömbökkel.)

A *láncolt listák* a véges sorozatok tárolására egy alternatív megoldást kínálnak. Előnyük, hogy a sorrendtartó beszúrás és törlés hatékonyan, $\Theta(1)$ időben megoldható, ha a művelet pozíciója már megfelelőképpen adott. Hátrányuk, hogy a láncolt lista i. elemét a legrosszabb esetben csak $\Theta(i)$ idő alatt érhetjük el.

Mivel a tömbök és a láncolt listák is véges sorozatokat, azaz lineáris struktúrákat tárolnak[9], ezért ezeket *lineáris adatszerkezeteknek* nevezzük.

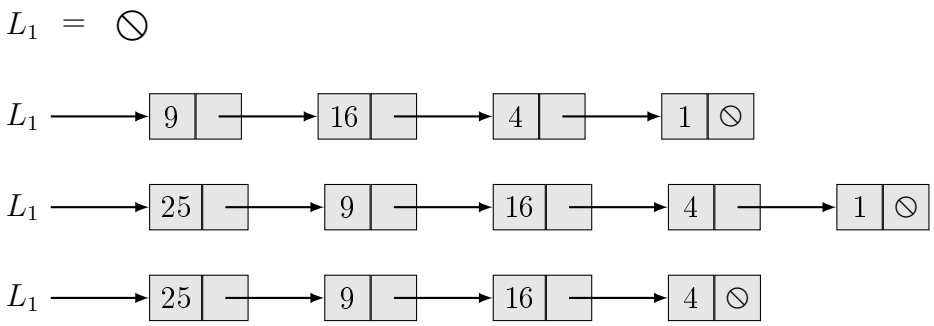
A *láncolt listák* legalapvetőbb típusai az *egyirányú* és a *kétirányú listák*. Míg az egyirányú listákon csak a lista elejétől a vége felé tudunk haladni, a kétirányú listákon visszafelé is mozoghatunk, ami néhány feladat megoldásánál előnyös lehet. Ennek az az ára, hogy adott hosszúságú nemüres sorozatot kétirányú listában tárolva több memóriára van szükségünk, mintha ugyanazt a sorozatot a neki megfelelő egyirányú listában tárolnánk, és a kétirányú listák esetében az elemi listamódosító műveletek (befűzés, kifűzés) is több értékadó utasításból állnak.

7.1. Egyirányú listák (one-way or singly linked lists)

Ebben az alfejezetben egyirányú listák két legfontosabb altípusa,
– az *egyszerű egyirányú listák (S1L = Simple One-way List)* és

[9]Ezen azt sem változtat, amikor halmazt vagy zsákot (multihalmazt) reprezentálunk velük, hiszen a reprezentáció – akaratunktól függetlenül – ekkor is sorrendiséget határoz meg a halmaz vagy zsák elemei között.

– a *fejelemes egyirányú listák (H1L = One-way List with Header node)* részletes tárgyalására kerül sor, de bevezetjük még a végelemes és a ciklikus egyirányú listákat is.



3. ábra. Az L_1 mutató egyszerű egyirányú listákat azonosít egy képzeletbeli program futásának különböző szakaszaiban. Az első sorban a lista, üres lista állapotában látható.

Az egyirányú listák elemeinek osztálya a következő:

E1
$+key : \mathcal{T}$ ... // satellite data may come here $+next$: **E1***
+**E1**() { $next := \oslash$ }

Itt az **E1*** olyan mutatót (pointert) jelöl, ami **E1** típusú objektum címét tartalmazhatja; vagy az értéke $\oslash$,[10] vagy definiálatlan.

Ha pl. adott a p : E1* pointer, ami egy egy **E1** típusú objektumra mutat, akkor a mutatott objektumot $*p$ jelöli. Ezután a mutatott objektum mezői (adattagjai) $(*p).key$ és $(*p).next$, amiket, a C/C++ nyelveket követve szemléletesebben $p \to key$ (a p által mutatott objektum key mezője) és $p \to next$ (a p által mutatott objektum $next$ mezője) alakban szokás írni.

7.1.1. Egyszerű egyirányú listák (S1L)

A 3. ábrának megfelelően az üres S1L-t $\oslash$ pointer azonosítja, míg a nemüres S1L azonosító pointere közvetlenül a lista első elemére mutat.

A 3. ábrán az első négy sorban egyszerű egyirányú listákra láthatunk példákat. Az első sorban az $L_1 = \oslash$ formulával az L_1 üres listát írtuk le. A második

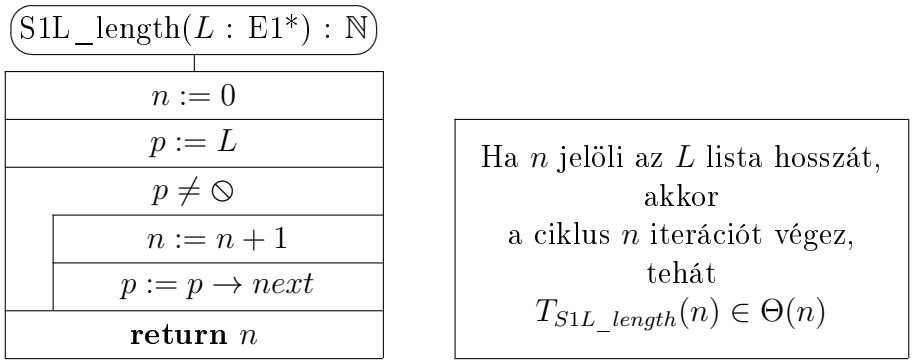
[10] A $\oslash$ szimbólum szokásos olvasatai a **null**, illetve a **nil**.

sorban látható egyszerű egyirányú lista a $\langle 9; 16; 4; 1\rangle$ sorozatot reprezentálja. Ennek megfelelően a lista négy db **E1** típusú objektumból, pontosabban listaelemből áll, amiknek a *key* mezői sorban tartalmazzák a $\langle 9; 16; 4; 1\rangle$ sorozat elemeit. Az L_1 pointer az első objektumra, azaz listaelemre mutat. Ennek megfelelően $L_1 \to key = 9$, és az $L_1 \to next$ pointer mutat a lista második elemére, amiből $L_1 \to next \to key = 16$ következik. Ha végrehajtjuk a $p := L_1 \to next \to next$ értékadást, akkor p a lista harmadik elemére mutat, $p \to key = 4$, $p \to next$ a negyedik listaelemre mutat, $p \to next \to key = 1$, $p \to next \to next = \oslash$, és p-re a $p := p \to next \to next$ értékadást is végrehajtva $p = \oslash$ lesz.

Ha $p = \oslash$, akkor a $*p$ kifejezés hibás, így a $p \to key$ és a $p \to next$ kifejezések is azok, kiértékelésük futási hibát eredményez. (C/C++-ban pl. ez a hiba a *segmentation violation* egyik esete.)[11]

Ha a p pointernek egyáltalán nem adunk értéket, akkor az értéke – ugyanúgy, mint más típusú változók esetében – definiálatlan: Lehet, hogy $p = \oslash$, de az is lehet, hogy $p \neq \oslash$; a p tulajdonképpen bármilyen memóriacímet tartalmazhat, de az is lehet, hogy nem létező memóriacímre hivatkozik. Ekkor a $*p$, a $p \to key$ és a $p \to next$ kifejezések is definiálatlanok, azaz nem tudhatjuk, hogy mi lesz a kiértékelésük eredménye.

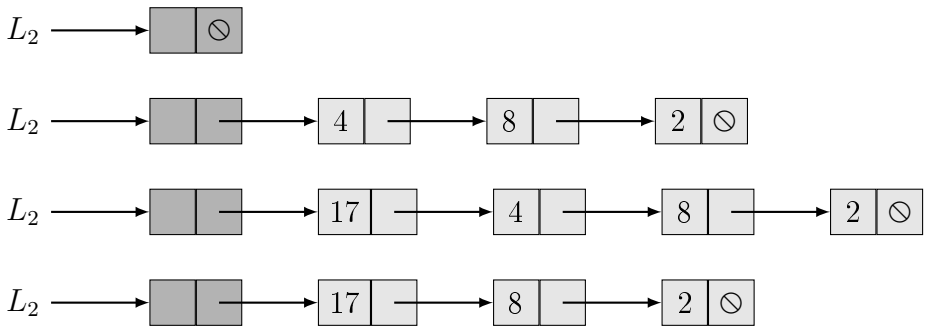
Első programozási példának megadjuk az S1L_length(L:E1*):$\mathbb{N}$ függvényt, ami az L S1L, azaz egyszerű egyirányú lista hosszát számolja ki. Ne felejtsük el, hogy absztrakt (azaz logikai) szinten L egy lista, míg konkrét (azaz fizikai) szinten L csak egy memóriacím, ami a listát *azonosítja*. Az, hogy az L:E1* pointer absztrakt szinten L S1L, tulajdonképpen a függvény által elvégzett számítás helyességének *előfeltétele*.



[11] A $p = \oslash$ esetet úgy is elképzelhetjük, hogy a p pointer egy úgynevezett *seholsincs objektum*ra (*NO = Nowhere Object*) mutat, ami teljesen üres, és így, ha a tartalmához akarunk hozzáférni, akkor nem létező dologra akarunk hivatkozni, ami szükségszerűen programfutási hibához vezet. (Tehát a *NO* címe $\oslash$, de ezen a címen csak a *semmit* találhatjuk.)

7.1.2. Fejelemes listák (H1L)

A fejelemes listák (H1L) szerkezete hasonló az S1L-ekéhez, de a H1L-ek mindig tartalmaznak egy nulladik, ún. fejelemet. A H1L-t a fejelemére mutató pointer azonosítja. A fejelem *key* mezője definiálatlan[12], a *next* pointere pedig a H1L-nek megfelelő S1L-t azonosítja. Ebből következik, hogy az üres H1L-nek is van fejeleme, aminek a *next* pointere $\oslash$. Fejelemes listákra a 4. ábrán láthatunk példákat.



4. ábra. Az L_2 fejelemes listákat azonosít egy képzeletbeli program futásának különböző szakaszaiban. Az első sorban a lista, üres lista állapotában látható.

A 4. ábra első sorában L_2 üres H1L, amit az mutat, hogy a fejelem *next* mezője $\oslash$, azaz $L_2 \to next = \oslash$. Ezenkívül $L_2 \to key$ definiálatlan.[13]

A 4. ábra második sorában L_2 három elemű H1L, mert a fejelemet nem számoljuk bele a fejelemes lista hosszába. $L_2 \to next$ mutat a H1L első elemére. Így $L_2 \to next \to key = 4$, $L_2 \to next \to next$ mutat a lista második elemére stb.

A következő H1L_length(H:E1*):$\mathbb{N}$ függvény tetszőleges H H1L hosszát számolja ki. Kihasználjuk, hogy $H \to next$ a H1L-nek megfelelő S1L.

H1L_length(H : E1*) : $\mathbb{N}$

return S1L_length($H \to next$)

$T_{H1L\_length}(n) \in \Theta(n)$
ahol n a H H1L hossza

7.1.3. Egyirányú listák kezelése

A különböző listaműveleteknél a listaelemekben levő kulcsok (és az esetleges egyéb járulékos adatok) listaelemek közti mozgatását kerüljük.

[12] esetleg a lista hosszát, vagy valamely egyéb tulajdonságát tartalmazhatja

[13] Ha a lista hosszát tartalmazná, itt $L_2 \to key = 0$ lenne.

Előnyben részesítjük a listák megfelelő átláncolását, mivel kerüljük a felesleges adatmozgatást, és *nem* tudjuk, hogy egy gyakorlati alkalmazásban mennyi járulékos adatot tartalmaznak az egyes listaelemek. (Lehet hogy a lista elemei E1 részosztályaihoz tartoznak.)

Alább, a baloldali kód tetszőleges egyirányú lista $*p$ eleme után fűzi a $*q$ objektumot. Feltesszük, hogy a végrehajtása előtt $*q$ éppen nincs listába fűzve. $T \in \Theta(1)$.

Alább, a jobboldali kódnál feltesszük, hogy $*p$ és $*q$ valamely egyirányú lista egymás utáni elemei, azaz $p \to next = q \neq \oslash$. Ezzel az előfeltétellel kifűzi a listából a $*q$ elemet. $T \in \Theta(1)$. Ha a $*q$ objektumot ezután azonnal listába fűzzük (átfűzés), akkor a $q \to next := \oslash$ utasítás elhagyható.

// Let $*p$ be followed by $*q$.
$q \to next := p \to next$
$p \to next := q$

// Provided that $*p$ is followed
// by $*q$, unlink $*q$.
$p \to next := q \to next$
$[q \to next := \oslash]$

A fentiek az egyirányú listák alapműveletei. Fejelemes egyirányú listák (H1L) esetén ezek segítségével minden összetett listamódosító művelet megadható (bár a kód optimalitása érdekében gyakran eltérünk ezektől az utasításpároktól). Egyszerű egyirányú listák (S1L) esetén még két további alapművelet szükséges, sorban a lista legelejére történő beszúrásra, illetve az első elem kifűzésére. (Ld. alább!) Emiatt a fejelemes listákat kezelő programok kódja gyakran kevesebb esetszétválasztást tartalmaz, mint a nekik megfelelő, egyszerű listákat kezelő programok, hiszen mindig valami után kell beszúrni, és mindig valami mögül kell kifűzni.

Cserébe, minden egyes fejelemes lista eggyel több objektumot tartalmaz, mint a neki megfelelő egyszerű lista, ami valamelyest megnöveli a program tárigényét. Ha egy alkalmazásban sok rövid listánk van, a tárigények különbsége szignifikáns lehet. Ezért pl. hasító táblákban (hash tables) sosem használunk fejelemes listákat.

// Insert $*q$ at the
// front of list L.
$q \to next := L$
$L := q$

// Unlink the first element of list L.
$q := L$
$L := q \to next$
$[q \to next := \oslash]$

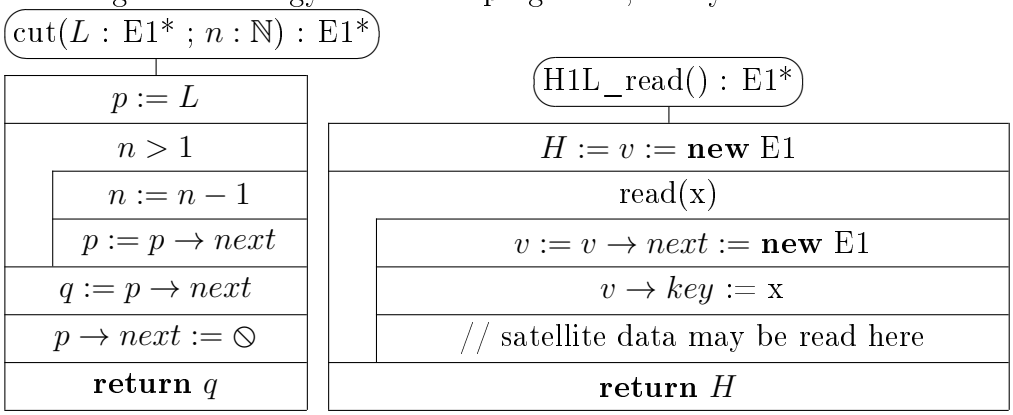
Az is lehet, hogy egy (rész)feladatban a listát mindig csak a legelején, azaz veremszerűen kell módosítani; vagy olyan a feladat, hogy a lista első

eleme biztosan a helyén marad. Ezekben az esetekben a fejelem csak akadály. Ha éppen van, célszerű a feladatot megoldó alprogramot a fejelemet követő egyszerű egyirányú listára (S1L) meghívni.

Akkor sem célszerű minden egyes lista elejére fejelemet generálni, ha egy program – mint pl. a láncolt listákra alkalmazott összefésülő rendezés (merge sort) – átmenetileg sok rövid listát hoz létre, hiszen így a fejelemek allokálása és deallokálása a futási időt jelentősen megnövelheti (hacsak nem élünk valamilyen ügyes trükkel :).

Olyan eset is lehet, amikor a fejelem helyett ún. *végelemet* célszerű alkalmazni, azaz a lista végén van egy olyan elem, aminek a kulcsát nem használjuk, és az üres lista csak egy végelemet tartalmaz. Erre példa a sorok láncolt, optimális megvalósítása: Ilyenkor a listára két külső pointer mutat, az egyik a lista első, a másik a végelemére (*trailer*). (A részleteket itt is az Olvasóra bízzuk.) A láncolt listákon szereplő extra elemeket, mint a fejelem vagy a végelem, és más, a lista egy-egy szakaszát határoló listaelemeket összefoglaló néven *őrszem* (*sentinel*) elemeknek hívjuk.

Most még röviden tárgyalunk két alprogramot, amelyek hasznosak lesznek.



A cut(L:E1*;n:ℕ):E1* függvény kettévágja az L S1L-t. Az első n elemét hagyja L-ben, és visszaadja a lista levágott maradékát azonosító pointert. A listaelemek sorrendjét megtartja. $T_{cut}(n) \in \Theta(n)$.

A *H1L_read*():E1* függvény beolvas egy adatsort a kurrens inputról, a bemenet sorrendje szerint egy H1L-t épít belőlük, és visszaadja a fejeleme címét. Feltesszük, hogy a read(&x:$\mathcal{T}$):$\mathbb{B}$ függvény képes a következő adat beolvasására x-be, és akkor ad vissza igaz értéket, ha a beolvasás előtt még volt adat a bemeneten. (Különben x definiálatlan marad, és a függvény hamis értéket ad vissza.) $T_{H1L\_read}(n) \in \Theta(n)$, ahol n a már felépített H1L hossza.

7.1. Feladat. *Próbáljuk megírni az S1L_read():E1* függvényt, amely beol-*

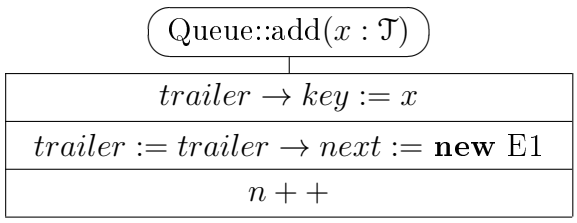
vas egy adatsort a kurrens inputról, a bemenet sorrendje szerint egy S1L-t épít belőlük, és visszaadja az első eleme címét, vagy $\oslash$-t, ha az input üres volt! Tartsuk meg a $\Theta(n)$ műveletigényt! Egyszerűbb vagy bonyolultabb lett a kód, mint a H1L_read():E1 függvény esetében? Miért?*

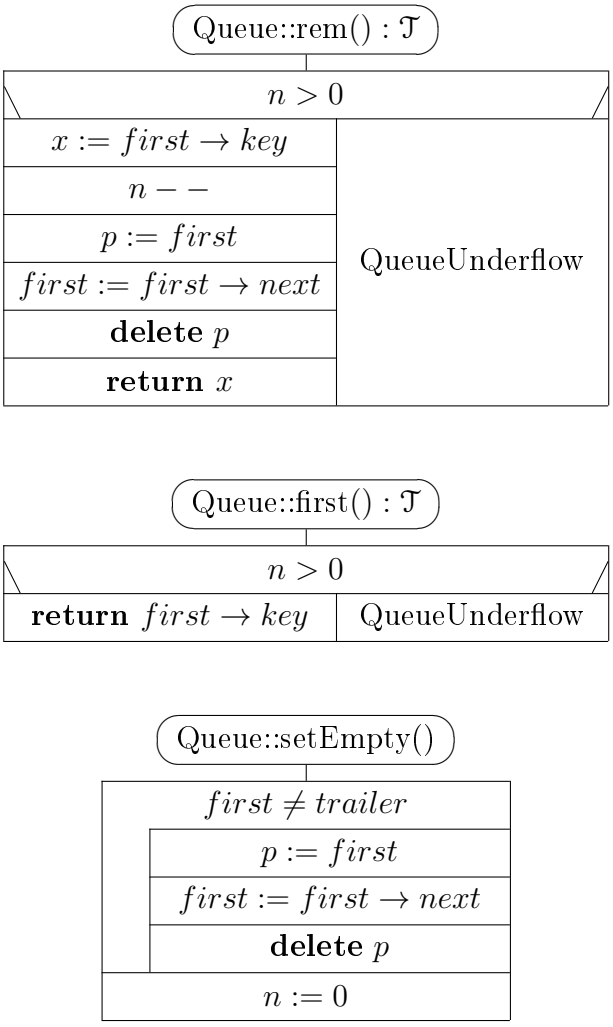
7.2. Példa. *Valósítsuk meg a Queue osztályt egyirányú, végelemes listával! A listára két külső pointer mutat, az egyik ($first$) az első, a másik ($trailer$) a (nemdefiniált kulcsú) végelemére. A sor összes műveletének műveletigénye $\Theta(1)$ legyen, a setEmpty() és a destruktor kivételével. Ez utóbbiakat $\Theta(n)$ műveletigénnyel valósítsuk meg, ahol n a kiürítendő sor hossza. A* **new** *és a* **delete** *utasítások műveletigényeit $\Theta(1)$-nek tekintjük. (Bővebben ld. erről a 7.1.4. alfejezetet!) Mint általában, most is feltesszük, hogy a* **new** *utasítások végrehajtásához elegendő memória áll rendelkezésre.*

Megoldás:

Queue
$-$ $first, trailer$: E1* // a one-way list with trailer represents the queue $-$ $n : \mathbb{N}$ // n is the actual length of the queue

+ Queue(){ $first := trailer :=$ **new** E1 ; $n := 0$ } // create an empty queue
+ add($x : \mathcal{T}$) // join x to the end of the queue
+ rem() : $\mathcal{T}$ // remove and return the first element of the queue
+ first() : $\mathcal{T}$ // return the first element of the queue
+ length() : $\mathbb{N}$ {**return** n}
+ isEmpty() : $\mathbb{B}$ {**return** $n = 0$}
+ setEmpty() // reinitialize the queue
+ $\sim$ Queue() { setEmpty() ; **delete** $trailer$ }





□

7.3. Feladat. *Az L pointer egy monoton növekvően rendezett egyszerű láncolt lista (S1L) első elemére mutat ($L \neq \oslash$).*

Írjuk meg a duplDel(L,&D:E1) eljárást, ami a duplikált adatoknak csak az első előfordulását hagyja meg! $T(n) \in O(n)$, ahol n az L lista hossza (n a program számára nem adott). A lista tehát szigorúan monoton növekvő lesz. A feleslegessé váló elemekből hozzuk létre a D pointerű, monoton csökkenő, egyszerű láncolt listát!*

7.4. Feladat. *Adott az $A : \mathcal{T}[n]$ tömb.*

Írjuk meg a listaba($A : \mathcal{T}[]$; &H:E1) eljárást, ami előállítja az A tömb által reprezentált absztrakt sorozat láncolt ábrázolását a H H1L-ben. A szükséges listaelemeket az ismert* `new E1` *művelet segítségével nyerjük. Feltesszük,*

hogy ha nincs elég memória, akkor a `new` *művelet null pointert ad vissza. Ebben az esetben írjuk ki azt, hogy „Memory Overflow!", aztán azonnal fejezzük be az eljárást! Ilyenkor a H listában csak azok az elemek legyenek, amelyeket sikeresen generáltunk!* $T_{\text{listaba}}(n) \in O(n)$ *legyen! (Feltesszük, hogy* $T_{\text{new}} \in \Theta(1)$*.)*

7.5. Feladat. *Adott az* $A : \mathfrak{T}[n]$ *tömb.*

Írjuk meg a listaba($A : \mathfrak{T}[]$ *; &L:E1*) eljárást, ami előállítja az A tömb által reprezentált absztrakt sorozat láncolt ábrázolását az L S1L-ben. A szükséges listaelemeket az ismert* `new E1` *művelet segítségével nyerjük. Feltesszük, hogy ha nincs elég memória, akkor a* `new` *művelet null pointert ad vissza. Ebben az esetben írjuk ki azt, hogy „Memory Overflow!", aztán azonnal fejezzük be az eljárást! Ilyenkor az L listában csak azok az elemek legyenek, amelyeket sikeresen generáltunk!* $T_{\text{listaba}}(n) \in O(n)$ *legyen! (Feltesszük, hogy* $T_{\text{new}} \in \Theta(1)$*.)*

7.6. Feladat. *Tekintsük a H fejpointerű H1L rendezését a következő algoritmussal! Először megkeressük a lista minimális elemét, majd átfűzzük a fejelem után. Ezután megkeressük a második legkisebb elemét és átfűzzük az első minimum után. Folytassuk ezen a módon a lista első* $(n-1)$ *elemére! Pl.:*

$\langle 3;9;7;1;6;2\rangle \to \langle 1 \ / \ 9;7;3;6;2\rangle$
$\to \langle 1;2 \ / \ 7;3;6;9\rangle \to \langle 1;2;3 \ / \ 7;6;9\rangle$
$\to \langle 1;2;3;6 \ / \ 7;9\rangle \to \langle 1;2;3;6;7 \ / \ 9\rangle$
$\to \langle 1;2;3;6;7;9\rangle$

Írjunk struktogramot erre a – minimumkiválasztásos rendezés néven közismert – algoritmusra MinSelSort(H:E1) néven! Mi lesz a fő ciklus invariánsa? Miért elég csak az első* $(n-1)$ *elemre lefuttatni? (*n *= a lista hossza.) Adjuk meg az* $MT(n)$ *és* $mT(n)$ *függvényeket a minimumkiválasztásos rendezésre az ismert* Θ*-jelöléssel!*

7.7. Feladat. *Adott a P:E1*[n] pointer tömb, amelynek elemei egyszerű láncolt listákat azonosítanak. (Mindegyik vagy* $\oslash$ *pointer, vagy egy lista első elemére mutat.)*

Írjuk meg az összefűz(P, L) *eljárást, ami a listákat sorban egymás után fűzi az L egyszerű listába. (A formális paraméterek pontos specifikálása a feladat része.)* $T(m, n) \in O(m+n)$*, ahol m az eredeti listák összhossza (m a program számára ismeretlen).*

7.8. Feladat. *Az L pointer egy rendezetlen, egyszerű láncolt lista első elemére mutat. Feltehető, hogy a lista nem üres.*

Írjuk meg a MaxRend(L) eljárást, ami a listát monoton növekvően, L hosszától független, konstans mennyiségű memóriafelhasználással, maximumkiválasztással rendezi! (A formális paraméter pontos specifikálása a feladat része.) $MT(n) \in O(n^2)$, ahol n az L lista hossza (n a program számára nem adott).

[Felveszünk egy rendezett, kezdetben üres segédlistát. (Így inicializáljuk.) A rendezés menetekből áll. Minden menetben kiválasztjuk a rendezetlen lista legnagyobb elemét, és átfűzzük a rendezett lista elejére.]

7.9. Feladat. *Legyenek H_u és H_i szigorúan monoton növekvően rendezett H1L-ek! Írjuk meg a unionIntersection(H_u, H_i : E1*) eljárást, ami a H_u listába H_i megfelelő elemeit átfűzve, a H_u listában az eredeti listák – mint halmazok – unióját állítja elő, míg a H_i listában a metszetük marad! Ne allokáljunk és ne is deallokáljunk listaelemeket, csak az listaelemek átfűzésével oldjuk meg a feladatot! $MT(n_u, n_i) \in \Theta(n_u+n_i)$, ahol a H_u H1L hossza n_u, a H_i H1L hossza pedig n_i. Minkét lista maradjon szigorúan monoton növekvően rendezett H1L!*

7.1.4. Dinamikus memóriagazdálkodás

Az objektumok dinamikus létrehozására a **new** $\mathcal{T}$ műveletet fogjuk használni, ami egy $\mathcal{T}$ típusú objektumot hoz létre, és visszaadja a címét. Ezért tudunk egy vagy több mutatóval hivatkozni a frissen létrehozott objektumra. Az objektumok dinamikus létrehozása egy speciálisan a dinamikus helyfoglalásra fenntartott memóriaszegmens szabad területének rovására történik. Fontos ezért, hogy ha már egy objektumot nem használunk, a memória neki megfelelő darabja visszakerüljön a szabad memóriához. Erre fogjuk használni a **delete** p utasítást, ami a p mutató által hivatkozott objektumot törli.

Maga a p mutató a $p :=$ **new** $\mathcal{T}$ utasítás végrehajtása előtt is létezik, mert azt a mutató deklarációjának kiértékelése hozza létre.[14] Ugyanígy, a p mutató a **delete** p végrehajtása után is létezik, egészen az őt (automatikusan) deklaráló eljárás vagy függvény végrehajtásának befejezéséig.

Mi magunk is írhatunk optimalizált, hatékony dinamikus memória helyfoglaló és felszabadító rutinokat; speciális esetekben akár úgy is, hogy a fenti műveletek konstans időt igényelnek. (Erre egy példa a gyakorlatokon is elhangzik.)

Általános esetben azonban a dinamikus helyfoglalásra használt szabad memória a különböző típusú és méretű objektumokra vonatkozó létrehozá-

[14] Az absztrakt programokban (struktogram, pszeudokód) automatikus deklarációt feltételezünk: az eljárás vagy függvény meghívásakor az összes benne használt lokális skalár változó automatikusan deklarálódik (egy változó alapértelmezésben lokális).

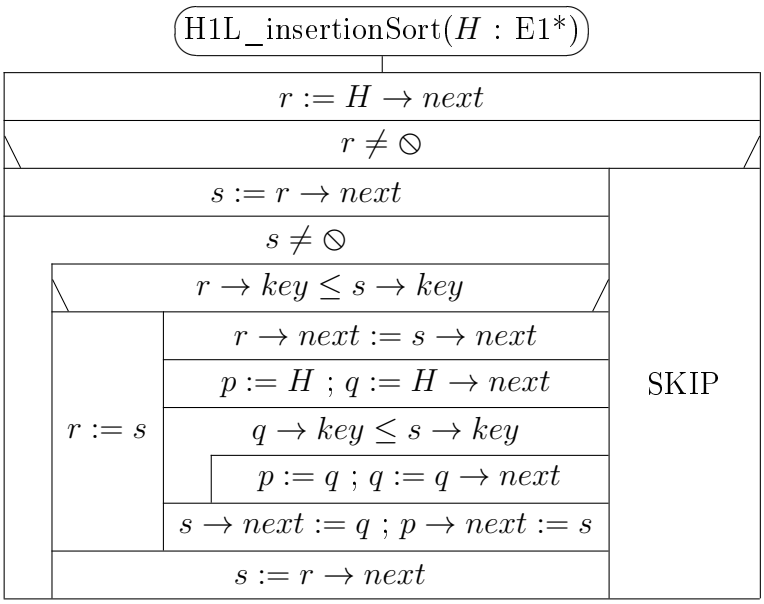
sok és törlések (itt **new** és **delete** utasítások) hatására feldarabolódik, és viszonylag bonyolult könyvelést igényel, ha a feldarabolódásnak gátat akarunk vetni. Ezt a problémát a különböző rendszerek különböző hatékonysággal kezelik.

Az absztrakt programokban az objektumokat dinamikusan létrehozó (**new**) és törlő (**delete**) utasítások műveletigényeit konstans értékeknek, azaz $\Theta(1)$-nek vesszük, azzal a megjegyzéssel, hogy valójában nem tudjuk, mennyi. Ezért a lehető legkevesebbet használjuk a **new** és a **delete** utasításokat.

A tömbök dinamikus létrehozásáról és törléséről ld. (3.1)!

7.1.5. Beszúró rendezés H1L-ekre

Ismertetjük a fejelemes listákra a beszúró rendezést. Látható, hogy a rendezett beszúrás művelete egyszerűbb, mintha egyszerű egyirányú listára írtuk volna meg, mert a lista elejére való beszúrást nem kell külön kezelni.



7.10. Feladat. *Lássuk be a fenti algoritmus helyességét, és bizonyítsuk be, hogy a műveletigénye, ugyanúgy, mint a tömbös változaté:*

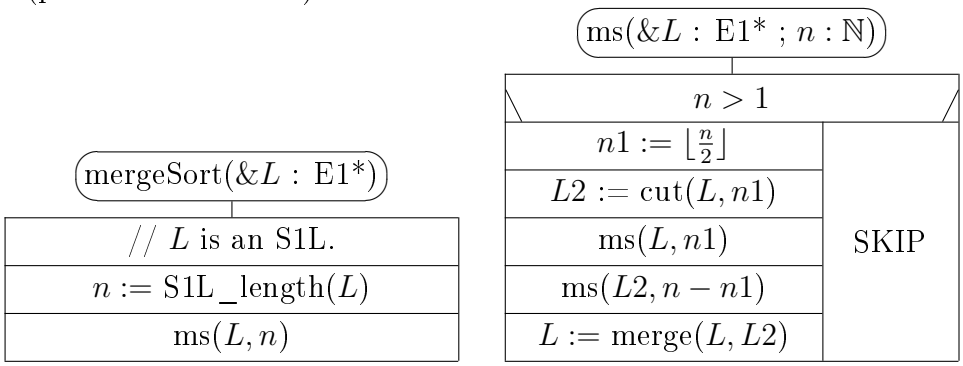
$$mT_{IS}(n) \in \Theta(n)$$

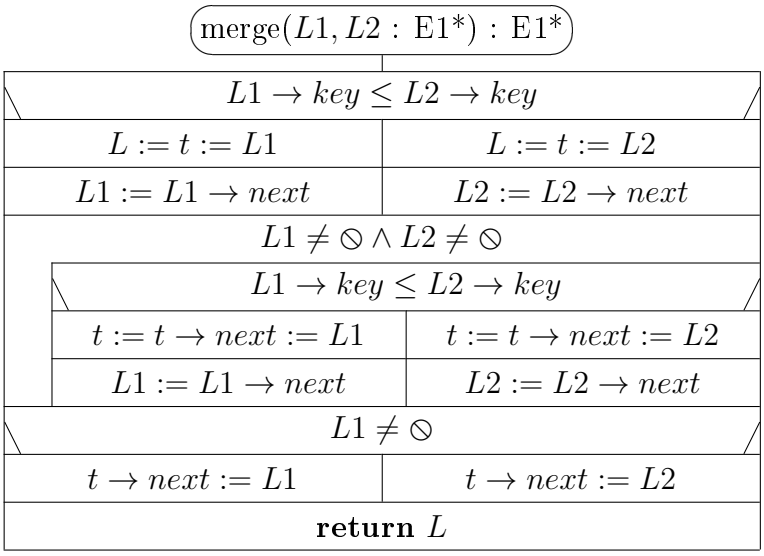
$$AT_{IS}(n), MT_{IS}(n) \in \Theta(n^2)$$

ahol n a H fejelemes lista hossza. Gondoljuk meg azt is, hogy mi biztosítja a rendezés stabilitását!

7.1.6. Az összefésülő rendezés S1L-ekre

A merge sort-ot most egyszerű (egyirányú) listákra írjuk meg. Ha fejelemes listákkal dolgoznánk, itt a listák darabolása során sok fejelemet kellene kezelni, ami nem lenne hatékony, már amennyiben a fejelemeket dinamikusan kell (pl. **new** utasítással) létrehozni.





7.11. Feladat. *Lássuk be a fenti algoritmus helyességét, és indokoljuk, hogy a műveletigényére, hasonlóan, mint a tömbös változatéra*

$$mT_{MS}(n), MT_{MS}(n) \in \Theta(n \log n)$$

teljesül, ahol n az L lista hossza. Gondoljuk meg a stabilitást is!

7.12. Feladat. *A fenti merge($L1, L2$) függvény szimmetrikus. Próbáljunk aszimmetrikus megoldás adni, ami az $L1$ lista elemei közé fésüli $L2$ elemeit! Melyik megoldás adott gyorsabb kódot?*

7.1.7. Ciklikus egyirányú listák

Ciklikus esetben az utolsó listaelem *next* mezője nem a $\oslash$-t tartalmazza, hanem visszamutat a lista elejére. Ciklikus egyirányú listák segítségével jól lehet pl. körkörös vagy sor jellegű absztrakt struktúrákat reprezentálni.

Fejelem nélküli nemüres lista esetén az utolsó elem *next* pointere az első listaelemre mutat, és a listára kívülről gyakran az utolsó elemén keresztül érdemes hivatkozni (ha egyáltalán értelmezzük az első és utolsó elem fogalmát ebben az esetben). Üres lista esetén viszont a fejelem nélküli ciklikus listát a $\oslash$ reprezentálja.

Fejelemes egyirányú ciklikus lista esetén az utolsó listaelem *next* pointere a fejelemre mutat, speciálisan üres lista esetén tehát a fejelem *next* pointere visszamutat a fejelemre. A ciklikus listák esetében a fejelemes és a végelemes lista ugyanazt jelenti, hiszen ez a két őrszem ugyanaz. A listát azonosító külső pointer az őrszemre mutat.

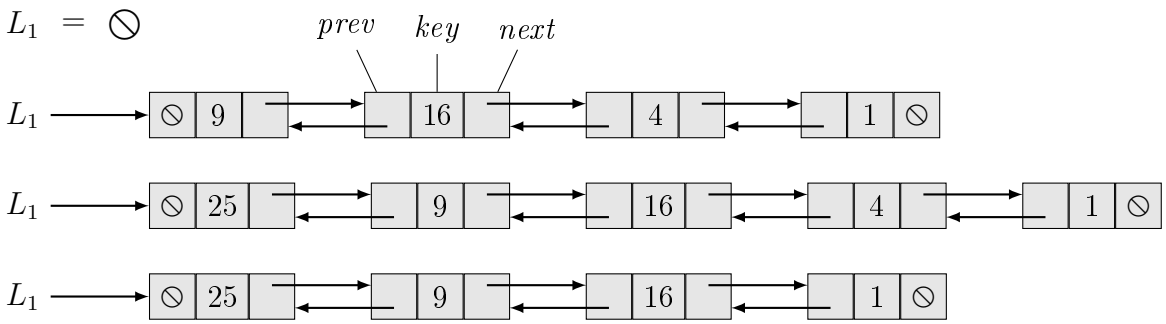
7.13. Feladat. *A 7.2. példát és megoldását mintának véve, valósítsuk meg a* Queue *(sor) adattípust (ld. (6.2) alfejezet)*
– egyszerű láncolt listával (S1L) úgy, hogy amennyiben a lista nemüres, az első elemére az L, az utolsó elemére a t pointer mutat, ha viszont a lista üres, akkor $L = \oslash$, t értéke pedig nem definiált;
– őrszemes egyirányú ciklikus listával;
– őrszem nélküli egyirányú ciklikus listával!
Tartsuk meg a különböző megvalósítások mindegyik műveletére a $\Theta(1)$ műveletigényt, kivéve a setEmpty() műveletet és a destruktort, ahol $\Theta(n)$ lesz az új műveletigény, n-nel a sor hosszát jelölve!

Hasonlítsuk össze a négy reprezentációt (adatszerkezetet) és implementációt (műveleteket) egymással, majd a tömbös megvalósítással (6.2)!

7.2. Kétirányú listák (two-way or doubly linked lists)

Ebben az alfejezetben a kétirányú listák két legfontosabb altípusa,
– az *egyszerű kétirányú listák (S2L = Simple Two-way List)* és
– a *fejelemes ciklikus kétirányú listák (C2L = Cyclic Two-way List with header)*
tárgyalására kerül sor, ez utóbbira részletesen.

A kétirányú listák elemeiben a *next* pointer mellett találhatunk egy *prev* pointert is, ami a lista megelőző elemére mutat.



5. ábra. Az L_1 mutató egyszerű kétirányú listákat (S2L) azonosít egy képzeletbeli program futásának különböző szakaszaiban. Az első sorban a listát üresre inicializáló értékadás látható.

7.2.1. Egyszerű kétirányú listák (S2L)

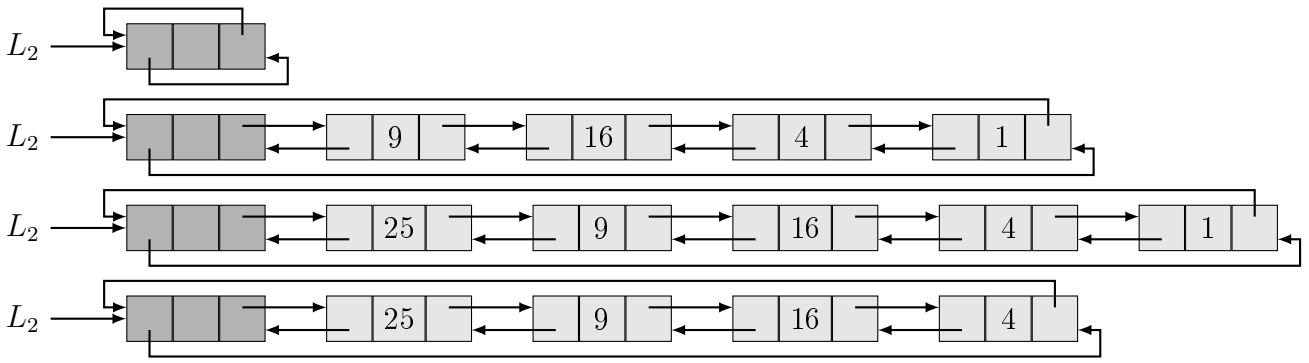
Az 5. ábra 1. sorában Az L_1 listát üresre inicializátuk, a második sorban a $\langle 9; 16; 4; 1 \rangle$ sorozatot rerezentálja. A harmadik sorban beszúrtuk még a 25 kulcsú kétirányú listaelemet a lista elejére. A negyedik sorban töröltük az utolsó előtti elemét. Látható, hogy a lista módosításai során különbözőképpen kell kezelni a lista első, utolsó és közbülső elemeit. Mindazonáltal a hasító tábláknál ez a listatípus bizonyul majd célszerűbbnek a *fejelemes ciklikus kétirányú listákhoz* képest, amiket a következő alfejezetekben tárgyalunk, és ahol a listamódosító műveletek egyszerűbbek és hatékonyabbak, mint ennél a listatípusnál.

Mivel azonban a listaelemek beszúrása és eltávolítása is másképpen megy a listák elején, végén és közbül, ezen kényelmetlenség miatt, és plusz futási idő miatt ezt a listatípust a hasító táblákon kívül viszonylag ritkán használjuk.

7.2.2. Ciklikus kétirányú listák (C2L)

A *fejelemes* és a *fejelem nélküli* **ciklikus kétirányú listák (C2L)** elemeinek osztálya, és az alapvető listakezelő műveleteik is ugyanazok. Általában szoktunk használni fejelemet, mert így a listakezelés tovább egyszerűsödik. Nem kell ugyanis külön kezelni az üres listába való beszúrást (hiszen az is tartalmaz már egy fejelemet) és az utolsó listaelem törlését sem (ui. a fejelem akkor is a listában marad). Az alábbiakban ezért **C2L** alatt alapértelmezésben **fejelemes ciklikus kétirányú listát** értünk.[15]

[15]Ha a programunk sok rövid listát használ, és takarékoskodni szeretnénk a memóriával, ennek ellenére célszerűbb fejelem nélküli C2L-eket, vagy esetleg S2L-eket használni.



6. ábra. Az L_2 mutató fejelemes kétirányú ciklikus listákat (C2L) azonosít egy képzeletbeli program futásának különböző szakaszaiban. Az első sorban az L_2 lista üres állapotában látható.

A ciklikus kétirányú listák (C2L) elemeinek osztálya és a listák alapvető műveletei következők. Vegyük észre, hogy mindegyik műveletigény $\Theta(1)$.

E2
$+prev, next$: **E2*** // refer to the previous and next neighbour or be **this** $+key : \mathcal{T}$
+ E2() { $prev := next :=$ **this** }

precede(q, r : E2*)

// $(*q)$ will precede $(*r)$
$p := r \to prev$
$q \to prev := p$; $q \to next := r$
$p \to next := r \to prev := q$

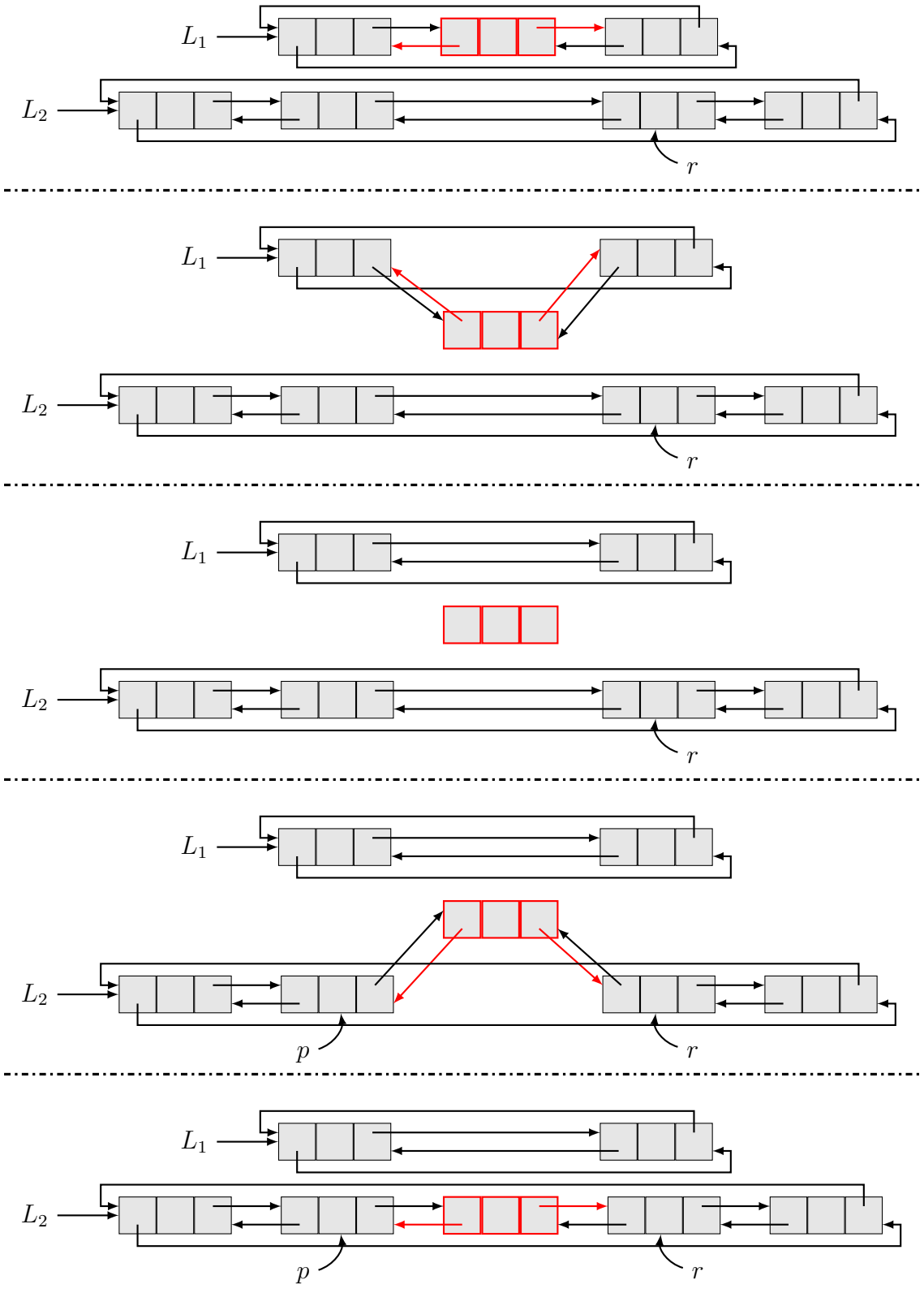
follow(p, q : E2*)

// $(*q)$ will follow $(*p)$
$r := p \to next$
$q \to prev := p$; $q \to next := r$
$p \to next := r \to prev := q$

unlink(q : E2*)

// remove $(*q)$
$p := q \to prev$; $r := q \to next$
$p \to next := r$; $r \to prev := p$
$q \to prev := q \to next := q$

Az unlink(q) és a precede(q, r) műveletek szemléltetése a 7. ábrán található.



7. ábra. Az unlink(q) és a precede(q, r) metódus szemléltetése. A piros listaelemre ($*q$) hívjuk meg a beszúrást, és ($*r$)-től balra kerül beszúrásra L_2 listába.

7.14. Megjegyzés. *Vegyük észre, hogy a fenti a precede(q, r) vagy a follow(p, q) elemi listaműveletek bármelyike az unlink(q) eljárással együtt már elég erős ahhoz, hogy segítségükkel a C2L-eken belül tetszőleges listaátalakítás megoldható úgy, hogy a listaelemek prev és next pointereinek explicit módosítására nincs szükség.*

7.15. Feladat. *Definiálja az egyszerű kétirányú listák (S2L) elemtípusát, és a fenti három művelet megfelelőit. Milyen plusz paraméterekre lesz szükség az egyes műveleteknél? Tartsa mindháromnál a $\Theta(1)$ műveletigényt!*

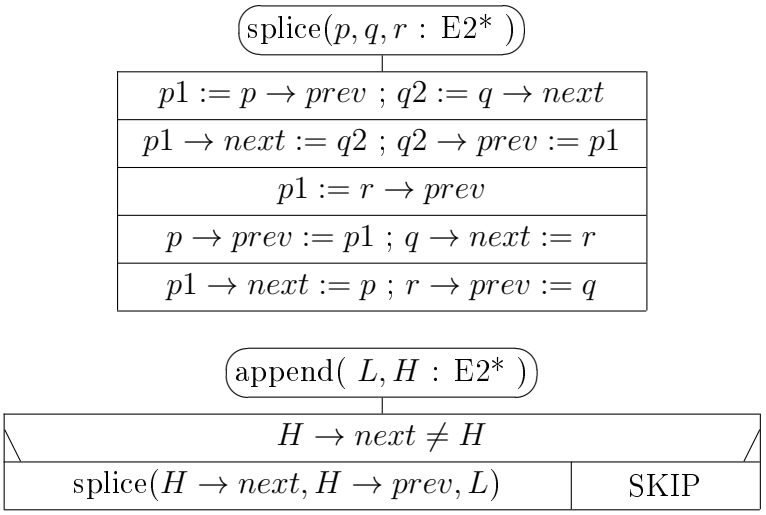
7.16. Példa. *Tegyük fel, hogy $*p$ és $*q$ ugyanannak a C2L-nek az elemei, és $*p$ után jön valahol $*q$ vagy speciálisan $p = q$ is lehet, de a C2L $[p, \ldots, q]$ szakasza nem tartalmazza sem a fejelemet, sem a $*r$ elemet, ami akár ennek, akár egy másik C2L-nek is lehet eleme!*

Definiálja a splice(p, q, r) elemi listaműveletet, amely a fenti előfeltétellel tetszőleges C2L egy adott $[p, \ldots, q]$ szakaszát eltávolítja, majd az eltávolított szakaszt egy C2L adott $*r$ eleme elé fűzi a listába!

Definiálja erre építve az append(L, H) eljárást is, ami átfűzi az L C2L végére a H C2L elemeit, az eredeti sorrendben! Mindkettő műveletigénye $\Theta(1)$ legyen!

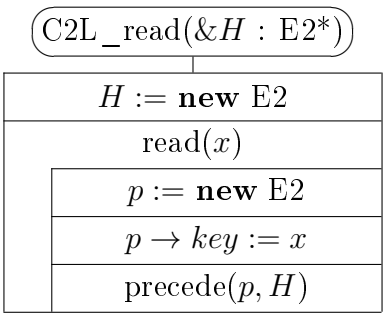
Vegyük észre, hogy a $\Theta(1)$ műveletigény csak úgy tartható be, ha megengedjük a listaelemek prev és next pointereinek explicit módosítását! Mekkora lesz a műveletigény, ha az átláncolások csak a fentebb definiált precede(q, r), follow(p, q) és unlink(q) eljárások segítségével végezhetők?

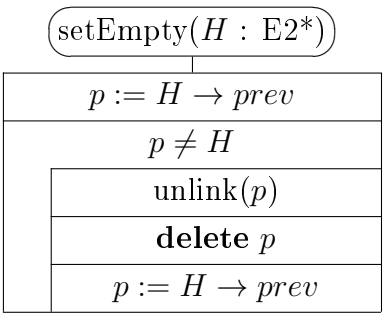
Megoldás vázlata:



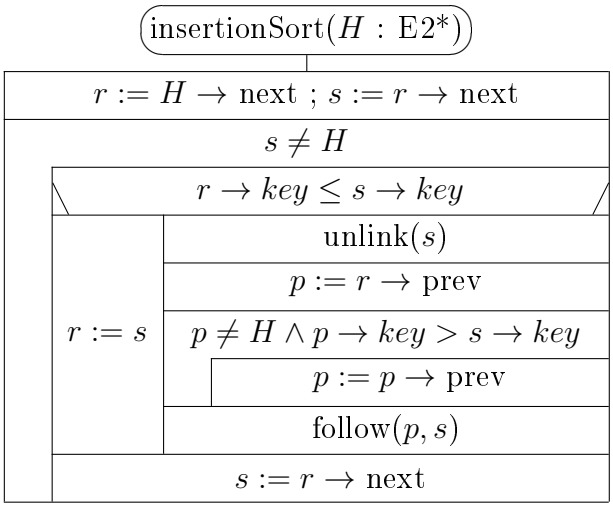
7.2.3. Példaprogramok fejelemes, ciklikus kétirányú listákra (C2L)

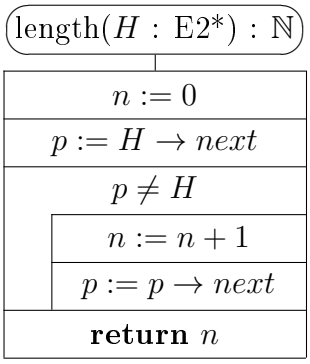
Tetszőleges C2L elképzelhető kiegyenesítve. Így a fejelem kétszer, a lista elején és a végén is megjelenik. (Ld. alább!) A listaelemek közti szimmetrikus kapcsolatot irányítatlan élekkel szimbolizáljuk. A follow(p, q), precede(q, r) és az unlink(q) elemi listaműveletekkel így a listakezelést absztraktabban, ugyanakkor egyszerűbben képzelhetkük el, például
$H \to$ [/]—[5]—[2]—[7]—[/] $\leftarrow H$ a $\langle 5; 2; 7 \rangle$ sorozat egy lehetséges ábrázolása,
$H \to$ [/]—[/] $\leftarrow H$ a $\langle\rangle$ üres sorozaté.





$H \to$ [/]—[/] $\leftarrow H$
$H \to$ [/]—[5]—[/] $\leftarrow H$
$H \to$ [/]—[5]—[2]—[/] $\leftarrow H$
$H \to$ [/]—[5]—[2]—[7]—[/] $\leftarrow H$





$H \to$ [/]—[5]—[2]—[7]—[2]—[/] $\leftarrow H$
$H \to$ [/]—[2]—[5]—[7]—[2]—[/] $\leftarrow H$
$H \to$ [/]—[2]—[5]—[7]—[2]—[/] $\leftarrow H$
$H \to$ [/]—[2]—[2]—[5]—[7]—[/] $\leftarrow H$

7.17. Feladat. *Lássuk be a fenti algoritmusok helyességét, és bizonyítsuk be, hogy a műveletigényekre, ugyanúgy, mint a korábbi változatokéra, az alábbi állítások teljesülnek!*

$$T_{C2L\_read}(n), T_{setEmpty}(n), T_{length}(n) \in \Theta(n)$$

$$mT_{IS}(n) \in \Theta(n)$$

$$AT_{IS}(n), MT_{IS}(n) \in \Theta(n^2)$$

ahol n a H fejelemes lista hossza (a C2L_ read(&H : E2)-nél az eljáráshívás után), és* IS *az* insertionSort *rövidítése. Gondoljuk meg azt is, hogy mi biztosítja a beszúró rendezés stabilitását!*

7.18. Feladat. *Adott az $A : \mathcal{T}[n]$ tömb.*

Írjuk meg a listaba($A : \mathcal{T}[]$; &H:E2) eljárást, ami előállítja az A tömb által reprezentált absztrakt sorozat láncolt ábrázolását a H C2L-ben. A szükséges listaelemeket az ismert* `new E2` *művelet segítségével nyerjük. Feltesszük, hogy ha nincs elég memória, akkor a* `new` *művelet null pointert ad vissza. Ebben az esetben írjuk ki azt, hogy „Memory Overflow!", aztán azonnal fejezzük be az eljárást! Ilyenkor a H listában csak azok az elemek legyenek, amelyeket sikeresen generáltunk! $T_{\text{listaba}}(n) \in O(n)$ legyen! (Feltesszük, hogy $T_{\text{new}} \in \Theta(1)$.)*

7.19. Feladat. *A H pointer egy monoton növekvően rendezett nemüres C2L fejelemére mutat. A D pointer egy üres C2L fejelemére mutat.*

Írjuk meg a duplDel(H, D:E2) eljárást, ami a duplikált adatoknak csak az első előfordulását hagyja meg! $T(n) \in O(n)$, ahol n a H lista hossza (n a program számára nem adott). A lista tehát szigorúan monoton növekvő lesz. A feleslegessé váló elemekből hozzuk létre a D fejpointerű, monoton növekvő C2L-t! (A fejelem már megvan.)*

7.20. Feladat. *Tekintsük a H fejpointerű C2L rendezését a következő algoritmussal! Először megkeressük a lista maximális elemét, majd átfűzzük a fejelem elé. Ezután megkeressük a második legnagyobb elemét és átfűzzük az első maximum elé. Folytassuk ezen a módon, összesen $(n-1)$ maximumkeresést végezve! (n = a lista hossza.) Itt tehát a listát egy rendezetlen és egy rendezett szakaszra bontjuk, ahol a rendezett szakasz a lista végén kezdetben üres. Mindig a rendezetlen szakaszon keresünk maximumot, és a maximális kulcsú elemet átfűzzük a rendezett szakasz elejére. Pl.:*

$\langle 3;1;9;2;7;6\rangle \to \langle 3;1;6;2;7 \mid 9\rangle$
$\to \langle 3;1;6;2 \mid 7;9\rangle \to \langle 3;1;2 \mid 6;7;9\rangle$
$\to \langle 2;1 \mid 3;6;7;9\rangle \to \langle 1 \mid 2;3;6;7;9\rangle \to \langle 1;2;3;6;7;9\rangle$

Írjunk struktogramot erre a – maximumkiválasztásos rendezés néven közismert – algoritmusra MaxSelSort(H:E2) néven! Mi lesz a fő ciklus invariánsa? Miért elég csak az első $(n-1)$ elemre lefuttatni? Adjuk meg az $MT(n)$ és $mT(n)$ függvényeket a maximumkiválasztásos rendezésre az ismert Θ-jelöléssel!*

7.21. Feladat. *Tekintsük az H fejpointerű, E2 elemtípusú C2L rendezését a következő algoritmussal! Először megkeressük a lista minimális elemét, majd átfűzzük a fejelem után. Ezután megkeressük a második legkisebb elemét és átfűzzük az első minimum után. Folytassuk ezen a módon a lista első $(n-1)$ elemére (ahol n jelöli a lista elemeinek számát)! Pl.:*

$<\ 3;9;7;1;6;2\ > \rightarrow <\ 1;\ 3;9;7;6;2\ >$
$\rightarrow <\ 1;2;\ 3;9;7;6\ > \rightarrow <\ 1;2;3;\ 9;7;6\ >$
$\rightarrow <\ 1;2;3;6;\ 9;7\ > \rightarrow <\ 1;2;3;6;7;\ 9\ >$

Írjunk struktogramot erre a – minimumkiválasztásos rendezés néven közismert – algoritmusra, minKivRend(H) néven (n a program számára nincs megadva)! Mi lesz a fő ciklus invariánsa? Miért elég csak az első $(n-1)$ elemre lefuttatni? Adjuk meg az $MT(n)$ és $mT(n)$ függvényeket a minimumkiválasztásos rendezésre az előadásról ismert Θ-jelöléssel, ahol n a lista hossza!

7.22. Feladat. *A H pointer egy C2L fejelemére mutat. P:E2*[k] definiálatlan pointerek tömbje, m definiálatlan egész szám.*

Írjuk meg a növBont(L, P, m) eljárást, ami meghatározza és m-ben visszaadja a H-beli, monoton növekvően rendezett szakaszok számát. Ezen szakaszok első elemeire az eljárás végrehajtásának eredményeként sorban a P tömb $P[1\,..\,m]$ résztömbjének elemei mutatnak. $MT(n) \in O(n)$, ahol n az L lista hossza (a program számára nem *adott).*

Feltehető, hogy P-nek elég sok eleme van, azaz az eljárás által kiszámolt m értékre $k \geq m$. A szigorúan monoton csökkenő részeket egyelemű monoton növekvő szakaszok sorozatának tekintjük. Pl. $\langle 5,6,4,2,1,3\rangle$ monoton növekvő szakaszai $[\langle 5,6\rangle, \langle 4\rangle, \langle 2\rangle, \langle 1,3\rangle]$, ahol $m = 4$. Az eljárás az L listát nem változtatja meg, csak m-et és $P[1\,..\,m]$-et állítja be.

7.23. Példa. *Legyenek H_u és H_i szigorúan monoton növekvően rendezett C2L-ek! Írjuk meg a unionIntersection$(H_u, H_i$: E2*$)$ eljárást, ami a H_u listába H_i megfelelő elemeit átfűzve, a H_u listában az eredeti listák – mint halmazok – unióját állítja elő, míg a H_i listában a metszetük marad! Ne allokáljunk és ne is deallokáljunk listaelemeket, csak az listaelemek átfűzésével oldjuk meg a feladatot! $MT(n_u, n_i) \in \Theta(n_u + n_i)$, ahol a H_u C2L hossza n_u, a H_i C2L hossza pedig n_i. Minkét lista maradjon szigorúan monoton növekvően rendezett C2L!*

Mutasson q és r sorban a H_u és a H_i lista első elemére, pl.:

$H_u \to [/]—[\overset{q}{2}]—[4]—[6]—[/] \leftarrow H_u$
$H_i \to [/]—[\underset{r}{1}]—[4]—[8]—[9]—[/] \leftarrow H_i$

Az alábbi invariánssal dolgozunk, a következő jelölésekkel:

$$key(q, H) = \begin{cases} q \to key & \text{ha} \quad q \neq H \\ \infty & \text{ha} \quad q = H \end{cases}$$

Ha p és q ugyanannak a listának két (nem feltétlenül különböző) elemére mutat, akkor a (p, q) részlista a lista p és q közötti része, a határokat nem értve bele, a $[p, q)$ részlista pedig a lista p-vel kezdődő, de q előtti része (ami $p = q$ esetén üres).

- A H_u és H_i C2L-ek végig szigorúan növekvők maradnak, és együttvéve végig ugyanazokat a listaelemeket tartalmazzák, valamint q a H_u, r pedig a H_i valamelyik elemére mutat,
- a (H_u, q) részlista az eredeti listák rendezett uniójának prefixe, és az unió azon elemeit tartalmazza, amelyek kulcsai kisebbek, mint $min(key(q, H_u), key(r, H_i))$, de nem elemei a (H_i, r) részlistának,
- a (H_i, r) részlista az eredeti H_i lista elemei közül azokat tartalmazza, amelyek az eredeti H_u és H_i listák rendezett metszetének is elemei, és a kulcsai kisebbek, mint $min(key(q, H_u), key(r, H_i))$,
- a $[q, H_u)$ és $[r, H_i)$ részlisták még változatlanok.

A program futásának illusztrációja:

$H_u \to [/]—[\overset{q}{2}]—[4]—[6]—[/] \leftarrow H_u$
$H_i \to [/]—[\underset{r}{1}]—[4]—[5]—[8]—[9]—[/] \leftarrow H_i$

$H_u \to [/]—[1]—[\overset{q}{2}]—[4]—[6]—[/] \leftarrow H_u$
$H_i \to [/]—[\underset{r}{4}]—[5]—[8]—[9]—[/] \leftarrow H_i$

$H_u \to [/]—[1]—[2]—[\overset{q}{4}]—[6]—[/] \leftarrow H_u$
$H_i \to [/]—[\underset{r}{4}]—[5]—[8]—[9]—[/] \leftarrow H_i$

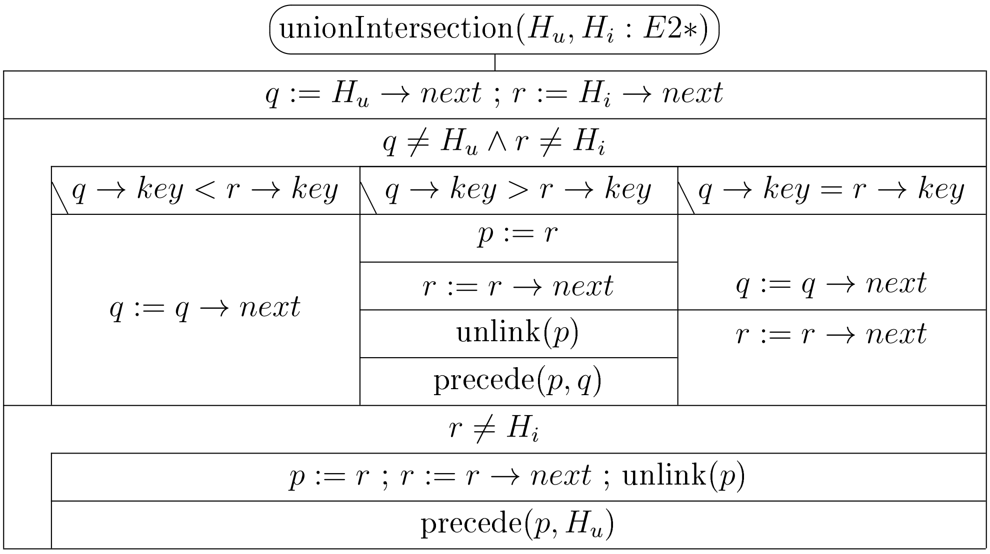
$H_u \to [/]—[1]—[2]—[4]—[\overset{q}{6}]—[/] \leftarrow H_u$
$H_i \to [/]—[4]—[\underset{r}{5}]—[8]—[9]—[/] \leftarrow H_i$

$H_u \to [/]—[1]—[2]—[4]—[5]—[\overset{q}{6}]—[/] \leftarrow H_u$
$H_i \to [/]—[4]—[\underset{r}{8}]—[9]—[/] \leftarrow H_i$

$H_u \to [/]—[1]—[2]—[4]—[5]—[6]—\overset{q}{[/]} \leftarrow H_u$
$H_i \to [/]—[4]—\underset{r}{[8]}—[9]—[/] \leftarrow H_i$

$H_u \to [/]—[1]—[2]—[4]—[5]—[6]—[8]—\overset{q}{[/]} \leftarrow H_u$
$H_i \to [/]—[4]—\underset{r}{[9]}—[/] \leftarrow H_i$

$H_u \to [/]—[1]—[2]—[4]—[5]—[6]—[8]—[9]—\overset{q}{[/]} \leftarrow H_u$
$H_i \to [/]—[4]—\underset{r}{[/]} \leftarrow H_i$



7.24. Feladat. *Írja meg a quickSort(H:E2∗){ QS(H, H) } eljárás struktogramjait, az alprogramjait is kifejetve, ahol az eljárás stabil rendezéssel, és a quicksort-nál szokásos aszimptotikus műveletigénnyel monoton növekvően rendezi a H C2L-t, a QS(p, s:E2∗) rekurzív segédeljárás pedig ezen belül a* (p, s) *részlistát rendezi quicksort-tal, kezdve a* (p, s) *részlista particionálásával, annak első eleme (q), mint tengely szerint!*

Az előbbi feladat megoldásához segítségül a particionálás illusztrációja következik az alábbi példán $p = H = s$ esetén, ahol q mutat a tengelyre, r pedig a (q, s) részlista futó pointere. A (q, s) részlistáról a tengelynél kisebb kulcsú listaelemeket sorban átfűzzük közvetlenül a tengely elé, így biztosítva a particionálás lieáris műveletigényét és a rendezés stabilitását.

$H \to \overset{p}{[/]}—\underset{q}{[5]}—\underset{r}{[2]}—[4]—[6]—[5]—[2]—\overset{s}{[/]} \leftarrow H$

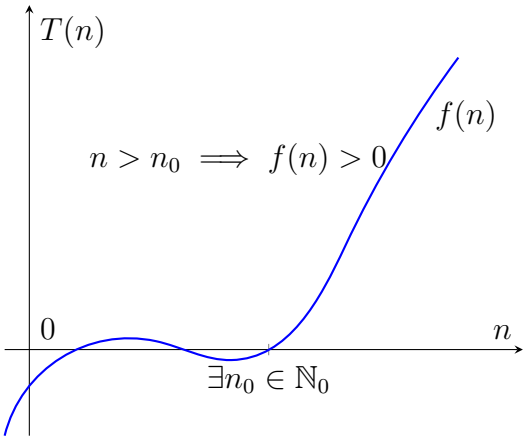
$$H \to \overset{p}{[/]}—[2]—\underset{q}{[5]}—\underset{r}{[4]}—[6]—[5]—[2]—\overset{s}{[/]} \leftarrow H$$

$$H \to \overset{p}{[/]}—[2]—[4]—\underset{q}{[5]}—\underset{r}{[6]}—[5]—[2]—\overset{s}{[/]} \leftarrow H$$

$$H \to \overset{p}{[/]}—[2]—[4]—\underset{q}{[5]}—[6]—\underset{r}{[5]}—[2]—\overset{s}{[/]} \leftarrow H$$

$$H \to \overset{p}{[/]}—[2]—[4]—\underset{q}{[5]}—[6]—[5]—\underset{r}{[2]}—\overset{s}{[/]} \leftarrow H$$

$$H \to \overset{p}{[/]}—[2]—[4]—[2]—\underset{q}{[5]}—[6]—[5]—\underset{r}{\overset{s}{[/]}} \leftarrow H$$



8. ábra. Aszimptotikusan pozitív (AP) függvény

8. Függvények aszimptotikus viselkedése

(a $\Theta, O, \Omega, \prec, \succ, o, \omega$ matematikája)

E fejezet célja, hogy tisztázza a programok hatékonyságának nagyságrendjeivel kapcsolatos alapvető fogalmakat, és az ezekhez kapcsolódó függvényosztályok legfontosabb tulajdonságait.

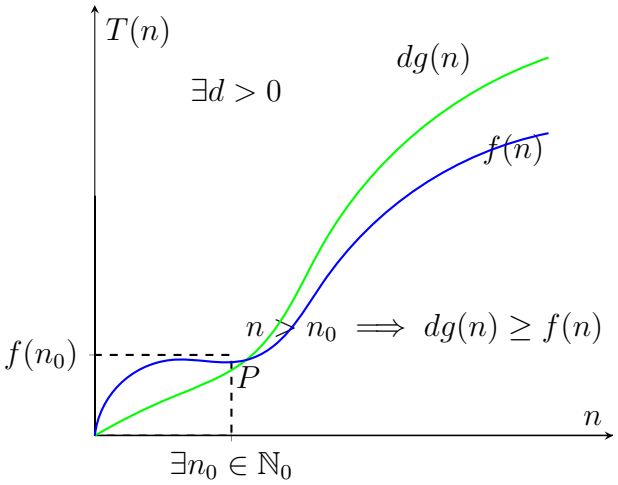
8.1. Definíció. *Valamely $P(n)$ tulajdonság elég nagy n -ekre pontosan akkor teljesül, ha $\exists N \in \mathbb{N}$, hogy $\forall n \in \mathbb{N}$-re $n \geq N$ esetén igaz $P(n)$.*

8.2. Definíció. *Az f AP (aszimptotikusan pozitív) függvény, ha elég nagy n-ekre $f(n) > 0$. (8. ábra)*

Egy tetszőleges helyes program futási ideje és tárigénye is nyilvánvalóan, tetszőleges megfelelő mértékegységben (másodperc, perc, Mbyte stb.) mérve pozitív számérték. Amikor (alsó és/vagy felső) becsléseket végzünk a futási időre vagy a tárigényre, legtöbbször az input adatszerkezetek méretének[16] függvényében végezzük a becsléseket. Így a becsléseket leíró függvények természetesen $\mathbb{N} \to \mathbb{R}$ típusúak. Megkövetelhetnénk, hogy $\mathbb{N} \to \mathbb{P}$ típusúak legyenek, de annak érdekében, hogy képleteink minél egyszerűbbek legyenek, általában megelégszünk azzal, hogy a becsléseket leíró függvények aszimptotikusan pozitívak (AP) legyenek.

8.3. Jelölések. *Az f, g, h (esetleg indexelt)* latin betűkről *ebben a fejezetben feltesszük, hogy $\mathbb{N} \to \mathbb{R}$ típusú, aszimptotikusan pozitív függvényeket jelölnek,*

[16] tömb mérete, láncolt lista hossza, fa csúcsainak száma stb.



9. ábra. f a nagy Ordó(g) függvényosztályhoz tartozik ($f \in O(g)$)

míg a φ, ψ görög betűkről *csak azt tesszük fel, hogy* $\mathbb{N} \to \mathbb{R}$ *típusú függvényeket jelölnek.*

8.4. Definíció. *Az* $O(g)$ *függvényhalmaz olyan* f *függvényekből áll, amiket elég nagy* n *helyettesítési értékekre, megfelelelő pozitív konstans szorzóval felülről becsül a* g *függvény:*

$O(g) = \{f : \exists d \in \mathbb{P}$, *hogy elég nagy* n *-ekre* $d * g(n) \geq f(n).\}$

$f \in O(g)$ *esetén azt mondjuk, hogy* g aszimptotikus felső korlátja f*-nek (9. ábra); szemléletesen:* f *legfeljebb* g*-vel arányos.*

8.5. Definíció. *Az* $\Omega(g)$ *függvényhalmaz olyan* f *függvényekből áll, amiket elég nagy* n *helyettesítési értékekre, megfelelelő pozitív konstans szorzóval alulról becsül a* g *függvény:*

$\Omega(g) = \{f : \exists c \in \mathbb{P}$, *hogy elég nagy* n *-ekre* $c * g(n) \leq f(n).\}$

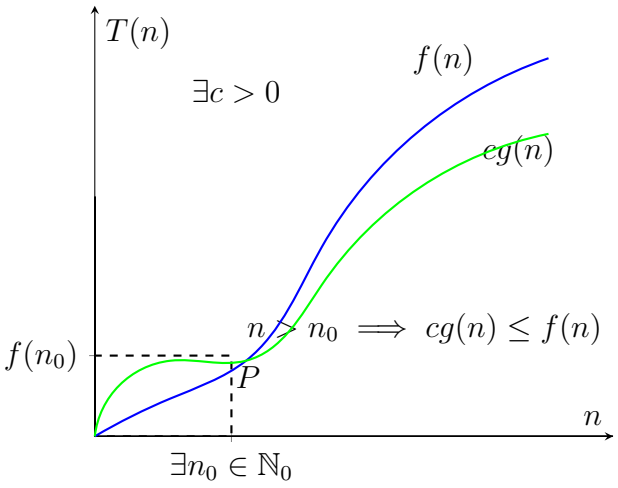
$f \in \Omega(g)$ *esetén azt mondjuk, hogy* g aszimptotikus alsó korlátja f*-nek (10. ábra); szemléletesen:* f *legalább* g*-vel arányos.*

8.6. Definíció. $\Theta(g) = O(g) \cap \Omega(g)$

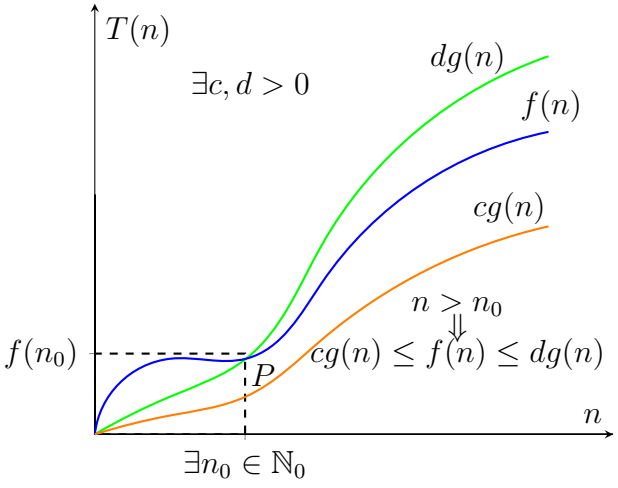
8.7. Következmény. *A* $\Theta(g)$ *függvényhalmaz olyan* f *függvényekből áll, amiket elég nagy* n *helyettesítési értékekre, megfelelelő pozitív konstans szorzókkal alulról és felülről is becsül a* g *függvény (11. ábra):*

$\Theta(g) = \{f : \exists c, d \in \mathbb{P}$, *hogy elég nagy* n *-ekre*

$$c * g(n) \leq f(n) \leq d * g(n).\}$$



10. ábra. f a nagy Omega(g) függvényosztályhoz tartozik ($f \in \Omega(g)$)



11. ábra. f a Theta(g) függvényosztályhoz tartozik ($f \in \Theta(g)$)

$f \in \Theta(g)$ *esetén tehát azt mondhatjuk, hogy* g aszimptotikus alsó és felső korlátja f*-nek (11. ábra); szemléletesen:* f *durván* g*-vel arányos.*

Arra, hogy egy függvény egy másikhoz képest nagy n értékekre elhanyagolható, bevezetjük az *aszimptotikusan kisebb* fogalmát.

8.8. Definíció.

$$\varphi \prec g \iff \lim_{n\to\infty} \frac{\varphi(n)}{g(n)} = 0$$

Ilyenkor azt mondjuk, hogy φ aszimptotikusan kisebb, *mint* g*. (Vegyük észre, hogy* φ *nem okvetlenül AP!) AP függvényekre* $f \prec g \iff f \in o(g)$*, azaz definíció szerint:*

$$o(g) = \{f : f \prec g\}$$

8.9. Definíció.

$$f \succ \psi \iff \psi \prec f$$

Ilyenkor azt mondjuk, hogy f aszimptotikusan nagyobb, *mint* ψ*. (Vegyük észre, hogy* ψ *nem okvetlenül AP!) AP függvényekre* $f \succ g \iff f \in \omega(g)$*, azaz definíció szerint:*

$$\omega(g) = \{f : f \succ g\}$$

8.10. Tulajdonság. *(A függvényosztályok kapcsolata)*

$$\Theta(g) = O(g) \cap \Omega(g)$$

$$o(g) \subsetneqq O(g) \setminus \Omega(g)$$

$$\omega(g) \subsetneqq \Omega(g) \setminus O(g)$$

Példa nevezetes AP függvények nagyságrendjére:
$1 \prec \log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec 2^n \prec n!$ (12. ábra)

8.11. Tulajdonság. *(Tranzitivitás)*

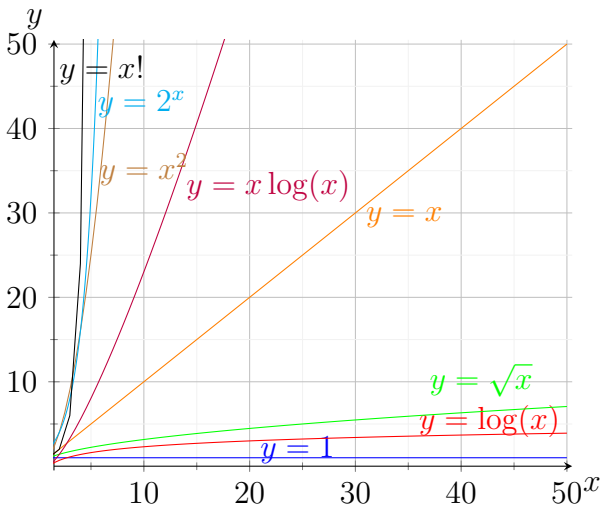
$$f \in O(g) \wedge g \in O(h) \Longrightarrow f \in O(h)$$

$$f \in \Omega(g) \wedge g \in \Omega(h) \Longrightarrow f \in \Omega(h)$$

$$f \in \Theta(g) \wedge g \in \Theta(h) \Longrightarrow f \in \Theta(h)$$

$$\varphi \prec g \wedge g \prec h \Longrightarrow \varphi \prec h$$

$$f \succ g \wedge g \succ \psi \Longrightarrow f \succ \psi$$



12. ábra. Nevezetes függvények növekedése

8.12. Tulajdonság. *(Szimmetria)*

$$f \in \Theta(g) \iff g \in \Theta(f)$$

8.13. Tulajdonság. *(Felcserélt szimmetria)*

$$f \in O(g) \iff g \in \Omega(f)$$

$$f \prec g \iff g \succ f$$

8.14. Tulajdonság. *(Aszimmetria)*

$$f \prec g \implies \neg(g \prec f)$$

$$f \succ g \implies \neg(g \succ f)$$

8.15. Tulajdonság. *(Reflexivitás)*

$$f \in O(f) \wedge f \in \Omega(f) \wedge f \in \Theta(f)$$

8.16. Következmény. *($\Longrightarrow$: 8.12, 8.11.3 ; $\Longleftarrow$: 8.15.3 alapján.)*

$$f \in \Theta(g) \iff \Theta(f) = \Theta(g)$$

8.17. Tulajdonság. *(A $\prec$ és a $\succ$ relációk irreflexívek.)*

$$\neg(f \prec f)$$

$$\neg(f \succ f)$$

8.18. Következmény. *Mivel az* $\cdot \in \Theta(\cdot)$ *bináris reláció reflexív, szimmetrikus és tranzitív, azért az aszimptotikusan pozitív függvények halmazának egy osztályozását adja, ahol* f *és* g *akkor és csak akkor tartozik egy ekvivalenciaosztályba, ha* $f \in \Theta(g)$*. Ilyenkor azt mondhatjuk, hogy az* f *függvény* aszimptotikusan ekvivalens *a* g *függvénnyel.*

Mint a továbbiakban látni fogjuk, megállapíthatók ilyen ekvivalenciaosztályok, és ezek a programok hatékonyságának mérése szempontjából alapvetőek lesznek. Belátható például, hogy tetszőleges k-adfokú, pozitív főegyütthatós polinom *aszimptotikusan ekvivalens* az n^k függvénnyel. Ilyen ekvivalenciaosztályok sorba is állíthatók az alábbi tulajdonság alapján.

8.19. Tulajdonság.

$$f_1, g_1 \in \Theta(h_1) \wedge f_2, g_2 \in \Theta(h_2), \wedge f_1 \prec f_2 \Longrightarrow g_1 \prec g_2$$

A most következő definíció tehát értelmes az előbbi tulajdonság miatt.

8.20. Definíció.

$$\Theta(f) \prec \Theta(g) \iff f \prec g$$

A függvények aszimptotikus viszonyának megállapításához hasznos az alábbi tétel.

8.21. Tétel.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Longrightarrow f \prec g$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{P} \Longrightarrow f \in \Theta(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Longrightarrow f \succ g$$

Bizonyítás. Az első és az utolsó állítás a $\prec$ és a $\succ$ relációk definíciójából közvetlenül adódik. A középsőhöz vegyük figyelembe, hogy $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, így elég nagy n értékekre $|\frac{f(n)}{g(n)} - c| < \frac{c}{2}$, azaz

$$\frac{c}{2} < \frac{f(n)}{g(n)} < \frac{3c}{2}$$

Mivel g AP, elég nagy n-ekre $g(n) > 0$, ezért átszorozhatunk vele. Innét

$$\frac{c}{2} * g(n) < f(n) < \frac{3c}{2} * g(n)$$

és végül $f \in \Theta(g)$ adódik. □

8.22. Következmény.

$$k \in \mathbb{N} \wedge a_0, a_1, \ldots, a_k \in \mathbb{R} \wedge a_k > 0 \Longrightarrow a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \in \Theta(n^k)$$

Bizonyítás.

$$\lim_{n\to\infty} \frac{a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0}{n^k} =$$

$$\lim_{n\to\infty} \left(\frac{a_k n^k}{n^k} + \frac{a_{k-1} n^{k-1}}{n^k} + \cdots + \frac{a_1 n}{n^k} + \frac{a_0}{n^k} \right) =$$

$$\lim_{n\to\infty} \left(a_k + \frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k} \right) =$$

$$\lim_{n\to\infty} a_k + \lim_{n\to\infty} \frac{a_{k-1}}{n} + \cdots + \lim_{n\to\infty} \frac{a_1}{n^{k-1}} + \lim_{n\to\infty} \frac{a_0}{n^k} =$$

$$a_k + 0 + \cdots + 0 + 0 = a_k \in \mathbb{P} \Longrightarrow$$

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \in \Theta(n^k)$$

□

8.23. Lemma. Az alábbi, ún. **L'Hospital szabály**t gyakran alkalmazhatjuk, amikor a 8.21. tétel szerinti $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ határértéket szeretnénk kiszámítani.

Ha elég nagy helyettesítési értékekre az f és g függvények valós kiterjesztése differenciálható, valamint

$$\lim_{n\to\infty} f(n) = \infty \wedge \lim_{n\to\infty} g(n) = \infty \wedge \exists \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \Longrightarrow$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

8.24. Következmény. *(8.21. és 8.23. alapján)*

$$c, d \in \mathbb{R} \wedge c < d \Longrightarrow n^c \prec n^d$$

$$c, d \in \mathbb{P}_0 \wedge c < d \Longrightarrow c^n \prec d^n$$

$$c, d \in \mathbb{R} \wedge d > 1 \Longrightarrow n^c \prec d^n$$

$$d \in \mathbb{P}_0 \Longrightarrow d^n \prec n! \prec n^n$$

$$c, d \in \mathbb{P} \wedge c, d > 1 \Longrightarrow \log_c n \in \Theta(\log_d n)$$

$$\varepsilon \in \mathbb{P} \Longrightarrow \log n \prec n^\varepsilon$$

$$c \in \mathbb{R} \wedge \varepsilon \in \mathbb{P} \Longrightarrow n^c \log n \prec n^{c+\varepsilon}$$

Bizonyítás. Az $\varepsilon \in \mathbb{P} \Longrightarrow \log n \prec n^{\varepsilon}$ állítás bizonyításához szükségünk lesz a L'Hospital szabályra (8.23. Lemma).

$$\lim_{n\to\infty} \frac{\log n}{n^{\varepsilon}} = \log e \lim_{n\to\infty} \frac{\ln n}{n^{\varepsilon}} = \log e \lim_{n\to\infty} \frac{\ln' n}{(n^{\varepsilon})'} = \frac{\log e}{\varepsilon} \lim_{n\to\infty} \frac{\frac{1}{n}}{n^{\varepsilon-1}} =$$

$$= \frac{\log e}{\varepsilon} \lim_{n\to\infty} \frac{1}{n^{\varepsilon}} = \frac{\log e}{\varepsilon} 0 = 0$$

□

8.25. Következmény. *(Nevezetes műveletigény nagyságrendek viszonya)*

$$\Theta(1) \prec \Theta(\log n) \prec \Theta(\sqrt{n}) \prec \Theta(n) \prec \Theta(n * \log n) \prec \Theta(n^2) \prec \Theta(2^n) \prec \Theta(n!)$$

8.26. Tulajdonságok. .
(A $O(\cdot), \Omega(\cdot), \Theta(\cdot), o(\cdot), \omega(\cdot)$ függvényosztályok zártsági tulajdonságai)

$$f \in O(g) \wedge c \in \mathbb{P} \Longrightarrow c * f \in O(g)$$

$$f \in O(h_1) \wedge g \in O(h_2) \Longrightarrow f + g \in O(h_1 + h_2)$$

$$f \in O(h_1) \wedge g \in O(h_2) \Longrightarrow f * g \in O(h_1 * h_2)$$

$$f \in O(g) \wedge \varphi \prec f \Longrightarrow f + \varphi \in O(g)$$

(Hasonlóan az $\Omega(\cdot), \Theta(\cdot), o(\cdot), \omega(\cdot)$ függvényosztályokra.)

Most arra térünk ki, hogy az $O(g), \Omega(g), \Theta(g)$ függvényosztályok definíciója hogyan viszonyul a $\Theta(g)$ függvényosztályról a korábbi fejezetekben kialakított képhez. Kiderül, hogy a korábbi jellemzés pontosan megfelel a fenti definícióknak.

8.27. Tétel. $f \in O(g) \iff \exists d \in \mathbb{P}$ *és* $\exists \psi \prec g$, *hogy elég nagy n-ekre*

$$d * g(n) + \psi(n) \geq f(n)$$

8.28. Tétel. $f \in \Omega(g) \iff \exists c \in \mathbb{P}$ *és* $\exists \varphi \prec g$, *hogy elég nagy n-ekre*

$$c * g(n) + \varphi(n) \leq f(n)$$

8.29. Tétel. $f \in \Theta(g) \iff \exists c, d \in \mathbb{P}$ *és* $\exists \varphi, \psi \prec g$, *hogy elég nagy n-ekre*

$$c * g(n) + \varphi(n) \leq f(n) \leq d * g(n) + \psi(n)$$

Ld. még ezzel kapcsolatban az alábbi címen az 1.3. alfejezetet! [2]
<http://people.inf.elte.hu/fekete/algoritmusok_jegyzet
. /01_fejezet_Muveletigeny.pdf>

8.1. $\mathbb{N} \times \mathbb{N}$ értelmezési tartományú függvények

Vegyük észre, hogy a fenti függvényosztályokat eddig csak olyan függvényekre értelmeztük, amelyek értelmezési tartománya a természetes számok halmaza. Ha értelmezési tartománynak az $\mathbb{N} \times \mathbb{N}$-et tekintjük, az alapvető fogalmak a következők.

8.30. Definíció. *$g : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ függvény AP,*
ha elég nagy n és elég nagy m értékekre $g(n, m) > 0$.

8.31. Megjegyzés. *Az alfejezet hátralevő részében az egyszerűség kedvéért feltesszük, hogy $f, g, h : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ AP függvényeket jelölnek.*

8.32. Definíció. *$O(g) = \{f \mid \exists d \in \mathbb{P}$, hogy $f(n, m) \leq d * g(n, m)$, tetszőleges elég nagy n és elég nagy m értékekre$\}$.*

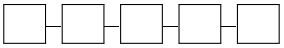
8.33. Definíció. *$\Omega(g) = \{f \mid \exists c \in \mathbb{P}$, hogy $f(n, m) \geq c * g(n, m)$, tetszőleges elég nagy n és elég nagy m értékekre$\}$.*

8.34. Definíció. *$\Theta(g) = \{f \mid \exists c, d \in \mathbb{P}$, hogy $c * g(n, m) \leq f(n, m) \leq d * g(n, m)$, tetszőleges elég nagy n és elég nagy m értékekre$\}$.*

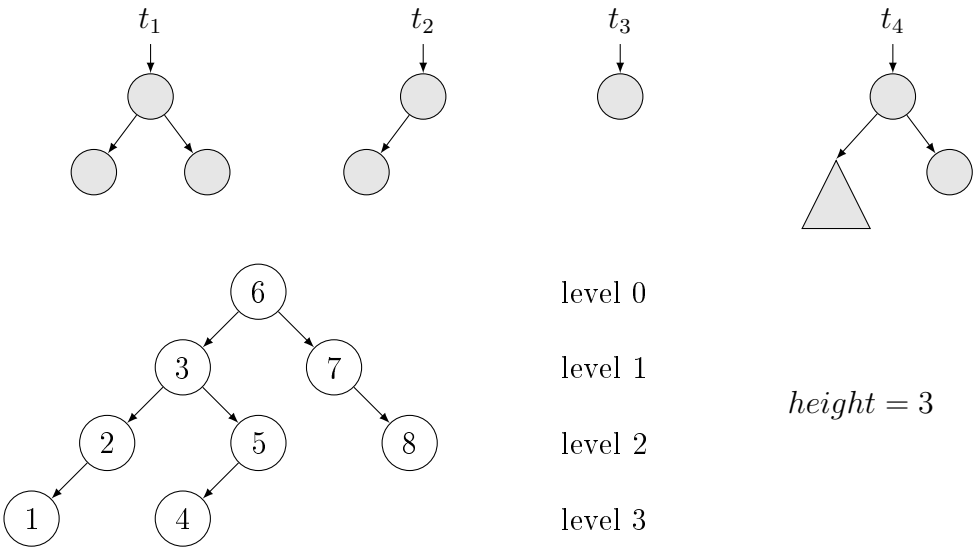
8.35. Megjegyzés. *A korábban a természetes számokon értelmezett függvényekre vonatkozó tételek az itt tárgyaltakra természetes módon általánosíthatók.*

9. Fák, bináris fák (Trees, binary trees)

A (bináris) fákat nagy méretű adathalmazok és multihalmazok (zsákok) ábrázolására, de egyéb adatreprezentációs célokra is gyakran használjuk.

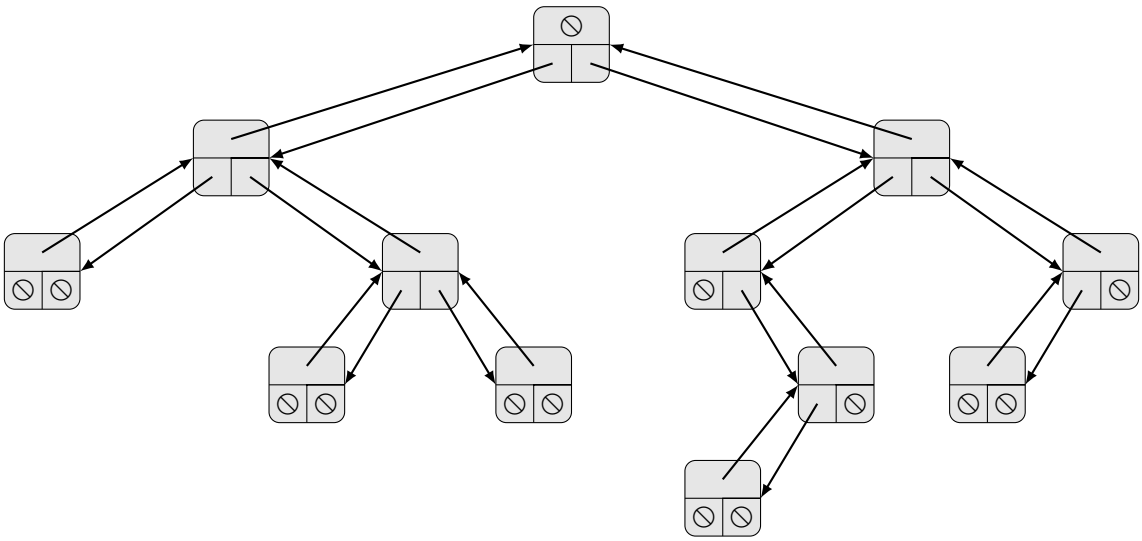
Az egydimenziós tömbök és a láncolt listák esetében minden adatelemnek legfeljebb egy rákövetkezője van, azaz az adatelemek lineárisan kapcsolódnak egymáshoz a következő séma szerint:  .

A **bináris fák** esetében minden adatelemnek vagy szokásos nevén *csúcsnak* (angolul *node*) legfeljebb kettő rákövetkezője van: egy *bal* (*left*) és/vagy egy *jobb* (*right*) rákövetkezője. Ezeket a csúcs *gyerekeinek* (*children*) nevezzük. A csúcs a gyerekei *szülője* (*parent*), ezek pedig egymás *testvérei* (*siblings*). Ha egy csúcsnak nincs gyereke, *levélnek* (*leaf*) hívjuk, ha pedig nincs szülője, *gyökér* (*root*) csúcsnak nevezzük. *Belső csúcs* (*internal node*) alatt nem-levél csúcsot értünk. A fában egy csúcs *leszármazottai* (*descendants*) a gyerekei és a gyerekei leszármazottai. Hasonlóan, egy csúcs *ősei* (*ancestors*) a szülője és a szülője ősei. A fákat felülről lefelé szokás lerajzolni: fent van a gyökér, lent a levelek (ld. 13. ábra).



13. ábra. Egyszerű bináris fák. Egy konkrét elemét a fának körrel jelöljük. Amennyiben nem fontos, hogy milyen szerkezete van egy adott részfának, azt háromszöggel jelöljük.

Az $\oslash$ *üres fának* (*empty tree*) nincs csúcsa.



14. ábra. Bináris fa szülő mutatóval.

Egy tetszőleges nemüres t fát a gyökércsúcsa ($*t$) határoz meg, mivel ennek a fa többi csúcsa a leszármazottja.

A $*t$ bal/jobb gyerekéhez tartozó fát a t *bal/jobb részfájának (left/right subtree)* nevezzük. Jelölése $t \to left$ illetve $t \to right$ (szokás a $left(t)$ és $right(t)$ jelölés is). Ha $*t$-nek nincs bal/jobb gyereke, akkor $t \to left = \oslash$ illetve $t \to right = \oslash$. Ha a gyerek létezik, jelölése $*t \to left$ illetve $*t \to right$. (Itt a „$\to$” erősebben köt, mint a „$*$”. Pl. $*t \to left = *(t \to left)$)

A t bináris fának ($t = \oslash$ esetén is) *részfája* (*subtree*) önmaga. Ha $t \neq \oslash$, részfái még a $t \to left$ és a $t \to right$ részfái is.

A t *valódi részfája* az f, ha a t részfája az f és $t \neq f \neq \oslash$
(f is *proper subtree* of $t \iff f$ is subtree of $t \ \wedge \ \oslash \neq f \neq t$).

A $*t$-ben tárolt kulcs jelölése $t \to key$ (illetve $key(t)$).

Ha $*g$ egy fa egy csúcsa, akkor a szülője a $*g \to parent$, és a szülőjéhez, mint gyökércsúcshoz tartozó fa a $g \to parent$ (illetve $parent(g)$). Ha $*g$ a teljes fa gyökere, azaz nincs szülője, akkor $g \to parent = \oslash$.

Megjegyezzük, hogy a gyakorlatban – ugyanúgy, mint a tömbök és a láncolt listák elemeinél – a kulcs általában csak a csúcsban tárolt adat egy kis része, vagy abból egy függvény segítségével számítható ki. Mi az egyszerűség kedvéért úgy tekintjük, mintha a kulcs az egész adat lenne, mert az adatszerkezetek műveleteinek lényegét így is be tudjuk mutatni.

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 ($i \in [0\,..\,r)$), 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.

Beszélhetünk a fa *szintjeiről* (*levels*). A gyökér van a *nulladik* szinten. Az i-edik szintű csúcsok gyerekeit az $(i+1)$-edik szinten találjuk. A fa *magassága* (*height*) egyenlő a legmélyebben fekvő levelei szintszámával. Az üres fa magassága $h(\oslash) = -1$. Néha szoktak a fa mélységéről is beszélni, ami ugyanaz, mint a magassága.

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, és valamely $r \in \mathbb{N}$-re tetszőleges csúcs kimenő élei a $[0\,..\,r)$ egy részhalmaza elemeivel egyértelműen[17] vannak címkézve. (Ezzel szemben a szabad fák összefüggő, körmentes irányítatlan gráfok.)

9.1. Listává torzult, szigorúan bináris, tökéletes és majdnem teljes bináris fák

Azokat a fákat, amelyekben minden belső (azaz nem-levél) csúcsnak egy gyereke van, *listává torzult fáknak* nevezzük. Például a 14. ábrán látható bináris fa jobb oldali részfájának bal részfája listává torzult. A 13. ábrán t_2 és t_3 listává torzult.

Azokat a bináris fákat, amelyekben minden belső (azaz nem-levél) csúcsnak két gyereke van, *szigorúan bináris fáknak* nevezzük (angolul *strictly binary tree* vagy *full binary tree*). Ha ez utóbbiaknak minden levele azonos szinten van, *tökéletes bináris fákról* beszélünk (angolul *perfect binary tree*). (Ilyenkor az összes levél szükségszerűen a fa legmélyebb szintjén található, a felsőbb szinteken levő csúcsok pedig belső csúcsok, így két-két gyerekük van.) Tetszőleges h magasságú tökéletes bináris fa csúcsainak száma tehát $1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$.

Például a 13. ábrán t_3 0 magasságú, t_1 1 magasságú tökéletes bináris fa, a 16. ábrán pedig kettő magasságú, tökéletes bináris fák láthatók. Megjegyezzük, hogy a fenti definíció szerint az üres fa is tökéletes.

[17] Az egyértelműség itt azt jelenti, hogy egyetlen csúcsnak sincs két azonos címkéjű kimenő éle.

Ha egy tökéletes bináris fa levélszintjéről nulla, egy vagy több levelet elveszünk, de nem az összeset, az eredményt *majdnem teljes bináris fának* nevezzük (angolul *nearly complete binary tree*).

Az üres fát is majdnem teljesnek tekintjük, így minden tökéletes bináris fa egyben majdnem teljes is (fordítva viszont nem igaz).

Tetszőleges h mélységű, nemüres, majdnem teljes bináris fa csúcsainak száma a fenti definíció szerint $n \in [2^h \,..\, 2^{h+1})$, és így $h = \lfloor \log n \rfloor$. Az alsó szinten levő leveleket elvéve pedig egy $h-1$ mélységű tökéletes bináris fát kapunk.

Például a 13. ábrán t_3 0 magasságú, t_2 és t_1 1 magasságú, majdnem teljes bináris fák; a 14. ábrán látható bináris fát pedig 3 magasságú, majdnem teljes bináris fává alakíthatnánk, ha a legalsó (4.) szintjén lévő csúcsát törölnénk.

9.2. Bináris fák mérete és magassága

Bináris fa *mérete* (*size*) alatt a csúcsainak számát értjük. A t bináris fa méretét $|t|$, vagy $n(t)$, vagy ha a szövegösszefüggésből egyértelmű, melyik fáról van szó, egyszerűen csak n jelöli.

Emlékeztetünk, hogy tetszőleges bináris fában a gyökér van a *nulladik* szinten. Az i-edik szintű csúcsok gyerekeit az $(i+1)$-edik szinten találjuk. A fa *magassága* (*height*) egyenlő a legmélyebben fekvő levelei szintszámával.

A t bináris fa magasságát $h(t)$, vagy ha a szövegösszefüggésből egyértelmű, melyik fáról van szó, egyszerűen csak h jelöli.

Az üres fa magassága $h(\oslash) = -1$. Így tetszőleges nemüres t bináris fára:

$$h(t) = 1 + max(h(t \to left), h(t \to right))$$

9.1. Tétel. *Tetszőleges $n > 0$ méretű és $h \geq 0$ magasságú (azaz nemüres* [angolul *nonempty*]*) bináris fára*

$$\lfloor \log n \rfloor \leq h \leq n - 1$$

Bizonyítás. Először a $\lfloor \log n \rfloor \leq h$ egyenlőtlenséget bizonyítjuk be. A h mélységű bináris fák között nyilván a tökéletes bináris fának van a legtöbb csúcsa. Mivel erre $n = 2^{h+1} - 1$ (ld. 9.1.), tetszőleges bináris fára $n < 2^{h+1}$. Innét $n > 0$ miatt $\log n < \log 2^{h+1} = h + 1$, amiből $\lfloor \log n \rfloor \leq h$ közvetlenül adódik. Mint 9.1-ben láttuk, tetszőleges majdnem teljes bináris fára teljesül az egyenlőség.

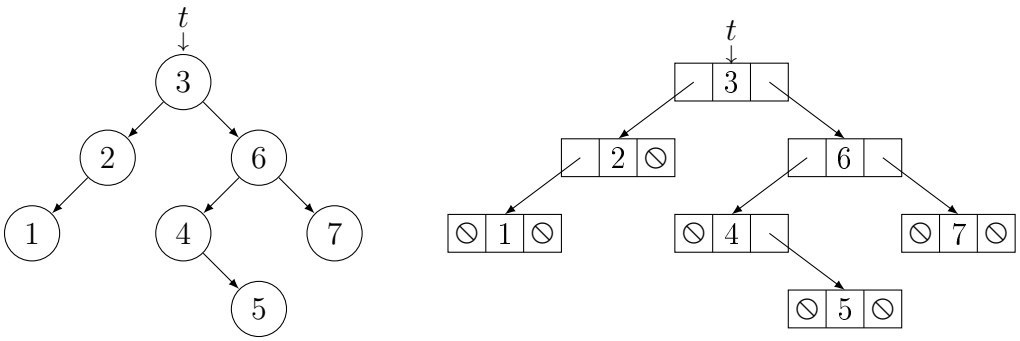
A $h \leq n - 1$ egyenlőtlenséghez gondoljuk meg, hogy tetszőleges h magasságú fa szintjeit 0-tól h-ig sorszámoztuk, ami összesen $h + 1$ szintet jelent. Mivel a fa minden szintjén van legalább egy csúcs, ezért tetszőleges fára

$n \geq h + 1$, azaz $h \leq n - 1$, ahol $h = n - 1$ pontosan akkor teljesül, ha a fa listává torzult. □

9.2. Feladat. *Mutassunk példát olyan t bináris fára, amire $\lfloor \log n \rfloor = h$, bár t-re a majdnem teljesség kritériuma nem teljesül. Melyik az a legkisebb h magasság, amire adható ilyen bináris fa?*

9.3. Bináris fák láncolt (linked) ábrázolásai

A legtermészetesebb és az egyik leggyakrabban használt a bináris fák **láncolt ábrázolása**. Ld. például a 15. ábrát!



15. ábra. Ugyanaz a bináris fa grafikus és láncolt ábrázolással.

Az üres fa reprezentációja a ⊘ pointer, jelölése tehát ugyanaz, mint az absztrakt fáknál. A bináris fa csúcsait pl. az alábbi **Node** osztály objektumaiként ábrázolhatjuk. Tetszőleges, láncoltan ábrázolt bináris fát egy **Node*** típusú pointer azonosít. Ez ⊘, ha a fa üres. A gyökércsúcsára hivatkozik, ha a fa nemüres.

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

Néha hasznos, ha a csúcsokban van egy *parent* szülő pointer is, mert a fában így felfelé is tudunk haladni. A 14. ábrán megfigyelhetünk egy szülő mutatóval rendelkező bináris fát.

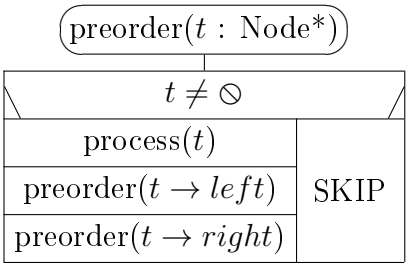
Node3
+ $key : \mathcal{T}$ // $\mathcal{T}$ valamilyen ismert típus + $left, right, parent$: Node3*
+ Node3(p:Node3*) { $left := right := \oslash$; $parent := p$ } + Node3($x : \mathcal{T}$, p:Node3*) { $left := right := \oslash$; $parent := p$; $key := x$ }

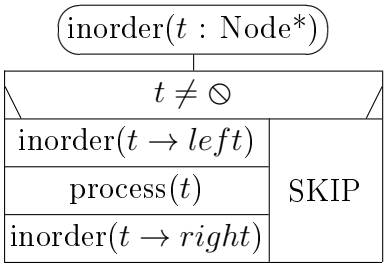
9.4. (Bináris) fák bejárásai (Binary) tree traversals

A (bináris) fákkal dolgozó programok gyakran kapcsolódnak a négy klasszikus bejárás (traversal) némelyikéhez, amelyek adott sorrend szerint bejárják a fa csúcsait, és minden csúcsra ugyanazt a műveletet hívják meg, amivel kapcsolatban megköveteljük, hogy futási ideje $\Theta(1)$ legyen (ami ettől még persze összetett művelet is lehet). A $*f$ csúcs feldolgozása lehet például $f \to key$ kiíratása.

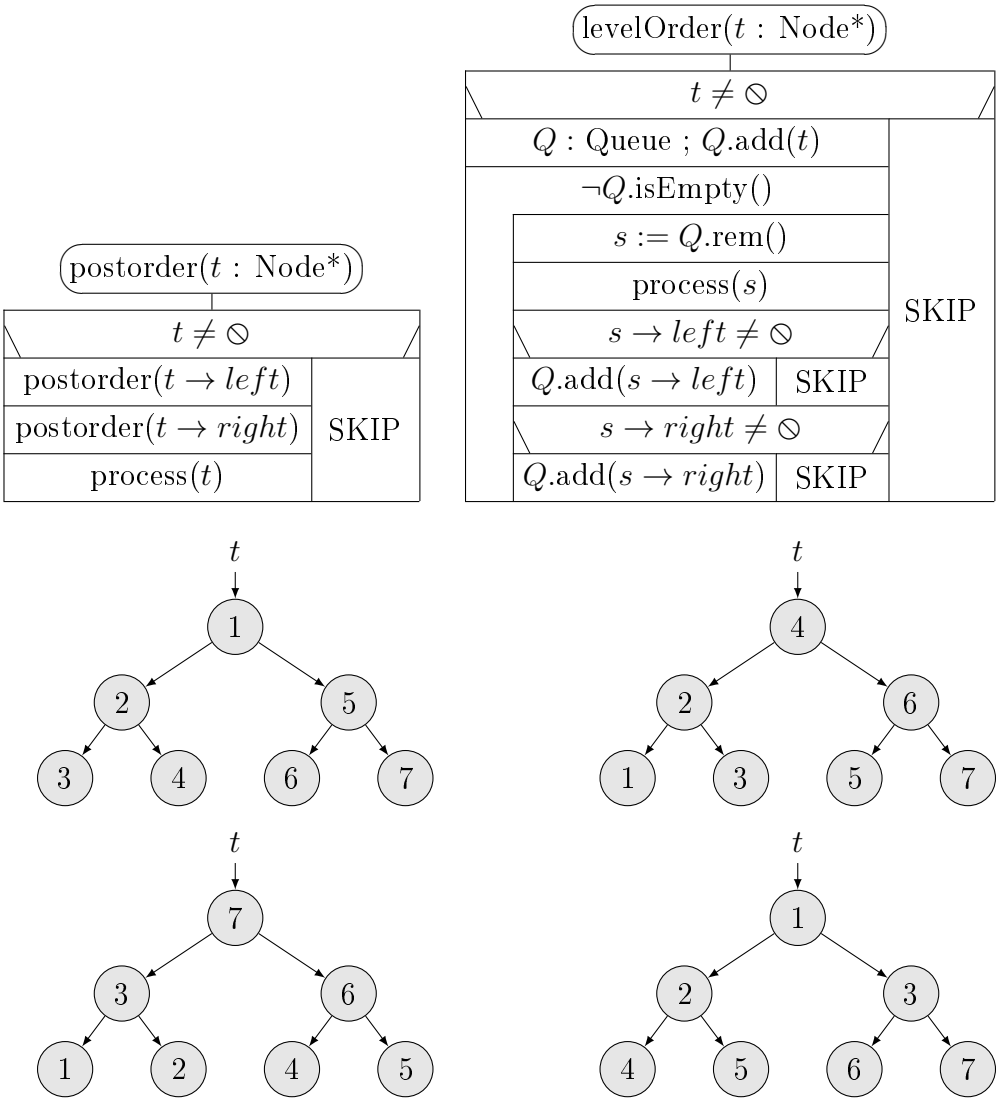
Üres fára mindegyik bejárás az üres program. Nemüres r-áris fákra
- a *preorder* bejárás először a fa gyökerét dolgozza fel, majd sorban bejárja a $0\,..\,r-1$-edik részfákat;
- a *postorder* bejárás előbb sorban bejárja a $0\,..\,r-1$-edik részfákat, és a fa gyökerét csak a részfák után dolgozza fel;
- az *inorder* bejárás először bejárja a *nulladik* részfát, ezután a fa gyökerét dolgozza fel, majd sorban bejárja az $1\,..\,r-1$-edik részfákat;
- a *szintenkénti* bejárás (*Breadth First or Level Order traversal*) a csúcsokat a gyökértől kezdve szintenként, minden szintet balról jobbra bejárva dolgozza fel.

Az első három bejárás tehát nagyon hasonlít egymásra. Nevük megsúgja, hogy a gyökércsúcsot a részfákhoz képest mikor dolgozzák fel. Bináris fákra[18] a struktogramok a következők.





[18] A struktogramokban a „$*t$” csúcs feldolgozását „process(t)” jelöli.



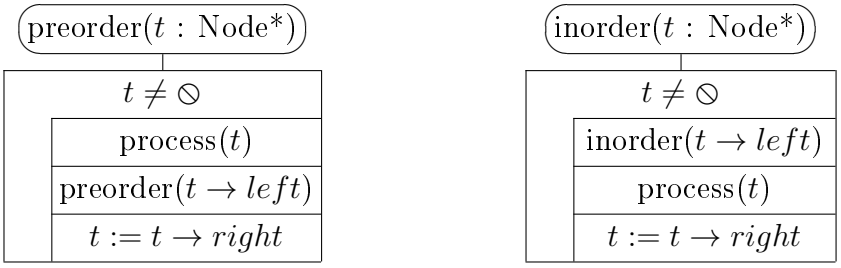
16. ábra. Bal felső sarokban preorder, jobb felsőben inorder, bal alsóban postorder, jobb alsóban szintenkénti bejárása látható a t bináris fának.

Állítás: $T_{\text{preorder}}(n), T_{\text{inorder}}(n), T_{\text{postorder}}(n), T_{\text{levelOrder}}(n) \in \Theta(n)$, ahol n a fa *mérete*, azaz csúcsainak száma.

Igazolás: Az első három bejárás pontosan annyiszor hívódik meg, amennyi részfája van az eredeti bináris fának (az üres részfákat is beleszámolva), és egy-egy hívás végrehajtása $\Theta(1)$ futási időt igényel. Másrészt n szerinti teljes indukcióval könnyen belátható, hogy tetszőleges n csúcsú bináris fának $2n+1$ részfája van. A szintenkénti bejárás pedig mindegyik csúcsot a ciklus egy-egy végrehajtásával dolgozza fel.

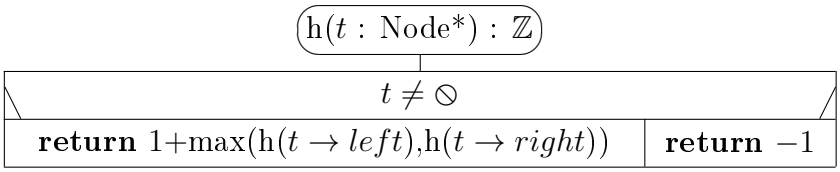
A szintenkénti bejárás helyessége azon alapszik, hogy egy fa csúcsainak szintenkénti felsorolásában a korábbi csúcs gyerekei megelőzik a későbbi csúcs gyerekeit. Ez az állítás nyilvánvaló, akár egy szinten, akár különböző szinten van a két csúcs a fában.

A preorder és az inorder bejárások hatékonysága konstans szorzóval javítható, ha a végrekurziókat ciklussá alakítjuk. (Mivel a t paramétert érték szerint adjuk át, az aktuális paraméter nem változik meg. [Általában nem célszerű cím szerint átvett formális paramétereket ciklusváltozóként vagy segédváltozóként használni.])



9.4.1. Fabejárások alkalmazása: bináris fa magassága

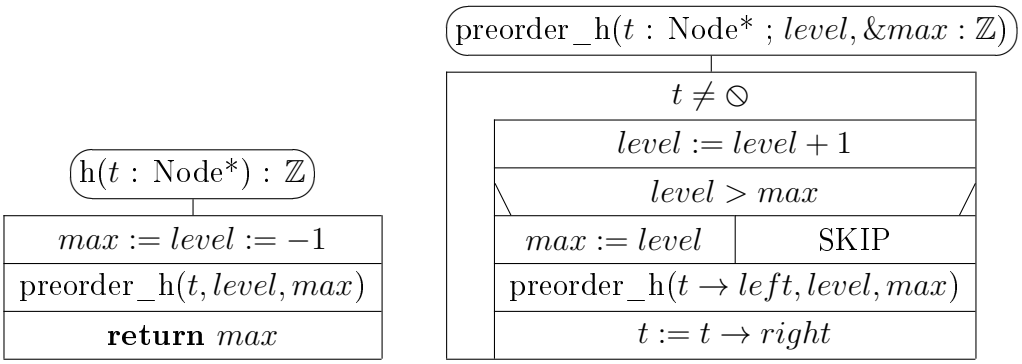
Természetesen az a legegyszerűbb, ha a definíciót kódoljuk le:



Mint látható, ez lényegében véve egy függvényként kódolt postorder bejárás.[19] Talán elsőre meglepő, de ezt a feladatot preorder bejárással is könnyen megoldhatjuk. Mint a legtöbb rekurzív programnál, itt is lesz egy nemrekurzív keret, ami előkészíti a legkülső rekurzív hívást, s a végén is biztosítja a megfelelő interfészt. Lesz két extra paraméterünk. Az egyik (*level*) azt tárolja, milyen mélyen (azaz hányadik szinten) járunk a fában. A másik (*max*)

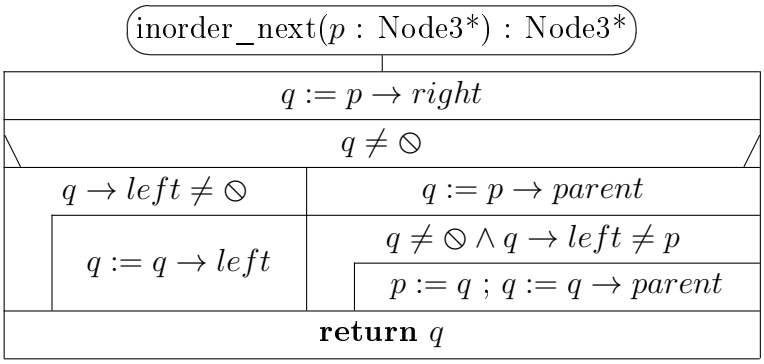
[19] Azért csak *lényegében véve*, mert a részfák bejárásának sorrendje a max() függvény paramétereinek kiértékelési sorrendjétől függ, és az eljárások, függvények aktuális paramétereinek kiértékelési sorrendjét általában nem ismerjük.

pedig azt, hogy mi a legmélyebb szint, ahol eddig jártunk. Ezután már csak a bejárás és a maximumkeresés összefésülésére van szükség.[20]



9.4.2. Példa a szülő (parent) pointerek használatára

Előfordul például olyan alkalmazás, ahol szükségünk van a bináris fában egy p : Node3* pointer által mutatott csúcs ($p \neq \oslash$) inorder bejárás szerinti rákövetkezőjének címére. Ha nincs ilyen rákövetkező, a függvény $\oslash$-t ad vissza. Nyilván $MT(h(t)) \in O(h(t))$, ahol t a $*p$ csúcsot tartalmazó bináris fa.



9.3. Feladat. *Írjuk meg az inorder_megel(p) függvényt, ami a $*p$ csúcs inorder bejárás szerinti megelőzőjét adja vissza; ha nincs ilyen, $\oslash$-t! Tartsuk meg az $O(h(t))$ maximális műveletigényt!*

[20] Így a végrehajtáshoz csak egy függvényhívásra, $n + 1$ eljáráshívásra és n ciklus-iterációra lesz szükség, míg az előbbi esetben $2n+1$ függvényhívásra, ha a max() függvény hívásait nem számítjuk (ami könnyen kibontható egy szekvencia + elágazássá). Mivel egy ciklus-iteráció a gyakorlatban gyorsabb, mint egy rekurzív hívás, a „preoder" magasság számítás a gyorsabb.

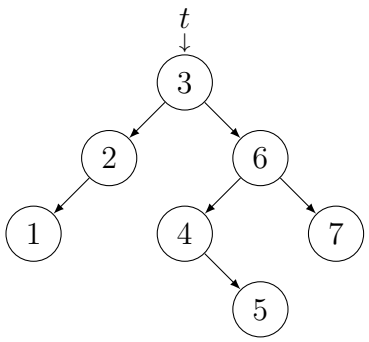
9.5. Bináris fák zárójelezett, szöveges formája

Tetszőleges nemüres bináris fa zárójeles, azaz szöveges alakja (textual form):

(balRészFa Gyökér jobbRészFa)

Az üres (rész)fát egy üres zárójelpár reprezentálja. pl. (). A könnyebb olvashatóság kedvéért a levélcsúcsok üres részfáihoz tartozó üres zárójelpárokat elhagyjuk. Pl. az "5" kulcsú, egy csúcsú bináris fa szöveges reprezentációja (() (5) ()) helyett egyszerűen (5). A bináris fák szöveges ábrázolásánál könnyebb olvashatóság kedvéért többféle zárójelpárt is használhatunk. A zárójeles ábrázolás lexikai elemei:
(1) nyitó zárójel, (2) csukó zárójel és (3) a csúcsok címkéi.

Például a 17. ábrán látható egy bináris fa grafikus reprezentációja, egyszerű zárójeles alakja (csak kerek zárójeleket használva), és az elegáns zárójeles alakja is, ami hasonlít az előbbi formához, de többféle zárójelpárt alkalmazunk. Pl. ha egy részfa gyökércsúcsa a nulladik szinten van, akkor a hozzá tartozó részfát { }, ha az elsőn, akkor [ ], ha a másodikon, akkor ( ), ha a harmadikon, akkor $\langle\ \rangle$, ha a negyediken, akkor újra { } zárójelek közé tehetjük és így tovább; ha az aktuális l szintre $l \bmod 4 = 0$, akkor { }, ha $l \bmod 4 = 1$, akkor [ ], ha $l \bmod 4 = 2$, akkor akkor ( ), ha $l \bmod 4 = 3$, akkor $\langle\ \rangle$ zárójeleket használhatunk.



A balra látható t bináris fa

- egyszerű zárójeles alakja:
  (((1) 2 ()) 3 ((() 4 (5)) 6 (7)))
- elegáns zárójeles alakja:
  { [(1) 2 ()] 3 [($\langle\rangle$ 4 $\langle 5 \rangle$) 6 (7)] }

17. ábra. Ugyanaz a bináris fa grafikus és szöveges ábrázolással. Vegyük észre, hogy ez utóbbiból a zárójeleket elhagyva, a fa inorder bejárását kapjuk!

9.6. Bináris keresőfák (binary search trees)

Egy bináris fát *keresőfának* nevezünk, ha minden belső csúcsára
és annak y kulcsára igazak az alábbi követelmények:

- A csúcs bal részfájában tetszőleges csúcs x kulcsára $x < y$.
- A csúcs jobb részfájában tetszőleges csúcs z kulcsára $z > y$.
(Például a 17. ábrán egy bináris keresőfa látható.)

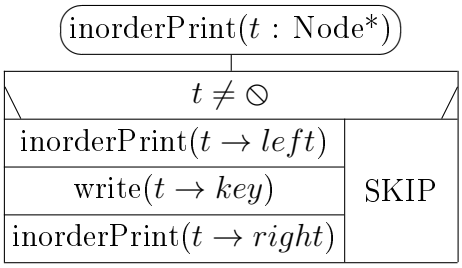
Egy bináris fát *rendezőfának* (*binary sort tree*) nevezünk, ha
minden belső csúcsára és annak y kulcsára igazak az alábbi követelmények:
- A csúcs bal részfájában tetszőleges csúcs x kulcsára $x \leq y$.
- A csúcs jobb részfájában tetszőleges csúcs z kulcsára $z \geq y$.

A keresőfában tehát minden kulcs egyedi, míg a rendezőfában lehetnek duplikált és többszörös kulcsok is. A továbbiakban a műveleteket *bináris keresőfákra* írjuk meg, de ezek könnyen átírhatók a bináris rendezőfák esetére.

A bináris keresőfák tekinthetők véges halmazok, vagy szigorúan monoton növekvő sorozatok reprezentációinak.

Jelölje $H(t)$ a t bináris keresőfa által reprezentált halmazt! Ekkor definíció szerint $H(\oslash) = \{\}$, illetve $H(t) = H(t \to left) \cup \{t \to key\} \cup H(t \to right)$, amennyiben $t \neq \oslash$.

A t bináris keresőfa által reprezentált sorozatot megkaphatjuk, ha Inorder bejárással kiíratjuk a fa csúcsainak tartalmát:



9.4. Feladat. *Bizonyítsuk be, hogy a fenti program a* t bináris keresőfa *kulcsait szigorúan monoton növekvő sorrendben írja ki! (Ötlet: teljes indukció* t *mérete vagy magassága szerint.)*

Bizonyítsuk be azt is, hogy ha a fenti program a t bináris fa *kulcsait szigorúan monoton növekvő sorrendben írja ki, akkor az* keresőfa*!*

A fenti feladat állításainak alapvető következménye, hogy amennyiben egy bináris fa transzformáció a fa inorder bejárását nem változtatja meg, és adott egy bináris keresőfa, akkor a fa a transzformáció végrehajtása után is bináris keresőfa marad (mivel a kiindulási fa inorder bejárása szigorúan monoton növekvő kulcssorozatot ad, és ez a transzformáció után is így marad).

Ezt a tulajdonságot a bináris keresőfák kiegyensúlyozásánál fogjuk kihasználni, az AVL fákról szóló fejezetben.

A továbbiakban feltesszük, hogy a bináris keresőfákat láncoltan ábrázoljuk, szülő pointerek nélkül, azaz a fák csúcsai **Node** típusúak, és az üres fát a $\oslash$ pointer reprezentálja. (Ld. a **Bináris fák reprezentációi: láncolt ábrázolás** fejezetet!)

9.7. Bináris keresőfák: keresés, beszúrás, törlés

A gyakorlati programokban egy adathalmaz tipikus műveletei a következők: adott kulcsú rekordját keressük, a rekordot a kulcsa szerint beszúrjuk, illetve adott kulcsú rekordot törlünk. Ha a keresés a megtalált rekord címét adja vissza, így az adatmezők frissítése is megoldható, és ha nincs a keresett kulcsú rekord, azt egy $\oslash$ pointer visszaadásával jelezhetjük.

Az alábbi programokban az egyszerűség kedvéért az adat és a kulcs ugyanaz, mert a bináris fák kezelésének jellegzetességeit így is be tudjuk mutatni. Ugyanezen okból a műveleteket egy halmaz és egy kulcs közötti halmazműveletek megvalósításának tekintjük.

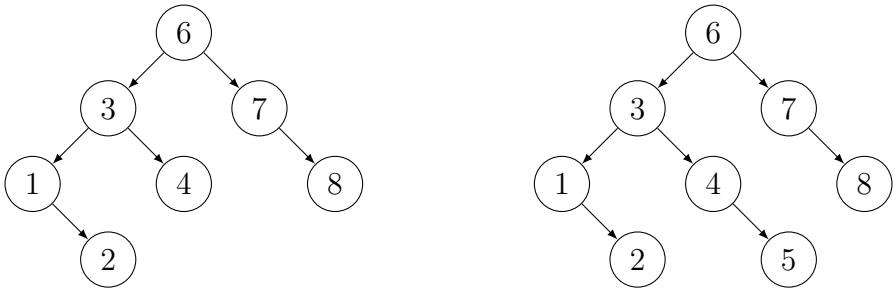
A bináris keresőfák műveletei:
- search(t, k) függvény : ha $k \in H(t)$, akkor a k kulcsú csúcsra mutató pointerrel tér vissza, különben a $\oslash$ hivatkozással,
- insert(t, k) : a $H(t) := H(t) \cup \{k\}$ absztrakt művelet megvalósítása,
- min(t) függvény : a $t \neq \oslash$ fában a minimális kulcsú csúcs címét adja vissza, pontosabban, az inorder bejárás szerinti első csúcsét,
- remMin$(t, minp)$: a $t \neq \oslash$ fából kivesszük a min(t) függvény által meghatározott csúcsot, és a címét a $minp$ pointerben adjuk vissza.
- del(t, k) : a $H(t) := H(t) \setminus \{k\}$ absztrakt művelet megvalósítása,
Mindegyik műveletre $MT(h) \in \Theta(h)$ (ahol $h = h(t)$).
(Ld. az alábbi struktogramokat!)

A search(t, k) függvény a t fában, a bináris keresőfa definíciója alapján megkeresi a k kulcs helyét. A kulcsot akkor és csak akkor találja meg, ha ott nemüres részfa van.

A insert(t, k) eljárás is megkeresi a t fában a k kulcs helyét. Ha ott egy üres részfát talál, akkor az üres részfa helyére tesz egy új levélcsúcsot, k kulccsal (18. ábra).

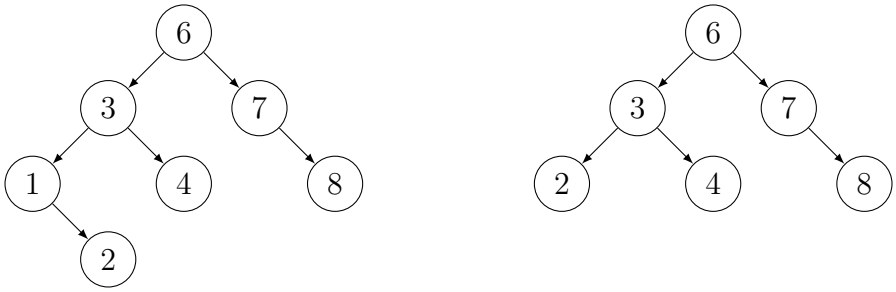
A min(t) függvény a t nemüres fa "bal alsó" csúcsára hivatkozó pointerrel tér vissza.

A remMin$(t, minp)$ eljárás $minp$-ben a t nemüres fa "bal alsó" (a legkisebb kulcsot tartalmazó) csúcsára mutató pointerrel tér vissza, de még előtte a



18. ábra. A bal oldali bináris keresőfába az 5 beszúrásával a jobb oldalit kapjuk. Először az 5-öt összehasonlítjuk a gyökércsúcs kulcsával, ami 6. Mivel $5 < 6$, az 5-öt a bal oldali részfába szúrjuk be. Ezután az 5-öt a 3-mal hasonlítjuk össze ($5 > 3$), majd a 4-gyel. Mivel $5 > 4$, a 4-es csúcs jobb oldali, üres részfájába kell beszúrni. Ehhez létre kell hozni egy új csúcsot aminek 5 lesz a kulcsa, és minkét részfája üres.

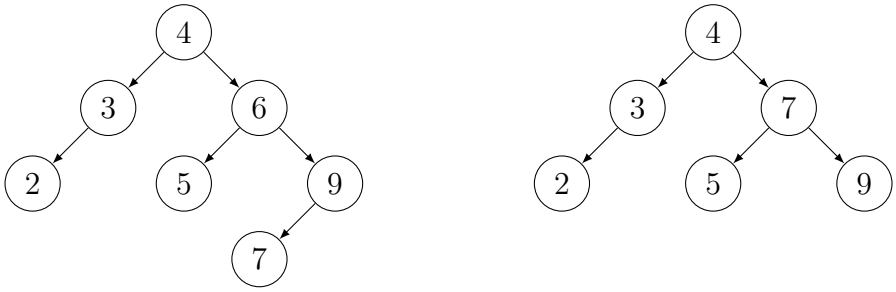
csúcsot kifűzi a fából, azaz a csúcshoz tartozó részfa helyére teszi a csúcs jobb oldali részfáját (19. ábra).



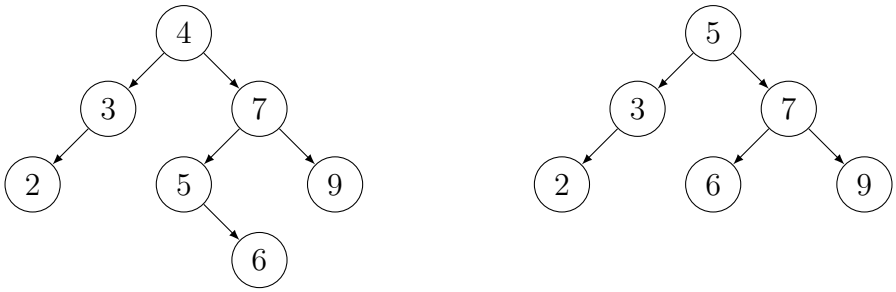
19. ábra. A bal oldali bináris keresőfából a minimális kulcsú csúcsot eltávolítva a jobb oldali keresőfa marad. Ehhez eredeti fa leginkább balra eső csúcsát eltávolítva, azt a csúcs jobb oldali részfájával helyettesítjük. Vegyük észre, hogy a bal oldali fából az 1 kulcsú csúcsot törölve a végeredményként adódó fa ugyanaz, mint az előbb.

A $\mathrm{del}(t, k)$ eljárás szintén megkeresi a t fában a k kulcs helyét. Ha megtalálta a k kulcsot tartalmazó csúcsot, még két eset lehet. (1) Ha a csúcs egyik részfája üres, akkor a csúcshoz tartozó részfa helyére teszi a csúcs másik részfáját. (2) Ha a csúcsnak két gyereke van, akkor a $\mathrm{remMin}(t, minp)$ eljárás segítségével *kiveszi* a jobb oldali részfából a minimális kulcsú csúcsot, és a k kulcsú csúcs helyére teszi, hiszen ez a kulcs a bal oldali részfa kulcsainál nagyobb, a jobb oldali részfa maradék kulcsainál pedig kisebb (20. és 21. ábra). [Vegyük észre, hogy a (2) esetben a bal oldali részfából a maximális

kulcsú csúcsot is kivehetnénk (22. ábra)! Az egyértelműség kedvéért a számonkéréseknél ilyenkor mindig a jobb oldali részfa minimumát kell kivenni.]

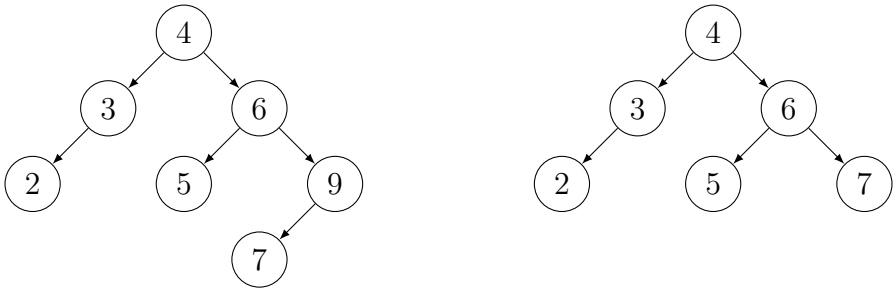


20. ábra. A bal oldali bináris keresőfából a 6 kulcsú csúcsot törölve kapjuk a jobb oldalit. Ennek a csúcsnak 2 gyereke van. Ezért a jobb oldali részfája legkisebb kulcsú csúcsát eltávolítjuk, és a 6 kulcsú csúcs helyére tesszük.

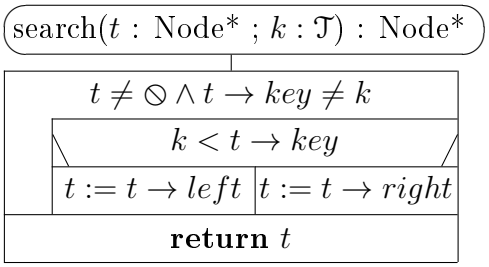


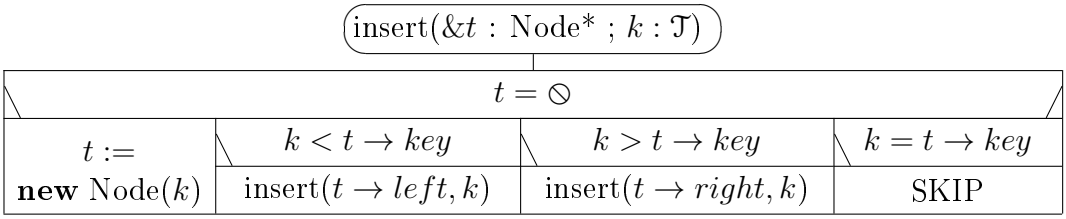
21. ábra. A bal oldali keresőfa gyökércsúcsát törölve kapjuk a jobb oldalit. Mivel a gyökércsúcsnak két gyereke van, a jobb oldali részfája legkisebb kulcsú csúcsát eltávolítjuk, és a 4 kulcsú csúcs helyére tesszük. (Vegyük észre, hogy a bal oldali részfa legnagyobb kulcsú csúcsát is eltávolíthatnánk, hogy a gyökércsúcs helyére tegyük. Az egyértelműség kedvéért a számonkéréseknél ilyenkor mindig a jobb oldali részfa minimumát kell eltávolítani.)

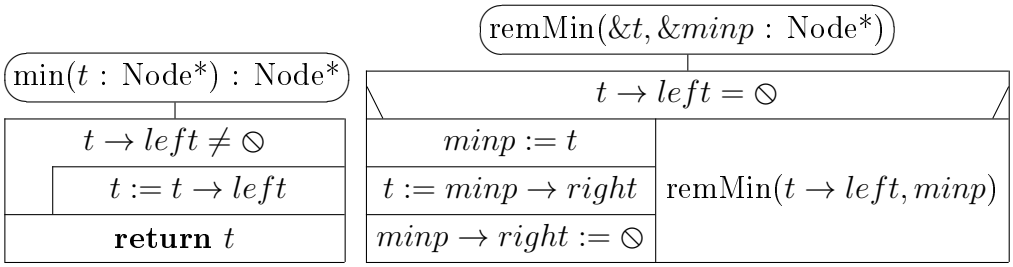
9.5. Feladat. *Írjuk meg a $t \neq \emptyset$ bináris keresőfából a maximális kulcsú csúcs kiolvasása / kivétele műveleteket (22. ábra). Mekkora lesz a futási idő? Miért? Írjuk át a fenti műveleteket szülő pointeres csúcsok esetére! Próbáljuk meg a nemrekurzív programokat rekurzívvá, a rekurzívakat nemrekurzívvá átírni, megtartva a futási idők nagyságrendjét!*

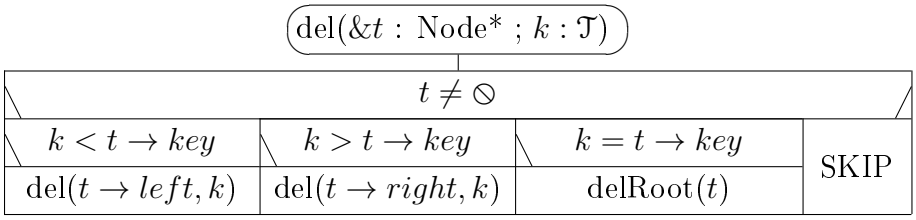


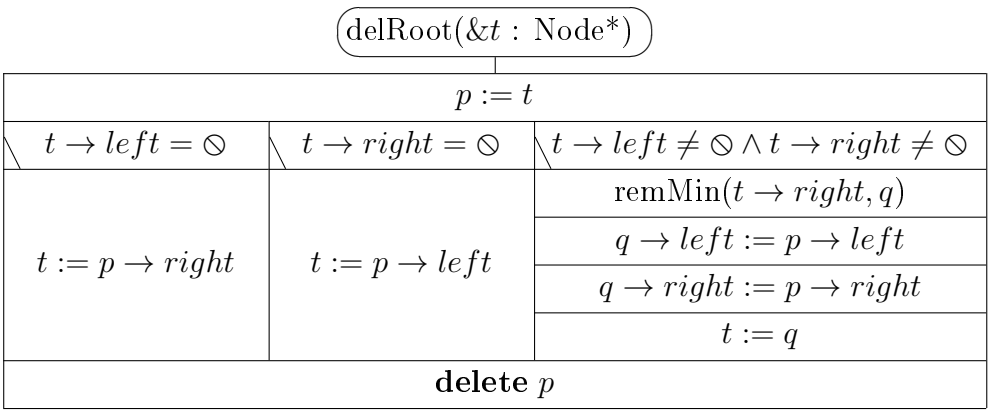
22. ábra. A bal oldali bináris keresőfa fa legnagyobb kulcsú csúcsát eltávolítva a jobb oldalit kapjuk: Az eredeti fa jobb szélső csúcsát kivesszük, és a saját bal részfájával helyettesítjük. Vegyük észre, hogy az eredeti fából a 9 kulcsú csúcs törlése is ugyanazt a fát eredményezi.











Mivel mindegyik műveletre $MT(h) \in \Theta(h)$ (ahol $h = h(t)$), a műveletek hatékonysága alapvetően a bináris keresőfa magasságától függ. Ha a fának n csúcsa van, a magassága $\lfloor \log n \rfloor$ (majdnem teljes fa esete) és $(n-1)$ (listává torzult fa esete) között változik. Ez teljesen attól függ, hogy milyen műveleteket, milyen sorrendben és milyen kulcsokkal hajtunk végre.

Szerencsére az a tapasztalat, hogy ha a kulcsok sorrendje, amikkel a beszúrásokat és törléseket végezzük, véletlenszerű, akkor a fa magassága általában $O(\log n)$ (bizonyítható, hogy nagy n-ekre átlagosan kb. $1,4 \log n$), és így a beszúrás és a törlés sokkal hatékonyabb, mintha az adathalmaz tárolására tömböket vagy láncolt listákat használnánk, ahol csak $O(n)$ átlagos műveletigényt tudnánk garantálni.

9.6. Feladat. *Igaz-e, hogy a bináris keresőfák fenti műveleteinek bármelyikére $mT(n) \in \Theta(1)$?*

Sok alkalmazás esetén azonban nem megengedhető az a kockázat, hogy ha a keresőfa (majdnem) listává torzul, akkor a műveletek hatékonysága is hasonló lesz, mint a láncolt listák esetén. Az ideális megoldás az lenne, ha tudnánk garantálni, hogy a fa majdnem teljes legyen. Nem ismerünk azonban olyan $O(\log n)$ futási idejű algoritmusokat, amelyek pl. a beszúrási és törlési műveletek során ezt biztosítani tudnák.

A vázolt probléma megoldására sokféle *kiegyensúlyozott keresőfa* fogalmat vezettek be. Ezek közös tulajdonsága, hogy a fa magassága $O(\log n)$, és a keresés, beszúrás, törlés műveleteinek futási ideje a fa magasságával arányos, azaz szintén $O(\log n)$. A kiegyensúlyozott keresőfák közül a legismertebbek az *AVL fák*, a *piros-fekete fák* (ezek idáig bináris keresőfák), a *B fák* és a *B+ fák* (ezek viszont már nem bináris fák). A legrégebbi, talán a legegyszerűbb, és a központi tárban kezelt adathalmazok ábrázolására mindmáig széles körben használt konstrukció az *AVL fa*. A modern adatbáziskezelő programok, fájlrendszerek stb., tehát azok az alapszoftverek és alkalmazások, amik háttértáron kezelnek keresőfákat, leginkább a *B+ fákat* részesítik előnyben. Ezért a következő félévben ez utóbbi két keresőfa típus tárgyalásával folytatjuk.

9.8. Teljes bináris fák, kupacok
Complete binary trees, heaps

A továbbiak előtt felidézzük a tökéletes (perfect) és a majdnem teljes (nearly complete) bináris fákkal kapcsolatos tudnivalókat.

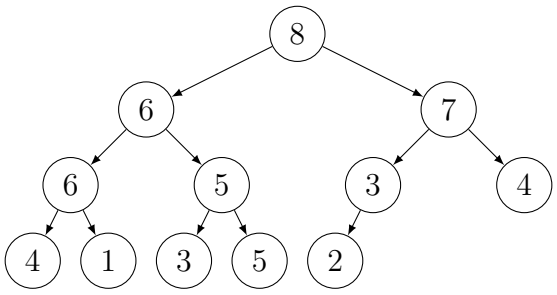
Azokat a bináris fákat, amelyekben minden belső (azaz nem-levél) csúcsnak két gyereke van, *szigorúan bináris fáknak* (*full binary trees*) nevezzük. Ha ez utóbbiaknak minden levele azonos szinten van, *tökéletes bináris fákról* beszélünk. (Ilyenkor az összes levél szükségszerűen a fa legmélyebb szintjén található, a felsőbb szinteken levő csúcsok pedig belső csúcsok, így két-két gyerekük van.) Tetszőleges h mélységű tökéletes bináris fa csúcsainak száma tehát $1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$.

Ha egy tökéletes bináris fa levélszintjéről nulla, egy vagy több levelet elveszünk, de nem az összeset, az eredményt *majdnem teljes bináris fának* nevezzük. Tetszőleges h mélységű, majdnem teljes bináris fa csúcsainak száma ezért $n \in 2^h \,..\, 2^{h+1} - 1$, és így $h = \lfloor \log n \rfloor$. Az alsó szinten levő leveleket elvéve pedig egy $h - 1$ mélységű tökéletes bináris fát kapunk.

9.7. Feladat. *Bizonyítsuk be, hogy tetszőleges nemüres, szigorúan bináris fának (*stricly binary tree*) pontosan eggyel több levélcsúcsa van, mint belső csúcsa.*

9.8. Feladat. *Egy bináris fa* méret szerint kiegyensúlyozott*, ha tetszőleges csúcsa bal és jobb részfájának mérete legfeljebb eggyel térhet el. Bizonyítsuk be, hogy a méret szerint kiegyensúlyozott bináris fák halmaza a majdnem teljes bináris fák halmazának valódi részhalmaza!*

9.9. Feladat. *Írjunk olyan eljárást, ami egy szigorúan monoton növekvő tömbből* méret szerint kiegyensúlyozott *bináris keresőfa másolatot készít, $O(n)$ futási idővel!*



23. ábra. majdnem teljes, balra tömörített (azaz teljes) bináris fa, ami egyben *(bináris maximum) kupac* is.

Egy majdnem teljes bináris fa *balra tömörített*, ha az alsó szintjén egyetlen levéltől balra sem lehet új levelet beszúrni. (Ld. a 23. ábrát!) Ez azt jelenti, hogy egy vele azonos mélységű tökéletes bináris fával összehasonlítva csak az alsó szint jobb széléről hiányozhatnak csúcsok (de a bal szélső csúcs kivételével akár az összes többi csúcs is hiányozhat). Eszerint bármely balra tömörített, majdnem teljes bináris fa alsó szintje fölötti szintjének bal szélétől egy vagy több belső csúcsot találunk, amelyeknek az utolsó kivételével biztosan két-két gyereke van. Ha az utolsónak csak egy gyereke van, akkor ez bal gyerek. A szint jobb szélén lehetnek levelek. A magasabb szinteken minden csúcsnak két gyereke van.

A balra tömörített, majdnem teljes bináris fákat más néven *teljes* (*complete*) bináris fáknak is nevezzük (hiszen, tetszőleges ilyen fát szintenként balról jobbra bejárva, egészen a legutolsó csúcsáig egyetlen csúcs sem hiányzik, a vele azonos magasságú tökéletes bináris fához viszonyítva).

Egy teljes bináris fát *maximum-kupacnak* (*heap*) nevezünk, ha minden belső csúcs kulcsa nagyobb-egyenlő, mint a gyerekeié. Ha minden belső csúcs kulcsa kisebb-egyenlő, mint a gyerekeié, *minimum-kupacról* beszélünk. Ebben a félévben *kupac* alatt maximum-kupacot értünk.

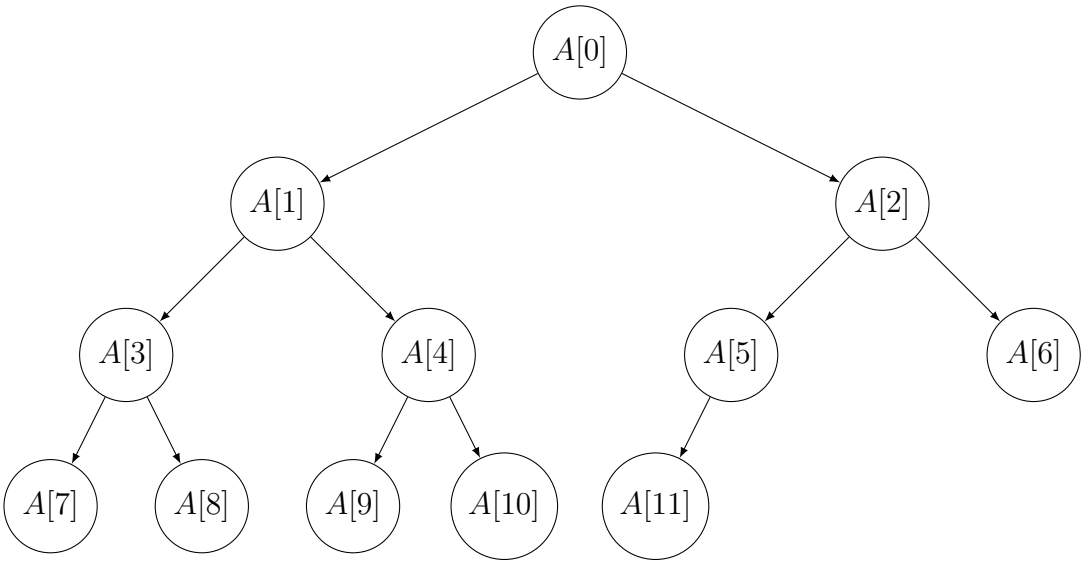
Vegyük észre, hogy bármely nemüres kupac maximuma a gyökércsúcsában mindig megtalálható, minimuma ugyanígy a levelei között, továbbá a kupac részfái is mindig kupacok. Egy kupac bal- és jobb oldali részfájában levő kulcsok között viszont nincs semmi nagyságrendi kapcsolat. Az elsőbbségi (prioritásos) sorokat általában kupacok segítségével ábrázoljuk.

Egy teljes bináris fát *csonka kupacnak* nevezünk, ha minden szülő-gyerek párosban a szülő kulcsa nagyobb-egyenlő, mint a gyereke kulcsa, kivéve, ha a szülő a gyökércsúcs. A gyökércsúcs kulcsa is definiált, de lehet, hogy kisebb, mint a gyereke kulcsa.

A csonka kupacok egy tömb kupaccá alakítása során jönnek majd létre, mint átmeneti adatszerkezetek. (Ld. a "Kupacrendezés" fejezetet!)

9.9. Teljes bináris fák aritmetikai ábrázolása

A teljes bináris fákat, speciálisan a kupacokat, szokás szintfolytonosan egy tömbben ábrázolni, ami legyen az egyszerűség kedvéért az $A : \mathcal{T}[m]$ tömb! Ha a fának n csúcsa van, akkor az $n \leq m$ feltételnek kell teljesülnie, és a csúcsokat szintfolytonosan a tömb első n elemében, az $A[0\,..\,n)$ résztömbben tároljuk, ahol $A[0]$ a fa gyökércsúcsa, feltéve, hogy $n > 0$ ($n = 0$ az üres fa esete). Az $A[i]$ csúcs gyerekei $A[left(i)]$ és $A[right(i)]$, feltéve, hogy $left(i) < n$, illetve $right(i) < n$. Ha egy csúcsnak két gyereke van, akkor a szintfolytonos ábrázolás miatt $right(i) = left(i) + 1$. Ha tehát $right(i) < n$, akkor az $A[i]$ csúcsnak két gyereke van, ha $right(i) = n$, akkor az $A[i]$ csúcsnak csak a bal oldali gyereke létezik, ha pedig $left(i) \geq n$, akkor az $A[i]$ csúcs levélcsúcs. Az $A[j]$ csúcs szülője $A[parent(j)]$, a $j > 0$ feltétellel.



24. ábra. Egy 12 csúcsú, teljes bináris fa, amit az A tömb első 12 elemében tároltunk. A fa csúcsait szintfolytonosan helyeztük el az $A[0\,..\,12)$ résztömbben.

Amennyiben létezik, az i indexű csúcs bal oldali gyerekének indexe $left(i) = 2i + 1$, mert az i indexű csúcsot a szintfolytonos ábrázolásban i csúcs előzi meg (az $A[0\,..\,i)$ résztömb elemei). Az i indexű csúcs bal oldali gyerekét tehát (a szintfolytonos ábrázolás miatt) megelőzi ennek az i csúcsnak a $2i$ gyereke, plusz a gyökércsúcs, aminek nincs szülője. Ez összesen $2i + 1$ csúcs, tehát $left(i)$ indexe $2i + 1$. A szintfolytonos reprezentációból következően pedig $right(i) = left(i) + 1 = 2i + 2$. (Ld. a 24. ábrát!)

Más részről, ha egy csúcs indexe $j > 0$, akkor a szülőjének indexe $parent(j) = \lfloor \frac{j-1}{2} \rfloor$. Ez abból következik, hogy akár $j = left(i) = 2i + 1$, akár $j = right(i) = 2i + 2$ esetén $\lfloor \frac{j-1}{2} \rfloor = i$.

9.10. Feladat. *Tegyük fel, hogy az n elemű, teljes bináris fát az egytől indexelt $A/1 : \mathcal{T}[m]$ tömbben tároljuk szintfolytonosan, azaz az $A[1\,..\,n]$ résztömbben. Ebben az esetben $A[1]$ a fa gyökércsúcsa, feltéve, hogy $n > 0$ ($n = 0$ az üres fa esete).*

Gondoljuk meg, hogy ekkor az $A[i]$ csúcs gyerekeire $left(i) = 2i$, illetve $right(i) = 2i + 1$, sorban a $2i \leq n$, illetve a $2i < n$ feltétellel.
Az $A[j]$ csúcs szülőjére pedig $j > 1$ esetén $parent(j) = \lfloor \frac{j}{2} \rfloor$.

9.10. Kupacok és elsőbbségi (prioritásos) sorok

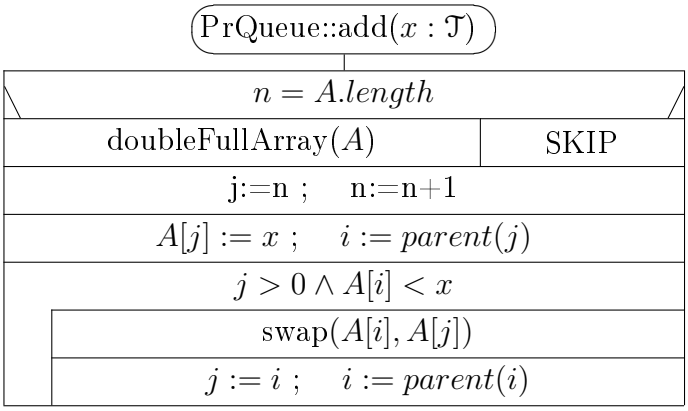
Az elsőbbségi sor (priority queue) egy zsák (multihalmaz), amelybe be tudunk tenni újabb elemeket, és ki tudjuk választani, illetve kivenni az egyik maximális elemét. (Min prioritásos sor esetén az egyik minimális elemét tudjuk kiválasztani, illetve kivenni.)

Az alábbiakban a prioritásos sor típust a PrQueue osztály segítségével írjuk le. Az elsőbbségi sor aktuális elemeit az $A[1\,..\,n]$ résztömb tartalmazza, ami egy kupac.

PrQueue
− $A : \mathcal{T}[]$ // $\mathcal{T}$ is some known type ; $A.length$ is the physical − constant $m0 : \mathbb{N}_+ := 16$ // size of the PrQ, its default is m0. − $n : \mathbb{N}$ // $n \in [0\,..\,A.length]$ is the actual length of the PrQ

+ PrQueue($m : \mathbb{N}_+ := m0$){ $A :=$ **new** $\mathcal{T}[m]$; $n := 0$ } // create an empty PrQ
+ add($x : \mathcal{T}$) // insert x into the priority queue
+ remMax():$\mathcal{T}$ // remove and return the maximal element of the priority queue
+ max():$\mathcal{T}$ // return the maximal element of the priority queue
+ isEmpty() : $\mathbb{B}$ {**return** $n = 0$}
+ $\sim$ PrQueue() { **delete** A }
+ setEmpty() {$n := 0$} // reinitialize the priority queue

Ha az A tömb n hosszúságú kezdőszelete egy kupac aritmetikai ábrázolása,
$MT_{\text{add}:n<A.length}(n) \in \Theta(\log n)$, $MT_{\text{add}:n=A.length}(n) \in \Theta(n)$,
$AT_{\text{add}}(n) \in O(\log n)$, $mT_{\text{add}}(n) \in \Theta(1)$,
$MT_{\text{remMax}}(n) \in \Theta(\log n)$, $mT_{\text{remMax}}(n) \in \Theta(1)$, $T_{\max}(n) \in \Theta(1)$.



Az add(x) esetében a szintfolytonosan első üres, azaz az $A[j]$ helyen x-et hozzákapcsoljuk a kupachoz. Ezzel elronthatjuk a kupacot, ezért x-et *fölemeljük*, azaz addig cserélgetjük – mindig az aktuális szülőjével, az $A[i]$-vel –, amíg van szülője, és $x >$ mint a szülője. Így x felfelé mozog a kupacban, és $>$ mint a leszármazottai. Ha x felér a gyökérbe, vagy már $\leq$ mint a szülője, akkor helyreállt a kupac. (Ld. a 25. és a 26. ábrákat!)

$MT_{\text{add}:n<A.length}(n) \in \Theta(\log n)$, hiszen a ciklus legfeljebb annyiszor iterál, amennyi a fa magassága, azaz (n input értékéhez viszonyítva) $\lfloor \log(n+1) \rfloor$-szer, amiből $\log n \leq MT_{\text{add}:n<A.length}(n) = \lfloor \log(n+1) \rfloor + 1 \leq \log n + 2$.

$MT_{\text{add}:n=A.length}(n) \in \Theta(n)$, hiszen a doubleFullArray(A) eljáráshívás ciklusa $n = A.length$ iterációt hajt végre, ami dominál az x fölemelése és egyéb tevékenységek $O(\log n)$ műveletigénye fölött.

$mT_{\text{add}}(n) \in \Theta(1)$, ha ugyanis $n < A.length$, akkor a doubleFullArray(A) eljárás nem hívódik meg, és ha a kupachoz adott elem (x) elég kicsi, akkor a ciklus egyet sem iterál, innét pedig $mT_{\text{add}}(n) = 1$.

$T_{\max}(n) \in \Theta(1)$, mivel $T_{\max}(n) = 1$.

$AT_{\text{add}}(n) \in O(\log n)$, mivel

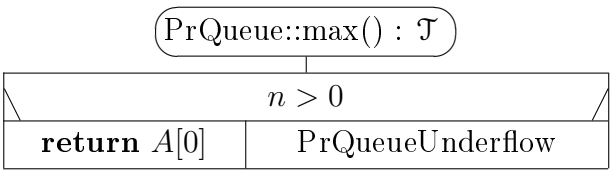
- egyrészt $MT_{\text{add}:n<A.length}(n) \in O(\log n)$,
- másrészt $n = A.length$ esetén a doubleFullArray(A) eljáráshívás ciklusa n iterációt hajt végre, ami kettesével képzeletben szétoszthatóan az előző $n/2$ add() hívás között, amikor biztosan nem hívtuk meg a

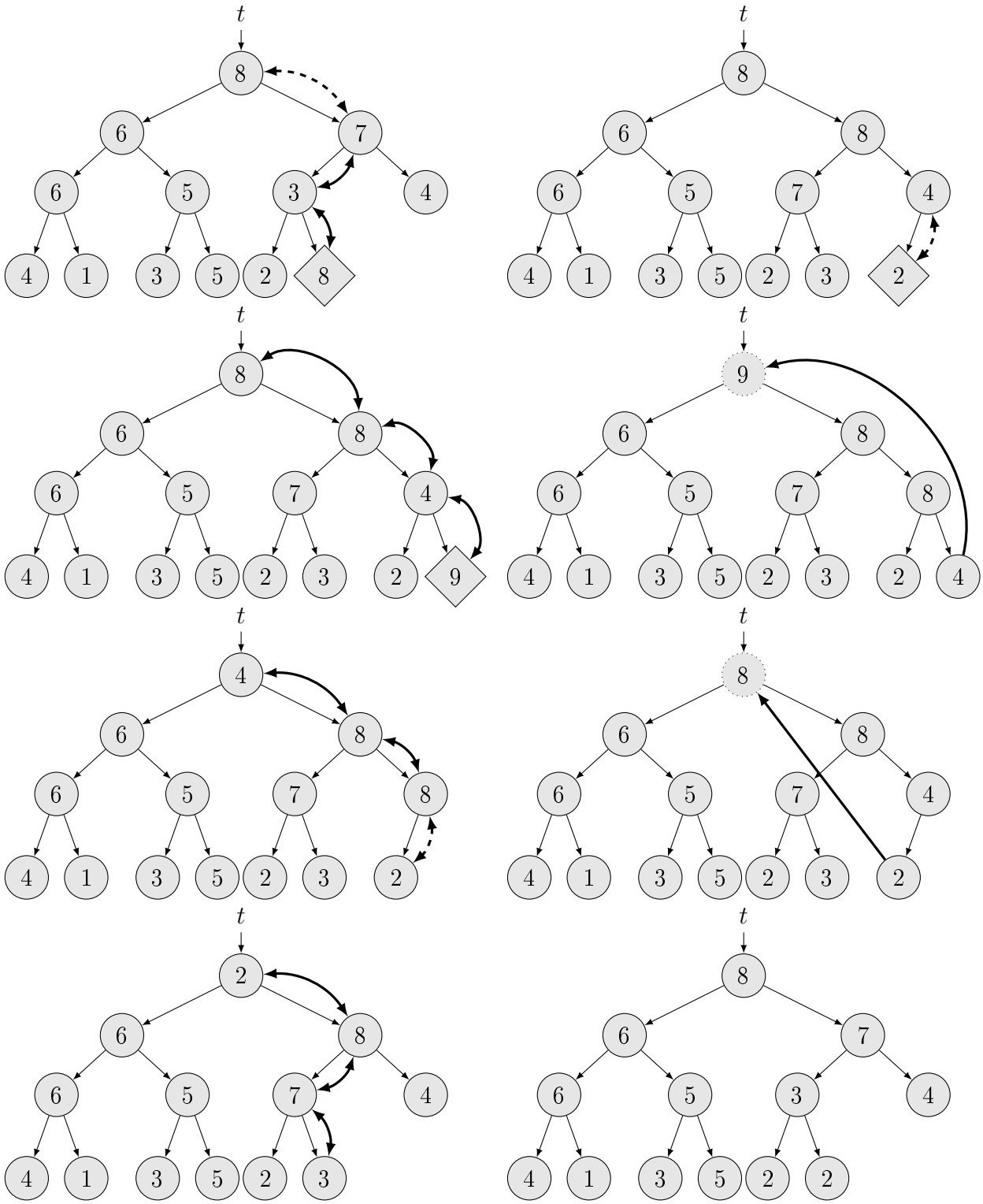
op	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-	8	6	7	6	5	3	4	4	1	3	5	2			
add(8)	8	6	7	6	5	*3	4	4	1	3	5	2	#8		
...	8	6	*7	6	5	#8	4	4	1	3	5	2	3		
...	*8	6	#8	6	5	7	4	4	1	3	5	2	3		
.	8	6	8	6	5	7	4	4	1	3	5	2	3		
add(2)	8	6	8	6	5	7	*4	4	1	3	5	2	3	#2	
.	8	6	8	6	5	7	4	4	1	3	5	2	3	2	
add(9)	8	6	8	6	5	7	*4	4	1	3	5	2	3	2	#9
...	8	6	*8	6	5	7	#9	4	1	3	5	2	3	2	4
...	*8	6	#9	6	5	7	8	4	1	3	5	2	3	2	4
.	9	6	8	6	5	7	8	4	1	3	5	2	3	2	4
remMax()	∼9	6	8	6	5	7	8	4	1	3	5	2	3	2	∼4
$max := 9$	*4	6	#8	6	5	7	8	4	1	3	5	2	3	2	
...	8	6	*4	6	5	7	#8	4	1	3	5	2	3	2	
...	8	6	8	6	5	7	*4	4	1	3	5	2	3	#2	
return 9	8	6	8	6	5	7	4	4	1	3	5	2	3	2	
remMax()	∼8	6	8	6	5	7	4	4	1	3	5	2	3	∼2	
$max := 8$	*2	6	#8	6	5	7	4	4	1	3	5	2	3		
...	8	6	*2	6	5	#7	4	4	1	3	5	2	3		
...	8	6	7	6	5	*2	4	4	1	3	5	2	#3		
return 8	8	6	7	6	5	3	4	4	1	3	5	2	2		

25. ábra. Egy, az $A/1 : \mathbb{N}[15]$ tömbben ábrázolt kupac változásai, add() és remMax() műveletek hatására. Az add() műveleteknél „#” mutatja az aktuális elemet, „$*$” a szülőjét. Hasonlóan, a lesüllyesztéseknél „$*$” jelöli a szülőt, „#” a nagyobbik gyerekét. A remMax() műveleteknél „∼” jelzi az eltávolítandó maximumot és a helyére teendő kulcsot is.

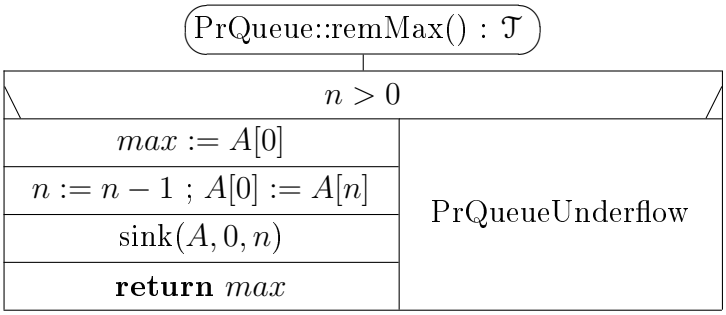
doubleFullArray(A) eljárást. Így mind az $n/2$ add() hívás $O(\log n)$ műveletigénye kettővel nő meg, azaz $O(\log n)$ marad.

A fenti technikát *amortizált műveletigény számításnak* nevezzük, és nyilvánvalóan alaklmazható a vermeknél a push(), a soroknál pedig az add() metódus átlagos műveletigényének meghatározásához is.





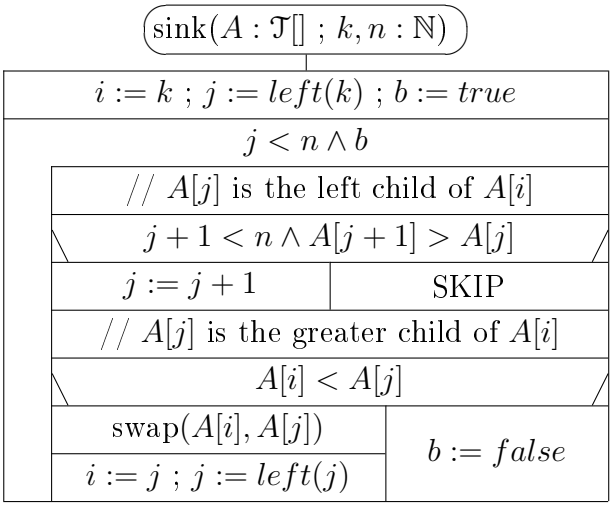
26. ábra. Heap operation visualization based on Figure 25:
add(8), add(2), add(9), remMax()=9, remMax()=8.



A remMax() metódus a maximum elmentése után a kupac gyökerébe, $A[0]$-ba teszi át a szintfolytonosan utolsó elemet. Ezzel a kupac mérete eggyel csökken, és a gyökerénél valószínűleg el is romlik (ún. csonka kupac lesz). Ezért a "sink" eljárás segítségével (amit mindig $k = 0$-val hív meg) a gyökérbe átrakott elemet ($A[i]$) addig süllyeszti lefelé, amíg a helyére nem kerül. Ennek során a lesüllyesztendő elemet mindig az aktuálisan nagyobb gyerekével cseréli meg, amíg még a levélszint fölött van, és a nagyobbik gyereke nagyobb nála. Ilyen módon a ciklus során a lesüllyedő elem mindig kisebb, mint az ősei. Amikor a ciklus megáll, akkor vagy leért a levélszintre, vagy $\geq$ mint a gyerekei, és kupac helyreáll.

9.11. Feladat. *A* remMax() *metódus fenti magyarázatánál kimondatlanul feltettük, hogy az* $n := n - 1$ *utasítás után még* $n > 0$*, azaz az* $A[0] := A[n]$ *értékadással egy nemüres csonka kupac alakul ki. Vizsgáljuk meg azt az esetet, amikor az* $n := n - 1$ *utasítás után* $n = 0$ *teljesül!*

Vegyük észre, hogy a sink eljárást az itt szükségesnél kicsit általánosabban írtuk meg! Akkor is működik, ha a lesüllyesztendő elem eredetileg tetszőleges részfa gyökerében van. Ilyenkor az adott részfa – mint a gyökerénél szabálytalan kupac – kupaccá alakítására fogjuk használni. (Ld. a "Kupacrendezés" fejezetet!)



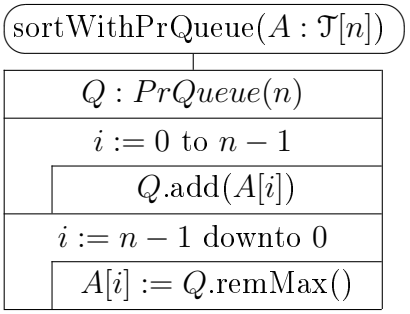
A remMax() metódus műveletigényének meghatározásához először megvizsgáljuk a sink() eljárást. Mivel aszimptotikus műveletigényt számolunk, feltehető, hogy a k gyökerű csonka kupac magassága $h \geq 0$. Akkor $MT_{\text{sink}}(h) = h + 1$, ugyanis a ciklus legfeljebb h iterációt végez. Továbbá $1 \leq mT_{\text{sink}}(h) \leq 2$, mert lehet, hogy a ciklus legfeljebb egyet iterál. Speciálisan $k = 0$ esetére: $\log n \leq MT_{\text{sink}}(n) = h + 1 = \lfloor \log n \rfloor + 1 \leq \log n + 1$ ahol n a kupac aktuális mérete.

Innét a kupacnak a remMax meghívásának pillanatában érvényes n méretére $\log n \leq \log(n-1) + 1 \leq MT_{\text{remMax}}(n) \leq \log(n-1) + 2 \leq \log n + 2$.
Ebből $MT_{\text{remMax}}(n) \in \Theta(\log n)$.

$mT_{\text{remMax}}(n) \in \Theta(1)$, hiszen
$2 = 1 + 1 \leq mT_{\text{remMax}}(n) = 1 + mT_{\text{sink}}(n-1) \leq 1 + 2 = 3$.

9.10.1. Rendezés elsőbbségi sorral

A fenti elsőbbségi sor segítségével könnyen és hatékonyan rendezhetünk tömböket:



A fenti rendezésben a Q.add($A[i]$) és az $A[i] := Q$.remMax() utasítások $O(\log n)$ hatékonyságúak, hiszen az elsőbbségi sort reprezentáló kupac magassága $\leq \log n$. Ezért a rendezés műveletigénye $O(n \log n)$.

Maximális műveletigénye $\Theta(n \log n)$. Ha ugyanis az input tömb szigorúan monoton növekvő, akkor az első ciklus által megvalósított kupacépítés minden új csúcsot a fa gyökeréig mozgat fel. Amikor pedig a végső kupac leendő leveleit szúrjuk be, már legalább $\lfloor \log n \rfloor - 1$ mélységű a fa, és a leendő levelek száma $\lceil \frac{n}{2} \rceil$. Ekkor tehát csak a levelek beszúrásának futási ideje $\Omega(n \log n)$, így a teljes futási idő is. Az előbbi $O(n \log n)$ korláttal együtt adódik az állítás.

A fenti rendezés maximális műveletigénye tehát aszimptotikusan kisebb, mint a beszúró rendezésé, ami $\Theta(n^2)$. Ráadásul az $\frac{n \log n}{n^2} = \frac{\log n}{n}$ hányados már $n = 1000$ esetén is csak ≈ 0.01, $n = 10^6$ esetén pedig ≈ 0.00002, tehát gyorsan tart a nullához. A sortWithPrQueue hátránya, hogy a rendezendő tömbbel azonos méretű $M(n) \in \Theta(n)$ munkamemóriát igényel – a prioritásos sorban tárolt kupac számára –, míg a beszúró rendezésre $M(n) \in \Theta(1)$, hiszen csak néhány egyszerű segédváltozóra van szükségünk. Ezt a problémát kupacrendezéssel oldhatjuk meg.

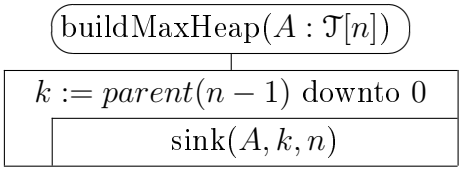
9.11. Kupacrendezés (heap sort)

A kupacrendezés a fenti sortWithPrQueue($A : \mathcal{T}[]$) optimalizálása.

Egyrészt az algoritmust *helyben rendezővé* alakítjuk: az $A : \mathcal{T}[n]$ tömbön kívül csak $\Theta(1)$ memóriát használunk, míg a fenti eljárásnak szüksége van egy $\Theta(n)$ méretű segédtömbre, amiben a prioritásos sort ábrázoló kupacot tárolja.

Másrészt a kupac felépítését optimalizáljuk, ami a fenti esetben magában véve is $O(n \log n)$ műveletigényű, maximális futási ideje pedig $\Theta(n \log n)$.

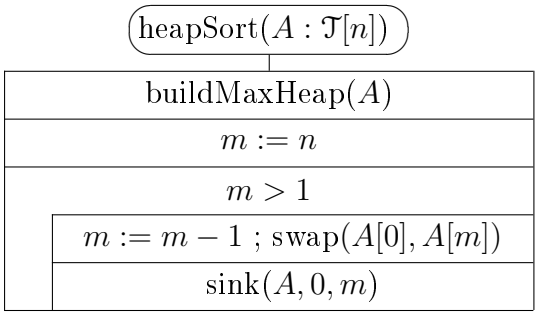
Először tehát kupaccá alakítjuk a tömböt, most lineáris műveletigénnyel (ld a 27. ábrát):



A fenti eljárás magyarázatához: Eredetileg az $A : \mathcal{T}[n]$ tömb, mint teljes bináris fa magassága $h = \lfloor \log n \rfloor$. Levelei önmagukban egyelemű kupacok. A $(h-1)$-edik szinten levő belső csúcsokhoz, mint gyökerekhez tartozó bináris fák tehát (egy magasságú) úgynevezett *csonka kupacok*, amikben egyedül a

gyökércsúcs tartalma lehet "rossz helyen". Ezeket tehát helyreállíthatjuk a gyökér lesüllyesztésével az alatta levő csonka kupacba. Ezután már a $(h-2)$-edik szinten levő csúcsokhoz, mint gyökerekhez tartozó bináris fák lesznek (kettő magasságú) *csonka kupacok*, amiket hasonlóan állíthatunk helyre, és így tovább, szintenként visszafelé. Utolsóként az $A[0]$-t süllyesztjük le az alatta lévő csonka kupacba, és ezzel az $A : \mathcal{T}[n]$ tömb kupaccá alakítása befejeződik. Annak érdekében, hogy a szintekkel ne kelljen külön foglalkozni, a fenti eljárásban a lesüllyesztéseket a szintfolytonosan utolsó belső csúccsal kezdtük (ami az utolsó csúcs, azaz $A[n-1]$ szülője), és innét haladtunk szintfolytonosan visszafelé.

A kupacrendezés fő eljárása:

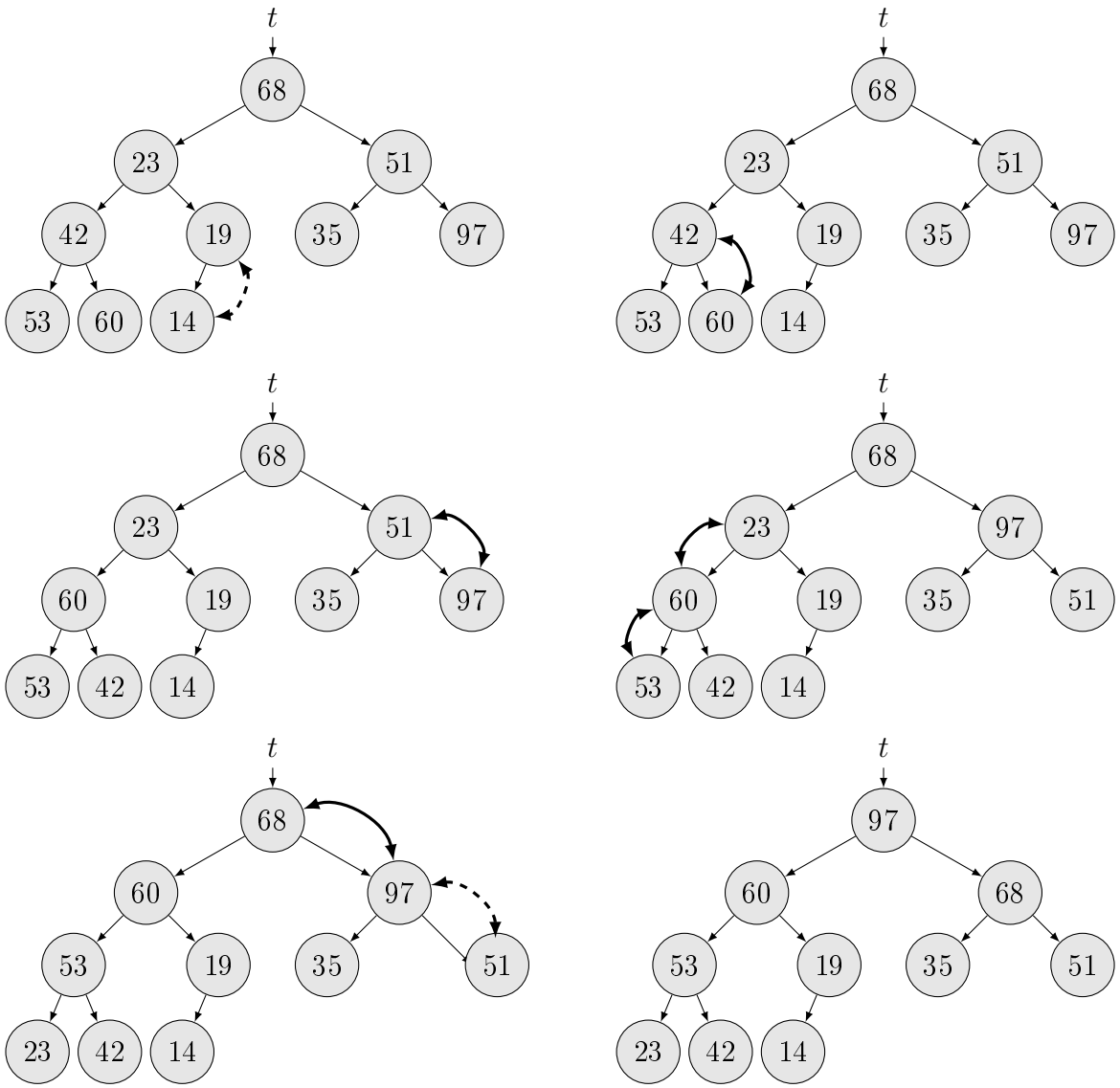


A kupaccá alakítás után tehát ezt ismételjük: Megcseréljük a kupac maximumát, azaz gyökerében lévő elemet ($A[0]$) a kupac szintfolytonosan utolsó elemével, miközben levágjuk a maximumot ($m := m-1$), majd lesüllyesztjük az $A[0\,..\,m)$ csonka kupacban a gyökérbe tett elemet. Így az előbbinél eggyel kisebb méretű kupacot kapunk ($A[0\,..\,m)$). Ezt a ciklusmagot addig ismételjük, amíg az $A[0\,..\,m)$ kupac mérete nagyobb, mint egy. Így az $A[0\,..\,n)$ tömbben visszafelé megkapjuk az első, második stb. maximumokat, és végül az $A[0]$-ban a minimumot, azaz a tömb rendezve lesz. (Ld. a 28. és a 29. ábrákat!

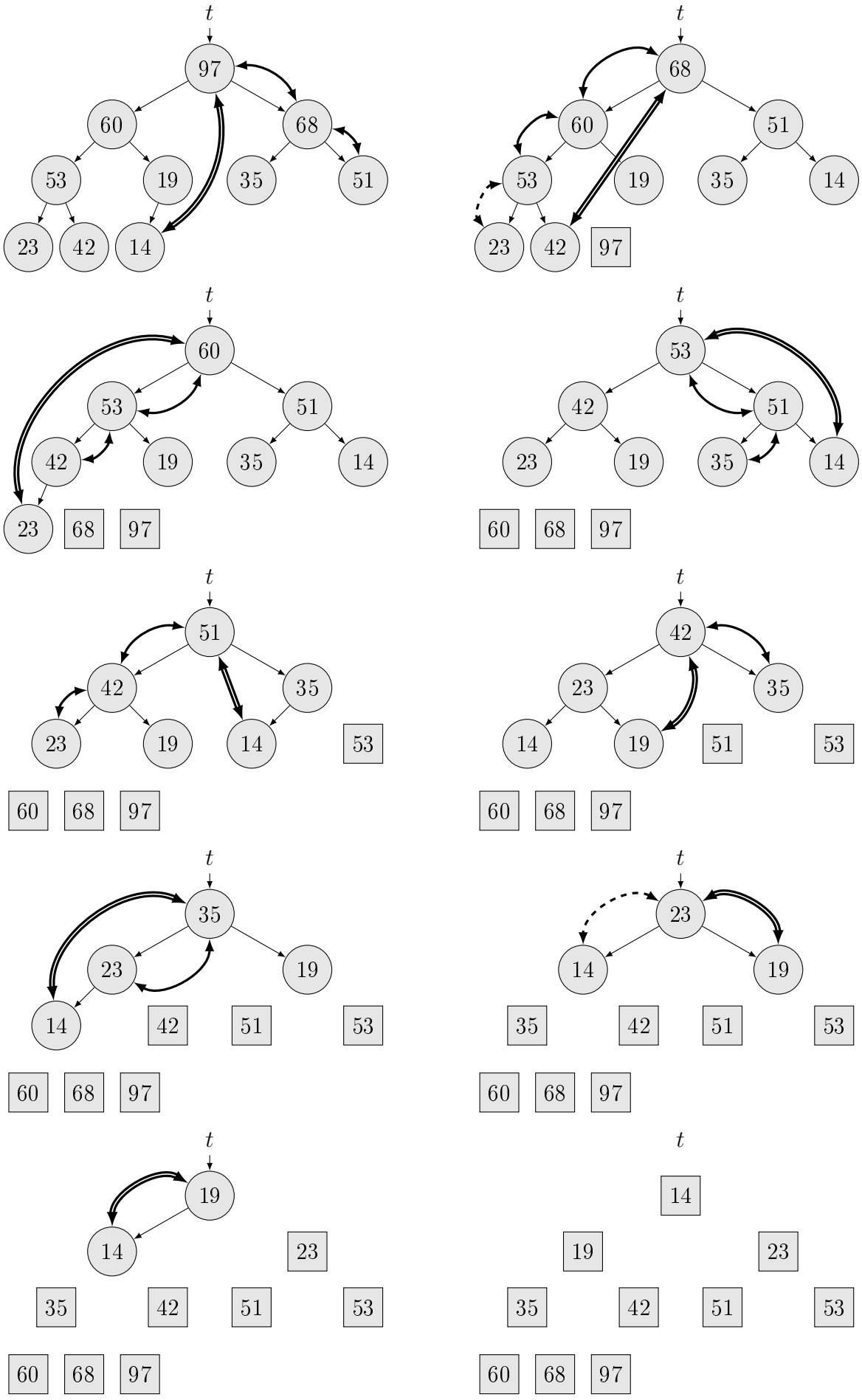
9.11.1. A kupacrendezés műveletigénye

A kupaccá alakítás utáni rész futási idejét a sink$(A, 0, m)$ hívások határozzák meg, amiknek műveletigénye $O(\log m)$ ($m \in \{n-1, n-2, \ldots, 1\}$), durva felső becsléssel $O(\log n)$. Az $(n-1)$ hívás tehát összesen $O(n \log n)$ futási idejű, és ezt nagyságrendileg a ciklus $(n-1)$ iterációja nem befolyásolja, mert ez csak $O(n)$ műveletigényt ad hozzá, ami aszimptotikusan kisebb, mint $O(n \log n)$.

A kupaccá alakítás műveletigényéről hasonlóan látható, hogy maga is $O(n \log n)$, ui. az iteráció és a sink(A, k, n) eljáráshívás is $\lfloor \frac{n}{2} \rfloor$-ször hajtódik



27. ábra. A $\langle 68, 23, 51, 42, 19, 35, 97, 53, 60, 14\rangle$ tömb kupaccá alakítása. A szaggatott nyilak olyan kulcsösszehasonlításokat jeleznek, ahol a lesüllyesztés véget ér, mert nincs szükség cserére. Jegyezzük meg, hogy végig tömbbel dolgozunk, a bináris fák csak a szemléltetéshez használatosak. Végül a $\langle 97, 60, 68, 53, 19, 35, 51, 23, 42, 14\rangle$ tömböt kapjuk. A teljes kupacrendezésnek ez csak az első része.

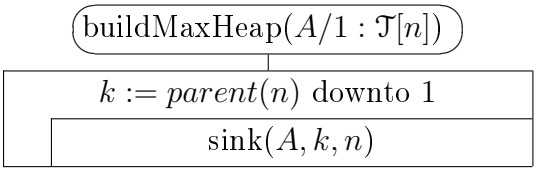


28. ábra. Heap sort visualization based on Figure 29. Here, we start with the result of buildMaxHeap(). Double arrows represent the swap of the largest element with the last element of the heap, fitting into its desired place. The subsequent arrows demonstrate the sinking process after the initial swap. Dashed arrows compare where the sinking ends because no swap is needed.

op	0	1	2	3	4	5	6	7	8	9
sink	68	23	51	42	*19	35	97	53	60	#14
sink	68	23	51	*42	19	35	97	53	#60	14
sink	68	23	*51	60	19	35	#97	53	42	14
sink	68	*23	97	#60	19	35	51	53	42	14
...	68	60	97	*23	19	35	51	#53	42	14
sink	*68	60	#97	53	19	35	51	23	42	14
...	97	60	*68	53	19	35	#51	23	42	14
.	97	60	68	53	19	35	51	23	42	14
swap	∼97	60	68	53	19	35	51	23	42	∼14
sink	*14	60	#68	53	19	35	51	23	42	+97
...	68	60	*14	53	19	35	#51	23	42	+97
swap	∼68	60	51	53	19	35	14	23	∼42	+97
sink	*42	#60	51	53	19	35	14	23	+68	+97
...	60	*42	51	#53	19	35	14	23	+68	+97
.	60	53	51	*42	19	35	14	#23	+68	+97
swap	∼60	53	51	42	19	35	14	∼23	+68	+97
sink	*23	#53	51	42	19	35	14	+60	+68	+97
...	53	*23	51	#42	19	35	14	+60	+68	+97
swap	∼53	42	51	23	19	35	∼14	+60	+68	+97
sink	*14	42	#51	23	19	35	+53	+60	+68	+97
...	51	42	*14	23	19	#35	+53	+60	+68	+97
swap	∼51	42	35	23	19	∼14	+53	+60	+68	+97
sink	*14	#42	35	23	19	+51	+53	+60	+68	+97
...	42	*14	35	#23	19	+51	+53	+60	+68	+97
swap	*42	23	35	14	#19	+51	+53	+60	+68	+97
sink	*19	23	#35	14	+42	+51	+53	+60	+68	+97
swap	∼35	23	19	∼14	+42	+51	+53	+60	+68	+97
sink	*14	#23	19	+35	+42	+51	+53	+60	+68	+97
swap	∼23	14	∼19	+35	+42	+51	+53	+60	+68	+97
sink	*19	#14	+23	+35	+42	+51	+53	+60	+68	+97
swap	*19	#14	+23	+35	+42	+51	+53	+60	+68	+97
sink	*14	+19	+23	+35	+42	+51	+53	+60	+68	+97
.	+14	+19	+23	+35	+42	+51	+53	+60	+68	+97

29. ábra. A teljes heap sort szemléltetése az $A : \mathbb{N}[10]$ tömbön. A lesüllyesztéseknél „$*$” jelöli a szülőt, „#” a nagyobbik gyerekét. A swap műveleteknél „$\sim$” mutatja a megcserélendő elemeket. A végleges helyükre érkezett kulcsokat „+”előjellel különböztettük meg.

végre, és a k gyökerű részfa magassága $\leq \log n$, így az eljáráshívás műveletigénye durva felső becsléssel $O(\log n)$. Az $\lfloor \frac{n}{2} \rfloor$ hívás tehát összesen $O(n \log n)$ futási idejű, és ezt nagyságrendileg a ciklus $\lfloor \frac{n}{2} \rfloor$ iterációja nem befolyásolja, mert ez csak $O(n)$ műveletigényt ad hozzá, ami aszimptotikusan kisebb, mint $O(n \log n)$.



Ennél finomabb becsléssel alább belátjuk, hogy a kupaccá alakítás $\Theta(n)$ műveletigényű. Ettől a teljes futási idő továbbra is $O(n \log n)$, hiszen a szekvenciában ebből a szempontból az aszimptotikusan nagyobb műveletigényű programrész dominál, de a sortWithPrQueue(A) eljárás kupacépítő ciklusához képest hatékonyabb kupaccá alakítással a gyakorlatban így is jelentős futási időt takaríthatunk meg.

9.12. Feladat. *Lássuk be, hogy a kupacrendezés maximális műveletigénye $\Theta(n \log n)$; használjuk fel ehhez a 10.2. definíciót és a 10.4. tételt!*

9.13. Feladat. *Lássuk be, hogy a kupacrendezés minimális műveletigénye $\Theta(n)$, és ez például akkor áll elő, amikor az input tömb minden eleme egyenlő!*

9.11.2. A kupaccá alakítás műveletigénye lineáris*

Szükségünk lesz a teljes, azaz a balra tömörített, majdnem teljes bináris fák néhány tulajdonságára. Az alábbi gondolatmenet egytől indexelt tömbökre vonatkozik. Ez nyilván nem befolyásolja a műveletigényt, de így kicsit egyszerűbb lesz a gonolatmenet és a képletek.

Emlékeztetünk arra, hogy ha egy ilyen, n csúcsú fát szintfolytonosan az $A/1 : \mathcal{T}[n]$ tömbben tárolunk, akkor $last(n) = n$, valamint az i indexű csúcs gyerekeinek indexei $2i$, illetve $2i + 1$, feltéve, hogy a gyerekek léteznek, azaz $2i \leq n$, illetve $2i + 1 \leq n$. Más részről, ha egy csúcs indexe $j > 1$, akkor a szülőjének indexe $\lfloor \frac{j}{2} \rfloor$.

A fentiekből adódik, hogy az $1..k$ sorszámú csúcsok szülei az $1..\lfloor \frac{k}{2} \rfloor$ sorszámú csúcsok, mert
- egyrészt, ha egy csúcs j indexére $1 \leq j \leq k$, és van szülője, azaz $j \geq 2$, akkor a szülője indexére $1 \leq \lfloor \frac{j}{2} \rfloor \leq \lfloor \frac{k}{2} \rfloor$;
- másrészt, ha az i indexű csúcsra $1 \leq i \leq \lfloor \frac{k}{2} \rfloor$, akkor ennek bal gyerekére $2 \leq 2i \leq 2\lfloor \frac{k}{2} \rfloor \leq k$, azaz van gyereke az $1..k$ sorszámú csúcsok között.

Eszerint egy n csúcsú teljes teljes bináris fának $fele(n) = \lfloor \frac{n}{2} \rfloor$ belső csúcsa van. Ha ugyanis a csúcsokat sorfolytonosan az $1 \,..\, n$ sorszámokkal indexeljük, akkor az előbbi állítás szerint a szüleik sorszámai $1 \,..\, \lfloor \frac{n}{2} \rfloor$.

Az előbb belátott állítás szerint egy n csúcsú, teljes bináris fának $\lceil \frac{n}{2} \rceil$ levele van.

Jelölje a továbbiakban n_d tetszőleges n méretű, teljes bináris fa d magasságú részfáinak számát, $n_{\geq d}$ pedig a fa legalább d magasságú részfáinak számát, ahol $1 \leq d \leq h = \lfloor \log n \rfloor$, azaz h az eredeti fa magassága! Ekkor $n_{\geq 1} = fele(n)$, hiszen a legalább egy magasságú részfák gyökércsúcsai éppen a belső csúcsok. Továbbá $n_{\geq 2} = fele(fele(n)) = fele^2(n)$, hiszen a legalább kettő magasságú részfák gyökércsúcsai éppen a legalább egy magasságú részfák gyökércsúcsainak szülei, és mivel az utóbbiak szinfolytonosan $1 \,..\, fele(n)$ sorszámúak, azért az előbbiek $1 \,..\, fele^2(n)$ sorszámúak. Teljes indukcióval adódik

$$\sum_{m=d}^{h} n_m = n_{\geq d} = fele^d(n) \leq \frac{n}{2^d}$$

Most már áttérhetünk a kupaccá alakítás lineáris műveletigényének bizonyítására.

Szemléletesen fogalmazva, a sortWithPrQueue(A) eljárás kupacépítő ciklusához képest abból adódik a hatékonyság növekedése, hogy ott az elemek túlnyomó részét már a végsőhöz közeli magasságú kupacba szúrjuk be, míg itt a fa leveleit, az elemek felét egyáltalán nem kell lesüllyeszteni, ezek szüleit, az elemek negyedét egy magasságú fában süllyesztjük le, nagyszüleit, az elemek nyolcadát kettő magasságú fában stb.

Ahhoz, hogy a buildMaxHeap(A) műveletigénye $\Theta(n)$, először belátjuk, hogy maximális műveletigényére $MT_b(n) \leq 2n + 1$ teljesül.

A kupaccá alakítás, a buildMaxHeap(A) $MT_b(n)$ maximális műveletigényét az eljárás meghívása, a ciklus $\lfloor \frac{n}{2} \rfloor$ iterációja, a lesüllyesztő eljárás $\lfloor \frac{n}{2} \rfloor$-szeri meghívása és benne a lesüllyesztő ciklus végrehajtásai adják. Tetszőleges d magasságú részfára a lesüllyesztő ciklus műveletigénye legfeljebb d. Az összes, n_d darab d magasságú részfákra tehát a lesüllyesztő ciklusok műveletigénye összesen legfeljebb $d * n_d$. Mivel a lesüllyesztő eljárás hívásai során $d \in 1 \,..\, h$, a kupaccá alakítás alatt a lesüllyesztő ciklusok műveletigényének összege legfeljebb $\sum_{d=1}^{h} d * n_d$. Innét

$$MT_b(n) \leq 1 + 2 \left\lfloor \frac{n}{2} \right\rfloor + \sum_{d=1}^{h} d * n_d$$

Vegyük most figyelembe, hogy $\sum_{d=1}^{h} d * n_d$ az alábbi alakba írható:

n_1+
$n_2 + n_2+$
$n_3 + n_3 + n_3+$
$\dots$
$n_h + n_h + n_h + \dots + n_h \quad (h$ tagú összeg)
Ezt oszloponként összeadva azt kapjuk, hogy

$$\sum_{d=1}^{h} d * n_d = \sum_{d=1}^{h} \sum_{m=d}^{h} n_m = \sum_{d=1}^{h} n_{\geq d} = \sum_{d=1}^{h} fele^d(n) \leq \sum_{d=1}^{h} \frac{n}{2^d} \leq n \sum_{d=1}^{\infty} \frac{1}{2^d} = n$$

Eredményeinket behelyettesítve kapjuk, hogy

$$MT_b(n) \leq 1 + \left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2} \right\rfloor \right) + \sum_{d=1}^{h} d * n_d \leq 1 + n + n = 2n + 1$$

Most belátjuk még, hogy $mT_b(n) \geq n$, amihez elég meggondolni, hogy a kupaccá alakításból a lesüllyesztések belső műveletigényét elhanyagolva, tehát csak a külső eljáráshívást, a ciklusiterációkat és a lesüllyeszt-hívásokat számolva $mT_b(n) \geq 1 + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2} \right\rfloor \geq n$. Innét $n \leq mT_b(n) \leq MT_b(n) \leq 2n + 1$, amiből

$$MT_b(n), mT_b(n) \in \Theta(n)$$

adódik.

10. Rendezések alsókorlát-elemzése (Lower bounds for sorting)

10.1. Tétel. *Tetszőleges rendező algoritmusra* $mT(n) \in \Omega(n)$.

Bizonyítás. Világos, hogy tetszőleges helyes rendező algoritmusnak mind az n rendezendő elem értékét ellenőriznie kell, és mindegyik alprogram hívás és ciklusiteráció is csak korlátos számú elemet ellenőriz (a belé ágyazott alprogram hívások és ciklusiterációk nélkül). Legyen ezeknek a korlátoknak a maximuma k. Akkor $mT(n) * k \geq n \Longrightarrow mT(n) \geq \frac{1}{k}n \Longrightarrow mT(n) \in \Omega(n)$. □

10.1. Összehasonlító rendezések és a döntési fa modell (Comparison sorts and the decision tree model)

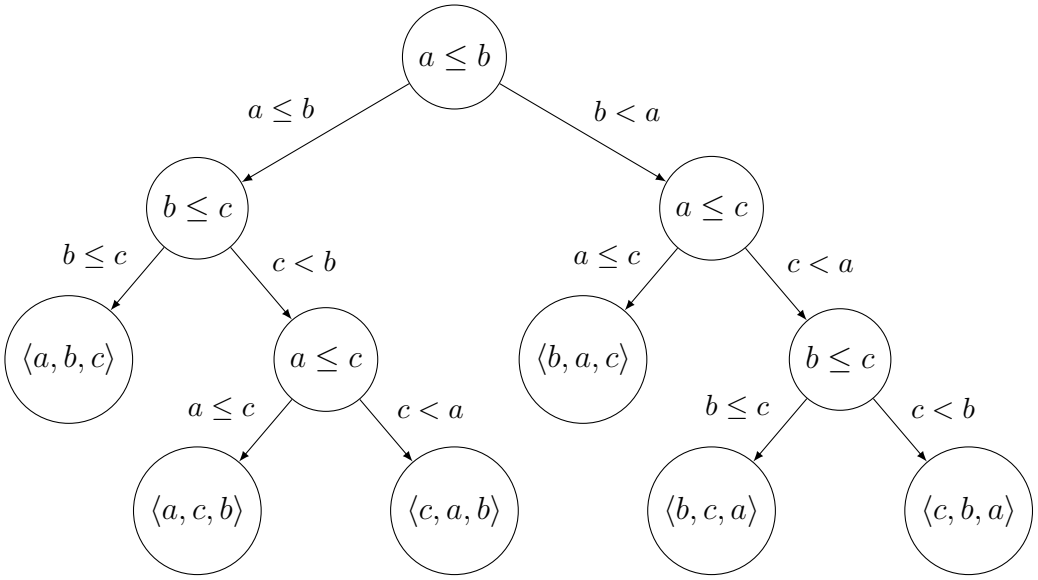
10.2. Definíció. *Tetszőleges rendező algoritmus akkor összehasonlító rendezés (comparison sort), ha az input elemeinek rendezéséhez csak az elemek kulcsainak összehasonlításából nyer információt. Ez azt jelenti, hogy ha adott az* $\langle a_1, a_2, \ldots a_n \rangle$ *input kulcssorozat, és ennek két kulcsát akarjuk összehasonlítani, akkor az* $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \neq a_j$, $a_i \geq a_j$, *vagy* $a_i > a_j$ *kulcs-összehasonlítások valamelyikét végezzük el. Nem vizsgáljuk meg a kulcsok belső szerkezetét, és más egyéb módon sem szerzünk róluk információt.*

Az eddig tárgyalt rendezési algoritmusok, tehát az *insertion sort, heap sort, merge sort* és a *quicksort* is összehasonlító rendezések.

A következő alfejezetben először a rendező algoritmus által elvégzett kulcs összehasonlítások maximális számára ($MC(n)$) adunk alsó becslést, majd a maximális műveletigényre ($MT(n)$). Ehhez az alsó becsléshez feltehető, hogy a rendezendő elemek kulcsai különböznek egymástól. Ezzel ugyanis a lehetséges inputok halmazát csökkentjük, és ha ezen a kisebb halmazon adunk alsó korlátot pl. $MC(n)$-re, akkor ez a teljes (bővebb) halmazon is alsó becslést ad $MC(n)$-re. (Hasonló mondható el $MT(n)$-ről is.) Mivel a rendezendő elemek kulcsai különböznek egymástól, az $a_i = a_j$ és az $a_i \neq a_j$ alakú összehasonlítások nem adnak információt a rendezéshez, és így csak az $a_i < a_j$, $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ alakú kulcs összehasonlítások maximális számára fogunk alsó becslést adni. (Ez nyilván az összes kulcsösszehasonlítás maximális számára is alsó becslés.)

Az összehasonlító rendezéseket ezután ún. döntési fákkal modellezzük. Ennek belső csúcsaiban vannak a kulcs összehasonlítások, leveleiben a rendezés eredményeként adódó lehetséges sorozatok. Az egyes részfákban a felette levő

kulcs összehasonlítások eredményeivel összhangban levő levelek vannak (feltéve, hogy a rendező algoritmus helyes). Az esetleges unáris belső csúcsokat törölhetjük: Ezek nem adnak információt a program további működéséhez. Ezért feltehető, hogy a döntési fa szigorúan bináris.



30. ábra. A beszúró rendezés (insertion sort) döntési fája az $\langle a, b, c \rangle$ input sorozatra. Tetszőleges 3 elemű input sorozatnak $3! = 6$ permutációja van, feltéve, hogy a sorozat elemei különbözők. Így a döntési fának is legalább $3! = 6$ levelének kell lennie.

A 30. ábrán a beszúró rendezés döntési fáját láthatjuk $n = 3$ esetben, azaz amikor a rendezendő sorozat 3 elemű.

Általánosságban tegyük föl, hogy $\langle a_1, a_2, \ldots a_n \rangle$ a rendezendő sorozat. A döntési fában mindegyik belső csúcsot egy kulcsösszehasonlítás címkézi, a leveleket pedig az $\langle a_1, a_2, \ldots a_n \rangle$ permutációi. Tetszőleges rendezés egy út megtételének felel meg a döntési fában a gyökértől valamelyik levélig, ahol a belső csúcsok felenek meg a közbenső kulcsösszehasonlításoknak és döntéseknek, a levelek pedig a rendezés lehetséges eredményeinek. Ha pl. egy belső csúcsot az $a_i \leq a_j$ összehasonlítás címkéz, a bal részfában a további összehasonlítások annak felelnek meg, hogy már tudjuk, hogy $a_i \leq a_j$, a jobb részfában pedig a további összehasonlítások annak felelnek meg, hogy már tudjuk, hogy $a_i > a_j$. Mire egy tetszőleges levélhez érünk, a rendezés meghatározza az $\langle a_1, a_2, \ldots a_n \rangle$ megfelelő sorba rendezését. Így $MC(n) =$ a leghosszabb körmentes út hosszával a döntési fában a gyökértől levélig, ami

viszont éppen a döntési fa h magassága, azaz $h = MC(n)$.

Mivel a rendezés inputja a rendezett sorozat tetszőleges permutációja lehet, bármelyik helyes rendező algoritmus képes kell legyen az input tetszőleges permutációját előállítani. Így az n elemű input sorozat mind az $n!$ permutációja meg kell jelenjen egy-egy levélen azaz a döntési fának legalább $n!$ levele van.

10.2. Alsó korlát a legrosszabb esetre (A lower bound for the worst case)

10.3. Tétel. *Bármely összehasonlító rendezés végrehajtásához a legrosszabb esetben $MC(n) \in \Omega(n \log n)$ kulcsösszehasonlítás szükséges.*

Bizonyítás. A fenti eszmefuttatás alapján elegendő alsó becslést adni a döntési fa $h = MC(n)$ magasságára. Jelölje a levelei számát l, akkor $n! \leq l$, továbbá $l \leq 2^h$, ugyanis egy h magasságú bináris fának legfeljebb 2^h levele van. Innét $n! \leq l \leq 2^h$, amiből $n! \leq 2^h$ adódik.

Következésképpen

$$MC(n) = h \geq \log n! = \sum_{i=1}^{n} \log i \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \log i \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \log \left\lceil \frac{n}{2} \right\rceil \geq \left\lceil \frac{n}{2} \right\rceil * \log \left\lceil \frac{n}{2} \right\rceil \geq$$

$$\geq \frac{n}{2} * \log \frac{n}{2} = \frac{n}{2} * (\log n - \log 2) = \frac{n}{2} * (\log n - 1) = \frac{n}{2} \log n - \frac{n}{2} \in \Omega(n \log n)$$

A $\sum_{i=\lceil \frac{n}{2} \rceil}^{n} \log \left\lceil \frac{n}{2} \right\rceil \geq \left\lceil \frac{n}{2} \right\rceil * \log \left\lceil \frac{n}{2} \right\rceil$ egyenlőtlenséghez elegendő belátni, hogy az összegnek legalább $\left\lceil \frac{n}{2} \right\rceil$ tagja van. A tagok száma nyilván $n - (\left\lceil \frac{n}{2} \right\rceil - 1) = (n - \left\lceil \frac{n}{2} \right\rceil) + 1 = \left\lfloor \frac{n}{2} \right\rfloor + 1$, ami, ha n páratlan szám, éppen $\left\lceil \frac{n}{2} \right\rceil$, ha pedig n páros, akkor $\left\lceil \frac{n}{2} \right\rceil + 1$. □

10.4. Tétel. *Tetszőleges összehasonlító rendezésre $MT(n) \in \Omega(n \log n)$.*

Bizonyítás. Mindegyik alprogram hívás és ciklusiteráció is csak korlátos számú kulcs összehasonlítást végez el (a belé ágyazott alprogram hívások és ciklusiterációk nélkül). Legyen ezeknek a korlátoknak a maximuma k.

Akkor $MT(n) * k \geq MC(n) \Longrightarrow MT(n) \geq \frac{1}{k} MC(n) \Longrightarrow MT(n) \in \Omega(MC(n))$. A 10.3. tétellel ($MC(n) \in \Omega(n \log n)$) és az „$\cdot \in \Omega(\cdot)$” reláció tranzitivitásával azonnal adódik ez a tétel ($MT(n) \in \Omega(n \log n)$). □

Vegyük észre, hogy a *heap sort* és a *merge sort*, *aszimptotikusan optimálisak* abban az értelemben, hogy a műveletigényük $O(n \log n)$ felső korlátja összeillik a (legrosszabb esetre vonatkozó) $MT(n) \in \Omega(n \log n)$ alsó korláttal. (Ld. a 10.4. tételt!) Az előbbi tulajdonságokból azonnal következik, hogy mindkettőre $MT(n) \in \Theta(n \log n)$.

Ld. még:
http://people.inf.elte.hu/fekete/algoritmusok_jegyzet/
. 19_fejezet_Rendezesek_alsokorlat_elemzese.pdf

11. Rendezés lineáris időben (Sorting in linear time)

Alapvetően nem kulcsösszehasonlítással rendezünk (10.2. definíció), így ezekre az algoritmusokra nem vonatkozik az összehasonlító rendezések alaptétele (10.4. tétel).

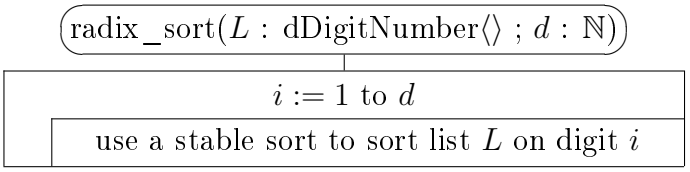
Beláttuk, hogy tetszőleges összehasonlító rendezésre (comparison sort algorithm) $MT(n) \in \Omega(n \log n)$. Ha tehát aszimptotikusan jobb rendezéseket keresünk, olyan algoritmusokra van szükségünk, amelyek a kulcsok összehasonlítása helyett (vagy mellett) másképpen (is) nyernek információt a kulcsok nagyság szerinti sorrendjére vonatkozóan.

Az alábbiakban ismertetendő radix sort (számjegypozíciós rendezés), szétválogató rendezés (distributing sort) és counting sort (leszámláló rendezés) a kulcsok összehasonlítása helyett osztályozzák a kulcsokat. A bucket sort (egyszerű edényrendezés) emellett még kulcsösszehasonlító segédrendezést is használ.

Mivel a 10.1. tétel szerint tetszőleges rendező algoritmusra $mT(n) \in \Omega(n)$, egy S rendezés aszimptotikusan optimális, ha $MT_S(n) \in O(n)$. (Ebben az esetben tehát $MT_S(n), mT_S(n) \in \Theta(n)$.) Az ebben a fejezetben tárgyalt radix sort és counting sort aszimptotikusan optimálisak. A bucket sort esetében viszont csak a várható (azaz „átlagos") és a minimális műveletigény lineáris.

11.1. Radix rendezés

A Radix rendezés (Radix Sort = Számjegypozíciós rendezés) általában felteszi, hogy a kulcsok r alapú számrendszerben felírt, d számjegyű, előjel nélküli, azaz dDigitNumber típusú egész számok. (Szükség esetén vezető nullákat írunk a számok elé, hogy pontosan d számjegyük legyen.) A számok számjegyeit jobbról balra sorszámozzuk. Az első számjegy a legkevésbé szignifikáns, a jobb szélső számjegy, és így tovább, a d-edik a legszignifikánsabb, a bal szélső számjegy. A rendezésnek d menete van. Az első menetben az első (a jobb szélső, a legkevésbé szignifikáns) számjegy szerint rendezünk stabil rendezéssel, az i-edik menetben a jobbról i-edik számjegy szerint, és az utolsó menetben a jobbról d-edik (a bal szélső, a legszignifikánsabb) számjegy szerint, mindig stabil rendezéssel, az alábbi séma szerint (ahol L absztrakt lista, vagy más néven véges sorozat: dDigitNumber$\langle\rangle$ olyan véges sorozatokat jelöl, amiknek dDigitNumber az elemtípusa).



Az első menet (first pass) után tehát számok a legkevésbé szignifikáns (jobbról első, azaz jobbszélső) számjegyük szerint rendezettek. Mikor (a következő, második menetben) a jobbról második számjegyük szerint rendezünk, a rendezés stabilitása miatt az azonos (jobbról) második számjegyűek a (jobbról) első számjegyük szerint rendezve maradnak, hiszen a stabil rendezések az azonos kulcsú elemeket meghagyják a bemenet (azaz az input) sorrendjében. Így a 2. menet után a számok már a két jobbszélső számjegyük szerint lesznek rendezve. Mikor a 3. menetben jobbról a 3. számjegy szerint rendezünk, akkor az azonos 3. számjegyűek már a jobbról első két számjegyük szerint maradnak rendezve. Így a 3. menet után a számok már a 3 jobbszélső számjegyük szerint lesznek rendezve. Így tovább, amikor a d-edik menetben jobbról a d-edik számjegy szerint rendezünk, akkor az azonos d-edik számjegyűek már a jobbról első $d-1$ számjegyük szerint maradnak rendezve. Így a d-edik menet után a számok már a d jobbszélső számjegyük szerint lesznek rendezve. Mivel összesen d számjegyük van, a számok ekkor már teljesen rendezve lesznek.

Annak érdekében, hogy a Radix rendezés helyesen működjön, szükséges, hogy a számjegyek szerinti rendezések stabilak legyenek. A hatékonyság miatt pedig az is fontos, hogy lineáris időben rendezzenek.

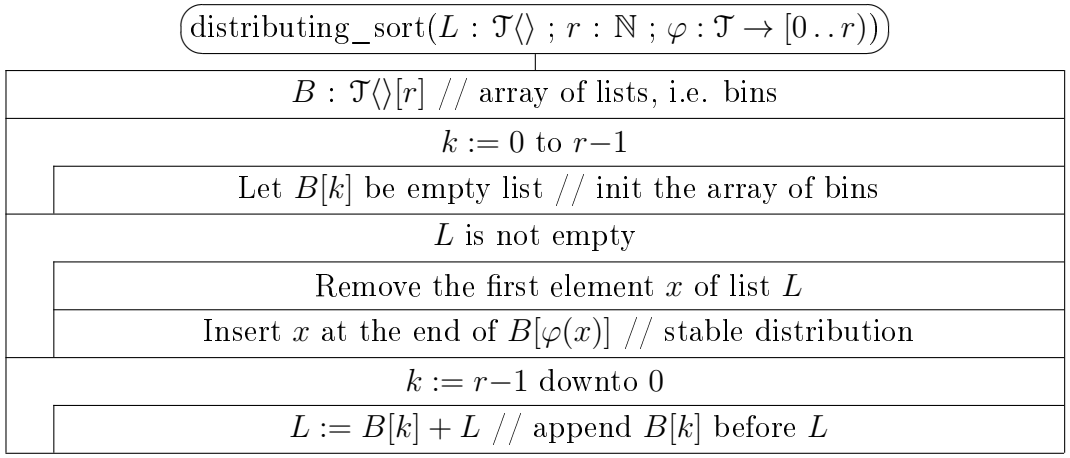
A *szétválogató rendezés* (distributing sort) megfelelő láncolt listák adott számjegy szerinti rendezésére, míg a *leszámláló rendezés* (counting sort) tömbök esetén. Mindkettő stabil, és lineáris a műveletigényük.

11.2. Szétválogató rendezés (distributing sort)

A szétválogató rendezés stabil, és láncolt listákra optimális, azaz lineáris műveletigényű implementációja könnyen megvalósítható, így a Radix rendezés egy hatékony verzióját építhetjük rá.

Az alábbi tárgyalás egy kicsit általánosabb, mint amit a radix sort igényel. Amikor a radix sort segédeljárásaként a szétválogató rendezést használjuk, ez utóbbi φ kulcsfüggvénye segítségével a megfelelő számjegyet kell kiválasztanunk.

A rendezési probléma: Adott az n hosszúságú, $\mathcal{T}$ elemtípusú L absztrakt lista, $r \in O(n)$ pozitív egész szám, és a $\varphi : \mathcal{T} \to [0\,..\,r)$ kulcskiválasztó függvény. Rendezzük L-et stabil rendezéssel, lineáris műveletigénnyel!



A szétválogató rendezés hatékonysága: Az első és az utolsó ciklus r-szer iterál, a középső pedig n-szer. Amennyiben a lista végéhez fűzés és a konkatenálás $\Theta(1)$ időben megoldható, a szétválogató rendezés műveletigénye nyilván $\Theta(n + r) = \Theta(n)$ lesz, az $r \in O(n)$ (természetes) feltétel miatt.

11.3. Radix rendezés listákra

A radix rendezés által felhasznált (stabil) szétválogató rendezés, r alapú számrendszer esetén, a számokat – az i-edik számjegyük értéke szerint – sorban szétválogatja a $B_0, B_1, B_2, \ldots B_{r-1}$ – kezdetben üres – polcokra (bins), ügyelve arra, hogy az egyes polcokon megmaradjon a számoknak a szétválogatás előtti sorrendje. Ezután a polcok tartalmát – a polcok egymás közötti és a számoknak a polcokon belüli sorrendjét is megtartva – összefűzi L-be.

Mivel r darab polcot kell üresre inicializálni, n elemet szétválogatni, majd az r db polcot, valójában listát összefűzni, ez megoldható $\Theta(n + r)$ műveletigénnyel. A radix rendezés pedig ezt a – nyilvánvalóan stabil – rendezést d-szer hívja meg, így a teljes műveletigénye $\Theta(d * (n + r))$. Ha d konstans, és $r \in O(n)$, akkor ebből

$$T_{\text{radix\_sort}}(n) \in \Theta(n).$$

A 31. ábrán látható példában $r = 4$ és $d = 3$, azaz a kulcsok 4-es számrendszerbeli, 3 jegyű számok, és a B_0, B_1, B_2, B_3 polcokat használjuk.[21]

[21]Sok ember számára az lenne természetes, hogy a számokat először a balszélső és utoljára a jobbszélső számjegyük szerint rendezze. A radix rendezésben azonban a későbbi menetek fontosabbak az eredmény szempontjából, mint a korábbiak: Az i-edik menetben

Az input (azaz bemeneti) lista, szimbolikus jelöléssel ($r = 4; d = 3$):
$L = \langle 103, 232, 111, 013, 211, 002, 012 \rangle$

1. menet (a számok jobbról 1., azaz jobbszélső számjegyei szerint):
$B_0 = \langle \rangle$
$B_1 = \langle 111, 211 \rangle$
$B_2 = \langle 232, 002, 012 \rangle$
$B_3 = \langle 103, 013 \rangle$
$L = \langle 111, 211, 232, 002, 012, 103, 013 \rangle$

2. menet (a számok jobbról 2., azaz középső számjegyei szerint):
$B_0 = \langle 002, 103 \rangle$
$B_1 = \langle 111, 211, 012, 013 \rangle$
$B_2 = \langle \rangle$
$B_3 = \langle 232 \rangle$
$L = \langle 002, 103, 111, 211, 012, 013, 232 \rangle$

3. menet (a számok jobbról 3., azaz balszélső számjegyei szerint):
$B_0 = \langle 002, 012, 013 \rangle$
$B_1 = \langle 103, 111 \rangle$
$B_2 = \langle 211, 232 \rangle$
$B_3 = \langle \rangle$
$L = \langle 002, 012, 013, 103, 111, 211, 232 \rangle$

31. ábra. A szétválogató-összefűző Radix rendezés bemutatása

Természetes módon adódik, hogy a gyakorlatban a fenti absztrakt listák (más néven véges sorozatok), – azaz a bemenet, a polcok és az algoritmus kimenete is – láncolt listák legyenek, mivel nem tudhatjuk, hogy az egyes polcokon $[0\,..\,n]$ között (ahol n az input mérete) hány tételt akarunk elhelyezni, r darab n méretű segédtömb pedig a gyakorlatban túl sok munkamemóriát jelentene. Ha viszont a polcokat láncolt listák reprezentálják, akkor a memória allokálások (pl. **new** utasítás) és deallokálások (pl. **delete** utasítás) elkerülése végett célszerű, ha a bemenet és a kimenet is – a polcokkal azonos típusú – láncolt lista. (Ha az interfész egy tömb, de a polcok láncolt listák, összesen $n * d$ memória allokálás, és ugyanennyi deallokálás szükséges, ami várhatólag annyira megnöveli a $\Theta(n)$ műveletigényben rejtett „konstanst", hogy a rendezés gyakorlatilag használhatatlanná válik.)

Ha a polcok láncolt listák, akkor ezen listák végéhez is direkt hozzáférés szükséges, hogy az egyes menetek bemeneti listája elemeinek szétpakolása hatékony legyen, azaz minden elemre $\Theta(1)$ idő alatt megtörténjen. Ez megvalósítható pl. S1L-ekkel és a nemüres „polcok" végére mutató pointerekkel, illetve (ha lusták vagyunk, némi konstans szorzó árán) C2L-ek alkalmazásával.

Néha használjuk a radix rendezést összetett kulcsú rekordok rendezésére. Ilyen összetett kulcs például a dátum, ami három komponenst (*év, hó, nap*) tartalmaz. Először tehát a legkevésbé szignifikáns *nap*, majd a *hó*, végül a legszignifikánsabb *év* mező szerint rendezünk, stabil rendezéssel. (Természetesen használhatnánk összehasonlító rendezést is, amikor két kulcs [azaz dátum] összehasonlításánál először az *év* mezőket vennénk figyelembe, ha ezek egyenlők, a *hó* mezőket, és ha ezek is egyenlők, a *nap* mezőket. Tekintetbe véve azonban, hogy a radix rendezéshez itt elég 3 menet, és feltételezve, hogy nincs túl sok lehetséges évszám, a fenti szétválogató-összefűző stratégiával valószínűleg már néhány ezer tétel esetén is gyorsabb rendezést kapunk, mint bármelyik összehasonlító rendezéssel.)

11.1. Példa. *Tegyük fel, hogy adott az L fejelemes C2L, a listaelemek kulcsai r alapú számrendszerben felírt, d számjegyű számok, és adott a digit(i, r, x) függvény, ami $\Theta(1)$ műveletigénnyel ki tudja nyerni az x kulcs jobbról i-edik számjegyét, ahol a számjegyeket* 1*-től d-ig sorszámozzuk.*

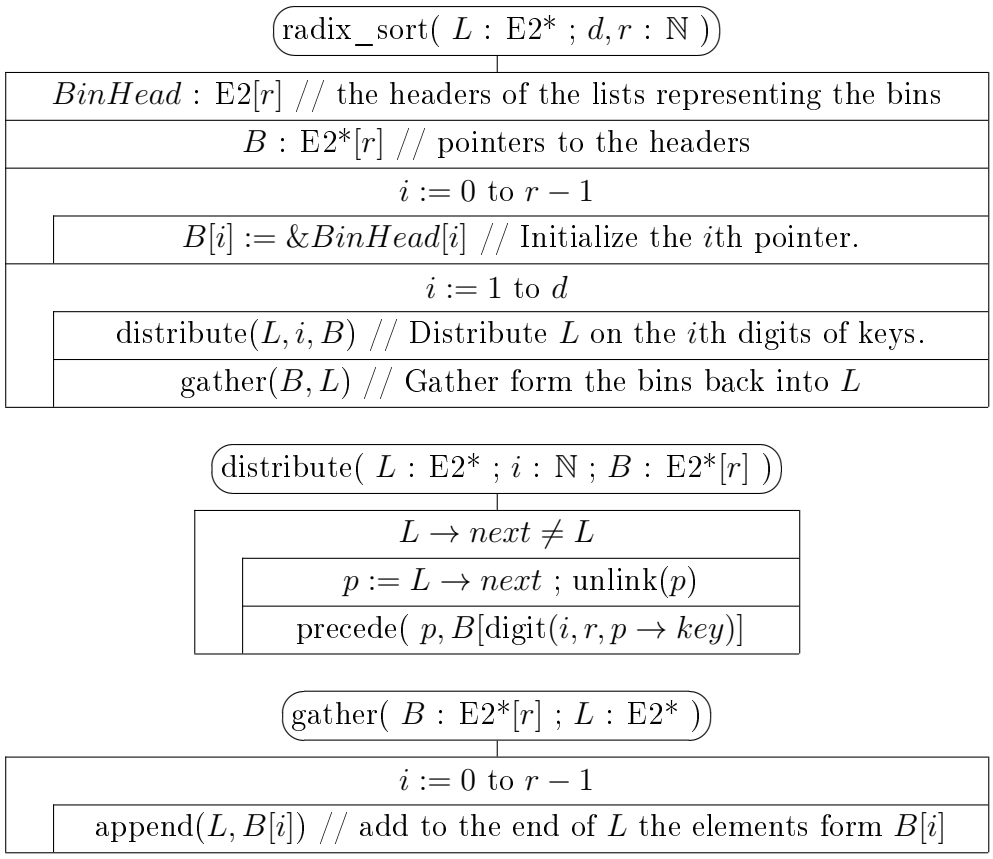
---

az első $i-1$ menet eredménye alárendelődik az i-edik menetnek. Így a végén a számok elsősorban a jobbszélső számjegyük szerint lennének rendezve, és általában nem ezt akarjuk.

Egy másik megközelítés szerint szintén a balszélső számjeggyel kezdhetnénk, de utána, még az összefűzés előtt, rekurzívan rendezhetnénk a polcokat a többi számjegy szerint, minden rekurziós szinten újra és újra, újabb és újabb segédpolcokat létrehozva, majd törölve. Így viszont a polcokkal kapcsolatos rengeteg adminisztráció lelassítaná a programot.

Adja meg a radix rendezés struktogramját a fenti feltételekkel, Theta(n) műveletigénnyel, ahol n a lista hossza, $r \in O(n)$, d pedig pozitív egész konstans.

Megoldás:



(Az append(L, H) eljárást a 7.16. példánál dolgoztuk ki.)
$T_{\text{append}} \in \Theta(1)$, így $T_{\text{gather}}(r) \in \Theta(r)$.
$T_{\text{distribute}}(n) \in \Theta(n)$, ahol $n = |L|$.
Ezért $T_{\text{radix\_sort}}(n, d, r) \in \Theta(r + d(n + r))$.
Következésképp, ha d konstans és $r \in O(n)$, akkor $T_{\text{radix\_sort}}(n) \in \Theta(n)$.

11.2. Feladat. *Oldjuk meg a 11.1. példát azzal a változtatással, hogy L S1L!*

11.3. Feladat. *Oldjuk meg a 11.1. példát és a 11.2. feladatot azzal a változtatással, hogy az L kulcsai 4 bájtos, előjel nélküli egész számok, $r = 256$ és nem áll rendelkezésünkre a számjegyeket kinyerő függvény, a C-ben szokásos aritmetikai léptetések és a bitenkénti & művelet viszont adott!*

11.4. Leszámláló rendezés (Counting sort)

Míg a radix sort fentebb ismertetett verziója (a számjegyek szerinti szétválogatásokkal és az összefűzésekkel) láncolt listák rendezésére alkalmazható hatékonyan, a leszámláló rendezés a radix sort ideális segédrendezése, ha a rendezendő adatokat egy tömb tartalmazza.

Emlékeztetünk arra, hogy egy rendezés akkor *stabil*, ha megtartja az egyenlő kulcsú elemek eredeti sorrendjét. A leszámláló rendezés stabil, és műveletigénye miatt is megfelelő, mint a radix sort segédrendezése.

Az alábbiakban a leszámláló rendezést egy kicsit általánosabban tárgyaljuk, mint ahogy azt a számjegypozíciós rendezéshez szükséges. Ha a radix sort-hoz használjuk, feltehető, hogy a counting sort φ függvénye a megfelelő számjegyet választja ki.

A rendezési feladat: Adott az $A{:}\mathcal{T}[n]$ tömb, $r \in O(n)$ pozitív egész, $\varphi : \mathcal{T} \to [0\,..\,r)$ kulcsfüggvény. Rendezzük az A tömböt lineáris időben stabil rendezéssel úgy, hogy az eredmény a $B{:}\mathcal{T}[n]$ tömbben keletkezzék!

counting_sort($A, B : \mathcal{T}[n]$; $r : \mathbb{N}$; $\varphi : \mathcal{T} \to [0\,..\,r)$)

- $C : \mathbb{N}[r]$ // counter array
- $k := 0$ to $r-1$
  - $C[k] := 0$ // init the counter array
- $i := 0$ to $n-1$
  - $C[\varphi(A[i])]{+}{+}$ // count the items with the given key
- $k := 1$ to $r-1$
  - $C[k]$ += $C[k-1]$ // $C[k] :=$ the number of items with key $\leq k$
- $i := n-1$ downto 0
  - $k := \varphi(A[i])$ // $k :=$ the key of $A[i]$
  - $C[k]--$ // The next one with key k must be put before $A[i]$ where
  - $B[C[k]] := A[i]$ //Let $A[i]$ be the last of the {unprocessed items with key k}

A fenti struktogram első ciklusában kinullázzuk a C számláló tömböt.

A második ciklusban minden lehetséges k kulcsra $C[k]$-ban megszámoljuk, hogy hány db k kulcsú elem van.

A harmadik ciklusban minden lehetséges k kulcsra összeadjuk $C[k]$-ban, hogy hány $\leq k$ kulcsú elem van. Mivel ≤ 0 kulcsú elem ugyanannyi van, mint 0 kulcsú, $C[0]$ értéke nem változik. Nagyobb k kulcsokra viszont annyi

$\leq k$ kulcsú elem van, amennyi pontosan k kulcsú + k-nál kisebb kulcsú (vagyis $\leq k-1$ kulcsú). $C[k]$ új értékét tehát úgy kaphatjuk meg, ha $C[k]$ régi értékéhez hozzáadjuk $C[k-1]$ új értékét.

A negyedik ciklusban a bemeneti tömbön visszafelé haladva minden elemet az eredmény tömbbe a helyére teszünk, vagyis tetszőleges k kulcsra először az utolsó k kulcsú elemet dolgozzuk fel, és betesszük az utolsó helyre, ahova k kulcsú elem kerülhet, pontosan az eredmény tömb $C[k]$-adik, vagyis a $C[k]-1$ indexű elemébe. $C[k]$ mutatja meg ugyanis, hogy hány darab legfeljebb k kulcsú elem van a bemeneten. Ehhez először a $C[k]$-t eggyel csökkentjük, az aktuális elemet $B[C[k]]$-ba másoljuk. Így a fordított bejárás szerint következő k kulcsú elem közvetlenül a mostani elé fog kerülni stb. Emiatt tehát tetszőleges k kulcsra a k kulcsú elemek az eredmény tömbben is az eredeti sorrendjükben maradnak, és a kisebb kulcsú elemek meg is előzik a nagyobb kulcsúakat, azaz *stabil rendezést* kapunk.

A műveletigény nyilván $\Theta(n+r)$. Feltételezve, hogy $r \in O(n)$, $\Theta(n+r) = \Theta(n)$, azaz $T(n) \in \Theta(n)$.

A leszámláló rendezés szemléltetése: Feltesszük, hogy kétjegyű, négyes számrendszerbeli számokat kell rendezni a jobb oldali számjegyük, mint kulcs szerint, azaz a φ kulcsfüggvény a jobb oldali számjegyet választja ki.

A bemenet:

	0	1	2	3	4	5
$A:$	02	32	30	13	10	12

A C:$\mathbb{N}[4]$ számláló tömb alakulása [ahol az első oszlopban – a struktogram első ciklusának megfelelően – kinullázzuk a C számláló tömböt; a következő hat oszlopban – a struktogram második ciklusának megfelelően – minden lehetséges k kulcsra (itt számjegyre) megszámoljuk, hogy hány k kulcsú elem van; a $\sum$ jelzésű oszlopban – a struktogram harmadik ciklusának megfelelően – minden lehetséges k kulcsra (itt számjegyre) összeadjuk, hogy hány $\leq k$ kulcsú elem van; az utolsó hat oszlopban pedig – a struktogram negyedik ciklusának megfelelően – a bemeneti tömbön visszafelé haladva minden elemet a helyére teszünk, ahogy azt fentebb részleteztük]:

	C	02	32	30	13	10	12	$\sum$	12	10	13	30	32	02
0	0			1		2		2		1		0		
1	0							2						
2	0	1	2				3	5	4				3	2
3	0				1			6			5			

A kimenet :

	0	1	2	3	4	5
$B:$	30	10	02	32	12	13

Most pedig feltesszük, hogy az előző leszámláló rendezés eredményeként adódott kétjegyű, négyes számrendszerbeli számok sorozatát kell rendezni a bal oldali számjegyük, mint kulcs szerint, azaz a φ kulcsfüggvény a bal oldali számjegyet választja ki.

A bemenet:

	0	1	2	3	4	5
$B:$	30	10	02	32	12	13

A C:$\mathbb{N}[4]$ számláló tömb alakulása:

	C	30	10	02	32	12	13	$\sum$	13	12	32	02	10	30
0	0			1				1				0		
1	0		1			2	3	4	3	2			1	
2	0							4						
3	0	1			2			6			5			4

A kimenet :

	0	1	2	3	4	5
$A:$	02	10	12	13	30	32

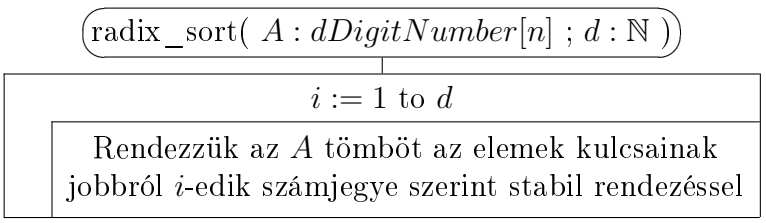
Mivel az első leszámláló rendezés a bemenetet a jobb oldali számjegyek szerint rendezte, és a második leszámláló rendezés az első eredményét rendezte tovább a bal oldali számjegyek szerint stabil rendezéssel, a végeredményben az azonos bal-számjegyű számok a jobb-számjegyük szerint rendezve maradtak, és így a végeredményben a számok már mindkét számjegyük szerint rendezettek.

Ezért tehát a fent szemléltetett két *counting sort* egy *radix rendezés* 1. és 2. menetét adja, és mivel a számainknak most csak két számjegyük van, egy teljes radix rendezést hajtottunk végre.

11.5. Radix rendezés (Radix-Sort) tömbökre ([4] 8.3)

A rendezendő A tömb kulcsai r alapú számrendszerben felírt, d-jegyű nemnegatív egész számok. A jobbról első számjegy helyiértéke a legkisebb, míg a d-ediké a legmagasabb.[22]

[22] Általánosítva, a kulcs felbontható d kulcs direkt szorzatára, ahol a jobbról első legkevésbé szignifikáns, míg a d-edik a leglényegesebb. Pl. ha a kulcs dátum, akkor először a napok, majd a hónapok és végül az évek szerint alkalmazunk stabil rendezést. Ez azért jó, mert – a stabilitás miatt – amikor a hónapok szerint rendezünk, az azonos hónapba eső elemek a napok szerint rendezettek maradnak, és amikor az évek szerint rendezünk, az azonos évbe eső elemek hónapok és ezen belül napok szerint szintén sorban maradnak.



Ha a stabil rendezés a leszámláló rendezés, akkor a műveletigény $\Theta(d(n+r))$, mivel a teljes rendezés d leszámláló rendezésből áll. Feltételezve, hogy d konstans és $r \in O(n)$, $\Theta(d(n+r)) = \Theta(n)$, azaz $T(n) \in \Theta(n)$.

Ha pl. a kulcsok négy bájtos nemnegatív egészek, választhatjuk számjegyeknek a számok bájtjait, így $d = 4$ és $r = 256$, mindkettő n-től független konstans, tehát a feltételek teljesülnek, és a rendezés lineáris időben lefut. Az i-edik számjegy, azaz bájt kinyerése egyszerű és hatékony. Ha *key* a kulcs, akkor pl. C++-ban

$$(key >> (8 * (i-1)))\&255$$

az i-edik számjegye[23]. Ld. még [4] 8.3-ban, hogy n rendezendő adat, b bites természetes szám kulcsok és m bites „számjegyek" ($r = 2^m$) esetén, a radix rendezésben, b és n függvényében, hogyan érdemes m-et megválasztani!

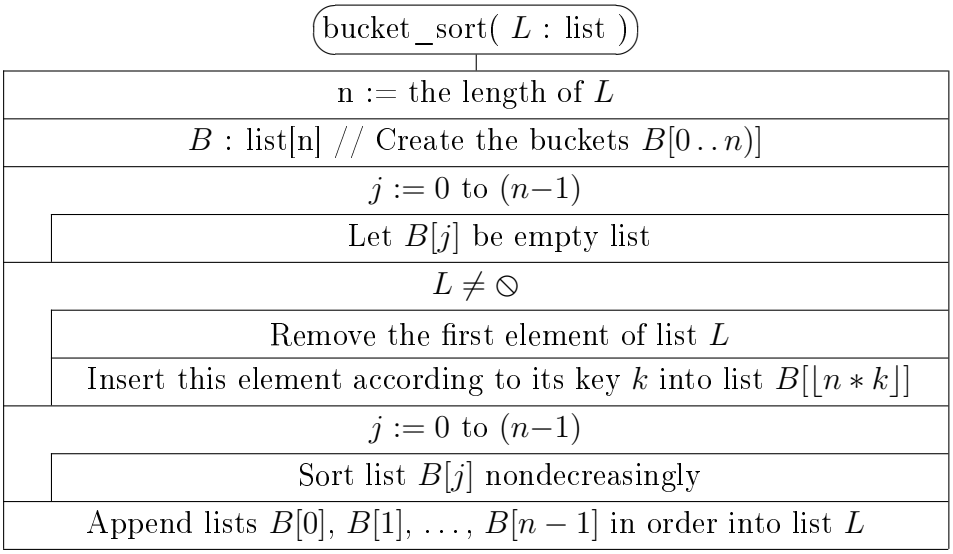
11.4. Feladat. *Részletezzük a* radix-sort *fenti, absztrakt programját úgy, hogy a stabil rendezéshez leszámláló rendezést használunk, és a „számjegyek" a teljes b bites kulcs m bites szakaszai! Vegyük figyelembe, hogy ettől a counting sort paraméterezése is változik, és hogy érdemes felváltva hol az eredeti $A[0\,..\,n)$ tömbből a $B[0\,..\,n)$ segédtömbbe, hol a $B[0\,..\,n)$-ből az $A[0\,..\,n)$-be végezni a leszámláló rendezést!*

11.6. Egyszerű edényrendezés (bucket sort)

Feltesszük, hogy a rendezendő elemek kulcsai a $[0;1)$ valós intervallum elemei. (Ha a kulcs egész vagy valós szám típusú, vagy azzá konvertálható, továbbá tudunk a kulcsok számértékeire alsó és felső határt mondani, akkor a kulcsok számértékei nyilván normálhatók a $[0;1)$ valós intervallumra. Ha ugyanis $a < b$ valós számok, és a k kulcsra $k \in [a;b)$, akkor $\frac{k-a}{b-a} \in [0;1)$.)

Az alábbi algoritmus akkor lesz hatékony, ha az input kulcsai a $[0;1)$ valós intervallumon egyenletesen oszlanak el. (Az L [absztrakt] listát (más néven véges sorozatot) – mint mindig – most is monoton növekvően rendezzük. Az edények [buckets] rendezésére valamilyen korábbról ismert rendezést használunk.)

[23] A fenti C++-os képlet tovább egyszerűsödik, ha a programban i helyett az eltolás mértékét tartjuk nyilván (ez persze egyenlő $8 * (i-1)$-gyel).



Nyilván $mT(n) \in \Theta(n)$, és a fenti egyenletes eloszlást feltételezve $AT(n) \in \Theta(n)$ is teljesül. $MT(n)$ pedig attól függ, hogy a $B[j]$ listákat milyen módszerrel rendezzük. Pl. egyszerű beszúró rendezést használva $MT(n) \in \Theta(n^2)$, összefésülő rendezéssel viszont $MT(n) \in \Theta(n \log n)$.

11.5. Feladat. *Részletezze az elemi utasítások szintjéig a fenti kódot, feltéve, hogy a listák egyszerű láncolt listák (S1L)! Vegyük észre, hogy az edénybe (bucket) való beszúrásnál – ha nem törekszünk a rendezés stabilitására – az edényt reprezentáló S1L elejére érdemes a listaelemet beszúrni. (Az edények rendezését nem kell részletezni.)*

11.6. Feladat. *Tegye a 11.5. feladatot megoldó rendezést stabillá,* $MT(n) \in \Theta(n \log n)$*;* $mT(n), AT(n) \in \Theta(n)$ *műveletigénnyel!*

12. Hasító táblák (hash tables)

A mindennapi programozási gyakorlatban sokszor van szükségünk ún. szótárakra (dictionaries), amelyek műveletei: (1) adat beszúrása a szótárba, (2) kulcs alapján a szótárban a hozzá tartozó adat megkeresése, (3) a szótárból adott kulcsú, vagy egy korábbi keresés által lokalizált adat eltávolítása.

Az AVL fák, B+ fák (ld. a következő félévben) és egyéb kiegyensúlyozott keresőfák mellett a szótárakat gyakran hasító táblákkal valósítják meg, feltéve, hogy a műveleteknek nem a maximális, hanem az átlagos futási idejét szeretnék minimalizálni. (A kiegyensúlyozott keresőfák esetében ti. elsősorban a maximális műveletigényt optimalizáljuk: beszúrás, keresés és törlés esetén is elvárt az $O(\log n)$ maximális műveletigény.) Hasító táblát használva ugyanis a fenti műveletekre elérhető az ideális, $\Theta(1)$ átlagos futási idő azon az áron, hogy a maximális műveletigény általában $\Theta(n)$.

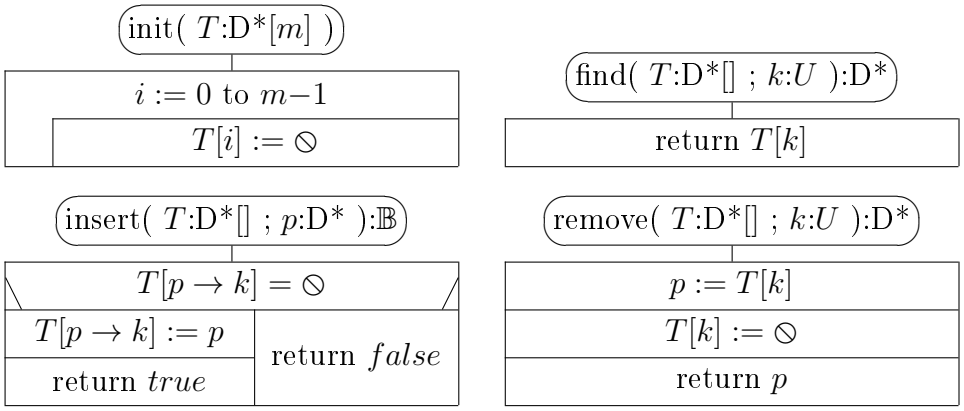
Jelölések:
m : a hasító tábla mérete
$T[0\,..\,m)$: a hasító tábla
$T[0], T[1], \ldots, T[m-1]$: a hasító tábla rései (slot-jai)
$\oslash$: üres rés a hasító táblában (direkt címzésnél és a kulcsütközések láncolással való feloldása esetén)
E : üres rés (empty slot) kulcsa a hasító táblában (nyílt címzésnél)
D : törölt rés (deleted slot) kulcsa a hasító táblában (nyílt címzésnél)
n : a hasító táblában tárolt adatok száma
$\alpha = n/m$: a hasító tábla kitöltöttségi aránya (load factor)
U : a kulcsok univerzuma; $k, k', k_i \in U$
$h : U \to 0\,..\,(m-1)$: hasító függvény (hash function)

Feltesszük, hogy a hasító tábla nem tartalmazhat két vagy több azonos kulcsú elemet, és hogy $h(k)$, $\Theta(1)$ időben számolható.

12.1. Direkt címzés (direct-address tables)

Feltesszük, hogy $U = [0\,..\,m)$, ahol $m \geq n$, de m nem túl nagy.
A $T : D*[m]$ hasító tábla rései pointerek, amik D típusú adatrekordokra mutatnak. A rekordoknak van egy $k : U$ kulcsmezőjük és járulékos mezőik. A hasító táblát $\oslash$ pointerekkel inicializáljuk.

D
$+\ k : U$ // k is the key
$+\ \ldots$ // satellite data



Nyilván $T_{\text{init}}(m) \in \Theta(m)$. A másik három művelet futási ideje pedig $\Theta(1)$.

12.2. Hasító táblák (hash tables)

Hasító függvény (hash function): Ha $|U| >> n$, a direkt címzés nem alkalmazható, vagy nem gazdaságos, ezért $h : U \to 0\,..\,(m-1)$ hasító függvényt alkalmazunk, ahol tipikusan $|U| >> m$ (a kulcsok U „univerzumának" elemszáma sokkal nagyobb, mint a hasító tábla m mérete). A k kulcsú adatot a $T[0\,..\,m)$ hasító tábla $T[h(k)]$ résében tároljuk (próbáljuk tárolni).

Feltesszük, hogy a hasító táblában minden rekord kulcsa egyedi, azaz két tetszőleges rekord kulcsa különböző.

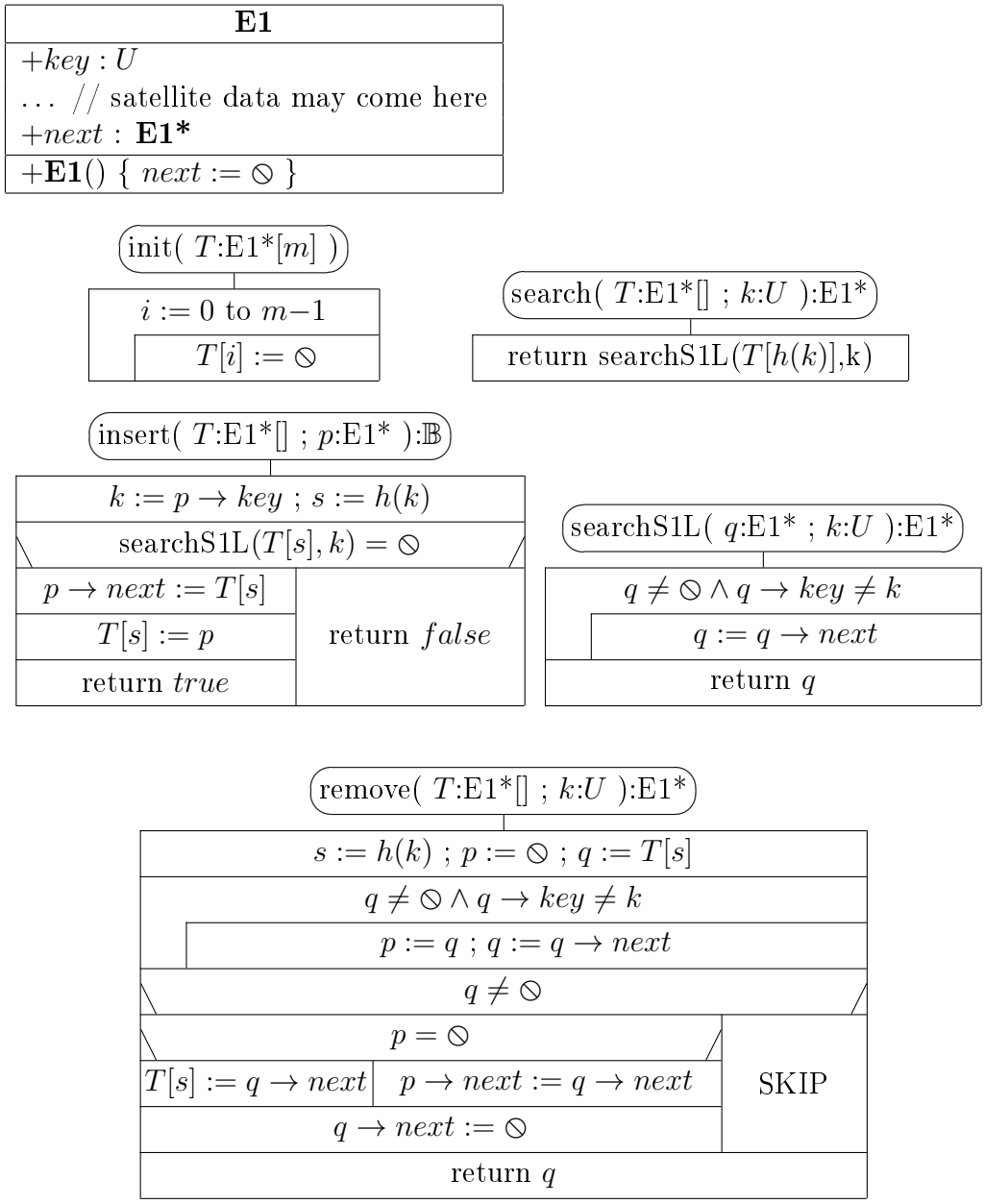
A $h : U \to [0\,..\,m)$ függvény *egyszerű egyenletes hasítás (simple uniform hashing)*, ha a kulcsokat a rések között egyenletesen szórja szét, azaz hozzávetőleg ugyanannyi kulcsot képez le az m rés mindegyikére. Tetszőleges hasító függvénnyel kapcsolatos elvárás, hogy egyszerű egyenletes hasítás legyen.

Kulcsütközések (key collisions): Ha két adat $k_1 \neq k_2$ kulcsára $h(k_1) = h(k_2)$, kulcsütközésről beszélünk. Mivel $|U| >> m$, kulcsütközés szinte biztosan előfordul, ezért kezelni kell.

Ha például a kulcsok egész számok, és $h(k) = k \bmod m$, akkor pontosan azok a kulcsok képeződnek le az s-edik résre, amelyekre $s = k \bmod m$.

12.3. Kulcsütközések feloldása láncolással (collision resolution by chaining)

Feltesszük, hogy a hasító tábla rései egyszerű láncolt listákat (S1L) azonosítanak, azaz T:E1*[m], ahol a listaelemekben a szokásos *key* és a *next* mezőkön kívül általában járulékos mezők (satellite data) is vannak. Ha a hasító függvény két vagy több elem kulcsait a hasító táblának ugyanarra a résére képzi le, akkor ezeket az elemeket az ehhez a réshez tartozó listában tároljuk.



Nyilván $T_{\text{init}}(m) \in \Theta(m)$. A másik három műveletre pedig $mT \in \Theta(1)$, $MT(n) \in \Theta(n)$, $AT(n,m) \in \Theta(1+\frac{n}{m})$.

$AT(n,m) \in \Theta(1+\frac{n}{m})$ feltétele, hogy a $h : U \to 0\,..\,(m-1)$ függvény *egyszerű egyenletes hasítás* legyen, és még az indokolja, hogy a résekhez tartozó listák átlagos hossza $= \frac{n}{m} = \alpha$.

Általában feltesszük még, hogy $\frac{n}{m} \in O(1)$. Ebben az esetben nyilván $AT(n,m) \in \Theta(1)$ is teljesül.

12.4. Jó hasító függvények (good hash functions)

Osztó módszer (division method): Ha a kulcsok egész számok, gyakran választják a

$$h(k) = k \bmod m$$

hasító függvényt, ami gyorsan és egyszerűen számolható, és ha m olyan prím, amely nincs közel a kettő hatványokhoz, általában egyenletesen szórja szét a kulcsokat a $0\,..\,(m-1)$ intervallumon.

Ha pl. a kulcsütközést láncolással szeretnénk feloldani, és kb. 2000 rekordot szeretnénk tárolni $\alpha \approx 3$ kitöltöttségi aránnyal, akkor a 701 jó választás: A 701 ui. olyan prímszám, ami közel esik a 2000/3-hoz, de a szomszédos kettő hatványoktól, az 512-től és az 1024-től is elég távol van.

Kulcsok a $[\,0\,;1)$ intervallumon (egyszerű szorzó módszer):
Ha a kulcsok egyenletesen oszlanak el, a

$$h(k) = \lfloor k * m \rfloor$$

függvény is kielégíti az egyszerű, egyenletes hasítás feltételét.

Szorzó módszer (multiplication method): Ha a kulcsok valós számok, tetszőleges $0 < A < 1$ konstanssal alkalmazható a

$$h(k) = \lfloor \{k * A\} * m \rfloor$$

hasító függvény. ($\{x\}$ az x törtrésze.) Nem minden lehetséges konstanssal szór egyformán jól. Knuth az $A = \frac{\sqrt{5}-1}{2} \approx 0,618$ választást javasolja, mint ami a kulcsokat valószínűleg szépen egyenletesen fogja elosztani a rések között. Az osztó módszerrel szemben előnye, hogy nem érzékeny a hasító tábla méretére.

Előjel nélküli egész kulcsok esetén, ha a táblaméretet kettő hatványnak választjuk, kikerülhető a viszonylag lassú, valós aritmetika. Tegyük fel, hogy a kulcsok w biten ábrázolt természetes számok. Ekkor $0 \leq k \leq 2^w - 1$. Ábrázoljuk szintén w biten az $s = \lfloor A * 2^w \rfloor$ konstanst. Tegyük fel, hogy a táblaméret $m = 2^p$. Legyen $q = 2^w - 1$ és $r = w - p$. Ekkor a fenti szorzó módszert jól közelíthetjük az alábbi hasító függvény definícióval, feltéve, hogy dupla szavas, előjel nélküli egész aritmetikát használunk. (& a bitenkénti „és” művelet, és >> jelöli a bitléptetést jobbra.)

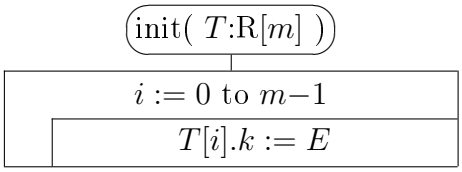
$$h(k) = ((s * k)\&q) >> r$$

Mindhárom módszer feltételezi, hogy a kulcsok számok. Ha pl. sztringek, a karaktereket tekinthetjük megfelelő számrendszerbeli előjel nélküli egész számok számjegyeinek, és így a sztringeket könnyen értelmezhetjük úgy, mint nagy, természetes számokat.

12.5. Nyílt címzés (open addressing)

Feltesszük, hogy az adatrekordok közvetlenül a résekben vannak; a $T : R[m]$ hasító tábla R típusú rekordjainak van egy $k : U \cup \{E, D\}$ kulcsmezőjük és járulékos mezőik, ahol E és D extremális konstansok ($E, D \notin U$), sorban az üres (Empty) és a törölt (Deleted) rések (slots) jelölésére.

R
+ $k : U \cup \{E, D\}$ // k is a key or it is Empty or Deleted
+ ... // satellite data



Jelölések a nyílt címzéshez:
$h : U \times 0\,..\,(m-1) \to 0\,..\,(m-1)$: próbafüggvény (probing function)
$\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$: potenciális próbasorozat (potential probing sequence)

Feltesszük, hogy a hasító táblában nincsenek duplikált (double) kulcsok[24].

Az üres (empty) és a törölt (deleted) réseket (slots) együtt *szabad (free)* réseknek nevezzük. (A többi rés *foglalt (occupied).*) Egyetlen hasító függvény helyett most m darab hasító függvényünk van:

$$h(\cdot, i) : U \to 0\,..\,(m-1) \qquad (i \in 0\,..\,(m-1))$$

12.5.1. Nyílt címzés: beszúrás és keresés, ha nincs törlés

Ha a hasító táblából nem akarunk törölni (ahogy ez sok alkalmazásban így is van), a beszúrás is egyszerűbb.

A k kulcsú r adat **beszúrás**ánál például először a $h(k,0)$ réssel próbálkozunk. Ha ez foglalt és a kulcsa nem k, folytatjuk a $h(k,1)$-gyel stb., mígnem üres rést találunk, vagy k kulcsú foglalt rést találunk, vagy kimerítjük az összes

[24]Tetszőleges adattárolóban egy kulcs duplikált, ha az adattárolóban (itt a hasító táblában) legalább kétszer fordul elő.

lehetséges próbát, azaz a $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ *potenciális próbasorozatot.* Ha üres rést találunk, ebbe tesszük az adatot, különben sikertelen a beszúrás.

A $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ sorozatot azért nevezzük *potenciális próbasorozatnak*, mert a beszúrás, keresés vagy törlés során ennek ténylegesen csak egy prefixét állítjuk elő. A potenciális próbasorozatnak azt a prefixét, amit egy beszúrás, keresés (vagy törlés) esetén ténylegesen előállítunk, *aktuális próbasorozatnak (actual probing sequence)* nevezzük. A potenciális próbasorozattal szemben megköveteljük, hogy a $\langle 0, 1, \ldots, (m-1)\rangle$ egy permutációja (permutation) legyen, azaz, hogy az egész hasító táblát lefedje (és így ne hivatkozzon kétszer vagy többször ugyanarra a résre). Ha a beszúrás a $h(k, i-1)$ próbánál áll meg, akkor (és csak akkor) a beszúrás (azaz az aktuális próbasorozat) hossza i.

A k kulcsú adat **keresés**énél is a fenti potenciális próbasorozatot követjük, és akkor állunk meg, ha megtaláltuk a keresett kulcsú foglalt rést (sikeres keresés), illetve ha üres rést találunk vagy kimerítjük a potenciális próbasorozatot (sikertelen keresés). Ha a keresés a $h(k, i-1)$ próbánál áll meg, akkor (és csak akkor) a keresés (azaz az aktuális próbasorozat) hossza i.

Ideális esetben egy tetszőleges potenciális próbasorozat a $\langle 0, 1, \ldots, (m–1)\rangle$ sorozatnak mind az $m!$ permutációját azonos valószínűséggel állítja elő. Ilyenkor *egyenletes hasításról (uniform hashing)* beszélünk.

Amennyiben a táblában nincsenek törölt rések – egyenletes hasítást és a hasító tábla $0 < \alpha < 1$ kitöltöttségét feltételezve –, egy sikertelen keresés (unsuccessful search) illetve egy sikeres beszúrás (successful insertion) várható hossza legfeljebb

$$\frac{1}{1-\alpha}$$

míg egy sikeres keresés (successful search) illetve sikertelen beszúrás (unsuccessful insertion) várható hossza legfeljebb

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

Ez azt jelenti, hogy egyenletes hasítást feltételezve, pl. 50%-os kitöltöttség mellett egy sikertelen keresés (illetve egy sikeres beszúrás) várható hossza legfeljebb 2, míg egy sikeres keresésé (illetve egy sikertelen beszúrásé) kisebb, mint 1,387; 90%-os kitöltöttség mellett pedig egy sikertelen keresés (illetve egy sikeres beszúrás) várható hossza legfeljebb 10, míg egy sikeres keresésé (illetve egy sikertelen beszúrásé) kisebb, mint 2,559 [4].

12.5.2. Nyílt címzésű hasítótábla műveletei, ha van törlés is

A **törlés** egy sikeres keresést követően a megtalált i rés kulcsának *törölt*-re (D) állításából áll. Itt az *üres*-re (E) állítás helytelen lenne, mert ha például feltesszük, hogy a k kulcsú adatot kulcsütközés miatt a $T[h(k,1)]$ helyre tettük, majd töröltük a $h(k,0)$ helyen levő adatot (azaz a $T[h(k,0)]$ rést üresre állítottuk), akkor egy ezt követő keresés nem találná meg a k kulcsú adatot.

A **keresés**nél tehát átlépjük a törölt réseket is, és csak akkor állunk meg,
- ha megtaláltuk a keresett kulcsú elemet (sikeres keresés),
- ha üres rést találunk vagy kimerítjük a potenciális próbasorozatot (sikertelen keresés).

A **beszúrás**nál is egy teljes keresést végzünk el, most a beszúrandó adat kulcsára, de ha közben találunk törölt rést, az első ilyent megjegyezzük.
- Ha a keresés sikeres, akkor a beszúrás sikertelen (hiszen duplikált kulcsot nem engedünk meg).
- Ha a keresés sikertelen, és találtunk közben törölt rést, akkor a beszúrandó adatot az elsőként talált törölt résbe tesszük (hogy a jövőbeli keresések hossza a lehető legkisebb legyen).
- Ha a keresés sikertelen, de nem talál törölt rést, viszont üres résen áll meg, akkor a beszúrandó adatot ebbe az üres résbe tesszük.
- Ha a keresés sikertelen, de nem talál sem törölt, sem üres rést, és így azért áll meg, mert a potenciális próbasorozatot kimeríti, akkor a beszúrás sikertelen (mert a hasítótábla tele van).

Ha elég sokáig használunk egy nyílt címzésű hasító táblát, elszaporodhatnak a törölt rések, és elfogyhatnak az üres rések, holott a tábla esetleg közel sincs tele. Ez azt jelenti, hogy pl. a sikertelen keresések az egész táblát végig fogják nézni. Ez ellen a tábla időnkénti frissítésével védekezhetünk, amikor megszüntetjük a törölt réseket. (A legegyszerűbb megoldás kimásolja az adatokat egy temporális területre, üresre inicializálja a táblát, majd a kimentett adatokat egyesével újra beszúrja.)

12.5.3. Lineáris próba (Linear probing)

Ebben és a következő két alfejezetben áttekintünk három stratégiát a $\langle h(k,0), h(k,1), \ldots, h(k,m–1)\rangle$ próbasorozat (szükséges kezdőszeletének) előállítására. Mindegyiknél lesz egy elsődleges $h_1 : U \to 0\,..\,(m-1)$ hasítófüggvény, aminek az értéke egyben az első próba indexét, $h(k,0)$-t adja. Ebből kiindulva lépkedünk a réseken tovább, ha ez szükséges. Elvárás, hogy h_1

egyszerű egyenletes hasítás legyen. Mindhárom stratégiánál feltesszük, hogy a kulcsok természetes számok (így lehet pl. $D = -1$ és $E = -2$).

A próbák közül a legegyszerűbb, a lineáris próba definíciója a következő.

$$h(k, i) = (h_1(k) + i) \bmod m \qquad (i \in 0\,..\,(m-1))$$

Könnyű implementálni, de összesen csak m db különböző potenciális próbasorozat van, az egyenletes hasításhoz szükséges $m!$ db próbasorozathoz képest, hiszen ha két kulcsra $h(k_1, 0) = h(k_2, 0)$, akkor az egész próbasorozatuk megegyezik. Ez a *másodlagos csomósodás (secondary clustering)*.

Ezenkívül, a különböző próbasorozatok összekapcsolódásával foglalt rések hosszú, összefüggő sorozatai alakulhatnak ki, megnövelve a várható keresési időt. Ezt a jelenséget *elsődleges csomósodásnak (primary clustering)* nevezzük. Minél hosszabb egy ilyen „csomó", annál valószínűbb, hogy a következő beszúráskor a hossza tovább fog növekedni. Ha pl. két szabad rés között (ciklikusan értve) i db foglalt rés van, akkor legalább $(i + 2)/m$ a valószínűsége, hogy a következő sikeres beszúráskor ez a csomó még hosszabb lesz, és az is lehet, hogy közben összekapcsolódik egy másik csomóval.

Meg kell itt jegyeznünk, hogy megfontolásaink a standard memóriamodellre vonatkoznak. Hierachikus modell esetén az elsődleges csomósodást előnyünkre is fordíthatjuk. (Ld. [4] Fourth Edititon, 2022.)

12.5.4. Négyzetes próba (Quadratic probing)*

$$h(k, i) = (h_1(k) + c_1 i + c_2 i^2) \bmod m \qquad (i \in 0\,..\,m-1)$$

ahol $c_1, c_2 \in \mathbb{R}; c_2 \neq 0$. A különböző próbasorozatok nem kapcsolódnak össze, de itt is csak m db különböző próbasorozat van. A *másodlagos csomósodás* tehát itt is megjelenik.

A négyzetes próba konstansainak megválasztása Annak érdekében, hogy a próbasorozat az egész táblát lefedje, a c_1, c_2 konstansokat körültekintően kell megválasztani. Ha például a tábla m mérete kettő hatvány, akkor $c_1 = c_2 = 1/2$ jó választás. Ráadásul ilyenkor

$$h(k, i) = \left(h_1(k) + \frac{i + i^2}{2}\right) \bmod m \qquad (i \in 0\,..\,m-1)$$

Ezért

$$(h(k, i) - h(k, i-1)) \bmod m = \left(\frac{i + i^2}{2} - \frac{(i-1) + (i-1)^2}{2}\right) \bmod m =$$

$$i \bmod m \qquad (i \in 1\,..\,(m-1))$$

azaz

$$h(k,i) = (h(k,i-1) + i) \bmod m \qquad (i \in 1\,..\,(m-1))$$

12.1. Feladat. *Készítsük el a fenti rekurzív képlet segítségével a négyzetes próba ($c_1 = c_2 = 1/2$) esetére a beszúrás, a keresés és a törlés struktogramjait!*

12.5.5. Kettős hasítás (Double hashing)

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m \qquad (i \in 0\,..\,(m-1))$$

ahol $h_1 : U \to 0\,..\,(m-1)$ és $h_2 : U \to 1\,..\,(m-1)$ hasító függvények. A potenciális próbasorozat pontosan akkor fedi le az egész hasító táblát, ha $h_2(k)$ és m relatív prímek. Ezt a legegyszerűbb úgy biztosítani, ha a m kettő hatvány és $h_2(k)$ minden lehetséges kulcsra páratlan szám, vagy m prímszám. Például ha m prímszám (ami lehetőleg ne essen kettő hatvány közelébe) és m' kicsit kisebb (mondjuk $m' = m - 1$ vagy $m' = m - 2$) akkor

$$h_1(k) = k \bmod m$$

$$h_2(k) = 1 + (k \bmod m')$$

egy lehetséges választás.

A kettős hasításnál minden különböző $(h_1(k), h_2(k))$ párokhoz különböző potenciális próbasorozat tartozik. Ezért, ha m prímszám vagy kettő hatvány, $\Theta(m^2)$ különböző potenciális próbasorozat lehetséges: Ha m prímszám, akkor $m * m'$; ha m kettő hatvány, akkor $\frac{m^2}{2}$. Például, ha $m = 1609$ és $m' = 1607$ (ikerprímek), akkor 2_585_663 különböző potenciális próbasorozatunk van, szemben a lineáris és a négyzetes próba esetén adott 1609-cel.

Általános m értékekre viszont nem egyszerű biztosítani, hogy m-hez képest $h_2(k)$ minden lehetséges kulcsra relatív prím legyen, továbbá az ilyen $h_2(k)$ értékek viszonylag ritkán helyezkedhetnek el az $1\,..\,(m-1)$ intervallumban, mert az m osztóinak a többszörösei kiesnek.

Ezért m-et pímszámnak vagy kettő hatványnak szokták választani. Ezekben az esetekben a kettős hasításra nem jelemző sem az elsődleges, sem a másodlagos csomósodás. Továbbá, bár a kettős hasítás potenciális próbasorozatainak száma (ezekben az esetekben is) messze van az egyenletes hasítás $m!$ számú potenciális próbasorozatától, úgy tűnik, hogy a teljesítménye (az aktuális próbasorozatok hossza), ezekben az esetekben jól közelíti az egyenletes hasításra kapott elméleti eredményeket. [4]

12.2. Feladat. *Bizonyítsuk be, hogy a kettős hasítás potenciális próbasorozata pontosan akkor fedi le az egész hasító táblát, ha $h_2(k)$ és m relatív prímek.*

A kettős hasítás műveleteinek szemléltetése: Mivel $h(k,i) = (h_1(k) + ih_2(k)) \bmod m \quad (i \in 0\,..\,(m-1))$, azért $h(k,0) = h_1(k)$ és $h(k,i+1) = (h(k,i) + d) \bmod m$, ahol $d = h_2(k)$. Az első próba helyének $(h_1(k))$ meghatározása után tehát mindig d-vel lépünk tovább, ciklikusan.

Legyen most $m = 11$ $h_1(k) = k \bmod 11$ $h_2(k) = 1 + (k \bmod 10)$.

Az alábbi táblázat 1. „op" (művelet) oszlopában, minden műveletnél a karakter a művelet kódja, azaz i=insert, s=search, d=delete; közvetlenül utána jön keresett, vagy törlendő adat kulcsa (ami értelemszerűen sosem E vagy D). A táblázatban nem foglalkozunk a járulékos adatokkal, csak a kulcsokat jelöljük. A 2. oszlopban találjuk a próbasorozatot, aminek első tagja az elsődleges hasító függvény érteke, azaz $h(k,0) = h_1(k)$. A 3. oszlopban következik a $d = h_2(k)$ lépésköz, de csak akkor, ha ezt szükséges kiszámolni, azaz az aktuális próbasorozat hossza ≥ 2. A 4. oszlopban jön a művelet sikerességének jelzése, ahol „$+$ = sikeres" és „$-$ = sikertelen".

Ha egy beszúrás közben törölt rést találunk, ne felejtsük el, hogy a beszúrás algoritmusa megjegyzi az első törölt rést, amit talál, ha talál ilyet még a keresés befejezése előtt, és ekkor, sikertelen keresés esetén a sikeres beszúrás a kulcsot ide teszi. (Ld. a 12.5.2. alfejezetet!) Ezt a törölt rést a táblázatban a próbasorozatnál aláhúzással jelöljük. (Az alábbi példában ld. alulról a 2. és a 4. sort!) A táblázat következő 11 oszlopában az üres réseket egyszerűen üresen hagytuk. Mindegyik foglalt résbe beírtuk a megfelelő kulcsot, míg a törölt réseket D betűvel jelöltük.

op	probes	h_2	s	0	1	2	3	4	5	6	7	8	9	10
init			+											
i32	10		+											32
i40	7		+								40			32
i37	4		+					37			40			32
i15	4; 10; 5	6	+					37	15		40			32
i70	4; 5; 6	1	+					37	15	70	40			32
s15	4; 10; 5	6	+					37	15	70	40			32
s104	5; 10; 4; 9	5	−					37	15	70	40			32
d15	4; 10; 5	6	+					37	D	70	40			32
s70	4; 5; 6	1	+					37	D	70	40			32
i70	4; <u>5</u>; 6	1	−					37	D	70	40			32
d37	4		+					D	D	70	40			32
i104	<u>5</u>; 10; 4; 9	5	+					D	104	70	40			32
s15	4; 10; 5; 0	6	−					D	104	70	40			32

12.3. Példa. A kettős hasítás programozása: *Írja meg beszúrás, a keresés és a törlés struktogramjait, ahol x a beszúrandó adat, k a keresett kulcs,*

illetve a törlendő adat kulcsa.

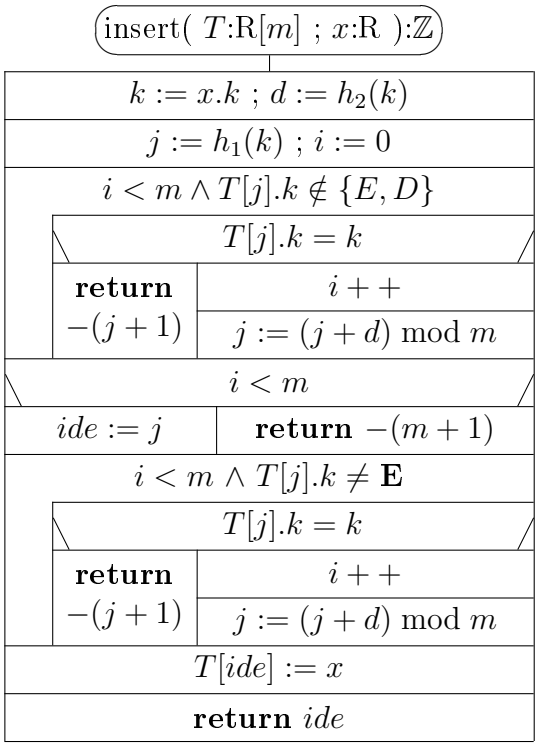
A hasító tábla a $T[0\,..\,m)$, azaz hagyományosan (célszerűen) nullától indexeljük, és m a mérete.

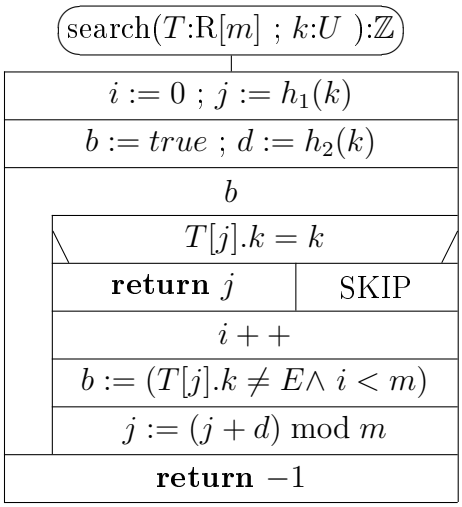
Sikeres keresésnél a keresett kulcsú adat pozícióját adjuk vissza. A sikertelen keresést a „-1” visszaadásával jelezzük.

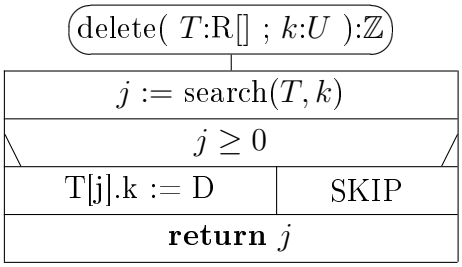
Sikeres beszúrásnál a beszúrás pozícióját adjuk vissza. Sikertelen beszúrásnál két eset van. Ha nincs elég hely a táblában, azt a „$-(m+1)$” visszaadásával jelezzük. Ha a j indexű résben megtaláljuk a táblában a beszúrandó adat kulcsát, azt a „$-(j+1)$” visszaadásával jelezzük.

Vegyük észre, hogy a nyílt címzésű hasító tábla üresre inicializálásának és az adott kulcsú foglalt rés törlésének struktogramja nem függ attól, hogy a beszúrást és a keresést milyen stratégiával hajtjuk végre. Sikeres törlésnél a törölt adat pozícióját adjuk vissza. A sikertelen törlést a „-1” visszaadásával jelezzük.

Megoldás:






delete(T:R[] ; k:U):$\mathbb{Z}$
$j := \text{search}(T, k)$
$j \geq 0$
T[j].k := D
SKIP
return j