Algoritmusok és adatszerkezetek I. előadásjegyzet

Ásványi Tibor – asvanyi@inf.elte.hu

2017. január 22.

1. Bevezetés

Az itt következő előadásjegyzetekben a láncolt listákról, a rendezésekről és a programok aszimptotikus műveletigényéről szóló fejezetek még nem teljesek; egyenlőre csak a bináris és általános fákkal kapcsolatos részeket dolgoztam ki részletesen, illetve a B+ fák témakörben angol nyelvű egyetemi tananyag fordítását bocsátom rendelkezésükre. Az előadásokon tárgyalt programok struktogramjait igyekeztem minden esetben megadni, a másolási hibák kiküszöbölése érdekében. Ebben a tananyagban sajnos még egyáltalán nincsenek a megértést segítő ábrák. Ezek bőségesen megtalálhatók az ajánlott segédanyagokban.

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 (2016)
http://aszt.inf.elte.hu/~asvanyi/ad/ad1jegyzet.pdf

[2] Ásványi Tibor, AVL tree balancing rules (2016)
http://aszt.inf.elte.hu/~asvanyi/ad/AVL-tree_balancing.pdf

[3] Ásványi Tibor, Algoritmusok I. gyakorló feladatok (2015)
http://aszt.inf.elte.hu/~asvanyi/ad/ad1feladatok.pdf

[4] Fekete István, Algoritmusok jegyzet
http://people.inf.elte.hu/fekete/algoritmusok_jegyzet/

[5] Fekete István, Algoritmusok jegyzet (régi)
http://aszt.inf.elte.hu/~asvanyi/ad/FeketeIstvan/

[6] Carl Burch, Ásványi Tibor, B+ fák
http://aszt.inf.elte.hu/~asvanyi/ad/B+ fa.pdf

[7] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
magyarul: Új Algortimusok, *Scolar Kiadó*, Budapest, 2003.
ISBN: 963 9193 90 9
angolul: Introduction to Algorithms (Third Edititon),
The MIT Press, 2009.

[8] 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

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

Saját jegyzeteiken kívül elsősorban az ebben a jegyzetben ([1] a [2] kiegészítéssel), illetve az itt hivatkozott helyeken [4, 5, 6, 7] leírtakra támaszkodhatnak. Wirth [8] és Weiss [9] klasszikus munkáinak megfelelő fejezetei pedig értékes segítséget nyújthatnak a mélyebb megértéshez. Ennek a jegyzetnek a „*”-gal jelölt alfejezetei szintén a mélyebb megértést szolgálják, azaz nem részei a vizsga anyagának.

A vizsgákon az elméleti kérdések egy-egy tétel bizonyos részleteire vonatkoznak. Lesznek még megoldandó feladatok, amiket [3] alapján állítok össze.

2. Tematika

Minden tételhez: Egy algoritmus, program, művelet bemutatásának mindig része a műveletigény elemzése. Hivatkozások: például a „[4] 1, 14, 18" jelentése: a [4] sorszámú szakirodalom adott fejezetei.

1. Függvények aszimptotikus viselkedése $(O, o, \Omega, \omega, \Theta)$. Programok műveletigénye (futási idő nagyságrendje: $T(n), mT(n), AT(n), MT(n)$), példák: beszúró rendezés (insertion sort) és összefésülő (összefuttató) rendezés (merge sort) ([1]; [4] 1, 14, 18; [5]; [7] 1-3.)

2. Elemi, lineáris adatszerkezetek: tömbök, láncolt listák, láncolt listák típusai, listakezelés. ([1]; [4] 3, 6; [7] 10; [8] 4.1 - 4.3)

3. Elemi adattípusok: vermek, sorok ([1]; [4] 3-4; [7] 10.1), megvalósításuk tömbös és láncolt reprezentációk esetén (láncolt listás megvalósítás a gyakorlatokon). Vermek felhasználása: kifejezések lengyel formára hozása, lengyel forma kiértékelése (ld. gyakorlat).

4. Elsőbbségi (prioritásos) sorok megvalósítása rendezetlen illetve rendezett tömbbel [1], rendezetlen, illetve rendezett listával (HF), továbbá kupaccal ([1]; [4] 8; [7] 6.5). A reprezentációk és az implementációk összehasonlítása a műveletek hatékonysága szempontjából.

5. Fák, bináris fák, bejárások, láncolt reprezentáció, példák ([1]; [4] 7, 11.2.3; [7] 10.4, 12.1).

6. Speciális bináris fák, aritmetikai ábrázolás, kupac, kupac műveletei, kupacrendezés (heapsort) ([1]; [4] 8.3-8.5, 16; [7] 6).

7. Véletlen építésű bináris keresőfák és műveleteik ([1]; [4] 11; [7] 12; [8] 4.4).

8. AVL fák és műveleteik: kiegyensúlyozási sémák, programok. Az AVL fa magassága ([1, 2], [4] 12; [5]; [8] 4.5).

9. Általános fák, bináris láncolt reprezentáció, bejárások ([1]; [8] 4.7 bevezetése). Egyéb reprezentációk? (HF)

10. B+ fák és műveleteik [6].

11. Gyorsrendezés (quicksort) ([1]; [7] 7; [5]; [4] 17).

12. A beszúró, összefésülő, kupac és gyors rendezés összehasonlítása. Az összehasonlító rendezések alsókorlát-elemzése ([4] 19; [7] 8.1).

3. Jelölések

$\mathbb{N} = \{0; 1; 2; 3; \ldots\}$ a természetes számok halmaza.
$\mathbb{Z} = \{\ldots -3; -2, -1; 0; 1; 2; 3; \ldots\}$ az egész számok halmaza.
$\mathbb{R}$ a valós számok halmaza.
$\mathbb{P}$ a pozitív valós számok halmaza.
$\lg n = \begin{cases} \log_2 n & \text{ha} \quad n > 0 \\ 0 & \text{ha} \quad n = 0 \end{cases}$
$fele(n) = \lfloor \frac{n}{2} \rfloor$, ahol $n \in \mathbb{N}$

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, F, L betűk pointereket (azaz mutatókat, memóriacímeket, illetve ilyen típusú változókat) jelölnek.

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

Az x, y, z *alapértelmezésben* T típusú változókat jelölnek.

A tömböket pl. így jelöljük: $A[1..n], B[i..j], C[k..m]{:}T$. Az alapértelmezett elemtípus a T, amit az A és a B tömbnél nem adtunk meg, míg a C tömbnél megadtuk.

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.

3.1. A struktogramok paraméterlistái, érték és referencia módú paraméterek

A struktogramokhoz mindig tartozik egy eljárásnév és általában egy paraméterlista is (ami esetleg üres, de a „()” zárójelpárt ott is, és a megfelelő eljáráshívásban is kiírjuk). *Ha egy struktogramhoz csak név tartozik, akkor az úgy értendő, mintha a benne található kód a hívás helyén lenne.*

Ha egy paraméterlistával ellátott struktogramban olyan változónév szerepel, ami a paraméterlistán nem szerepel, akkor ez alapértelmezésben a struktogrammal leírt eljárás lokális változója.

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

A skalár típusú paraméterek referencia módú paraméterek is lehetnek, de akkor ezt a formális paraméter listán a paraméter neve előtt egy & jellel jelölni kell (pl. swap($\&x, \&y$)).[3] A formális paraméter listán a felesleges &-prefixek hibának tekintendők, mert a referencia módú skalár paraméterek kezelése az eljárás futása során a legtöbb implementációban lassúbb, mint az érték módú paramétereké.

Az aktuális paraméter listán nem jelöljük külön a referencia módú paramétereket[4], mert egyrészt a formális paraméter listáról kiderül, hogy egy tetszőleges paraméter érték vagy referencia módú-e, másrészt az &-prefix az aktuális paraméternél teljesen mást jelent (tudniillik, hogy a jelölt adatra mutató pointert szeretnénk a hívott eljárásnak – az ott jelölt paramétermód szerint – átadni).

Ha az eljárásokra szövegben hivatkozunk, a paraméterek 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[5] típusú paraméterek csak referencia módú paraméterek lehetnek. (Pl. a tömböket, rekordokat nem szeretnénk a paraméterátvételkor lemásolni.) A nem-skalár típusok esetén ezért egyáltalán nem jelöljük a paramétermódot, hiszen az egyértelmű.

3.2. Tömb típusú paraméterek a struktogramokban

Ha pl. az $A[1..n]$ kifejezés egy struktogram formális paraméter listáján, vagy egy eljáráshívás aktuális paraméter listáján szerepel, akkor az egy progra-

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

[2] Az érték módú paraméterek esetén, az eljáráshíváskor 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.

[3] A referencia módú paraméterek esetén, az eljáráshíváskor 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 például a fenti swap($\&x, \&y$) eljárás törzse $\{z{:=}x;\ x{:=}y;\ y{:=}z;\}$, akkor a swap($a, b$) eljáráshívás hatására az a és b változók értéke megcserélődik. Ha az eljárásfej swap($x, \&y$) lenne, az eljáráshívás hatása „$b := a$" lenne, ha pedig az eljárásfej swap(x, y) lenne, az eljáráshívás logikailag ekvivalens lenne a SKIP utasítással.

[4] Pl. a swap($\&x, \&y$) eljárás egy lehetséges meghívása: swap($A[i], A[j]$).

[5] 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.

mozási nyelven két paramétert jelentene, az A-t és az n-et, azzal a plusz információval, hogy az A egy 1-től n-ig indexelt vektor.

Ha pl. az $A[m..n]$ kifejezés egy struktogram formális paraméter listáján, vagy egy eljáráshívás aktuális paraméter listáján szerepel, akkor az egy programozási nyelven három paramétert jelentene, az A-t, az m-et és az n-et, azzal a plusz információval, hogy az A egy m-től n-ig indexelt vektor.

Ha tehát egy struktogram egy formális paramétere pl. az $A[1..n]$, akkor a neki megfelelő eljáráshívásban a megfelelő aktuális paraméter lehet pl. $B[1..m]$, de nem lehet pl. $C[k..m]$, hiszen az előbbi kettő két, az utóbbi pedig három paramétert jelöl, és nem lehet az aktuális paraméter pl. $C[k..10]$ sem, mert ez ugyan csak két paramétert jelöl, de pl. a k nem feleltethető meg az n-nek, és nem adható értékül az 1-nek sem.

4. Algoritmusok

Az *algoritmus* egy jól definiált kiszámítási eljárás, amely valamely adatok (*bemenet* vagy *input*) felhasználásával újabbakat (*kimenet*, *eredmény* vagy *output*) állít elő. (Gondoljunk pl. két egész szám legnagyobb közös osztójára [lnko(x, y) függvény], 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 $a_{p_1} \leq a_{p_2} \leq \ldots \leq a_{p_n}$.

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

4.1. Az $A[1..n]$ vektor monoton növekvő rendezése

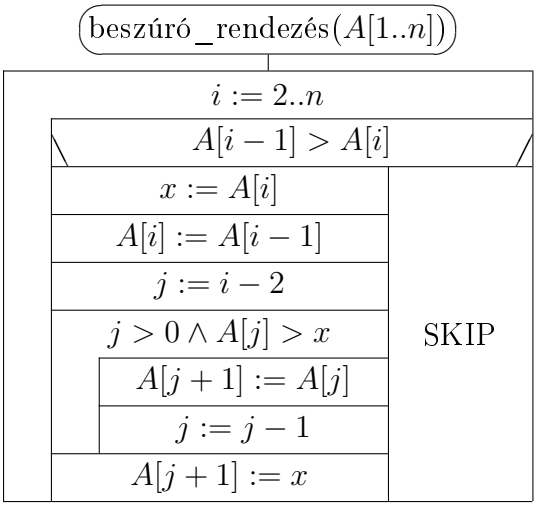
Egy sorozatot legegyszerűbben egy $A[k..u]$ vektorban tárolhatunk, ahol k a sorozat kezdő indexe u pedig az utolsó elem indexe. Az elemtípust általában nem adjuk meg, hangsúlyozva, hogy az algoritmus általános, bár a rendezéseknél feltesszük, hogy a vektor elemtípusára teljes rendezés definiált és az értékadó utasítás is értelmezve van. Ha mégis feltüntetjük a T elemtípust, akkor az $A[k..u]$ vektort $A[k..u] : T$ alakban adjuk meg.

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. a $\langle 5, 4, 2, 8, 3\rangle$-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: $\langle 5 \mid 4, 2, 8, 3\rangle \to \langle 4, 5 \mid 2, 8, 3\rangle \to \langle 2, 4, 5 \mid 8, 3\rangle \to \langle 2, 4, 5, 8 \mid 3\rangle \to \langle 2, 3, 4, 5, 8 \mid \rangle \to \langle 2, 3, 4, 5, 8\rangle$.

A rendezett beszúrás technikája attól függ, hogyan tároljuk a sorozatot. Ha egy tömbben, akkor 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 szomszéja, 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 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 $A[1..n]$ vektort rendezi monoton növekvően az előbb ismertetett egyszerű beszúró rendezéssel. A fő ciklus invariánsa:

$2 \leq i \leq (n+1) \wedge A[1..n]$ az input vektor egy permutáltja,
ami az $(i-1)$ -edik eleméig monoton növekvően rendezett.

Összefoglalva a működést: Ha $n < 2$, akkor az $A[1..n]$ vektor 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 $A[1..1]$ rendezett, azaz $i = 2$-re fennáll az invariáns. A fő ciklus magja ezután mindig $A[i]$-t szúrja be a vektor rendezett szakaszába, i-t eggyel növeli és tartja az invariánst. Mikor tehát i eléri az $n+1$ értéket, már a teljes $A[1..n]$ vektor rendezett, és ekkor be is fejeződik az eljárás.

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] Mivel ez az egyszerű rendezés a rendezendő vektoron kívül csak néhány segédváltozót igényel, extra tárigénye minimális. Így elsősorban a futási idejére lehetünk kíváncsiak. 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.

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

Meghatározhatjuk, vagy legalább becslés(eke)t adhatunk viszont arra, hogy adott n méretű input esetén az S algoritmus *hány eljáráshívást hajt végre + hányat iterálnak összesen kódban szereplő különböző ciklusok.* Megkülönböztetjük a legrosszabb vagy maximális $MT_S(n)$, a várható vagy átlagos $AT_S(n)$ és a legjobb vagy minimális $mT_S(n)$ eseteket. 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 szinonímákként fogjuk használni, és ezek alatt az *eljáráshívások 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)! A rendezés során egyetlen eljáráshívás hajtódik végre, és ez a beszúró_rendezés($A[1..n]$) 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 vektor mérete, általában a kérdéses kód által manipulált adatszerkezet mérete.[7] Lehet, hogy a belső ciklus egyet sem iterál, pl. ha a fő ciklus mindig a jobboldali ágon fut le, mert $A[1..n]$ 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 $j = 0$-ig fut. (Ez akkor áll elő, ha a vektor 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 $(i-2)$-ször iterál. Mivel az i, 2-től n-ig fut, a belső ciklus összesen legfeljebb $\sum_{i=2}^{n}(i-2)$ iterációt hajt végre. Innét

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

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

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

Látható, hogy a minimális futási idő becslése az $A[1..n]$ input vektor 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 együtthatós lineáris függvényét, $\Theta(n^2)$ pedig az n összes, pozitív együtthatós másodfokú függvényét. (Általában egy tetszőleges $g(n)$, a program hatékonyságának becslésével kapcsolatos függvényre a $\Theta(g(n))$ függvényosztály pontos definícióját a 7. 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 jogos, mert a futási idő általában nagyméretű 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 legfeljebb $(k+s)*n$ -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 vektort a rendezés előtt $\Theta(n)$ időben megfordítani, és így monoton növekvően előrendezett vektort 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:

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

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

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. 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.

4.1.3. A futási időkre vonatkozó becslések magyarázata*

Jelölje most a beszúró_rendezés($A[1..n]$) eljárás *tényleges* maximális és minimális futási idejét sorban $MT(n)$ és $mT(n)$!

Világos, hogy a rendezés akkor fut le a leggyorsabban, ha a fő ciklus minden elemet a végső helyén talál, azaz mindig a jobboldali ágon fut le. (Ez akkor áll elő, ha a vektor már eleve monoton növekvően rendezett.) Legyen a a fő ciklus jobboldali ága egyszeri végrehajtásának műveletigénye, 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 $mT(n) = a * (n-1) + b$. Legyen most $p = min(a,b)$ és $P = max(a,b)$; ekkor $0 < p \leq P$, és
$p * (n-1) + p \leq mT(n) = a * (n-1) + b \leq P * (n-1) + P$, ahonnét
$p * n \leq mT(n) \leq P * n$, azaz

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

Most adjunk becslést a beszúró rendezés maximális futási idejére, $(MT(n))$! 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 a vektor 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 $j = 0$ esetet), ahol $c, d > 0$ állandók. Ezzel a jelöléssel:

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

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

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

$q * (n + \frac{(n-1)*(n-2)}{2}) \leq MT(n) \leq Q * (n + \frac{(n-1)*(n-2)}{2})$, azaz

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

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, átlagos és maximális futási idejeire is általánosítható. (Ezt azonban már az Olvasóra bízzuk.)

4.1.4. Összefésülő rendezés (mergesort)

Az összefésülő rendezés (mergesort, rövidítve MS) nagy elemszámú sorozatokat is viszonylag gyorsan rendez. Ráadásul a legjobb és a legrosszabb eset között nem mutatkozik nagy eltérés. Első megközelítésben azt mondhatjuk, hogy a maximális és a minimális futási ideje is $n \lg n$ -nel arányos. Ezt a szokásos Θ jelöléssel a következőképpen fejezzük ki.

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

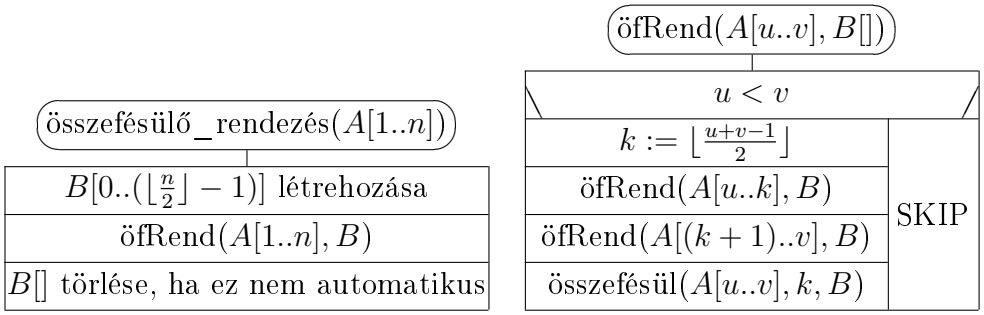
Ez alatt, a Θ jelölés fogalmát tovább tágítva, azt értjük, hogy úgy a maximális, mint a minimális futási idő alulról és felülről is becsülhető egy-egy $a * n \lg n + \delta(n)$ alakú függvénnyel, ahol $a > 0$ konstans, és az n -et növelve a $\frac{\delta(n)}{n \lg n}$ tetszőlegesen közel kerül nullához, azaz nagy n -ekre az $n \lg n$ -hez képest a $\delta(n)$ értéke elhanyagolható. Kicsit formálisabban fogalmazva:

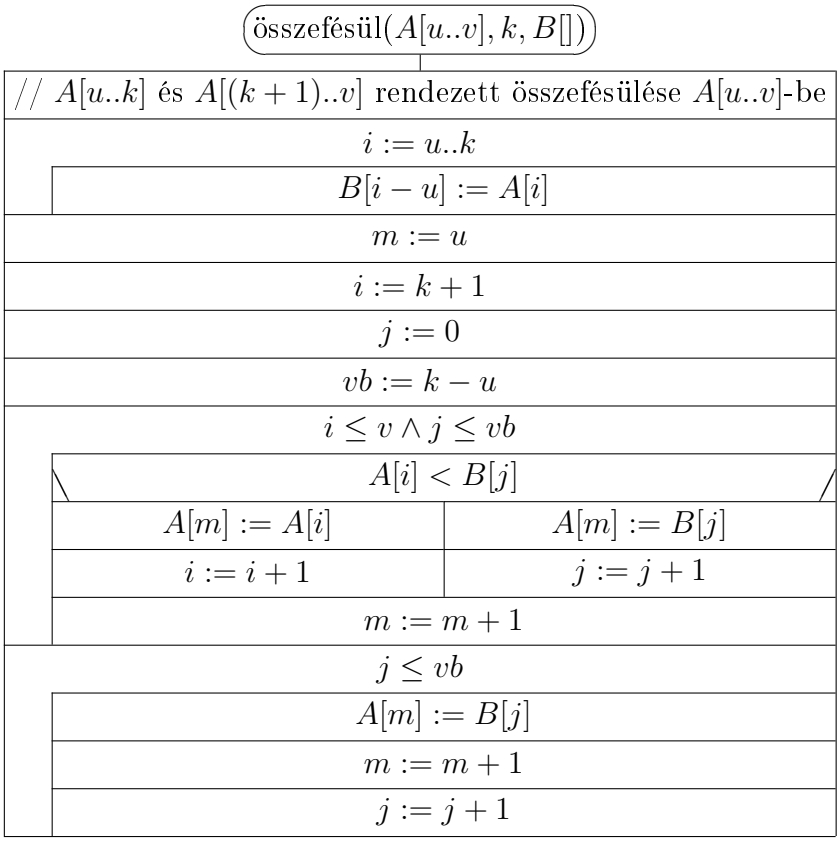
Tegyük fel, hogy $f, g : \mathbb{N} \to \mathbb{R}$ függvények; f aszimptotikusan nemnegatív (azaz valamely küszöbszámtól nemnegatív), g pedig aszimptotikusan pozitív (azaz valamely küszöbszámtól pozitív) függvény. Ebben az esetben

$f \in \Theta(g)$ akkor és csak akkor teljesül[8], ha vannak olyan $c, d > 0$ konstansok és $\varphi, \psi : \mathbb{N} \to \mathbb{R}$ függvények, hogy $\forall n \in \mathbb{N}$-re:

$$c * g(n) + \varphi(n) \leq f(n) \leq d * g(n) + \psi(n) \wedge \lim_{n \to \infty} \frac{\varphi(n)}{g(n)} = 0 \wedge \lim_{n \to \infty} \frac{\psi(n)}{g(n)} = 0$$

[8] Ez az állítás valójában a definíció egyik fontos következménye.
Ld. bővebben a 7. fejezetben!





Először az összefésül($A[u..v], k, B[]$) eljárás műveletigényét határozzuk meg. Bevezetjük az $n = v - u + 1$ jelölést. $mT_{merge}(n)$ az eljárás minimális, $MT_{merge}(n)$ a maximális műveletigénye, amiket most is az eljáráshívások + a ciklusiterációk számával közelítünk. Most csak 1 eljáráshívás van + az 1. ciklus $\lfloor n/2 \rfloor$ iterációja + a 2. és 3. ciklus iterációi: a B vektorban lévő $\lfloor n/2 \rfloor$ elemet biztosan visszamásoljuk az A-ba, ami legalább $\lfloor n/2 \rfloor$ iteráció. Ha viszont a 2. és a 3. ciklus mindegyik elemet átmásolja, az összesen a maximális n iteráció. Innét

$mT_{merge}(n) \geq 1 + \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{2} \rfloor \geq \lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor = n$, valamint

$MT_{merge}(n) \leq 1 + \lfloor \frac{n}{2} \rfloor + n \leq \lceil \frac{n}{2} \rceil + n \leq 2n - 1$, ahonnét

$$n \leq mT_{merge}(n) \leq MT_{merge}(n) \leq 2n - 1.$$

(Innét $mT_{merge}(n), MT_{merge}(n) \in \Theta(n)$ adódik.)

Most a rekurzív eljárás, az öfRend$(A[u..v], B[])$ műveletigényét határozzuk meg. Jelölje sorban $mT_{ms}(n)$ és $MT_{ms}(n)$ a minimális és a maximális műveletigényét! Kezdjük az $MT_{ms}(n)$ felső becslésével, majd folytatjuk az $mT_{ms}(n)$ alsó becslésével. Belátjuk, hogy nagyságrendben $MT_{ms}(n)$ legfeljebb $n \lg n$ -nel arányos, $mT_{ms}(n)$ pedig legalább $n \lg n$ -nel arányos, ahonnét adódik, hogy mindkettő, és végső soron az egész rendezés műveletigénye is $\Theta(n \lg n)$ nagyságrendű.

$MT_{ms}(n)$ felső becsléséhez először a rekurzió legnagyobb mélységét számoljuk ki. Világos, hogy a rekurzív hívások egy bináris fát alkotnak (ld. a 8. fejezet bevezető részét). A 0. szinten van a legkülső ($1 = 2^0$ db.) rekurzív hívás, ami a teljes, n hosszú tömbre vonatkozik. Feltéve, hogy $n \geq 2$, az 1. szinten $2 = 2^1$ rekurzív hívás van, amik $\lfloor n/2 \rfloor$, illetve $\lceil n/2 \rceil$ hosszú tömbökre vonatkoznak. Feltéve, hogy $n \geq 4$, a 2. szinten $4 = 2^2$ rekurzív hívás van, amik sorban $\lfloor \lfloor n/2 \rfloor /2 \rfloor$, $\lceil \lfloor n/2 \rfloor /2 \rceil$, $\lfloor \lceil n/2 \rceil /2 \rfloor$, illetve $\lceil \lceil n/2 \rceil /2 \rceil$ hosszú tömbökre vonatkoznak. Feltéve, hogy $n \geq 2^i$, az i-edik szinten 2^i rekurzív hívás van, és mindegyik közelítőleg $n/2^i$ hosszú tömbökre vonatkozik.

A továbbiakban feltesszük, hogy $n > 1$ (hiszen „nagy” n-ekre érdekes a műveletigény). Az $M = \lceil \lg n \rceil$ jelöléssel $2^{M-1} < n \leq 2^M$. Definiáljuk a $Fele(n) = \lceil n/2 \rceil$ függvényt! Ekkor $\forall i \in 0..M$-re, az i. rekurziós szinten a leghosszabb részvektor, amire a rekurzív eljárást meghívjuk, $Fele^i(n)$ hosszú, és $2^{M-i-1} < Fele^i(n) \leq 2^{M-i}$. Innét az M. rekurziós szinten lesz a leghoszabb részvektor hossza 1, és legkésőbb itt minden ágon leáll a rekurzió.

Most az i. rekurziós szinten (ahol $i \in 0..M$) végrehajtott, legfeljebb 2^i rekurzív „öfRend” hívás i. szintre vonatkozó műveletigényeinek összegére adunk felső becslést. Az i. szinten végrehajtott tetszőleges „öfRend” hívásnak az i. szintre vonatkozó műveletigénye alatt az eljáráshívás végrehajtását értjük a benne szereplő két rekurzív hívás nélkül, mert azokat már az $(i + 1)$. rekurziós szint műveletigényénél vesszük figyelembe. Az i. szinten végrehajtott j. „öfRend” hívásban az A vektor aktuális részvektorának hosszát $l_{i,j}$-vel jelölve, $l_{i,j} > 1$ esetén a rekurzív eljárás bal ága hajtódik végre, és $MT_{merge}(l_{i,j}) \leq 2l_{i,j} - 1$, ahonnét az aktuális „öfRend” hívásnak az i. rekurziós szintre vonatkozó műveletigényére $MT_{ms(i)}(l_{i,j}) \leq 2l_{i,j}$ adódik. Ez a felső becslés nyilván $l_{i,j} = 1$ esetén is igaz, hiszen ekkor az „öfRend” eljárás jobboldali ága hajtódik végre, és $MT_{ms(i)}(l_{i,j}) = 1$. Az

i. szinten végrehajtott „öfRend” hívások számát jelölje h_i ($h_i \leq 2^i$)! Az összes, az i. szinten végrehajtott „öfRend” hívást tekintve, a teljes műveletigény $MT_{ms(i)}(n) \leq \sum_{j=1}^{h_i} 2l_{i,j} = 2\sum_{j=1}^{h_i} l_{i,j} \leq 2n$, hiszen az i. szinten az $l_{i,j}$ hosszú részvektorok az eredeti $A[1..n]$ vektor diszjunkt szakaszait képezik. Mivel 0-tól $M = \lceil \lg n \rceil$ -ig $M = \lceil \lg n \rceil + 1$ rekurziós szint van, $MT_{ms}(n) \leq 2n(\lceil \lg n \rceil + 1) \leq 2n((\lg n) + 2) = 2n \lg n + 4n$, azaz

$$MT_{ms}(n) \leq 2n \lg n + 4n$$

Az $mT_{ms}(n)$ alsó becsléshez csak a rendezett összefésülések műveletigényei összegének alsó becslését vesszük figyelembe. Ehhez meghatározzuk, hogy a rekurzív hívásokban milyen mélységig lesz az adott i. szinten minden rekurzív hívásban $l_{i,j} > 1$, mert ezeken a szinteken minden rekurzív hívásban meghívódik az összefésülés, és $\sum_{j=1}^{h_i} l_{i,j} = n$, így az összefésülések összes műveletigényére az i. szinten $mT_{merge(i)}(n) \geq \sum_{j=1}^{h_i} l_{i,j} = n$, ahonnét $mT_{ms(i)}(n) \geq mT_{merge(i)}(n) \geq n$ adódik.

A fenti kritikus mélység meghatározásához, az $m = \lfloor \lg n \rfloor$ jelöléssel $2^{m+1} > n \geq 2^m$. A $fele(n) = \lfloor n/2 \rfloor$ függvénnyel kapjuk, hogy $\forall i \in 0..m$ -re $2^{m+1-i} > fele^i(n) \geq 2^{m-i}$, ahol $fele^i(n)$ az $l_{i,j}$ részvektor-hosszok minimuma. Ebből $2^2 > fele^{m-1}(n) \geq 2$; továbbá $2 > fele^m(n) \geq 1$, azaz $fele^m(n) = 1$. Innét adódik, hogy $\forall i \in 0..(m-1)$-re az i. szinten minden rekurzív hívásban legalább 2 hosszú a rendezendő részvektor, és így $mT_{ms(i)}(n) \geq n$, azaz $mT_{ms}(n) \geq nm = n\lfloor \lg n \rfloor \geq n((\lg n) - 1) = n \lg n - n$. Ahonnét

$$mT_{ms}(n) \geq n \lg n - n.$$

Ebből a teljes rendezésre, ami a rekurzív „öfRend” eljáráshoz képest pontosan egy többlet eljáráshívást tartalmaz, azt kapjuk, hogy

$$n \lg n - n \leq mT_{MS}(n) \leq AT_{MS}(n) \leq MT_{MS}(n) \leq 2n \lg n + 4n + 1$$

Tekintetbe véve, hogy

$$\lim_{n\to\infty} \frac{-n}{n \lg n} = 0 \quad \wedge \quad \lim_{n\to\infty} \frac{4n+1}{n \lg n} = 0,$$

a fenti struktogramok előtti állítással kapjuk, hogy

$$mT_{MS}(n), AT_{MS}(n), MT_{MS}(n) \in \Theta(n \lg n).$$

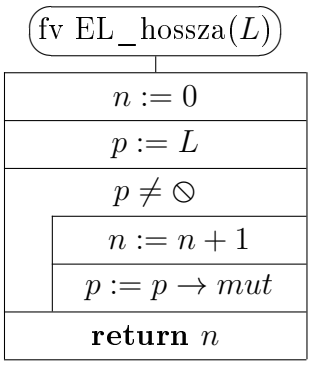
5. Láncolt listák

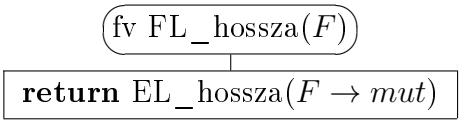
5.1. Egyirányú láncolt listák

(egyszerű listák (EL) és fejelemes listák (FL))

Az egyirányú láncolt listák elemeinek osztálya (az egyszerűség kedvéért):

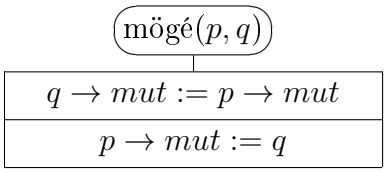
Elem
$+kulcs$: valamilyen ismert típus
$+mut$: Elem*

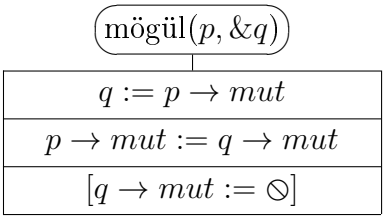


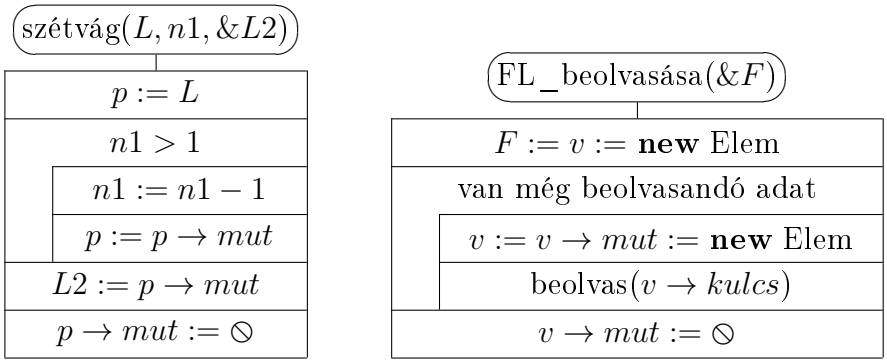


A különböző listaműveleteknél a listaelemekben levő kulcsok (és az esetleges egyéb járulékos adatok) listaelemek közti mozgatását kerüljük.

Előnyben részesítjük a listák megfelelő átláncolását, mivel *nem* tudjuk, hogy egy gyakorlati alkalmazásban mennyi járulékos adatot tartalmaznak az egyes listaelemek.







Az objektumok dinamikus létrehozására a **new** T műveletet fogjuk használni, ami egy T típusú objektumot hoz létre és visszadja 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.

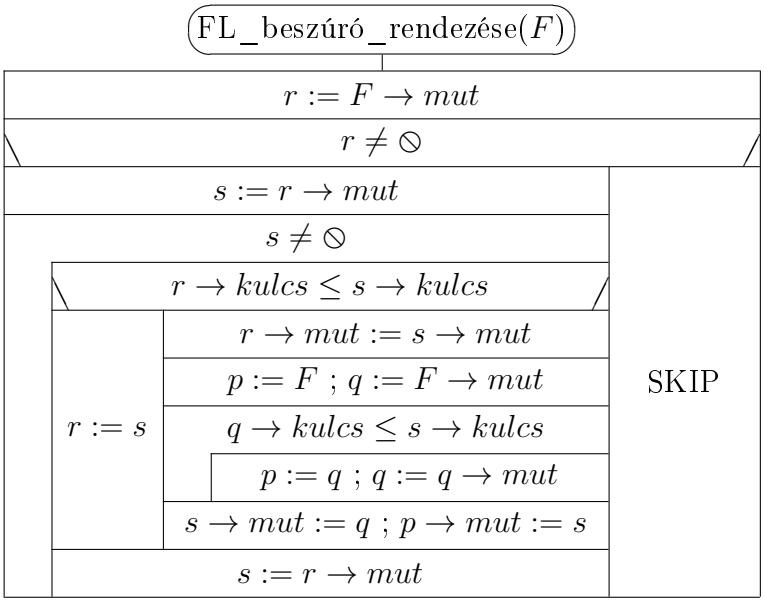
Maga a p mutató a $p :=$ **new** T utasítás végrehajtása előtt is létezik, mert azt a mutató deklarációjának kiértékelése hozza létre.[9] Ugyanígy, a p mutató a **delete** p végrehajtása után is létezik, egészen az őt (automatikusan) deklaráló eljárás vagy függvény végrehajtásának befejezéséig.

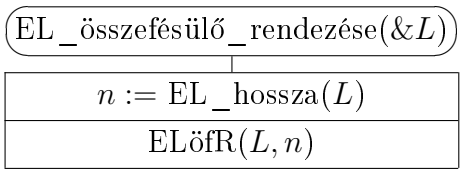
Mi magunk is írhatunk optimalizált, hatékony dinamikus memória helyfoglaló és felszabadító rutinokat; speciális esetekben akár úgy is, hogy a fenti műveletek konstans időt igényelnek. (Erre egy példa a gyakorlatokon is elhangzik.)

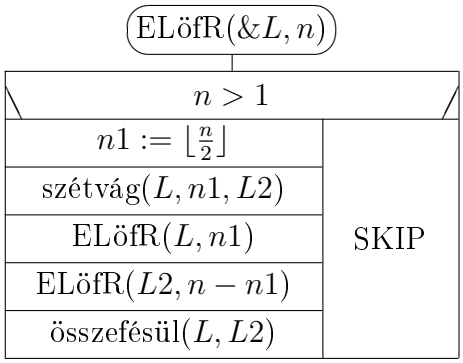
Általános esetben azonban a dinamikus helyfoglalásra használt szabad memória a különböző típusú és méretű objektumokra vonatkozó létrehozások és törlések (itt **new** és **delete** utasítások) hatására feldarabolódik, és viszonylag bonyolult könyvelést igényel, ha a feldarabolódásnak gátat akarunk vetni. Ezt a problémát a különböző rendszerek különböző hatékonysággal kezelik.

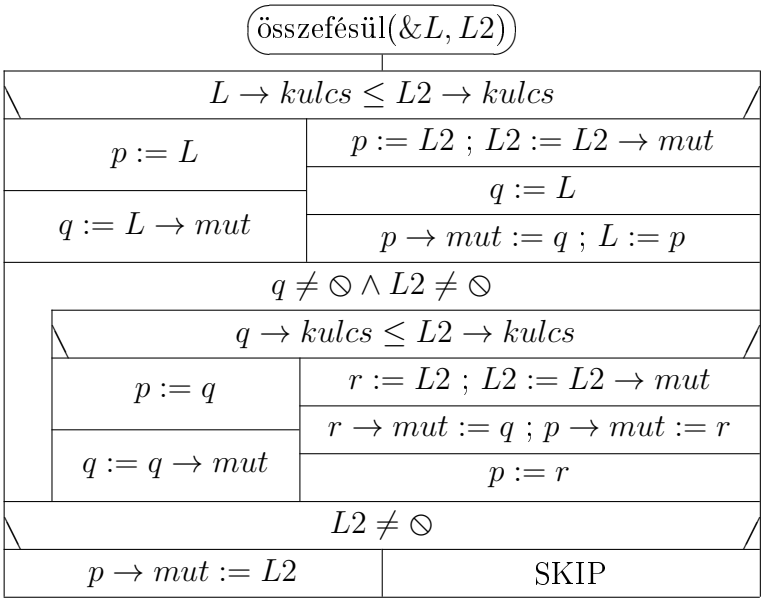
Az absztrakt programokban az objektumokat dinamikusan létrehozó (**new**) és törlő (**delete**) utasítások műveletigényeit konstans értékeknek, azaz $\Theta(1)$-nek vesszük, azzal a megjegyzéssel, hogy valójában nem tudjuk, mennyi. Ezért a lehető legkevesebbet használjuk a **new** és a **delete** utasításokat.

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


FL_beszúró_rendezése(F)
r := F → mut
r ≠ ⊘
s := r → mut
s ≠ ⊘
r → kulcs ≤ s → kulcs
r := s
r → mut := s → mut
p := F ; q := F → mut
q → kulcs ≤ s → kulcs
p := q ; q := q → mut
s → mut := q ; p → mut := s
s := r → mut
SKIP


EL_összefésülő_rendezése(&L)
n := EL_hossza(L)
ELöfR(L, n)


ELöfR(&L, n)
n > 1
n1 := ⌊n/2⌋
szétvág(L, n1, L2)
ELöfR(L, n1)
ELöfR(L2, n − n1)
összefésül(L, L2)
SKIP

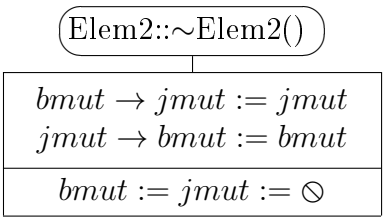


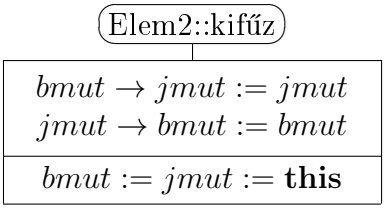
5.2. Kétirányú ciklikus láncolt listák (KCL)

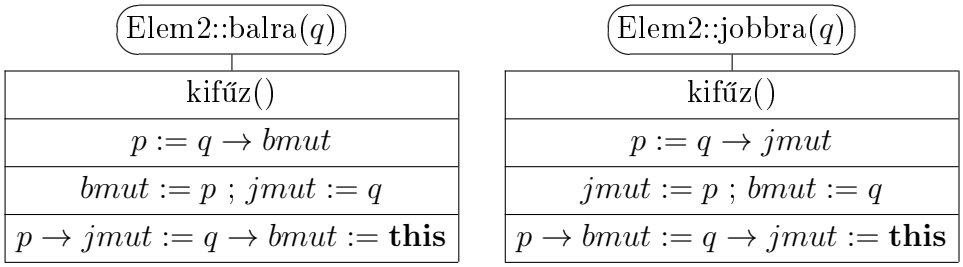
A KCL-ek elemeinek osztálya:

Elem2
$-bmut, jmut$: a bal illetve a jobb szomszédra mutat vagy **this** $+kulcs$: valamilyen ismert típus

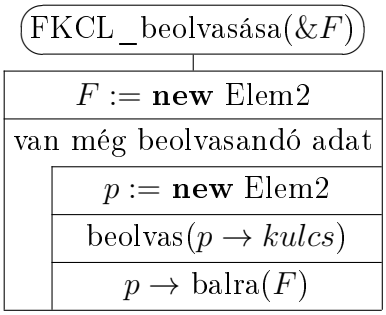
+ Elem2() { $bmut := jmut :=$ **this** } // egyelemű KCL-t képez belőle
+ $\sim$ Elem2() törlés előtt kifűzi
+ kifűz() // kifűzi és egyelemű KCL-t képez belőle
+ balra(q) // átfűzi a $*q$ KCL elemtől balra
+ jobbra(q) // átfűzi a $*q$ KCL elemtől jobbra
+ **fv** bal() { **return** $bmut$ } // a bal szomszédja címe
+ **fv** jobb() { **return** $jmut$ } // a jobb szomszédja címe

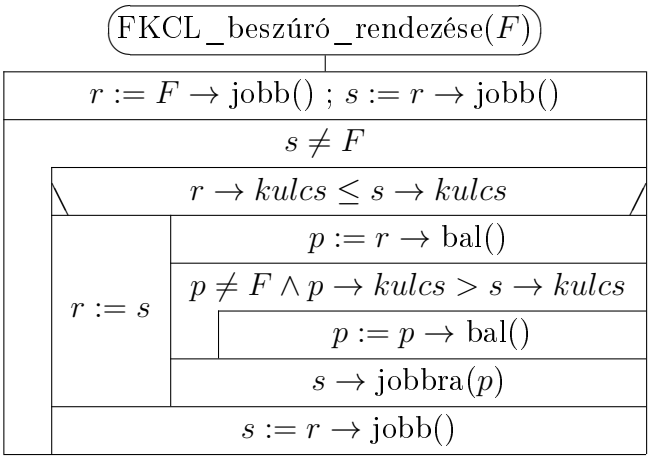






5.2.1. Fejelemes, kétirányú ciklikus láncolt listák (FKCL), mint a KCL-ek speciális esete





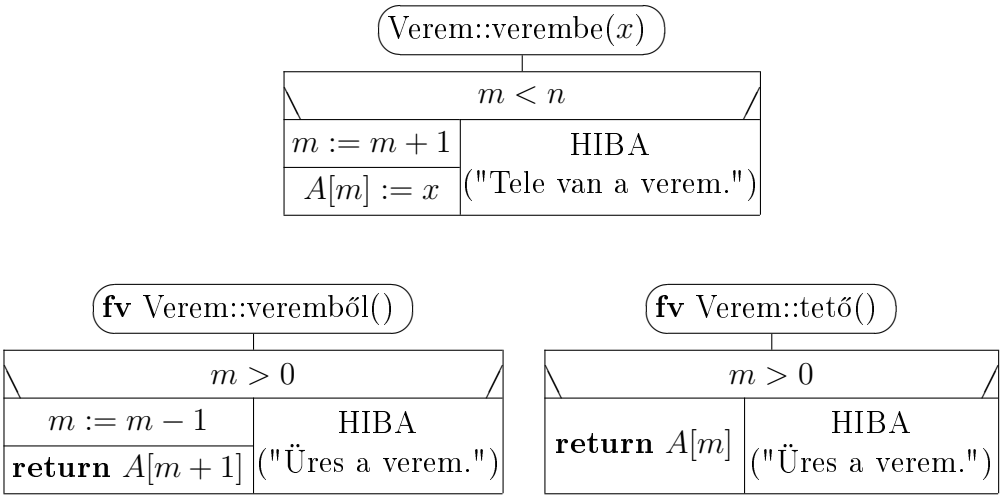
6. Absztrakt adattípusok (ADT-k)

6.1. Vermek

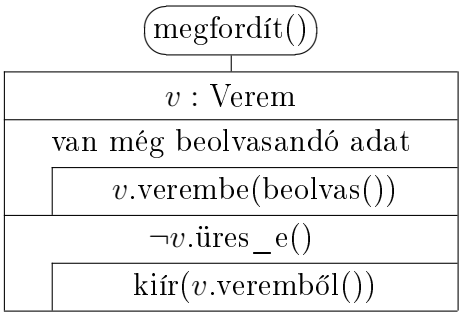
A vermet most statikus tömb ($A[1..n] : T$) segítségével reprezentáljuk, ahol n a verem maximális mérete, T a verem elemeinek típusa.

Verem
- $A[1..n] : T$ // T ismert típus, n globális konstans - $m : 0..n$ // m a verem aktuális mérete

+ Verem() $\{m := 0\}$ // üresre inicializálja a vermet
// destruktort itt nem kell definiálni
+ verembe(x) // beteszi x-et a verembe, felülre
+ **fv** veremből() // kiveszi és visszaadja a verem felső elemét
+ **fv** tető() // megnézi és visszaadja a verem felső elemét
+ **fv** tele_e()$\{$**return** $m = n\}$
+ **fv** üres_e()$\{$**return** $m = 0\}$



Példa a verem egyszerű használatára a gyakorlat anyagából: Az input adatok kiírása fordított sorrendben. (A vermet az alábbi struktogramban szokások ellenére azért deklaráltuk, hogy jelezzük: üresre inicializáltuk. Az egyszerűség kedvéért feltesszük, hogy a verem elég nagy a teljes input befogadásához.)

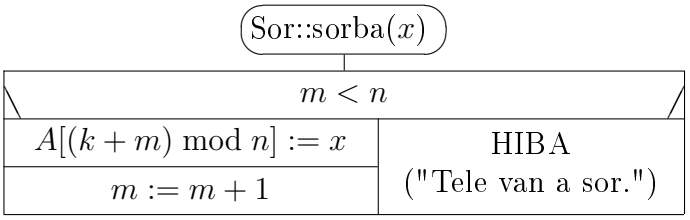


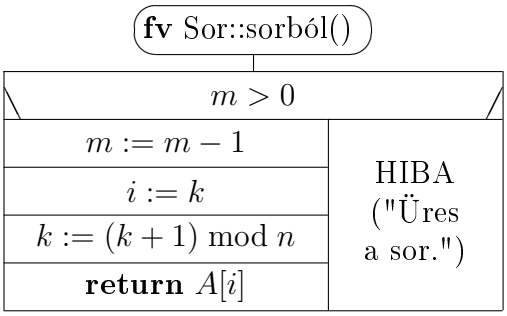
6.2. Absztrakt adattípusok (ADT-k): sorok

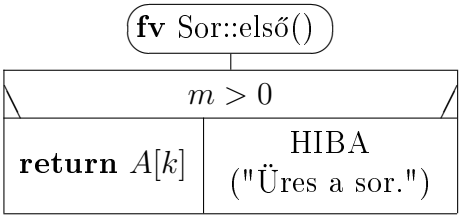
A sort statikus tömb ($A[0..n-1:T$) segítségével reprezentáljuk, ahol az n globális konstans a sor fizikai mérete, T a sor elemeinek típusa.

Sor
$-A[0..n-1]:T$ $-m:0..n$ // m a sor aktuális mérete $-k:0..n-1$ // k a sor kezdő pozíciója az A tömbben

+ Sor(){ m:=0 ; $k:=0$ } // üresre inicializálja a sort
+ sorba(x) // beteszi x-et a sor végére
+ **fv** sorból() // kiveszi és visszaadja a sor első elemét
+ **fv** első() // megnézi és visszaadja a sor első elemét
+ **fv** hossz(){**return** m}
+ **fv** üres_e(){**return** $m=0$}





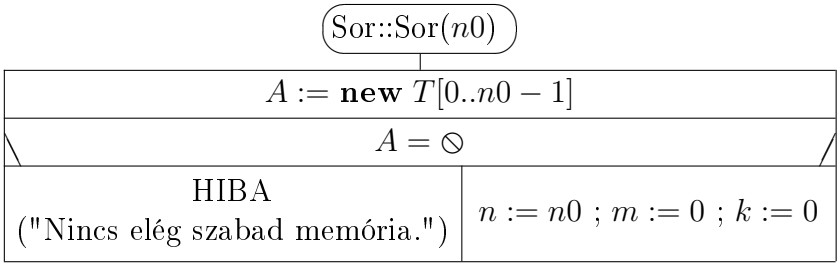


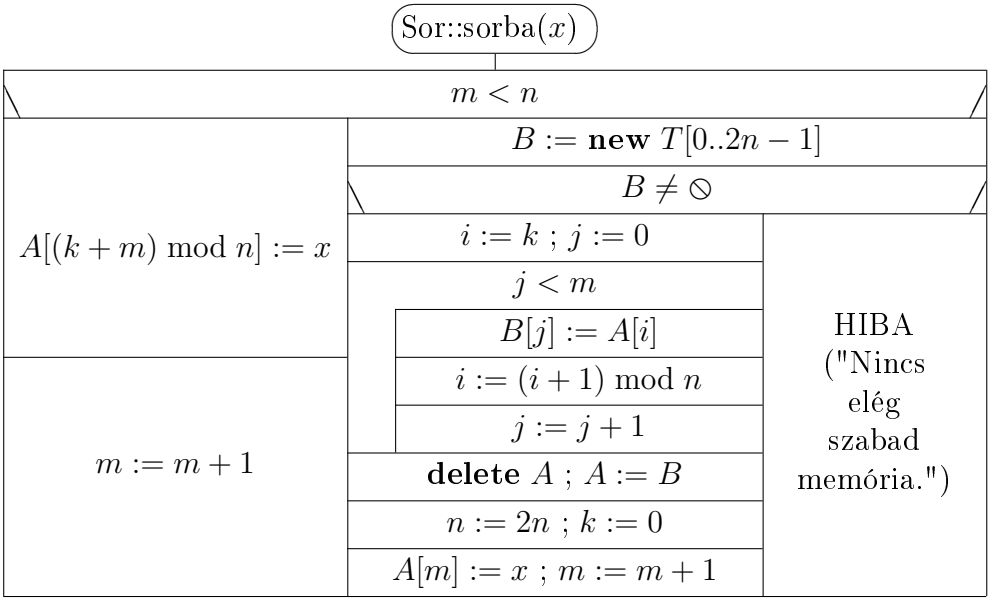
6.2.1. Sor reprezentálása dinamikus tömbbel*

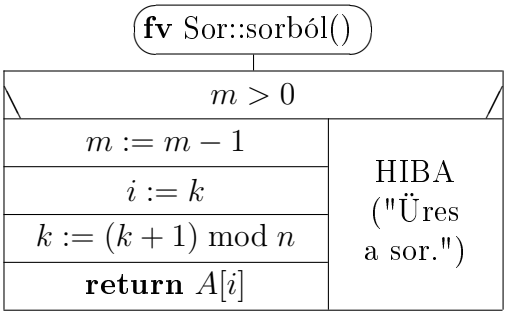
A sort dinamikusan létrehozott tömb ($A :=$ **new** $T[0..n-1]$) segítségével reprezentáljuk, ahol n a sor fizikai mérete, T a sor elemeinek típusa. Ha nincs elég memória, a **new** művelet $\oslash$-t ad vissza. (A konstruktor paramétere a kezdőméret, s ha később az s.sorba(x) művelettel a tömb betelik, kétszer akkorát foglalunk, és az adatokat átmásoljuk.)

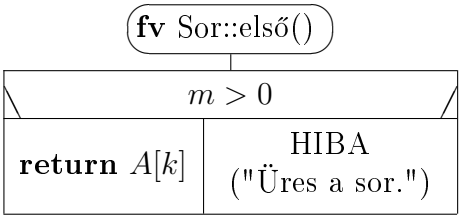
Sor
$-A : T*$ // a konstruktor létrehozza az $A[0..n-1] : T$ tömböt $-n : 1..$ // n a sor fizikai mérete $-m : 0..n$ // m a sor aktuális mérete $-k : 0..n-1$ // k a sor kezdő pozíciója az A tömbben

+ Sor($n0$) // üresre inicializálja a sort $n0$ fizikai mérettel
+ $\sim$ Sor() {**delete** A}
+ üresSor(){$m := 0$; $k := 0$} // kiüríti a sort
+ sorba(x) // beteszi x-et a sor végére
+ **fv** sorból() // kiveszi és visszaadja a sor első elemét
+ **fv** első() // megnézi és visszaadja a sor első elemét
+ **fv** hossz(){**return** m}
+ **fv** üres_e(){**return** $m = 0$}









HF: A vermet hasonlóan átírni.

6.3. Absztrakt adattípusok (ADT-k): Elsőbbségi (prioritásos) sorok

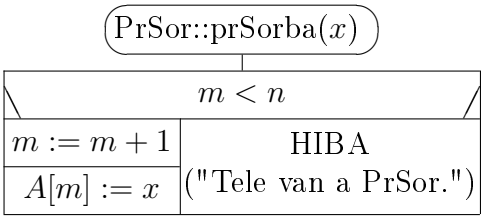
PrSor
- $A[1..n] : T$ // T ismert típus, n globális konstans - $m : 0..n$ // m a PrSor aktuális mérete

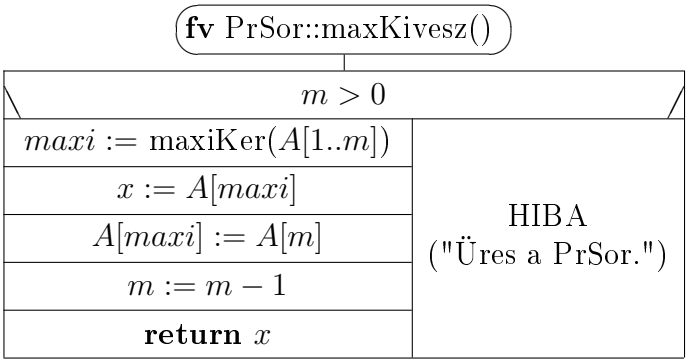
+ PrSor() $\{m := 0\}$ // üresre inicializálja a PrSort
// destruktort itt nem kell definiálni
+ prSorba(x) // beteszi x-et a prSorba
+ **fv** maxKivesz() // kiveszi és visszaadja a PrSor maximális elemét
+ **fv** max() // megnézi és visszaadja a PrSor maximális elemét
+ **fv** tele_e()$\{$**return** $m = n\}$
+ **fv** üres_e()$\{$**return** $m = 0\}$

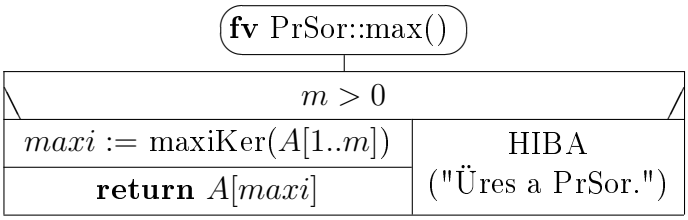
A PrSor aktuális elemeit az $A[1..m]$ résztömb tartalmazza.

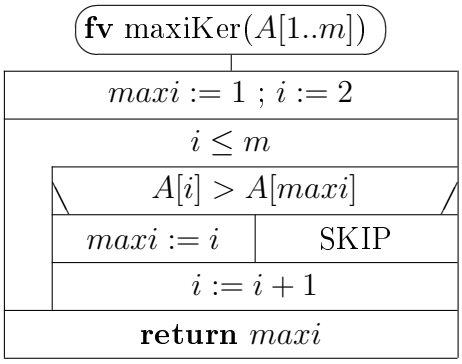
6.3.1. $A[1..m]$ rendezetlen

$T_{\text{prSorba}}(m) \in \Theta(1)$, $T_{\text{maxKivesz}}(m) \in \Theta(m)$, $T_{\max}(m) \in \Theta(m)$:





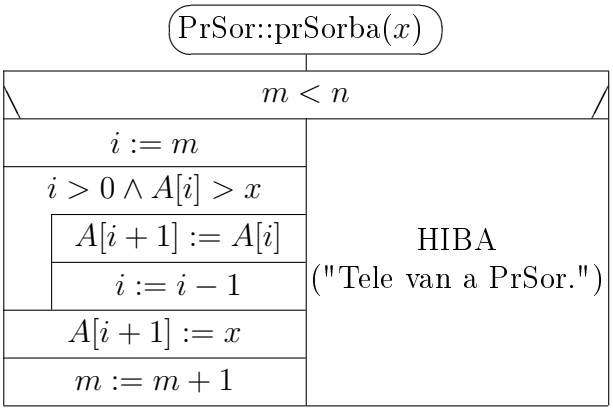


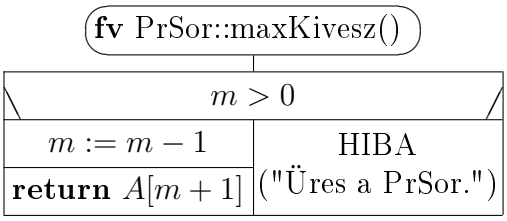


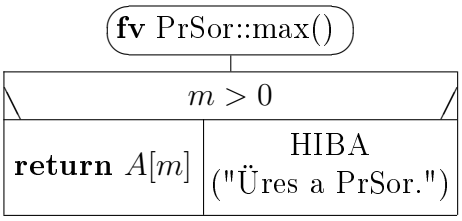
HF: Lássuk be, hogy elérhető $T_{\max}(m) \in \Theta(1)$, míg $T_{\text{prSorba}}(m) \in \Theta(1)$ és $T_{\text{maxKivesz}}(m) \in \Theta(m)$ marad! Módosítsuk a PrSor osztályt ennek megfelelően! (Ötlet: vegyünk fel egy új, privát adattagot az osztályba, ami nemüres prioritásos sor esetén a legnagyobb elem indexe. Módosítsuk ennek megfelelően a metódusokat!)

6.3.2. $A[1..m]$ monoton növekvően rendezett

$MT_{\text{prSorba}}(m), AT_{\text{prSorba}}(m) \in \Theta(m); mT_{\text{prSorba}}(m) \in \Theta(1)$
$T_{\text{maxKivesz}}(m) \in \Theta(1)$, $T_{\max}(m) \in \Theta(1)$:







Vegyük észre, hogy
- **ha** $A[1..m]$ **rendezetlen**, a prSorba(x) eljárás megfelel a Verem típus verembe(x) metódusának, míg
- **ha** $A[1..m]$ **rendezett**, a maxKivesz() és a max() tagfüggvények az ottani veremből() illetve tető() műveletek hasonmásai.

A "régi" műveletek így $\Theta(1)$ időben futnak. Az "újak" viszont lineáris műveletigényűek, ezért hatékonyabb megoldást keresünk.

7. Függvények aszimptotikus viselkedése

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

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

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

7.2 Definíció *Az f AN (aszimptotikusan nemnegatív) függvény, ha elég nagy n-ekre $f(n) \geq 0$.*

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

Egy tetszőleges helyes program futási ideje és tárigénye is nyilvánvalóan, tetszőleges megfelelő mértékbegységben (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[10] 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 nemnegatívak, esetleg aszimptotikusan pozitívak legyenek.

7.4 Jelölések: *Az f, g, h* latin betűkről *ebben a fejezetben feltesszük, hogy $\mathbb{N} \to \mathbb{R}$ típusú, aszimptotikusan nemnegatív (AN) függvényeket jelölnek, míg a φ, ψ* görög betűkről *csak azt tesszük fel, hogy $\mathbb{N} \to \mathbb{R}$ típusú függvényeket jelölnek.*

7.5 Definíció *Az $O(g)$ függvényhalmaz olyan f függvényekből áll, amiket elég nagy n helyettesítési értékekre, megfelelelő pozitív konstans szorzóval jól becsül felülről a g függvény:*
$O(g) = \{f : \exists d \in \mathbb{P}$, *hogy elég nagy n -ekre* $d * g(n) \geq f(n).\}$

7.6 Definíció *Az $\Omega(g)$ függvényhalmaz olyan f függvényekből áll, amiket elég nagy n helyettesítési értékekre, megfelelelő pozitív konstans szorzóval jól becsül alulról a g függvény:*
$\Omega(g) = \{f : \exists c \in \mathbb{P}$, *hogy elég nagy n -ekre* $c * g(n) \leq f(n).\}$

7.7 Definíció *A $\Theta(g)$ függvényhalmaz olyan f függvényekből áll, amiket elég nagy n helyettesítési értékekre, megfelelelő pozitív konstans szorzókkal alulról és felülről is jól becsül a g függvény:*
$\Theta(g) = \{f : \exists c, d \in \mathbb{P}$, *hogy elég nagy n -ekre*
. $c * g(n) \leq f(n) \leq d * g(n).\}$

7.8 Tulajdonság

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

7.9 Tétel *Ha f AN, g pedig AP függvény, akkor*
$f \in O(g) \iff \exists d \in \mathbb{P}$ *és ψ, hogy $\forall n \in \mathbb{N}$ -re*

$$d * g(n) + \psi(n) \geq f(n) \quad \wedge \quad \lim_{n \to \infty} \frac{\psi(n)}{g(n)} = 0$$

[10] vektor mérete, láncolt lista hossza, fa csúcsainak száma stb.

7.10 Tétel *Ha f AN, g pedig AP függvény, akkor*
$f \in \Omega(g) \iff \exists c \in \mathbb{P}$ *és* φ, *hogy* $\forall n \in \mathbb{N}$ *-re*

$$c * g(n) + \varphi(n) \leq f(n) \quad \wedge \quad \lim_{n\to\infty} \frac{\varphi(n)}{g(n)} = 0$$

7.11 Tétel *Ha f AN, g pedig AP függvény, akkor*
$f \in \Theta(g) \iff \exists c, d \in \mathbb{P}$ *és* φ, ψ, *hogy* $\forall n \in \mathbb{N}$ *-re*

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

$$\lim_{n\to\infty} \frac{\varphi(n)}{g(n)} = 0 \quad \wedge \quad \lim_{n\to\infty} \frac{\psi(n)}{g(n)} = 0$$

Ld. még ezzel kapcsolatban az alábbi címen az 1.3. alfejezetet! [4]
<http://people.inf.elte.hu/fekete/algoritmusok_jegyzet/01_fejezet_Muveletigeny.pdf>.

8. Fák, bináris fák

Megállapítottuk, hogy a prioritásos sorok rendezetlen és rendezett tömbös reprezentációjával is lesz olyan művelet, ami lineáris műveletigényű, hiszen rendezetlen tömb esetén a maxKivesz() megvalósításához maximumkeresés, rendezett tömb esetén a pedig a prSorba(x) implementációjához rendezett beszúrás szükséges.

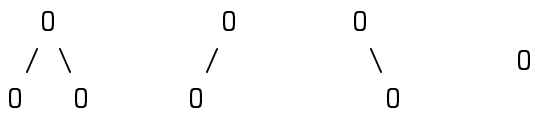
HF: (1) Igazoljuk, hogy a láncolt listás reprezentációk esetén hasonló eredményekre juthatunk! (2) Vegyük észre, hogy egy apró ötlettel a max() függvény műveletigényét rendezetlen tömb illetve lista esetén is $\Theta(1)$-re csökkenthetjük! (A maximum helyét folyamatosan tartsuk nyilván!)

Bináris fák segítségével viszont elérhetjük, hogy a maxKivesz() és a prSorba(x) metódusok maximális futási ideje $\Theta(\lg n)$, a többi műveleté pedig $\Theta(1)$ legyen.

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

A **bináris fák** esetében minden adatelemnek vagy szokásos nevén *csúcsnak* legfeljebb kettő rákövetkezője van: egy *bal* és/vagy egy *jobb* rákövetkezője. Ezeket a csúcs *gyerekeinek* nevezzük. A csúcs a gyerekei *szülője*, ezek pedig egymás *testvérei*. Ha egy csúcsnak nincs gyereke, *levélnek* hívjuk, ha pedig nincs szülője, *gyökér* csúcsnak nevezzük. *Belső csúcs* alatt nem-levél

csúcsot értünk. A fában egy csúcs leszármazottai a gyerekei és a gyerekei leszármazottai. Hasonlóan, egy csúcs ősei a szülője és a szülője ősei. A fákat felülről lefelé szokás lerajzolni: fent van a gyökér, lent a levelek. Egyszerű bináris fák:



Az Ω *üres fának* nincs csúcsa. 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* nevezzük. Jelölése $t \to bal$ illetve $t \to jobb$ (szokás a $bal(t)$ és $jobb(t)$ jelölés is). Ha $*t$-nek nincs bal/jobb gyereke, akkor $t \to bal = \Omega$ illetve $t \to jobb = \Omega$. Ha a gyerek létezik, jelölése $*t \to bal$ illetve $*t \to jobb$.

A t bináris fának ($t = \Omega$ esetén is) *részfája* önmaga. Ha $t \neq \Omega$, részfái még a $t \to bal$ és a $t \to jobb$ részfái is. A t *valódi részfája* f, ha t részfája f, és $t \neq f \neq \Omega$.

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

Megjegyezzzük, 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.

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

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 a $0..r-1$ *szelektorokkal* szokás sorszámozni. Ha egy csúcsnak nincs i-edik gyereke ($i \in 0..r-1$), akkor az i-edik részfa üres.

Így tehát a bináris fa és a *2-áris* fa lényegében ugyanazt jelenti, azzal, hogy itt a $bal \sim 0$ és a $jobb \sim 1$ szelektor-megfeleltetést alkalmazzuk.

Beszélhetünk a fa *szintjeiről*. A gyökér van a *nulladik* szinten. Az i-edik szintű csúcsok gyerekeit az $(i+1)$-edik szinten találjuk. A fa magassága egyenlő a legmélyebben fekvő levelei szintszámával. Az üres fa magassága $h(\Omega) = -1$. Így tetszőleges nemüres t bináris fára:

$$h(t) = 1 + max(h(t \to bal), h(t \to jobb))$$

Az itt tárgyalt fákat *gyökeres fáknak* is nevezik, mert tekinthetők olyan irányított gráfoknak, amiknek az élei a gyökércsúcstól a levelek felé vannak

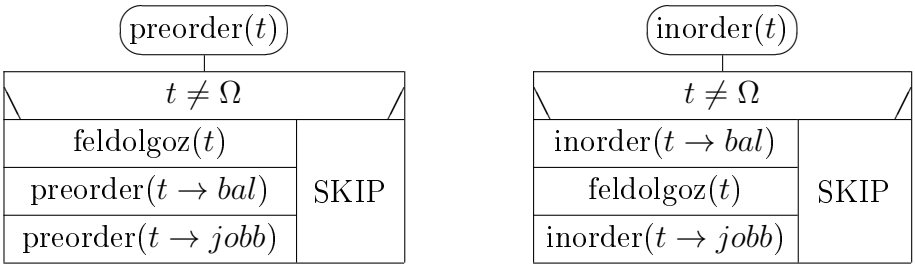
irányítva, a gyökérből minden csúcs pontosan egy úton érhető el, és valamely $r \in \mathbb{N}$-re tetszőleges csúcs kimenő élei a $0..r-1$ szelektorok egy részhalmaza elemeivel egyértelműen[11] van címkézve. (Ezzel szemben a szabad fák összefüggő, körmentes irányítatlan gráfok.)

8.1. (Bináris) fák bejárásai

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

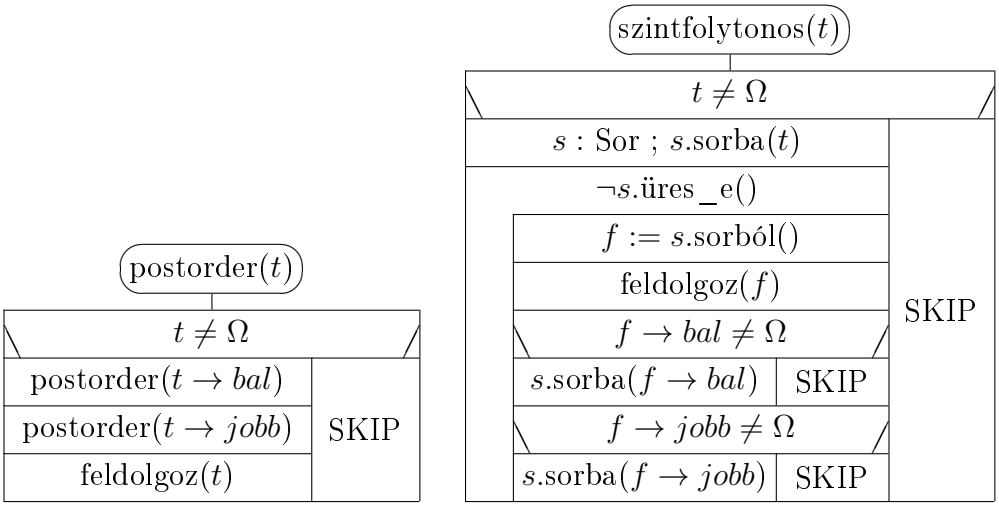
Üres fára mindegyik bejárás az üres program. Nemüres r-áris fákra
- a *preorder* bejárás először a fa gyökerét dolgozza fel, majd sorban bejárja a $0..r-1$-edik részfákat;
- a *postorder* bejárás előbb sorban bejárja a $0..r-1$-edik részfákat, és a fa gyökerét csak a részfák után dolgozza fel;
- az *inorder* bejárás először bejárja a *nulladik* részfát, ezután a fa gyökerét dolgozza fel, majd sorban bejárja az $1..r-1$-edik részfákat;
- a szintfolytonos bejárás 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[12]:



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

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

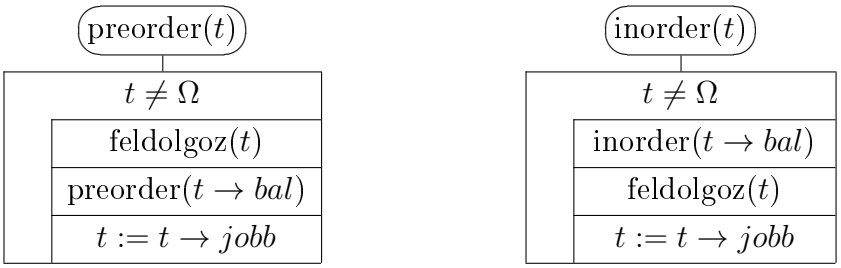


Állítás: $T_{\text{preorder}}(n), T_{\text{inorder}}(n), T_{\text{postorder}}(n), T_{\text{szintfolytonos}}(n) \in \Theta(n)$, ahol n a fa *mérete*, azaz csúcsainak száma.

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 könyen belátható, hogy tetszőleges n csúcsú bináris fának $2n+1$ részfája van. A szintfolytonos bejárás pedig mindegyik csúcsot a ciklus egy-egy végrehajtásával dolgozza fel.

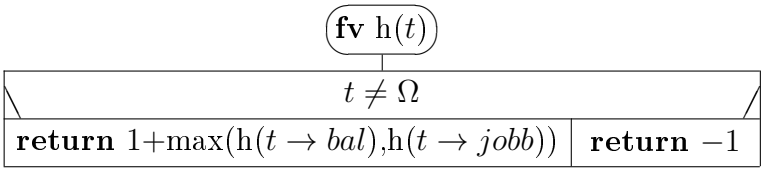
A szintfolytonos bejárás helyessége azon alapszik, hogy egy fa csúcsainak szintfolytonos felsorolásában a korábbi csúcs gyerekei megelőzik a későbbi csúcs gyerekeit. Ez az állítás nyilvánvaló, akár egy szinten, akár különböző szinten van a két csúcs a fában.

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

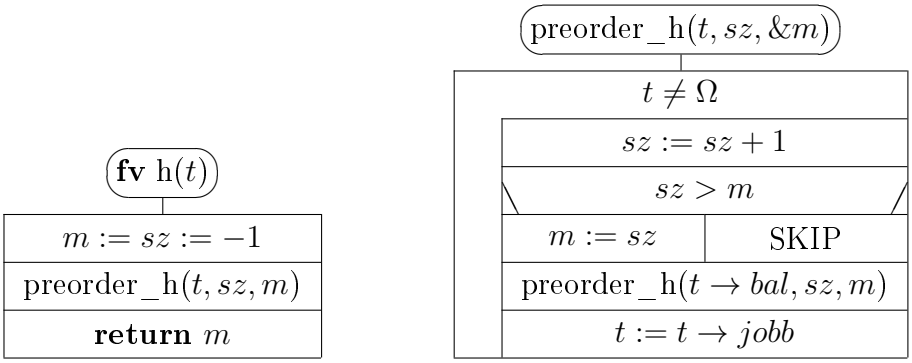


8.1.1. Fabejárások alkalmazása: bináris fa magassága

Természetesen az a legegyszerűbb, ha a definíciót kódoljuk le:



Mint látható, ez lényegében véve egy függvényként kódolt postorder bejárás.[13] 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 (sz) azt tárolja, milyen mélyen (azaz hányadik szinten) járunk a fában. A másik (m) pedig azt, hogy mi a legmélyebb szint, ahol eddig jártunk. Ezután már csak a bejárás és a maximumkeresés összefésülésére van szükség.[14]



8.2. Bináris fák reprezentációi: láncolt ábrázolás

A legtermészetesebb és az egyik leggyakrabban használt a **láncolt ábrázolás**. Az Ω (üres fa) reprezentációja a $\oslash$ (NIL). A bináris fa csúcsait pl. az alábbi osztály objektumaiként ábrázolhatjuk.

[13] 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.

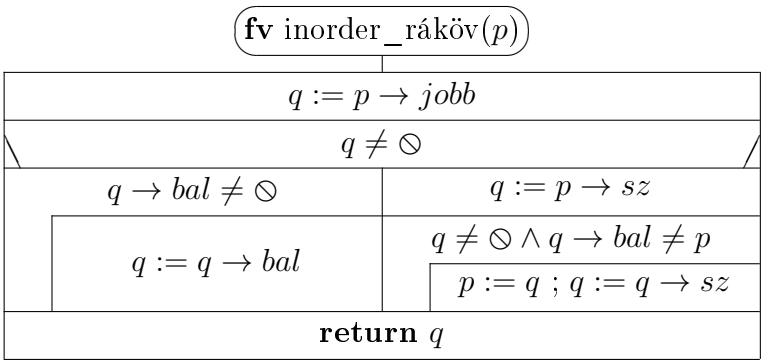
[14] Így a végrehajtáshoz csak egy függvényhívásra, $n+1$ eljáráshívásra és n ciklus-iterációra lesz szükség, míg az előbbi esetben $2n+1$ függvényhívásra, ha a max() függvény hívásait nem számítjuk (ami könnyen kibontható egy szekvencia + elágazássá). Mivel egy ciklus-iteráció a gyakorlatban gyorsabb,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

Csúcs
+ $kulcs$: T // T valamilyen ismert típus + $bal, jobb$: Csúcs*
+ Csúcs() { $bal := jobb := \oslash$ } // egycsúcsú fát képez belőle + Csúcs(x:T) { $bal := jobb := \oslash$; $kulcs := x$ }

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

Csúcs3
+ $kulcs$: T // T valamilyen ismert típus + $bal, jobb, sz$: Csúcs3*
+ Csúcs3(p:Csúcs3*) { $bal := jobb := \oslash$; $sz := p$ } // egy levelet hoz létre + Csúcs3(x:T, p:Csúcs3*) { $bal := jobb := \oslash$; $sz := p$; $kulcs := x$ }

Előfordul például olyan alkalmazás, ahol szükségünk van a bináris fában egy p : Csúcs3* 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.



HF: 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!

8.3. Speciális bináris fák, kupacok

Azokat a bináris fákat, amelyekben minden belső (azaz nem-levél) csúcsnak két gyereke van, *szigorúan bináris fáknak* nevezzük. Ha ez utóbbiaknak minden levele azonos szinten van, *teljes 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 teljes bináris fa csúcsainak száma tehát $1+2+4+...+2^h = 2^{h+1}-1$.

Ha egy teljes 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 $h = \lfloor \lg n \rfloor$. Az alsó szinten levő leveleket elvéve pedig egy $h-1$ mélységű teljes bináris fát kapunk.

HF: Egy bináris fa *méret szerint kiegyensúlyozott*, ha tetszőleges nemüres részfája 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!

Egy majdnem teljes bináris fa *balra tömörített*, ha az alsó szintjén egyetlen levéltől balra sem lehet új levelet beszúrni. Ez azt jelenti, hogy egy vele azonos mélységű teljes 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 a jobb szélső kivételével biztosan két-két gyereke van. Ha a jobb szélső belső csúcsnak csak egy gyereke van, akkor ez bal gyerek. A szint jobb szélén lehetnek levelek. A magasabb szinteken minden csúcsnak két gyereke van.

Egy balra tömörített, majdnem 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 jobboldali részfájában levő kulcsok között viszont nincs semmi nagyságrendi kapcsolat. Az elsőbbségi (prioritásos) sorokat általában kupacok segítségével ábrázoljuk.

Egy balra tömörített, majdnem teljes bináris fát *lyukas kupacnak* nevezünk, ha a fa egyik csúcsa a lyuk, és minden szülő-gyerek párosban a szülő kulcsa nagyobb-egyenlő, mint a gyereke kulcsa, kivéve, ha a páros egyik tagja maga a lyuk. A lyuk kulcsa nemdefiniált, de ha a lyuknak van szülője, akkor annak kulcsa $\geq$ mint a lyuk leszármazottainak kulcsai.

Vegyük észre, hogy egy lyukas kupacban a lyuk kulcsába mindig tudunk olyan értéket írni, hogy szabályos kupacot kapjunk! A lyukas kupacok az

alapvető kupacműveletek (új elem beszúrása, maximum kivétele) során alakulnak ki, amiket aztán a lyuk megfelelő mozgatásával állítunk helyre. (Ld. a "Kupacok és elsőbbségi sorok" fejezetet!)

Egy balra tömörített, majdnem 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!)

8.4. Bináris fák aritmetikai ábrázolása

A balra tömörített, majdnem teljes bináris fákat, speciálisan a kupacokat, szokás szintfolytonosan, egy vektorban ábrázolni. Ha pl. egy $A[1..n]$ tömb első m elemét használjuk, akkor az i indexű csúcs gyerekeinek indexei $2i$, illetve $2i+1$, feltéve, hogy a gyerekek léteznek, azaz $2i \leq m$, illetve $2i+1 \leq m$. A baloldali gyerek indexe azért $2i$, mert az i-edik csúcsot a szintfolytonos ábrázolásban $(i-1)$ csúcs előzi meg. Az i-edik csúcs baloldali gyerekét tehát megelőzi ennek az $(i-1)$ csúcsnak a $(2i-2)$ gyereke, plusz a gyökércsúcs, aminek nincs szülője. Ez összesen $(2i-1)$ csúcs, és így az i-edik csúcs baloldali gyerekének indexe $2i$, a jobboldalié pedig $2i+1$.

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

A fentiekből adódik, hogy az $1..k$ sorszámú csúcsok szülei az $1..\lfloor \frac{k}{2} \rfloor$ sorszámú csúcsok:
- 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, balra tömörített, majdnem 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 előző szakaszban belátott állítás szerint egy n csúcsú, balra tömörített, majdnem teljes bináris fának $\lceil \frac{n}{2} \rceil$ levele van.

Jelölje a továbbiakban n_d tetszőleges n méretű, balra tömörített, majdnem teljes bináris fa d magasságú részfáinak számát, $n_{\geq d}$ pedig a fa legalább d magasságú részfáinak számát, ahol $1 \leq d \leq h = \lfloor \lg n \rfloor$, azaz h az eredeti fa magassága! Ekkor $n_{\geq 1} = fele(n)$, hiszen a legalább egy magasságú részfák gyö-

kércsúcsai 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

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

HF: Bizonyítsuk be, hogy ha az A tömböt nullától indexeljük, akkor $A[i]$ gyerekei $A[2i+1]$ illetve $A[2i+2]$, a $2i+1 < m$, illetve a $2i+2 < m$ feltétellel, $A[j]$ szülője pedig $j > 0$ esetén $A[\lfloor \frac{j-1}{2} \rfloor]$!

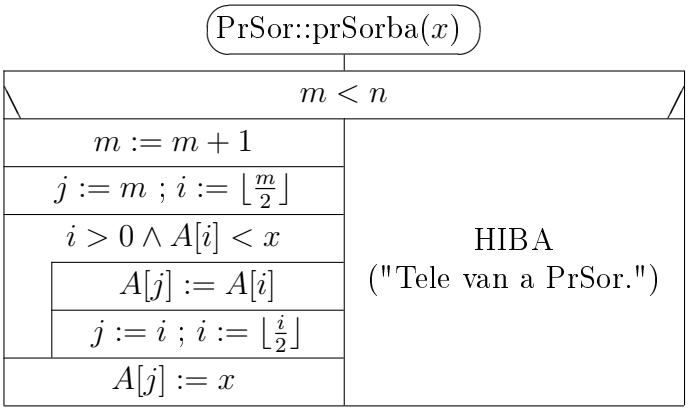
8.5. Kupacok és elsőbbségi (prioritásos) sorok

PrSor
- $A[1..n] : T$ // T ismert típus, n globális konstans - $m : 0..n$ // m a PrSor aktuális mérete

+ PrSor() $\{m := 0\}$ // üresre inicializálja a PrSort
// destruktort itt nem kell definiálni
+ üresPrSor() $\{m := 0\}$ // kiüríti a PrSort
+ prSorba($x : T$) // beteszi x-et a prSorba
+ **fv** maxKivesz() : T // kiveszi és visszaadja a PrSor maximális elemét
+ **fv** max() : T // megnézi és visszaadja a PrSor maximális elemét
+ **fv** tele_e() : $\mathbb{L}$ $\{$**return** $m = n\}$
+ **fv** üres_e() : $\mathbb{L}$ $\{$**return** $m = 0\}$

Ha az $A[1..m]$ résztömb egy kupac aritmetikai ábrázolása,
$MT_{\text{prSorba}}(m) \in \Theta(\lg m)$, $mT_{\text{prSorba}}(m) \in \Theta(1)$, $MT_{\text{maxKivesz}}(m) \in \Theta(\lg m)$, $mT_{\text{maxKivesz}}(m) \in \Theta(1)$, $T_{\max}(m) \in \Theta(1)$, mivel az alábbi "prSorba" és "lesüllyeszt" eljárások fő ciklusa maximum annyiszor fut le, amennyi a fa magassága.

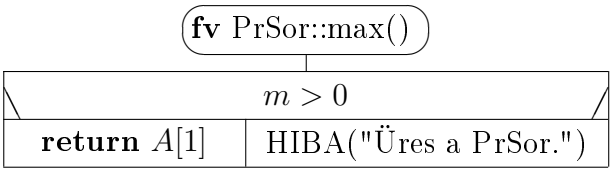
A prSorba(x) esetében a szintfolytonosan első üres helyen egy lyukat ($A[j]$) kapcsolunk a kupachoz. Így egy *lyukas kupacot* kapunk, aminek egyenlőre egy levele a lyuk. Azután a lyukat addig cserélgetjük – mindig az aktuális szülőjével ($A[i]$) – amíg van szülője, és $x >$ mint a szülő. Így a lyuk felfelé mozog a kupacban, és $x >$ mint a lyuk leszármazottai. Ha a lyuk felér a gyökérbe, vagy x már $\leq$ mint a lyuk szülője, akkor beletesszük x-et, és így helyreáll a kupac.

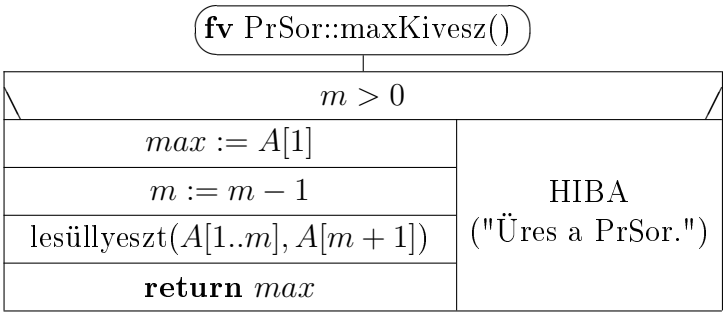


$MT_{\text{prSorba}}(m) \in \Theta(\lg m)$; hiszen a ciklus legfeljebb annyiszor fut le, amennyi a fa magassága, azaz (m input értékéhez viszonyítva) $\lfloor \lg(m+1) \rfloor$-szer, amiből $\lg m \leq MT_{\text{prSorba}}(m) = \lfloor \lg(m+1) \rfloor + 1 \leq \lg m + 2$.

$mT_{\text{prSorba}}(m) \in \Theta(1)$, ha ugyanis x elég kicsi, akkor a ciklus egyszer sem fut le, és innét $mT_{\text{prSorba}}(m) = 1$.

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

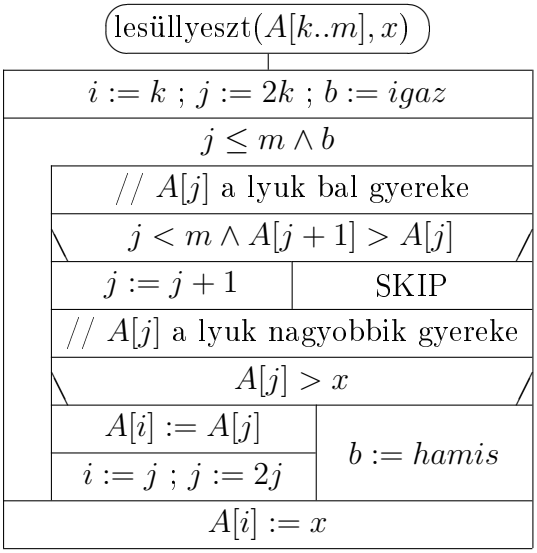




A maxKivesz() metódus a kupac gyökerében, $A[1]$-ben hoz létre egy lyukat. Ezután a szintfolytonosan utolsó elemet levágja a lyukas kupacról, majd a "lesüllyeszt" eljárás segítségével (amit mindig $k = 1$-gyel hív meg) a lyukat ($A[i]$) addig süllyeszti lefelé, amíg a levágott elem (x) bele nem illik. Ennek

során a lyukat mindig az aktuálisan nagyobb gyerekével cseréli meg, amíg még a levélszint fölött van, és a lyuk nagyobbik gyereke $> x$. Ilyen módon a ciklus során $x <$ mint a lyuk ősei. Amikor a ciklus megáll, vagy leért a lyuk a levélszintre, vagy $x \geq$ mint a lyuk gyerekei. Ekkor x-et a lyukba téve szabályos kupacot kapunk.

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



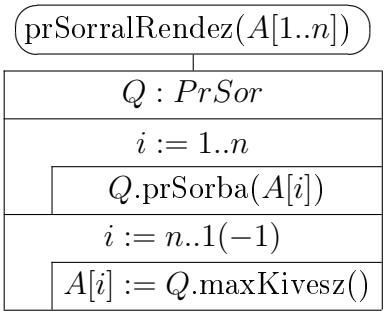
A maxKivesz() metódus műveletigényének meghatározásához először megvizsgáljuk a lesüllyeszt() (rövidítve „le") eljárást. Ha a k gyökerű lyukas kupac magassága h, akkor $MT_{\text{le}}(h) = h + 1$, ugyanis a ciklus legfeljebb h iterációt végez, és $1 \leq mT_{\text{le}}(h) \leq 2$, mert lehet, hogy a ciklus legfeljebb egyet iterál. Speciálisan $k = 1$ esetére: $\lg m \leq MT_{\text{le1}}(m) = h + 1 = \lfloor \lg m \rfloor + 1 \leq \lg m + 1$.

Innét $\lg m \leq \lg(m-1) + 1 \leq MT_{\text{maxKivesz}}(m) \leq \lg(m-1) + 2 \leq \lg m + 2$.
Ebből $MT_{\text{maxKivesz}}(m) \in \Theta(\lg m)$.

$mT_{\text{maxKivesz}}(m) \in \Theta(1)$, hiszen
$2 = 1 + 1 \leq mT_{\text{maxKivesz}}(m) = 1 + mT_{\text{le1}}(m-1) \leq 1 + 2 = 3$.

8.5.1. Rendezés elsőbbségi sorral

A fenti elsőbbségi sor segítségével könnyen és hatékonyan rendezhetünk tömböket:



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

Maximális műveletigénye $\Theta(n \lg n)$. Ha ugyanis az input vektor 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 $\lfloor \lg n \rfloor - 1$ mélységű a fa, és a leendő levelek száma $\lceil \frac{n}{2} \rceil$. Ekkor tehát csak a levelek beszúrásának futási ideje $\Omega(n \lg n)$, így a teljes futási idő is. Az előbbi $O(n \lg n)$ korláttal együtt adódik az állítás.

A fenti rendezés tehát, maximális műveletigényét tekintve, aszimptotikusan az eddig ismert legjobb rendezésünkkel, az összefuttató rendezéssel (mergesort) egyenrangú. Hátránya, hogy a rendezendő vektorral azonos méretű munkamemóriát igényel, a prioritásos sorban tárolt kupac számára. (Az összefuttató rendezés vektoros változata kb. feleakkorát igényel, bár így nagyságrendileg mindkét rendezésnek $\Theta(n)$ a tárigénye, míg az összefuttató rendezés láncolt listás változata csak $\Theta(\lg n)$ munkatárat használ, mégpedig a rekurzív hívások adminisztrálásához.) A továbbiakban megismerjük a kupacrendezést (heapsort), ami egyrészt helyben rendez, azaz csak $\Theta(1)$ munkatárat igényel, másrészt a rendezés első fázisát, a kupacépítést optimalizálja, így a gyakorlatban a fenti rendezésnél gyorsabb, bár aszimptotikusan a kupacrendezés maximális műveletigénye is $\Theta(n \lg n)$.

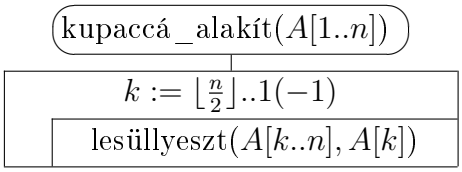
8.6. Kupacrendezés (Heapsort)

A kupacrendezés a fenti prSorralRendez($A[1..n]$) optimalizálása.

Egyrészt az algoritmust *helyben rendezővé* alakítjuk: az $A[1..n]$ 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.

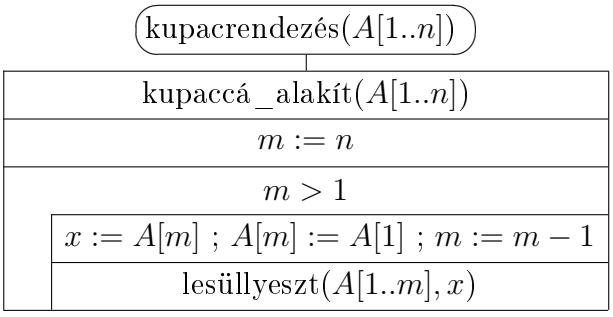
Másrészt a kupac felépítését optimalizáljuk, ami a fenti esetben magában véve is $O(n \lg n)$ műveletigényű, maximális futási ideje pedig $\Theta(n \lg n)$.

Először tehát kupaccá alakítjuk a tömböt, most lineáris műveletigénnyel:



A fenti eljárás magyarázatához: Eredetileg az $A[1..n]$ tömb, mint balra tömörített majdnem teljes bináris fa magassága $h = \lfloor \lg n \rfloor$. 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 $A[1]$-et süllyesztjük le az alatta lévő csonka kupacba, és ezzel az $A[1..n]$ tömb kupaccá alakítása befejeződött. Annak érdekében, hogy a szintekkel ne kelljen külön foglalkozni, a fenti eljárásban a lesüllyesztéseket a szintfolytonosan utolsó belső csúccsal kezdtük, és innét haladtunk szintfolytonosan visszafelé.

Ezután a teljes rendezés:

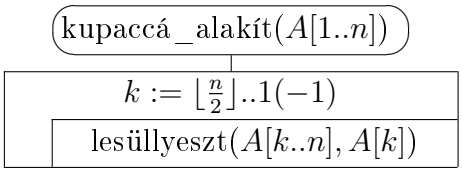


A kupaccá alakítás után tehát ezt ismételjük: A kupacról levágjuk a szintfolytonosan utolsó csúcsát, a helyére tesszük a kupac maximumát, majd a levágott csúcsot (x) lesüllyesztjük az $A[1..m]$, gyökerénél lyukas kupacba. Így az előbbinél eggyel kisebb méretű kupacot kapunk. Ezt a ciklusmagot addig ismételjük, amíg a kupac mérete nagyobb, mint egy. Így az $A[1..n]$ tömbben visszafelé megkapjuk az első, második stb. maximumokat, és végül az $A[1]$-ben a minimumot, azaz a tömb rendezve lesz.

8.6.1. A kupacrendezés műveletigénye

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

A kupaccá alakításról hasonlóan látható, hogy maga is $O(n \lg n)$, de ennél finomabb becsléssel belátjuk, hogy $\Theta(n)$ műveletigényű. Ettől a teljes futási idő továbbra is $O(n \lg n)$, hiszen a szekvenciában ebből a szempontból az aszimptotikusan nagyobb műveletigényű programrész dominál, de a prSorralRendez($A[1..n]$) eljárás kupacépítő ciklusához képest hatékonyabb kupaccá alakítással a gyakorlatban így is jelentős futási időt takaríthatunk meg.



Szemléletesen fogalmazva, a prSorralRendez($A[1..n]$) 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 kupaccá_alakít($A[1..n]$) műveletigénye $\Theta(n)$, először belátjuk, hogy maximális műveletigényére $MT_k(n) \leq 2n+1$ teljesül.

Idézzük fel ehhez, hogy a 8.4. alfejezet végén az $A[1..n]$ vektorban tárolt balra tömörített majdnem teljes fa d magasságú részfáinak számát n_d-vel, a legfeljebb d magasságú részfák számát pedig $n_{\geq d}$-vel jelöltük. Bevezettük még a $fele(n) = \lfloor \frac{n}{2} \rfloor$ függvényt, a fa magasságára pedig a $h = \lfloor \lg n \rfloor$ jelölést, és a

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

összefüggést kaptuk. A kupaccá alakítás $MT_k(n)$ maximális műveletigényét az eljárás meghívása, a ciklus $\lfloor \frac{n}{2} \rfloor$ iterációja és a lesüllyesztő eljárás végrehajtásai adják. Tetszőleges d magasságú részfára a lesüllyesztő eljárás műveletigénye legfeljebb $d+1$. Az összes, n_d darab d magasságú részfákra

tehát a lesüllyesztések műveletigénye összesen legfeljebb $n_d * (d+1)$. Mivel a lesüllyesztő eljárás hívásai során $d \in 1..h$, a kupaccá alakítás alatt a lesüllyesztések műveletigényének összege legfeljebb $\sum_{d=1}^{h} n_d * (d+1)$. Innét

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

Vegyük most figyelembe, hogy $\sum_{d=1}^{h} n_d = \sum_{m=1}^{h} n_m = n_{\geq 1} = fele(n) = \left\lfloor \frac{n}{2} \right\rfloor$, és hogy $\sum_{d=1}^{h} d * n_d$ az alábbi alakba írható:
n_1+
$n_2 + n_2+$
$n_3 + n_3 + n_3+$
$\ldots$
$n_h + n_h + n_h + \ldots + n_h \quad$ (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

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

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

$$MT_k(n), mT_k(n) \in \Theta(n)$$

adódik.

Felvetődhet a kérdés, hogy finomabb becsléssel nem lehetne-e a kupacrendezésben a kupaccá alakítás utáni ciklusról is belátni, hogy $\Theta(n)$ a műveletigénye, amiből következne, hogy az egész kupacrendezés futási ideje is $\Theta(n)$. A válasz sajnos nem, mert később, az összehasonlító rendezések alaptételei segítségével be fogjuk tudni látni, hogy a kupacrendezés maximális és átlagos futási ideje $\Theta(n \lg n)$.

HF: Lássuk be, hogy a kupacrendezés minimális futási ideje $\Theta(n)$, ami például akkor áll elő, amikor az input tömb minden eleme egyenlő!

8.7. Bináris keresőfák

Egy bináris fát *keresőfának* nevezünk, ha minden nemüres r részfájára és annak a gyökerében lévő y kulcsra igazak az alábbi követelmények:
- Ha x egy tetszőleges csúcs kulcsa az r bal részfájából, akkor $x < y$.
- Ha z egy tetszőleges csúcs kulcsa az r jobb részfájából, akkor $z > y$.

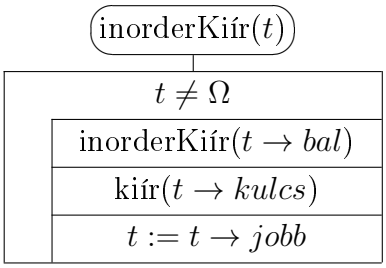
Egy bináris fát *rendezőfának* nevezünk, ha minden nemüres r részfájára és annak a gyökerében lévő y kulcsra igazak az alábbi követelmények:
- Ha x egy tetszőleges csúcs kulcsa az r bal részfájából, akkor $x \leq y$.
- Ha z egy tetszőleges csúcs kulcsa az r jobb részfájából, akkor $z \geq y$.

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 $H(\Omega) = \{\}$, illetve $H(t) = H(t \to bal) \cup \{t \to kulcs\} \cup H(t \to jobb)$, amennyiben $t \neq \Omega$.

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



HF: 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 házi feladat állításának 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 **Csúcs** típusúak, és az Ω üres fát a $\oslash$ pointer reprezentálja. (Ld. a **Bináris fák reprezentációi: láncolt ábrázolás** fejezetet!)

8.8. 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 visszadásával 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 műveletek:
- **fv** keres(t, k) : ha $k \in H(t)$, akkor a k kulcsú csúcsra mutató pointerrel tér vissza, különben a $\oslash$ hivatkozással,
- beszúr(t, k) : a $H(t) := H(t) \cup \{k\}$ absztrakt művelet megvalósítása,
- **fv** min(t) : a $t \neq \oslash$ fában a minimális kulcsú csúcsra áll,
- minKivesz$(t, minp)$: a $t \neq \oslash$ fából kivesszük a legkisebb kulcsot tartalmazó csúcsot.
- töröl(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 keres(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.

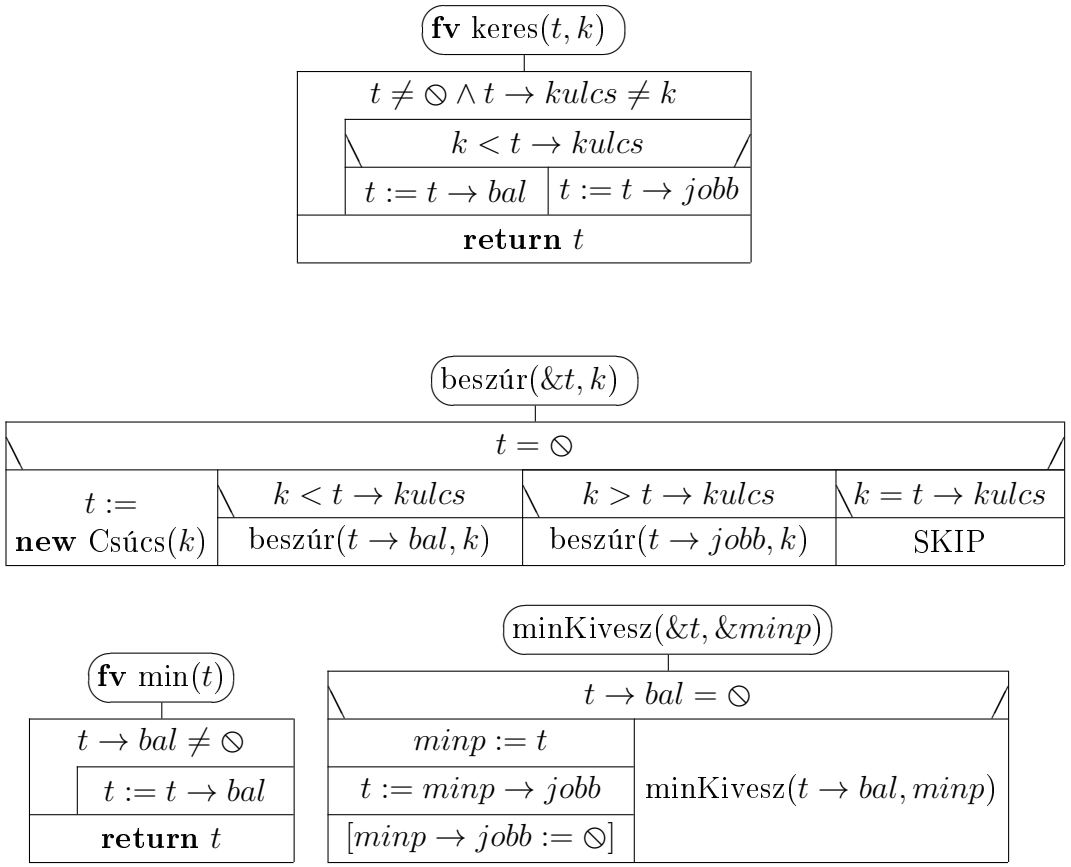
A beszúr(t, k) eljárás is megkeresi a t fában a k kulcs helyét. Ha ott egy üres részfát talál, akkor az üres részfa helyére tesz egy új levélcsúcsot, k kulccsal.

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

A minKivesz$(t, minp)$ eljárás $minp$-ben a t nemüres fa "bal alsó" (a legkisebb kulcsot tartalmazó) csúcsára mutató pointerrel tér vissza, de még előtte a csúcsot kifűzi a fából, azaz a csúcshoz tartozó részfa helyére teszi a csúcs jobboldali részfáját.

A töröl(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 minKivesz eljárás segítségével *kiveszi* a jobboldali részfából a minimális kulcsú csúcsot, és a k kulcsú csúcs helyére teszi, hiszen ez a kulcs a baloldali részfa kulcsainál nagyobb, a jobboldali részfa maradék kulcsainál pedig kisebb. (Vegyük észre, hogy a (2) esetben a baloldali részfából a maximális kulcsú csúcsot is kivehetnénk!)

HF: Írjuk meg a $t \neq \oslash$ fából a maximális kulcs kiolvasása / kivétele műveleteket. 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!



Mj.: A minKivesz$(t, minp)$ eljárásban a „$minp \to jobb := \oslash$” utasítás arra szolgál, hogy a kifűzött csúcs ne tartalmazzon hivatkozást a fa egy részfájára. Az alábbi „töröl(t, k)” eljárásból, pontosabban annak „gyökeretTöröl(t)” segédeljárásából való meghívás esetén azonban ez az utasítás felesleges.

töröl$(\&t, k)$

$t \neq \oslash$			
$k < t \to kulcs$	$k > t \to kulcs$	$k = t \to kulcs$	SKIP
töröl$(t \to bal, k)$	töröl$(t \to jobb, k)$	gyökeretTöröl(t)	

gyökeretTöröl$(\&t)$

$t \to bal = \oslash$	$t \to jobb = \oslash$	$t \to bal \neq \oslash \wedge t \to jobb \neq \oslash$
$p := t$	$p := t$	minKivesz$(t \to jobb, p)$
$t := p \to jobb$	$t := p \to bal$	$p \to bal := t \to bal$; $p \to jobb := t \to jobb$
delete p	**delete** p	[$t \to bal := t \to jobb := \oslash$;] delete t ; $t := p$

Mj.: A „gyökeretTöröl(t)” segédeljárásban a „$t \to bal := t \to jobb := \oslash$” utasítás a **Csúcs** osztály jelenlegi definíciója mellett felesleges. Ha azonban kiegészítenénk az osztályt egy olyan destruktorral, ami törli a csúcs gyerekeit (és ezért végső soron – rekurzívan – a csúcshoz tartozó részfát), akkor szükséges lenne, annak érdekében, hogy a „delete t” utasítás csak a $(*t)$ csúcsot törölje.

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 $\lfloor \lg n \rfloor$ (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. Ezért az eddig tárgyalt keresőfákat pontosabban *véletlen építésű bináris keresőfáknak* nevezik.

Szerencsére az a tapasztalat, hogy ha a kulcsok sorrendje, amikkel a beszúrásokat és törléseket végezzük, véletlenszerű, akkor a fa magassága általában $O(\lg n)$ (bizonyítható, hogy nagy n-ekre átlagosan kb. $1,4 \lg n$), és így a beszúrás és a törlés sokkal hatékonyabb, mintha az adathalmaz tárolására

tömböket vagy láncolt listákat használnánk, ahol csak $O(n)$ hatékonyságot tudnánk garantálni.

HF: Rendezetlen illetve rendezett tömbök esetén mely műveletekre tudnánk az $MT(n) \in \Theta(n)$ hatékonyságot javítani? Mennyire? Láncolt listák esetén?

HF: Igaz-e, hogy véletlen építésű bináris keresőfák esetén a fenti műveletek bármelyikére $mT(n) \in \Theta(1)$?

HF*: Írjunk olyan eljárást, ami egy szigorúan monoton növekvő vektorból majdnem teljes bináris keresőfa másolatot készít, $O(n)$ futási idővel!

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 $O(\lg n)$ 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 $O(\lg n)$, és a keresés, beszúrás, törlés műveleteinek futási ideje a fa magasságával arányos, azaz szintén $O(\lg n)$. 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 ez utóbbi két keresőfa típus tárgyalásával folytatjuk.

8.9. AVL fák

Az AVL fák kiegyensúlyozásának szabályaival kapcsolatos ábrák [2]-ben találhatók: <http://aszt.inf.elte.hu/~asvanyi/ad/AVL-tree_balancing.pdf >.

Az *AVL fák* magasság szerint kiegyensúlyozott bináris keresőfák. Egy bináris fa *magasság szerint kiegyensúlyozott*, ha minden csúcsa kiegyensúlyozott. (Mostantól *kiegyensúlyozott* alatt *magasság szerint kiegyensúlyozottat* értünk.) Egy bináris fa egy $(*p)$ csúcsa *kiegyensúlyozott*, ha a csúcs $(p \to s)$ egyensúlyára $|p \to s| \leq 1$. A $(*p)$ csúcs egyensúlya definíció szerint:

$$p \to s = h(t \to jobb) - h(t \to bal)$$

Az AVL fákat láncoltan reprezentáljuk. A csúcsokban az s egyensúly attribútumot expliciten tároljuk. (A másik lehetőség a két részfa magasságainak tárolása lenne.) A Csúcs osztályt tehát a következőképpen módosítjuk:

Csúcs
+ *kulcs* : T // T valamilyen ismert típus + $s : -1..1$ // a csúcs egyensúlya + *bal*, *jobb* : Csúcs*
+ Csúcs() { *bal* := *jobb* := ⊘ ; $s := 0$ } // egycsúcsú fát képez belőle + Csúcs(x:T) { *bal* := *jobb* := ⊘ ; $s := 0$; *kulcs* := x }

Egyenlőre részletes bizonyítás nélkül közöljük az alábbi eredményt:

Tétel: Tetszőleges n csúcsú nemüres AVL fa h magasságára:

$$\lfloor \lg n \rfloor \leq h \leq 1,45 \lg n, \quad \text{azaz} \quad h \in \Theta(\lg n)$$

A bizonyítás vázlata: Először a h magasságú, nemüres KBF-ek (kiegyensúlyozott, bináris fák) n méretére adunk alsó és felső becslést. Az $n < 2^{h+1}$ becslésből azonnal adódik $\lfloor \lg n \rfloor \leq h$. Másrészt meghatározzuk a h mélységű, legkisebb méretű KBF-ek csúcsainak f_h számát. Erre kapjuk, hogy $f_0 = 1$, $f_1 = 2$, $f_h = 1 + f_{h-1} + f_{h-2} \quad (h \geq 2)$. Ezért az ilyen fákat *Fibonacci fáknak hívjuk.* Mivel tetszőleges h magasságú KBF n méretére $n \geq f_h$, némi matematikai ügyességgel kaphatjuk a $h \leq 1,45 \lg n$ egyenlőtlenséget.

Mivel az AVL fák magassága $\Theta(\lg n)$, ezért a bináris keresőfák keres(t, k) és min(t) függvényeire t AVL fa esetén automatikusan $MT(n) \in \Theta(\lg n)$, ahol $n = |t|$.

A beszúr(t, k), a töröl(t, k) és a minKivesz$(t, minp)$ eljárások azonban változtatják a fa alakját. Így elromolhat a kiegyensúlyozottság, és már nem garantált a fenti műveletigény. Ennek elkerülésére ezeket az eljárásokat úgy módosítjuk, hogy minden egyes rekurzív eljáráshívás után ellenőrizni fogjuk, hogyan változott a megfelelő részfa magassága, és ez hogyan befolyásolta a felette levő csúcs kiegyensúlyozottságát, szükség esetén helyreállítva azt. Ez minden szinten legfeljebb konstans mennyiségű extra műveletet fog jelenteni, és így a kiegészített eljárások futási ideje megtartja az $MT(n) \in \Theta(\lg n)$ (ahol $n = |t|$) nagyságrendet.

A példákhoz a *(bal_ részfa gyökér jobb_ részfa)* jelölést vezetjük be, ahol az üres részfákat elhagyjuk.

Például a ((2)4((6)8(10))) bináris keresőfa gyökere a 4; bal részfája a (2), ami egyetlen levélcsúcsból (és annak üres bal és jobb részfáiból) áll; jobb részfája a ((6)8(10)), aminek gyökere a 8, bal részfája a (6), jobb részfája a (10). *Az így ábrázolt fák inorder bejárása a zárójelek elhagyásával adódik.* A belső csúcsok egyensúlyait *ks* formában jelöljük, ahol k a csúcsot azonosító kulcs, s pedig a csúcs egyensúlya, a 0:∘, 1:+, 2:++, -1:−, -2:−− megfeleltetéssel. Mivel a levélvsúcsok egyensúlya mindig nulla, a leveleknél nem jelöljük az egyensúlyt. A fenti AVL fa például a megfelelő egyensúlyokkal a következő: ((2)4+((6)8∘(10)))

Az AVL fák műveletei során a kiegyensúlyozatlan részfák kiegyensúlyozásához *forgatásokat* fogunk használni. Az alábbi forgatási sémákban görög kisbetűk jelölik a részfákat (amelyek minden esetben AVL fák lesznek), latin nagybetűk pedig a csúcsok kulcsait:

Balra forgatás (Left rotation): $(\alpha\ \mathrm{T}\ (\beta\ \mathrm{J}\ \gamma)) \to ((\alpha\ \mathrm{T}\ \beta)\ \mathrm{J}\ \gamma)$
Jobbra forgatás (Right rotation): $((\alpha\ \mathrm{B}\ \beta)\ \mathrm{T}\ \gamma) \to (\alpha\ \mathrm{B}\ (\beta\ \mathrm{T}\ \gamma))$

Vegyük észre, hogy a fa inorder bejárását egyik forgatás sem változtatja, így a bináris keresőfa tulajdonságot is megtartják. Mint látni fogjuk, a kiegyensúlyozatlan részfák kiegyensúlyozása minden esetben egy vagy két forgatásból áll. *Ezért a bináris keresőfa tulajdonságot a kiegyensúlyozások is megtartják.*

8.10. AVL fák: beszúrás

Nézzük most először a bináris keresőfák beszúr(t, k) eljárását! Az 1 beszúrásával a ((2)4+((6)8∘(10))) AVL fából az (((1)2−)4∘((6)8∘(10))) AVL fa adódik, a 3 beszúrásával pedig a ((2+(3))4∘((6)8∘(10))) AVL fa. A 7 beszúrásával azonban a ((2)4++((6+(7))8−(10))) fát kapjuk, a 9 beszúrásával pedig a ((2)4++((6)8+((9)10−))) fát. Mindkettő bináris keresőfa, de már nem AVL fa, mivel a 4++ csúcs nem kiegyensúlyozott.

A legutolsó esetben a bináris keresőfa könnyen kiegyensúlyozható, ha meggondoljuk, hogy az a következő sémára illeszkedik, amiben görög kisbetűk jelölik a kiegyensúlyozott részfákat, latin nagybetűk pedig a csúcsok kulcsait:
$(\alpha\ \mathrm{T}{+}{+}\ (\beta\ \mathrm{J}{+}\ \gamma))$, ahol T=4, α=(2), J=8, β=(6) és γ=((9)10−).

A kiegyensúlyozáshoz szükséges transzformáció a fa *balra forgatása* [2]:

$$(\alpha\ \mathrm{T}{+}{+}\ (\beta\ \mathrm{J}{+}\ \gamma)) \to ((\alpha\ \mathrm{T}{\circ}\ \beta)\ \mathrm{J}{\circ}\ \gamma)$$

A fenti példán ez a következőt jelenti:
((2) 4++ ((6)8+((9)10−))) → (((2)4∘(6)) 8∘ ((9)10−))

A transzformáció helyességének belátásához bevezetjük a $h = h(\alpha)$ jelölést. Ebből a kiinduló fára a T++ és J+ egyensúlyok miatt
$h((\beta\ \mathrm{J}\ \gamma)) = h+2$, $h(\gamma) = h+1$ és $h(\beta) = h$ adódik, innét pedig az eredmény fára $h(\alpha) = h = h(\beta)$ miatt T$\circ$; továbbá $h((\alpha\ \mathrm{T}\circ\ \beta)) = h+1 = h(\gamma)$ miatt J$\circ$ adódik.

Az eredeti ((2) 4+ ((6)8$\circ$(10))) fába a 7 beszúrásával adódó
((2) 4++ ((6+(7))8$-$(10))) kiegyensúlyozatlan bináris keresőfa kiegyensúlyozásával pedig (((2)4$-$) 6$\circ$ ((7)8$\circ$(10))) adódik, amire az

$$(\ \alpha\ \mathrm{T}{+}{+}\ ((\beta\ \mathrm{B}{-}\circ{+}\ \gamma)\ \mathrm{J}{-}\ \delta)\ ) \to ((\alpha\ \mathrm{T}\circ\circ{-}\ \beta)\ \mathrm{B}\circ\ (\gamma\ \mathrm{J}{+}\circ\circ\ \delta))$$

séma [2] szolgál, ahol α, β, γ és δ AVL fák, és a három jelből álló egyensúlyok sorban három esetet jelentenek úgy, hogy a bal oldalon az i-edik alternatívának a jobb oldalon is az i-edik eset felel meg ($i \in \{1;2;3\}$).

A fenti séma a példában T=4, $\alpha = (2)$, J=8, $\delta = (10)$, B=6, + egyensúllyal (harmadik eset), $\beta = \Omega$ és $\gamma = (7)$ helyettesítéssel alkalmazható.

A transzformációt *kettős forgatásnak* is tekinthetjük: Az
(α T ((β B γ) J δ)) fára először a J csúcsnál alkalmaztunk egy jobbra forgatást, aminek eredményeképpen az (α T (β B (γ J δ)) bináris keresőfát kaptuk, majd az eredmény fát balra forgattuk, ami az ((α T β) B (γ J δ)) AVL fát eredményezte.

A kettős forgatás helyességének ellenőrzéséhez kiszámítjuk az eredmény fa egyensúlyait. Most is bevezetjük a $h = h(\alpha)$ jelölést. Mivel a kettős forgatás előtti fában T++, azért $h(((\beta\ \mathrm{B}\ \gamma)\ \mathrm{J}\ \delta)) = h+2$. Innét J$-$ miatt
$h((\beta\ \mathrm{B}\ \gamma)) = h+1$ és $h(\delta) = h$. Most B lehetséges egyensúlyai szerint három eset van.
(1) B$-$ esetén $h(\beta) = h$ és $h(\gamma) = h-1 \Rightarrow$ az eredmény fában T$\circ$ és J+.
(2) B$\circ$ esetén $h(\beta) = h$ és $h(\gamma) = h \Rightarrow$ az eredmény fában T$\circ$ és J$\circ$.
(3) B+ esetén $h(\beta) = h-1$ és $h(\gamma) = h \Rightarrow$ az eredmény fában T$-$ és J$\circ$.
Mindhárom esetben $h((\alpha\ \mathrm{T}\ \beta)) = h+1 = h((\gamma\ \mathrm{J}\ \delta))$, így B$\circ$.
Ezzel beláttuk a kettős forgatási séma helyességét.

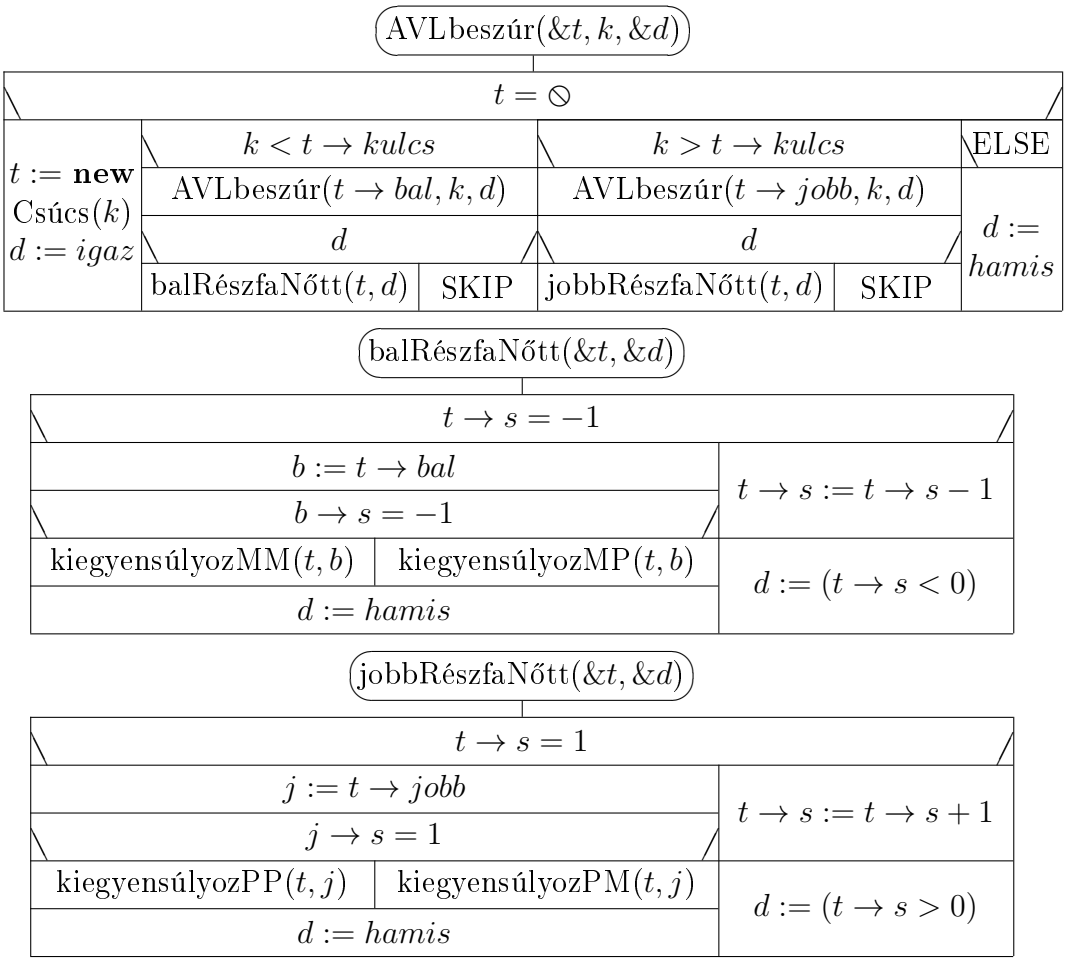
Vegyük észre, hogy a fentiek alapján, a B csúcsnak a kettős forgatás előtti egyensúlyát s-sel, a T és a J csúcsoknak pedig a kettős forgatás utáni egyensúlyát rendre s_t-vel, illetve s_j-vel jelölve a következő összefüggéseket írhatjuk fel:

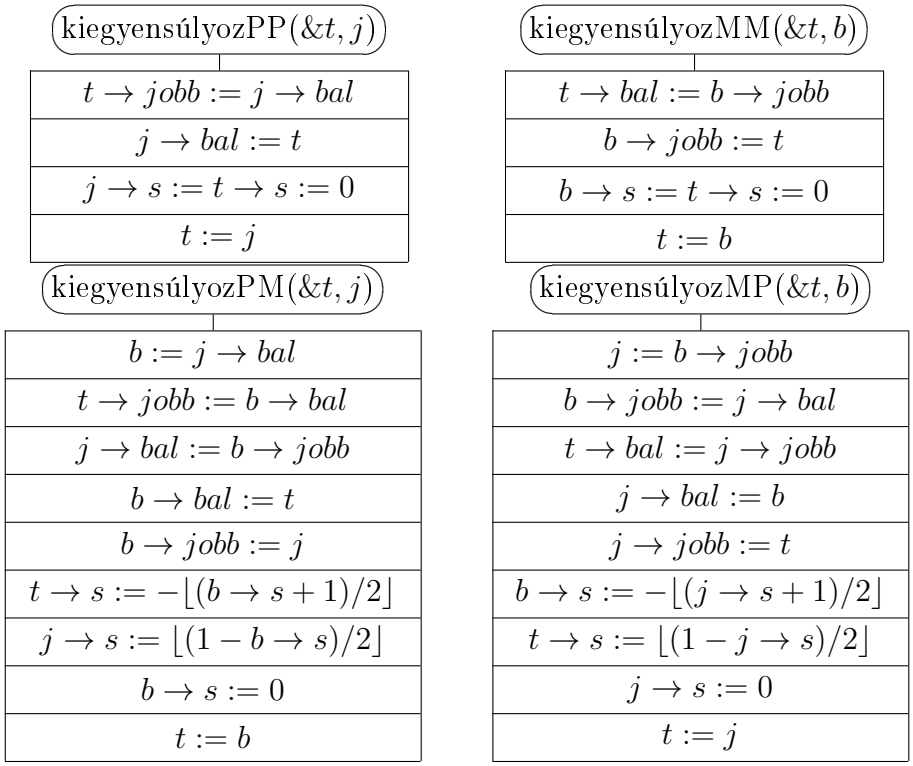
$$s_t = -\lfloor (s+1)/2 \rfloor \quad \text{és} \quad s_j = \lfloor (1-s)/2 \rfloor$$

Az eddigi két kiegyensúlyozási séma mellett természetes módon adódnak ezek tükörképei, a forgatások előtt a T$-$$-$ csúccsal a gyökérben. Ezek tehát a következők [2]:

$$((\alpha \text{ B}- \beta) \text{ T}-- \gamma) \to (\alpha \text{ B}\circ (\beta \text{ T}\circ \gamma))$$
$$((\alpha \text{ B}+ (\beta \text{ J}-\circ+ \gamma)) \text{ T}-- \delta) \to ((\alpha \text{ B}\circ\circ- \beta) \text{ J}\circ (\gamma \text{ T}+\circ\circ \delta))$$
$$s_b = -\lfloor (s+1)/2 \rfloor \quad \text{és} \quad s_t = \lfloor (1-s)/2 \rfloor$$
ahol s a J csúcs kettős forgatás előtti egyensúlya; s_b, illetve s_t pedig rendre a B és T csúcsoknak a kettős forgatás utáni egyensúlya

Eddig nem beszéltünk arról, hogy az AVL fában az új csúcs beszúrása után mely csúcsoknak és milyen sorrendben számoljuk újra az egyensúlyát, sem hogy mikor ütemezzük be a kiegyensúlyozást. Csak a beszúrás nyomvonalán visszafelé haladva kell újraszámolnunk az egyensúlyokat, és csak itt kell kiegyensúlyoznunk, ha kiegyensúlyozatlan csúcsot találunk. Így a futási idő a fa magasságával arányos marad, ami AVL fákra $O(\lg n)$. Most tehát részletezzük a beszúró és kiegyensúlyozó algoritmus működését. Ennek során a fa magassága vagy eggyel növekszik (d=igaz), vagy ugyanannyi marad (d=hamis).





Az AVL fába való beszúrást röviden összefoglalva:
1. Megkeressük a kulcs helyét a fában.
2. Ha a kulcs benne van a fában, KÉSZ vagyunk.
3. Ha a kulcs helyén egy üres részfa található, beszúrunk az üres fa helyére egy új, a kulcsot tartalmazó levélcsúcsot, azzal, hogy ez a részfa eggyel magasabb lett.
4. egyet fölfelé lépünk a keresőfában. Mivel az a részfa, amiből fölfele léptünk, eggyel magasabb lett, az aktuális csúcs egyensúlyát megfelelőképp módosítjuk. (Ha a jobb részfa lett magasabb, hozzáadunk az egyensúlyhoz egyet, ha a bal, levonunk belőle egyet.)
5. Ha az aktuális csúcs egyensúlya 0 lett, akkor az aktuális csúcshoz tartozó részfa alacsonyabb ága hozzánőtt a magasabbikhoz, tehát az aktuális részfa most ugyanolyan magas, mint a beszúrás előtt volt, és így egyetlen más csúcs egyensúlyát sem kell módosítani: KÉSZ vagyunk.
6. Ha az aktuális csúcs új egyensúlya 1 vagy -1, akkor előtte 0 volt, ezért az aktuális részfa magasabb lett eggyel. Ekkor a 4. ponttól folytatjuk.
7. Ha az aktuális csúcs új egyensúlya 2 vagy -2,[15] akkor a hozzá tartozó részfát ki kell egyensúlyozni. A *kiegyensúlyozás után az aktuális részfa visszanyeri*

[15] A 2 és -2 eseteket a struktogramban nem számoltuk ki expliciten, hogy az egyensúly tárolására elég legyen két bit.

a beszúrás előtti magasságát, ezért már egyetlen más csúcs egyensúlyát sem kell módosítani: KÉSZ vagyunk.

Az az állítás, hogy ebben az algoritmusban a *kiegyensúlyozás után az aktuális részfa visszanyeri a beszúrás előtti magasságát*, még igazolásra vár. Azt az esetet nézzük meg, amikor a kiegyensúlyozandó részfa gyökere T++. A T−− eset hasonlóan fontolható meg. A T++ esethez tartozó kiegyensúlyozási sémák [2]:

$$(\alpha\ \mathrm{T}{+}{+}\ (\beta\ \mathrm{J}{+}\ \gamma)) \to ((\alpha\ \mathrm{T}{\circ}\ \beta)\ \mathrm{J}{\circ}\ \gamma).$$
$$(\ \alpha\ \mathrm{T}{+}{+}\ ((\beta\ \mathrm{B}{-}{\circ}{+}\ \gamma)\ \mathrm{J}{-}\ \delta)\ ) \to ((\alpha\ \mathrm{T}{\circ}{\circ}{-}\ \beta)\ \mathrm{B}{\circ}\ (\gamma\ \mathrm{J}{+}{\circ}{\circ}\ \delta))$$

Először belátjuk, hogy ha a T csúcs jobboldali J gyereke + vagy − súlyú, akkor a fenti sémák közül a megfelelő alkalmazható: Mivel a beszúró algoritmus a fában a beszúrás helyétől egyesével fölfele lépked, és az első kiegyensúlyozatlan csúcsnál azonnal kiegyensúlyoz, ez alatt nincs kiegyensúlyozatlan csúcs, azaz az α, β és γ, illetve a δ részfák is kiegyensúlyozottak, ez pedig éppen a fenti forgatások feltétele, amellett, hogy bináris keresőfát akarunk kiegyensúlyozni, amit viszont a kiegyensúlyozás nélküli beszúró algoritmus garantál.

Most még be kell látni, hogy a fenti sémák minden esetet lefednek, azaz J+ vagy J−: egyrészt, J nem lehet a beszúrás által létrehozott új csúcs, mert különben T-nek a beszúrás előtti jobboldali részfája üres lett volna, tehát most nem lehetne T++. Másrészt, ha a fölfele lépkedés során nulla egyensúlyú csúcs áll elő, akkor a fölötte levő csúcsok egyensúlya már nem módosul, így kiegyensúlyozatlan csúcs sem állhat elő. Márpedig most T++. Így tehát az új csúcstól T-ig fölfelé vezető úton minden csúcs, azaz J egyensúlya is + vagy −.

Most belátjuk, hogy a kiegyensúlyozások visszaállítják a részfa beszúrás előtti magasságát. A T++ kiegyensúlyozatlan csúcs a beszúrás előtt kiegyensúlyozott volt. Mivel a beszúrás, a beszúrás helyétől kezdve T-ig fölfelé vezető úton mindegyik részfa magasságát pontosan eggyel növelte, a beszúrás előtt T+ volt. Ezért a beszúrás előtt, $h = h(\alpha)$ jelöléssel, a T gyökerű részfa $h+2$ magas volt. A beszúrás után tehát T++ lett, és így a T gyökerű részfa $h+3$ magas lett. A beszúrás utáni, de még a kiegyensúlyozás előtti állapotot tekintve a továbbiakban megkülönböztetjük a T jobboldali J gyerekére a J+ és a J− eseteket.

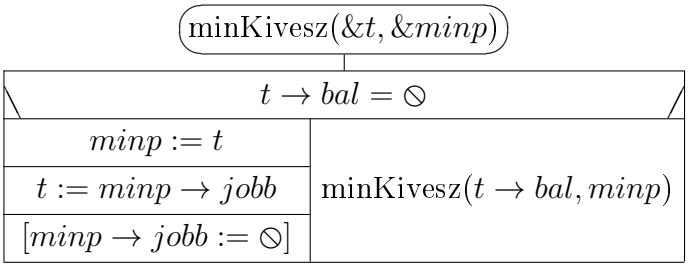
J+ esetén már láttuk, hogy $h(\alpha) = h = h(\beta)$ és $h(\gamma) = h + 1$. Ezért a kiegyensúlyozás után $h((\alpha\ \mathrm{T}\ \beta)\ \mathrm{J}\ \gamma)) = h + 2$.

J− esetén pedig már láttuk, hogy $h(\alpha) = h = h(\delta)$ és $h((\beta\ \mathrm{B}\ \gamma)) = h+1$. Ezért $h(\beta), h(\gamma) \leq h$. Így a kiegyensúlyozás után $h((\alpha\ \mathrm{T}\ \beta)\ \mathrm{B}\ (\gamma\ \mathrm{J}\ \delta)) = h+2$.

Ezzel beláttuk, hogy a kiegyensúlyozások mindkét esetben visszaállítják a részfa beszúrás előtti magasságát, mégpedig úgy, hogy a beszúrás által eggyel megnövelt magasságot eggyel csökkentik. Szimmetria okokból ez hasonlóan látható be T−− esetén is, figyelembe véve a B− és a B+ eseteket.

8.11. AVL fák: a minKivesz(t, min) eljárás

Bináris keresőfákra a minKivesz eljárás:

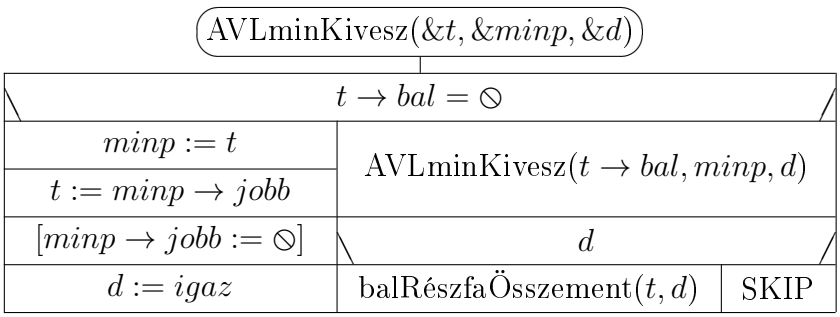


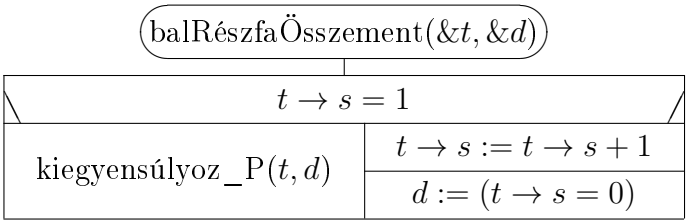
Ezt például a ((2)4+((6)8∘(10))) AVL fára alkalmazva, a $minp \to kulcs = 2$ eredmény mellett, a (4++((6)8∘(10))) kiegyensúlyozatlan bináris keresőfa adódna. Látjuk, hogy a kiegyensúlyozatlan 4++ csúcs jobboldli gyereke a 8∘. Ennek az esetnek a kezelésére új kiegyensúlyozási sémát kell alkalmaznunk. Szerencsére egy balra forgatás, a megfelelő egyensúly beállítások mellett most is megoldja a problémát [2]:

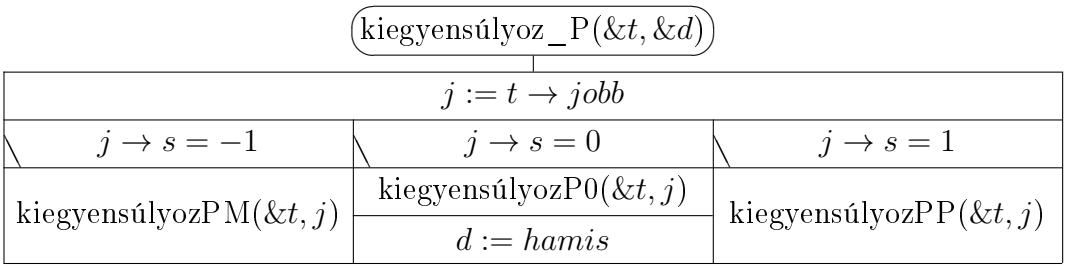
$$(\alpha \text{ T++ } (\beta \text{ J}\circ\ \gamma)) \to ((\alpha \text{ T+ } \beta) \text{ J}-\ \gamma).$$

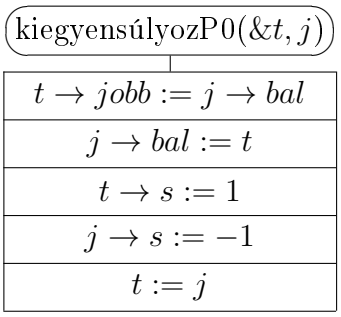
T++ miatt $h = h(\alpha)$ jelöléssel nyilván $h(\beta) = h + 1 = h(\gamma)$, így a forgatás után az egyensúlyokat a fenti séma szerint kell beállítani. A fa magassága a forgatás előtt és után is $h + 3$ lesz, ez a fajta kiegyensúlyozás tehát az eddigiekkel szemben *nem* csökkenti az aktuális részfa magasságát.

Ezután az AVL fákra alkalmazott, a megfelelő kiegyensúlyozásokkal kiegészített minKivesz eljárás pl. a következő lehet (d most azt jelöli, hogy csökkent-e az aktuális részfa magassága):









Az algoritmus helyessége hasonlóan gondolható meg, mint a beszúrás esetén. Lényeges különbség azonban, hogy ott minden beszúrást csak egyetlen kiegyensúlyozás követ, míg itt előfordulhat, hogy a minimális csúcs eltávolítása után, minden felette lévő szinten ki kell egyensúlyozni.

HF: Mutassunk ilyen, legalább négy magasságú fát!

8.12. AVL fák: törlés

Bináris keresőfákra a töröl(t, k) és a gyökeretTöröl(t) eljárások:

töröl$(\&t, k)$

$t \neq \oslash$			
$k < t \to kulcs$	$k > t \to kulcs$	$k = t \to kulcs$	SKIP
töröl$(t \to bal, k)$	töröl$(t \to jobb, k)$	gyökeretTöröl$(\&t)$	

gyökeretTöröl$(\&t)$

$t \to bal = \oslash$	$t \to jobb = \oslash$	$t \to bal \neq \oslash \wedge t \to jobb \neq \oslash$
$p := t$	$p := t$	minKivesz$(t \to jobb, p)$
$t := p \to jobb$	$t := p \to bal$	$p \to bal := t \to bal$; $p \to jobb := t \to jobb$
delete p	**delete** p	[$t \to bal := t \to jobb := \oslash$;] delete t ; $t := p$

Ezeket fogjuk most az az AVLminKivesz$(\&t, \&minp, \&d)$ eljárás mintájára kiegészíteni a d paraméterrel; majd a rekurzív töröl hívásokból, valamint a minKivesz eljárásból való visszatérés után megkérdezzük, csökkent-e az aktuális részfa magassága. Ha igen, a minKivesz eljárás mintájára módosítjuk a $t \to s$ értéket, ha kell, kiegyensúlyozunk, és ha kell, d-t beállítjuk.

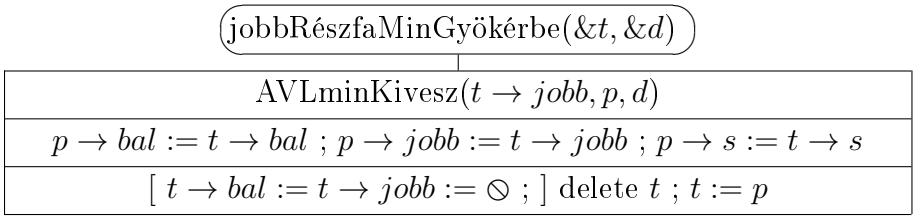
AVLtöröl$(\&t, k, \&d)$

<table>
<tr><td colspan="5">$t \neq \oslash$</td></tr>
<tr><td colspan="2">$k < t \to kulcs$</td><td colspan="2">$k > t \to kulcs$</td><td>$k = t \to kulcs$</td></tr>
<tr><td colspan="2">AVLtöröl$(t \to bal, k, d)$</td><td colspan="2">AVLtöröl$(t \to jobb, k, d)$</td><td rowspan="3">AVLgyTöröl (t, d)</td></tr>
<tr><td colspan="2">d</td><td colspan="2">d</td></tr>
<tr><td>balRészfaÖsszement (t, d)</td><td>SKIP</td><td>jobbRészfaÖsszement (t, d)</td><td>SKIP</td></tr>
</table>

$d := hamis$

AVLgyTöröl$(\&t, \&d)$

$t \to bal = \oslash$	$t \to jobb = \oslash$	$t \to bal \neq \oslash \wedge t \to jobb \neq \oslash$	
$p := t$	$p := t$	jobbRészfaMinGyökérbe(t, d)	
$t := p \to jobb$	$t := p \to bal$		
delete p	**delete** p	d	
$d := igaz$	$d := igaz$	jobbRészfaÖsszement(t, d)	SKIP



Itt is lehetséges, hogy több szinten, a legrosszabb esetben akár minden szinten is ki kell egyensúlyozni. Mivel azonban egyetlen kiegyensúlyozás sem tartalmaz se rekurziót, se ciklust, és ezért konstans számú eljáráshívásból áll, ez sem itt, sem az AVLminKivesz eljárásnál nem befolyásolja a futási időnek az AVLbeszúr eljárásra is érvényes $MT(n) \in \Theta(\lg n)$ (ahol $n = |t|$) nagyságrendjét.

HF: A balRészfaÖsszement(t, d) eljárás mintájára dolgozzuk ki a jobbRészfaÖsszement(t, d) eljárást, ennek segédeljárásait, és az ehhez szükséges kiegyensúlyozási sémát [2]! Mutassuk be ennek működését néhány példán! Mutassunk néhány olyan, legalább négy magasságú fát és kulcsot, amelyekre az AVLtöröl eljárás minden, a fizikailag törölt csúcs feletti szinten kiegyensúlyozást végez!

8.13. Az AVL fák magassága*

Tétel: Tetszőleges n csúcsú nemüres AVL fa h magasságára:

$$\lfloor \lg n \rfloor \leq h \leq 1,45 \lg n$$

Bizonyítás:

Az $\lfloor \lg n \rfloor \leq h$ egyenlőtlenség bizonyításához elég azt belátni, hogy ez tetszőleges nemüres bináris fára igaz. Egy tetszőleges bináris fa nulladik (gyökér) szintjén legfeljebb $2^0 = 1$ csúcs van, az első szintjén maximum $2^1 = 2$ csúcs, a második szintjén nem több, mint $2^2 = 4$ csúcs. Általában, ha az i-edik szinten 2^i csúcs van, akkor az $i+1$-edik szinten legfeljebb 2^{i+1} csúcs, hiszen minden csúcsnak maximum két gyereke van. Innét egy h mélységű bináris fa csúcsainak n számára $n \leq 2^0 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1 < 2^{h+1}$. Innét
$\lfloor \lg n \rfloor \leq \lg n < \lg 2^{h+1} = h + 1$, amiből $\lfloor \lg n \rfloor \leq h$.

A $h \leq 1,45 \lg n$ egyenlőtlenség bizonyításához elég azt belátni, hogy ez tetszőleges nemüres, kiegyensúlyozott bináris fára (KBF-re) igaz. Ehhez először

meghatározzuk egy $h \geq 0$ magasságú (azaz nemüres) KBF csúcsainak minimális f_h számát. Nyilván $f_0 = 1$ és $f_1 = 2$, hiszen egy nulla magasságú KBF csak a gyökércsúcsból áll, az egy magasságú KBF-ek pedig ((B)G(J)), ((B)G), vagy (G(J)) alakúak.

Az is világos, hogy az $< f_0, f_1, f_2, \ldots >$ sorozat szigorúan monoton növekvő. (Ennek igazolásához vegyünk egy $i+1$ magas, f_{i+1} csúcsú t KBF-et! Ekkor a t bal és jobb részfái közül az egyik magassága i. Legyen ez az f részfa! Jelölje most $s(b)$ egy tetszőleges b bináris fa csúcsainak számát! Akkor $f_{i+1} = s(t) > s(f) \geq f_i$.)

Ezután $h \geq 2$ esetén $f_h = 1 + f_{h-1} + f_{h-2}$. (Ha ugyanis t egy h magasságú, minimális, azaz f_h méretű KBF, akkor ennek bal és jobb részfáiban is, a részfák magasságához mérten a lehető legkevesebb csúcs van. Az egyik részfája kötelezően $h-1$ magas, ebben tehát f_{h-1} csúcs van. Mivel t KBF, a másik részfája $h-1$ vagy $h-2$ magas. A másik részfában tehát f_{h-1} vagy f_{h-2} csúcs van, és $f_{h-2} < f_{h-1}$, tehát a másik részfában f_{h-2} csúcs van, és így $f_h = 1 + f_{h-1} + f_{h-2}$.)

A képlet emlékeztet a Fibonacci sorozatra:
$F_0 = 0$, $F_1 = 1$, $F_h = F_{h-1} + F_{h-2}$, ha $h \geq 2$.

Megjegyzés: A h magasságú, és f_h méretű KBF-eket ezért *Fibonacci fáknak* hívjuk. Az előbbiek szerint egy $h \geq 2$ magasságú Fibonacci fa mindig $(\varphi_{h-1}$ G $\varphi_{h-2})$ vagy $(\varphi_{h-2}$ G $\varphi_{h-1})$ alakú, ahol φ_{h-1} és φ_{h-2} $h-1$, illetve $h-2$ magasságú Fibonacci fák.

HF: Rajzoljunk $h \in 0..4$ magasságú Fibonacci fákat!

Megvizsgáljuk, van-e valami összefüggés az
$< f_h : h \in \mathbb{N} >$ és az $< F_h : h \in \mathbb{N} >$ sorozatok között.

h	0	1	2	3	4	5	6	7	8	9
F_h	0	1	1	2	3	5	8	13	21	34
f_h	1	2	4	7	12	20	33			

A fenti táblázat alapján az első néhány értékre $f_h = F_{h+3} - 1$. Teljes indukcióval könnyen látható, hogy ez tetszőleges $h \geq 0$ egész számra igaz: Feltéve, hogy $0 \leq h \leq k \geq 1$ esetén igaz, $h = k+1$-re:
$f_h = f_{k+1} = 1 + f_k + f_{k-1} = 1 + F_{k+3} - 1 + F_{k+2} - 1 = F_{k+4} - 1 = F_{h+3} - 1$.

Tudjuk, hogy

$$F_h = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right]$$

Az F_h-ra vonatkozó explicit képlet segítségével összefüggést adunk tetszőleges KBF n mérete és h magassága között.

$$n \geq f_h = F_{h+3} - 1 = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right] - 1 \geq$$

$$\geq \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left| \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right| \right] - 1$$

Mivel $2 < \sqrt{5} < 3$, azért $|1-\sqrt{5}|/2 < 1$ és $\left| \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right| < 1$. Eszerint

$$n > \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \right] - 1 = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^3 \left(\frac{1+\sqrt{5}}{2} \right)^h - \frac{1+\sqrt{5}}{\sqrt{5}}$$

Mivel

$$\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^3 = \frac{1+3\sqrt{5}+3(\sqrt{5})^2+(\sqrt{5})^3}{8\sqrt{5}} = \frac{16+8\sqrt{5}}{8\sqrt{5}} = \frac{2+\sqrt{5}}{\sqrt{5}}$$

Ezt behelyettesítve az előbbi, n -re vonatkozó egyenlőtlenségbe:

$$n > \frac{2+\sqrt{5}}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^h - \frac{1+\sqrt{5}}{\sqrt{5}}$$

Az első együtthatót tagokra bontva, a disztributív szabállyal:

$$n > \left(\frac{1+\sqrt{5}}{2} \right)^h + \frac{2}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^h - \frac{1+\sqrt{5}}{\sqrt{5}}$$

Most

$$\frac{2}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^h - \frac{1+\sqrt{5}}{\sqrt{5}} \geq 0 \Longleftrightarrow \left(\frac{1+\sqrt{5}}{2} \right)^h \geq \frac{1+\sqrt{5}}{2} \Longleftrightarrow h \geq 1$$

Eszerint $h \geq 1$ esetén

$$n > \left(\frac{1+\sqrt{5}}{2} \right)^h$$

$h = 0$ -ra pedig $n = 1$, és így $n = \left(\frac{1+\sqrt{5}}{2}\right)^h$

A fentiekből tetszőleges, nemüres KBF n méretére és h magasságára

$$n \geq \left(\frac{1+\sqrt{5}}{2}\right)^h$$

Innét, ha vesszük mindkét oldal kettes alapú logaritmusát, majd $\lg \frac{1+\sqrt{5}}{2}$ -tel osztunk:

$$h \leq \frac{1}{\lg \frac{1+\sqrt{5}}{2}} \lg n$$

Mivel $1,44 < 1,44042 < \frac{1}{\lg \frac{1+\sqrt{5}}{2}} < 1,4404201 < 1,45$, azért tetszőleges, nemüres KBF n méretére és h magasságára

$$h \leq 1,45 \lg n$$

9. Általános fák

Az általános fák esetében, a bináris fákkal összehasonlítva, egy csúcsnak tetszőlegesen sok gyereke lehet. Itt azonban, tetszőleges csúcshoz tartozó részfák száma pontosan egyenlő a gyerekek számával, azaz nem tartoznak hozzá üres részfák. Ha a gyerekek sorrendje lényeges, akkor *rendezett fákról* beszélünk.

Általános fákkal modellezhetjük például a számítógépünkben a mappák hierarchiáját, a programok blokkstruktúráját, a függvénykifejezéseket, a családfákat és bármelyik hierarchikus struktúrát.

Vegyük észre, hogy ugyan minden konkrét általános fában van az egy csúcshoz tartozó gyerekek számára valamilyen r felső korlát, de ettől a fa még nem tekinthető r-árisnak, mert ez a korlát nem abszolút: a fa tetszőleges csúcsa gyerekeinek száma tetszőlegesen növelhető. Másrészt, mivel itt nem értelmezzük az *üres részfa* fogalmát, ez mégsem általánosítása az r-áris fa fogalmának. Értelmezzük azonban a gyökércsúcs (nincs szülője) és a levélcsúcs (nincs gyereke) fogalmát, azaz továbbra is gyökeres fákról beszélünk.[16]

Az általános fák természetes ábrázolási módja a *bináris láncolt* reprezentáció. Itt egy csúcs szerkezete a következő (ahol most *egy* az első gyerekre, *tv* pedig a következő testvérre mutat). A $*p$ csúcs akkor levél, ha $p \to egy = \oslash$; és a $*p$ csúcs akkor utolsó testvér, ha $p \to tv = \oslash$.

[16]Ellentétben az ún. *szabad fákkal*, amik irányítatlan, körmentes gráfok.

Csúcs
+ egy, tv : Csúcs* // egy: első gyerek; tv: következő testvér + $kulcs$: T // T ismert típus
+ Csúcs() { $egy := tv := \oslash$ } // egycsúcsú fát képez belőle + Csúcs(x:T) { $egy := tv := \oslash$; $kulcs := x$ }

Természetesen itt is lehet *szülő* pointer, ami a gyerekek közös szülőjére mutat vissza.

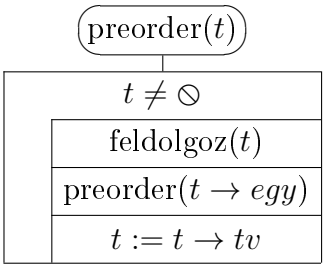
HF: Próbáljunk megadni az általános fákra másfajta láncolt reprezentációkat is! Hasonlítsuk össze az általunk adott ábrázolásokat és a bináris láncolt reprezentációt, memóriaigény és rugalmasság szempontjából!

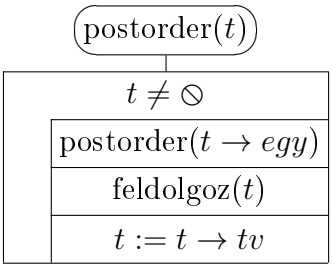
A szöveges (zárójeles) reprezentációban az általános fáknál a gyökeret előre szokás venni. Így egy nemüres fa általános alakja (G $t_1 \dots t_n$), ahol G a gyökércsúcs tartalma, $t_1 \dots t_n$ pedig a részfák.

Így pl. az (1 (2 (5)) (3) (4 (6) (7)))) általános fában az 1 van a gyökérben, a gyerekei a 2, a 3 és a 4, a hozzájuk tartozó részfák pedig sorban a (2 (5)), a (3) és a (4 (6) (7))). A fa levelei az 5, a 3, a 6 és a 7.

HF: Rajzoljuk le a fenti fát! Írjunk programot, ami kiír egy binárisan láncolt fát a fenti zárójeles alakban! Írjunk olyat is, ami egy szövegfájlból visszaolvassa! Készítsük el a kiíró és a visszaolvasó programokat az általunk kidolgozott alternatív láncolt ábrázolásokra is! (A visszaolvasó programokat általában nehezebb megírni.)

Az általános fa preorder bejárása[17] az $egy \sim bal$ és $tv \sim jobb$ megfeleltetéssel a bináris reprezentáció preorder bejárását igényli. Az általános fa postorder bejárásához[18] azonban (az előbbi megfeleltetéssel) a bináris reprezentáció inorder bejárása szükséges.[19]





[17] először a gyökér, majd sorban a gyerekeihez tartozó részfák

[18] előbb sorban a gyökér gyerekeihez tartozó részfák, végül a gyökér

[19] A fájlrendszerekben általában preorder bejárás szerint keresünk, míg a függvénykifejezéseket (eltekintve most a lusta kiértékeléstől, aminek a tárgyalása túl messzire vezetne) postorder bejárással értékeljük ki.

HF: Lássuk be a fenti bejárások helyességét! (Ötlet: A testvérek listájának mérete szerinti teljes indukció pl. célravezető.) Lássuk be egy ellenpélda segítségével, hogy az általános fa inorder bejárása[20] nem szimulálható a bináris reprezentáció egyik nevezetes bejárásával[21] sem! Írjuk meg a bináris láncolt reprezentációra az általános fa inorder bejárását és szintfolytonos bejárását is!

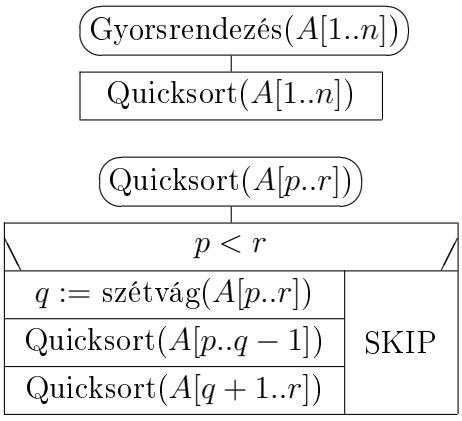
HF: Írjuk meg a fenti bejárásokat az általunk adott aletrnatív láncolt reprezentációkra is! Írjunk olyan programokat, amelyek képesek tetszőleges általános fa egy adott reprezentációjából egy adott másik ábrázolású másolatot készíteni!

10. B+ fák és műveleteik

http://aszt.inf.elte.hu/~asvanyi/ad/B+ fa.pdf

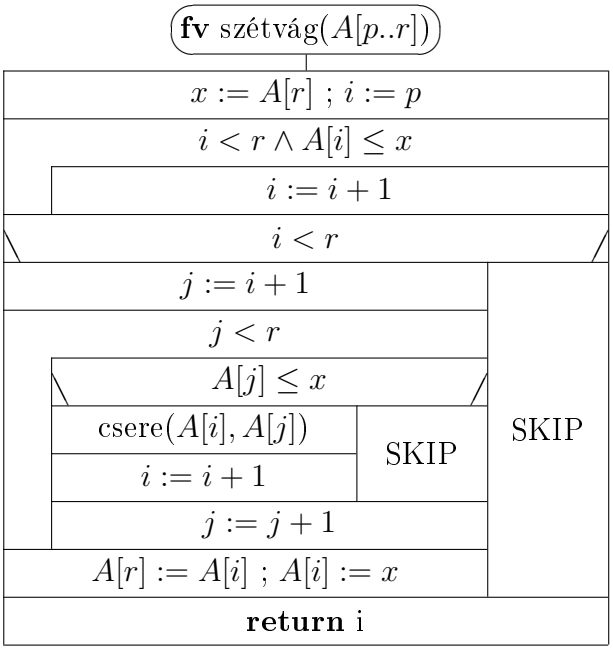
11. Gyorsrendezés (Quicksort)

http://aszt.inf.elte.hu/~asvanyi/ad/FeketeIstvan/quick_sort_FI.doc



[20] A (G $t_1 \ldots t_n$) fára $n > 0$ esetén előbb t_1-et járja be, majd feldolgozza G-t, aztán bejárja sorban a $t_2 \ldots t_n$ részfákat; $n = 0$ esetén csak feldolgozza G-t.

[21] preorder, inorder, postorder, szintfolytonos



Vezessük be a következő jelöléseket:

- A' az A vektor kezdeti állapota, a szétvág függvény meghívásakor.
- $A[k..m] \leq x$ akkor és csak akkor, ha tetszőleges l-re, $k \leq l \leq m$ esetén $A[l] \leq x$
- $A[k..m] > x$ akkor és csak akkor, ha tetszőleges l-re, $k \leq l \leq m$ esetén $A[l] > x$

A szétvág fv előfeltétele: $p < r$
(A p és r indexhatárok itt természetesen konstansok.)

A szétvág fv második ciklusának invariánsa:
$A[p..r]$ az $A'[p..r]$ permutációja $\wedge$ $A[r] = x \wedge p \leq i < j \leq r \wedge$
$A[p..(i-1)] \leq x \wedge A[i..(j-1)] > x$

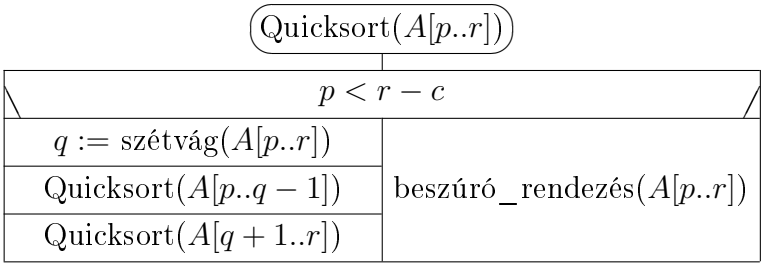
A szétvág fv utófeltétele:
$A[p..r]$ az $A'[p..r]$ permutációja $\wedge$ $p \leq i \leq r \wedge A[i] = A'[r] \wedge$
$A[p..(i-1)] \leq x \wedge A[(i+1)..r] > x$

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

A rekurzív eljárás műveletigényének ezen alapuló elemzése a saját, kézzel írt előadás jegyzetben, illetve a jelzett szakirodalomban! (Az előadáson [és a vizsga anyagában] csak $mT(n) \in O(n \lg n)$ és $MT(n) \in \Omega(n^2)$ bizonyítása szerepel.)

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

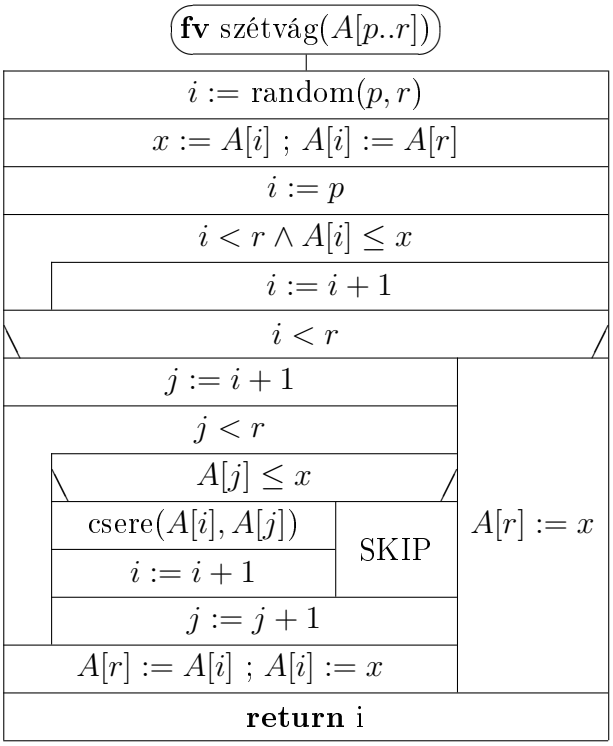
Ismert, hogy kisméretű tömbökre a beszúró rendezés hatékonyabb, mint a gyors rendezések (mergesort, heapsort, quicksort). Ezért a Quicksort($A[p..r]$) eljárás jelentősen gyorsítható, ha kisméretű tömbökre áttérünk beszúró rendezésre. Mivel a szétvágások során sok kicsi tömb áll elő, így ezzel a program futása sok ponton gyorsítható:



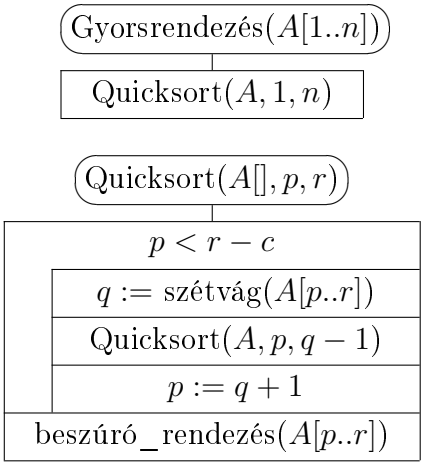
Itt $c \in \mathbb{N}$ konstans. Optimális értéke sok tényezőtől függ, de általában 20 és 30 között mozog.

HF: Hogyan tudnánk az összefésülő rendezést hasonló módon gyorsítani?

Előrendezett inputokra a Quicksort lelassul, mert a szétvág fv egyenetlenül vág. Annak érdekében, hogy az ilyen „balszerencsés” bemenetek valószínűségét csökkentsük, és azért is, mert az előrendezett inputok rendezése a gyakorlatban egy fontos feladatosztály, érdemes a szétvágáshoz használt elemet az $A[p..r]$ résztömb egy véletlenszerű elemének választani:



11.1. A gyorsrendezés végrekurzió-optimalizált változata*



12. Az összehasonlító rendezések alsókorlát-elemzése

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