Algoritmusok és adatszerkezetek I. előadásjegyzet

Ásványi Tibor – asvanyi@inf.elte.hu

2019. február 10.

Tartalomjegyzék

1. Bevezetés

Az itt következő előadásjegyzetekben bizonyos fejezetek még nem teljesek. Az előadásokon tárgyalt programok struktogramjait igyekeztem minden esetben megadni, a másolási hibák kiküszöbölése érdekében. E tananyagon kívül a megértést segítő ábrák találhatóak bőségesen az ajánlott segédanyagokban.

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

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

Hivatkozások

[1] Ásványi Tibor, Algoritmusok és adatszerkezetek I. előadásjegyzet (2018)
http://aszt.inf.elte.hu/~asvanyi/ad/ad1jegyzet.pdf

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

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

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

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

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

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

Az angol nyelvű szakirodalom jelentős része letölthető pl. az alábbi honlapról.
http://gen.lib.rus.ec/

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

2. Tematika

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3. Néhány alapvető jelölés és elméleti háttere

$\mathbb{B} = \{false; true\}$ a logikai (boolean) értékek halmaza.
$\mathbb{N} = \{0; 1; 2; 3; \ldots\}$ 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.
$\mathbb{P}_0$ a nemnegatív valós számok halmaza.

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

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

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

A változók láthatósága és hatásköre is az őket tartalmazó struktogram, élettartamuk pedig az első, őket tartalmazó utasítás végrehajtásától a struktogram befejezéséig tart. Kivételt képeznek a globális változók, amelyek a program egész végrehajtása alatt élnek és láthatók is. A változók lokálisan nem definiálhatók felül.

3.1. Tömbök

A tömböket pl. így deklarálhatjuk: $A, Z : \mathcal{T}[n]$. Ekkor A és Z is n elemű, $\mathcal{T}$ elemtípusú vektorok. A tömbök „ismerik" a méretüket, az A vektor mérete pl. $A.M$, ami nem változtatható meg. (Itt tehát $A.M == Z.M == n$.)

A tömböket általában 1-től indexeljük, de azokat, amelyek neve z vagy Z betűvel végződik, zérustól.

A fenti A változó tehát egy n elemű vektor-objektumra hivatkozik, aminek elemeit az $A[1], \ldots, A[n]$ kifejezésekkel azonosíthatjuk, a Z pedig egy másik n elemű vektor-objektumra hivatkozik, de ennek elemeit a $Z[0], \ldots, Z[n-1]$ kifejezésekkel azonosíthatjuk. A és Z valójában pointerek, amik a megfelelő vektor-objektum memóriacímét tartalmazzák.

Ha csak a $\mathcal{T}$ elemtípusú tömbökre hivatkozó P pointert akarjuk deklarálni, ezt a $P : \mathcal{T}[]$ deklarációs utasítással tehetjük meg.

Ezután a P pointer inicializálható pl. a $P = Z$ értékadó utasítással, miután P is a Z által hivatkozott tömb-objektumra mutat, és így $P.M == Z.M$, valamint $Z[0]$-nak $P[1]$, ..., $Z[n-1]$-nek $P[n]$ felel meg.

Új tömb-objektumot dinamikusan pl. a **new** $\mathcal{T}[n]$ kifejezéssel hozhatunk létre, ami egy n elemű, $\mathcal{T}$ elemtípusú vektort-objektumot hoz létre, és a címét visszaadja. A $P =$ **new** $\mathcal{T}[n]$ utasítás hatására pl. a P pointer az új vektor-objektumra fog hivatkozni, amelynek elemei sorban $P[1], \ldots P[n]$; mérete pedig $P.M == n$.

A $Qz =$ **new** $\mathcal{T}[n]$ utasítás hatására a Qz pointer új, n elemű vektor-objektumra fog hivatkozni, amelynek elemei sorban $Qz[0], \ldots Qz[n-1]$. Ezután a $Zr = Qz$ értékadás hatására a Zr pointer is a Qz által hivatkozott tömb-objektumra hivatkozik, amelynek elemei a Zr pointeren keresztül sorban a $Zr[1], \ldots Zr[n]$ kifejezésekkel érhetők el. Pl. a $Zr[1] = 3$ értékadás után $Qz[0] == 3$ is igaz lesz, hiszen $Zr[1]$ és $Qz[0]$ ugyanazt a memóriarekeszt azonosítják. Hasonlóan, a $Qz[1] = 5$ értékadás után $Zr[2] == 5$ is igaz lesz stb.

A mi (C/C++-hoz hasonló) modellünkben, a memóriaszivárgás elkerülése érdekében, a dinamikusan (**new** utasítással) létrehozott, és már feleslegessé vált objektumokat expliciten törölni kell. Ezzel ugyanis az objektum által lefoglalt memóriaterület újra felhasználhatóvá, míg ellentétes esetben az alkalmazás számára elérhetetlenné válik. Az ilyen memóriadarabok felhalmozódása jelentősen csökkentheti a programunk által használható memóriát, abban szemetet képez. (A JAVA és más hasonló környezetek a memória-szemét kezelésére automatikus eszközöket biztosítanak, de ennek a hatékonyság oldaláról nézve súlyos ára van. Mivel a mi algoritmusaink egyik legfontosabb akalmazási területe a rendszerprogramozás, mi ilyen automatizmusokat nem feltételezünk.)

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

delete P; **delete** Qz;

amik törlik a P és a Qz pointerek által hivatkozott tömb-objektumokat, de nem törlik a pointereket magukat. A fenti törlések hatására a P és a Qz pointerek nem definiált memóriacímeket tartalmaznak, és később újra érteket kaphatnak.

3.2. Szögletes zárójelek közé írt utasítások

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

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

A továbbiakban az eljárásokat, függvényeket és az osztályok metódusait együtt *alprogramoknak* nevezzük.

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

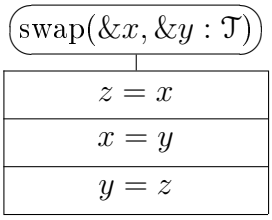
Ha egy paraméterlistával ellátott alprogram struktogramjában olyan változónév szerepel, ami a paraméterlistán nem szerepel, és nem is az alprogram külső (azaz globális) változója, akkor ez a struktogrammal leírt alprogram lokális változója.

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

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

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

[2] Az érték szerinti paraméterátadás esetén, az 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.



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

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

Az aktuális paraméter listán nem jelöljük külön a cím szerinti paraméterátadást, mert a formális paraméter listáról kiderül, hogy egy tetszőleges paramétert érték vagy cím szerint kell-e átadni. (Összhangban a C++ jelölésekkel.) Pl. a swap eljárás egy lehetséges meghívása: swap$(A[i], A[j])$.

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

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

3.4. Tömb típusú paraméterek a struktogramokban

A tömböket a formális paraméter listákon tömbre hivatkozó pointerként jelölhetjük, pl. a

linearSearch$(A : \mathcal{T}[]\ ;\ x : \mathcal{T}) : \mathbb{N}$

függvényfej olyan függvényre utalhat, ami az A vektorban megkeresi az

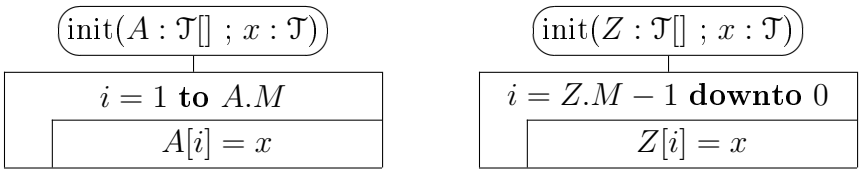
[3]A nem-skalár típusok az összetett típusok. Pl. a tömb, sztring, rekord, fájl, halmaz, zsák, sorozat, fa és gráf típusok, valamint a tipikusan `struct`, illetve `class` kulcsszavakkal definiált osztályok.

x első előfordulását, és visszaadja annak indexét; vagy nullát, ha $x \notin \{A[1], \ldots, A[A.M]\}$. Alprogram híváskor a tömb paramétereknél az aktuális paraméterben levő memóriacím – amely a megfelelő tömb-objektum címe – a formális paraméterbe másolódik, így az is ugyanarra a tömb-objektumra fog hivatkozni. Ezért, ha a hívott alprogram futása során, a formális paraméter által hivatkozott tömböt megváltoztatjuk, ez az aktuális paraméter által hivatkozott tömbbel is azonnal megtörténik. Így alprogram híváskor az aktuális paraméter tömb címét ugyan érték szerint adjuk át, a cím (azaz pointer) által hivatkozott tömb-objektum mégis cím szerint adódik át.

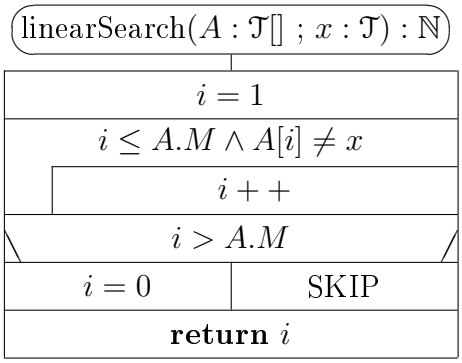
3.5. Eljárások, függvények, ciklusok, rekurzió

Először egy egyszerű eljárást nézünk meg két változatban, ami egy tetszőleges egy dimenziós tömb elemeit ugyanazzal az értékkel inicializálja. Mindkét esetben a Pascal programozási nyelvből esetleg már ismerős léptető ciklust alkalmazunk. Annyi a különbség, hogy az első esetben sorban haladunk az elemeken, míg a másodikban sorban visszafelé, és az A vektor 1-től, míg a Z zérustól indexelődik (mivel a neve Z-re végződik, ld. (3.1)).

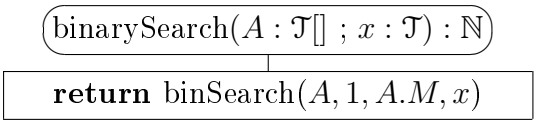
Vegyük észre, hogy az init eljárás két változata a tömbök és a paraméterátvétel tulajdonságai miatt ekvivalens.



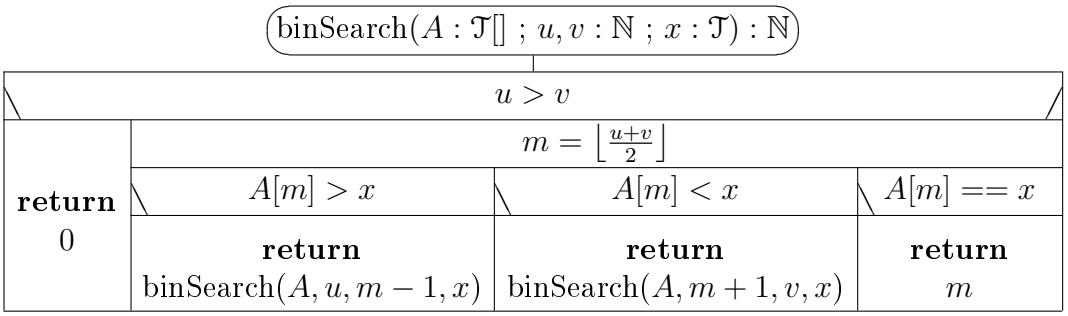
Ebben a jegyzetben megkülönböztetjük az eljárás és a függvény fogalmát. Az eljárások a környezetükkel csak a paramétereiken (és esetleg külső változókon) keresztül kommunikálnak, míg a függvényeknek visszatérési értékük is van, amit a szokásos módon használhatunk fel. (A mi *eljárás* fogalmunknak a C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



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



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



Formailag a binSearch(A, u, v, x) rekurzív függvény, mivel van olyan programága, ahol önmagát hívja (amit rekurzív hívásnak nevezünk). A számítógép minden alprogram hívásra ugyanúgy, az alprogram hívások lokális adatait a *call stack*-ben tárolja, tehát a rekurzív hívásokat is ugyanúgy kezeli, mint a nemrekurzívakat: tetszőleges alprogram lokális adatainak akár több példánya

is lehet a *call stack*-ben. Ez tehát önmagában nem okoz technikai nehézséget. A rekurzív alprogramoknál azonban gondoskodnunk kell a rekurzió megfelelő leállításáról, hiszen az alprogram elvileg a végtelenségig hívogathatja önmagát. Ezt szolgálják a rekurzív alprogramokban az alább ismertetetendő ún. *leálló ágak*, amiket a bemenő adatok közül az ún. *alapesetek* aktiválnak.

Most igazoljuk, hogy a binSearch(A, u, v, x) függvény visszad egy $k \in u..v$ indexet, amelyre $A[k] == x$; vagy nullát, ha ilyen k index nem létezik. Működését tekintve, először megnézi, hogy az $u..v$ intervallum nem üres-e. Ha üres, akkor az $A[u..v]$ résztömb is az, tehát x-et nem tartalmazza, azaz nullát kell visszaadni. Ha az $u..v$ intervallum nemüres, m lesz az $A[u..v]$ résztömb középső elemének indexe. Ha $A[m] > x$, akkor az A vektor monoton növekvő rendezettsége miatt x csak az $A[u..(m-1)]$ résztömbben lehet, ha pedig $A[m] < x$, akkor x csak az $A[(m+1)..v]$ résztömbben lehet. Mindkét esetben egy lépésben feleztük a résztömb méretét, amin keresni kell, és a továbbiakban, rekurzívan, ugyanez történik. (Ha szerencsénk van, és $A[m] == x$, akkor persze azonnal leállhatunk.) Így, könnyen belátható, hogy $n = A.M$ jelöléssel, legfeljebb $\lceil \lg n \rceil + 1$ lépésben elfogy az $u..v$ intervallum, és megáll az algoritmus, hacsak nem áll meg hamarabb az $A[m] == x$ feltételű programágon.

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

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

3.6. Programok, alprogramok és hatékonyságuk

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1. Vektor monoton növekvő rendezése

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

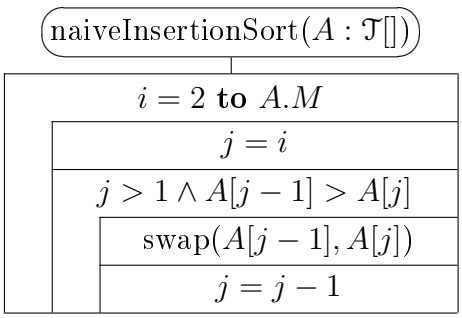
Egy sorozatot legegyszerűbben egy vektorban tárolhatunk, amit az egyes rendező eljárások paraméterlistáján általában „A : $\mathcal{T}$[]”-vel fogunk jelölni. Emlékeztetünk, hogy az A fizikailag egy pointer, amely az úgynevezett vektor-objektum memóriacímét tartalmazza, és így alkalmas a vektor azonosítására. A vektor-objektum a vektor elemeinek számát ($A.M$), és a vektor

elemeit $(A[1], \ldots, A[A.M])$ tartalmazza. Ha $A.M == 0$, akkor a vektornak nincs eleme. $A[k..u]$ az a részvektor, amelyben k az első elem indexe, u pedig az utolsó elem indexe. Ha $k > u$, akkor a részvektor üres. Az $A[1..A.M]$ részvektor az A vektor minden elemét tartalmazza. A vektor rendezéseknél feltesszük, hogy a vektor $\mathcal{T}$ elemtípusára teljes rendezés definiált, és az értékadó utasítás is értelmezve van.

4.1.1. Beszúró rendezés (Insertion sort)

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

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

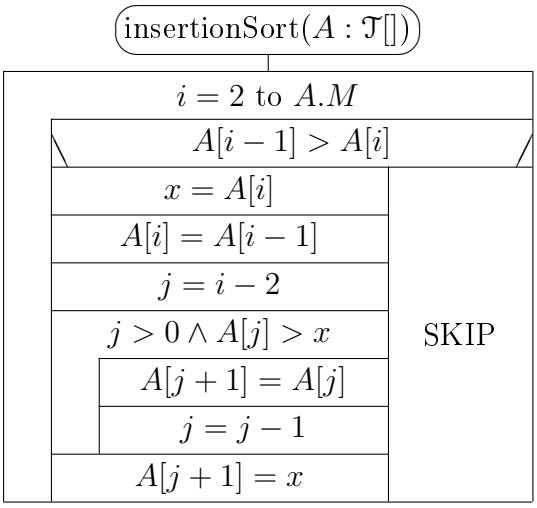


A fenti naiv megoldás azonban sok felesleges adatmozgatással jár, hiszen azt az elemet, amit a helyére szeretnénk vinni, újra és újra kivesszük a vektorból, majd visszatesszük bele. Nyilván hatékonyabb lenne, ha az elején kivennénk, majd amikor már megvan a helye, csak akkor tennénk vissza a tömbbe.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

Az előbbi megfontolás alapján a beszúró rendezés alapváltozatában a beszúrást úgy végezzük el, hogy a rendezetlen szakasz első elemét (legyen x)

összehasonlítjuk a rendezett szakasz utolsó elemével (legyen u). Ha $u \leq x$, akkor x a helyén van, csak a rendezett szakasz felső határát kell eggyel növelni. Ha $u > x$, akkor x-et elmentjük egy temporális változóba, és u-t az x helyére csúsztatjuk. Úgy képzelhetjük, hogy u régi helyén most egy „lyuk" keletkezett. Az x a lyukba pontosan akkor illik bele, ha nincs bal szomszédja, vagy ez $\leq x$. Addig tehát, amíg a lyuknak van bal 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 1. ábrát és az alábbi struktogramot!)

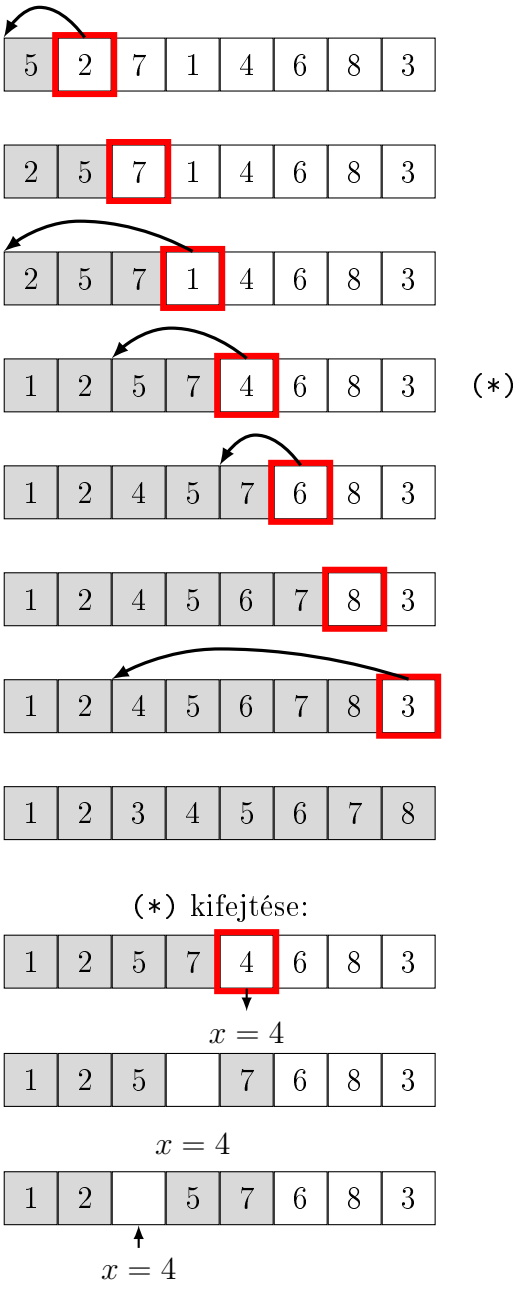
Tekintsük például a $\langle 2, 4, 5, 8, 3\rangle$ tömböt, ami a 8-ig rendezett, és már csak a 3-at kell rendezetten beszúrni. Először úgy találjuk, hogy $8 > 3$, így a 3-at kivesszük x-be, majd a lyukat (jelölje „_") a helyére mozgatjuk, és végül beletesszük a 3-at: $\langle 2, 4, 5, 8, 3\rangle \to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



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

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

Összefoglalva a működést: Ha $A.M < 2$, akkor az A vektor üres, vagy egyelemű, ezért rendezett, és a program fő ciklusa egyszer sem fut le. Ha $A.M \geq 2$, a rendezés meghívásakor csak annyit tudhatunk, hogy $A[1..1]$ rendezett, tehát $i = 2$-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 $A.M + 1$ értéket, már a teljes A vektor rendezett, és ekkor be is fejeződik az eljárás.



1. ábra. A beszúró rendezés szemléltetése.

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

4.1.2. Programok hatékonysága – és a beszúró rendezés

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

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

A továbbiakban, a programok futási idejével kapcsolatos számításoknál, a *műveletigény* és a *futási idő*, valamint a *költség* kifejezéseket 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)![7] A rendezés során egyetlen eljáráshívás hajtódik végre, és ez az

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

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

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

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.[8] 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

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

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

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 8. fejezetben fogjuk megadni.)

figyelembe vesszük őket.

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

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 nem többször, mint $(k+s)*n$ -szer fut le. Ha viszont a bemenet monoton csökkenően előrendezett, az algoritmus műveletigénye is közel marad a legrosszabb esethez. (Bár ha ezt tudjuk, érdemes a 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:

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

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

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

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

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

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

4.2. A futási időkre vonatkozó becslések magyarázata*

Jelölje most az insertionSort($A : \mathcal{T}[]$) eljárás *tényleges* maximális és minimális futási idejét sorban $MT(n)$ és $mT(n)$, ahol $n = A.M$!

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 futási ideje, b pedig az eljárás meghívásával, a fő ciklus előkészítésével és befejezésével, valamint az eljárásból való visszatéréssel kapcsolatos futási idők összege! Ekkor a és b nyilván pozitív konstansok, és $mT(n) = a * (n-1) + b$. 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 az input 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.)

4.3. Rendezések stabilitása

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

A beszúró rendezés (insertion sort) például – úgy, ahogy ebben a jegyzetben tárgyaljuk – stabil. A fenti vektorrendező algoritmusnál ez abból látható, hogy a tömb rendezett szakaszába az újabb elemeket jobbról balra szúrjuk be, és a beszúrandóval egyenlő kulcsú elemeket már nem lépjük át.

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

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

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

4.4. Kiválasztó rendezések (selection sorts)

4.2. Feladat. *Tekintsük az $A[1..n]$ tömb rendezését a következő algoritmussal! Először megkeressük a tömb minimális elemét, majd megcseréljük $A[1]$-gyel. Ezután megkeressük a második legkisebb elemét és megcseréljük $A[2]$-vel. Folytassuk ezen a módon az $A[1..n]$ első $(n-1)$ elemére! (Itt tehát a vektort egy rendezett és egy rendezetlen szakaszra bontjuk. A rendezett szakasz a tömb elején kezdetben üres. A minimumkeresés mindig a rendezetlen részen történik, és a csere után a rendezett szakasz mindig eggyel hosszabb lesz.) Pl.:*

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

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

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

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

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

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

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

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

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

5.1. Összefésülő rendezés (merge sort)

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

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

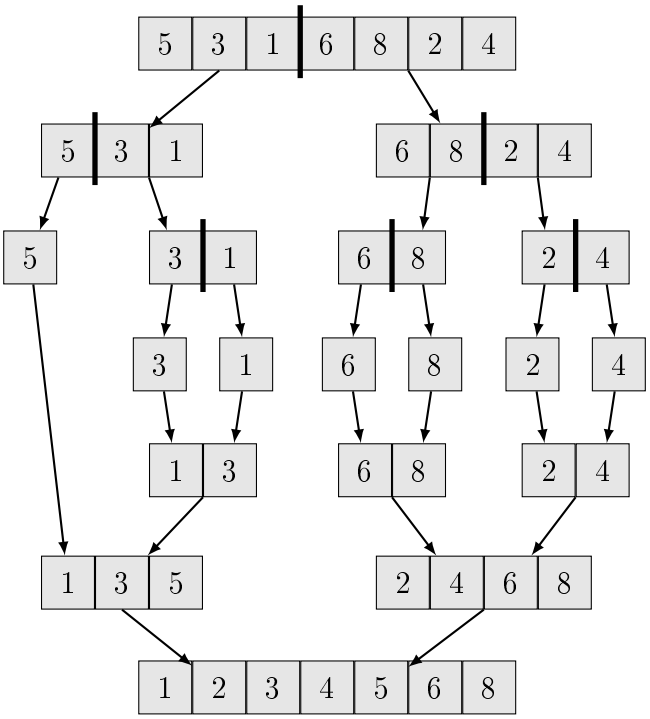
Az üres és az egyelemű sorozatok eleve rendezettek; a hosszabb sorozatokat pedig elfelezzük, a két fél-sorozatot ugyanezzel a módszerrel rendezzük, és a rendezett fél-sorozatokat rendezetten összefésüljük.

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

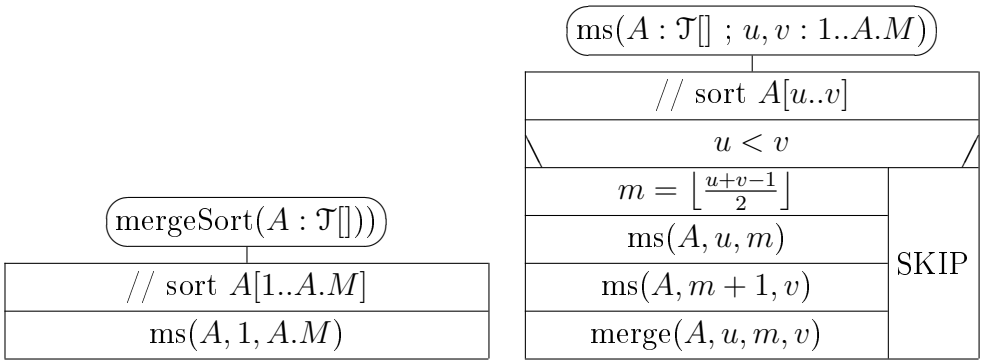
Az összefésülő rendezés stabil (azaz megőrzi az egyenlő kulcsú elemek bemeneti sorrendjét). A legrosszabb esetének műveletigénye aszimptotikusan optimális az ún. összehasonító rendezések között (a részletek a 10. fejezetben olvashatók).

Az összefésülő rendezés (merge sort, 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.

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



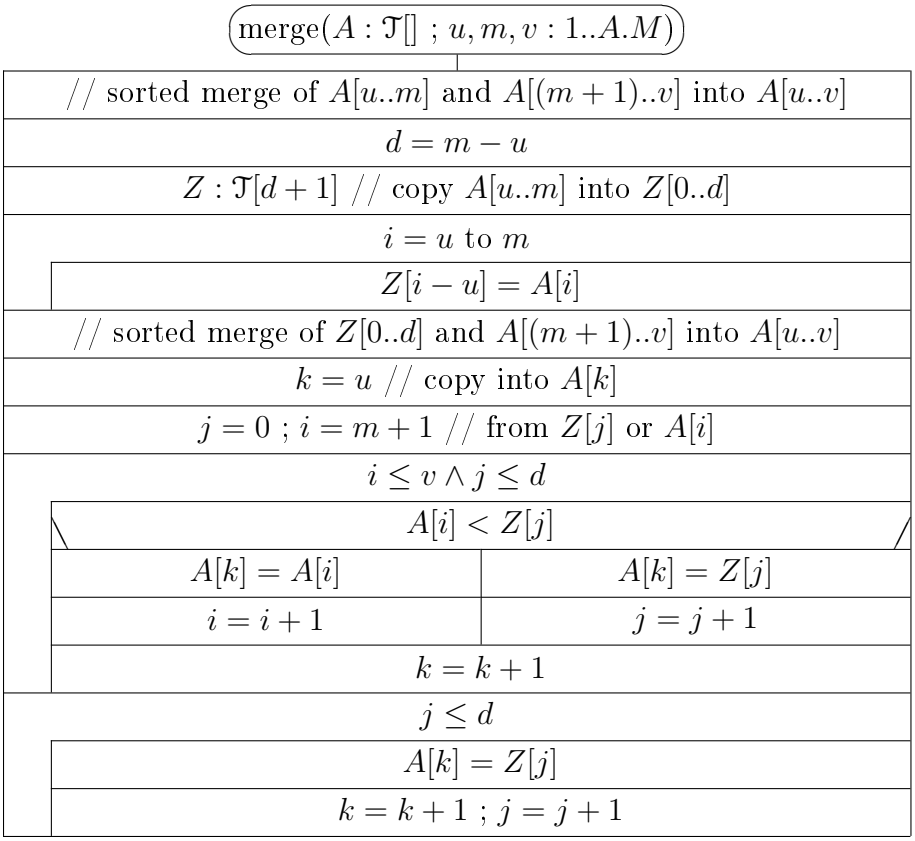
2. ábra. Az összefésülő rendezés szemléltetése.



A fenti választással $A[u..m]$ és $A[(m+1)..v]$ egyforma hosszúak lesznek, ha $A[u..v]$ páros hosszúságú, és $A[u..m]$ eggyel rövidebb lesz, mint $A[(m+1)..v]$, ha $A[u..v]$ páratlan hosszúságú, ugyanis

$$hossz(A[u..m]) = m - u + 1 = \left\lfloor \frac{u+v-1}{2} \right\rfloor - u + 1 =$$

$$\left\lfloor \frac{u+v-1}{2} - u + 1 \right\rfloor = \left\lfloor \frac{v-u+1}{2} \right\rfloor = \left\lfloor \frac{hossz(A[u..v])}{2} \right\rfloor$$



A tulajdonképpeni összefésülést a merge(A, u, m, v) eljárás második és harmadik ciklusa végzi el. A $Z[0..d]$ segédtömbre azért van szükség, hogy az összefésülés során az output ne írja felül az inputot. A triviális megoldás mindkét résztömböt átmásolná, de elég a baloldalit, mert az összefésülés során végig igaz lesz, hogy $k < i$. Ugyanis mindkét összefésülő ciklus magjában $j \leq d = m - u$, továbbá e ciklusok invariáns tulajdonsága, hogy az $A[(m+1)..v]$ résztömbből eddig $i - m - 1$ elemet másoltunk $A[u..v]$-be, míg a $Z[0..d]$ segédtömbből j elemet, míg az $A[u..v]$ résztömbbe másolt elemek száma pontosan $k - u$, ami a másik kettőből kimásolt elemek összege. Ezért

$$k - u = i - m - 1 + j \leq i - m - 1 + m - u$$

$$k - u \leq i - 1 - u$$

$$k \leq i - 1$$

Most megvizsgáljuk, miért osztottuk szét a tulajdonképpeni összefésülést két egymás utáni ciklusba. Az összefésül eljárás második ciklusa addig fut, amíg $Z[0..d]$-ben és $A[(m+1)..v]$-ben is van $A[u..v]$-be átmásolandó elem.

Ha először a $Z[0..d]$ tömbnek érünk a végére, akkor $j = d+1 = m-u+1$, és így $Z[0..d]$-ből mind a $d+1 = m-u+1$ elemet átmásoltuk $A[u..v]$-be, míg az $A[(m+1)..v]$-ből $i-m-1$ elemet másoltunk oda. Mivel összesen $k-u$ elemet másoltunk $A[u..v]$-be, így

$$k-u = (m-u+1)+(i-m-1)$$

$$k-u = i-u$$

$$k = i$$

azaz az $A[(m+1)..v]$ résztömb hátralevő elemei már a helyükön vannak. Ilyenkor az utolsó ciklus egyszer sem fog lefutni, és erre is van szükség.

Ha viszont a középső ciklusban először az $A[(m+1)..v]$ résztömbnek érünk a végére, akkor az utolsó ciklus a lehető leghatékonyabban másolja a helyükre a $Z[0..d]$ tömb még hátralévő elemeit.

Az összefésülő rendezés (merge sort) stabilitását az biztosítja, hogy a már rendezett résztömbök (általánosságban sorozatok) összefésülésekor, egyenlő kulcsok esetén a baloldali résztömbből (sorozatból) származó kulcsot tesszük először a helyére.

5.1.1. A merge eljárás műveletigénye

A merge(A, u, m, v) eljárás műveletigényének meghatározásához bevezetjük az $l = v-u+1$ jelölést. A merge eljárást csak akkor hívjuk meg, ha $u < v$, azaz $l \geq 2$. $mT_{merge}(l)$ az eljárás minimális, $MT_{merge}(l)$ a maximális műveletigénye. Most csak 1 eljáráshívás van + az 1. ciklus $\lfloor l/2 \rfloor$ iterációja + a 2. és 3. ciklus iterációi: a B vektorban lévő $\lfloor l/2 \rfloor$ elemet biztosan visszamásoljuk az A-ba, ami legalább $\lfloor l/2 \rfloor$ iteráció. Ha viszont a 2. és a 3. ciklus mindegyik elemet átmásolja, az összesen a maximális l iteráció. Innét

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

$MT_{merge}(l) \leq 1 + \left\lfloor \frac{l}{2} \right\rfloor + l \leq 2l$, ahonnét

$$l \leq mT_{merge}(l) \leq MT_{merge}(l) \leq 2l$$

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

5.1.2. A merge sort műveletigénye: szemléletes megközelítés

Az 5.1. alfejezet elején megemlítettük, hogy az összefésülő rendezés (merge sort, rövidítve MS) nagy elemszámú sorozatokat is viszonylag gyorsan rendez. Ráadásul a legjobb és a legrosszabb eset között nem mutatkozik nagy

eltérés. Első megközelítésben azt mondhatjuk, hogy a maximális és a minimális futási ideje is $n \lg n$ -nel arányos, ahol $n = A.M$. Ezt a szokásos Θ jelöléssel a következőképpen fejezzük ki.

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

A fenti összefüggést szemléletesen a következőképpen indokolhatjuk. Látható módon a műveletek túlnyomó részét a merge eljárás végzi el. Ezért az ez által végzett munkára adunk becslést, hogy az egész rendezés műveletigényének nagyságrendjét is megkapjuk. A merge(A, u, m, v) eljárás minden egyes meghívásának műveletigénye az 5.1.1. alfejezet szerint $\Theta(l)$, ahol l az aktuális résztömb hossza ($l = v - u + 1$). Mivel a rekurzív ms(A, u, v) eljárásminden hívásban felezi a rendezendő résztömb hosszát, ezért a rekurziónak kb. $\lg n + 1$ szintje van, és az alsó egy vagy két szint kivételével minden rekurziós szinten igaz az, hogy a merge hívások résztömbjei együtt lefedik az egész A tömböt. Így egy tetszőleges szint összes merge hívásának műveletigényét összeadva $\Theta(n)$ nagyságrendű műveletigény adódik (az alsó két szintet leszámítva, ahol ez kevesebb is lehet). A szintenkénti műveletigényt a szintek számával szorozva nagyságrendben $\Theta(n \lg n)$ műveletigény adódik.

A fenti műveletigény matematikailag precíz kiszámítását a 9.11. alfejezetben fogjuk elvégezni.

5.2. Gyorsrendezés (Quicksort)

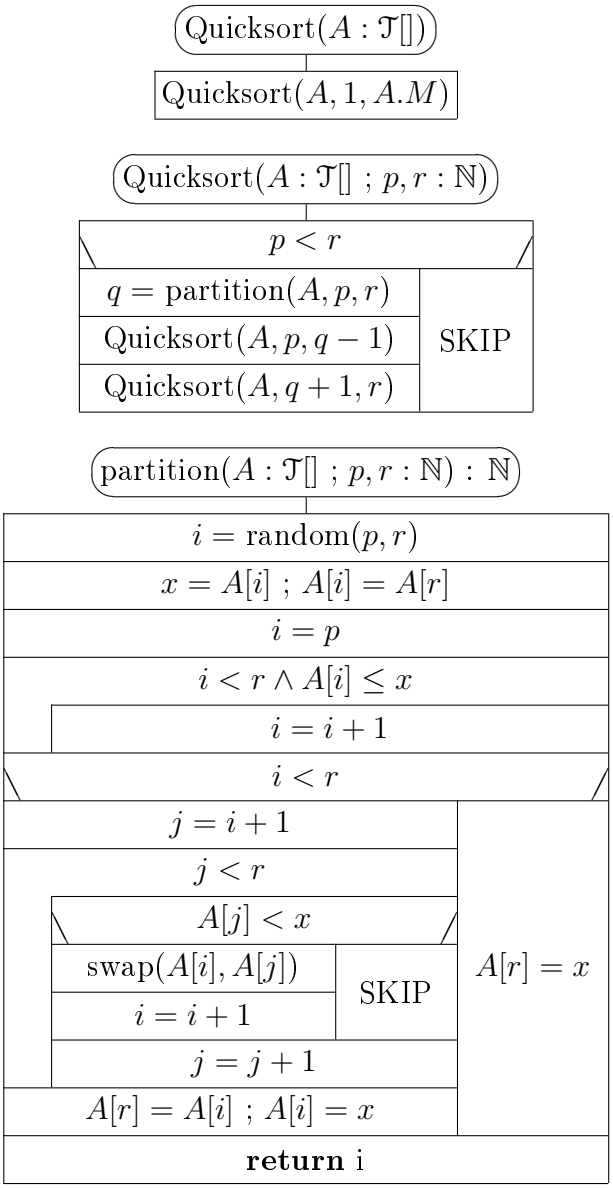
A *gyorsrendezés (quicksort)* az „oszd meg és uralkodj" elvet képviselő algoritmusok másik klasszikus példája. Tetszőleges, nagy méretű zsákból először kiválasztunk egy tengelyt (pivot), majd a maradékot két kisebb részre bontjuk: az egyik a tengelynél kisebb, a másik a nagyobb elemeket tartalmazza.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

Az algoritmus lépései vektorokra, pontokba szedve:

- Válaszd ki a rendezendő (rész)tömb egy tetszőleges elemét! Ez lesz a *tengely* (angolul *pivot*).
- Részekre bontás (partitioning): Rendezd át úgy a vektort, hogy minden, a tengelynél kisebb elem a tengely előtt, a nagyobbak pedig utána jöjjenek! (A tengellyel egyenlők bármelyik részbe kerülhetnek.) Ezzel az ún. particionálással (partition) a tengely már a végleges helyére került.

- Alkalmazd rekurzívan a fenti lépéseket, külön a tengelynél kisebb elemek résztömbjére, és külön a tengelynél nagyobb elemek résztömbjére!
- Az üres és az egyelemű résztömbök a rekurzió alapesetei. Ezek ui. már eleve készen vannak, így nem is kell őket rendezni.

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



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

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

Feltesszük, hogy az $A[p..r]$ résztömböt bontjuk részekre, és a tengely a második 5-ös, azaz a résztömb 4. (a p+3 indexű) elemét választottuk tengelynek.

A bemenet:

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

Az 1. (a kereső) ciklus előkészítése:

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

Az 1. ciklus megkeresi az első, a tengelynél (x) nagyobb elemet, ha van ilyen.

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

Találtunk a tengelynél nagyobb elemet. A j változó a következő elemre áll. Ettől a pillanattól igaz, hogy $p \leq i < j \leq r$. Felbontottuk az $A[p..r]$ részvektort négy szakaszra:
Ezek: $A[p..(i-1)], A[i..(j-1)], A[j..(r-1)]$ és $A[r]$.
Ezekre a szakaszokra az igaz, hogy
$A[p..(i-1)] \leq x \wedge A[i..(j-1)] \geq x$, $A[j..(r-1)]$ ismeretlen és $A[r]$ definiálatlan (ez a tengely üres helye). Ez a tulajdonság a $p \leq i < j \leq r$ állítással együtt a 2. ciklus invariáns tulajdonsága. (Vegyük észre, hogy a tengellyel egyenlő elemek az 1. és a 2. szakaszban is lehetnek.)) A ciklus végrehajtását elkezdve $A[j]$-ről kiderül, hogy meg kell cserélni $A[i]$-vel, hogy csatlakozhasson az $A[p..r]$ első, a tengelynél $\leq$ elemek szakaszához. Mivel az invariáns alapján $A[i] \geq x$, neki a 2. szakasz (a tengelynél $\geq$ elemek szakasza) végén is jó helye lesz. (Megvastagítottuk a megcserélendő elemeket.)

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

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

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

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

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

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

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

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

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

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

Most már az első 2 szakasz lefedi $A[p..(r-1)]$-et, és a 2. szakasz az $i < j$ invariáns miatt nemüres. Ezért a tengelyt berakhatjuk a nála $\leq$ és a nála $\geq$ elemek közé úgy, hogy a 2. szakasz első elemét az $A[p..r]$ résztömb végére tesszük, és a megürült helyre, $A[i]$-be bemásoljuk a tengelyt. (A szemléletesség kedvéért a tengelyt megvastagítottuk.)

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

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

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

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

A partition fv helyességének ellenőrzéséhez vezessük még be a következő jelöléseket:

- A_0 az A vektor kezdeti állapota, a partition függvény meghívásakor.
- $A[u..v] + x$ tömb objektum az x elemnek az $A[u..v]$ résztömb végéhez kapcsolásával adódik.

A partition fv előfeltétele: $1 \leq p < r \leq A.M$
(A p és r indexhatárok a függvényen belül természetesen konstansok.)

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

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

5.2.1. A gyorsrendezés (quicksort) műveletigénye

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

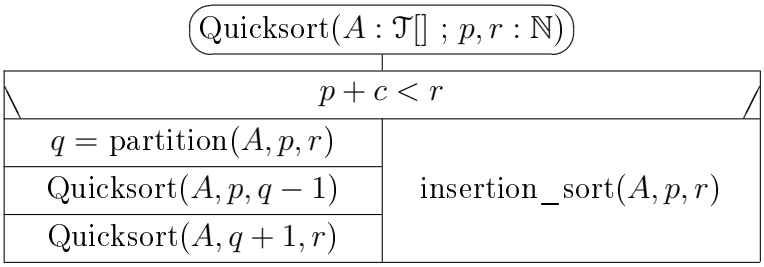
A quicksort műveletigényére ebből a következő adódik. (A részleteket ld. az MSc-n!) A várható vagy átlagos műveletigény aszimptotikusan a legjobb esethez esik közel, és a legrosszabb eset valószínűsége nagyon kicsi.

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

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

5.2.2. Vegyes gyorsrendezés

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

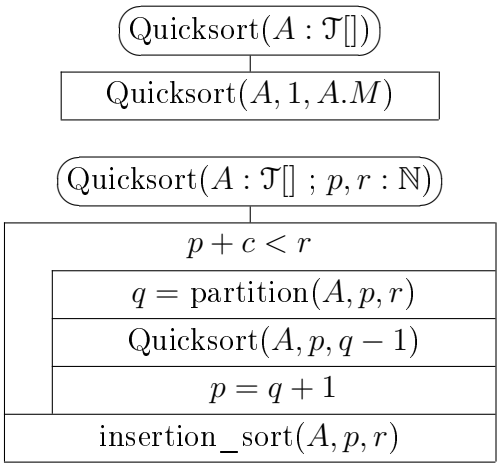


Itt $c \in \mathbb{N}$ konstans. Optimális értéke sok tényezőtől függ, de általában 20 és 40 között mozog.

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

A Quicksort legrosszabb esetének $\Theta(n^2)$ műveletigénye kiküszöbölhető, azaz biztosítható az $MT(n) \in \Theta(n \lg n)$ műveletigény, ha a vegyes gyorsrendezés rekurzív eljárásában figyeljük a rekurziós mélységet is, és pl. $2 \lg n$ mélység meghaladása esetén az aktuális résztömbre – ha még a beszúró rendezésre nem érdemes átváltani – áttérünk valamelyik olyan gyors rendezésre, ami tudja garantálni a $\Theta(n \lg n)$ legrosszabb műveletigényt. Alkalmazhatunk itt segédeljárásként pl. kupacrendezést (ld. 9.10) vagy összefésülő rendezést is. Így társíthatjuk a gyorsrendezés átlagosan legjobb futási idejét valamelyik másik gyors rendező algoritmus tökéletes megbízhatóságával.

5.2.3. A gyorsrendezés végrekurzió-optimalizált változata*



6. Elemi adatszerkezetek és adattípusok

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

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

Az *adattípus* a mi értelmezésünkben egy adatszerkezet, a rajta értelmezett műveletekkel együtt.

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

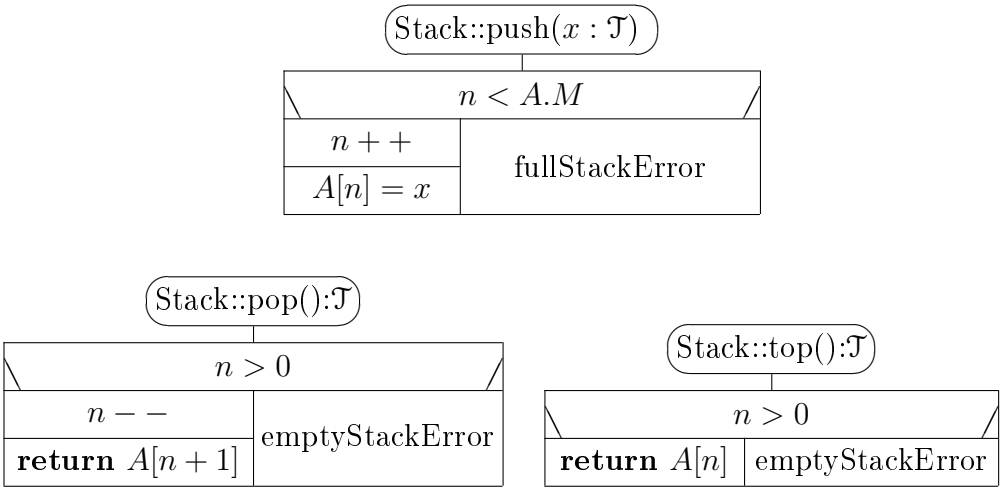
Az adattípusok megvalósítását gyakran UML jelöléssel, osztályok segítségével fogjuk leírni. A lehető legegyszerűbb nyelvi elemekre szorítkozunk. (Nem alkalmazunk sem öröklődést, sem template-eket, sem kivételkezelést.)

6.1. Vermek

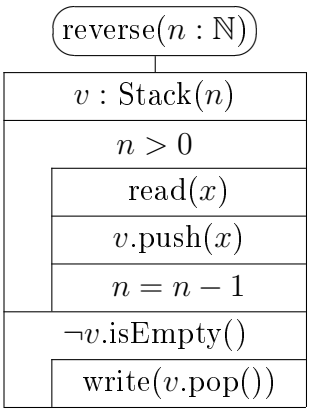
A *verem (stack)* adattípus LIFO (Last-In First-Out) adattároló, aminél tehát mindig csak az utoljára benne eltárolt, és még benne lévő adat érhető el, illetve törölhető. Tipikus műveleteit az alábbi megvalósítás mutatja.

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

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



Példa a verem egyszerű használatára a gyakorlat anyagából: n db input adat kiírása fordított sorrendben. Feltesszük, hogy a read(x) a kurrens inputról olvassa be x-be a következő input adatot. A write(x) a kurrens outpura írja x értékét.



A vermek műveleteit egyszerű, rekurziót és ciklust nem tartalmazó metódusokkal írtuk le. Ezért mindegyik műveletigénye $\Theta(1)$, ami – legalábbis együtt az összes elvégzett különféle művelet átlagos műveletigényét tekintve – alapkövetelmény minden verem megvalósítással kapcsolatban.

6.1. Feladat. *Írjuk meg a Stack osztályt dinamikusan allokált tömbbel! Ha a push művelet úgy találja, hogy már tele van a tömb, cserélje le nagyobbra, pontosan kétszer akkorára! Ügyeljünk a nagyságrendileg optimális átlagos futási időre! (Ebben az esetben a push műveletre* $mT(n) \in \Theta(1)$ *és* $MT(n) \in \Theta(n)$*, de együtt az összes elvégzett különféle művelet átlagos műveltigénye továbbra is* $\Theta(1)$ *marad.)*

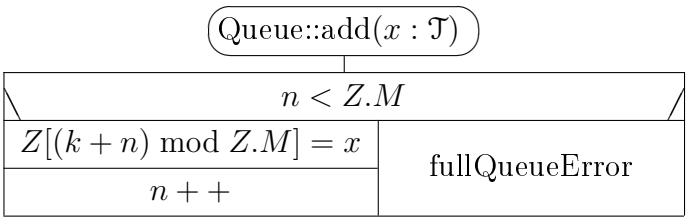
6.2. Sorok

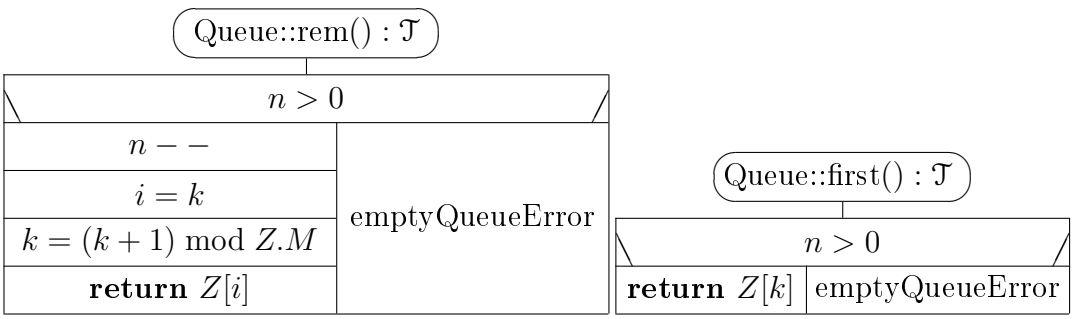
A *sor (queue)* adattípus FIFO (First-In First-Out) adattároló, aminél tehát a még benne lévő adatok közül adott pillanatban csak a legrégebben benne eltárolt érhető el, illetve törölhető. Tipikus műveleteit az alábbi megvalósítás mutatja.

A sort nullától indexelt tömb ($Z : \mathcal{T}[]$) segítségével reprezentáljuk, ahol az $Z.M$ a sor maximális mérete, $\mathcal{T}$ a sor elemeinek típusa. (A sort természetesen ábrázolhatjuk láncolt listák segítségével is (ld. a 7. fejezetet), a lista végéhez közvetlen hozzáférést biztosítva.)

Queue
$-Z : \mathcal{T}[]$ $\mathcal{T}$ is some known type $-n : \mathbb{N}$ // $n \in 0..Z.M$ is the actual length of the queue $-k : \mathbb{N}$ // $k \in 0..(Z.M-1)$ is the starting position of the queue in array Z

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





A vermek és a sorok műveleteit egyszerű, rekurziót és ciklust nem tartalmazó metódusokkal írtuk le. Ezért mindegyik műveletigénye $\Theta(1)$, ami – legalábbis együtt az összes elvégzett különféle művelet átlagos műveletigényét tekintve – alapkövetelmény minden verem és sor megvalósítással kapcsolatban.

6.2. Feladat. *Írjuk meg a Queue osztályt dinamikusan allokált tömbbel! Ha az add művelet úgy találja, hogy már tele van a tömb, cserélje le nagyobbra, pontosan kétszer akkorára! Ügyeljünk a nagyságrendileg optimális átlagos futási időre! (Ebben az esetben az add műveletre* $mT(n) \in \Theta(1)$ *és* $MT(n) \in \Theta(n)$*, de együtt az összes elvégzett különféle művelet átlagos műveltigénye továbbra is* $\Theta(1)$ *marad.)*

6.3. Feladat. *Tegyük fel, hogy adott a Stack osztály, ami a Stack(), ∼Stack(), push(x:*$\mathcal{T}$*), pop():*$\mathcal{T}$*, isEmpty():*$\mathbb{B}$ *műveletekkel (elvileg) korlátlan méretű vermeket tud létrehozni és kezelni. A destruktor művetigénye* $\Theta(n)$*, a többi műveleté* $\Theta(1)$*.*

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

6.4. Feladat. *Tegyük fel, hogy adott a Queue osztály, ami a Queue(), ∼Queue(), add(x:*$\mathcal{T}$*), rem():*$\mathcal{T}$*, length():*$\mathbb{N}$ *műveletekkel (elvileg) korlátlan méretű sort tud létrehozni és kezelni. A destruktor művetigénye* $\Theta(n)$*, a többi műveleté* $\Theta(1)$*.*

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

7. Láncolt listák (Linked Lists)

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

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

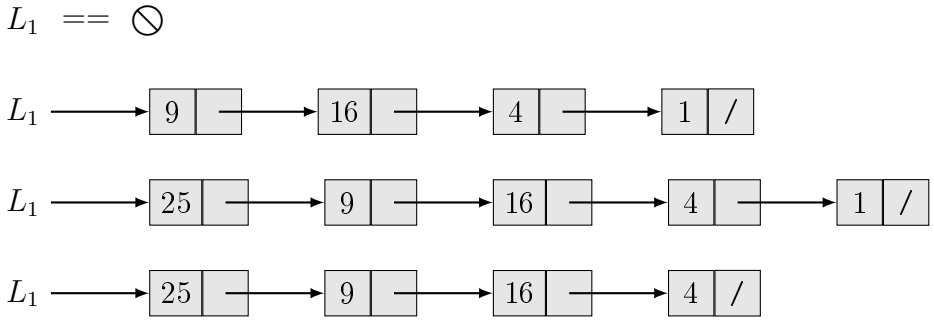
Mivel a vektorok és a láncolt listák is véges sorozatokat, azaz lineáris struktúrákat tárolnak[9], ezért ezeket *lineáris adatszerkezeteknek* nevezzük.

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

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

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

Ebben az alfejezetben egyirányú listák két legfontosabb altípusa,
– az *egyszerű egyirányú listák (S1L = Simple 1-way List)* és
– a *fejelemes egyirányú listák (H1L = Header node + 1-way List)*
részletes tárgyalására kerül sor, de bevezetjük még a végelemes és a ciklikus egyirányú listákat is.



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

Az egyirányú listák elemeinek osztálya a következő:

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

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

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

7.1.1. Egyszerű egyirányú listák (S1L)

Az L:E1* üres S1L pontosan akkor, ha $L == \oslash$.
Az L:E1* nemüres S1L pontosan akkor, ha $L \neq \oslash$, és $L \rightarrow next$ egy S1L-t

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

azonosít (ami persze lehet üres, vagy nemüres).

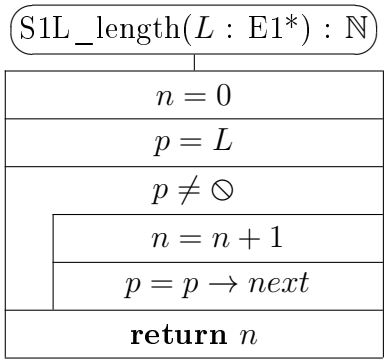
A 3. ábrán az első négy sorban egyszerű egyirányú listákra láthatunk példákat. Az első sorban az $L_1 == \oslash$ formulával az L_1 üres listát írtuk le. A második sorban látható egyszerű egyirányú lista a $\langle 9; 16; 4; 1\rangle$ sorozatot reprezentálja. Ennek megfelelően a lista négy db **E1** típusú objektumból, pontosabban listaelemből áll, amiknek a *key* mezői sorban tartalmazzák a $\langle 9; 16; 4; 1\rangle$ sorozat elemeit. Az L_1 pointer az első objektumra, azaz listaelemre mutat. Ennek megfelelően $L_1 \to key == 9$, és az $L_1 \to next$ pointer mutat a lista második elemére, amiből $L_1 \to next \to key == 16$ következik. Ha végrehajtjuk a $p = L_1 \to next \to next$ értékadást, akkor p a lista harmadik elemére mutat, $p \to key == 4$, $p \to next$ a negyedik listaelemre mutat, $p \to next \to key == 1$, $p \to next \to next == \oslash$, és p-re a $p = p \to next \to next$ értékadást is végrehajtva $p == \oslash$ lesz.

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

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

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

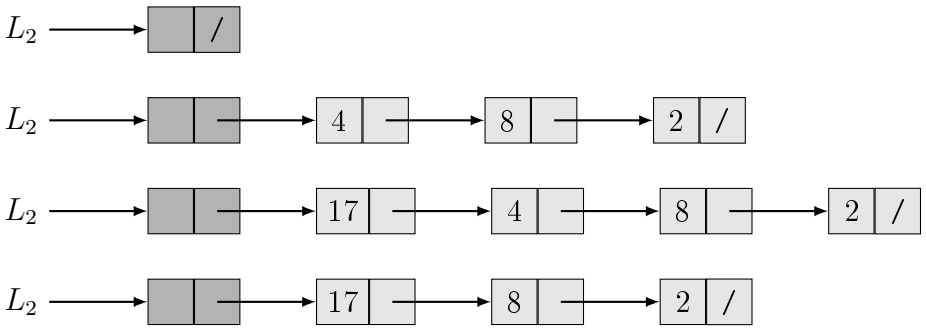
[11] A $p == \oslash$ esetet úgy is elképzelhetjük, hogy a p pointer egy úgynevezett *seholsincs objektum*ra (*NO = Nowhere Object*) mutat, ami teljesen üres, és így, ha a tartalmához akarunk hozzáférni, akkor nem létező dologra akarunk hivatkozni, ami szükségszerűen programfutási hibához vezet. (Tehát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



Ha n jelöli az L lista hosszát,
akkor
a ciklus n iterációt végez,
tehát
$T_{S1L\_length}(n) \in \Theta(n)$

7.1.2. Fejelemes listák (H1L)

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



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

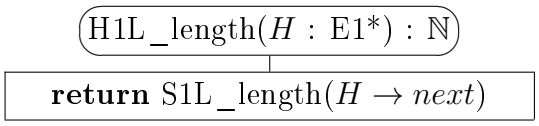
A 4. ábra első sorában L_2 üres H1L, amit az mutat, hogy a fejelem *next* mezője $\oslash$, azaz $L_2 \to next == \oslash$. Ezenkívül $L_2 \to key$ definiálatlan.[13]

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

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

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

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



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

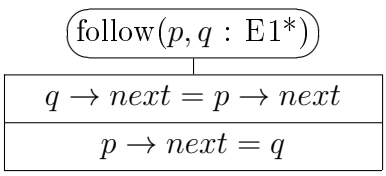
7.1.3. Egyirányú listák kezelése

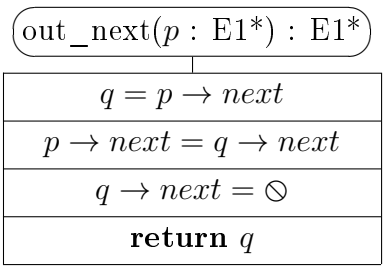
A különböző listaműveleteknél a listaelemekben levő kulcsok (és az esetleges egyéb járulékos adatok) listaelemek közti mozgatását kerüljük.

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

A follow(p, q:E1*) eljárás tetszőleges egyirányú lista $*p$ eleme után fűzi a $*q$ objektumot. Feltesszük, hogy a follow(p, q) hívás előtt $*q$ éppen nincs listába fűzve. $T_{follow} \in \Theta(1)$.

Az out_next(p:E1*):E1* függvény kifűzi tetszőleges egyirányú lista $*p$ eleme utáni elemét, és visszaadja a kifűzött elem címét. Feltesszük, hogy $*p$ valamely egyirányú lista eleme, de nem az utolsó. $T_{out\_next} \in \Theta(1)$.





A fentiek az egyirányú listák alapműveletei. Fejelemes egyirányú listák (H1L) esetén ezek segítségével minden összetett listamódosító művelet megadható. Egyszerű egyirányú listák (S1L) esetén még két további alapművelet szükséges, a lista legelejére történő beszúrásra, illetve az első elem kifűzésére. (Ezek megírását az Olvasóra bízzuk.) Emiatt a fejelemes listákat kezelő programok kódja gyakran kevesebb esetszétválasztást tartalmaz, mint a nekik megfelelő, egyszerű listákat kezelő programok, hiszen mindig valami után kell beszúrni, és mindig valami mögül kell kifűzni.

Cserébe, minden egyes fejelemes lista eggyel több objektumot tartalmaz, mint a neki megfelelő egyszerű lista, ami valamelyest megnöveli a program

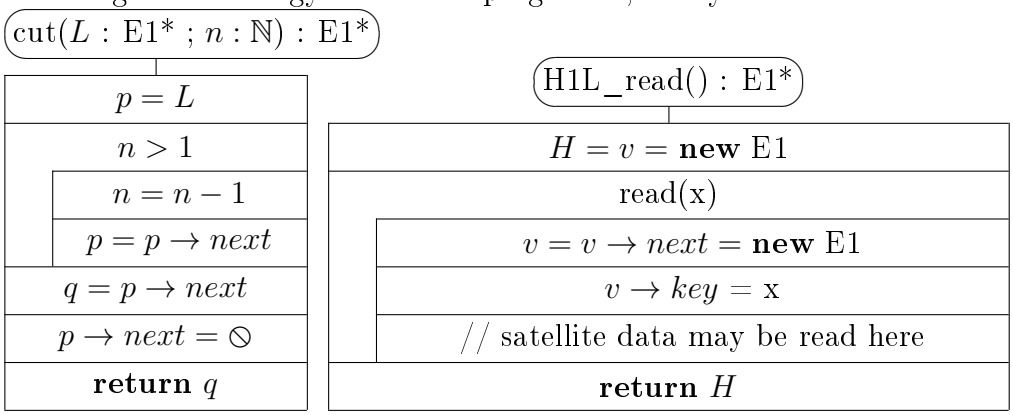
tárigényét. Ha egy alkalmazásban sok rövid listánk van, a tárigények különbsége szignifikáns lehet. Ezért pl. hasító táblákban (hash tables) sosem használunk fejelemes listákat.

Az is lehet, hogy egy (rész)feladatban a listát mindig csak a legelején, azaz veremszerűen kell módosítani; vagy olyan a feladat, hogy a lista első eleme biztosan a helyén marad. Ezekben az esetekben a fejelem csak akadály. Ha éppen van, célszerű a feladatot megoldó alprogramot a fejelemet követő egyszerű egyirányú listára (S1L) meghívni.

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

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

Most még röviden tárgyalunk két alprogramot, amelyek hasznosak lesznek.



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

A *H1L_read*():E1* függvény beolvas egy adatsort a kurrens inputról, a bemenet sorrendje szerint egy H1L-t épít belőlük, és visszaadja a fejeleme címét. Feltesszük, hogy a read($\&x$:$\mathcal{T}$):$\mathbb{B}$ függvény képes a következő adat beolvasására x-be, és akkor ad vissza igaz értéket, ha a beolvasás előtt még

volt adat a bemeneten. (Különben x definiálatlan marad, és a függvény hamis értéket ad vissza.) $T_{H1L\_read}(n) \in \Theta(n)$, ahol n a már felépített H1L hossza.

7.1. Feladat. *Próbáljuk megírni az S1L_read():E1* függvényt, amely beolvas egy adatsort a kurrens inputról, a bemenet sorrendje szerint egy S1L-t épít belőlük, és visszaadja az első eleme címét, vagy $\oslash$-t, ha az input üres volt! Tartsuk meg a $\Theta(n)$ műveletigényt! Egyszerűbb vagy bonyolultabb lett a kód, mint a H1L_read():E1* függvény esetében? Miért?*

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

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

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

Írjuk meg a listaba($A : \mathfrak{T}[]$; &H:E1) eljárást, ami előállítja az A tömb által reprezentált absztrakt sorozat láncolt ábrázolását a H H1L-ben. A szükséges listaelemeket az ismert* `new E1` *művelet segítségével nyerjük. Feltesszük, hogy ha nincs elég memória, akkor a* `new` *művelet null pointert ad vissza. Ebben az esetben írjuk ki azt, hogy „Memory Overflow!", aztán azonnal fejezzük be az eljárást! Ilyenkor a H listában csak azok az elemek legyenek, amelyeket sikeresen generáltunk! $T_{\text{listaba}}(n) \in O(n)$ legyen! (Feltesszük, hogy $T_{\text{new}} \in \Theta(1)$.)*

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

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

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

Pl.:

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

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

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

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

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

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

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

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

7.1.4. Dinamikus memóriagazdálkodás

Az objektumok dinamikus létrehozására a **new** $\mathcal{T}$ műveletet fogjuk használni, ami egy $\mathcal{T}$ típusú objektumot hoz létre, és 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** $\mathcal{T}$ utasítás végrehajtása előtt is létezik, mert azt a mutató deklarációjának kiértékelése hozza létre.[14] Ugyanígy, a p mutató a **delete** p végrehajtása után is létezik, egészen az őt (automatikusan) deklaráló eljárás vagy függvény végrehajtásának befejezéséig.

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

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

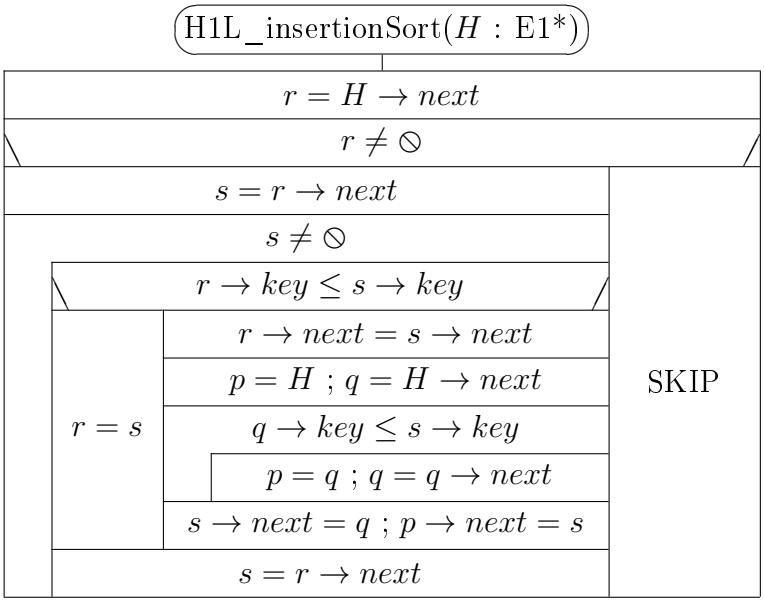
Az absztrakt programokban az objektumokat dinamikusan létrehozó (**new**) és törlő (**delete**) utasítások műveletigényeit konstans értékeknek, azaz $\Theta(1)$-nek vesszük, azzal a megjegyzéssel, hogy valójában nem tudjuk, mennyi. Ezért a lehető legkevesebbet használjuk a **new** és a **delete** utasításokat.

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

7.1.5. Beszúró rendezés H1L-ekre

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

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



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

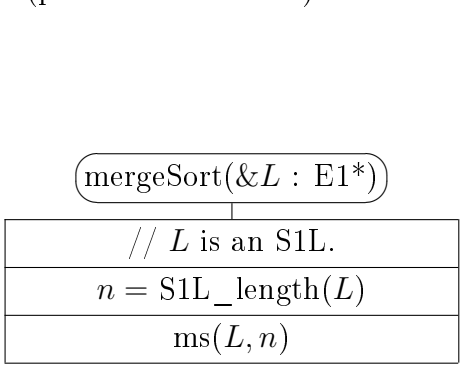
$$mT_{IS}(n) \in \Theta(n)$$

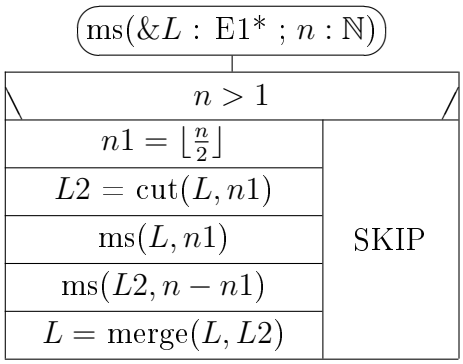
$$AT_{IS}(n), MT_{IS}(n) \in \Theta(n^2)$$

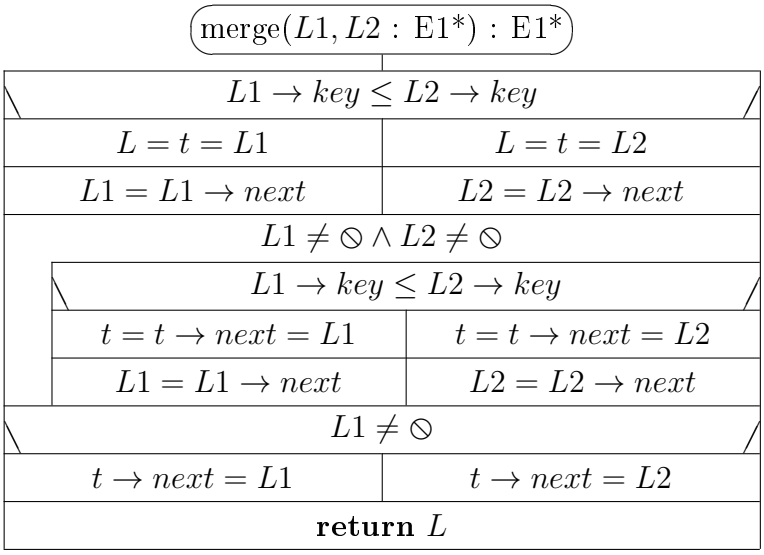
ahol n a H fejelemes lista hossza. Gondoljuk meg azt is, hogy mi biztosítja a rendezés stabilitását!

7.1.6. Az összefésülő rendezés S1L-ekre

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







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

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

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

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

7.1.7. Ciklikus egyirányú listák

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

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

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

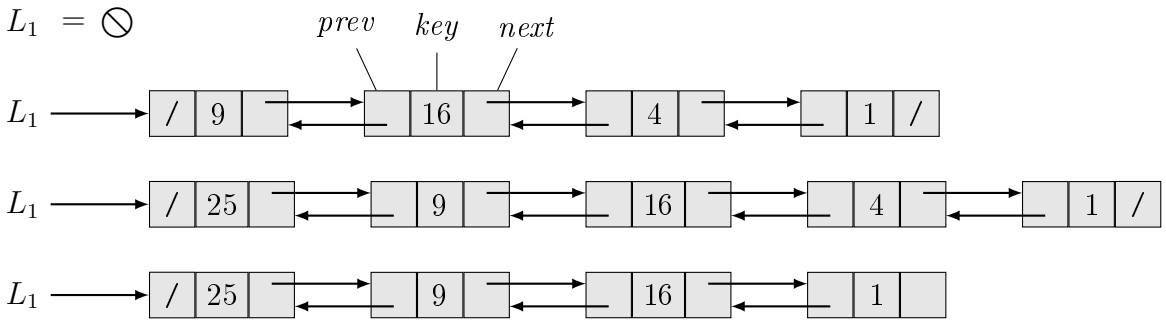
7.12. Feladat. *Valósítsuk meg a Queue (sor) adattípust (ld. (6.2) alfejezet) – egyszerű láncolt listával (S1L), úgy, hogy amennyiben a lista nemüres, az első elemére az L, az utolsó elemére a t pointer mutat, ha pedig a lista üres, akkor* $L == \oslash$, *t értéke pedig nem definiált,*
– végelemes, egyirányú, nemciklikus listával (47. oldal),
– őrszemes egyirányú ciklikus listával,
– őrszem nélküli egyirányú ciklikus listával!
Tartsuk meg a különböző megvalósítások mindegyik műveletére a $\Theta(1)$ *műveletigényt, kivéve a destruktorokat, ahol* $\Theta(n)$ *lesz az új műveletigény, n-nel a sor hosszát jelölve! Hasonlítsuk össze a négy implementációt egymással, majd a tömbös megvalósítással (6.2)!*

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

Ebben az alfejezetben a kétirányú listák két legfontosabb altípusa,
– az *egyszerű kétirányú listák (S2L = Simple 2-way List)* és
– a *fejelemes ciklikus kétirányú listák (C2L = Cyclic 2-way List with header)*
tárgyalására kerül sor, ez utóbbira részletesen.

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

7.2.1. Egyszerű kétirányú listák (S2L)



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

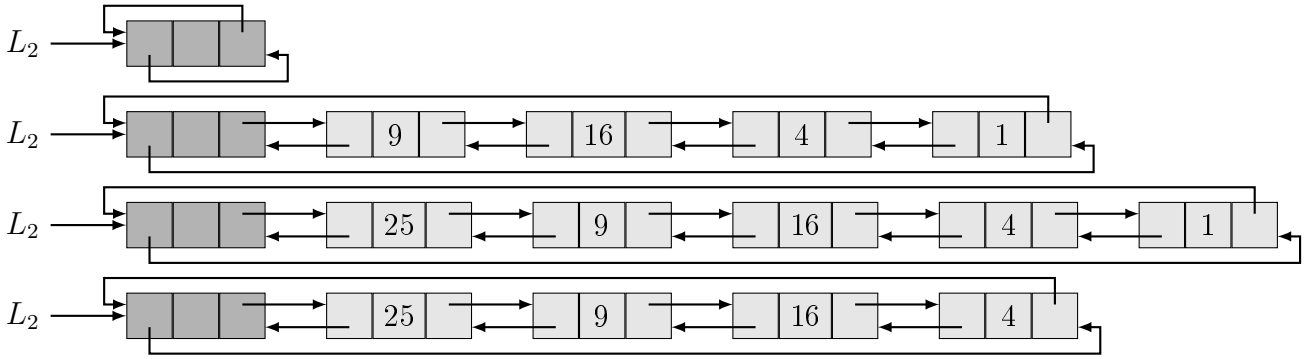
Az 5. ábra 1. sorában Az L_1 listát üresre inicializátuk, a második sorban a $\langle 9; 16; 4; 1\rangle$ sorozatot rerezentálja. A harmadik sorban beszúrtuk még a 25 kulcsú kétirányú listaelemet a lista elejére. A negyedik sorban töröltük

az utolsó előtti elemét. Látható, hogy a lista módosításai során különböző-képpen kell kezelni a lista első, utolsó és közbülső elemeit. Mindazonáltal a hasító tábláknál ez a listatípus bizonyul majd célszerűbbnek a *fejelemes ciklikus kétirányú listákhoz* képest, amiket a következő alfejezetekben tárgyalunk, és ahol a listamódosító műveletek egyszerűbbek és hatékonyabbak, mint ennél a listatípusnál.

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

7.2.2. Ciklikus kétirányú listák (C2L)

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



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

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

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

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

precede(q, r : E2*)
// insert $(*q)$ before $(*r)$
$p = r \rightarrow prev$
$q \rightarrow prev = p$; $q \rightarrow next = r$
$p \rightarrow next = r \rightarrow prev = q$

follow(p, q : E2*)
// after $(*p)$ insert $(*q)$
$r = p \rightarrow next$
$q \rightarrow prev = p$; $q \rightarrow next = r$
$p \rightarrow next = r \rightarrow prev = q$

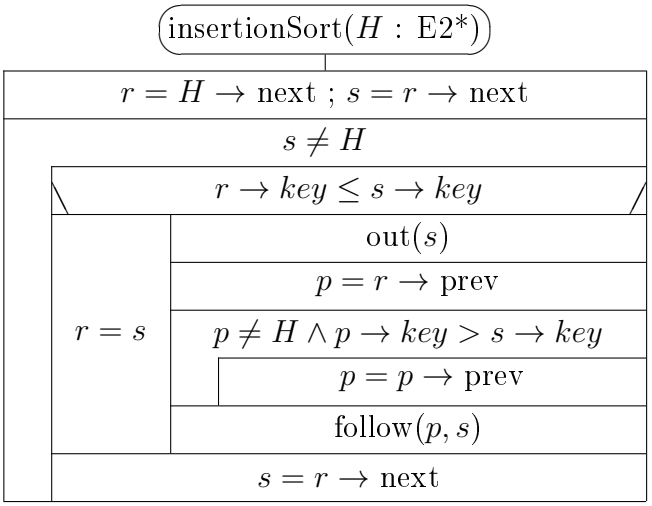
out(q : E2*)
// remove $(*q)$
$p = q \rightarrow prev$; $r = q \rightarrow next$
$p \rightarrow next = r$; $r \rightarrow prev = p$
$q \rightarrow prev = q \rightarrow next =$ **this**

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

7.2.3. Példaprogramok fejelemes, kétirányú ciklikus listákra (C2L)

A C2L listatípust szemlélteti a 6. ábra.

C2L_read($\&H$: E2*)	
$H =$ **new** E2	
read(x)	
	$p =$ **new** E2
	$p \rightarrow key = x$
	precede(p, H)



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

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

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

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

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

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

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

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

Írjuk meg a duplDel(H, D:E2) eljárást, ami a duplikált adatoknak csak az első előfordulását hagyja meg! $T(n) \in O(n)$, ahol n a H lista hossza (n a*

program számára nem adott). A lista tehát szigorúan monoton növekvő lesz. A feleslegessé váló elemekből hozzuk létre a D fejpointerű, monoton növekvő C2L-t! (A fejelem már megvan.)

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

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

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

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

`< 3;9;7;1;6;2 >` $\to$ `< 1; 3;9;7;6;2 >`
$\to$ `< 1;2; 3;9;7;6 >` $\to$ `< 1;2;3; 9;7;6 >`
$\to$ `< 1;2;3;6; 9;7 >` $\to$ `< 1;2;3;6;7; 9 >`

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

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

Írjuk meg a növBont(L, P, m*) eljárást, ami meghatározza és* m*-ben visszadja a H-beli, monoton növekvően rendezett szakaszok számát. Ezen szakaszok első elemeire az eljárás végrehajtásának eredményeként sorban a P*

tömb $P[1..m]$ résztömbjének elemei mutatnak. $MT(n) \in O(n)$, ahol n az L lista hossza (a program számára nem *adott).*

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

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

8. Függvények aszimptotikus viselkedése

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

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

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

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

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[16] függvényében végezzük a becsléseket. Így a becsléseket leíró függvények természetesen $\mathbb{N} \to \mathbb{R}$ típusúak. Megkövetelhetnénk, hogy $\mathbb{N} \to \mathbb{P}$ típusúak legyenek, de annak érdekében, hogy képleteink minél egyszerűbbek legyenek, általában megelégszünk azzal, hogy a becsléseket leíró függvények aszimptotikusan pozitívak (AP) legyenek.

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

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

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

$f \in O(g)$ *esetén azt mondjuk, hogy g* aszimptotikus felső korlátja *f-nek.*

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

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

$f \in \Omega(g)$ *esetén azt mondjuk, hogy g* aszimptotikus alsó korlátja *f-nek.*

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

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

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

$f \in \Theta(g)$ esetén tehát azt mondhatjuk, hogy g aszimptotikus alsó és felső korlátja *f-nek. (Ezt a 8.9 tulajdonságnál is láthatjuk.)*

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

8.7. Definíció.

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

Ilyenkor azt mondjuk, hogy f aszimptotikusan kisebb*, mint g. Ezt úgy is írjuk, hogy $f \in o(g)$, azaz definíció szerint:*

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

8.8. Definíció.

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

Ilyenkor azt mondjuk, hogy f aszimptotikusan nagyobb*, mint g. Ezt úgy is írjuk, hogy $f \in \omega(g)$, azaz definíció szerint:*

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

8.9. Tulajdonság. *(A függvényosztályok kapcsolata)*

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

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

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

Példa függvények nagyságrendjére:
$\lg n \prec n \prec n \lg n \prec n^2 \prec n^2 \lg n \prec n^3$

8.10. Tulajdonság. *(Tranzitivitás)*

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

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

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

$$f \prec g \wedge g \prec h \Longrightarrow f \prec h$$

$$f \succ g \wedge g \succ h \Longrightarrow f \succ h$$

8.11. Tulajdonság. *(Szimmetria)*

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

8.12. Tulajdonság. *(Felcserélt szimmetria)*

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

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

8.13. Tulajdonság. *(Aszimmetria)*

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

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

8.14. Tulajdonság. *(Reflexivitás)*

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

8.15. Következmény. *(8.11 és 8.14 alapján)*

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

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

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

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

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

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

8.18. Tulajdonság.

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

A most következő definíció tehát értelmes az előbbi tulajdonság miatt.

8.19. Definíció.

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

A függvények aszimptotikus viszonyának megállapításához hasznos az alábbi tétel.

8.20. Tétel.

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

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

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

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

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

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

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

azaz $f \in \Theta(g)$ $\square$

8.21. Következmény.

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

Bizonyítás.

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

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

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

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

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

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

□

8.22. Lemma. Az alábbi, ún. **L'Hospital szabály**t gyakran alkalmazhatjuk, amikor a 8.20. tétel szerinti $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ határértéket szeretnénk kiszámítani.

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

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

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

8.23. Következmény. *(8.20. és 8.22. alapján)*

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

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

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

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

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

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

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

Bizonyítás. Az $\varepsilon \in \mathbb{P} \Longrightarrow \lg n \prec n^{\varepsilon}$ állítás bizonyításához szükségünk lesz a L'Hospital szabályra (8.22. Lemma).

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

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

□

8.24. Következmény.

$$\Theta(\lg n) \prec \Theta(n) \prec \Theta(n * \lg n) \prec \Theta(n^2) \prec \Theta(n^2 * \lg n) \prec \Theta(n^3)$$

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

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

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

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

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

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

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

8.26. Tétel. *$f \in O(g) \iff \exists d \in \mathbb{P}$ és ψ, hogy $\lim_{n\to\infty} \frac{\psi(n)}{g(n)} = 0$, és elég nagy n-ekre*

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

8.27. Tétel. *$f \in \Omega(g) \iff \exists c \in \mathbb{P}$ és φ, hogy $\lim_{n\to\infty} \frac{varphi(n)}{g(n)} = 0$, és elég nagy n-ekre*

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

8.28. Tétel. *$f \in \Theta(g) \iff \exists c, d \in \mathbb{P}$ és φ, ψ, hogy $\lim_{n\to\infty} \frac{varphi(n)}{g(n)} = 0$, $\lim_{n\to\infty} \frac{\psi(n)}{g(n)} = 0$, és elég nagy n-ekre*

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

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

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

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

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

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

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

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

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

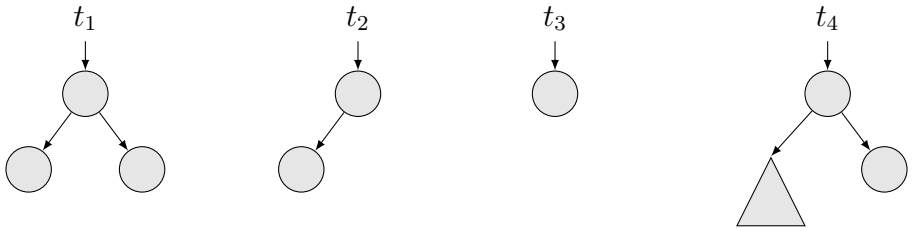
8.34. Mj. *A korábban a természetes számokon értelmezett függvényekre vonatkozó tételek az itt tárgyaltakra természetes módon általánosíthatók.*

9. Fák, bináris fák

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

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

A **bináris fák** esetében minden adatelemnek vagy szokásos nevén *csúcsnak* (angolul *node*) legfeljebb kettő rákövetkezője van: egy *bal* (*left*) és/vagy egy *jobb* (*right*) rákövetkezője. Ezeket a csúcs *gyerekeinek* (*children*) nevezzük. A csúcs a gyerekei *szülője* (*parent*), ezek pedig egymás *testvérei* (*siblings*). Ha egy csúcsnak nincs gyereke, *levélnek* (*leaf*) hívjuk, ha pedig nincs szülője, *gyökér* (*root*) csúcsnak nevezzük. *Belső csúcs* (*internal node*) alatt nem-levél csúcsot értünk. A fában egy csúcs *leszármazottai* (*descendants*) a gyerekei és a gyerekei leszármazottai. Hasonlóan, egy csúcs *ősei* (*ancestors*) a szülője és a szülője ősei. A fákat felülről lefelé szokás lerajzolni: fent van a gyökér, lent a levelek (ld. 7. ábra).



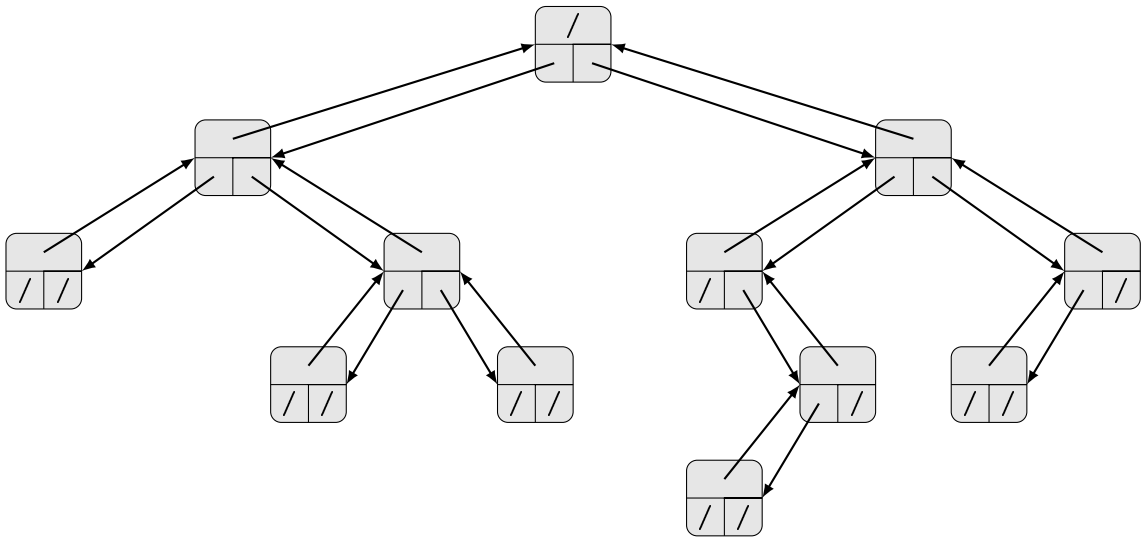
7. ábra. Egyszerű bináris fák. Egy konkrét elemét a fának körrel jelöljük. Amennyiben nem fontos, hogy milyen szerkezete van egy adott részfának, azt háromszöggel jelöljük.

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

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

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



8. ábra. Bináris fa szülő mutatóval.

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

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

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.

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 $left \sim 0$ és a $right \sim 1$ szelektor-megfeleltetést alkalmazzuk.

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

Az itt tárgyalt fákat *gyökeres fáknak* is nevezik, mert tekinthetők olyan irányított gráfoknak, amiknek az élei a gyökércsúcstól a levelek felé vannak irányítva, a gyökérből minden csúcs pontosan egy úton érhető el, és valamely $r \in \mathbb{N}$-re tetszőleges csúcs kimenő élei a $0..r-1$ szelektorok egy részhalmaza elemeivel egyértelműen[17] van címkézve. (Ezzel szemben a szabad fák összefüggő, körmentes irányítatlan gráfok.)

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

Azokat a fákat, amelyekben minden belső (azaz nem-levél) csúcsnak egy gyereke van, *listává torzult fáknak* nevezzük. Például a 8. ábrán látható bináris fa jobboldali részfájának bal részfája listává torzult. A 7. ábrán t_2 és t_3 listává torzult.

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

Például a 7. ábrán t_3 0 magasságú, t_1 1 magasságú teljes bináris fa, a 9. ábrán pedig kettő magasságú, teljes bináris fák láthatók. Megjegyezzük, hogy a fenti definíció szerint az üres fa is teljes.

Ha egy teljes bináris fa levélszintjéről nulla, egy vagy több levelet elveszünk, de nem az összeset, az eredményt *majdnem teljes bináris fának* nevezzük.

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

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

Például a 7. ábrán t_3 0 magasságú, t_2 és t_1 1 magasságú, majdnem teljes bináris fák; a 8. ábrán látható bináris fát pedig 3 magasságú, majdnem teljes bináris fává alakíthatnánk, ha a legalsó (4.) szintjén lévő csúcsát törölnénk.

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

9.2. Bináris fák mérete és magassága

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

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

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

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

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

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

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

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

A $h \leq n - 1$ egyenlőtlenséghez gondoljuk meg, hogy tetszőleges h magasságú fa szintjeit 0-tól h-ig sorszámoztuk, ami összesen $h + 1$ szintet jelent. Mivel a fa minden szintjén van legalább egy csúcs, ezért tetszőleges fára $n \geq h + 1$, azaz $h \leq n - 1$, ahol $h = n - 1$ pontosan akkor teljesül, ha a fa listává torzult. □

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

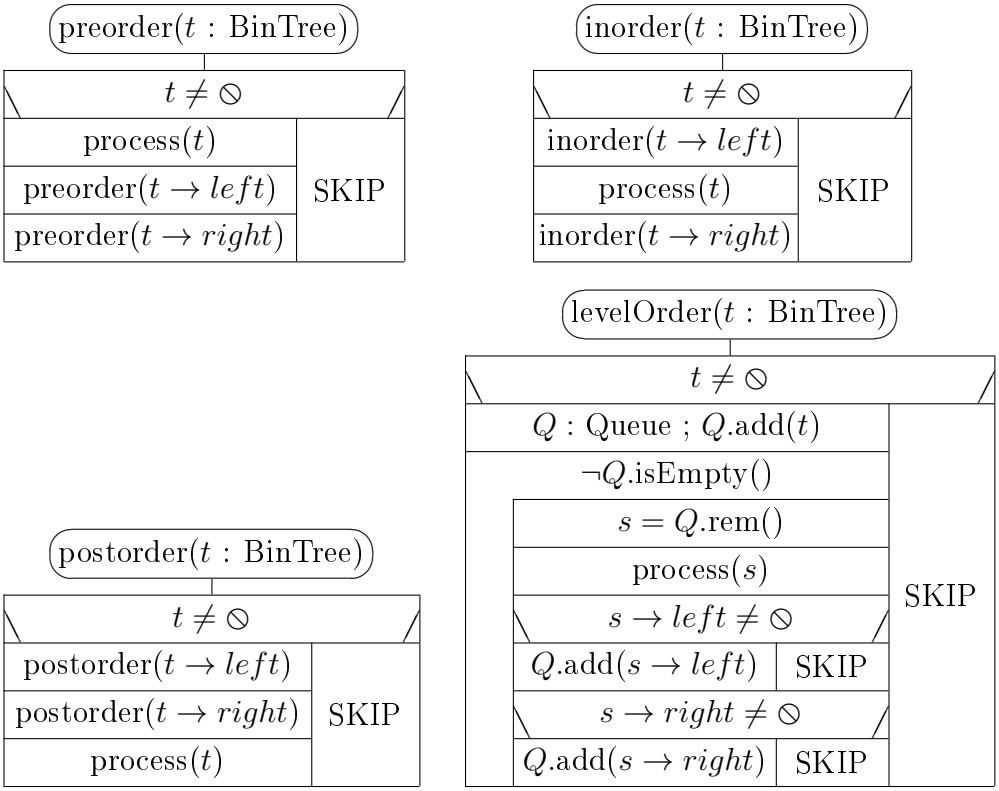
9.3. (Bináris) fák bejárásai

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 $f \to key$ kiíratása.

Üres fára mindegyik bejárás az üres program. Nemüres r-áris fákra
- a *preorder* bejárás először a fa gyökerét dolgozza fel, majd sorban bejárja a

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

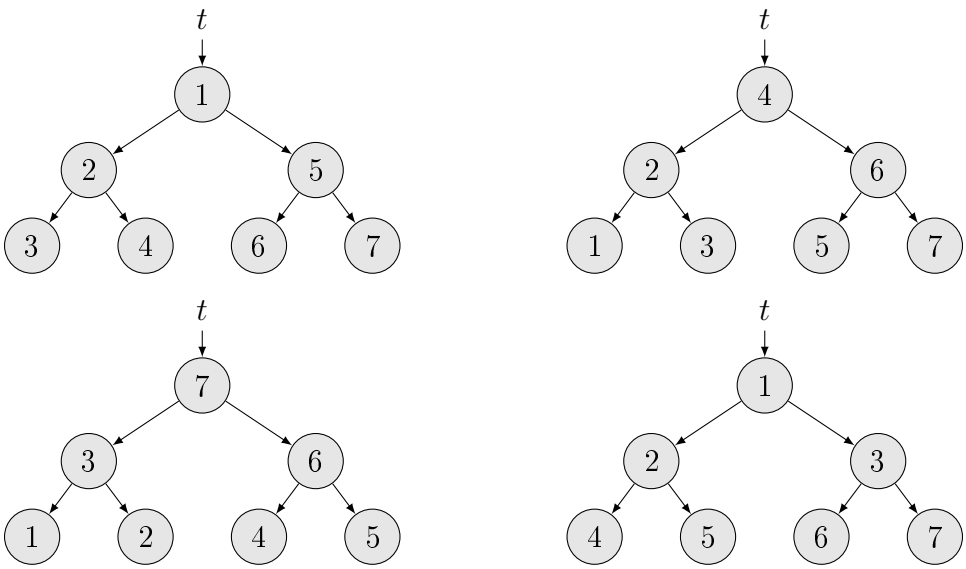
Az első három bejárás tehát nagyon hasonlít egymásra. Nevük megsúgja, hogy a gyökércsúcsot a részfákhoz képest mikor dolgozzák fel. Bináris fákra[18] az absztrakt programok a következők. (A BinTree egyenlőre a bináris fák absztrakt típusát jelöli.)



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

Igazolás: Az első három bejárás pontosan annyiszor hívódik meg, amennyi részfája van az eredeti bináris fának (az üres részfákat is beleszámolva), és egy-egy hívás végrehajtása $\Theta(1)$ futási időt igényel. Másrészt n

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

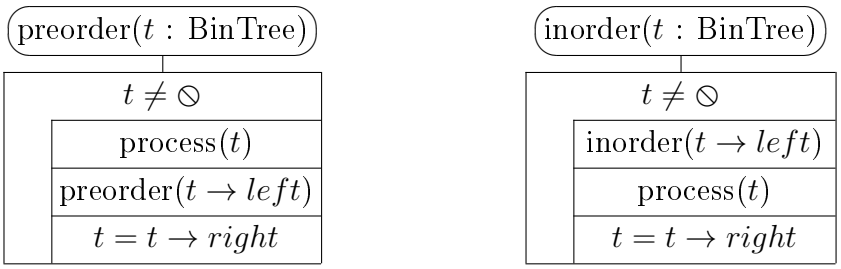


9. ábra. Bal felső sarokban preorder, jobb felsőben inorder, bal alsóban postorder, jobb alsóban szintenkénti bejárása látható a t bináris fának.

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

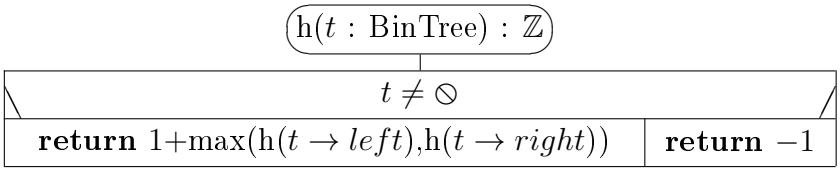
A szintenkénti bejárás helyessége azon alapszik, hogy egy fa csúcsainak szintenkénti felsorolásában a korábbi csúcs gyerekei megelőzik a későbbi csúcs gyerekeit. Ez az állítás nyilvánvaló, akár egy szinten, akár különböző szinten van a két csúcs a fában.

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

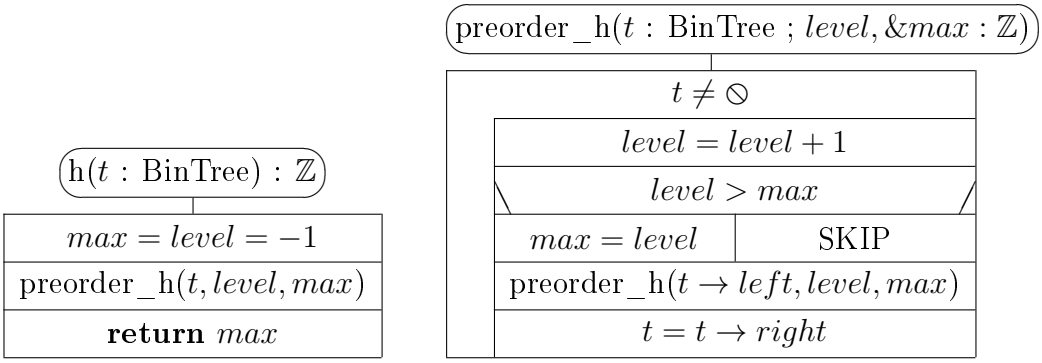


9.3.1. Fabejárások alkalmazása: bináris fa magassága

Természetesen az a legegyszerűbb, ha a definíciót kódoljuk le:



Mint látható, ez lényegében véve egy függvényként kódolt postorder bejárás.[19] Talán elsőre meglepő, de ezt a feladatot preorder bejárással is könnyen megoldhatjuk. Mint a legtöbb rekurzív programnál, itt is lesz egy nemrekurzív keret, ami előkészíti a legkülső rekurzív hívást, s a végén is biztosítja a megfelelő interfészt. Lesz két extra paraméterünk. Az egyik (*level*) azt tárolja, milyen mélyen (azaz hányadik szinten) járunk a fában. A másik (*max*) pedig azt, hogy mi a legmélyebb szint, ahol eddig jártunk. Ezután már csak a bejárás és a maximumkeresés összefésülésére van szükség.[20]



9.4. Bináris fák reprezentációi

9.4.1. Bináris fák láncolt ábrázolásai

A legtermészetesebb és az egyik leggyakrabban használt a bináris fák **láncolt ábrázolása**. Az üres fa reprezentációja a $\oslash$ pointer, jelölése tehát nem válto-

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

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

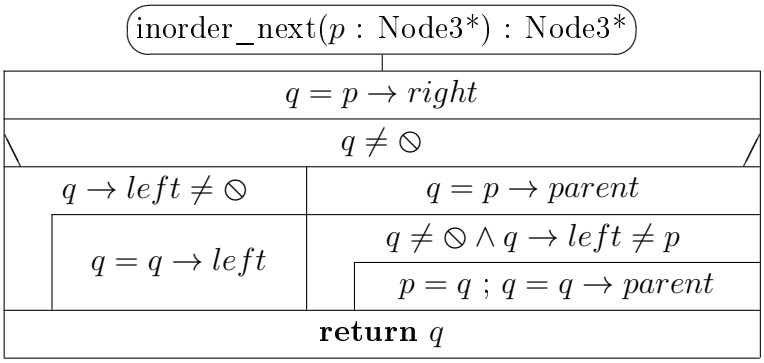
zik. A bináris fa csúcsait pl. az alábbi osztály objektumaiként ábrázolhatjuk, ahol a **BinTree** absztrakt típus reprezentációja egyszerűen a **Node***.

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

Néha hasznos, ha a csúcsokban van egy *parent* szülő pointer is, mert a fában így felfelé is tudunk haladni. A 8. ábrán megfigyelhetünk egy szülő mutatóval rendelkező bináris fát.

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

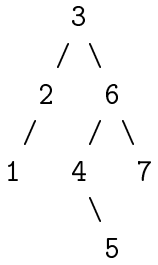
Előfordul például olyan alkalmazás, ahol szükségünk van a bináris fában egy p : Node3* pointer által mutatott csúcs ($p \neq \oslash$) inorder bejárás szerinti rákövetkezőjének címére. Ha nincs ilyen rákövetkező, a függvény $\oslash$-t ad vissza. Nyilván $MT(h(t)) \in O(h(t))$, ahol t a $*p$ csúcsot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



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

9.4.2. Bináris fák zárójelezett, szöveges formája

Tetszőleges nemüres bináris fa egyszerű zárójeles alakja:
(balRészFa Gyökér jobbRészFa)
Az üres fát az üres string reprezentálja.
A zárójeles ábrázolás lexikai elemei:
- nyitó zárójel,
- csukó zárójel
- és a csúcsok címkéi.
Például, az



bináris fa egyszerű, zárójeles alakja: „(((1) 2) 3 ((4 (5)) 6 (7)))”
Az elegáns, zárójeles alak hasonló, de többféle zárójelet használunk. Pl. ha egy részfa gyökércsúcsa a nulladik szinten van, akkor a hozzá tartozó részfát { }, ha az elsőn, akkor [ ], ha a másodikon, akkor ( ), ha a harmadikon, akkor újra { } zárójelek közé tesszük és így tovább; ha az aktuális l szintre $l \bmod 3 == 0$, akkor { }, ha $l \bmod 3 == 1$, akkor [ ], ha $l \bmod 3 == 2$, akkor akkor ( ) zárójeleket használunk. Például a fenti bináris fa elegáns, zárójeles alakja: { [(1) 2] 3 [(4 {5}) 6 (7)] }.

9.4.3. Bináris fák aritmetikai ábrázolása

A részleteket ld.: 9.7, 9.8.

9.5. 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:

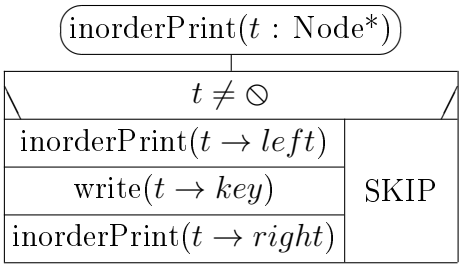
- Ha x egy tetszőleges csúcs kulcsa az r bal részfájából, akkor $x \leq y$.
- Ha z egy tetszőleges csúcs kulcsa az r jobb részfájából, akkor $z \geq y$.

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 $H(\oslash) = \{\}$, illetve $H(t) = H(t \to left) \cup \{t \to key\} \cup H(t \to right)$, amennyiben $t \neq \oslash$.

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



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

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

A fenti feladat 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 **Node** típusúak, és az üres fát a $\oslash$ pointer reprezentálja. (Ld. a **Bináris fák reprezentációi: láncolt ábrázolás** fejezetet!)

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

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

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

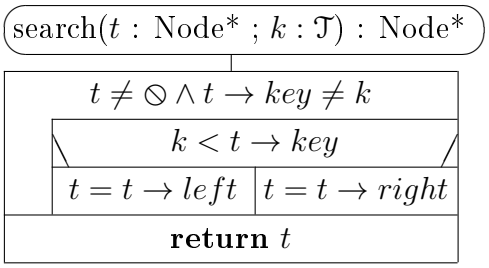
A min(t) függvény a t nemüres fa "bal alsó" csúcsára hivatkozó pointerrel tér vissza.

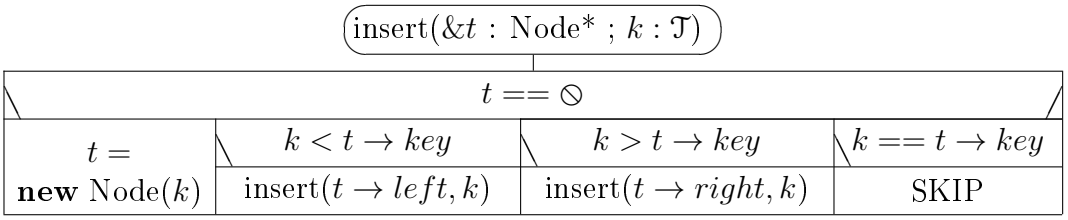
A remMin$(t, minp)$ eljárás $minp$-ben a t nemüres fa "bal alsó" (a legkisebb kulcsot tartalmazó) csúcsára mutató pointerrel tér vissza, de még előtte a csúcsot kifűzi a fából, azaz a csúcshoz tartozó részfa helyére teszi a csúcs jobboldali részfáját.

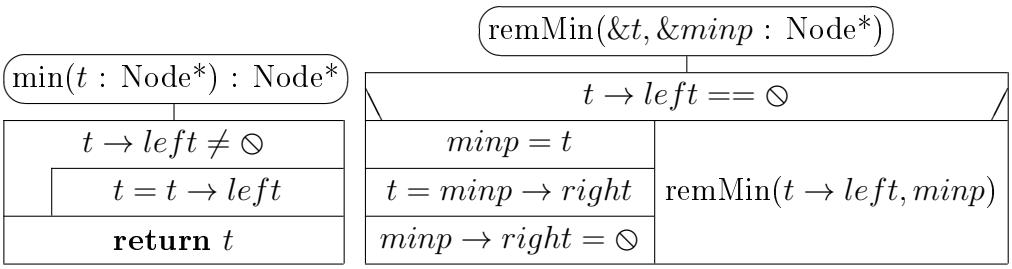
A del(t, k) eljárás szintén megkeresi a t fában a k kulcs helyét. Ha megtalálta a k kulcsot tartalmazó csúcsot, még két eset lehet. (1) Ha a csúcs egyik részfája üres, akkor a csúcshoz tartozó részfa helyére teszi a csúcs másik részfáját. (2) Ha a csúcsnak két gyereke van, akkor a minKivesz eljárás segítségével *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

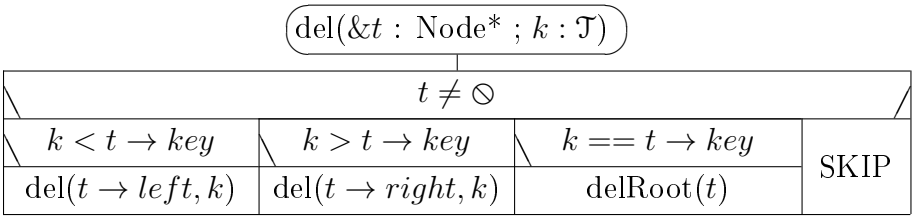
jobboldali részfa maradék kulcsainál pedig kisebb. (Vegyük észre, hogy a (2) esetben a baloldali részfából a maximális kulcsú csúcsot is kivehetnénk!)

9.5. Feladat. *Írjuk meg a $t \neq \oslash$ bináris keresőfából a maximális kulcsú csúcs 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!*









delRoot(&t : Node*)

$t \to left == \oslash$	$t \to right == \oslash$	$t \to left \neq \oslash \wedge t \to right \neq \oslash$
$p = t$	$p = t$	remMin$(t \to right, p)$
$t = p \to right$	$t = p \to left$	$p \to left = t \to left$; $p \to right = t \to right$
delete p	**delete** p	delete t ; $t = p$

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.

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 $O(n)$ átlagos műveletigényt tudnánk garantálni.

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

Sok alkalmazás esetén azonban nem megengedhető az a kockázat, hogy ha a keresőfa (majdnem) listává torzul, akkor a műveletek hatékonysága is hasonló lesz, mint a láncolt listák esetén. Az ideális megoldás az lenne, ha tudnánk garantálni, hogy a fa majdnem teljes legyen. Nem ismerünk azonban olyan $O(\lg n)$ 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 a következő félévben ez utóbbi két keresőfa típus tárgyalásával folytatjuk.

9.7. Szintfolytonos bináris fák, kupacok

A feladatok előtt felidézzük a teljes és a majdnem teljes fákkal kapcsolatos tudnivalókat.

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.

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

9.8. Feladat. *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!*

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

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 az utolsó kivételével biztosan két-két gyereke van. Ha az utolsónak csak egy gyereke van, akkor ez bal gyerek. A szint jobb szélén lehetnek levelek. A magasabb szinteken minden csúcsnak két gyereke van.

A balra tömörített, majnem teljes bináris fákat más néven *szintfolytonos* bináris fáknak is nevezzük (hiszen, tetszőleges ilyen fát szintenként balról

jobbra bejárva, egészen a legutolsó csúcsáig egyetlen csúcs sem hiányzik, a vele azonos magasságú teljes bináris fához viszonyítva).

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

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

9.8. Szintfolytonos bináris fák aritmetikai ábrázolása

A szintfolytonos bináris fákat, speciálisan a kupacokat, szokás szintfolytonosan, egy vektorban ábrázolni. Ha pl. egy $A : \mathcal{T}[m]$ tömb első n elemét használjuk, akkor az i indexű csúcs gyerekeinek indexei $2i$, illetve $2i+1$, feltéve, hogy a gyerekek léteznek, azaz $2i \leq n$, illetve $2i+1 \leq n$. 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 $\lfloor \frac{j}{2} \rfloor = i$.

9.10. Feladat. *Bizonyítsuk be, hogy ha egy n elemű, szintfolytonos bináris fát a nullától indexelt $Z : \mathcal{T}[m]$ tömbben szintfolytonosan tárolunk, akkor a $Z[i]$ csúcs gyerekei $Z[2i+1]$ illetve $Z[2i+2]$, a $2i+1 < n$, illetve a $2i+2 < n$ feltétellel, a $Z[j]$ csúcs szülője pedig $j > 0$ esetén $Z[\lfloor \frac{j-1}{2} \rfloor]$.*

9.9. Kupacok és elsőbbségi (prioritásos) sorok

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

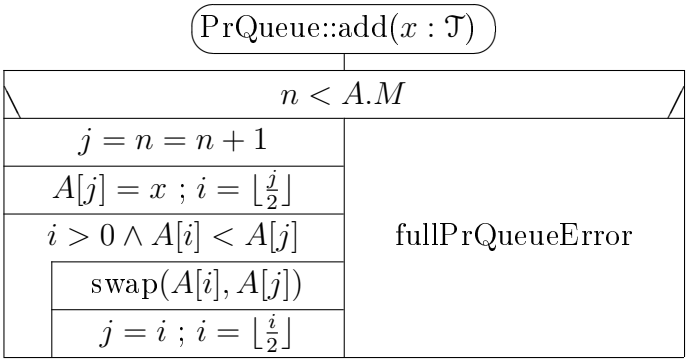
Az alábbiakban a prioritásos sor típust a PrQueue osztály segítségével írjuk le. Az elsőbbségi sor aktuális elemeit az $A[1..n]$ résztömb tartalmazza, ami egy kupac.

PrQueue
- $A : \mathcal{T}[]$ // $\mathcal{T}$ is some known type - $n : \mathbb{N}$ // $n \in 0..A.M$ is the actual length of the priority queue

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

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

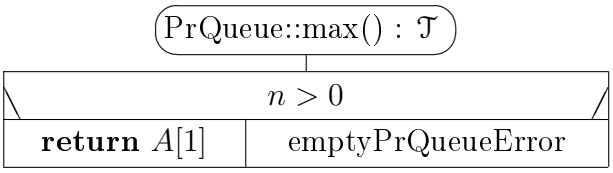
Az add(x) esetében a szintfolytonosan első üres helyen x-et hozzákapcsoljuk a kupachoz ($A[j]$). Ezzel elronthatjuk a kupacot, ezért x-et addig cserélgetjük – mindig az aktuális szülőjével ($A[i]$) – amíg van szülője, és $x >$ mint a szülő. Így a lyuk felfelé mozog a kupacban, és $x >$ mint a leszármazottai.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

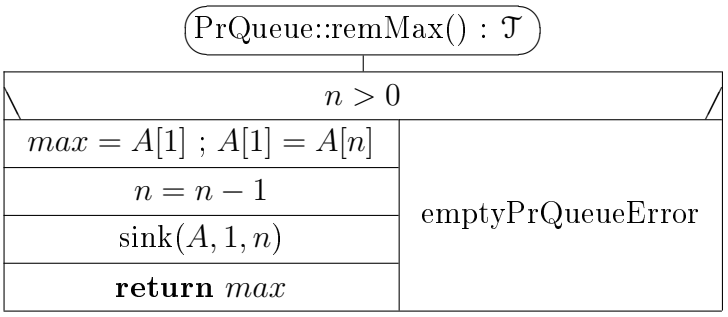


$MT_{\text{add}}(n) \in \Theta(\lg n)$; hiszen a ciklus legfeljebb annyiszor fut le, amennyi a fa magassága, azaz (n input értékéhez viszonyítva) $\lfloor \lg(n+1) \rfloor$-szer, amiből $\lg n \leq MT_{\text{add}}(n) = \lfloor \lg(n+1) \rfloor + 1 \leq \lg n + 2$.

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

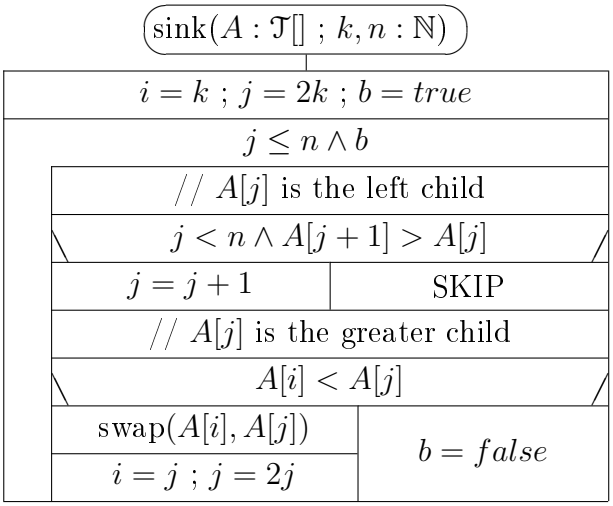
$T_{\max}(n) \in \Theta(1)$, mivel $T_{\max}(n) = 1$.





A remMax() metódus a maximum elmentése után a kupac gyökerébe, $A[1]$-be teszi át a szintfolytonosan utolsó elemet. Ezzel a kupac mérete eggyel csökken, és a gyökerénél valószínűleg el is romlik (ún. csonka kupac lesz). Ezért a "sink" eljárás segítségével (amit mindig $k = 1$-gyel hív meg) a gyökérbe átrakott elemet ($A[i]$) addig süllyeszti lefelé, amíg a helyére nem kerül. Ennek során a lesüllyesztendő elemet mindig az aktuálisan nagyobb gyerekével cseréli meg, amíg még a levélszint fölött van, és a nagyobbik gyereke nagyobb nála. Ilyen módon a ciklus során a lesüllyedő elem mindig kisebb, mint az ősei. Amikor a ciklus megáll, akkor vagy leért a levélszintre, vagy $\geq$ mint a gyerekei, és kupac helyreáll.

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



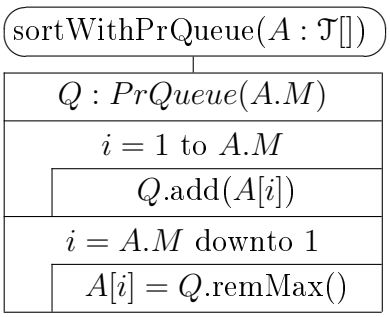
A remMax() metódus műveletigényének meghatározásához először megvizsgáljuk a sink() eljárást. Ha a k gyökerű lyukas kupac magassága h, akkor $MT_{\text{sink}}(h) = h + 1$, ugyanis a ciklus legfeljebb h iterációt végez, és $1 \leq mT_{\text{sink}}(h) \leq 2$, mert lehet, hogy a ciklus legfeljebb egyet iterál. Speciálisan $k = 1$ esetére: $\lg n \leq MT_{\text{sink1}}(n) = h + 1 = \lfloor \lg n \rfloor + 1 \leq \lg n + 1$.

Innét n input értékére $\lg n \leq \lg(n-1) + 1 \leq MT_{\text{remMax}}(n) \leq \lg(n-1) + 2 \leq \lg n + 2$.
Ebből $MT_{\text{remMax}}(n) \in \Theta(\lg n)$.

$mT_{\text{remMax}}(n) \in \Theta(1)$, hiszen
$2 = 1 + 1 \leq mT_{\text{remMax}}(n) = 1 + mT_{\text{sink1}}(n-1) \leq 1 + 2 = 3$.

9.9.1. Rendezés elsőbbségi sorral

A fenti elsőbbségi sor segítségével könnyen és hatékonyan rendezhetünk tömböket:



A fenti rendezésben a szokásos $n = A.M$ jelöléssel a Q.add($A[i]$) és az $A[i] = Q$.remMax() 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 maximális műveletigénye tehát aszimptotikusan kisebb, mint a beszúró rendezésé, ami $\Theta(n^2)$. Ráadásul az $\frac{n \lg n}{n^2}$ hányados már $n = 1000$ esetén is csak ≈ 0.01, $n = 10^6$ esetén pedig ≈ 0.00003, tehát gyorsan tart a nullához. A sortWithPrQueue hátránya, hogy a rendezendő vektorral azonos méretű $M(n) \in \Theta(n)$ munkamemóriát igényel, a prioritásos sorban tárolt kupac számára, míg a beszúró rendezésre $M(n) \in \Theta(1)$, hiszen csak néhány egyszerű segédváltozóra van szükségünk. Ezt a problémát kupacrendezéssel oldhatjuk meg.

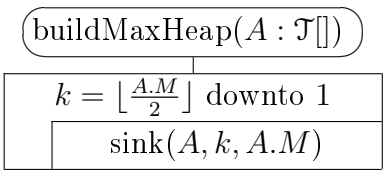
9.10. Kupacrendezés (heap sort)

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

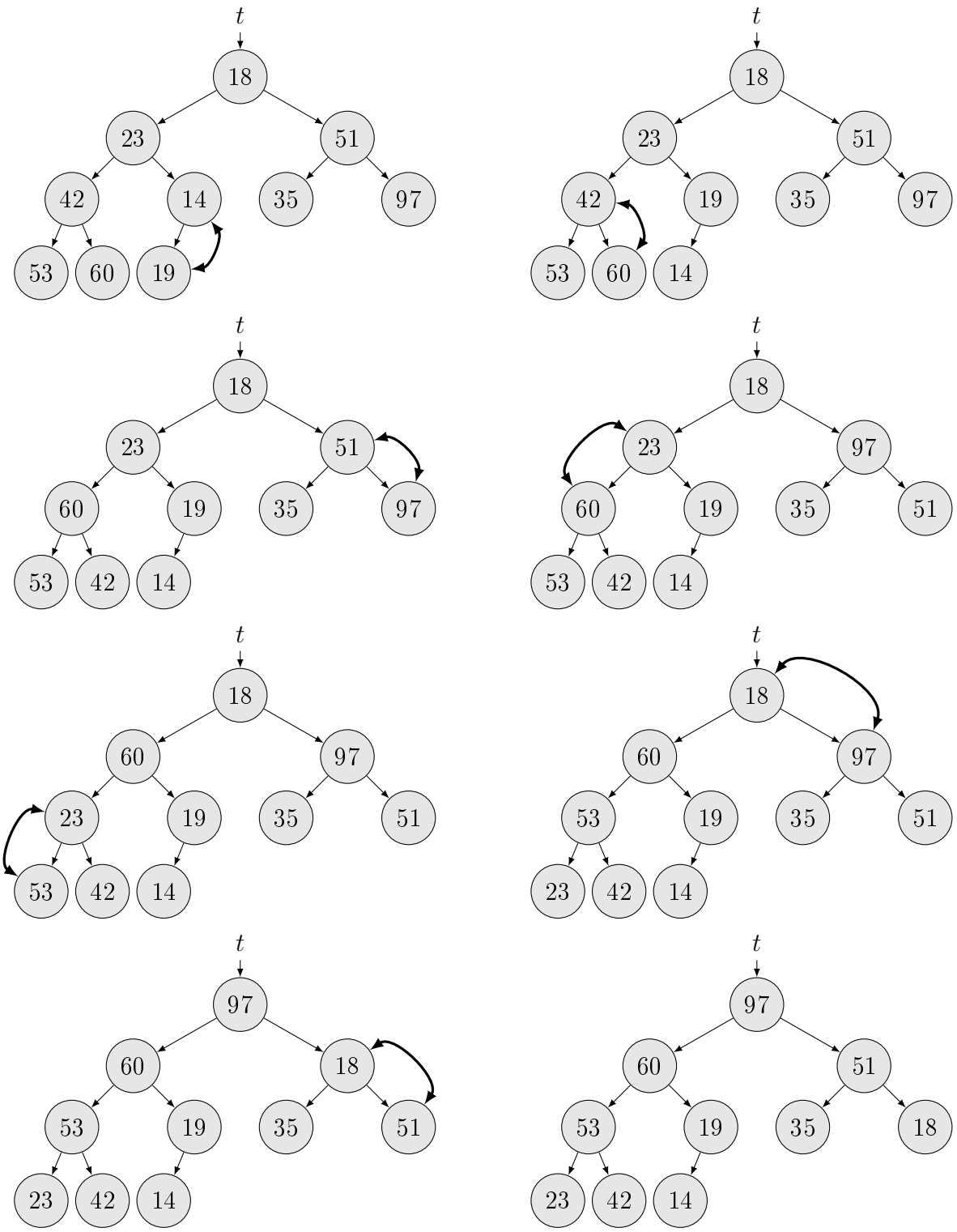
Egyrészt az algoritmust *helyben rendezővé* alakítjuk: az $A : \mathcal{T}[n]$ tömbön kívül csak $\Theta(1)$ memóriát használunk, míg a fenti eljárásnak szüksége van egy $\Theta(n)$ méretű segédtömbre, amiben a prioritásos sort ábrázoló kupacot tárolja.

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

Először tehát kupaccá alakítjuk a tömböt, most lineáris műveletigénnyel:



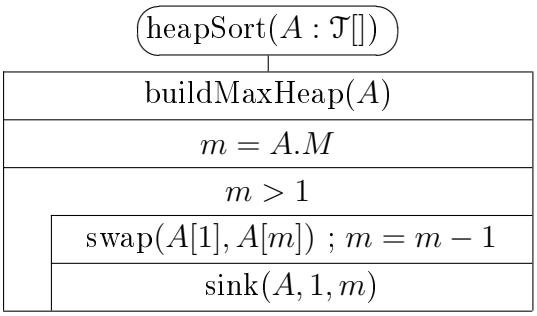
A fenti eljárás magyarázatához: Eredetileg az $A : \mathcal{T}[n]$ tömb, mint balra tömörített, majdnem teljes bináris fa magassága $h = \lfloor \lg n \rfloor$. Levelei önmagukban egyelemű kupacok. A $(h-1)$-edik szinten levő belső csúcsokhoz, mint gyökerekhez tartozó bináris fák tehát (egy magasságú)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



10. ábra. A $\langle 18, 23, 51, 42, 14, 35, 97, 53, 61, 19\rangle$ tömb kupaccá alakítása. Jegyezzük meg, hogy végig tömbbel dolgozunk, a bináris fák csak a szemléltetéshez használatosak. Végeredményben a $\langle 97, 60, 51, 53, 19, 35, 18, 23, 42, 14\rangle$ tömböt kapjuk.

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 $A : \mathcal{T}[n]$ 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:



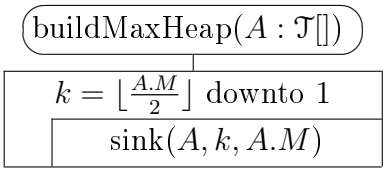
A kupaccá alakítás után tehát ezt ismételjük: Megcseréljük a kupac maximumát, azaz gyökerében lévő elemet ($A[1]$) a kupac szintfolytonosan utolsó elemével, levágjuk a maximumot, majd lesüllyesztjük a helyére tett elemet. Így az előbbinél eggyel kisebb méretű kupacot kapunk. Ezt a ciklusmagot addig ismételjük, amíg a kupac mérete nagyobb, mint egy. Így az A tömbben visszafelé megkapjuk az első, második stb. maximumokat, és végül az $A[1]$-ben a minimumot, azaz a tömb rendezve lesz.

9.10.1. A kupacrendezés műveletigénye

A kupaccá alakítás utáni rész futási idejét a sink$(A, 1, m)$ hívások határozzák meg, amiknek műveletigénye $O(\lg m)$ $(m = n-1, n-2, \ldots, 1)$, durva felső becsléssel $O(\lg n)$, ahol $n = A.M$. 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 alább 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 sortWithPrQueue(A) eljárás kupacépítő ciklusához képest hatékonyabb kupaccá alakítással a gyakorlatban így is jelentős futási időt takaríthatunk meg.



9.11. Feladat. *Lássuk be, hogy a kupacrendezés maximális műveletigénye* $\Theta(n \lg n)$*, ahol* $n = A.M$*, és ez például akkor áll elő, amikor az input tömb szigorúan monoton csökkenő!*

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

9.10.2. A kupaccá alakítás műveletigénye lineáris

Annak belátásához, hogy a buildMaxHeap($A : \mathcal{T}[]$) műveletigénye $\Theta(n)$, ahol $n = A.M$, szükségünk lesz a balra tömörített, majdnem teljes bináris fák néhány tulajdonságára.

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

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

Eszerint egy n csúcsú, 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 előbb 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 gyökércsúcsai éppen a belső csúcsok. Továbbá $n_{\geq 2} = fele(fele(n)) = fele^2(n)$, hiszen a legalább kettő magasságú részfák gyökércsúcsai éppen a legalább egy magasságú részfák gyökércsúcsainak szülei, és mivel az utóbbiak szinfolytonosan $1..fele(n)$ sorszámúak, azért az előbbiek $1..fele^2(n)$ sorszámúak. Teljes indukcióval adódik

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

Most már áttérhetünk a kupaccá alakítás lineáris műveletigényének bizonyítására.

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

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

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

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

Vegyük most figyelembe, hogy $\sum_{d=1}^{h} d * n_d$ az alábbi alakba írható:
n_1+
$n_2 + n_2+$
$n_3 + n_3 + n_3+$
$\ldots$
$n_h + n_h + n_h + \ldots + n_h \quad$ (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

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

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

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

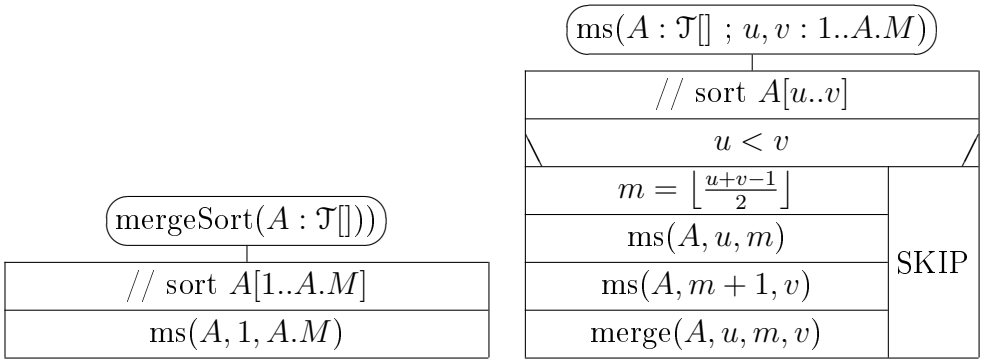
adódik.

9.11. A merge sort műveletigényének kiszámítása

Most, hogy már otthonosan mozgunk a bináris fák világában, újra megpróbálkozhatunk a címben jelzett feladat kivitelezésével. Az 5.1. alfejezetben azt állítottuk, hogy az összefésülő rendezés (merge sort, rövidítve MS) tömbös változatának műveletigényére $n =$ a rendezendő vektor mérete jelöléssel az alábbi összefüggés igaz.

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

Tekintsük ehhez az összefésülő rendezés tömbös változatának felső szintjét!



9.13. Feladat. *Vegyük észre, hogy a fenti rekurzív program ms(...) eljáráshívásai szigorúan bináris fát alkotnak, és bizonyítsuk be, hogy – ebből következően – a fenti merge sort algoritmus tetszőleges $n > 0$ méretű input vektor esetén pontosan $2n - 1$ -szer hívja meg az ms(...) rekurzív eljárást, ahol az ms(...) hívások fájának $n - 1$ belső csúcsa és n levele van.*

9.14. Feladat. *Egy bináris fa a* levelek száma szerint kiegyensúlyozott, *ha tetszőleges nemüres részfája bal és jobb részfája leveleinek száma legfeljebb eggyel térhet el.*

Bizonyítsuk be, hogy a levelek száma szerint kiegyensúlyozott szigorúan bináris fák halmaza a majdnem teljes bináris fák halmazának valódi részhalmaza! (Alkalmazhatunk például a fa magassága szerinti teljes indukciót!

Vegyük észre, hogy a fenti rekurzív program ms(...) eljáráshívásai a levelek száma szerint kiegyensúlyozott szigorúan bináris fát alkotnak, és – ebből következően – az ms(...) hívások fája majdnem teljes, és a 9.13. feladat szerint $n > 0$ méretű input vektor esetén pontosan $2n - 1$ csúcsa van, így a magasságára
$\lfloor \lg n \rfloor \leq h = \lfloor \lg(2n-1) \rfloor \leq \lfloor \lg(2n) \rfloor = \lfloor \lg(n) + 1 \rfloor = \lfloor \lg n \rfloor + 1$,
tehát $\lfloor \lg n \rfloor \leq h \leq \lfloor \lg n \rfloor + 1$.

A 9.13. feladat szerint tehát $n > 0$ méretű input vektor esetén, az ms(...) hívások fájának $2n - 1$ csúcsa van, így a teljes merge sort műveletigénye a merge(...) eljárás végrehajtásai nélkül $2n$.

Az 5.1.1. alfejezetben beláttuk, hogy $l = v - u + 1$ jelöléssel az összefésülő rendezésben a merge(A, u, m, v) eljárás végrehajtásának műveletigényére

$$l \leq mT_{merge}(l) \leq MT_{merge}(l) \leq 2l$$

Most a merge(A, u, m, v) eljárás összes végrehajtásának teljes műveletigényére adunk becsléseket. Ehhez vegyük észre, hogy az ms(A, u, v) rekurzív

eljárás hívásai majdnem teljes fájának leveleiben $u = v$, míg belső csúcsaiban $u < v$, tehát ez utóbbiakban fog meghívódni a merge(A, u, m, v) eljárás. Az ms$(\ldots)$ hívások fájának alsó szintjén tehát nem hívódik meg a merge(A, u, m, v) eljárás. Egy szinttel feljebb valahányszor meghívódik, míg a magasabb szinteken mindig meghívódik. A legalsó szinttől eltekintve bármelyik szintre igaz az, hogy az adott szint ms$(\ldots)$ hívásai együtt lefedik a teljes A vektort, azaz az egyes hívások által lefedett részvektorok összhossza pontosan n.

Az alsó két szinttől eltekintve minden szinten, az ms(A, u, v) rekurzív eljárás minden hívásában meghívódik a merge(A, u, m, v) eljárás, így az adott szinten, ez utóbbi hívások is lefedik az A vektort. Az $l \leq mT_{merge}(l) \leq MT_{merge}(l) \leq 2l$ becslés, valamint a szorzás és összeadás diszributivitásának szabályai miatt tehát a rekurzió alsó két szintjétől eltekintve minden szinten, a merge$(\ldots)$ hívások összes műveletigénye $\geq n$. A rekurzió legalsó szintjétől eltekintve pedig minden szinten, a merge$(\ldots)$ hívások összes műveletigénye $\leq 2n$. (A legalsó szinten ez nulla.)

Mivel a rekurzió h mélységére a 9.14. feladat szerint $h \geq \lfloor \lg n \rfloor$ (ahol a rekurziós fa gyökere 0 mélségben van), a merge(A, u, m, v) eljárás összes végrehajtására együtt

$$mT(n) \geq n(\lfloor \lg n \rfloor - 1) \geq n(\lg n - 2)$$

Mivel a rekurzió h mélységére a 9.14. feladat szerint $h \leq \lfloor \lg n \rfloor + 1$, a merge$(A, u, m, v)$ eljárás összes végrehajtására együtt

$$MT(n) \leq 2n(\lfloor \lg n \rfloor + 1) \leq 2n(\lg n + 1)$$

Összegezve, a merge(A, u, m, v) eljárás összes végrehajtására együtt

$$n \lg n - 2n \leq mT(n) \leq MT(n) \leq 2n(\lg n) + 2n$$

Hozzáadva ehhez a merge$(\ldots)$ eljárás végrehajtásai nélküli $2n$ műveletigényt, a teljes mergeSort(A) eljárásra

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

adódik. Ebből a 8.28. tétellel kapjuk a bizonyítandó összefüggést.

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

9.15. Feladat. *A fentiekhez hasonlóan lássuk be, hogy a 7.1.6. alfejezetben ismertetet, egyszerű láncolt listákra (S1L) vonatkozó mergeSort(L) eljárásra is teljesül a fenti műveletigény, ahol n a rendezendő L lista hossza.*

10. Az összehasonlító rendezések alsókorlát-elemzése

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

Proof. Clearly we have to check all the n items and only a limited number of items is checked without starting a new subprogram call or loop iteration. Let this limit be k. Then $mT(n) \geq \frac{1}{k}n \Longrightarrow mT(n) \in \Omega(n)$. □

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

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

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

In this section, we assume without loss of generality that all the input elements are distinct[21]. Given this assumption, comparisons of the form $a_i == a_j$ and $a_i \neq a_j$ are useless, so we can assume that no comparisons of this form are made[22]. We also note that the comparisons $a_i < a_j$, $a_i \leq a_j$, $a_i \geq a_j$, and $a_i > a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j. We therefore assume that all comparisons have the form $a_i \leq a_j$. [4]

We can view comparison sorts abstractly in terms of decision trees. A decision tree is a strictly binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. [4]

[21] In this way we restrict the set of possible inputs and we are going to give a lower bound for the worst case of comparison sorts. Thus, if we give a lower bound for the maximum number of key comparisons ($MC(n)$) and for maximum time complexity ($MT(n)$) on this restricted set of input sequences, it is also a lower bound for them on the whole set of input sequences, because $MC(n)$ and $MT(n)$ are surely $\geq$ on a larger set than on a smaller set.

[22] Anyway, if such comparisons are made, neither $MC(n)$ nor $MT(n)$ are decreased.

Let us suppose that $\langle a_1, a_2, \ldots a_n \rangle$ is the input sequence to be sorted. In a decision tree, each internal node is labeled by $a_i \leq a_j$ for some elements of the input. We also annotate each leaf by a permutation of $\langle a_1, a_2, \ldots a_n \rangle$. The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each internal node indicates a comparison $a_i \leq a_j$. The left subtree then dictates subsequent comparisons once we know that $a_i \leq a_j$, and the right subtree dictates subsequent comparisons knowing that $a_i > a_j$. When we come to a leaf, the sorting algorithm has established the appropriate ordering of $\langle a_1, a_2, \ldots a_n \rangle$. Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree for a comparison sort to be correct. Thus, we shall consider only decision trees in which each permutation appears as a leaf of the tree. [4]

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

10.3. Tétel. *Bármely összehasonító rendezés végrehejásához a legrosszabb esetben* $MC(n) \in \Omega(n \lg n)$ *kulcsösszehasonlítás szükséges.*

Proof. From the preceding discussion, it suffices to determine the height $h = MC(n)$ of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height h has no more than 2^h leaves, we have $n! \leq l \leq 2^h$. [4]

Consequently

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

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

□

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

Proof. Only a limited number of key comparisons are performed without starting a new subprogram call or loop iteration. Let this limit be k. Then

$MT(n) \geq \frac{1}{k} MC(n) \Longrightarrow MT(n) \in \Omega(MC(n))$. Together with theorem 10.3 and transitivity we receive this theorem. $\square$

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

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

11. Rendezés lineáris időben

Alapvetően nem kulcsösszehasonlítással rendezünk ([4] 8.2), így ezekre az algoritmusokra nem vonatkozik az öszehasonlító rendezések alaptétele ([4] 8.1 tétel).

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

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

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

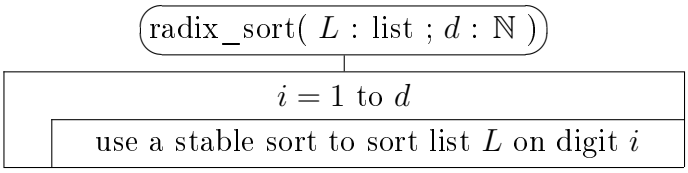
11.1. Radix rendezés (listákra)

Meglepő helyen találhatunk megoldást a lineáris időben való rendezés problémájára. Számítógép múzeumokban még láthatók lyukkártya rendező gépek. Minden kártyának 80 oszlopa és 12 sora van. Egy gép mindegyik oszlopba ezen 12 hely bármelyikére lyukat tudott ütni. A lyukkártya rendező gépet mechanikusan programozták, hogy adott i ($i \in 1..80$) érték szerint, sorban mindegyik lyukkátyát az i-edik oszlopában található lyuk helye alapján 12 sor-szerűen működő kártyatároló valamelyikébe helyezze el.[23] Ezután egy operátor összegyűjtötte a kártyákat, úgy, hogy az i-edik oszlopukban az első helyen kilyukasztottak kerültek felülre, a második helyen kilyukasztottak alájuk, és így tovább, a 12. helyen kilyukasztottak legalulra. Ha a kártyákra decimális számokat írunk, mindegyik oszlopban csak 10 helyet használunk. Egy d-jegyű szám d oszlopot igényel.

A fenti példától elvonatkoztatva, a Radix rendezés általában felteszi, hogy a kulcsok r alapú számrendszerben felírt, d számjegyű, előjel nélküli egész számok. (Szükség esetén vezető nullákat írunk a számok elé, hogy pontosan d

[23] Hasonló alapelven működő *card sorting machine*-ek ma is forgalomban vannak, persze a modern gépek már nem lyukkártyákat válogatnak szét.

számjegyük legyen.) A számok számjegyeit jobbról balra sorszámozzuk. Az első számjegy a legkevésbé szignifikáns, a jobb szélső számjegy, és így tovább, a d-edik a legszignifikánsabb, a bal szélső számjegy. A rendezésnek d menete van. Az első menetben az első (a jobb szélső, a legkevésbé szignifikáns) számjegy szerint rendezünk stabil rendezéssel, az i-edik menetben a jobbról i-edik számjegy szerint, és az utolsó menetben a jobbról d-edik (a bal szélső, a legszignifikánsabb) számjegy szerint, mindig stabil rendezéssel, az alábbi séma szerint (ahol L absztrakt lista, vagy más néven véges sorozat).



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

Annak érdekében, hogy a Radix rendezés helyesen működjön, szükséges, hogy a számjegyek szerinti rendezések stabilak legyenek. A lyukkártya rendező gép által végrehajtott szétválogatás stabil, de az operátornak is oda kell figyelni, hogy ne keverje össze a kártyákat, ahogy összegyűjti őket a sor-szerűen működő tárolókból, bár ugyanabban a tárolóban minden kártyának a szortírozást vezérlő oszlopában ugyanaz a „számjegy” található. (Ezután ezeket a sor-szerűen működő tárolókat egyszerűen csak polcoknak fogjuk nevezni.)

A lyukkártyáktól ismert séma szerint, a radix rendezés által felhasznált stabil rendezés, r alapú számrendszer esetén, a számokat – az i-edik számjegyük értéke szerint – sorban szétválogatja a $Z_0, Z_1, Z_2, \ldots Z_{r-1}$ – kezdetben üres

– polcokra (bins), ügyelve arra, hogy az egyes polcokon megmaradjon a számoknak a szétválogatás előtti sorrendje. Ezután a polcok tartalmát – a polcok egymás közötti és a számoknak a polcokon belüli sorrendjét is megtartva – összefűzi L-be.[24] Mivel r darab polcot kell üresre inicializálni, n elemet szétválogatni, majd az r db polcot, valójában listát összefűzni, ez megoldható $\Theta(n+r)$ műveletigénnyel. Mivel a radix rendezés ezt a – nyilvánvalóan stabil – rendezést d-szer hívja meg, a teljes műveletigénye $\Theta(d*(n+r))$. Ha d konstans, és $r \in O(n)$, akkor ebből

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

A 11. ábrán látható példában $r = 4$ és $d = 3$, azaz a kulcsok 4-es számrendszerbeli, 3 jegyű számok, és a Z_0, Z_1, Z_2, Z_3 polcokat használjuk.

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

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

Néha használjuk a radix rendezést összetett kulcsú rekordok rendezésére. Ilyen összetett kulcs például a dátum, ami három komponenst (*év, hó, nap*)

[24] Sok ember számára az lenne természetes, hogy a számokat először a balszélső és utoljára a jobbszélső számjegyük szerint rendezze. A radix rendezésben azonban a későbbi menetek fontosabbak az eredmény szempontjából, mint a korábbiak: Az i-edik menetben az első $i-1$ menet eredménye alárendelődik az i-edik menetnek. Így a végén a számok elsősorban a jobbszélső számjegyük szerint lennének rendezve, és általában nem ezt akarjuk.

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

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

1. menet (a számok jobbról 1., azaz jobbszélső számjegyei szerint):
$Z_0 = \langle \rangle$
$Z_1 = \langle 111, 211 \rangle$
$Z_2 = \langle 232, 002, 012 \rangle$
$Z_3 = \langle 103, 013 \rangle$
$L = \langle 111, 211, 232, 002, 012, 103, 013 \rangle$

2. menet (a számok jobbról 2., azaz középső számjegyei szerint):
$Z_0 = \langle 002, 103 \rangle$
$Z_1 = \langle 111, 211, 012, 013 \rangle$
$Z_2 = \langle \rangle$
$Z_3 = \langle 232 \rangle$
$L = \langle 002, 103, 111, 211, 012, 013, 232 \rangle$

3. menet (a számok jobbról 3., azaz balszélső számjegyei szerint):
$Z_0 = \langle 002, 012, 013 \rangle$
$Z_1 = \langle 103, 111 \rangle$
$Z_2 = \langle 211, 232 \rangle$
$Z_3 = \langle \rangle$
$L = \langle 002, 012, 013, 103, 111, 211, 232 \rangle$

11. ábra. A szétválogató-összefűző Radix rendezés bemutatása

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

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

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

Megoldás:

radix_sort(L : E2* ; d, r : $\mathbb{N}$)

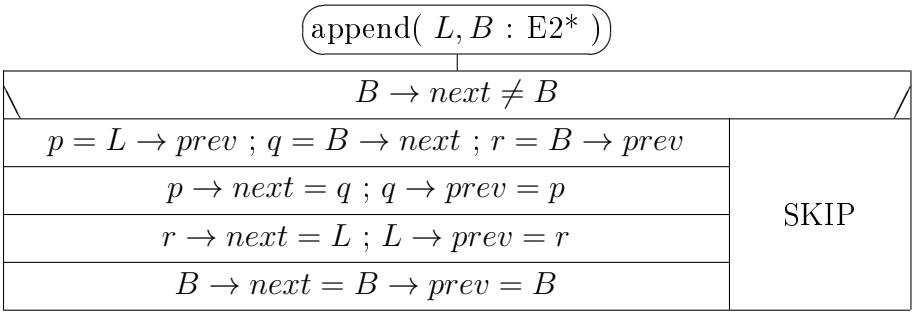
- $BinHeadZ$: E2[r] // the headers of the lists representing the bins
- Bz : E2*[r] // pointers to the headers
- $i = 0$ to $r - 1$
  - $Bz[i] = \&BinHeadZ[i]$ // Initialize the ith pointer.
- $i = 1$ to d
  - distribute(L, i, Bz) // Distribute L on the ith digits of keys.
  - gather(Bz, L) // Gather form the bins back into L

distribute(L : E2* ; i : $\mathbb{N}$; Bz : E2*[])

- $L \rightarrow next \neq L$
  - $p = L \rightarrow next$; out(p)
  - precede(p, Bz[digit$(i, Bz.M, p \rightarrow key)$]

gather(Bz : E2*[] ; L : E2*)

- $i = 0$ to $Bz.M - 1$
  - append($L, Bz[i]$) // add to the end of L the elements form $Bz[i]$



Clearly, $T_{\text{append}} \in \Theta(1)$, so $T_{\text{gather}} \in \Theta(r)$ where $r = Bz.M$.
And $T_{\text{distribute}}(n) \in \Theta(n)$ where $n = |L|$.
Thus $T_{\text{radix\_sort}}(n, d, r) \in \Theta(r + d(n + r))$.
Consequently, if d is constant and $r \in O(n)$, then $T_{\text{radix\_sort}}(n) \in \Theta(n)$.

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

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

11.2. Leszámláló rendezés (counting sort)

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

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

Az alábbiakban a leszámláló rendezést egy kicsit általánosabban tárgyaljuk, mint ahogy azt a számjegypozíciós rendezéshez szükséges. Ha a radix sort-hoz használjuk, feltehető, hogy a counting sort φ függvénye a megfelelő számjegyet választja ki.

A rendezési feladat: Adott az A:$\mathcal{T}[n]$ tömb, $r \in O(n)$ pozitív egész, $\varphi : \mathcal{T} \to 0..(r-1)$ kulcsfüggvény. Rendezzük az A tömböt lineáris időben stabil rendezéssel úgy, hogy az eredmény a B:$\mathcal{T}[n]$ tömbben keletkezzék!

counting_sort($A, B : \mathcal{T}[]$; $r : \mathbb{N}$; $\varphi : \mathcal{T} \to 0..(r-1)$)

$Z : \mathbb{N}[r]$ // counter array
$k = 0$ to $r-1$
$\quad Z[k] = 0$ // init the counter array
$i = 1$ to $A.M$
$\quad Z[\varphi(A[i])]{+}{+}$ // count the items with the given key
$k = 1$ to $r-1$
$\quad Z[k]$ += $Z[k-1]$ // $Z[k]$ = the number of items with key $\leq k$
$i = A.M$ downto 1
$\quad k = \varphi(A[i])$ // k = the key of $A[i]$
$\quad B[Z[k]] = A[i]$ //Let $A[i]$ be the last of the {unprocessed items with key k}
$\quad Z[k]--$ // The next one with key k must be put before $A[i]$

A fenti struktogram első ciklusában kinullázzuk a Z számláló tömböt.

A második ciklusban minden lehetséges k kulcsra $Z[k]$-ban megszámoljuk, hogy hány db k kulcsú elem van.

A harmadik ciklusban minden lehetséges k kulcsra összeadjuk $Z[k]$-ban, hogy hány $\leq k$ kulcsú elem van. Mivel ≤ 0 kulcsú elem ugyanannyi van, mint 0 kulcsú, $Z[0]$ értéke nem változik. Nagyobb k kulcsokra viszont annyi $\leq k$ kulcsú elem van, amennyi pontosan k kulcsú + k-nál kisebb kulcsú (vagyis $\leq k-1$ kulcsú). $Z[k]$ új értékét tehát úgy kaphatjuk meg, ha $Z[k]$ régi értékéhez hozzáadjuk $Z[k-1]$ új értékét.

A negyedik ciklusban a bemeneti tömbön visszafelé haladva minden elemet az eredmény vektorba a helyére teszünk, vagyis tetszőleges k kulcsra először az utolsó k kulcsú elemet dolgozzuk fel, és betesszük az utolsó helyre, ahova k kulcsú elem kerülhet, pontosan az eredmény tömb $Z[k]$-adik elemébe. $Z[k]$ mutatja meg ugyanis, hogy hány darab legfeljebb k kulcsú elem van a bemeneten. Ezután a $Z[k]$-t eggyel csökkentjük, és így a fordított bejárás szerint következő k kulcsú elem közvetlenül a mostani elé fog kerülni stb. Emiatt tehát tetszőleges k kulcsra a k kulcsú elemek az eredmény tömbben is az eredeti sorrendjükben maradnak, és a kisebb kulcsú elemek meg is előzik a nagyobb kulcsúakat, azaz *stabil rendezést* kapunk.

A műveletigény nyilván $\Theta(n+r)$. Feltételezve, hogy $r \in O(n)$, $\Theta(n+r) = \Theta(n)$, azaz $T(n) \in \Theta(n)$.

A leszámláló rendezés szemléltetése: Feltesszük, hogy kétjegyű, négyes számrendszerbeli számokat kell rendezni a jobboldali számjegyük, mint kulcs szerint, azaz a φ kulcsfüggvény a jobboldali számjegyet választja ki.

A bemenet:

	1	2	3	4	5	6
$A:$	02	32	30	13	10	12

A Z:$\mathbb{N}[4]$ számláló tömb alakulása [ahol az első oszlopban – a struktogram első ciklusának megfelelően – kinullázzuk a Z számláló tömböt; a következő hat oszlopban – a struktogram második ciklusának megfelelően – minden lehetséges k kulcsra (itt számjegyre) megszámoljuk, hogy hány k kulcsú elem van; a $\sum$ jelzésű oszlopban – a struktogram harmadik ciklusának megfelelően – minden lehetséges k kulcsra (itt számjegyre) összeadjuk, hogy hány $\leq k$ kulcsú elem van; az utolsó hat oszlopban pedig – a struktogram negyedik ciklusának megfelelően – a bemeneti tömbön visszafelé haladva minden elemet a helyére teszünk, ahogy azt fentebb részleteztük]:

	Z	02	32	30	13	10	12	$\sum$	12	10	13	30	32	02
0	0			1		2		2		1		0		
1	0							2						
2	0	1	2				3	5	4				3	2
3	0				1			6			5			

A kimenet :

	1	2	3	4	5	6
$B:$	30	10	02	32	12	13

Most pedig feltesszük, hogy az előző leszámláló rendezés eredményeként adódott kétjegyű, négyes számrendszerbeli számok sorozatát kell rendezni a baloldali számjegyük, mint kulcs szerint, azaz a φ kulcsfüggvény a baloldali számjegyet választja ki.

A bemenet:

	1	2	3	4	5	6
$B:$	30	10	02	32	12	13

A Z:$\mathbb{N}[4]$ számláló tömb alakulása:

	Z	30	10	02	32	12	13	$\sum$	13	12	32	02	10	30
0	0			1				1				0		
1	0		1			2	3	4	3	2			1	
2	0							4						
3	0	1			2			6			5			4

A kimenet :

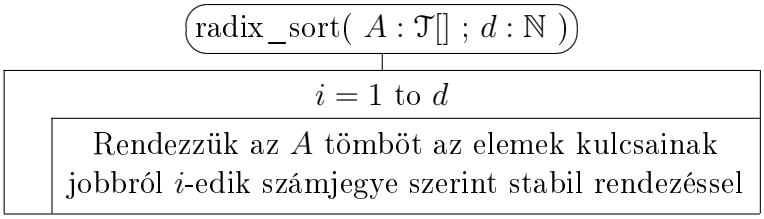
	1	2	3	4	5	6
$A:$	02	10	12	13	30	32

Mivel az első leszámláló rendezés a bemenetet a jobboldali számjegyek szerint rendezte, és a második leszámláló rendezés az első eredményét rendezte tovább a baloldali számjegyek szerint stabil rendezéssel, a végeredményben az azonos bal-számjegyű számok a jobb-számjegyük szerint rendezve maradtak, és így a végeredményben a számok már mindkét számjegyük szerint rendezettek.

Ezért tehát a fent szemléltetett két *counting sort* egy *radix rendezés* 1. és 2. menetét adja, és mivel a számainknak most csak két számjegyük van, egy teljes radix rendezést hajtottunk végre.

11.3. Radix rendezés (Radix-Sort) tömbökre ([4] 8.3)

A rendezendő A tömb kulcsai r alapú számrendszerben felírt, d-jegyű nemnegatív egész számok. A jobbról első számjegy helyiértéke a legkisebb, míg a d-ediké a legmagasabb.[25]



Ha a stabil rendezés a leszámláló rendezés, akkor a műveletigény $\Theta(d(n + r))(n = A.M)$, mivel a teljes rendezés d leszámláló rendezésből áll. Feltételezve, hogy d konstans és $r \in O(n)$, $\Theta(d(n + r)) = \Theta(n)$, azaz $T(n) \in \Theta(n)$.

Ha pl. a kulcsok négy bájtos nemnegatív egészek, választhatjuk számjegyeknek a számok bájtjait, így $d = 4$ és $r = 256$, mindkettő n-től független konstans, tehát a feltételek teljesülnek, és a rendezés lineáris időben lefut. Az i-edik számjegy, azaz bájt kinyerése egyszerű és hatékony. Ha *key* a kulcs, akkor pl. C++-ban

$$(key >> (8 * (i - 1)))\&255$$

az i-edik számjegye[26]. Ld. még [4] 8.3-ban, hogy n rendezendő adat, b bites természetes szám kulcsok és m bites „számjegyek” ($r = 2^m$) esetén, a radix rendezésben, b és n függvényében, hogyan érdemes m-et megválasztani!

[25] Általánosítva, a kulcs felbontható d kulcs direkt szozatára, ahol a jobbról első legkevésbé szignifikáns, míg a d-edik a leglényegesebb. Pl. ha a kulcs dátum, akkor először a napok, majd a hónapok és végül az évek szerint alkalmazunk stabil rendezést. Ez azért jó, mert – a stabilitás miatt – amikor a hónapok szerint rendezünk, az azonos hónapba eső elemek a napok szerint rendezettek maradnak, és amikor az évek szerint rendezünk, az azonos évbe eső elemek hónapok és ezen belül napok szerint szintén sorban maradnak.

[26] A fenti C++-os képlet tovább egyszerűsödik, ha a programban i helyett az eltolás mértékét tartjuk nyilván (ez persze egyenlő $8 * (i - 1)$-gyel).

11.4. Feladat. *Részletezzük a* radix-sort *fenti, absztrakt programját úgy, hogy a stabil rendezéshez leszámláló rendezést használunk, és a „számjegyek" a teljes b bites kulcs m bites szakaszai! Vegyük figyelembe, hogy ettől a counting sort paraméterezése is változik, és hogy érdemes felváltva hol az eredeti $A[1..n]$ tömbből a $B[1..n]$ segédtömbbe, hol a $B[1..n]$-ből az $A[1..n]$-be végezni a leszámláló rendezést!*

11.4. Egyszerű edényrendezés (bucket sort)

Feltesszük, hogy a rendezendő elemek kulcsai a $[0; 1)$ valós intervallum elemei. (Ha a kulcs egész vagy valós szám típusú, vagy azzá konvertálható, továbbá tudunk a kulcsok számértékeire alsó és felső határt mondani, akkor a kulcsok számértékei nyilván normálhatók a $[0; 1)$ valós intervallumra. Ha ugyanis $a < b$ valós számok, és a k kulcsra $k \in [a; b)$, akkor $\frac{k-a}{b-a} \in [0; 1)$.)

Az alábbi algoritmus akkor lesz hatékony, ha az input kulcsai a $[0; 1)$ valós intervallumon egyenletesen oszlanak el. (Az L [absztrakt] listát (más néven véges sorozatot) – mint mindig – most is monoton növekvően rendezzük. Az edények [buckets] rendezésére valamilyen korábbról ismert rendezést használunk.)

```
bucket_sort( L : list )
    n = the length of L
    Z : list[n] // Create the buckets Z[0..(n−1)]
    j = 0 to (n−1)
        Let Z[j] be empty list
    L ≠ ⊘
        Remove the first element of list L
        Insert this element according to its key k into list Z[⌊n ∗ k⌋]
    j = 0 to (n−1)
        Sort list Z[j] nondecreasingly
    Append lists Z[0], Z[1], ..., Z[n − 1] in order into list L
```

Nyilván $mT(n) \in \Theta(n)$, és a fenti egyenletes eloszlást feltételezve $AT(n) \in \Theta(n)$ is teljesül. $MT(n)$ pedig attól függ, hogy a $Z[j]$ listákat milyen módszerrel rendezzük. Pl. egyszerű beszúró rendezést használva $MT(n) \in \Theta(n^2)$, összefésülő rendezéssel viszont $MT(n) \in \Theta(n \lg n)$.

11.5. Feladat. *Részletezze az elemi utasítások szintjéig a fenti kódot, feltéve, hogy a listák egyszerű láncolt listák (S1L)! Vegyük észre, hogy az edénybe (bucket) való beszúrásnál – ha nem törekszünk a rendezés stabilitására – az edényt reprezentáló S1L elejére érdemes a listaelemet beszúrni. (Az edények rendezését nem kell részletezni.)*

11.6. Feladat. *Tegye a 11.5. feladatot megoldó rendezést stabillá,* $MT(n) \in \Theta(n \lg n)$*;* $mT(n), AT(n) \in \Theta(n)$ *műveletigénnyel!*

12. Hasító táblák

A mindennapi programozási gyakorlatban sokszor van szükségünk ún. szótárakra, amelyek műveletei: (1) adat beszúrása a szótárba, (2) kulcs alapján a szótárban a hozzá tartozó adat megkeresése, (3) a szótárból adott kulcsú, vagy egy korábbi keresés által lokalizált adat eltávolítása.

Az AVL fák, B+ fák (ld. a következő félévben) és egyéb kiegyensúlyozott keresőfák mellett a szótárakat gyakran hasító táblákkal valósítják meg, feltéve, hogy a műveleteknek nem a maximális, hanem az átlagos futási idejét szeretnék minimalizálni. (A kiegyensúlyozott keresőfák esetében ti. elsősorban a maximális műveletigényt optimalizáljuk: beszúrás, keresés és törlés esetén is elvárt az $O(\lg n)$ maximális műveletigény.) Hasító táblát használva ugyanis a fenti műveletekre elérhető az ideális, $\Theta(1)$ átlagos futási idő azon az áron, hogy a maximális műveletigény általában $\Theta(n)$.

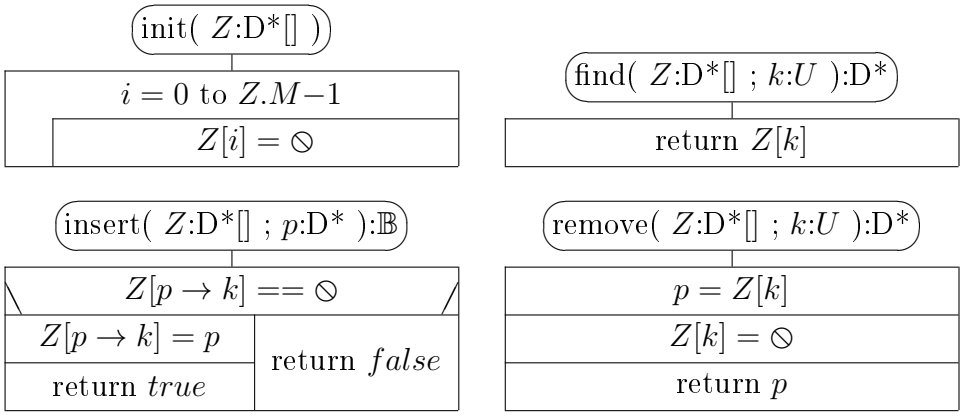
Jelölések:
m : a hasító tábla mérete
$Z[0..(m-1)]$: a hasító tábla
$Z[0], Z[1], \ldots, Z[m-1]$: a hasító tábla rései (slot-jai)
$\oslash$: üres rés a hasító táblában (direkt címzésnél és a kulcsütközések láncolással való feloldása esetén)
E : üres rés kulcsa a hasító táblában (nyílt címzésnél)
D : törölt rés kulcsa a hasító táblában (nyílt címzésnél)
n : a hasító táblában tárolt adatok száma
$\alpha = n/m$: a hasító tábla kitöltöttségi aránya (load factor)
U : a kulcsok univerzuma; $k, k', k_i \in U$
$h : U \to 0..(m-1)$: hasító függvény

Feltesszük, hogy a hasító tábla nem tartalmazhat két vagy több azonos kulcsú elemet, és hogy $h(k)$, $\Theta(1)$ időben számolható.

12.1. Direkt címzés (direct-address tables)

Feltesszük, hogy $U = 0..(m-1)$, ahol $m \geq n$, de m nem túl nagy.
A $Z : D * [m]$ hasító tábla rései pointerek, amik D típusú adatrekordokra mutatnak. A rekordoknak van egy $k : U$ kulcsmezőjük és járulékos mezőik. A hasító táblát $\oslash$ pointerekkel inicializáljuk.

D
$+\ k : U$ // k is the key
$+\ \ldots$ // satellite data



Nyilván $T_{\text{init}}(m) \in \Theta(m)$, ahol $m = Z.M$. A másik három műveletre pedig $T \in \Theta(1)$.

12.2. Hasító táblák (hash tables)

Hasító függvény (hash function): Ha $|U| >> n$, a direkt címzés nem alkalmazható, vagy nem gazdaságos, ezért $h : U \to 0..(m-1)$ hasító függvényt alkalmazunk, ahol tipikusan $|U| >> m$ (a kulcsok U „univerzumának" elemszáma sokkal nagyobb, mint a hasító tábla m mérete). A k kulcsú adatot a $Z[0..(m-1)]$ hasító tábla $Z[h(k)]$ résében tároljuk (próbáljuk tárolni).

Feltesszük, hogy a hasító táblában minden rekord kulcsa egyedi, azaz két tetszőleges rekord kulcsa különböző.

A $h : U \to 0..(m-1)$ függvény *egyszerű egyenletes hasítás*, ha a kulcsokat a rések között egyenletesen szórja szét, azaz hozzávetőleg ugyanannyi kulcsot képez le az m rés mindegyikére. Tetszőleges hasító függvénnyel kapcsolatos elvárás, hogy egyszerű egyenletes hasítás legyen.

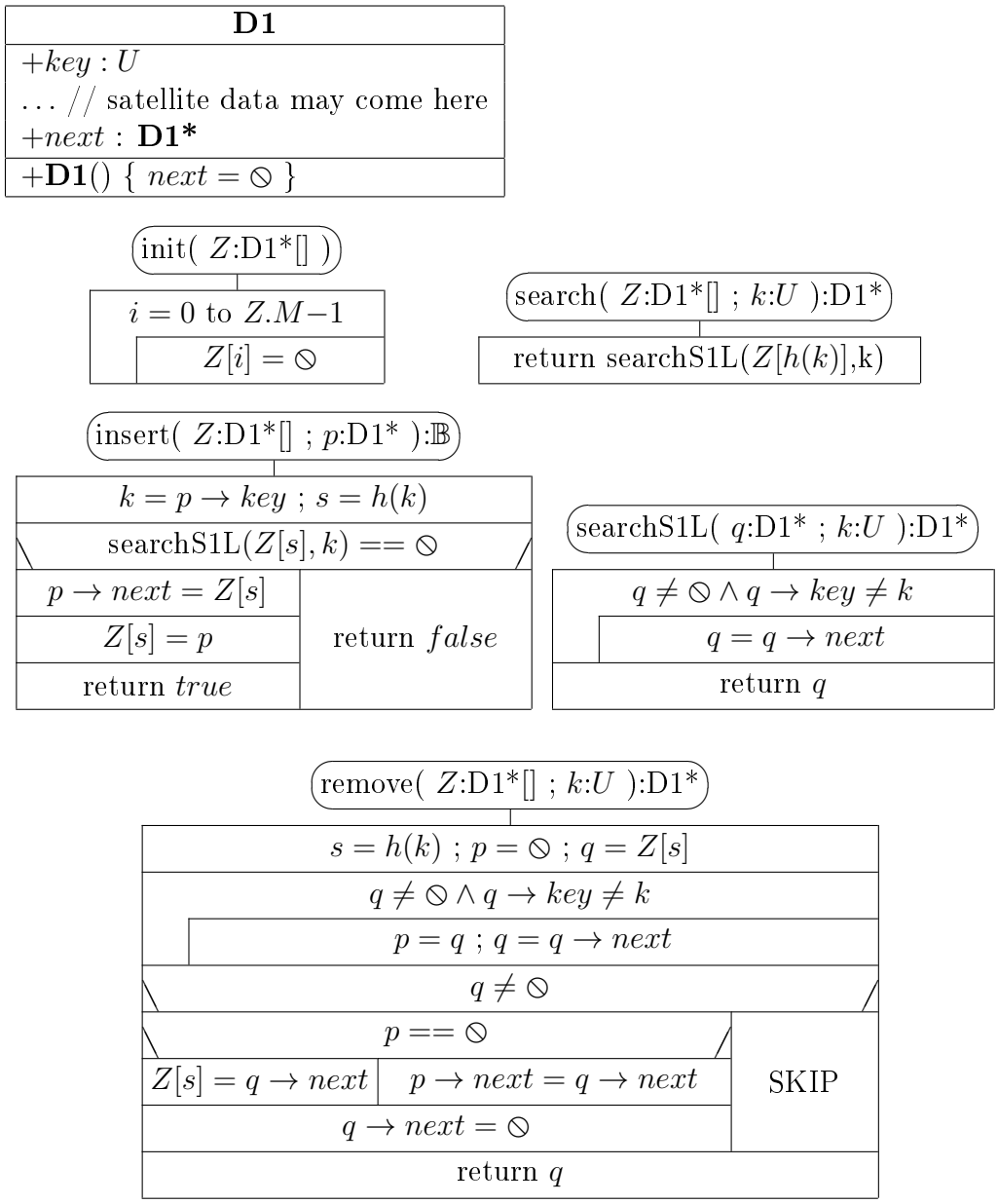
Kulcsütközések (collisions): Ha két adat k_1, k_2 kulcsára $h(k_1) = h(k_2)$, kulcsütközésről beszélünk. Mivel $|U| >> m$, kulcsütközés szinte biztosan előfordul, ezért kezelni kell.

Ha például a kulcsok egész számok, és $h(k) = k \bmod m$, akkor pontosan azok a kulcsok képeződnek le az s-edik résre, amelyekre $r = k \bmod m$.

12.3. Kulcsütközések feloldása láncolással (collision resolution by chaining)

Feltesszük, hogy a hasító tábla rései egyszerű láncolt listákat azonosítanak, azaz Z:D1*$[m]$, ahol a listaelemekben a szokásos *key* és a *next* mezőkön kívül általában járulékos mezők (satellite data) is vannak. Ha a hasító függvény

két vagy több elem kulcsait a hasító táblának ugyanarra a résére képzi le, akkor ezeket az elemeket az ehhez a réshez tartozó listában tároljuk.



Nyilván $T_{\text{init}}(m) \in \Theta(m)$, ahol $m = Z.M$. A másik három műveletre pedig $mT \in \Theta(1)$, $MT(n) \in \Theta(n)$, $AT(n,m) \in \Theta(1 + \frac{n}{m})$.

$AT(n,m) \in \Theta(1 + \frac{n}{m})$ feltétele, hogy a $h : U \to 0..(m-1)$ függvény *egyszerű egyenletes hasítás* legyen, és még az indokolja, hogy a résekhez tartozó listák átlagos hossza $= \frac{n}{m} = \alpha$.

Általában feltesszük még, hogy $\frac{n}{m} \in O(1)$. Ebben az esetben nyilván $AT(n,m) \in \Theta(1)$ is teljesül.

12.4. Jó hasító függvények (good hash functions)

Osztó módszer (division method): Ha a kulcsok egész számok, gyakran választják a

$$h(k) = k \bmod m$$

hasító függvényt, ami gyorsan és egyszerűen számolható, és ha m olyan prím, amely nincs közel a kettő hatványokhoz, általában egyenletesen szórja szét a kulcsokat a $0..(m-1)$ intervallumon.

Ha pl. a kulcsütközést láncolással szeretnénk feloldani, és kb. 2000 rekordot szeretnénk tárolni $\alpha \approx 3$ kitöltöttségi aránnyal, akkor a 701 jó választás: A 701 ui. olyan prímszm, ami közel esik a 2000/3-hoz, de a szomszádos kettő hatványoktól, az 512-től és az 1024-től is elég távol van.

Kulcsok a $[0\,;1)$ intervallumon: Ha egyenletesen oszlanak el, a

$$h(k) = \lfloor k * m \rfloor$$

függvény is kielégíti az egyszerű, egyenletes hasítás feltételét.

Szorzó módszer (multiplication method): Ha a kulcsok valós számok, tetszőleges $0 < A < 1$ konstanssal alkalmazható a

$$h(k) = \lfloor \{k * A\} * m \rfloor$$

hasító függvény. ($\{x\}$ az x törtrésze.) Nem minden lehetséges konstanssal szór egyformán jól. Knuth az $A = \frac{\sqrt{5}-1}{2} \approx 0,618$ választást javasolja, mint ami a kulcsokat valószínűleg szépen egyenletesen fogja elosztani a rések között. Az osztó módszerrel szemben előnye, hogy nem érzékeny a hasító tábla méretére.

Előjel nélküli egész kulcsok esetén, ha a táblaméretet kettő hatványnak választjuk, kikerülhető a viszonylag lassú, valós aritmetika. Tegyük fel, hogy a kulcsok w biten ábrázolt természetes számok. Ekkor $0 \leq k \leq 2^w - 1$. Ábrázoljuk szintén w biten az $s = \lfloor A * 2^w \rfloor$ konstanst. Tegyük fel, hogy a táblaméret $m = 2^p$. Legyen $q = 2^w - 1$ és $r = w - p$. Ekkor a fenti szorzó módszert jól közelíthetjük az alábbi hasító függvény definícióval, feltéve, hogy dupla szavas, előjel nélküli egész aritmetikát használunk. (& a bitenkénti „és” művelet, és $>>$ jelöli a bitléptetést jobbra.)

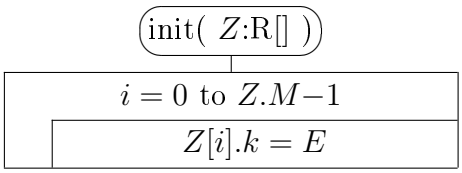
$$h(k) = ((s * k)\&q) >> r$$

Mindhárom módszer feltételezi, hogy a kulcsok számok. Ha pl. sztringek, a karaktereket tekinthetjük megfelelő számrendszerbeli előjel nélküli egész számok számjegyeinek, és így a sztringeket könnyen értelmezhetjük úgy, mint nagy, természetes számokat.

12.5. Nyílt címzés (open addressing)

Feltesszük, hogy az adatrekordok közvetlenül a résekben vannak; a $Z : R[m]$ hasító tábla R típusú rekordjainak van egy $k : U \cup \{E, D\}$ kulcsmezőjük és járulékos mezőik, ahol E és D extremális konstansok ($E, D \notin U$), sorban az üres (Empty) és a törölt (Deleted) rések jelölésére.

R
$+\ k : U \cup \{E, D\}$ // k is a key or it is Empty or Deleted
$+\ \ldots$ // satellite data



Jelölések a nyílt címzéshez:
$h : U \times 0..(m-1) \to 0..(m-1)$: hasító próba
$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$: potenciális próbasorozat

Feltesszük, hogy a hasító táblában nincsenek duplikált kulcsok[27].

Az üres és a törölt réseket együtt *szabad* réseknek nevezzük. (A többi rés *foglalt*.) Egyetlen hasító függvény helyett most m darab hasító függvényünk van:

$$h(\cdot, i) : U \to 0..(m-1) \qquad (i \in 0..(m-1))$$

12.5.1. Nyílt címzés: beszúrás és keresés, ha nincs törlés

Ha a hasító táblából nem akarunk törölni (ahogy ez sok alkalmazásban így is van), a beszúrás is egyszerűbb.

A k kulcsú r adat **beszúrás**ánál például először a $h(k,0)$ réssel próbálkozunk. Ha ez foglalt és a kulcsa nem k, folytatjuk a $h(k,1)$-gyel stb., mígnem

[27] Tetszőleges adattárolóban egy kulcs duplikált, ha az adattárolóban (itt a hasító táblában) legalább kétszer fordul elő.

üres rést találunk, vagy k kulcsú foglalt rést találunk, vagy kimerítjük az összes lehetséges próbát, azaz a $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ *potenciális próbasorozatot*. Ha üres rést találunk, ebbe tesszük az adatot, különben sikertelen a beszúrás.

A $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ sorozatot a azért nevezzük *potenciális próbasorozatnak*, mert a beszúrás, keresés vagy törlés során ennek ténylegesen csak egy prefixét állítjuk elő. A potenciális próbasorozatnak azt a prefixét, amit egy beszúrás, keresés (vagy törlés) esetén ténylegesen előállítunk, *aktuális próbasorozatnak* nevezzük. A potenciális próbasorozattal szemben megköveteljük, hogy a $\langle 0, 1, \ldots, (m-1)\rangle$ egy permutációja legyen, azaz, hogy az egész hasító táblát lefedje (és így ne hivatkozzon kétszer vagy többször ugyanarra a résre). Ha a beszúrás a $h(k, i-1)$ próbánál áll meg, akkor (és csak akkor) a beszúrás (azaz az aktuális próbasorozat) hossza i.

A k kulcsú adat **keresés**énél is a fenti potenciális próbasorozatot követjük, és akkor állunk meg, ha megtaláltuk a keresett kulcsú foglalt rést (sikeres keresés), illetve ha üres rést találunk vagy kimerítjük a potenciális próbasorozatot (sikertelen keresés). Ha a keresés a $h(k, i-1)$ próbánál áll meg, akkor (és csak akkor) a keresés (azaz az aktuális próbasorozat) hossza i.

Ideális esetben egy tetszőleges potenciális próbasorozat a $\langle 0, 1, \ldots, (m-1)\rangle$ sorozatnak mind az $m!$ permutációját azonos valószínűséggel állítja elő. Ilyenkor *egyenletes hasításról* beszélünk.

Amennyiben a táblában nincsenek törölt rések – egyenletes hasítást és a hasító tábla $0 < \alpha < 1$ kitöltöttségét feltételezve –, egy sikertelen keresés illetve egy sikeres beszúrás várható hossza legfeljebb

$$\frac{1}{1-\alpha}$$

míg egy sikeres keresés illetve sikertelen beszúrás várható hossza legfeljebb

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

Ez azt jelenti, hogy egyenletes hasítást feltételezve, pl. 50%-os kitöltöttség mellett egy sikertelen keresés (illetve egy sikeres beszúrás) várható hossza legfeljebb 2, míg egy sikeres keresésé (illetve egy sikertelen beszúrásé) kisebb, mint 1,387; 90%-os kitöltöttség mellett pedig egy sikertelen keresés (illetve egy sikeres beszúrás) várható hossza legfeljebb 10, míg egy sikeres keresésé (illetve egy sikertelen beszúrásé) kisebb, mint 2,559 [4].

12.5.2. Nyílt címzésű hasítótábla műveletei, ha van törlés is

A **törlés** egy sikeres keresést követően a megtalált i rés kulcsának *törölt*-re (D) állításából áll. Itt az *üres*-re (E) állítás helytelen lenne, mert ha például feltesszük, hogy a k kulcsú adatot kulcsütközés miatt a $Z[h(k,1)]$ helyre tettük, majd töröltük a $h(k,0)$ helyen levő adatot (azaz a $Z[h(k,0)]$ rést üresre állítottuk), akkor egy ezt követő keresés nem találná meg a k kulcsú adatot.

A **keresés**nél tehát átlépjük a törölt réseket is, és csak akkor állunk meg,
- ha megtaláltuk a keresett kulcsú elemet (sikeres keresés),
- ha üres rést találunk vagy kimerítjük a potenciális próbasorozatot (sikertelen keresés).

A **beszúrás**nál is egy teljes keresést végzünk el, most a beszúrandó adat kulcsára, de ha közben találunk törölt rést, az első ilyent megjegyezzük.
- Ha a keresés sikeres, akkor a beszúrás sikertelen (hiszen duplikált kulcsot nem engedünk meg).
- Ha a keresés sikertelen, és találtunk közben törölt rést, akkor a beszúrandó adatot az elsőként talált törölt résbe tesszük (hogy a jövőbeli keresések hossza a lehető legkisebb legyen).
- Ha a keresés sikertelen, de nem talál törölt rést, viszont üres résen áll meg, akkor a beszúrandó adatot ebbe az üres résbe tesszük.
- Ha a keresés sikertelen, de nem talál sem törölt, sem üres rést, és így azért áll meg, mert a potenciális próbasorozatot kimeríti, akkor a beszúrás sikertelen (mert a hasítótábla tele van).

Ha elég sokáig használunk egy nyílt címzésű hasító táblát, elszaporodhatnak a törölt rések, és elfogyhatnak az üres rések, holott a tábla esetleg közel sincs tele. Ez azt jelenti, hogy pl. a sikertelen keresések az egész táblát végig fogják nézni. Ez ellen a tábla időnkénti frissítésével védekezhetünk, amikor megszüntetjük a törölt réseket. (A legegyszerűbb megoldás kimásolja az adatokat egy temporális területre, üresre inicializálja a táblát, majd a kimentett adatokat egyesével újra beszúrja.)

12.5.3. Lineáris próba

Ebben és a következő két alfejezetben áttekintünk három stratégiát $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ próba sorozat (részleges) előállítására. Mindegyiknél lesz egy elsődleges $h_1 : U \to 0..(m-1)$ hasítófüggvény, aminek az értéke egyben az első próba indexét, $h(k,0)$-t adja. Ebből kiindulva lépkedünk a réseken tovább, ha ez szükséges. Elvárás, hogy h_1 egyszerű

egyenletes hasítás legyen. Mindhárom stratégiánál feltesszük, hogy a kulcsok természetes számok (így lehet pl. $D = -1$ és $E = -2$).

A próbák közül a legegyszerűbb, a lineáris próba definíciója a következő.

$$h(k, i) = (h_1(k) + i) \bmod m \qquad (i \in 0..(m-1))$$

ahol $h_1 : U \to 0..m-1$ hasító függvény. Könnyű implementálni, de összesen csak m db különböző próba sorozat van, az egyenletes hasításhoz szükséges $m!$ db próba sorozathoz képest, hiszen ha két kulcsra $h(k_1, 0) = h(k_2, 0)$, akkor az egész próba sorozatuk megegyezik. Ráadásul a különböző próba sorozatok összekapcsolódásával foglalt rések hosszú, összefüggő sorozatai alakulhatnak ki, megnövelve a várható keresési időt. Ezt a jelenséget *elsődleges csomósodásnak* nevezzük. Minél hoszabb egy ilyen „csomó", annál valószínűbb, hogy a következő beszúráskor a hossza tovább fog növekedni. Ha pl. két szabad rés között (ciklikusan értve) i db foglalt rés van, akkor legalább $(i+2)/m$ a valószínűsége, hogy a következő sikeres beszúráskor ez a csomó még hosszabb lesz, és az is lehet, hogy közben összekapcsolódik egy másik csomóval. Ez az egyszerű módszer csak akkor használható, ha a kulcsütközés valószínűsége elenyészően kicsi.

12.5.4. Négyzetes próba

$$h(k, i) = (h_1(k) + c_1 i + c_2 i^2) \bmod m \qquad (i \in 0..m-1)$$

ahol $h_1 : U \to 0..m-1$ hasító függvény; $c_1, c_2 \in \mathbb{R}; c_2 \neq 0$. A különböző próba sorozatok nem kapcsolódnak össze, de itt is csak m db különböző próba sorozat van, az egyenletes hasításhoz szükséges $m!$ db próba sorozathoz képest, hiszen ha két kulcsra $h(k_1, 0) = h(k_2, 0)$, akkor az egész próba sorozatuk itt is megegyezik. Ezt a jelenséget *másodlagos csomósodásnak* nevezzük.

A négyzetes próba konstansainak megválasztása Annak érdekében, hogy a próba sorozat az egész táblát lefedje, a c_1, c_2 konstansokat körültekintően kell megválasztani. Ha például a tábla m mérete kettő hatvány, akkor $c_1 = c_2 = 1/2$ jó választás. Ráadásul ilyenkor

$$h(k, i) = \left(h_1(k) + \frac{i + i^2}{2}\right) \bmod m \qquad (i \in 0..m-1)$$

Ezért

$$(h(k, i+1) - h(k, i)) \bmod m = \left(\frac{(i+1) + (i+1)^2}{2} - \frac{i + i^2}{2}\right) \bmod m =$$

$$(i+1) \bmod m$$

azaz

$$h(k, i+1) = (h(k, i) + i + 1) \bmod m$$

12.1. Feladat. *Készítsük el a fenti rekurzív képlet segítségével a négyzetes próba ($c_1 = c_2 = 1/2$) esetére a beszúrás, a keresés és a törlés struktogramjait!*

12.5.5. Kettős hasítás

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m \qquad (i \in 0..(m-1))$$

ahol $h_1 : U \to 0..(m-1)$ és $h_2 : U \to 1..(m-1)$ hasító függvények. A próba sorozat pontosan akkor fedi le az egész hasító táblát, ha $h_2(k)$ és m relatív prímek. Ezt a legegyszerűbb úgy biztosítani, ha a m kettő hatvány és $h_2(k)$ minden lehetséges kulcsra páratlan szám, vagy m prímszám. Például ha m prímszám (ami lehetőleg ne essen kettő hatvány közelébe) és m' kicsit kisebb (mondjuk $m' = m - 1$ vagy $m' = m - 2$) akkor

$$h_1(k) = k \bmod m$$

$$h_2(k) = 1 + (k \bmod m')$$

egy lehetséges választás.

A kettős hasításnál minden különböző $(h_1(k), h_2(k))$ párokhoz különböző próbasorozat tartozik. Ezért itt $\Theta(m^2)$ különböző próbasorozat lehetséges. A kettős hasítás, bár próbasorozatainak száma messze van az ideális $m!$ számú próbasorozattól, úgy tűnik, hogy jól közelíti annak működését.

A kettős hasítás műveleteinek szemléltetése: Mivel $h(k, i) = (h_1(k) + ih_2(k)) \bmod m \quad (i \in 0..(m-1))$, azért $h(k, 0) = h_1(k)$ és $h(k, i+1) = (h(k, i) + d) \bmod m$, ahol $d = h_2(k)$. Az első próba helyének $(h_1(k))$ meghatározása után tehát mindig d-vel lépünk tovább, ciklikusan.

Legyen most $m = 11$ $h_1(k) = k \bmod 11$ $h_2(k) = 1 + (k \bmod 10)$.

Az alábbi táblázat „műveletek” oszlopában, minden műveletnél az első karakter a művelet kódja, azaz i=insert, s=search, d=delete. Ezután jön a beszúrandó, vagy keresett, vagy törlendő adat kulcsa (ami értelemszerűen sosem E vagy D). A táblázatban nem foglalkozunk a járulékos adatokkal, csak a kulcsokat jelöljük. Itt következik indexben a $d = h_2(k)$ érték, de csak akkor, ha ezt szükséges kiszámolni, azaz az aktuális próbasorozat hossza ≥ 2. Ezután jön maga a próbasorozat „$\langle\rangle$” zárójelek között, és végül a művelet sikerességének jelzése, ahol „+ = sikeres” és „x = sikertelen”. A próbasorozat

utolsó eleménél a megfelelő rés kulcsát is indexben odaírtuk, ezzel utalva a művelet kimenetelének okára. Ha egy beszúrás közben törölt rést találtunk, az első ilyen mellé indexben egy D betűt írtunk, utalva arra, hogy a beszúrás algoritmusa megjegyzi az első törölt rést, amit talál. (Ld. a 12.5.2. alfejezetet!) A táblázat első 11 oszlopában az üres réseket egyszerűen üresen hagytuk. Mindegyik foglalt résbe beírtuk a megfelelő kulcsot, míg a törölt réseket D betűvel jelöltük. Ebben a táblázatban a könnyebb érthetőség kedvéért minden művelet után új sorba írtuk a táblának a művelet végrehajtása utáni tartalmát, kivéve a legutolsó keresést. (A zh-kon és a vizsgákon elég lesz akkor új sort nyitni, ha nemüres rés tartalma változik.)

0	1	2	3	4	5	6	7	8	9	10	műveletek
											$i32\langle 10_E\rangle$ +
										32	$i40\langle 7_E\rangle$ +
							40			32	$i37\langle 4_E\rangle$ +
				37			40			32	$i15_6\langle 4;10;5_E\rangle$ +
				37	15		40			32	$i70_1\langle 4;5;6_E\rangle$ +
				37	15	70	40			32	$s15_6\langle 4;10;5_{15}\rangle$ +
				37	15	70	40			32	$s104_5\langle 5;10;4;9_E\rangle$ x
				37	15	70	40			32	$d15_6\langle 4;10;5_{15}\rangle$ +
				37	D	70	40			32	$s70_1\langle 4;5;6_{70}\rangle$ +
				37	D	70	40			32	$i70_1\langle 4;5_D;6_{70}\rangle$ x
				37	D	70	40			32	$d37\langle 4_{37}\rangle$ +
				D	D	70	40			32	$i104_5\langle 5_D;10;4;9_E\rangle$ +
				D	104	70	40			32	$s15_6\langle 4;10;5;0_E\rangle$ x

12.2. Feladat. A kettős hasítás programozása: *Írja meg beszúrás, a keresés és a törlés struktogramjait, ahol x a beszúrandó adat, k a keresett kulcs, illetve a törlendő adat kulcsa.*

A hasító tábla a $Z[0..(m-1)]$, azaz hagyományosan (célszerűen) nullától indexeljük, és m a mérete, ahol m értékét a függvények $Z.M$-ből nyerik ki.

Sikeres keresésnél a keresett kulcsú adat pozícióját adjuk vissza. A sikertelen keresést a „-1” visszadásával jelezzük.

Sikeres beszúrásnál a beszúrás pozícióját adjuk vissza. Sikertelen beszúrásnál két eset van. Ha nincs elég hely a táblában, azt a „$-(m+1)$” visszadásával jelezzük. Ha a j indexű résben megtaláljuk a táblában a beszúrandó adat kulcsát, azt a „$-(j+1)$” visszaadásával jelezzük.

Vegyük észre, hogy a nyílt címzésű hasító tábla üresre inicializálásának és az adott kulcsú foglalt rés törlésének struktogramja nem függ attól, hogy a beszúrást és a keresést milyen stratégiával hajtjuk végre. Sikeres törlésnél a

törölt adat pozícióját adjuk vissza. A sikertelen törlést a „-1” visszadásával jelezzük.

Megoldás: