



NEUMANN JÁNOS
INFORMATIKAI KAR

Szénási Sándor

**Algoritmusok, adatszerkezetek
objektum-orientált megközelítésben**

ÓE-NIK

Budapest, 2024.

Szerző: Prof. Dr. Szénási Sándor
egyetemi tanár
szenasi.sandor@nik.uni-obuda.hu

0.13 (nem lektorált) verzió
2024. november 15.

Ez a jegyzet L^AT_EX segítségével készült.

A mű egyéni tanulmányozás céljára szabadon letölthető. minden egyéb felhasználás csak a szerző írásos engedélyével lehetséges.

Tartalomjegyzék

Bevezetés	1
I. Programozási paradigmák	5
1. Imperatív Programozás	6
1.1. Imperatív megközelítés	6
1.1.1. Neumann elvű számítógép felépítése	6
1.1.2. Az imperatív paradigma jellemzői	7
1.2. Memória kezelése	8
1.2.1. Típusok és változók	8
1.2.2. Statikus és dinamikus memóriakezelés	10
1.2.3. Mutatók és referenciák	11
1.3. Procedurális paradigmá	14
1.3.1. Absztrakciós szintek	14
1.3.2. Gépi kód	14
1.3.3. Assembly	15
1.3.4. Procedurális programozás	15
1.3.5. Moduláris programozás	16
1.3.6. Strukturált programozás elve	17
1.4. Objektumorientált paradigmá	18
1.4.1. Alapelvek	18
1.4.2. Objektumok	19
1.4.3. Osztályok	19
1.4.4. Öröklés	22
1.4.5. Polimorfizmus	23
1.4.6. Értékelés	28
1.5. Generikusok	29
2. Deklaratív programozás	34
2.1. Deklaratív paradigmá	34
2.2. Tisztán funkcionális nyelvek	36
2.2.1. Funkcionális program alapelemei	36
2.2.2. Végrehajtás menete	39
2.2.3. Adatszerkezetek	41
2.2.4. Példa - Rendezés funkcionális szemlélettel	43
2.2.5. λ -kalkulus*	44
2.2.6. Értékelés	47
2.3. Logikai programozás	48
2.3.1. Logikai program alapelemei	48
2.3.2. Értékelés	49
2.4. Hibrid megvalósítások	51
2.4.1. Megváltoztathatatlan objektumok	51
2.4.2. Rekurzió	51
2.4.3. Függvény referenciák megjelenése	52

2.4.4. Névtelen függvények, λ -kifejezések	53
II. Alapvető adatszerkezetek	55
3. Adatszerkezetek jellemzői	56
3.1. Absztakt adatszerkezetek	56
3.1.1. Adatszerkezetek	56
3.1.2. Absztrakt adattípusok	56
3.1.3. Objektum-orientált megvalósítás	57
3.2. Alapvető absztrakt adatszerkezetek	59
3.2.1. Általános jellemzők	59
3.2.2. Lista	59
3.2.3. Verem	60
3.2.4. Sor	62
3.2.5. Prioritásos sor	63
3.2.6. Halmaz	64
3.2.7. Szótár	65
3.2.8. Gráf	66
3.2.9. Utólagos áttekintés	67
3.3. Leképezés fizikai adatszerkezetekre	68
3.3.1. Implementációs rugalmasság	68
3.3.2. Szekvenciális megvalósítás	69
3.3.3. Láncolt megvalósítások	69
4. Tömb alapú implementációk	70
4.1. Adatok reprezentációja	70
4.1.1. Alapvető műveletek	70
4.1.2. Statikus megvalósítás	70
4.1.3. Dinamikus megvalósítás	71
4.1.4. Többdimenziós tömbök	72
4.2. Verem megvalósítása tömbbel	74
4.2.1. Alapvető felépítés	74
4.2.2. Létrehozás és megszüntetés	74
4.2.3. Műveletek megvalósítása	75
4.3. Sor megvalósítása tömbbel	79
4.3.1. Alapvető felépítés	79
4.3.2. Létrehozás és megszüntetés	80
4.3.3. Műveletek megvalósítása	81
4.4. Lista megvalósítása tömbbel	84
4.4.1. Alapvető felépítés	84
4.4.2. Létrehozás és megszüntetés	84
4.4.3. Műveletek megvalósítása	85
5. Egyszeresen láncolt adatszerkezetek	91
5.1. Láncolt szerkezetek felépítése	91
5.1.1. Alapvető felépítés	91
5.1.2. Egy elem felépítése	91
5.1.3. Lánc felépítése	92
5.1.4. Értékelése	92
5.2. Lista megvalósítása láncolt szerkezzel	94
5.2.1. Alapvető felépítés	94
5.2.2. Létrehozás és megszüntetés	95
5.2.3. Új elem felvétele	95
5.2.4. Elemek bejárása és egyedi elérése	101
5.2.5. Törlés láncolt listából	104
5.3. Verem megvalósítása láncolt szerkezzel	108

5.3.1. Alapvető felépítés	108
5.3.2. Létrehozás és megszüntetés	108
5.3.3. Műveletek megvalósítása	108
5.4. Sor megvalósítása láncolt szerkezzel	112
5.4.1. Alapvető felépítés	112
5.4.2. Létrehozás és megszüntetés	112
5.4.3. Műveletek megvalósítása	113
5.5. Prioritásos sor megvalósítása láncolt szerkezzel	116
5.5.1. Alapvető felépítés	116
5.5.2. Létrehozás, megszüntetés és kiolasások	116
5.5.3. Rendezett beszúrás megvalósítása	116
5.6. További lehetőségek	124
5.6.1. Láncolt szerkezet strázsa elemekkel	124
5.6.2. Kétirányú láncolt szerkezet	127
5.6.3. Többszörösen láncolt szerkezet	128
5.6.4. Ciklikus láncolt szerkezet	129
6. Bináris keresőfa	131
6.1. Bináris keresőfa felépítése	131
6.1.1. Alapvető fogalmak	131
6.1.2. Egy elem felépítése	132
6.1.3. Fa felépítése	133
6.2. Halmaz megvalósítása bináris keresőfával	135
6.2.1. Alapvető felépítés	135
6.2.2. Bejárások	135
6.2.3. Elem meglétének vizsgálata	147
6.2.4. Új elem felvétele	150
6.2.5. Elem törlése	153
7. Szótár	165
7.1. Szótár működése	165
7.2. Megvalósítás közvetlen címzéssel	166
7.2.1. Alapvető felépítés	166
7.2.2. Műveletek megvalósítása	167
7.2.3. Értékelés	168
7.3. Megvalósítás hasító táblázattal (kulcsütközés kezelés nélkül)	170
7.3.1. Hasítás alapelve	170
7.3.2. Alapvető felépítés	171
7.3.3. Műveletek megvalósítása	172
7.3.4. Értékelés	174
7.4. Hasítófüggvény megvalósítások	175
7.4.1. Az ideális hasítófüggvény	175
7.4.2. Néhány hasítófüggvény megvalósítás	175
7.4.3. További megfontolások	177
7.5. Megvalósítás hasító táblázattal (kulcsütközések kezelésével)	179
7.5.1. Kulcsütközés kezelése	179
7.5.2. Egy elem felépítése	179
7.5.3. Túlcordulási terület	179
7.5.4. Listák használata	183
7.5.5. Nyílt címzés	184
7.5.6. Többszörös hasítás	188

III. Megoldási stratégiák	190
8. Alapvető Optimalizációs Stratégiák	191
8.1. Alapfogalmak	191
8.1.1. Optimalizáció	191
8.1.2. A 0-1 hátizsák probléma	191
8.2. Nyers erő módszere	194
8.3. Mohó algoritmusok	199
8.4. Heurisztikus módszerek	202
8.4.1. A módszer bemutatása	202
8.4.2. Hegymászó algoritmus általános alakja	202
8.4.3. A 0-1 hátizsák probléma megoldása	203
9. Oszd meg és uralkodj	207
9.1. Oszd meg és uralkodj stratégia	207
9.1.1. A módszer bemutatása	207
9.1.2. A 0-1 hátizsák probléma megoldása	208
9.1.3. A módszer értékelése	211
9.2. Feljegyzéses módszer	212
9.2.1. A módszer bemutatása	212
9.2.2. A 0-1 hátizsák probléma megoldása	212
9.2.3. A módszer értékelése	214
9.3. Táblázatos módszer	215
9.3.1. Egy bevezető példa	215
9.3.2. A módszer bemutatása	217
9.3.3. A 0-1 hátizsák probléma megoldása (optimális érték)	217
9.3.4. A 0-1 hátizsák probléma megoldása (optimális pakolás)	220
9.3.5. A módszer értékelése	230
10. Visszalépéses keresés	231
10.1. Visszalépéses keresés stratégia	231
10.1.1. A módszer bemutatása	231
10.1.2. Visszalépéses keresés általános alakja	237
10.1.3. Visszalépéses keresés példák	242
10.2. Optimalizálás visszalépéses kereséssel	251
10.2.1. A módszer bemutatása	251
10.2.2. A 0-1 hátizsák probléma megoldása	253
10.2.3. A megoldás értékelése	254
10.3. Szétválasztás és korlátozás technika	255
10.3.1. A módszer bemutatása	255
10.3.2. Visszalépéses keresés kiegészítése	255
10.3.3. Visszalépéses keresés alapú optimalizálás kiegészítése	257
10.3.4. A 0-1 hátizsák probléma megoldása	257
10.3.5. A módszer értékelése	259
IV. Speciális adatszerkezetek	261
11. Részben kiegyszűlyozott fák	262
11.1. Kupac adatszerkezet	262
11.1.1. Felépítése	262
11.1.2. Kupactulajdonság fenntartása	263
11.1.3. Kupac építés	265
11.1.4. Alap műveletek összefoglalása	267
11.2. Kupacrendezés	271
11.3. Prioritásos sor megvalósítása bináris kupaccal	277
11.3.1. Új elem felvétele	277

11.3.2. Legnagyobb elem kivétele	279
11.3.3. Prioritás növelése/csökkentése	282
11.4. Piros-fekete fa	284
11.4.1. Piros-fekete szabályok	284
11.4.2. Forgatások	284
11.4.3. A beszúrás művelete	288
12. Súlyozatlan gráf	296
12.1. Súlyozatlan gráf felépítése	296
12.1.1. Alapvető fogalmak	296
12.1.2. Egy elem felépítése	296
12.2. Tárolási módok	297
12.2.1. Dinamikus tárolás	297
12.2.2. Tárolás csúcsmátrixban	297
12.2.3. Szomszédsági listában tárolás	298
12.3. Bejárások	300
12.3.1. Szélességi bejárás	300
12.3.2. Mélységi bejárás	305
12.4. Topologikus rendezés	312
12.4.1. Topologikus rendezés fogalma	312
12.4.2. Topologikus rendezés algoritmusa	312
13. Súlyozott gráf	318
13.1. Súlyozott gráf felépítése	318
13.2. Legrövidebb út keresése (Dijkstra algoritmus)	319
13.2.1. Az alapelvek	319
13.2.2. Dijkstra algoritmus pszeudokódja	320
13.2.3. Dijkstra algoritmus kimenetének feldolgozása	327
13.3. Minimális feszítőfa keresése	330
13.3.1. Minimális feszítőfa fogalma	330
13.3.2. Prim algoritmusa	331
13.3.3. Kruskal algoritmusa	331
14. B-fa	342
14.1. Fák kiegyensúlyozottsága	342
14.2. B-fa felépítése	343
14.3. Beszúrás B-fába	346
14.3.1. Beszúrás pszeudokódja	351
14.3.2. Beszúrás segédalgoritmusok	352
14.4. Törlés B-fából	355
14.4.1. Tartalom eltávolítása	355
14.4.2. Legalább t darab tartalom biztosítása	355
14.4.3. Törlés pszeudokódja	368
14.4.4. Törlés segéd algoritmusok	371
Irodalomjegyzék	371

Bevezetés

Ez a jegyzet az Óbudai Egyetem Neumann János Informatikai Karán tanuló mérnökinformatikus hallgatók számára íródott, hogy a Programozás II. tárgy előadásai és gyakorlatai mellett egy írásos anyag is segítse a felkészülésüket. Bár a jegyzet anélkül is érthető, mégis erősen javasolt az előadások látogatása, illetve a gyakorlatokon való aktív részvétel, mivel mindenkor tartalmazhat olyan témaköröket, amelyek itt nem jelennek meg.

A jegyzet erőteljesen épít az „Algoritmusok, adatszerkezetek I.” jegyzet [10] anyagára, az ott bevezetett fogalmakat (típusok, strukturált programozás, stb.) és technikákat (programozási tételek, rendezések, stb.) itt már ismertnek tekintjük, így azok újból bemutatására már nem kerül sor. Hasonló módon feltételezzük, hogy az olvasó rendelkezik legalább egy objektum-orientált programozási nyelvben közepes szintű ismeretekkel. Bár a jegyzet írásakor fontos szempont volt, hogy csak általános, konkrét programozási nyelvektől független leírásokat tartalmazzon, a megértést mégis nagyban segíti, ha az itt megjelenő fogalmakat és pszeudokódokat hozzá tudjuk kötni konkrét programozási nyelvi elemekhez (különösen kiemelve ez utóbbiakat, amelyeket esetenként célszerű lehet implementálni is egy választott nyelven).

A tananyag határainak megszabásakor arra törekedtünk, hogy az összhangban legyen a BSc képzésben résztvevő mérnökinformatikusok elvárásaival (illetve a tőlük elvártakkal). Emiatt a konkrét programozási nyelvek szintjénél jóval absztraktabb módon, általános formában tárgyaljuk a témaköröket. Az elméleti fejezetek azonban csak olyan szinten jelennek meg, hogy azok rávilágítanak az okokra és okozatokra, a legtöbb esetben nem ragaszkodunk a precíz matematikai definíciókhoz és bizonyításokhoz. Ez a szabatosságból tett engedmény remélhetőleg megtérül abban, hogy a jegyzet gördülékenyebben olvasható, könnyebben tanulható azok számára, akik most találkoznak először az itt leírtakkal. A szövegen található szakirodalmi hivatkozások pedig útmutatást adnak azoknak, akik mélyebben is el szeretnének merülni egy-egy témakörben.

Az egyes technikák bemutatása mindenkor rövid bemutatása, ennek megfelelően a szükséges modell (fogalmak, stb.) felállítása, megoldás elvi hátterének áttekintése, egy lehetséges megoldást jelentő pszeudokód bemutatása és annak részletes elemzése, majd pedig egy-egy példán keresztül a fentiek gyakorlati vizsgálata (néhány, bonyolultabb esetben a példa megelőzheti a pszeudokódot, ezzel is elősegítve az alaplev megértését). Fontos kihangsúlyozni, hogy általában csak egy lehetséges megoldást mutatunk be a sok közül, a legtöbb esetben találhatunk ennél rövidebbet vagy akár hatékonyabbat. A jegyzet célja ugyanis nem a végletekig optimalizált, legkifinomultabb módszerek bemutatása, ehelyett megelégszünk az alapelvök áttekintésével.

A szöveg gyakran tartalmaz utalásokat az egyes pszeudokódok soraira, illetve egyes ábrákra, esetenként alábráakra. A hivatkozások túl nagy száma már az olvashatóság rovására menne, emiatt ezek csak a külön figyelmet érdemlő, illetve kulcsfontosságúak jelentő részeknél jelennek meg. Célszerű emiatt az elvi háttér megértését követően mind a pszeudokódokat, mind pedig az ábrákat külön is áttekinteni. A megjegyzéseknek köszönhetően ezek önmagukban, a szövegtől függetlenül is könnyen értelmezhetők.

Fejezetek rövid áttekintése

A jegyzet felépítése a már említett Programozás II. tárgy tematikáját követi, és bár az egyes fejezetek más-más témaköröket dolgoznak fel, azok gyakran egymásra épülnek, így az egyes fejezetekben feltételezzük az előzőek ismeretét.

A 10. fejezetben egy széleskörűen alkalmazható problémamegoldási technikát, a *visszalépéses keresést* ismerhetjük meg. Ez nagyon hatékonyan használható olyan esetekben, ha több részfeladatot kell megoldanunk, és az ezekre adható megoldások valamilyen formában kizárlják egymást. A visszalépéses

keresés alapelvén túlmenően számos példát is tartalmaz a jegyzet, azt hangsúlyozva, hogy ugyanaz az algoritmus milyen sokrétűen használható egymástól jelentősen különböző feladatok megoldása során.

A későbbi fejezetek egy nagyobb egységet alkotnak, ugyanis különböző *adatszerkezeteket* mutatnak be. Adatszerkezetek alatt a számítógépben való tárolás lehetséges módjait értjük, amelyek alapvetően az alábbiak specifikálását jelentik:

- Az adatok memóriabeli tárolása: A számítógép memoriájában az adott adatszerkezet minden formában valósítja meg az egyes elemek, illetve a köztük lévő kapcsolatok eltárolását.
- Az adatokon végezhető műveletek: Az adatok eltárolásán túl biztosítanunk kell az ezekhez való hozzáférést is. Ez a különböző tárolási módot használó adatszerkezetek esetében egészen más algoritmusokat igényelhet. Bár az egyes módszerek más-más műveleteket tesznek lehetővé, általában elmondható, hogy az alábbiakhoz hasonlókkal találkozhatunk:
 - Inicializálás: Alaphelyzetbe hozza az adatszerkezetet. Ilyenkor még nem tárol elemeket, azonban már alkalmas a későbbi műveletek végrehajtására.
 - Beszúrás: Eltárol egy új elemet az adatszerkezetben. Ez a konkrét megvalósítástól függően különböző kiegészítő paramétereket igényelhet (új elem helye, kulcsa, stb.).
 - Bejárás: Az adatszerkezet minden egyes elemének a feldolgozása. Maga a feldolgozás művelete tetszőleges lehet, valójában nem is ez érdekes számunkra, hanem az elemek egyenkénti elérése.
 - Keresés: Egy megadott elem (kulcs alapján, tulajdonság alapján, stb.) megkeresése az adatszerkezetben.
 - Törlés: A megadott kulcsú elem eltávolítása az adatszerkezetből.

A fenti leírások meglehetősen elnagyoltak, ez azonban ezen a szinten még szándékos. Célunk annak bemutatása, hogy az egymástól jelentősen különböző adatszerkezetek is alapvetően hasonló műveleteket támogatnak, még ha a háttérben egészen máshogy is hajtják azokat végre. Hasonló módon azt sem akarjuk pontosítani, hogy milyen típusú adatokat tárolunk el (számot, szöveget, objektumot, stb.), ugyanis az összes módszernél látni fogjuk, hogy az adatszerkezetek felépítése és működése alapvetően független attól, hogy mit tárolunk el bennük.

A 7. fejezetben található *hasító táblázat* valószínűleg ismerős lesz mindenki számára, ez ugyanis a hagyományos tömbök általánosításaként fogható fel. Legnagyobb előnye, hogy az elemek kulcsa egyértelműen (egy függvény segítségével) meghatározza azok memóriabeli helyét, ennek köszönhetően nagyon gyors beszúrás/keresés/törlés műveleteket tudunk készíteni. Hátránya, hogy a bejárás viszont nehezen oldható meg, illetve nagy számú kulcs esetén a remek teljesítményt se tudjuk mindig biztosítani.

A tömbökhöz képest jóval nagyobb változást jelent majd a *láncolt lista* adatszerkezet, amellyel az 5. fejezet foglalkozik. Bár a listákat is eltárolhatjuk a tömbökhöz hasonló módon, de általában dinamikus adatszerkezetként tekintünk rájuk, tehát az elemek által ténylegesen elfoglalt memóriaterület futás közben is változhat a beszúrások és törlések hatására. Ez utóbbi már önmagában is meglehetősen nagy előnyt jelenthet, és ezt még kiegészíti az a jó tulajdonsága, hogy a módosító műveletek (beszúrás, törlés) minden csak a módosítandó elem közvetlen környezetét érintik, így gyorsan elvégzhetők. Az adatszerkezet sajnos számos hátránnal is bír, megnövekszik a tárhelyigény, illetve elveszítjük a gyors véletlen elérés (és ezzel a hatékony keresés) lehetőségét. A jegyzet többféle láncolt lista megvalósítással is foglalkozik.

A láncolt listák számos problémájára megoldást nyújt a 6. fejezetben található *bináris keresőfa*. Ez is rendelkezik a dinamikus adatszerkezetek előnyeivel, sőt, ebben az esetben már a keresésre is találunk meglehetősen hatékony megoldást. A helyfoglalás itt azonban még nagyobb lesz, illetve látni fogjuk, hogy a keresés is csak bizonyos esetekben lesz kielégítő.

A 14. fejezetben található *B-fa* kiküszöböli a bináris keresőfa gyengeségeit is. A fa felépítése biztosítja a folyamatos gyors karbantartási és keresési műveleteket, továbbá a tárolás is optimális módon történik, kihasználva napjaink számítógépeinek (merevlemezeinek) felépítését. Működése ugyan az előzőekhez képest jóval bonyolultabb algoritmusokat igényel, azonban érdemes vele foglalkozni, hiszen napjainkban is számos helyen találkozhatunk ennek gyakorlati használatával.

Az utolsó tárgyalt adatszerkezet a 12. fejezetben bemutatott *gráf* lesz. Ez szintén ismerős lehet, hiszen ez tulajdonképpen megfelel a matematika tárgyakból már előzőleg megismert gráfoknak. Jelen jegyzetben emiatt nem is szeretnénk elmerülni ezek gyakorlati használatában, hiszen az nagyon messzire vezetne, pusztán az adatszerkezetként való használat alapjait tekintjük át: milyen formában tudjuk eltárolni a

gráfokat, illetve hogyan tudunk hozzáérni a bennük eltárolt adatokhoz. A fejezet tartalmaz egy részletes megoldást a legrövidebb utak keresésére, de ezt tekintsük egy önkényesen kiragadott (látványos) példának a szakirodalomban található számtalan gráf algoritmus közül.

A jegyzetben szereplő pszeudokódokról

Az algoritmusok bemutatása mindenkor pszeudokódok segítségével történik. Ezek használata gyakran felvet kényes kérdéseket, mivel a konkrét programozási nyelvekkel ellentétben nincsenek egységes szabályok a pszeudokódok írására, így nehéz körülhatárolni, hogy milyen szintű műveleteket (pl. csere?) tekintünk a kódban alapműveletnek, vagy külön kifejtendőnek (bár jegyezzük meg, hogy ez a „hátrány” adja a pszeudokódok használatának létjogosultságát is, hiszen így szabadon határozhatjuk meg a használni kívánt absztrakciós szintet). A jegyzetben található kódok alapvetően alacsony szintűek, már-már a programozási nyelvekhez egészben hasonlóak, hogy ez is elősegítse az esetleges implementáció zökkenőmentességét. Ezt a szabályt azonban bármikor megszegjük, amikor a könnyebb olvashatóság azt megkívánja (pl. hal-mazműveleteket is használunk a kódokban).

A jegyzetben használt pszeudokódok az alábbi részekből állnak össze:

- **Bemenet:** Az algoritmus bemenetét jelentő paraméterek felsorolása. A paraméterek neve mellett mindenkor szerepel a paraméterek típusa, esetleg egy rövid leírás.
- **Kimenet:** Az algoritmus kimenete, ami alapvetően két formában jelenhet meg:
 - Függvények esetében a visszatérési érték: A pszeudokódban megengedjük, hogy egy függvénynek több visszatérési értéke is legyen. Mivel a visszatérési értéknek nincs „neve” így itt általában azt a változót adjuk meg névként, aminek az értékét visszaadja a függvény futása végén (néha előfordul, hogy ilyen változó nincs, ilyenkor a név egy egyéb, a visszatérési értékre jellemző szó).
 - Függvények vagy eljárások esetében a címszerint átadott paraméterek: Az algoritmus futása közben ezeket megváltoztathatja, ami hatással van a paraméterként átadott változókra is, emiatt ezeket is kimenetnek tekintjük. Névként itt a paraméter nevét használjuk.
- Függvény vagy eljárás fejléce, illetve törzse: Ez tartalmazza magát az algoritmust.
- **Függvény hívása:** Rekurzív függvények esetében ez a szakasz mutatja meg a meghívás pontos módját.
- **Felhasznált változók és függvények:** Ez a terület tartalmazza azokat a pszeudokódban használt változókat, illetve meghívott egyéb függvényeket, amelyek leírása a bemenet/kimenet részből nem egyértelmű.

A függvények és eljárások pszeudokódja megfelel a szakirodalomban már megsokott formának, emiatt ennek részletes bemutatását itt mellőzzük. Alapvetően a strukturált programozás eszközeit használjuk itt, esetenként kiegészítve objektum-orientált jellemzőkkel (néha a rövidebb leírás kedvéért megszegjük ezeket a szabályokat, de csak akkor, ha az nem okoz félreérteést).

Néhány további megjegyzés, értelmezési javaslat a pszeudokódokkal kapcsolatban:

- A változók esetében az olvashatóság érdekében nem adjuk meg külön azok deklarációját. Az egyes változók típusai sem szerepelnek abban az esetben, ha az egyértelmű (pl. tömbök címzésére használt változó nyilván egész lesz). Általában az alábbi elnevezéseket használjuk:
 - i, j, k, \dots : egész számok, tipikusan tömbök címzések
 - A, B, \dots : tömbök
 - p, q, e, r, \dots : elemre való hivatkozást tartalmaznak (a pontos típus implementációfüggő)

Amennyiben egy változó típusa nem egyértelmű, akkor az megtalálható a „Felhasznált változók és függvények” részben.

- Bizonyos esetekben szándékban nem adunk meg pontos típust. Például az adatszerkezetekben egy T betűvel helyettesítjük az eltárolni kívánt tartalom típusát, ezzel is azt hangsúlyozva, hogy az tetszőleges lehet (szám, szöveg, stb.). Ezeknél a típusoknál néha előünk valamilyen megszorítással, pl. hogy a T összehasonlítható legyen. Tehát lehet bármilyen típus, aminél el tudjuk dönten két elemről, hogy melyik kisebb.
- A tömbök indexelését (hacsak ezt külön nem jelezzük) minden 1-től kezdjük.
- Egy A tömb i . elemét az alábbiak szerint érjük el: $A[i]$. Néha hasonló értelemben használjuk az A_i jelölést. Ez utóbbit alapvetően ugyanazt jelenti, de itt az A -t inkább mint tetszőleges adatszerkezetet tekintjük, nem ragaszkodunk ahhoz (vagy legalábbis nem akarjuk kihangsúlyozni), hogy az tömb legyen. Néha mindenkét jelölés megjelenik egy kódban, ez általában nem okoz zavart, sőt, javítja az olvashatóságot, pl. $A[S[i]]$ helyett $A[S_i]$.
- Az $X.y$ formával jelöljük az X elem y tulajdonságát. Ez lehet akár egy tömb mérete, egy objektum mezője, egy szöveg első karaktere, stb.
- Az $f()$ vagy $X.f()$ formával jelöljük az f nevű függvény hívását, illetve az X elem f műveletének meghívását. Amennyiben nem nyilvánvaló, a „Felhasznált változók és függvények” rész tartalmazza, hogy mit csinál ez a függvény.
- Néhány speciális művelet, amiket gyakran használunk a kódokban:
 - $a \leftarrow b, a \leftrightarrow b$: értékadás, csere (hivatkozások esetében maguk a hivatkozások értéke változik, nem a hivatkozott elemé)
 - $a \wedge b, a \vee b, \neg a$: szokásos logikai operátorok (és, vagy, tagadás)
 - $a \in X, X \cup \{a\}$: szokásos halmazműveletek (eleme, unió)
 - $S \Leftarrow a, a \Leftarrow S$: az S sorba egy új elem elem elhelyezése, illetve kivétele
 - $V \Leftarrow a, a \Leftarrow V$: a V verembe egy új elem elem elhelyezése, illetve kivétele
 - \emptyset, \otimes : speciális konstansok, az üres illetve törlött tartalmat jelzik (pontos jelentésük az algoritmus leírásból derül ki)
 - $(a \leftarrow f ? b : c)$: a C nyelvekben megszokott művelet, amely szerint az a értéke legyen a b értéke akkor, ha az f igaz, illetve a c értéke egyébként

Végül egy fontos megjegyzés: néhány kivételtől eltekintve (nevezetesen algoritmusok) a jegyzetben megjelenő pszeudokódokra csak mint egy lehetséges implementációra kell tekinteni. Tehát a cél nem az itt szereplő kódok betűről-betűre való megtanulása, sokkal inkább az egyes technikák elvi háttérének alapos megértése, amiben segítséget nyújthat a pontos pszeudokód illetve a példát bemutató ábra. Amennyiben valaki valóban, minden részletre kiterjedően érti a végrehajtandó műveletek pontos menetét, akkor az ez alapján bármikor tud rekonstruálni egy jól működő pszeudokódot.

Köszönetnyilvánítás

Köszönettel tartozom Dr. Csink Lászlónak, aki a Programozás II. (illetve korábban Programozási Paradigmák és Technikák) nevű tárgy előadásait évek óta közösen tartjuk. Továbbá köszönettel tartozom Dr. Vámossy Zoltánnak, aki az évek folyamán számos értékes észrevétellel segítette a tárgy végső formájának kialakítását, majd pedig vállalta ennek a jegyzetnek a lektori feladatait is. Szintén köszönöm a sok segítséget Dr. Sergyán Szabolcsnak is, aki a Programozás I. tárgy oktatójaként, minden pedig az Algoritmusok, adatszerkezetek I. jegyzet szerzőjeként is napi szinten egyeztetünk és remélhetőleg egyeztetünk a jövőben is.

Végül, de nem utolsó sorban szeretném köszönetet mondani az Alkalmazott Informatikai Intézet munkatársai közül Cseri Orsolya Eszternek, Dr. Erdélyi Krisztinának, Nagy Tibor Istvánnak, Szabó Zsoltnak, Szántó Balázsnak és Urbán Andrásnak, aikik a Programozás II. tárgy laborjait tartották. Az évek folyamán számos értékes észrevételt fogalmaztak meg, amelyek beépültek ebbe a jegyzetbe is.

I. rész

Programozási paradigmák

1. fejezet

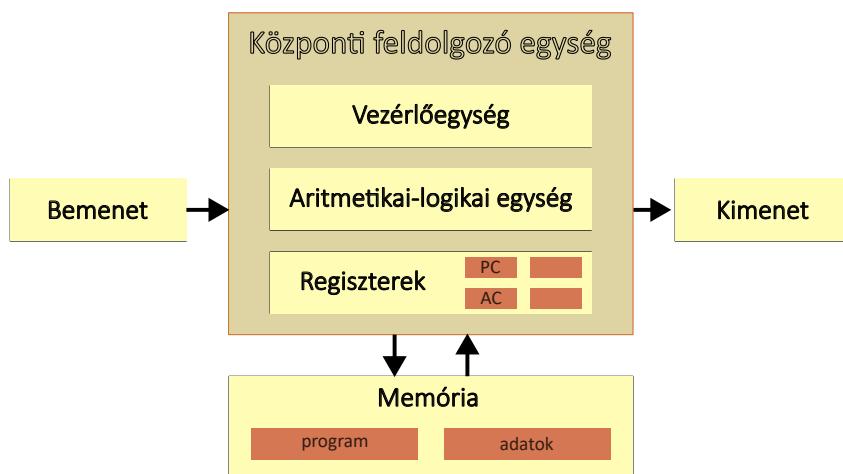
Imperatív Programozás

1.1. Imperatív megközelítés

1.1.1. Neumann elvű számítógép felépítése

Jelen jegyzet nem törekszik a számítógépes hardverrel kapcsolatos téma körök tárgyalására, azonban az imperatív programozási paradigma megértéséhez érdemes nagy vonalakban megismerni a napjainkban általánosan használt számítógépek elvi felépítését. Ezek a mai napig a Neumann János által 1945-ben publikált [13] elveken alapulnak. A Neumann által javasolt számítógép az alábbi részegységekből áll:

- központi feldolgozó egység, amelynek részei:
 - vezérlő egység,
 - aritmetikai logikai egység (ALU),
 - regiszterek,
- memória az adatok és utasítások tárolására,
- perifériák:
 - kimenet,
 - bemenet,
 - háttértár,
 - egyéb.



1.1. ábra. Neumann architektúra

Az ilyen módon megtervezett számítógép jellemzője, hogy az adatokat és a végrehajtandó utasításokat is egy közös memóriában tárolja. A vezérlő egység számon tartja, hogy a memóriában található program hánnyadik utasításánál jár a végrehajtás. Futtatás során ennek megfelelően kiolvassa a következő utasítást, és annak megfelelően utasítja a többi részegységet annak végrehajtására.

- Ezek tipikusan az adatokkal dolgoznak: valamelyen számítást végeznek vagy a memóriából írnak/olvasnak, esetleg a külső háttértár/perifériák irányába küldenek/fogadnak adatokat.
- Néhány utasításnak a vezérlésben van szerepe: egy utasítás végrehajtása után általában a következő utasításra lép a vezérlő egység, de feltételes vagy feltétel nélküli ugrások hatására ez megváltoztatható.

Ezekre az egyszerű lépésekre visszavezethető bármelyik manapság készített összetett alkalmazás (legen az szövegszerkesztő, operációs rendszer, vagy akár játék, stb.). A programozó feladata, hogy a programozási nyelv segítségével az összetett problémát visszavezesse ezekre az elemi lépésekre.

1.1.2. Az imperatív paradigmá jellemzői

A fizikai számítógéphez legközelebbi programozási nyelvek általában az imperatív programozási paradigmát használják. Ennek fő jellemzői az alábbiak:

- A program aktuális állapotát alapvetően a memória aktuális tartalma határozza meg. Ennek megfelelően egy program futása alatt azt értjük, hogy a memória tartalma változik, amit úgy is tekinthetünk, hogy a program egymást követően az egyes állapotokból átlép egy újba.
- Ezt valósítják meg az értékadó utasítások, amelyek célja az, hogy megváltoztassák a memória tartalmát. Természetesen van lehetőség a memória olvasására is.
- Ezeket az olvasásokat és értékadásokat persze megfelelő sorrendben kell végrehajtani, ezért szükség van feltételes végrehajtásra, és valamelyen ugró utasításra.
- Az imperatív programozás végső célja az, hogy a fentiek segítségével előállítsunk egy olyan utasításokból álló programot, amely bármely megfelelő kezdő memória állapotból (bemenet) az utasítások végrehajtásával addig változtatja a memória tartalmát, hogy az egy elvárt (kimenet) állapotba kerüljön át.

Elsőre ez nagyon absztrakt megfogalmazásnak tűnik, ami távol áll az általunk ismert szoftverfejlesztéstől, de valójában meglepően közel áll a valós hardver működéséhez. A napjainkban elterjedten használt imperatív programozási nyelveken írt programok túlnyomó része a változók értékének (tehát közvetve a memória állapotának) megváltoztatásáról szól.

Megjegyzés

De ennek az elméleti absztrakt megfogalmazásnak fontos szerepe van a programhelyesség bizonyítás során is, amikor matematikai módszerekkel szeretnénk belátni, hogy egy megadott program valóban jól működik.

Az imperatív paradigmá használata során a program végrehajtásának módja a „közvetlen vezérlés”, ami annyit jelent, hogy a végrehajtás pontosan a programozó által megadott sorrendben fogja érinteni az egyes utasításokat. Ez elsőre nyilvánvalónak tűnik, de a funkcionális és logikai paradigmánál látni fogjuk, hogy ez egyáltalán nem szükségszerű.

1.2. Memória kezelése

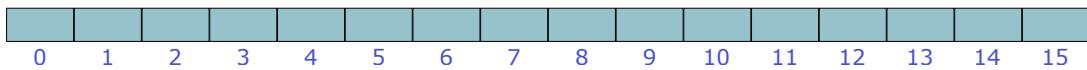
1.2.1. Típusok és változók

Egyeszerű típusok

Mint látható, az imperatív paradigmában központi eleme a memóriában tárolt adatok kezelése, ezért ennek lehetőségeit részletesen is megvizsgáljuk. Itt már eltávolodunk a hardvertől, csak az érdekel minket, amit egy programozó (sárga) szemüvegén látunk az egész rendszerből.

A napjainkban használt Neumann-elvű számítógépek a programokat és az azok által használt adatokat minden memóriában tárolják el. Habár egy magas szintű programozási nyelv esetében számos rétegen kell átverekedni magunkat hozzá (futtató környezet, operációs rendszer, stb.), de maga a memória egy lineárisan címzhető tárolóként képzelhető el, amelynek a memóriacím által meghatározott területeire tudunk adatokat írni/olvasni.

Ezeket a memóriaterületeket a későbbiekben a 1.2. ábrán látható módon fogjuk ábrázolni, ahol az egyes téglalapok jelképezik a külön-külön címzhető (1 byte méretű) memóriacellákat, az alattuk lévő számok pedig az egyes cellák memóriacímét mutatják.



1.2. ábra. Egy képzeletbeli 16 byte memóriát tartalmazó számítógép memóriájának ábrázolása.

A számítógépes programok működésük során általában kiolvassák, illetve módosítják az egyes memóriacellák tartalmát. Egy modern, magas-szintű programozási nyelven persze már nem az egyes cellák valódi címét kell használni, hanem ezt közvetve, a változóneveken keresztül tehetjük meg. A változó nevét úgy is felfoghatjuk, hogy azzal elneveztük a memória egy területét, és a későbbiekben ezzel a névvel tudunk erre a memóriacímre közvetlenül hivatkozni. A névhez tartozó memóriacím megjegyzése, illetve az esetleges szükséges további műveletek (pl. az egyes változók memóriacímének megállapítása, stb.) elvégzése a fordítóprogram feladata.

A név mellett minden változónak van valamilyen típusa. Ez meghatározza a változóban eltárolt értékek fizikai tárolási módját, illetve az azokon végezhető műveleteket. Például egy egész számot eltárolhatunk 32 biten, és a rajta értelmezhető műveletek lehetnek a alapvető aritmetikai műveletek (összadás, kivonás, stb.). Egy logikai érték eltárolható 1 de akár 8 biten is, itt értelemszerűen csak a logikai műveletek lesznek használhatók (pl. logikai „és”, illetve „vagy” kapcsolat).

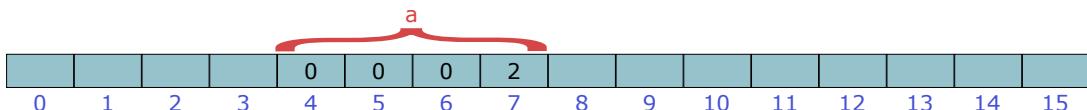
1.1. Algoritmus Változó értékkadás

1: $a \leftarrow 2$

Felhasznált változók és függvények

- **a** : Egy *egész* típusú változó.
-

Tehát ha létrehozunk egy **a** nevű *egész* szám típusú változót (ami tegyük fel, hogy 32 biten tárolódik a memóriában), és a programkódon belül ennek 2-t adunk értékül (1.1. algoritmus), akkor a memóriát az 1.3. ábrán látható módon képzeljük el.



1.3. ábra. A memóriában egy 32 bites **a** változó helyezkedik el tetszőleges helyen, amelynek az értéke 2.

A változó típusok, illetve az azokon végezhető műveletek programozási nyelvenként változhatnak, ezért ezekkel részleteiben nem foglalkozunk. A jegyzet során azt feltételezzük, hogy az alábbi alapvető típusok állnak rendelkezésre:

- **logikai** típus (értékei lehetnek: *igaz*, *hamis*),

- *egész* szám típus (pl. $-1, 0, 1, 2, \dots$),
- *egész⁺* jelű, természetes szám típus (pl. $0, 1, 2, \dots$),
- *szám* nevű, valós számokat tartalmazó típus (pl. $-3, 14$),
- *szám⁺* jelű, pozitív valós számokat tartalmazó típus (pl. $3, 14$),
- *szöveg* típus (pl. "Hello").

Ezek tényleges reprezentációja (tehát az általunk megadott érték fizikai memóriába való leképzésének módja) számunkra irreleváns, hiszen ez nem befolyásolja az algoritmusaink működését.

A pszeudokódokban megengedjük azt az egyszerűsítést, hogy általában nem deklaráljuk a változókat, illetve nem adjuk meg minden esetben a típusukat, ha ez egyértelműen kiderül a velük végzett műveletekből. Ettől persze eltérünk minden esetben, amikor ez szükséges.

Összetett típusok

A programozási nyelvekben általában lehetőség van saját összetett típusok létrehozására is (hívhatjuk ezeket rekordoknak, vagy struktúráknak is). Olyankor célszerű ezeket használni, amikor egymással szorosan összekapcsolódó egyszerű értékeket szeretnénk egy csoportként kezelní.

Az így kapott összetett típus mérete egyenlő az azt alkotó egyszerű típusok méretének összegével. Az egyes adattagok (mezők) külön-külön elérhetők, ezek típusa megegyezik a már említett egyszerű típusok valamelyikével, így a velük végezhető műveletek is ugyanazok. Alapesetben az összetett típusokon általában nem is értelmezünk műveleteket, azok csak a már meglévő típusok egy csoportba kapcsolására használatosak. De amennyiben ez szükséges, akkor saját műveleteket is definiálhatunk ezekre az új típusokra.

Példaként készíthetünk egy saját *Ember* típust, aminek adattagjai egy *férfi* nevű *logikai* érték (aminek értéke *igaz*, ha férfiról beszélünk és *hamis*, ha nőről), illetve egy *kor* nevű *egész* érték (ami az adott személy életkorát tartalmazza). Amennyiben az egészeket 32 biten, a logikai változót pedig 8 biten tároljuk, akkor az összetett típus mérete 5 byte lesz.

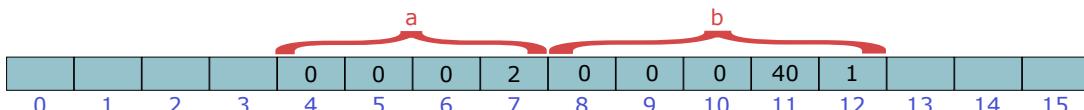
Amennyiben létrehozunk egy *b* változót is, amely *Ember* típusú, és az 1.2. algoritmusban látható módon értéket adunk az adattagjainak, akkor a memória az 1.4. ábrán látható állapotba kerül.

1.2. Algoritmus Összetett típus mezőinek beállítása.

- 1: $a \leftarrow 2$
 - 2: $b.\text{férfi} \leftarrow \text{igaz}$
 - 3: $b.\text{kor} \leftarrow 40$
-

Felhasznált változók és függvények

- *a* : Egy *egész* típusú változó.
 - *b* : Egy *Ember* típusú változó.
-



1.4. ábra. A memóriában egy 32 bites *a* változó helyezkedik el, és egy 40 bites *Ember* típusú *b* változó.

Az összetett típusok kívülről nézve egy típusként kezelhetők, és az alapvető értékadás/összehasonlítás műveletekben nem különböznek az egyszerű típusoktól.

- Értékadás során csak ugyanolyan típusú változónak lehet átadni egy összetett típusú változó értékét, és ilyenkor az összetett típus minden mezőjének adatai átmásolódnak. Ugyanez igaz egy paraméterátadás során is.
- Összehasonlítás során két azonos típusú összetett változó akkor egyenlő, ha minden mezőjük külön-külön is egyenlő.

Szándékosan nem kevertük az összetett típus ismertetésébe a lehetséges műveletek, illetve ezeken keresztül az osztályok, objektumok fogalmakat. Ezeket az összetett típusokat a fent vázolt egyszerűségükben szeretnénk megőrizni, és csak adatok tárolására használni.

Implementáció

Az összetett típusok és az objektumok fogalma programozási nyelvenként más-más formában jelenik meg. C# és C++ esetében a két fogalom jól elkülönül egymástól, az összetett típusokat „struct” kulcsszóval, az osztályokat pedig a már ismert „class” kulcsszóval hozzuk létre. Java esetében csak osztályokat tudunk létrehozni, így a jegyzetben szereplő összetett típusokat osztályokkal tudjuk majd helyettesíteni, persze így már nem lesznek igazak az értékadásra és összehasonlításra vonatkozó megállapításaink.

1.2.2. Statikus és dinamikus memóriakezelés

Alapvetően kétféle memóriakezelési módszert különböztethetünk meg:

- Statikus memóriakezelés esetében a memória lefoglalása nem igényel külön utasítást, ennek pontos módját a fordító már a fordítás során megállapítja. Ehhez meg kell állapítania a lefoglalandó memória méretét (ami adódik a típusból), illetve a használni kívánt memória tartományt (attól függően, hogy lokális vagy globális változóról van szó, esetleg egy objektum mezőjéről, stb.).
- Dinamikus memóriakezelés során a memória lefoglalását a programozó végzi el explicit módon az erre szolgáló LÉTREHOZ utasítással. Az így lefoglalt memória tipikusan egy kimondottan erre szolgáló nagyméretű területen kerül kiosztásra¹.

A dinamikus memória foglalás során az így lefoglalt memóriaterületnek nincs „neve”, miként azt a változóknál láttunk. Ilyenkor csak a lefoglalt memóriatartomány címét ismerjük, és ez alapján közvetve tudunk majd hozzáérni az itt tárolt adatokhoz.

A memória foglaláskor megadhatunk egy típust, és a későbbiekben úgy tekintjük, hogy a lefoglalt memóriaterület egy megadott típusú (név nélküli) változót tartalmaz, pl. LÉTREHOZ(*Ember*).

A lefoglalás módjához illeszkedik a memória felszabadításának módja is:

- Statikus memóriakezelés esetében a memória lefoglalásához hasonlóan annak felszabadítása is teljesen automatikus, a fordító végzi. Ennek ideje illeszkedik a változók élettartamához, tehát úgy tekinthetjük, hogy abban a pillanatban, amikor a változó élettartama lejárt, a változó megszűnik, az általa lefoglalt memória pedig felszabadul.
- Dinamikus memóriakezelés során a memória felszabadítása is a programozó feladata, bár ez erősen függ a programozási nyelvtől. A későbbiekben azt feltételezzük, hogy minden lefoglalt memóriát fel kell szabadítani a FELSZABADÍT hívással. Ennek ideális ideje az a pont, amikor már tudjuk, hogy nem lesz többé szükség a változóra.

Implementáció

Több modern programozási nyelv rendelkezik „automatikus szemétygyűjtő” mechanizmus-sal. Ennek alapelve, hogy a futtató környezet futásidőben megkeresi azokat az objektumokat, amelyek már nem érhetők el a programból egyik referencia-n keresztül sem, és ezeket automatikusan felszabadítja. Java és C# kódok esetében ezért nem lesz szükségünk a felszabadítás utasításra, de a pszeudokódokban meghagyjuk ennek használatát.

¹heap, ami nem összekeverendő a később bemutatott kupac adatszerkezettel, ami szintén heap angolul

1.3. Algoritmus Dinamikus és statikus memóriafoglalás.

```
1: a  $\leftarrow$  2
2: p  $\leftarrow$  LÉTREHOZ(Ember)
3: p.férfi  $\leftarrow$  igaz
4: p.kor  $\leftarrow$  40
5: FELSZABADÍT(p)
```

Felhasznált változók és függvények

- *a* : Egy *egész* típusú változó.
- *p* : Hivatkozás egy *Ember* típusú memóriaterületre.

Az 1.3. algoritmuson láthatóan az *a* változót statikusan hoztuk létre, az *Ember* típusú változót pedig dinamikusan, amit a használat után meg is szüntetünk. A *p* változó egy mutató/hivatkozás/referencia, aminek pontos szerepét tárgyalja a következő fejezet.

A későbbiekben az egyszerű típusokat mindig statikus, az összetett típusokat pedig minden dinamikus formában hozzuk létre.

Implementáció

Ez a megállapodás többé-kevésbé megegyezik a programozási nyelvekben egyébként is megszokott módszerrel. Például Java esetében a primitív típusok minden statikus, az objektumok pedig minden dinamikus létrehozást igényelnek. Más nyelvek (C++) persze lehetnek sokkal rugalmasabbak is ezen a téren.

1.2.3. Mutatók és referenciák

Az 1.3. ábrán látható tárolási módot hívjuk érték szerinti tárolásnak. Amennyiben külön ezt nem említjük, akkor általában erre gondolunk, tehát ilyenkor az eltárolni kívánt adat közvetlenül a változó neve által képviselt memóriacímre kerül (ezért is nevezzük közvetlen/direkt tárolásnak/hozzáférésnek is). A programozási nyelvek általában így tárolják el az egyszerű (primitív) típusokat, mivel ez garantálja a leggyorsabb működést, illetve az optimális helykihasználást.

A közvetlen hozzáférésen túl azonban arra is lehetőség van, hogy egy adott memóriacímen lévő adatot ne egy hozzárendelt néven keresztül érjünk el (sőt, elképzelhető, hogy magának a memóriacímnek nincs is neve), hanem közvetve egy másik változón keresztül.

Ezt a másik változót nevezzük mutatónak/referenciának (a gyakorlatban a két fogalom nem ugyanazt takarja, de a mi esetünkben eltekintünk a különbségektől²), esetleg hivatkozásnak, ezt a három fogalmat azonosnak tekintjük. Ezeket úgy a legegyszerűbb elköpzelni, hogy ezek olyan változók, amelyek tartalma egy memóriacím, az itt látható ábrákon és példákon is így fogjuk őket ábrázolni. Ezek gyakorlati értelme pedig az, hogy rajtuk keresztül közvetve tudunk írni/olvasni erről a memóriacímről.

Bár ezek a változók csak egy memóriacímet tárolnak, a típusbiztonság érdekében itt is célszerű jelölni, hogy azon a hivatkozott memóriacímen milyen típusú változó található. Ennek megfelelően pl. egy *Ember* változóra hivatkozó referenciát *Ember* típusúnak fogunk jelölni a pszeudokódokban. A konkrét programozási nyelvtől/implementációtól függően ez különféle megvalósításokat jelenthet, tehát próbáljuk meg általánosabban felfogni a típus működését: egy olyan típus, amelynek tartalma egy hivatkozás egy másik változóra/adatra/memóriaterületre.

A mutatók esetében bevezetünk egy speciális értéket (amit a \emptyset jellel jelölünk), ami azt jelenti, hogy az adott mutató éppen nem mutat semmilyen érvényes memóriacímrre.

A direkt és indirekt hozzáférés néha félreérthető lehet, nézzük meg pl. az 1.2. algoritmus 2. sorát, ahol a *b* egy közvetlen elérésű változó, aminek beállítottak a mezőjének az értékét. Az 1.3. algoritmus 3. sora pontosan ugyanigy néz ki, de itt a *b* egy mutató, amin keresztül módosítjuk a *b* által mutatott memóriacímén lévő *Ember* változó egy mezőjének értékét. Az egyértelműség kedvéért bevezethetnénk külön jelöléseket, hogy megkülönböztessük egymástól a két hozzáférésrési módot, azonban ezek csak rontanák a pszeudokódjaink olvashatóságát, ezért ezeket a elhagyjuk, csak megjegyzésekkel utalunk az elérés módjára azokon a helyeken, ahol ez esetleg félreérthető helyzetet teremtene.

²már csak azért is, mert maga a referencia szó másat jelent pl. C# és C++ esetében

Az egyértelműség kedvéért ezért megegyezhetünk abban, hogy a későbbiekben a dinamikusan létrehozott (tehát összetett típusú) objektumokra minden referencián keresztül fogunk hivatkozni, a statikusan létrehozott (tehát egyszerű típusú) változókra pedig minden közvetlen módon a változó nevével. Ennek ellenére az 1.4. ábrán látható példa, ezt tehát tekintsük csak egy kósza gondolatnak, a későbbiekben ezzel a lehetőséggel nem fogunk elni.

Érdemes azt is megvizsgálni, hogy milyen műveletek tudunk értelmezni a mutatókon:

- Az értékadást ezeknél a változóknál úgy tekintjük, hogy magának a referenciának adunk értéket (tehát megadjuk, hogy melyik memóriacímre mutasson).
 - Amennyiben az értékadás baloldalán egy mutató, a jobb oldalán pedig egy újonnan létrehozott változó van, akkor a mutató ennek a memóriacímét fogja eltárolni (a 2. sorban látható értékadás során ezért a *p* értéke 8 lesz, ami a létrehozott *Ember* memóriacíme).
 - Amennyiben az értékadás jobb oldalán is egy referencia van, akkor az abban a referenciában tárolt memóriacímét fogja megkapni (5. sorban emiatt a *q* is a 8 értéket kapja), tehát mindenkit hivatkozás ugyanoda mutat az értékadás után.
- Amennyiben pedig az összetett típusoknál már megismert *p.adattag* formátumot használjuk (6. sor), akkor az minden a *p* változó által hivatkozott memóriaterületen lévő elem *adattag* mezőjére fog vonatkozni (ami egyértelműen következik abból, hogy magának a mutatónak nincsenek adattagjai, az csak egy memóriacímét tárol).
- Amennyiben két mutatóra az összehasonlító operátorokat használjuk (= vagy \neq), akkor minden az általuk tárolt memóriacímek értékét hasonlíjuk össze, és nem a memóriacímek által hivatkozott objektumok adatait (ennek megfelelően a 9. sorban található összehasonlítás értéke igaz lesz).

Erre mutat egy példát az 1.4. algoritmus. A képzeletbeli számítógép memóriájában ez az 1.5. ábrán látható állapotot eredményezi a felszabadítás előtt. A felszabadítás után a 8-12. memóriacelláig eső részek újra szabad memóriaként jelennek meg.

1.4. Algoritmus Műveletek referenciákkal

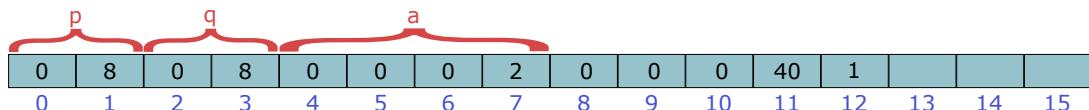
```

1: a  $\leftarrow$  2
2: p  $\leftarrow$  LÉTREHOZ(Ember)
3: p.férfi  $\leftarrow$  igaz
4: p.kor  $\leftarrow$  40
5: q  $\leftarrow$  p
6: ha p.férfi akkor
7:   ...
8: elágazás vége
9: ha p = q akkor
10:  ...
11: elágazás vége
12: p  $\leftarrow$   $\emptyset$ 
13: FELSZABADÍT(q)

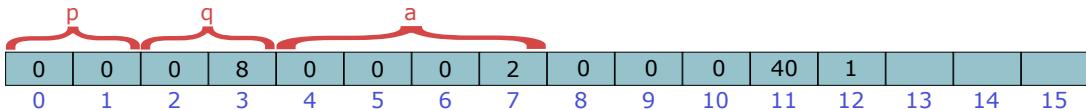
```

Felhasznált változók és függvények

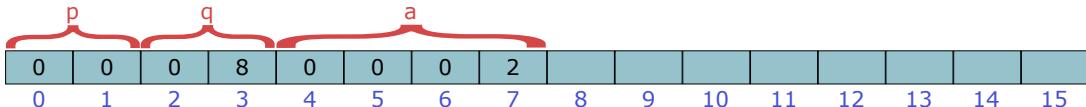
- *a* : Egy *egész* típusú változó.
- *p* : Egy *Ember* típusú változó, tehát hivatkozás egy *Ember* típusú memóriaterületre.
- *q* : Szintén egy *Ember* típusú változó.



1.5. ábra. Az 1.3. algoritmus állapota az 5. sorban. A mutatókat itt 16 bitesnek tekintjük, ez változhat programozási nyelvenként.



1.6. ábra. Az 1.3. algoritmus állapota a 12. sorban.



1.7. ábra. Az 1.3. algoritmus állapota a 13. sorban.

Vegyük észre, hogy a létrehozott **Ember** objektumhoz a memóriában nem tartozik közvetlen változó, a lefoglalt memóriaterületnek „nincs neve”. Ezt gyakran a mutatótól után **p**-nek nevezik, de ezt próbáljuk meg elkerülni, mivel nem igazán szerencsés. Mert ugyan tény, hogy a létrehozáskor a **p** referencia hivatkozik erre az objektumra, de a későbbiekben a **q** is rámutat, sőt, a 12. sor után a **p** már nem is hivatkozik erre az elemre.

Továbbá látható, hogy a **p** és **q** változók nem szűnnék meg létezni csak azért, mert az objektumot felszabadítottuk a 13. sorban. Ez sok veszélyt rejt magában, ugyanis az ezt követő programkódban bármilyen **p**-n vagy **q**-n keresztül végzendő indirekt hozzáférés futásidéjű hibát okoz.

Implementáció

Valójában önkényesen tettünk megállapításokat az alapján, hogy az egyes változó típusoknál (egyszerű/összetett/osztály) milyen címzést használunk, ez ugyanis ismét egy programozási nyelv függő jellemző. Az alapvetően elmondható, hogy szinte minden nyelv közvetlen címzéssel kezeli az egyszerű változókat (bár elég jelentős ellenpélda erre a Python). C# esetén az összetett típusok közvetlen címzést használnak, az osztályok viszont minden referenciaján át érhetők el. C++ esetében az objektumokat elérhetjük referenciaján át, de ugyanígy közvetlen címzéssel is létre lehet hozni őket. Java esetében ahogy említettük, eleve nincs is összetett típus, csak az osztályok, tehát ott minden referenciaját használunk.

1.3. Procedurális paradigmá

1.3.1. Absztrakciós szintek

A fejezet további részeiben áttekintünk különféle programozási paradigmákat, amelyek egyik jellegzetessége az lesz, hogy időről-időre egyre magasabb absztrakciós szinteket értek el. Az absztrakció szó jelentése elvonatkoztatás, tehát egy komplex doleg leegyszerűsítése, a fontos dolgok kiemelése, a kevésbé fontos részletek elhagyása (pl. a metró hálózat térképe, ami csak nagy vonalakban mutatja az állomások helyzetét, nem is törekszik a pontos ábrázolásra). Ez természetesen veszteségekkel jár, de az emberi elme korlátai miatt erre kényszerülünk: a fejünkben lévő hardver egyszerűen képtelen egy időben kezelni bizonyos méretet meghaladó problémákat, ezért azt kénytelenek vagyunk feldarabolni/egyszerűsíteni.

Ugyanez a probléma felmerült a szóterfejlesztés területén is. Egy operációs rendszer forrás kódja akár 50 millió soros is lehet, aminek teljes részletes felépítését senki se tudja egy időben a fejében tartani, hogy átlássa annak minden utasítását. Ezért itt is szükség van az absztrakcióra, amelynek különböző szintjei vannak: azt a részt, amelyikkel éppen dolgozunk, maximális pontossággal akarjuk látni, a környező részeket még többé-kevésbé részletesen, a távolabbi modulokat viszont már egyáltalán nem.

Kezdetben nagyon rövid program kódokat írtak, itt nem volt szükség az absztrakcióra. Azonban ahogy növekedik a megoldandó feladatok mérete, egyre magasabb absztrakciós szintekre volt szükség, amelyek segítségével elfogadható méretűvé lehet alakítani az egyébként kezelhetetlenül komplex rendszereket.

Az egyre magasabb absztrakciós szintek elérésének módja a *részletek elrejtése*. Tehát olyan módszereket kellett kidolgozni, hogy a programozók minden az aktuálisan kiválasztott problémát lássák csak a lehető legnagyobb részletességgel, miközben a többi részlet rejtve marad (ettől még persze ott vannak, és bármikor azokra is rá lehet fókusználni).

1.3.2. Gépi kód

A már említett Neumann-elvű gép működési elve az, hogy a végrehajtóegység lépésről-lépéstre végrehajtja a szekvenciálisan megadott programkódot. A legalacsonyabb absztrakciós szint esetében közvetlenül ezeket a processzor számára értelmezhető kódokat kell megadni.

Ahogy az 1.1.1. fejezetben látható, a program célja a memória tartalmának a módosítása, illetve a perifériákkal való kommunikáció. A processzor alapesetben egyesével végrehajtja az egymást követő utasításokat, néha viszont szükség lehet ennek megtörésére:

- Feltétel nélküli ugró utasításokkal: a program végrehajtása az ugró utasítás által megadott memóriacímen folytatódik.
- Feltételes ugró utasításokkal: amennyiben egy feltétel teljesül (ezen a szinten ezek elég egyszerűek lehetnek csak, pl. egyik változó értéke nagyobb-e mint egy másik?), akkor a végrehajtás egy megadott memóriacímen folytatódik, ha pedig nem teljesült a feltétel, akkor a soron következő utasítással.

Napjainkban már csak extrém esetekben kell ilyen alacsony szintre ereszkedni, de érdemes megjegyezni, hogy már ezzel a néhány utasítással is le lehet kódolni tetszőleges algoritmust. Ami nyilvánvaló, hiszen a később kidolgozott magasabb szintű programozási nyelvek fordítói is végső soron gépi kódot állítanak elő.

```
66 A1 00 00 00 00 66 8B 1D 04 00  
00 00 66 39 D8 7F 08 66 A3 08 00  
00 00 EB 07 66 89 1D 08 00 00 00
```

1.8. ábra. Két változó közül a kisebbik értékének másolása egy harmadik változóba (x86 gépi kód).

A rövid példából is látható (1.8. ábra), hogy az így készült kódok megírása és bármiféle módosítása meglehetősen nehézkes, ezért ez a módszer csak nagyon kisméretű feladatoknál használható. Ez a legalacsonyabb absztrakciós szint, tulajdonképpen a teljes programkód minden utasítása egy szinten, lineárisan jelenik meg, nincs benne semmiféle struktúra, részletek elrejtése, stb.

1.3.3. Assembly

A gépi kód nagyon alacsony szintje szinte lehetetlenné teszi a könnyen olvasható/karbantartható programkódok elkészítését. Ezért megjelentek az első assembly fordítók, amelyek alapvetően nem csökkentették a gépi kód előnyeit, mégis egy sokkal kényelmesebb környezetet nyújtottak. Ezt már néhány egyszerű technikai fejlesztéssel is el lehet érni:

- az egyes utasításokhoz szavakat jelöltek (ezek az úgynevezett mnemonikok);
- az egyes változókhöz változóneveket rendeltek, így már nem közvetlenül a memóriacímet kell megadni (és persze automatikusan kiszámolja a fordító a változók memóriabeli helyét);
- a programkódon belül is címkéket lehet megadni, tehát a kódbeli ugrásoknál se közvetlen memóriacímekkel kell dolgozni.

Ehhez persze már szükséges egy *fordító*³ program, ami az így szövegesen megírt assembly kódot lefordítja megadott processzor operációs kódjaivá, a memória címkék helyét kiszámolja és ezeket a számokat behelyettesíti a kódba, stb. Erre mutat példát az 1.9. ábrán látható assembly kód, ami lefordítva pontosan az 1.8. ábrán már megismert kódot adja.

```
section .text
    mov ax, [a]
    mov bx, [b]
    cmp ax,bx
    jg a_nagyobb
    mov [c],ax
    jmp vege
a_nagyobb:
    mov [c],bx
vege:

section .data
    a dw 5
    b dw 8
    c dw 0
```

1.9. ábra. Az *a* és *b* változók közül a kisebbik értékének másolása a *c* változóba (x86 assembly kód).

Az absztrakciós szint emelkedése látványos, ez a részletek elrejtését jelenti:

- a fordító elvégzi a programozó helyett az egyértelmű technikai számításokat, pl. a deklarált változók alapján azok memóriacímei, programkódban található címkék memóriacímei, ugró utasítások relatív lépésmérete, stb., a címkék tehát elrejtik a memória tényleges felépítését;
- az assembly kód más-más processzorokra lefordítva más-más gépi kódot eredményezhet, tehát a szövegesen megírt program elrejti a tényleges processzor architektúra részleteit.

Az assembly használata már koránt sem olyan extrém, mint a gépi kódé. Maximális teljesítmény elérésére érdemekben lehet megfontolni a használatát egy-egy részfeladat esetében, de nagyobb feladat megoldására nem alkalmas.

1.3.4. Procedurális programozás

Az assembly nagy segítséget nyújt a technikai problémák megoldásában (memóriacímek számítása, mnemonikok operációs kódoknak való megfeleltetése, stb.), a kód felépítése azonban még teljes egészében megfelel a gépi kódnak, hiszen ugyanazokat az utasításokat kell itt is megadni, csak ember számára olvashatóbb formában. Nagyobb programok esetében azonban ez már nem elég, hiszen több ezer soros assembly kódokat már nehéz értelmezni, ezért további absztrakciós szint növelésre volt szükség.

³compiler

Egy nagyobb rendszer fejlesztése során általában felmerülnek olyan részproblémák, amelyek megoldása egymáshoz nagyon hasonló utasítássorozatokat igényel, pl. ugyanazokat a számításokat kell elvégezni különféle bemeneti adatokon. Ezeket az azonos részeket nem érdemes újra és újra lekódolni, ehelyett célszerűbb ezeket kiemelni paraméterezhető és visszatérési értékkel rendelkező függvényekbe.

Ez a gondolat általánosítható a teljes program megtervezésére is. Ezt a szemléletmódot követi a „*procedurális programozás*”, amely esetében a megoldandó komplex feladatot felbontjuk több, egymással együttműködő (egymást hívó) eljárásra. Ezek az eljárások önmagukban is elkészíthetők, tesztelhetők, majd ezekből felépíthető a komplex megoldás.

Azon túl, hogy ez egy remek technikai egyszerűsítés, a szemléletmód magával hozott számos újítást is. A egyes eljárások önálló életet élnek, és megjelentek az alábbi újdonságok:

- eljárások paraméterezése, paraméterátadási módok (érték szerint, cím szerint);
- függvények visszatérési értéke;
- eljárások saját lokális változói, ezeken keresztül a változók hatókörére és élettartamára vonatkozó fogalmak;
- megfelelő implementáció esetén a rekurzió lehetősége.

Ennek legnyilvánvalóbb előnye a *kódújrafelhasználás*, hiszen nem szükséges ugyanazt a programkódot többször megírni. Ennek nem az idő/hely hiány a fő oka, hanem a rendszer későbbi karbantarthatósága. Egy jól megírt eljárás később sok helyen használható, illetve annak esetleges javítását/módosítását elég egy helyen elvégezni.

Az absztrakciós szint növekedése itt is a *részleteket elrejtésével* jár: a függvény meghívásakor annak csak a neve és a paraméterei láthatók a programkódban. Épp elég tudni, hogy az meg fog oldani egy adott részfeladatot, a megoldás részletes menete már nem érdekes (persze ha a programozó szeretné, akkor azt is megnézheti a függvény implementációjánál).

1.3.5. Moduláris programozás

A szoftverek növekedése és folyamatos terjedése miatt felmerült annak igénye, hogy a gyakran használt funkciókat önálló egységekkel lehessen fejleszteni, amelyeket később bármelyik fejlesztésbe be lehet csatolni. Természetesen nem úgy, hogy átmásolunk program részleteket, hanem ennek valamilyen fenntartható megoldást kellett keresni.

Ez vezetett a „*moduláris programozás*” igényéhez. Ennek kulcs gondolata az, hogy a fenntarthatóbb szoftverfejlesztés érdekében nem egymástól független nagy szoftvereket kellene fejleszteni, hanem ezek helyett kisebb, egymástól független általános modulokat. A végső szoftver elkészítése pedig már csak ezen modulok egymáshoz illesztését (szükség esetén kiegészítését) jelentené és ezek együttműködése adná a működő alkalmazást.

Ehhez számos technikai problémát meg kellett oldani. A moduláris programozás nem azt jelenti, hogy egyszerűen egymás mellé másoljuk a modulok forrásait és azokat egyben próbáljuk lefordítani, hiszen a különböző modulokban lehetnek azonos nevű függvények és változók, ez pedig névütközésekhez vezet.

Ezért tovább kellett bővíteni a programozási nyelvek eszköztárát:

- megjelentek a névterek, amelyek megoldották a névütközések problémáját;
- láthatósági szintekkel lehet biztosítani a modul belső konziszenciáját;
- elvált egymástól az interfész és az implementáció
 - az interfész egy felületet ad, amelyen keresztül lehet magával a modullal kommunikálni;
 - az implementáció meghatározza, hogy a modul milyen módon teljesíti a feladatait;
- az egyes modulok követelményként igényelhetnek egyéb modulokat,
- a modulok programozási nyelvi elemként is megjelentek.

Ez még magasabb szintű *kódújrafelhasználáshoz* vezet, hiszen ezek a modulok már teljesen önálló életet élnek, változtatás nélkül beépíthetők különféle rendszerekbe. A modulok egymástól függetlenül fejleszthetők, saját verziószámuk lehet, ami újszerű problémákat is fog okozni (dependency hell).

Ez vezetett az absztrakció egy következő szintjéhez, további *részletek elrejtéshez*: az egyes modulok meghatározták, hogy azok melyik változót és függvényeit tudják elérni más modulok (ezzel egy interfész nyújtanak a külvilág felé). A modul belseje (az implementáció) viszont már teljesen rejtve marad.

1.3.6. Strukturált programozás elve

Kezdetben a programozás általános irányelvek nélkül történt, a cél pusztán az volt, hogy az elkészült szoftver teljesítse a szükséges követelményeket. Az erőforrások szűkössége miatt esetleg felmerültek optimalizáziós igények, de a program felépítése mint olyan, nem volt prioritás. Ugyanazt a feladatot azonban többfélé utasítássorozattal is meg lehet oldani, ezért megjelentek programkészítési elvek, amelyek különféle szemléletmódot követve próbálták meghatározni a követendő utat.

A Neumann-elvű számítógép alapesetben egymás után hajtja végre az utasításokat, kivéve, ha eltérítik ettől a feltételes vagy feltétlen elágazások. A kezdeti imperatív programozás egy-az-egyben leképezte ezt a sémát: a program alapvetően egymást követő utasításokból állt, ezek végrehajtási sorrendjét pedig különféle vezérlési szerkezetekkel lehetett befolyásolni (amelyeket a fordító végső soron átalakított a már megismert különböző ugró utasításokra). Mivel a programozók ezek teljesen adhoc módon használhatták, ez jelentősen megnehezítette a kód olvashatóságát. Ezért már a 60-as években felmerült az igény, hogy a programok készítését olyan szabályokkal támogassák, amelyek segítik annak későbbi karbantartását.

A jelentős részt Edsger W. Dijkstra nevéhez kapcsolható „*strukturált programozás*” egy programépítési alapelv, amely szabványosítja a lehetséges vezérlési szerkezeteket, ezzel korlátozza is ezek számát:

- Szekvencia: egymást követő utasítások, amelyeket a megadott sorrendben kell végrehajtani. Ezek lehetnek elemi utasítások, vagy akár eljárás hívások.
- Feltételes elágazás: a program aktuális állapotát vizsgáló feltételtől függő irányba folytatódik a program végrehajtása a megadott ágak közül.
- Feltételes ciklus: addig ismételi a ciklusmag műveleteket, ameddig egy feltétel teljesül.

Fontos megállapítása az elvnek, hogy nincs tetszőleges helyre ugró utasításunk, csak a fenti vezérlési szerkezeteket lehet használni. Az egy másik kérdés, hogy a fordító program természetesen az elágazásokat és a ciklusokat a gépi kódnál már megismert feltételes ugró utasításokra fogja visszavezetni, de ez a programkódban már nem látszik.

Az alapelv a programtervezésben is segítséget nyújt, mivel iránymutatást ad a feladat felülről lefelé irányuló megoldására. A komplex problémát az így megadott szerkezetek segítségével fel kell bontani több kisebb problémára (amelyeket a fenti vezérlési szerkezetek kötnek össze). Ezt a felbontást pedig addig kell ismételni, amíg el nem jutunk olyan elemi lépésekig, amelyekre már van utasításunk.

1.4. Objektumorientált paradigmá

1.4.1. Alapelvek

A procedurális programozás során élesen elváltak egymástól az adatok és az azokon végzett műveletek. Az egyes eljárások saját belső lokális változóikat használták a részfeladatok megoldására, azon adatok azonban, amelyekre több eljárásnak is szüksége volt, globális változókban jelentek meg. Ezek a változók a program bármely részéből elérhetők, írhatók és olvashatók voltak, ami nagy rugalmasságot adott a programozók számára, azonban számos veszéllyel is járt:

- Mivel a programok különböző részei azonos helyen lévő adatokkal dolgoztak, az egyik eljárás módosítása (pl. az adatokat máshogyan kell eltárolni, mint eddig) magával vonta azt, hogy a program más, ettől egyébként elvileg független funkciókat megvalósító részeit is módosítani kell.
- Emiatt a program méretének növekedésével annak karbantarthatósága rohamosan csökkent, hiszen bármilyen módosítás a forráskódban nem várt mellékhatásokkal járhatott a program ettől elvileg független funkcióiban is.
- A hibakeresés is nehézkes, hiszen ha valahol megjelenik egy hiba (egy változó hibás értéket tartalmaz), akkor azt nehéz lokalizálni, mivel a program bármelyik része hozzáférhetett bármelyik adathoz.

Ahogy egyre nagyobb és komplexebb szoftverek készültek, egyre inkább előtérbe kerültek a kezdetleges programozási módszertanok hátrányai. Ez vezetett az 1960-as évektől erősődő „szoftver krízishez”, ami többek között azt jelentette, hogy a méret növekedésével a szoftverek fejlesztése és főleg azok karbantartása nagyon bizonytalanná vált költség, idő, és minőség szempontokból. Ezért újszerű programépítési elvekre volt szükség.

Az egyik ilyen módszertan az „*objektumorientált programozás*” (OOP), ami alapjaiban közel áll az eddig tárgyalta megoldásokhoz, azokhoz képest azonban egy új, magas absztrakciós szintet vezetett be. A korai kezdetek (1962 Simula, 1972 Smalltalk) ellenére a paradigma csak később, a C++, Java, C# nyelvek megjelenésekor vált egyeduralkodóvá.

A procedurális megoldás hátránya az volt, hogy külön kezelte az adatokat és az azokon dolgozó eljárásokat. Ezzel szemben az OOP szemlélet újdonsága az, hogy észrevette az adatok és a rajtuk végezhető műveletek elválaszthatatlanságát, ami jelentősen megváltoztatta a programtervezési elveket. A strukturált programozás az adatokat egy nagy globális egységnak tekintette, az azokon végzett műveleteket pedig a már megismert vezérlési szerkezetek segítségével darabolta a kezelhető méretig. Az OOP viszont első lépésként megvizsgálja a probléma részleteit és ezeket próbálja meg modellezni (milyen adataik vannak, milyen műveleteket tudnak végrehajtani). Egy program tehát több, egymástól független, kisebb részproblémákat megoldani képes egységekből áll (amelyek tartalmazzák a megoldáshoz tartozó adatokat, műveleteket), az elkészült alkalmazás pedig ezek interakcióján alapul.

Itt érdemes megjegyezni, hogy az OOP nem leváltotta a hagyományos strukturált programozás alapélémeit, a már jól ismert szekvencia, elágazás, ciklus, stb. továbbra is megmaradtak alacsony szinten (hiszen a metodusok továbbra is ezeket tartalmazzák). Magasabb szinten azonban a teljes komplex probléma felbontására egy újszerű megközelítési lehetőséget adott.

A módszer alapelveiként az alábbiakat szokás kiemelni:

- Absztrakció: a már említett magasabb szintű tervezés lehetősége. Célja a való világ komplexitásának elrejtése a magasabb tervezési szinteken.
- Egységebzárás: az adatok és a rajtuk végezhető műveletek egy egységbe vannak zárva, ami biztonságosabb és könnyebben karbantartható kódot eredményez.
- Öröklődés: az egymáshoz hasonló objektumok leírhatók egy ősosztálytal, illetve a leszármazottak specializációjával. Ez támogatja a kód újrafelhasználhatóságát.
- Polimorfizmus: biztosítja, hogy egy megadott helyzetben különböző típusú objektumok is megjelenhessenek, amelyek egymással felcserélhetők. Ez biztosítja a kód rugalmasságát és javítja annak olvashatóságát.

1.4.2. Objektumok

Az OOP programok alapvető alkotóelemei az objektumok. A modellezés során fel kell mérni, hogy a feladat megoldásához milyen és hány darab objektumra van szükség, ezek pedig hogyan működnek majd egymással együtt. Ezeket az objektumokat kell implementálni, majd ezt követi azok használata: létre kell hozni a feladat megoldásához szükséges objektumokat, majd ezek egymás közötti interakcióik segítségével elvégzik a megoldáshoz szükséges lépéseket. Ez az interakció elképzelhető úgy, hogy az „egyik objektum üzenetet küld egy másiknak”, de mi a földhözragadtabb „egyik objektum meghívja a másik egy műveletét” terminológiát fogjuk használni, ami ugyanazt jelenti.

A teljes probléma objektumokra való darabolása egy természetes és egyszerű módját adja a nagyméretű feladat kisebb részekre bontásának. Ez *csökkenti a komplexitást*, mivel több, egymástól független kisebb kódot minden egyszerűbb karbantartani, mint egy nagyon.

Objektum alatt az egy részprobléma megoldáshoz szükséges adatok és a rajtuk végezhető műveletek együttesét értjük. Ez lényeges különbség a már megismert összetett típusokkal szemben, mivel azok csak az adatok tárolásáért feleltek, a műveleteket külső eljárások végezték rajtuk. A módszer előnye az úgynevezett *egységbázárás*: az adatok és a rajtuk végezhető műveletek egymástól elválaszthatatlanok. Egy objektum belső állapotát nem változtathatja meg másik objektum, illetve az objektum műveletei sem foglalkoznak más objektumok feladataival.

Ez azért fontos, mert így lehet egyszerűen garantálni, hogy minden objektum belső állapota (annak adatai) minden hibátlan legyen. Mivel a saját adatait csak maga az objektum éri el a saját műveletein keresztül, ezért ha ezek jól működnek, akkor biztosan nem fogja más elrontani a belső változók értékét. Mintha egy védőburok lenne az objektum körül, amely megvédi annak belső állapotát. Ha a programozó mégis lazítani szeretne ezen a szabályon, akkor azt a különböző láthatósági szintekkel megteheti.

Ez tudja garantálni a *robustusságot*, tehát garantálható, hogy az objektum belső állapota minden konzisztens legyen. Példaként nézzünk egy terméket reprezentáló objektumot, aminek vannak *ár* és *áfa* nevű belső mezői. Lehet egy *Árváltozás* metódusa, ami módosítja az árat és persze azonnal aktualizálja a hozzá kapcsolódó áfa mező értékét is. Mivel más objektumok nem férhetnek hozzá ezekhez az adatokhoz, így biztosak lehetünk benne, hogy a két érték minden szinkronban lesz.

Ez egy nagyon jó megoldást jelent a *felelősségek kezelésére* is. minden objektumnak csak a saját műveleteinek működéséért kell felelősséget vállalnia. Ugyanezt a procedurális szemléletet követve nem lehetett elérni, hiszen az egyes eljárások hozzáérnék és tetszőlegesen módosíthatják az elérhető adatokat. OOP esetében viszont az egyes objektumok csak kéréseket továbbítanak a további objektumok felé, az már a hívott objektum felelőssége, hogy ezekre milyen változásokkal reagál.

De az ehhez kapcsolódó *adat elrejtés* a másik irányba is hasznos lehet. Általános szabálynak tekintető, hogy az egyes objektumok nem érik el egy másik objektum belső állapotát, még olvasásra sem. Ez azzal a közvetett előnyel is jár, hogy más objektum biztosan nem fog a meglévő objektum felépítésére támaszkodni, így az abban lévő implementációs változások nem befolyásolhatják más objektumok működését.

Ennek köszönhetően a *kód karbantartása* is jóval egyszerűbb, mivel az objektum belső állapotának megváltoztatása biztosan nem okozhat hibákat a program egyéb részein, mivel ezek egyébként se férhetnek hozzá ezekhez a belső adatokhoz. Egy esetleges hiba esetén is jól lokalizálható a hiba helye, hiszen a belső adatokat csak az objektum saját műveletei módosíthatják, ezért a hibát is ott kell keresni.

1.4.3. Osztályok

Egy OOP program futása során objektumok jönnek létre a memoriában és ezek kommunikálnak egymással. A kérdés már csak az, hogy miként jönnek létre ezek az objektumok? Az objektumok létrehozása általában nem közvetlenül, hanem közvetve, osztályok⁴ segítségével történik. Ennek az az oka, hogy sokszor egymáshoz hasonló objektumokra van szükségünk, ezeket felesleges lenne minden újra definiálni. Ehelyett célszerű létrehozni egy osztályt, ami leírja, hogy az abba az osztályba tartozó objektumok milyen jellegű adatokat fognak eltárolni, illetve ezeken milyen műveleteket tudnak elvégezni (tehát meghatározza a leendő objektumok állapotait és viselkedését). Ez tulajdonképpen megfelel egy objektum tervnek, ami alapján a későbbi objektum példányokat lehet létrehozni (példányosítani).

⁴class

Számla
- tulajdonos : szöveg # egyenleg : egész
+ Jóváír(összeg : egész ⁺) + Terhel(összeg : egész ⁺)

1.10. ábra. Egy **Számla** osztály meghatározása UML segítségével.

Az osztályok segítik a kód újrafelhasználását, hiszen egy egyszer elkészített jól működő osztály a későbbiekben bármikor használható különböző területeken. Az egységbezárásának köszönhetően egy osztály objektumai különösebb probléma nélkül használhatók több különböző rendszerben is.

Erre mutat egy példát az 1.10. ábra, ami egy **Számla** osztályt határoz meg.

Megjegyzés
Érdemes megjegyezni, hogy az objektummal ellentétben az osztály fogalom nem kötelező része az OOP módszertannak. Számos OOP programozási nyelv létezik (mint pl. a prototípus-alapú JavaScript), ahol nincs osztály fogalom. Az egyszerűség kedvéért mi azonban abban maradunk, hogy minden objektumnak egy adott osztályba kell tartoznia.

Az osztály alkotóelemeinél egy láthatósági szint határozható meg:

- privát: csak az osztály számára érhető el (jelölése: -);
- publikus: minden osztály számára elérhető (jelölése: +);
- védett: csak az adott osztály és leszármazottai számára érhető el (jelölése: #).

Mezők

Az 1.10. ábra felső részében láthatók az osztály által meghatározott *mezők*⁵. Ez azt jelenti, hogy minden példányosított **Számla** objektumnak lesz saját *tulajdonos* és *egenleg* belső változója.

Az adatelrejtés érdekében ezeket a mezőket célszerű megvédeni a külső hozzáféréstől. Nem az adatlo-pás miatt, hanem a kód robusztussága és karbantarthatósága érdekében. Az elrejtés persze a flexibilitást csökkenti, de ez az ár általában elfogadható.

Pszeudokódjainkban az összetett típusoknál látható *p.mező* formában fogunk hivatkozni a *p* referenciával hivatkozott objektum *mező* nevűmezőjére.

Metódusok

Az objektum az adatok és a rajtuk végezhető műveletek összessége. Ez az *egységbezárás* már az osztály szinten is megjelenik, hiszen már itt meg kell határozni a megadott osztályhoz tartozó objektumok belső működését.

Az egyes objektumok által végrehajtható műveleteket képviselik a metódusok⁶. Ezek alapvetően megfelelnek a procedurális paradigma eljárásainak, egy fontos különbséggel: a metódusok mind egy-egy objektumhoz tartoznak, tehát azok futtatása is ezen objektum feladata/felelőssége. Ez a gyakorlatban is jól látható, hiszen egy metódus kódjának végrehajtása közben le lehet kérdezni, hogy éppen melyik objektum adott metódusa fut.

Egy metódus tipikusan a hozzá tartozó objektum mezőivel dolgozik, esetleg meg tudja hívni más objektumok műveleteit.

Pszeudokódjainkban a *p.METÓDUS()* formában fogunk hivatkozni a *p* által aktuálisan mutatott objektum *METÓDUS* nevű műveletének meghívására.

⁵field

⁶method

Itt is fontos a *információrejtés* megjelenése. Egy objektumnak lehetnek olyan műveletei, amelyeket más objektum számára is elérhetővé akarunk tenni, de lehetnek olyanok is, amelyeket csak maga az objektum tud használni.

Tulajdonságok

A mezők a objektum belső állapotát tárolják, a metódusok a műveleteket képviselik, a tulajdonságok pedig az objektumok (gyakran kívülről is elérhető) adatait.

Első ránézésre így talán a mezőkhöz hasonlítanak jobban, technikai megvalósítás tekintetében valójában sokkal inkább metódusok. Míg a mezők közvetlenül értékeket tárolnak, addig a tulajdonságok valójában programkódok, amelyek valamilyen adatokat biztosítanak. Ezek értéke van, hogy közvetlenül egy mezőből származik, de lehetnek akár a mezők aktuális értékei alapján számított adatok is. Bizonyos programozási nyelvekben nincs is külön nyelvi elem erre a cérla, pl. a Javaban „getter” és „setter” metódusokkal lehet a tulajdonságokat megvalósítani.

Érdemes azonban használni őket, mivel:

- segítik az *egységezésről*, hiszen a csak olvasható tulajdonságon keresztül ki lehet olvasni adatokat az objektumból, anélkül, hogy azok módosíthatók legyenek;
- létre lehet hozni *számított mezőket*, tehát olyan adatokat, amelyek közvetlenül nincsenek eltárolva az objektumban, hanem a tulajdonság lekérdezésekor számítódnak (pl. ha egy *Ember* típusú objektum tárolja a születési évet, akkor felesleges eltárolni az életkort is egy mezőben: jobb választás lehet egy *Életkor* nevű tulajdonság, ami olvasáskor automatikusan kiszámolja az illető aktuális korát a születési év alapján);
- lehetőséget ad a *lusta kiértékelés* használatára, amikor az objektumhoz tartozó adat ugyan eltárolásra kerül egy mezőben, de annak értéke csak akkor számítódik ki (vagy kerül beolvasásra az adatbázisból), amikor az először kerül lekérdezésre.

Az UML diagramokon a tulajdonságokat a mezőkhöz hasonlóan jelöljük majd, a különbség abból látható majd, hogy külön attribútumok jelzik, hogy tulajdonságról van szó, illetve azt, hogy annak milyen elérési lehetőségei vannak:

- „{olvasható}” - csak olvasható,
- „{írható/olvasható}” - írható és olvasható.

Konstruktur, destruktur

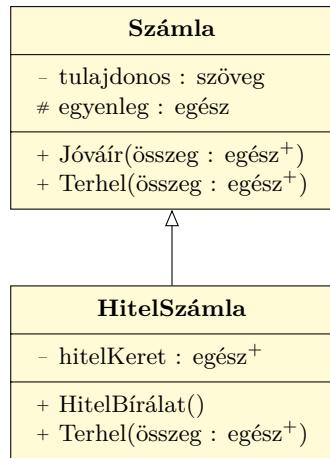
Az OOP számos védelmi lehetőséget tartalmaz arra, hogy egy objektum működése során ne kerülhessen hibás belső állapotba. Ennek viszont előfeltétele az, hogy minden objektum létrehozásának pillanatában már megfelelő állapotban legyen (innen már csak arra kell figyelni, hogy ne lehessen elrontani).

A konstruktorok szerepe egy új objektumpéldány létrehozása és inicializálása. Ez részben technikai lépésekkel jelent (memória foglalás, változók létrehozása, stb.), részben viszont az osztály készítője által megadott tetszőleges programkódot, amely alapján inicializálni tudja az objektumot: beállítja a változók kezdőértékét, létrehozza a további szükséges objektumokat, elvégzi a objektum működéséhez szükséges alapvető lépéseket.

A konstruktor abban különbözik egy metódustól, hogy ez csak egyszer fut le az objektum példányosításakor, viszont akkor kötelező módon. Tehát egy inicializáló metódussal ellentétben ezt biztos nem fogja a programozó elfelejteni meghívni. Emiatt ide érdemes elhelyezni minden olyan inicializáló kódöt, ami szükséges ahhoz, hogy csak megfelelő belső állapottal bíró objektumok jöhessenek létre. Ehhez néha külső paraméterek kellenek, ezeket mind a konstruktor paramétereként kell átadni a létrehozás pillanatában.

A destruktur szerepe hasonló, csak ez az objektum megszüntetéskor fut le. A modern, automatikus szemégtüjtő mechanizmussal bíró rendszerek esetében ez számos elvi kérdést vet fel, hiszen nem tudható előre, hogy mikor lesz egy objektum ténylegesen megszüntetve.

A pszeudokódokban ezért külön nem is jelezzük a konstruktor és destruktur hívásokat, feltételezzük, hogy azok minden objektum létrehozásakor és megszüntetéskor lefutnak.



1.11. ábra. Egy ***HitelSzámla*** leszármazott osztály meghatározása UML segítségével.

1.4.4. Öröklés

Az osztályoknak fontos szerepe volt abban, hogy az egymáshoz hasonló objektumokat egyszerűbben tudjuk összefogni. Az öröklés⁷ feladata pedig az, hogy az egymáshoz hasonló osztályokat tudjuk vele hatékonyabban kezelni.

Technikailag az öröklés egyszerűen megérthető: a leszármazott osztály öröklí az ős osztály minden elemét (mezők, metódusok, tulajdonságok). A leszármazott a beállított hozzáférési szintektől függően hozzá is férhet ezekhez.

Erre mutat egy példát az 1.11. ábra, ahol a ***HitelSzámla*** a ***Számla*** osztály egy leszármazottja. Ez azt jelenti, hogy a leszármazottnak is van *tulajdonos*, *egyenleg* mezője, illetve öröklí az ős műveleteit is.

Ennek egyik nyilvánvaló előnye a *kódújrafelhasználás* terén látható: csökkenti a kód redundanciát, javítja a meglévő kódok rugalmas felhasználhatóságát.

Az öröklés során lehetőség van a *kiterjesztésre* is, mivel a leszármazott nem csak megkapja az őse minden jellemzőjét, de lehetősége van azokat kiegészíteni. Ennek megfelelően felvehet új mezőket, új metódusokat és új tulajdonságokat. Ezek persze egymással szorosan összefüggnek, ha megjelentek új mezők a leszármazottban, ez tipikusan magával hordozza a műveletek módosítását is (különben mi értelmük lenne az új mezőknek?). Ez látható az 1.11. ábrán is, ahol megjelent egy új *hitelKeret* mező, illetve az ehhez kapcsolódó *HITELBÍRÁLAT* metódus.

Az öröklés során nem csak kiegészíteni lehet az ősosztályt, de az örökölt metódusok módosítása is engedélyezett. Ez az úgynevezett *metódus felülírás*⁸, amikor a leszármazott új implementációt ad az örökölt metódusnak. Ez lehet az örökölt metódus teljes cseréje, vagy annak csak kibővítése. Ez utóbbi technikailag azon alapul, hogy van lehetőség az eredeti örökölt metódus meghívására az új változatból. Az 1.11. ábra erre mutat példát, amikor a *TERHEL* metódus újra megjelenik a leszármazottban is. Ez azt jelenti, hogy a ***HitelSzámla*** esetében a terhelés másként működik, hiszen itt már figyelembe kell venni a hitelkeret értékét is.

Egy leszármazott osztály is lehet őse további osztályoknak, így öröklési láncok is létrehozhatók. Ez is jelentősen segíti a kód *olvashatóságát*, mivel az osztályokat egy logikus hierarchiába lehet rendezni.

Programozási nyelv függő, hogy engedélyezett-e a többszörös öröklődés, amikor egy leszármazottnak több őse is lehet. Ez számos elvi és technikai problémát felvet (névütközések kezelése, egy ős több ágon is megjelenhet a felmenők között, stb.), ezért sok napjainkban releváns nyelv (Java, C#) nem támogatja, bár számos ellenpélda is létezik (C++, Python). Ezen jegyzet keretein belül mi ezt a lehetőséget nem fogjuk használni.

⁷inheritance

⁸overriding, ami nem összekeverendő az overloading fogalommal

1.4.5. Polimorfizmus

Polimorfizmus öröklésen keresztül

A polimorfizmus szó „többalakúságot” jelent. Maga a koncepció a programozáson belül általában azt jelenti, hogy egy név több jelentéssel is bír, amelyek közül a tényleges jelentést a használat kontextusa határozza meg. Például az összeadás (+) operátor más-más műveletet hajt végre attól függően, hogy számokat (matematikai összeadás a szám típusának megfelelően) vagy szöveget aduk vele össze (szövegek összefűzése).

Egészen más módon, de az előzőleg tárgyalt öröklés esetében is felmerül a polimorfizmus lehetősége. Ehhez elsőként azt kell belátnunk, hogy bármelyik objektumot lehet az ő bármelyik őseként kezelni. Erre néhány példa:

- egy **Számla** típusú referenciával lehet hivatkozni egy **HitelSzámla** típusú objektumra;
- ha egy függvény **Számla** típusú objektumot vár paraméterként, akkor annak **HitelSzámla** objektum is átadható.

Elsőre ez furcsa lehet, hiszen a szigorúan típusos nyelvek esetében alapvetőnek tűnik, hogy egy érték minden ugyanolyan típusú, mint a rá hivatkozó referencia. Az újszerű működés oka azonban magából az öröklés fogalmából levezethető: a **HitelSzámla** az eredeti **Számla** minden mezőjét és metódusát örökölte, tehát biztos, hogy minden olyan műveletet, amit egy **Számla** tudott volna kezelni, azt a **HitelSzámla** is fog tudni. Tehát a **HitelSzámla** minden olyan helyen használható, ahol az eredeti **Számla**. Ebbe még a szigorúan típusos nyelvek se tudnak belekötni, ezért engedélyezni fogják az ilyen jellegű értékkadásokat/paraméterátadásokat.

A polimorfizmus ott jelenik meg, ha egy metódus paraméterként egy **Számla** objektumot vár, majd működése során meghívja annak **TERHEL** metódusát. Elképzelhető ugyanis, hogy paraméterként egy **HitelSzámla** objektum fog érkezni (az előbb beláttuk, hogy ezzel nincs gond), ilyenkor pedig ennek a **TERHEL** metódusa fog lefutni, ami viszont máshogy működik, mint amire eredetileg számítani lehetett. Tehát ugyanaz a metódushívás, ugyanazokkal a paraméterekkel más-más hatással járhat, ez a polimorfizmus.

Megjegyzés

A jegyzetnek nem célja néhány oldalon belül részletesen tárgyalni az OOP összes sajátosságát, ezért most nem foglalkozunk a késői és korai kötés lehetőségeivel, azt feltételezzük, hogy minden késői kötés történik

Ez egy apró technikai részletnek tűnik, de valójában óriási lehetőségeket ad, ez tekinthető az OOP programozás koronaékszerének [3]. Ez lehetőséget ad arra, hogy a későbbiekben bárki bármilyen **Számla** leszármazottat készíthessen, ezeket pedig minden át lehet adni az előbb említett metódusnak, és minden a saját **TERHEL** metódusának megfelelően fog működni. Ez egy nagyon rugalmas rendszert biztosít, ahol bármelyik objektum lecserélhető egy másiképp működő leszármazottjára.

Ez ismét jelentősen növeli a kód *olvashatóságát*, hiszen a polimorfizmus lehetőséget ad arra, hogy bizonyos metódusok különféle megvalósításokat kapjanak. A fenti példánál maradva, nem a számlát használó metódusban kell ellenőrizni, hogy a paraméterként kapott számla milyen típusú valójában (és ennek megfelelően kezelní a terhelését), hanem elég meghívni annak a szükséges metódusát, és ő olyan módon fog erre reagálni, ami a típusából következik.

Ez a laza kapcsolat az *absztrakciós szintet* is növeli, hiszen már nem csak az implementációs részletek maradnak rejtve, hanem a használt objektum tényleges típusa is. Ez akár zavaró is lehet persze, de a gyakorlatban egyértelműen bebizonyosodott, hogy ez egy jól használható programtervezési irány.

A polimorfizmus sokat segít a program *bővíthetőségeiben* is. Hiszen a későbbiekben lehetőség van új leszármazott osztályok létrehozásával új viselkedések létrehozására anélkül, hogy módosítanánk a már meglévő kódokat.

Interfész és implementáció elválasztása

Már korábban is felmerült, hogy az absztrakciós szint növelése során megjelenik az interfész és az implementáció elválasztásának kérdése. Érdemes tisztázni ezek jelentését a szoftverfejlesztés kontextusában:

- Az interfész azt határozza meg, hogy egy objektum milyen feladatokra használható. Milyen adatokat lehet belőle kiolvasni/módosítani, illetve milyen műveleteket lehet rajta végrehajtani. Tehát az interfész alapvetően a „mit” kérdésre ad választ, arra, hogy egy objektum mire használható.
- Implementáció pedig azt határozza meg, hogy az előzőleg megadott követelményeket miként valósítja meg egy objektum. Egy művelet esetében ez már a konkrét programkódot jelenti. De a mezők esetében is számos implementációs lehetőség merül fel (pl. minden adatszerkezetben tároljuk az adatokat). Az implementáció tehát arra ad választ, hogy egy objektum „hogyan” teljesíti az interfészben megadottakat.

A már egyszer felmerült példánál maradva: egy *Ember* objektum interfésze alapján látható, hogy az rendelkezik *Életkor* tulajdonsággal. Tehát ez a tulajdonság elérhető, ezen keresztül le lehet kérdezni az életkort, az objektum külső használójának elég ennyit tudnia. Az viszont már implementációs kérdés, hogy az életkor saját mezőben tárolódik, vagy a lekérdezés pillanatában számítódik a születési dátum alapján, vagy akár egy adatbázisból töltődik be. Ezek már implementációs részletek, amelyek az objektum megvalósításához mindenképpen szükségesek, ugyanakkor annak használatát nem befolyásolják.

A két jellemző persze szorosan összefügg, nincs értelme egy interfésznek, ha nem tartozik hozzá implementáció, és persze egy implementáció is haszontalan, ha közvetlenül vagy közvetve nem érhető el, hogy azt használni lehessen.

Érdemes azonban a kettőt külön kezelni, hiszen más a szerepük a szoftverfejlesztés folyamatában. Egy magasabb absztrakciós tervezési szinten elég az interfésekkel foglalkozni. Ilyen interfésekkel létre lehet hozni anélkül, hogy meg kellene adni a pontos implementációt, hiszen ezek célja csak annak jelzése, hogy bizonyos részproblémák megoldására milyen műveletekkel bíró objektumokat lehet majd felhasználni (pl. egy ügyfélkapcsolatokat kezelő rendszer üzenetküldő objektumának kell, hogy legyen egy *ÜZENETKÜLDÉS* nevű metódusa). A tervezést, de még az implementáció egy részét is el lehet készíteni csak ezen interfések segítségével (pl. a program egy pontján az interfésszel jelölhető, hogy ott egy olyan objektumot kell majd használni, ami megfelel az előbb említett követelménynek).

Az implementációs kérdésekkel már csak alacsonyabb szinten kell foglalkozni. Maradva az előző példánál, lehet készíteni olyan üzenetküldő objektumokat, amelyek rendelkeznek a szükséges *ÜZENETKÜLDÉS* műveettel, ezek lehetnek akár különböző típusúak is, csak teljesíteniük kell ezt a követelményt. Ezek az objektumok csereszabatosan használhatók azokon a helyeken, ahol ilyen típusú interfész megkötés jelent meg.

Absztrakt osztályok

Az *absztrakt osztályok*⁹ olyan osztályok, amelyek rendelkezhetnek ugyan mezőkkel és tényleges metódusokkal, de ezek mellett lehetnek olyan műveleteik is, amelyekhez nem tartozik implementáció. Ebből adódik, hogy ezekből az osztályokból objektum példányokat nem lehet létrehozni, hiszen néhány metódusuk egyszerűen nincs megírva.

Ezek a hiányzó metódusok persze nem valamilyen hiba folytán jönnek létre, ezek tudatos tervezés eredményei. Gyakori, hogy egy osztályhierarchia építése során olyan ősosztályok jönnek létre, amelyek számos alattuk lévő leszármazott közös mezőit és műveleteit össze tudják fogni, ezen műveletek egy része pedig már ebben az ősbén megírható. Azonban lehetnek olyan műveletek, amelyekről tudható, hogy minden leszármazottban kell majd, hogy legyenek, viszont az ősbén ehhez még nem tudunk implementációt adni, ezt majd a leszármazottaknak kell megtenniük.

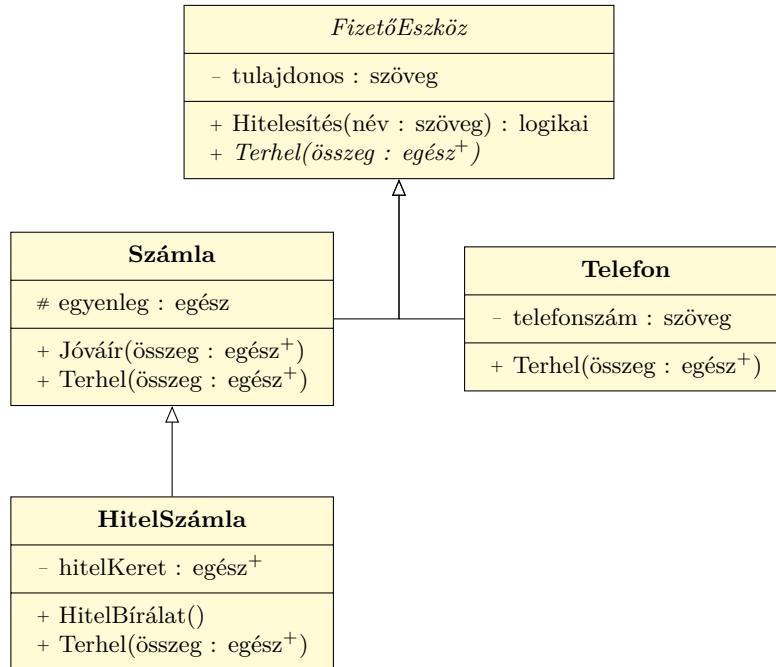
Erre mutat példát az 1.12. ábra. Megjelent egy új *FizetőEszköz* osztály, amibe átkerült a *tulajdonos* mező, és megjelent egy ehhez kapcsolódó *Hitelesítés* metódus. Ez utóbbi csak arra szeretne példa lenni, hogy az absztrakt osztálynak is lehetnek mezői és megvalósított metódusai, például a hitelesítés akkor ad vissza *igaz* értéket, ha a paraméterként átadott név megegyezik a privát mezőben tárolt tulajdonos névvel.

Az osztálynak van több leszármazottja is (*Számla* és *Telefon* osztályok), amelyek mind fizetőeszközök és rendelkeznek *TERHEL* metódussal. A polimorfizmus miatt ezt a metódust már a *FizetőEszköz* szintjén is célszerű lenne megadni, azonban itt még nem tudunk hozzá implementációt rendelni.

Ezek az úgynevezett *absztrakt metódusok*, amelyeknél csak a metódus szignatúrát (név, paraméterek, visszatérési érték) adjuk meg, de implementációt nem tudunk hozzá rendelni. Emiatt az egész osztály absztrakt lesz, tehát nem példányosítható, hiszen van legalább egy nem implementált művelete (ami

⁹abstract class

egyébként megfelel a való világbeli fogalomnak, hiszen a „fizetőeszköz” mint olyan, valóban csak egy absztrakt fogalom, nem létezik fizikailag).



1.12. ábra. Egy **FizetőEszköz** absztrakt osztály bevezetése.

Egy absztrakt osztály nem példányosítható, de ettől függetlenül ilyen típusú referenciákat létre lehet hozni. Például egy fizetést végző függvény (1.5. algoritmus) várhat paraméterként egy **FizetőEszköz** típusú objektumot, majd annak meghívhatja a **TERHEL** metódusát. **FizetőEszköz** típusú objektum nem létezhet, hiszen az az osztály absztrakt, viszont most már tudjuk, hogy annak leszármazottai is átadhatók paraméterül. Tehát a példánál maradva, csak **Számla**, **HitelSzámla** vagy **Telefon** objektumok érkezhetnek paraméterként, ezeknek pedig mindenek van implementált **TERHEL** metódusuk.

1.5. Algoritmus Absztrakt osztály használata.

Bemenet: ezzel - **FizetőEszköz** (ezzel kell fizetni)

Bemenet: ennyit - **egész⁺** (ennyit kell fizetni)

1: **függvény** **FIZETÉS**(ezzel, ennyit)

2: ezzel.**TERHEL**(ennyit)

3: **függvény vége**

Ez az a pont, ahol az osztály funkcionális leírása (tehát az interfész) már teljesen egyértelműen elválik a tényleges implementációtól. A fizetést író programozót nem érdekli, hogy miként fog működni a terhelés, neki csak az a fontos, hogy a paraméterként kapott objektumnak legyen ilyen művelete. A fenti megoldás pedig ezt típusbiztosan, elegánsan megoldotta.

Interfész nyelvi elem

Az absztrakt osztálynál már látható volt, hogy elvált egymástól az interfész és az implementáció. Egy további lépéssel eljutunk arra a pontra, amikor az implementáció már semmilyen formában nem jelenik meg, csak interfésekéről beszélünk. Ezek az interfések tehát csak műveleteket és tulajdonságokat határoznak meg, de azok implementációja már teljesen hiányzik.

Különböző OOP nyelvek különféle interfész megvalósításokra adnak lehetőséget. Van, hogy maga az interfész egy külön nyelvi elemként jelenik meg, van ahol az absztrakt osztályokhoz hasonló módon lehet ezt a funkciót elérni. Valójában nem maga a technikai megvalósítás a lényeg, hanem az újszerű tervezési koncepció, amit ezzel az eszközzel el lehet érni.

Mivel a legelterjedtebb OOP nyelvek (Java/C#) ez előbbi módon működnek, ezért mi is ezzel fogalkozunk. A későbbiekben, ha csak „interfészről” beszélünk, akkor minden erre a saját nyelvi elemre gondolunk.

Az interfész nyelvi elem felfogható úgy, hogy az egy szerződést/követelményrendszer jelent. Ez a szerződés azt határozza meg, hogy az interfész „megvalósító” osztályoknak milyen műveletekkel kell rendelkezniük:

- ez programozási nyelv szintjén metódus szignatúrákat jelent, tehát az interfész felsorolja, hogy milyen nevű/paraméterezettségű/visszatérési értékű metódusokra van szükség a megvalósításához;
- az interfész előírhat tulajdonságokat is (hiszen azok nagyon hasonlítanak a metódusokhoz);
- viszont az interfész semmilyen információt nem tartalmazhat az implementációval kapcsolatban (tehát nincsenek megírt metódusai), sem az objektum mezőivel és belső állapotával kapcsolatban (tehát nincsenek mezői).

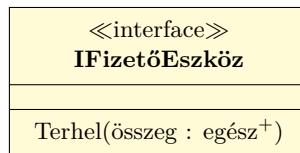
Az interfész tehát nem téveszthető össze az osztályokkal, hiszen ez nem tartalmaz mezőket, sem metódus megvalósításokat, és így persze objektumot se lehet belőle létrehozni.

Az interfések azonban két esetben is jól használhatók, amelyek segítségével a technika biztosítja a laza kommunikációs kapcsolatot minden oldal részére:

- Egy osztály megvalósíthat (implementálhat) egy interfészt: ez azt jelenti, hogy az osztály köteles megvalósítani az összes, az interfész által előírt követelményt (tehát legyenek az interfésekben meghatározott metódusai és tulajdonságai).
- Habár interfésből objektumot nem lehet létrehozni, interfész típusú referenciát viszont igen. Ezzel lehet hivatkozni bármelyik objektumra, amelyik teljesíti az adott interfész követelményeit, és ezen keresztül lehet hozzáérni az interfésekben meghatározott műveletekhez és tulajdonságokhoz.

Összefoglalva: az interfész egy képességet szimbolizál, egy szerződést a felek között. Az interfész implementáló objektum vállalja, hogy teljesíti az így előírt követelményeket. Az interfész referenciaiként használó oldal pedig nyugodtan bízhat benne, hogy a referencia csak olyan objektumokra hivatkozhat, amelyek teljesítik a szerződést.

Az 1.13. ábra egy példát mutat az interfész nyelvi elem UML jelölésére. Az így készített interfész majdnem ugyanaz, mint a már megismert absztrakt osztály változat (1.12. ábra), fontos különbség azonban, hogy ennek nem lehetnek mezői, ezért nincs *tulajdonos* mezője sem.



1.13. ábra. Az összehasonlíthatóságot jelző interfész

Ezzel eljutottunk az *absztrakció következő szintjére*: az interfész már teljesen elrejti a mögöttes komplex kódokat. Csak annyit mond, hogy az általa hivatkozott objektum valamelyen műveletekkel rendelkezik, de semmilyen információt nem tartalmaz ennek implementációjáról. Ezzel teljesen szét lehet választani a funkciókat és azok megoldásait. Az 1.6. algoritmus is ezt mutatja: semmi se lehet tudni a leendő fizetést végző objektumról, csak annyit, hogy megvalósítja a ***IFizetőEszköz*** interfészt, emiatt biztosan van neki **TERHEL** nevű metódusa. De ezen a szinten ez a tudás pont elegendő.

1.6. Algoritmus Interfész használata.

Bemenet: *ezzel - IFizetőEszköz (ezzel kell fizetni)*

Bemenet: *ennyit - egész⁺ (ennyit kell fizetni)*

1: **függvény FIZETÉS(ezzel, ennyit)**

2: *ezzel.TERHEL(ennyit)*

3: **függvény vége**

Vegyük észre, hogy az interfészen keresztül is eljutottunk a *polimorfizmusig*. Ha egy **I** interfész több objektum is megvalósít, akkor egy **I** típusú referenciaival ezen objektumok bármelyikére lehet hivatkozni. Az interfészen definiált metódusok meghívásakor pedig minden a hivatkozott objektum megfelelő metódusa fog lefutni.

Míg az öröklésnél az egyszeres öröklést preferáltuk, addig az interfészknél nincs ilyen megszorítás. Egy osztály megvalósíthat tetszőleges számú interfész, ezzel elérhető a *többszörös polimorfizmus* a többszörös öröklés negatív következményei nélkül.

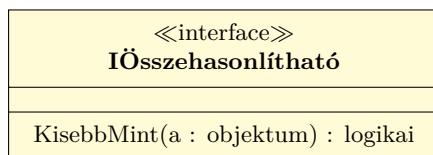
Példa - Rendezés interfészekkel

Az interfészek használatára egy nagyon szemléletes példa a rendezhetőség kezelése. Számos rendező algoritmus létezik, közöttük egy fontos csoportot képviselnek az úgynevezett „összehasonlító rendezések”, amelybe azok a rendező algoritmusok tartoznak, amelyek csak egyetlen követelményt támasztanak a rendezendő elemekre: el kell tudni két elem közül dönten, hogy az egyik kisebb-e mint a másik (ami egy rendezés esetében elég nyilvánvaló követelménynek tűnik). A legtöbb gyakran használt rendezés egyébként ide tartozik (buborékrendezés, quicksort, stb.).

Az algoritmus tervezésekor érdemes észrevenni azt a tényt, hogy valójában mindegy, hogy számokat, objektumokat, szövegeket rendezünk, maga az algoritmus ugyanaz lesz, csak annyi a követelmény, hogy el kell tudni dönten, hogy az egyik kisebb-e mint a másik. Technikailag ez úgy is megfogalmazható, hogy a rendezendő objektumoknak kell, hogy legyen egy **KISEBBMINT** nevű metódusuk, ami el tudja majd dönten ezt a kérdést (itt pl. megjelenik a felelősségek delegálásának technikája, hiszen nem célszerű és nem is kivitelezhető a rendező algoritmuson belül implementálni az összes létező összehasonlítható objektum összehasonlítási szabályát, hanem ezt a döntést jobb rábízni magára az objektumra).

Ezt a kikötést a legegyszerűbben egy **IÖsszehasonlítható** nevű interfész segítségével lehet megadni, amely meghatároz egy szerződést: mindenki, aki implementálja ezt az interfész, biztosan rendelkezni fog egy **KISEBBMINT** nevű metódussal. A rendező algoritmus pedig garantálni fogja, hogy ezeket a fajta objektumokat fogja tudni rendezni függetlenül attól, hogy miként is működik maga az összehasonlítás.

Az UML jelöléseknek megfelelően az 1.14. ábra mutatja ezt az interfész definíciót.



1.14. ábra. Az összehasonlíthatóságot jelző interfész.

Hogy a konkrét programozási nyelvben ezt miként oldható meg, az lényegtelen. Absztrakt osztályokkal is lehet, de ha van külön interfész nyelvi elem, akkor azzal célszerű.

A következő lépésként el lehet készíteni magát a rendezés algoritmusát, ami láthatóan nem foglalkozik azzal, hogy a rendezett elemek pontosan milyen típusúak lesznek. Csak az interfész által adott szerződésre támaszkodik, ugyanis előírja, hogy a rendezendő elemeknek meg kell valósítaniuk az **IÖsszehasonlítható** interfész, emiatt pedig biztosan lesz nekik **KISEBBMINT** metódusuk. Elképzelhető, hogy lesz nekik sok más mezőjük és metódusuk is, de a rendezés szempontjából ez teljesen érdektelen.

1.7. Algoritmus Buborékrendezés interfészkekkel.

Bemenet: A - tömb<IÖsszehasonlítható> (*rendezendő tömb*)

Kimenet: A - tömb<IÖsszehasonlítható> (*rendezett tömb*)

```
1: függvény BUBORÉKRENDEZÉS(A)
2:   ciklus  $i \leftarrow A.méret$ -tól 2-ig
3:     ciklus  $j \leftarrow 1$ -től ( $i - 1$ )-ig
4:       ha  $A[j + 1].KISEBBMINT(A[j])$  akkor
5:          $A[j] \leftrightarrow A[j + 1]$ 
6:       elágazás vége
7:     ciklus vége
8:   ciklus vége
9:   vissza  $A$ 
10:  függvény vége
```

1.4.6. Értékelés

Az eddigiek alapján az OOP előnyei nyilvánvalók: magas absztrakciós szint elérésének lehetősége, jól karbantartatható program, polimorfizmus, stb. A paradigma nagyon jól működik olyan esetekben, ahol több programozónak kell együtt dolgozni, hiszen az egyes modulokat, osztályokat egymástól függetlenül is lehet implementálni.

Fontos azonban, hogy megfelelő módon legyen megalkotva az osztály hierarchia, a hibás tervezés egy utólag nagyon nehezen javítható kódot eredményezhet. Nem igazi hátrány, de megjegyzendő, hogy nagyon kicsi (tipikusan iskolai) feladatok esetében az egész OOP kontraproduktív lehet. De ha valaki találkozik egy megfelelő méretű problémával, akkor azonnal látni fogja annak előnyeit.

Érdemes megjegyezni, hogy bizonyos (rész)feladatokra létezhet hatékonyabb, jobb megoldás, pl. funkcionális formában. Ez utóbbita megoldást jelent, hogy az elterjedt OOP nyelveket az utóbbi években kiegészítették funkcionális jellemzőkkel.

1.5. Generikusok

Típusos nyelvek korlátai

A változók használatánál tisztáztuk, hogy mindegyiknek kell, hogy legyen egy típusa, ami meghatározza annak eltárolási módját, illetve a rajta végrehajtható műveleteket. Ezek a különféle típusok minden programozási nyelvben megjelennek, abban azonban jelentősen különböznek, hogy a típusok meghatározását mennyire automatizálják.

- Szigorúan típusos nyelvek esetében az egyes változók típusa mindenkorán megadott, az nem változhat meg futás közben. Ezt a típust általában a programozónak kell meghatározni, bár elképzelhető, hogy a kontextusból adódik, és ilyenkor a fordító kitalálja magától. A fordító szigorúan ellenőrzi (statikus típusellenőrzés), hogy megadott típusú változóval csak azon a típuson értelmezhető műveleteket lehessen végrehajtani, és ugyanígy nem engedélyezi a tágabb típusokról (pl. valós számok) a szűkebb típusokra (pl. egész számok) való automatikus konverziót, ehhez explicit típusátalakítást kell.
 - Gyengén típusos nyelvek esetében ezek a szabályok kevésbé kötöttek, a program futás közben az értékadások alapján dönti el a változók típusát. Ez egyszerűbb/gyorsabb programkód íráshoz vezet, habár a szigorú típusellenőrzés hiánya miatt növeli a futásidejű hibák kockázatát.

Mindkét megoldásnak megvan a maga előnye és hátránya, ezért itt nem is szeretnénk állást foglalni, hogy melyik a jobb.¹⁰

A szigorú típusellenőrzés jelentősen csökkenti a potenciális hibákat, néha azonban gyakorlati kellemtelenségeket okoz. Az 1.8. algoritmus mutat erre egy példát, ahol látható egy függvényt, ami egy paraméterként átadott egész számokat tartalmazó tömbből és egy második paraméterként átadott egész számból megmondja, hogy az adott szám megtalálható-e a tömbben.

1.8. Algoritmus Egyszerű függvény egész szám paraméterekkel.

Bemenet: A - tömb<egész> (a számokat tartalmazó tömb)

Bemenet: x - egész (a keresett érték)

Kimenet: . - logikai (*benne van-e x az A tömbben?*)

- ```

1: függvény BENNEVAN(A , b)
2: i \leftarrow 1
3: ciklus amíg $i \leq A.méret \wedge A[i] \neq b$
4: i \leftarrow i + 1
5: ciklus vége
6: vissza $i \leq A.méret$
7: függvény vége

```

A program természetesen tökéletesen működik, de jól látható, hogy csak egész számokra. Amennyiben szükség lenne olyan változatokra is, amelyek működnek valós számokkal, vagy akár szövegekkel, akkor nem tudjuk felhasználni a már meglévő kódot, hanem egészen új függvényeket kell írni. Külön bosszantó, hogy ezek a függvények csak a bemenő paraméterek típusában különböznek a meglévőtől, maga az előírt algoritmusa teljesen ugyanaz minden esetben.

A probléma általánosítható az OOP nyelvek osztályaira is. Ha már van egy rendező algoritmus, ami rendezni tud minden **Ember** típusú objektumot, akkor ugyanaz nem használható a **Kutya** típusú objektumokra. Csak a típus különbség miatt újra kell írni az egész rendezést, pedig az algoritmus pontosan ugyanaz marad.

## Típusparaméterek

A fenti probléma megoldását a típusparaméterek adják. A modern szigorúan típusos programozási nyelvek lehetővé teszik, hogy változókat, függvényeket, osztályokat hozzunk létre úgynevezett típusparaméterekkel. Ez azt jelenti, hogy a már meglévő konkrét típusok helyett a program bármely részében

<sup>10</sup>Bár egy magára adó programozó csak szigorúan típusos nyelvekkel foglalkozik.

használhatók ezek a típusparaméterek, amelyek a változónevekhez hasonlók, azonban nem adatot, hanem típust képviselnek.

A fenti algoritmust megírhatjuk az alábbi formában:

---

### 1.9. Algoritmus Egyeszerű függvény T típusparaméterrel.

---

**Bemenet:**  $A$  - tömb $<\mathbf{T}>$  (egy  $\mathbf{T}$  típusú elemeket tartalmazó tömb)

**Bemenet:**  $x$  -  $\mathbf{T}$  (a keresett  $\mathbf{T}$  típusú érték)

**Kimenet:** . - logikai (benne van-e  $x$  az  $A$  tömbben?)

- 1: **függvény** BENNEVAN( $A$  ,  $b$ )
  - 2:      $i \leftarrow 1$
  - 3:     **ciklus amíg**  $i \leq A.méret \wedge A[i] \neq b$
  - 4:          $i \leftarrow i + 1$
  - 5:     **ciklus vége**
  - 6:     **vissza**  $i \leq A.méret$
  - 7: **függvény vége**
- 

Mint látható, az eddigi konkrét típus (*egész*) ki lett cserélve egy típusparamétere ( $\mathbf{T}$ ). A pszeudokódóból talán nem látható egyértelműen, de a  $\mathbf{T}$  tehát nem egy valódi típus neve, hanem csak egy paraméter, ami azt jelzi, hogy a függvény megírásakor még nem ismert az  $A$  tömb és a  $b$  változó pontos típusa, az majd csak futásidőben fog kiderülni. De mint jól látható, a programkód megírásában ez nem okoz problémát, hiszen tényleg mindegy, hogy a  $\mathbf{T}$  később egész szám lesz, vagy akár egy objektum. A keresési algoritmus minden esetben ugyanaz.

Típusparamétere lehet egy függvénynek (ezek a „*generikus függvények*”), ilyenkor tipikusan a megíváskor, az átadott paraméterek alapján dől el a  $\mathbf{T}$  valódi értéke. De egy osztálynak vagy interfésznek is lehet típusparamétere („*generikus osztályok és interfések*”), ilyenkor az objektum példányosításakor kell megadni, hogy az adott típusparaméter milyen típust képvisel.

Előfordulhat, hogy több típusparaméterre is szükség van, ilyenkor a már meglévő  $\mathbf{T}$  mellett bevezethetők más típusparaméterek is (pl.  $\mathbf{K}$ ). Ezek azt jelölik, hogy ez a két típus egymástól teljesen független. Értékük lehet ugyanaz a típus is, de akár különböző is.

#### Implementáció

A szigorúan típusos programozási nyelvek több évtizedes történetében újdonságnak számítanak a típusparaméterek, a Java és C# nyelvek is csak bizonyos verziószám felett engedélyezik. C++ esetében már régóta léteznek az úgynevezett template-k, amelyek egy még rugalmasabb rendszert biztosítanak, és bár működésük más jellegű, de mégis egészen hasonló működést lehet velük elérni. A gyengén típusos nyelvek esetében az egész koncepcióra nincs szükség, hiszen ott a fordító nem ragaszkodik a fordításidőjű szigorú ellenőrzéshez.

Generikus programozás alatt azt a módszert értjük, amikor a programkód még nem tartalmazza a konkrét típusokat, hanem azok majd csak később kerülnek megállapításra. Ennek célja a minél *magasabb absztrakciós szintű* és ezzel minél általánosabb programkódok elkészítése. A generikusok nagyon gyakran jelenniek meg a különféle adatszerkezeteknél, hiszen olyan tároló eszközököt szeretnénk készíteni, amelyek tetszőleges típust tudnak kezelni.

A pszeudokódjainknál ez nem fog nagy újdonságot okozni, hiszen eddig is nyugodtan írhattunk bárhol a  $\mathbf{T}$  típust, ami azt jelentette, hogy „valamilyen típus”. A későbbiekben viszont arra fogunk ügyelni, hogy ezt a technikát a konkrét implementációban is megvalósítsuk, hiszen van rá lehetőség.

### Megszorítások

A típusparaméterek nagy rugalmasságát adnak, hiszen ezt követően nem kell minden meghatározni a pontos típusokat, lehet írni olyan, általánosan használható algoritmusokat, amelyek tetszőleges típussal működnek. Az új nyelvi lehetőséget azonban úgy terveztek meg, hogy ne veszélyeztesse a szigorúan típusos nyelvek által garantált biztonságot.

Ez utóbbi jelentős korlátokat jelent, hiszen így egy  $\mathbf{T}$  típusú változónak csak egy másik  $\mathbf{T}$  típusú változó értéke adható értékül. A műveletekre hasonló korlátok vannak, egy  $\mathbf{T}$  típusú változón csak olyan műveleteket lehet végrehajtani, amelyek biztosan értelmezhetők a  $\mathbf{T}$  által képviselt típuson.

Mivel fordításkor még nem tudható, hogy futásidőben milyen típusnak fog megfelelni a  $\mathbf{T}$  típusparaméter, ezért a fordító csak olyan műveleteket engedélyez vele, amelyekkel minden típus bír. Ilyen művelet nincs sok: értékadás, egyenlőség ellenőrzése (nem véletlen, hogy a fenti példában pont ezeket használtuk csak).

Emiatt ebben a formában nem lehet egy generikus buborékrendezést se készíteni, hiszen annak alapfeltétele volt, hogy csak olyan adatokat lehet rendezni, amelyeken értelmezhető egy „kisebb” reláció. Ez viszont nem minden típusra igaz, ezért a fenti példában a két  $\mathbf{T}$  típusú  $A$ -beli elemen a fordító nem engedné az  $A[i+1] < A[i]$  összehasonlítást.

A problémára a megoldást a „megszorítások”<sup>11</sup> adják. Ezekkel el lehet érni, hogy egy típusparaméter csak bizonyos feltételeknek megfelelő típust tudjon képviselni. Ilyen feltétel lehet az, hogy legyen „összehasonlítható”, tehát minden  $\mathbf{T}$  típusú  $x$ ,  $y$  elemen értelmezhető az  $x < y$  reláció.

A fordító ezt követően engedi fogja, hogy a  $\mathbf{T}$  típusú változókkal összehasonlításokat végezzünk, hiszen rögzítettük, hogy a  $\mathbf{T}$  biztos ilyen típust fog képviselni. Ugyanígy a fordító feladata lesz ellenőrizni, hogy a függvény hívásakor vagy objektum létrehozásakor a  $\mathbf{T}$  csak olyan típust kaphasson, ami teljesíti a kitűzött feltételeket. Ezzel teljesítettük a szigorúan típusos nyelvek által elvárt biztonsági követelményeket, de mégis egy meglehetősen rugalmasan használható típusparamétereit kaptunk.

## 1.10. Algoritmus Buborékrendezés típusparaméterekkel.

**Bemenet:**  $A$  - tömb< $\mathbf{T}$ > (rendezendő tömb)

**Kimenet:**  $A$  - tömb< $\mathbf{T}$ > (rendezett tömb)

**Megszorítás:**  $\mathbf{T}$  összehasonlítható

```

1: függvény BUBORÉKRENDEZÉS(A)
2: ciklus $i \leftarrow A.méret$ -től 2-ig
3: ciklus $j \leftarrow 1$ -től ($i - 1$)-ig
4: ha $A[j + 1] < A[j]$ akkor
5: $A[j] \leftrightarrow A[j + 1]$
6: elágazás vége
7: ciklus vége
8: ciklus vége
9: vissza A
10: függvény vége

```

### Implementáció

A pszeudódokban szabadon használhatjuk a különféle típusparamétereket, illetve megszorításokat, a konkrét programozási nyelvek esetében azonban ezekre szigorú szabályok vonatkoznak. Bizonyos nyelvekben típusparaméterként csak osztály típust lehet megadni (tehát egyszerű típusokat nem). A megszorítások pedig általában valamelyen egyszerű programozástechnikai korlátot jelentenek, pl. paraméter nélküli konstruktor megléte, megadott osztálytól való származás, megadott interfész megvalósítása. Ez utóbbi azonban tökéletesen megfelel az igényeinknek, hiszen ezzel ki tudjuk kényszeríteni bizonyos műveletek meglétét.

### Példa - Bejáró tervezési minta

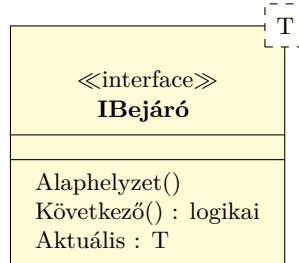
Végül nézzünk egy példát a már megismert polimorfizmus és a generikusok kapcsolatára. A típusparaméterek ugyanis nem csak az implementációban, hanem már az interfész szinten is megjelenhetnek.

Erre egy meglehetősen összetett példa a „Bejáró tervezési minta”, amit azért is érdemes most megvizsgálni, mert később az adatszerkezeteknél hasznos lesz. Ez a tervezési minta egy szabványos módot kínál különböző sorozatok (tipikusan adatszerkezetek) elemeinek egyesével való feldolgozására. Az adatszerkezeteket a későbbi fejezetekben fogjuk részletesen tárgyalni, jelenleg annyit érdemes tudni, hogy a

<sup>11</sup>constraint

különböző implementációk (pl. tömb, láncolt lista, fa, gráf) egészen másként tárolják el az adatokat. Jogos viszont az igény, hogy legyen egy általános módszer, amivel egyesével fel lehet dolgozni az egyes elemeket anélkül, hogy ismerni kellene a háttérben rejlő implementációs részleteket.

Mivel az egyes adatszerkezetek bejárásai különböző algoritmusokat és segédadatokat igényelnek, ezért ezeket célszerű az adatszerkezettől különálló objektumokban létrehozni, ezeket hívjuk majd „*bejárónak*”. Kihasználva az interfészek lehetőségeit, nem szükséges elmerülni az egyes adatszerkezetek különböző bejárási lehetőségeiben, ehelyett elég meghatározni egy szerződést, amivel definiálni lehet, hogy egy ilyen bejáró objektumnak milyen műveletekkel kell rendelkeznie. Ezt a legegyszerűbben egy interfész segítségével lehet megadni (1.15. ábra).

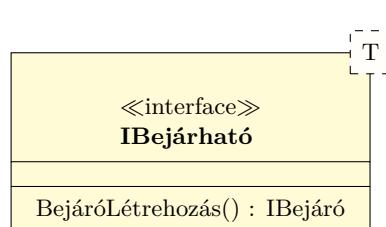


1.15. ábra. Bejáró interfész felépítése.

A bejáró objektumot általában úgy célszerű elképzelni, hogy rendelkezik egy belső „kurzorral”, ami az aktuálisan vizsgált elemre hivatkozik. Ez csak egy implementációs kérdés, ezért az interfészben nem is jelenik meg. Az egyes előírt műveletektől az alábbi működést várható el:

- **ALAPHELYZET:** Alaphelyzetbe állítja a bejáró objektumot. Ebben az alaphelyzetben a bejáró objektum képzeletbeli kurzora a legelső feldolgozandó elem előtt áll.
- **KÖVETKEZŐ:** Megpróbálja átléptetni a kurzort a következő elemre. Amennyiben ez sikerült (van következő elem), akkor a visszatérési értéke *igaz*. Amennyiben nem (már az utolsó elemen állt), akkor értéke *hamis*.
- **AKTUÁLIS:** Ezen a tulajdongáson keresztül lehet kiolvasni az adatszerkezet azon elemét, amin a kurzor éppen áll. Itt látható, hogy a bejáró miért generikus: ennek a tulajdonságnak a visszatérési értéke egy **T** típusú érték lesz.

Már csak egy technikai kérdés maradt hátra: miként kapunk egy bejáró objektumot az egyes adatszerkezetek bejárásához? Mivel minden adatszerkezet felépítése más lehet, ezért azok különböző bejárókat igényelhetnek (akár az is lehet, hogy egy adatszerkezetet több különböző bejáróval is feldolgozható). Itt megint érdemes használni a felelősség delegálásának elvét: nem az adatszerkezet használója hozza létre az adott adatszerkezetnek megfelelő bejárót, hanem ennek létrehozását rábízza magára az adatszerkezetre. Ezt a létrehozást is célszerű szabványosítani, ezért ehhez is tartozik egy interfész (az 1.16. ábra).



1.16. ábra. Bejárható interfész felépítése.

Ez egy nagyon egyszerű interfész, csak egyetlen metódust tartalmaz. Ez a **BEJÁRÓLÉTREHOZÁS** metódus létrehoz és visszaad egy olyan (megfelelően felparaméterezett) bejáró objektumot, ami képes lesz végiglépegetni az adatszerkezet elemein.

Láthatóan egy nagyon magas absztraktiós szinten dolgozunk, hiszen még azt se lehet tudni, hogy milyen adatszerkezetekkel kell dolgozni, azt pláne nem, hogy ezeket hogyan kell majd bejárni és

milyen módon kell majd implementálni ezeket a bejárókat. De jelen pillanatban ez nem is érdekes, elég megírni a szükséges szerződést, majd az adatszerkezet implementációjakor kell ezeknek a műveleteknek a megvalósításával foglalkozni.

Az interfész azonban már használható, például elkészíthető vele az 1.11. algoritmusban látható algoritmus. Ennek lépései:

1. Kér egy bejáró objektumot a paraméterként átadott *a* bejárható objektumtól. Az objektum felelőssége, hogy egy olyan bejárót adjon vissza, amivel végig tudunk majd lépegetni annak minden elemén.
2. Alaphelyzetbe állítja a bejáró objektumot, ami ilyenkor az első elem előre helyezi a kurzort.
3. A ciklus feltételben meghívjuk a **KÖVETKEZŐ** metódust. Amennyiben ennek visszatérési értéke *hamis*, az azt jelenti, hogy nincsenek további elemek az adatszerkezetben, ezért kilépünk a ciklusból. Vegyük észre, hogy már az első elem feldolgozása előtt meghívjuk a következő elemre lépő metódust, ezért fontos, hogy inicializáláskor a bejáró az első elem előtt álljon: ennek köszönhetően az első **KÖVETKEZŐ** híváskor pont az elsőre elemre fog lépni a kurzor.
4. Amennyiben igazat adott vissza, akkor kiolvassuk a bejáró által mutatott aktuális elemet és ezt feldolgozzuk.
5. Addig ismétljük ezt a ciklust, amíg van következő elem.

---

### 1.11. Algoritmus Egy bejárható adatszerkezet elemeinek feldolgozása.

---

**Bemenet:** *a* - **IBejárható**<*T*> (*a* bejárandó objektum)

```
1: függvény BEJÁRHATÓADATSZERKEZETFELDOLGOZÁSA(a)
2: bejáró ← a.BEJÁRÓLÉTREHOZÁS() ▷ Bejáró objektum létrehozása
3: bejáró.ALAPHELYZET() ▷ Bejáró objektum inicializálása
4: ciklus amíg bejáró.KÖVETKEZŐ() ▷ Következő elemre lépés, ha nincs, akkor kilépés
5: elem ← bejáró.AKTUÁLIS() ▷ Aktuális elem kiolvasása
6: ... az elem feldolgozása ...
7: ciklus vége
8: függvény vége
```

---

#### Felhasznált változók és függvények

- *bejáró* : Egy **IBejáró**<*T*> interfészt megvalósító objektum. Nem tudjuk, hogy pontosan milyen típusú, mivel az *a* hozzá létre számunkra.
  - *elem* : Egy *T* típusú lokális változó.
- 

Látható, hogy nem kellett azzal foglalkozni, hogy a bejáró milyen típusú elemeken megy végig. Ezt egyszerűen a *T* típusparaméter jelöli. Az adatszerkezet *T* típusú adatokat tartalmaz, emiatt egy *T* típusparaméterrel bíró bejárót ad vissza, ami majd *T* típusú elemeket fog visszaadni egyesével.

#### Megjegyzés

Bár a példákban a bejáró fogalmát összekapcsoltuk az adatszerkezetekkel, valójában nincs ilyen szoros kapcsolat közöttük. Bármilyen műveletre készíthetünk bejáratot, aminek az a célja, hogy egymás után adjon értékeket. Pl. egy tartományon belül a számok lekérdezéséhez, vagy akár valamelyen értékek generálásához.

## 2. fejezet

# Deklaratív programozás

### 2.1. Deklaratív paradigmá

Többféle programozási paradigmá/nyelv van, amelyeket általában jól el lehet helyezni az imperatív és deklaratív kategóriák közül valamelyikbe:

- Az imperatív paradigmá lényege, hogy a programozó feladata, hogy a programozási nyelven keresztül meghatározza, hogy a megoldandó feladatot milyen lépésekkel kell megoldani. Ehhez tipikusan meg kell határozna a megoldáshoz szükséges adatokat, az azokon végrehajtandó algoritmusokat, majd pedig ezekből fel kell építenie a szükséges programot. A program futása során valójában csak annyit tesz, hogy a programozó által megadott sorrendben végrehajtja az utasításokat. Ahogy az előző fejezetben láthattuk, az egyre magasabb absztraktiós szintek segítségével egyre távolabb kerül a programozó által írt forráskód a tényleges végrehajtott gépi kódjól, de ez a lényegen nem változtat, a programozónak kell meghatároznia, hogy **hogyan** kell megoldani a feladatot.
- A deklaratív programozást használó nyelvek általában jóval magasabb absztraktiós szintet képviselnek, de valójában nem ez a fő különbség. A programozó itt is egy programot ír, ez azonban már nem magát a megoldás lépéseiit kell, hogy tartalmazza, hanem „csak” magának a feladatnak a leírását, annak megoldási módját már a programozási nyelv/végrehajtó motor fogja kitalálni. Az idézőjel azért indokolt, mert a feladat leírását itt is egy előre meghatározott eszközrendszer segítségével tudjuk megtenni, és egy komplex feladat esetén ez a specifikáció is lehet kimondottan összetett és nehéz. A programozó feladata azonban mégis más, azt kell meghatároznia a deklaratív nyelv segítségével, hogy mi a feladat (**mit** kell csinálni), a megoldás lépéseiit tartalmazó „hogyan” már elmaradhat.

Elsőre futurisztikusnak tűnik a deklaratív hozzáállás, valójában már nagyon régóta velünk él, a legtöbben talán már használták is. Tipikus példája az SQL lekérdező nyelv, amivel nem foglalkozunk részletesebben, de egy egyszerű példa így is érthető lesz:

---

#### 2.1. Algoritmus A Csömörön lakó hallgatók lekérdezése SQL segítségével.

---

```
1: SELECT név FROM Hallgatók
2: WHERE irányítószám = 2141
3: ORDER BY név
```

---

A 2.1. algoritmus egy SQL lekérdezést mutat be, amelyik az adatbázisban található „Hallgatók” táblából kiválogatja azokat a rekordokat, amelyeknél az „irányítószám” mező értéke „2541”, majd listázza ezen rekordok „név” mezőit a név alapján rendezve.

Ha valaki gyakran használja az SQL-t, akkor talán fel se tűnik neki, hogy ez egy deklaratív megadás, hiszen a lekérdezés csak azt határozza meg, hogy mi a feladat, arra vonatkozóan nem tartalmaz utasításokat, hogy ezt milyen lépésekkel kell meghatározni. Lehet ugyan sejteni, hogy a megoldás miként fog végrehajtódni (meg kell nyitni a táblát tartalmazó fájlt, azon egy ciklussal végig kell lépegetni, minden

rekordnál ellenőrizni kell a feltételt, a feltételnek megfelelőket ki kell válogatni, majd ezeket rendezni) de ez nem érdekes, illetve lehet, hogy nem is igaz (egy adatbázis szerver ennél nagyságrendekkel bonyolultabban és hatékonyabban oldja meg a problémát, pl. a gyakran használt adatokat a memóriában tárolja, a kiválogatáshoz indexeket használ, ezeket fel tudja használni a rendezéshez is, stb.).

A fenti csak egy példa, emellett számos deklaratív eszköz létezik. Általában persze jellemző rájuk, hogy sokkal szűkebb körben használhatók, mint a teljesen általános problémamegoldásra alkalmas imperatív nyelvek. Ezekben a területeken viszont nagyon jól használhatók, nem csak maga a program elkészítése egyszerűbb és gyorsabb, de maga a végrehajtás is gyakran sokkal hatékonyabb a nyelvi specialitások kihasználása miatt.

A két fő programozási paradigmá, amelyek ezt az irányt képviselik a funkcionális és a logikai programozás paradigmák. Ez a fejezet ezekkel foglalkozik érintőlegesen, majd röviden áttekintjük a hibrid megvalósítások lehetőségeit, amikor a már meglévő imperatív nyelvekben jelennek meg funkcionális elemelek.

## 2.2. Tisztán funkcionális nyelvek

### 2.2.1. Funkcionális program alapelemei

#### Program alapvető felépítése

A Neumann-elvű gépek működéséhez igazodó imperatív programozás általában a végrehajtó gép állapot-változásaira összpontosított. A programozó létre tud hozni változókat (ezek aktuális értéke határozza meg az aktuális állapotot), amelyek értékét a megadott sorrendben lefutó értékadásokkal lehet módosítani (ami megfelel az állapotváltásoknak). Ehhez képest a funkcionális nyelvek rémisztőek lehetnek, hiszen itt nincsenek változók, sem ciklusok, de az utasításoknak sincs időbeli sorrendje (sőt, igazából utasítások sincsenek).

Szerencsére van élet a változtatható állapoton túl is, nagyon sok feladat egyszerűen megoldható a hagyományos értelemben vett változók használata nélkül.

Egy funkcionális program alapvetően az alábbiak szerint épül fel:

- definíciók sorozata, ezek többnyire megfelelnek a már megismert függvény fogalomnak, bár azokat jóval tágabban értelmezzük;
- egy kiértékelendő kifejezésből, ami magát a megoldandó feladatot képviseli.

Ez elég korlátosnak tűnik, de valójában így is sokféle feladatot meg lehet oldani:

- a kiértékelendő kifejezés lehet egy optimalizálási eljárás, amelyik a megadott paraméterek alapján meghatározza, hogy egy gyártási feladatot milyen ütemezéssel lehet a leghatékonyabban megvalósítani;
- akár egy animációt renderelő eljárást is készíthetünk, ahol egy  $f(t, x, y)$  kifejezés kiszámítja a képernyő  $x, y$  pixelének színét a  $t$  időpillanatban a rendelkezésre álló modell adatok alapján, ezzel pedig ki lehet rajzolni a teljes képernyő tartalmát. Ha ismertek a modelltérben lévő objektumok alakjai, azok mozgási adatai, a fényforrások helyzete, stb., akkor minden pillanatban ki tudjuk számolni, hogy a képernyő egy megadott pixelére milyen intenzitású és színű fénysugár fog bérkezni.

Mindezek persze megoldhatók az imperatív nyelvekkel is, de a funkcionális formában elképzelhető, hogy áttekinthetőbb, könnyebben karbantartható és főleg hatékonyabban futó programokat lehet készíteni.

#### Változók helyett értékek

A funkcionális nyelvekben nincsenek az imperatív paradigmából ismert változók, helyettük a matematikai környezetben már megismert „értékek”-hez hasonló megközelítéssel találkozunk. Érdemes átgondolni a két fogalom közötti különbséget. Egy  $x = 1$  utasítás teljesen egyértelmű a programozói gyakorlatban (az  $x$  változó értéke legyen 1) és matematikai oldalról is korrekt ( $x$  egyenlő 1-el). Egy  $x = x + 1$  utasítás imperatív programozói stílushoz szokott szemnek még mindig teljesen valid (az  $x$  változó értéke legyen az  $x$  változó aktuális értéke + 1), matematikai szemmel viszont már ellentmondást jelent ( $x$  értéke nem lehet egyenlő  $x + 1$  értékkel).

Jobb szó híján továbbra is változónak fogjuk nevezni ezeket a nyelvi elemeket, bár precízebb lenne a „nem frissíthető változó” elnevezés. Tehát használhatunk a programban névvel ellátott változókat, ezeknek azonban csak egyszer lehet értéket adni (értéket kötni hozzájuk), ez azonban a program futása közben nem változhat meg. Tehát nincsenek „destruktív értékadások”, amelyek során egy változó értéke elvész egy felülírás során. Imperatív programozási nyelvekhez szokva ez óriási különbséget jelent, hiszen úgy kell megírni a funkcionális programokat, hogy nem lehet bennük használni a klasszikus értelemben vett változókat.

Ebből a jellemzőből viszont adódik egy nagy előny is, a „hivatkozási helyfüggetlenség”. Tehát egy kifejezés értéke teljesen független attól, hogy a program melyik részét található, és attól is, hogy mikor kerül kiértékelésre. Egy megadott kifejezés értéke mindenhol ugyanaz lesz. Másképpen megfogalmazva: a függvényeknek nincsenek „mellékhatásai”, tehát egy függvény működése során nem változtatja meg a kifejezések értékét (tehát egy  $a + b$  kifejezés értéke mindenhol ugyanaz marad, ami abból adódik, hogy egyik függvény se fogja tudni megváltoztatni sem az  $a$ , sem pedig a  $b$  értékét).

Azokat a funkcionális nyelveket, amelyekben nincsenek mellékhatások, nevezzük „tisztán funkcionális nyelveknek” (pl. Haskell, Clean, Miranda), ellentétben a „nem tisztán funkcionális nyelkekkel” (pl. F#, LISP, Erlang).

Ennek a fontos korlátnak persze megvan a maga indoka: az így elkészített funkcionális programok nagyon hatékonyan futtathatók:

- Egy kifejezést minden elég egyszer kiértékelni, hiszen a program későbbi futása során minden ugyanaz lesz az értéke. Emiatt ezek a részeredmények eltárolhatók egy gyorsító tárból és bármikor gyorsan kikereshetők.
- A tisztán funkcionális programok nagyon hatékonyan párhuzamosíthatók, hiszen nincsenek bennük mellékhatások. Egy  $f(x) + g(x)$  kifejezést az imperatív fordító csak úgy tud kiszámítani, hogy elsőként elvégzi az  $f$ , majd ezt követően a  $g$  kiértékelését, hiszen elképzelhető, hogy az  $f$  futtatása közben megváltozik az  $x$  értéke (vagy bármilyen egyéb globális változó, amit minden függvény használ). Funkcionális nyelvek esetében a kiértékelés két külön szalon párhuzamosan is elindulhat, hiszen a kifejezések értéke minden ugyanaz.

### Magasabb rendű függvények

A matematikai jelöléssrendszerben zavaró lehet, hogy a függvény definiálás és a függvény alkalmazás jelölése azonos. Az első eset egy nevet rendel a megadott kifejezéshez (mint pl.  $f(x) = x^2$ ), a másik pedig a függvény alkalmazását jelenti az  $x$  paraméterre (tehát ugyanúgy  $f(x)$ ). A matematikai rendszerben ez nem okoz problémát, mert kontextus miatt egyértelmű, a programozási nyelvekben azonban ez nem szerencsés. Ezért függvény definíció gyanánt a programkódokban a  $::=$  műveletet fogjuk használni (pl.  $f(x) := x^2$ ), alkalmazásnál pedig a már megszokott hagyományos formát ( $f(x)$ ).

Függvények már az imperatív paradigmában is megjelentek, ezek a kód-újrafelhasználás okán jelentek meg, és utasításokat fogtak össze egy egységre. A függvénytellet lehetett végrehajtani: meghívni, ennek során a benne lévő utasítások lefutottak.

Funkcionális programozási nyelvekben a függvények használata sokkal rugalmasabb, azok önálló életet élnek.

- függvényt értékül adhatunk egy változónak (tehát nem a függvény visszatérési értékét, hanem magát a függvényt);
- ezen a változón keresztül használhatjuk is a neki értékül adott függvényt;
- függvény átadható paraméterként egy másik függvénynek;
- egy függvény visszatérési értéke lehet egy másik függvény;
- a fentiek nem csak névvel ellátott függvényekre, hanem név nélküli kifejezésekre is igazak.

Azokat a függvényeket, amelyek paraméterként más függvényeket várnak, vagy éppen visszatérési értékük egy függvény, „magasabb rendű függvényeknek” nevezzük.

---

## 2.2. Algoritmus Magasabbrendű függvények használata.

---

```

1: NégyzetreEmel(x) := x * x
2: Nagyobbít(a, fv) := a < fv(a)
3:
4: b = Nagyobbít(4, NégyzetreEmel)

```

---

A 2.2. algoritmus néhány példát mutat a függvényekkel való műveletekre. Az első sor megad egy már megszokott függvényt **NégyzetreEmel** néven, aminek definíciója a paraméterként átadott szám négyzetre emelése. A 2. sorban a **Nagyobbít** függvény paraméterként egy számot és egy másik függvényt vár. Visszatérési értéke pedig egy logikai érték, ami akkor igaz, ha a függvény visszatérési értéke a megadott paraméterrel nagyobb, mint az eredeti szám. A 4. sorban látható ez utóbbit egy lehetséges meghívása, a második paraméterként átadható a **NégyzetreEmel** függvény. A függvény neve utáni zárójelek (és a paraméter) hiánya is azt mutatja, hogy itt nem a **NégyzetreEmel** függvény meghívása és a visszatérési érték átadása történik, hanem maga a függvény lesz a paraméter.

## Névtelen kifejezések

Az imperatív nyelvekben is gyakran előfordul, hogy anélkül dolgozunk memóriabeli adatokkal, hogy azokhoz változónévét társítanánk. Például függvény híváskor ha konstans paramétert adunk át (pl.  $f(1)$ ), vagy éppen egy kifejezés értékét ( $f(a + b)$  vagy  $f(a > 2)$ ). A paraméterátadás mellett ez igaz a visszatérési értékekre is (pl. **vissza**  $a + 2$ ), és még számos példát lehetne hozni. Tehát ez már teljesen megszokott, hogy csak annak adunk nevet, amit megadott környezetben a későbbiekben is el akarunk érni. Ha ez nem áll fent, akkor a név is szükségtelen.

A funkcionális nyelvekben a függvények az adatokhoz hasonlóan eltárolhatók, paraméterként átadhatók, illetve lehetnek függvények visszatérési értékei. Ennek egyik megoldása az, hogy egy klasszikus módon létrehozott és elnevezett függvényre hivatkozunk a nevén keresztül. Azonban ez a lépés ki is hagyható, és ha úg cél szerűbb, akkor közvetlenül egy kifejezést is használhatunk ezeken a helyeken, amelyekhez nem rendelünk nevet.

Például a négyzetre emelést az alábbi kifejezés fogja képviselni:  $x \rightarrow x * x$ . A 2.3. algoritmus ugyanazt a feladatot oldja meg, mint az előző példa, csak kihagyjuk a **NégyzetreEmel** függvény definícióját. Az utolsó sor pedig arra mutat példát, hogy egy ilyen névtelen kifejezés is alkalmazható megadott paraméterekekkel, a  $d$  értéke 4 lesz.

---

### 2.3. Algoritmus Kifejezések használata.

---

```
1: Nagyobbít(a , fv) := $a < fv(a)$
2:
3: $b = \text{Nagyobbít}(4, x \rightarrow x * x)$
4: $d = (x \rightarrow x * x)(2)$
```

---

## Vezérlési szerkezetek

Az elágazásokhoz hasonló szerkezetekkel matematikában is találkozhatunk, amikor egy kifejezés kiszámítási módja valamilyen feltételektől függően több ágon írható csak le. Ennek megfelelően ezek a gyakran többágú elágazások a funkcionális nyelvekben is megjelennek. Ennek nincs semmilyen akadálya, hiszen maga az elágazás nem változtat meg semmilyen állapotot, így nem jár mellékhatalásokkal sem.

Sokkal érdekesebb a kérdés, hogy ha nincsenek változók, akkor miként lehet pl. ciklusokat készíteni? Az imperatív értelemben vett ciklust (ami addig ismétli a ciklusmagot, amíg a változók értékét vizsgáló ciklusfeltétel igaz) tulajdonképpen sehogy, viszont minden ciklus átalakítható rekurzív függvényhívások formájára.

Példaként nézzük meg, hogy miként tudunk írni egy imperatív függvényt, aminek visszatérési értéke 0 és a paraméterként megadott  $n$  közötti számok összege (2.4. algoritmus).

---

### 2.4. Algoritmus 0-tól $n$ -ig számok összeadása imperatív szemlélettel.

---

**Bemenet:**  $n$  - egész<sup>+</sup> (egy egész szám)  
**Kimenet:** . - egész<sup>+</sup> (az egész számok összege 0-n között)

```
1: függvény SZUMMA(n)
2: $s \leftarrow 0$
3: ciklus $i \leftarrow 1$ -től n -ig
4: $s \leftarrow s + i$
5: ciklus vége
6: vissza s
7: függvény vége
```

---

Funkcionális megoldás esetén (2.5. algoritmus) ez az irány nem járható, hiszen a paradigma nem támogatja sem a változók, se a hagyományos értelemben vett ciklusok használatát. Egy rekurzív függvény definícióval (ezt jelzi a  $\lambda$  jel) azonban egyszerűen megoldható a feladat, illetve látható, hogy az elágazások kezelése is egyszerűen megvalósítható.

## 2.5. Algoritmus 0-től $n$ -ig számok összeadása funkcionális szemlélettel.

```
1: Szumma(n) :=
2: 0 , ha n = 0;
3: Szumma(n-1) + n, egyébként
```

### Megjegyzés

A funkcionális szemléletű megoldások során nem követjük egyik konkrét funkcionális nyelv szintaktikáját sem. Egy új pszeudokód jelölésrendszer bevezetése is szükségtelen, hiszen a témát csak érintőlegesen tárgyaljuk. A példák remélhetőleg így is érhetők lesznek.

Vegyük észre, hogy a második megoldás egyáltalán nem használ a klasszikus értelemben vett változókat. Sem az *i* ciklusváltozó, sem pedig az *s* összegzéshez használt segédváltozó nem szükséges. Tehát mellékhatások nélkül sikerült megoldani ugyanazt a feladatot.

### Mintaillesztés

A funkcionális nyelvek egyik specialitása a „mintaillesztés”. Ez némileg hasonlít az OOP nyelvekben már megismert túlterheléshez (tehát amikor több azonos nevű, de különböző paraméterezeittségű függvény is jelen van egy osztályon belül). Funkcionális nyelveknél ez általában használható, bármelyik függvénynek készíthetünk több, különböző paraméterekkel bíró megvalósítást. Ezek közül minden a hívási paramétereknek megfelelő kerül végrehajtásra.

Mintaillesztés segítségével például az a 2.6. algoritmus segítségével tudjuk elkészíteni egy sorozat elemeinek az összegzését. Mint látható, a *Szumma* függvénynek két definíciója is van. Az egyik arra az esetre vonatkozik, amikor a függvény paraméterként 0-t kap, a másik pedig az ettől eltérő általános esetekre. A kiértékelés során pedig minden a lehető legspecifikusabb definíciót fogja választani a rendszer.

## 2.6. Algoritmus 0-től $n$ -ig számok összeadása mintaillesztéssel.

```
1: Szumma(0) := 0
2: Szumma(n) := Szumma(n-1) + n
```

A végeredmény persze ugyanaz, mint a 2.5. algoritmusban látható, de a gyakorlatban ez a megoldás sokszor jóval rugalmasabban használható. Sok esetben, amikor a rekurzív hívásban számos feltétel dönti el, hogy milyen esetekben mit kell kiszámítani, áttekinthetőbb lehet ez a forma.

### 2.2.2. Végrehajtás menete

#### Átalakítások szerepe

A funkcionális programok nem időérzékenyek, tehát itt a végrehajtás során nem változnak változók értékei, illetve program állapotai. A végrehajtás a keződő kifejezés kiértékelését jelenti, ami tipikusan különböző szabályok szerinti átalakításokat jelent. Ezek a redukciós lépések tulajdonképpen behelyettesítések, amelyeket egészen addig hajt végre a futtató motor amíg nem érkezik el egy végső minimális állapotba, amikor már nem lehet további szabályokat használni. Ezt nevezzük a kező kifejezés normál formájának.

Ilyen behelyettesítések lehetnek például:

- egy függvény név bármikor helyettesíthető a függvény definíciójával;
- egy függvény alkalmazása tulajdonképpen kicseréli a függvény definícióból a formális paraméter nevét a paraméter értékére;

- egy elágazás helyettesíthető azzal az ággal, amelyiket a feltétel megad (emlékeztetőül: a változók értéke nem függ a kiértékelés idejétől, tehát a feltételek igaz/hamis volta egyértelműen meghatározható);
- egyszerű aritmetikai kifejezések helyettesíthetők azok értékével (pl.  $2+3$  helyett írható, hogy 5).

Nézzük az alábbi egyszerű példát, miként lehet kiértékelni a 2.7. algoritmusban látható függvény egy megadott paraméterrel.

---

## 2.7. Algoritmus

A már megismert összegzés, csak egy sorba írva.

---

1: 1: Szumma(n) := 0, ha n = 0; Szumma(n-1) + n egyébként

---

A 2.1. ábra bemutatja a kiértékelés menetét abban az esetben, ha a függvény alkalmazásakor az átadott paraméter értéke 3. A fent felsorolt helyettesítések az ott megadott sorrendben futnak le, egymás után többször.

```

Szumma(3)
→(0, ha n = 0; Szumma(n-1) + n, egyébként)(n = 3)
→0, ha 3 = 0; Szumma(3-1) + 3, egyébként
→Szumma(3-1) + 3
→Szumma(2) + 3
→(0, ha n = 0; Szumma(n-1) + n, egyébként)(n = 2) + 3
→0, ha 2 = 0; Szumma(2-1) + 2, egyébként + 3
→Szumma(2-1) + 2 + 3
→Szumma(1) + 5
→(0, ha n = 0; Szumma(n-1) + n, egyébként)(n = 1) + 5
→0, ha 1 = 0; Szumma(1-1) + 1, egyébként + 5
→Szumma(1-1) + 1 + 5
→Szumma(0) + 6
→(0, ha n = 0; Szumma(n-1) + n, egyébként)(n = 0) + 6
→0, ha 0 = 0; Szumma(0-1) + 0, egyébként + 6
→0 + 6
→6

```

2.1. ábra. A redukció lépései.

A példa a funkcionális nyelvek működésének nagyon erős leegyszerűsítése, de jól látható, hogy a program egészében csak a szimbólumok manipulációján alapul. Ezeket az átalakításokat addig ismétlik, amíg elér egy szükséges végállapotot, ami a példában egy tovább már nem alakítható kifejezést jelent. Nincsenek változók, ciklusok, memóriaállapot, verem, stb., maga a rekurzió kezelése is egészen más, mint ahogy az imperatív nyelveknél látható volt.

Ez a fajta végrehajtás számos előnyvel bír:

- Egy függvény azonos paraméterekekkel alkalmazva minden ugyanazt az eredményt fogja visszaadni, ami lehetőséget ad az eredmények gyorsítótárazására.
- Mivel az egyes lépések értéke nem függ a végrehajtás idejétől, ezért az egymástól független átalakítások párhuzamosíthatók is, így egy nagyobb kifejezés kiértékelése hatékonyan és automatikusan gyorsítható.
- Nyugodtan el lehet hagyni azokat a függvényeket, amelyek eredményét nem használja más, hiszen ezek ezek biztosan nem fogják befolyásolni a végeredményt.
- A módszerrel megoldhatók olyan feladatok is, amelyeket a számszerű kiértékelést végző imperatív nyelvek nem alkalmasak. Pl. két függvény azonosságának igazolása azáltal, hogy az átalakításokkal azonos alakra lehet-e hozni őket.

- Nincs arra nézve korlát, hogy a függvényekkel milyen műveleteket lehet végrehajtani. Pl. lehet olyan művelet is, ami egy függvény deriváltját adja vissza.

### Kiértékelési stratégiák

Kérdés persze, hogy milyen sorrendben kell végrehajtani az előzőleg említett redukciós lépéseket. Az imperatív paradigma „közvetlen vezérlést” használt, tehát az utasítások abban a sorrendben futottak le, ahogy azokat a programozó leírta. Deklaratív nyelvek esetében ez általában már nem igaz, a programkódban található sorrend nem befolyásolja a futtatáskori sorrendet. Ehelyett két fő vezérlési mód használatos:

- A lusta kiértékelés<sup>1</sup> során egy kifejezés csak akkor kerül kiértékelésre, amikor annak az értékére szükség van (2.2. ábra). Ez persze gyakran egy hosszú láncot indít el, hiszen a kifejezés kiértékelése miatt szükség lehet egy másik kifejezés értékére, stb. A módszer előnye, hogy biztosan nem fog felesleges munkát végezni a rendszer, hiszen azokra a kifejezésekre, amelyek ugyan definiálva lettek, de nem szükségesek a végeredmény kiszámításához, soha nem kerül a végrehajtás.

```
NégyzetreEmel(NégyzetreEmel(3)+2)
→(NégyzetreEmel(3)+2)*(NégyzetreEmel(3)+2)
→(3*3+2)*(3*3+2)
→121
```

2.2. ábra. Lusta kiértékelés lépései.

- A mohó kiértékelés<sup>2</sup> esetén a végrehajtás éppen fordított logikát követ: amennyiben egy kifejezés kiértékelhető, akkor azt azonnal megteszí (2.3. ábra). Ami szintén elindíthat egy láncot, hiszen ennek hatására újabb kifejezések is kiértékelhetővé válnak, stb. Ez néha jobb teljesítményhez vezet, illetve a programvégrehajtás is könnyebben nyomonkövethető.

```
NégyzetreEmel(NégyzetreEmel(3)+2)
→NégyzetreEmel(3*3+2)
→NégyzetreEmel(9+2)
→NégyzetreEmel(11)
→11*11
→121
```

2.3. ábra. Mohó kiértékelés lépései.

Néhány speciális különbségtől eltekintve (pl. lusta kiértékelés esetén létezhetnek végtelen adatszerkezetek) mindenki megoldás működik a gyakorlatban. Bizonyos nyelvek esetében a kiértékelés módja egy optimalizációs döntés eredménye, vagy maga a programozó is befolyásolhatja.

Minden módszer okoz technikai problémákat is, hiszen a program utasításainak végrehajtási sorrendje már nem a programozó kezében van. Ez részben nehezíti a kód karbantartást, hibakeresést, de felvet újszerű elvi kérdéseket is a kimenet/bemenet kezelésével kapcsolatban.

### 2.2.3. Adatszerkezetek

Számítógépet tipikusan nagy mennyiséggű adatfeldolgozására érdemes használni, amelyeket az imperatív nyelvek különféle adatszerkezetekben tudnak eltárolni. Funkcionális nyelvek esetében a listáknak van fontos szerepük az adattárolás területén. Ezek értékei természetesen nem változtathatók meg, ehelyett szükség esetén újabb listákat lehet létrehozni, amelyek csak bizonyos elemeknél különböznek az eredetitől.

Az imperatív tömböknel megszokott kezelési módok a funkcionális listáknál általában nem használhatók. A klasszikus ciklussal való végigjárásnak pl. eleve nincs ciklus, nincs ciklusváltozó, illetve a lista

<sup>1</sup>lazy evaluation

<sup>2</sup>strict evaluation/eager evaluation

elemeit se biztos, hogy el lehet érni indexeléssel. Emellett lehetséges, hogy végtelen hosszú listákkal kell dolgozni.

Példáinkban a szögeletes zárójelek fogják jelölni a listájat, ahol [ ] az üres lista, a  $[x : y]$  jel pedig az  $x$  és  $y$  elemekből összeállít egy listát. Ha  $x$  vagy  $y$  egyedi elemek, akkor ők maguk kerülnek a listába, ha valamelyikük már maga is lista, akkor annak az elemei.

A 2.8. algoritmusban látható [SzámLista](#) függvény egy példát mutat be, hogy miként lehet előállítani listákat, amelyek 1-től a pozitív egész  $x$  paraméterig tartalmazzák az egész számokat. Az  $A$  értéke a kiértékelés végén egy lista lesz, aminek elemei  $[1 : 2 : 3 : 4 : 5]$ . Mint látható, nem okoz gondot, hogy nincs ciklus, illetve a lista nem módosítható. A megoldás rekurziót használ, és minden szinten egy új listát hoz létre az előző adatainak kiegészítésével.

---

## 2.8. Algoritmus

Lista generálása 1-től  $x$ -ig az egész számokból.

---

```
1: SzámLista(0) := []
2: SzámLista(x) := $[x : \text{Számok}(x-1)]$
3:
4: $A = \text{SzámLista}(5)$
```

---

A  $[ : ]$  jelölés azonban nem csak összefűzésre használható, hanem a paraméterlistában is. A későbbi példákban látható  $[első : többi]$  jelölés azt jelenti, hogy paraméterként egy listát vár a függvény, aminek az első eleme az  $első$  értéke lesz (ez tehát egy elem), a többi eleme pedig a  $többi$  értéke lesz (ez tehát egy lista).

Erre mutat példát a 2.9. algoritmus, ahol a [ListaÖsszegez](#) függvény visszatérési értéke a paraméterként átadott lista elemeinek az összege. Ez egy üres lista esetében értelemszerűen 0, általános esetben pedig a lista első eleméhez hozzá kell adni a lista első utáni elemeinek az összegét (ahol ez utóbbi egy rekurzív hívással számolható ki). Tehát az  $s$  értéke 10 lesz.

---

## 2.9. Algoritmus

Listában lévő számok összegének kiszámítása.

---

```
1: ListaÖsszegez([]) := 0
2: ListaÖsszegez([első : többi]) := első + ListaÖsszegez(többi)
3:
4: $s = \text{ListaÖsszegez}(5 : 2 : 3)$
```

---

A listák mellett a funkcionális nyelvek tipikus adatszerkezetei a rendezett n-es<sup>3</sup>, illetve egyéb matematikából már ismert fogalmak, mint például a halmazok. A lista kiértékelés megengedi, hogy nem kell a memóriában előre eltárolni az adatszerkezetek tényleges elemeit, azokat generátorokkal futás közben is elő lehet állítani. Tehát egy-egy elem mindenkor kerül kiszámításra, amikor arra éppen szükség van, ami így lehetővé teszi a végtelen listák kezelését is.

<sup>3</sup>tuple

## 2.2.4. Példa - Rendezés funkcionális szemlélettel

---

### 2.10. Algoritmus Sorozat rendezése funkcionális szemlélettel.

---

```
1: Rendez([]) := []
2: Rendez([első : többi]) :=
3: [Rendez(Válogat(többi, x→x<első)) : első : Rendez(Válogat(többi, x→x≥első))]
4:
5: Válogat([], feltétel) := []
6: Válogat([első : többi], feltétel) :=
7: [első : Válogat(többi, feltétel)] ha feltétel(első) = igaz
8: Válogat(többi, feltétel) egyébként
9:
10: A = Rendez([5 : 8 : 9 : 1 : 7 : 4])
```

---

A 2.10. algoritmus egy rendezést mutat be a funkcionális szemléletmódnak megfelelően. A jelölésmód továbbra is esetlen, de talán így is érthető a működési elv. Két függvényt definiálunk:

- A **Válogat** függvény szerepe, hogy az első paraméterként átadott sorozat elemei közül kiválogassák a második paraméterként megadott feltételnek megfelelőket, majd ezeket adja vissza egy új sorozatban.
- A **Rendez** függvény pedig a paraméterként átadott sorozatot rendezzi és visszatérési értéke egy másik sorozat a rendezett elemekkel.

Mint látható, a megoldás felhasználja a mintaillesztés lehetőségeit, tehát minden függvénynek két-két definíciója is megjelenik:

- Az első **Válogat** függvény azt az esetet képviseli, amikor első paraméterként egy üres sorozat érkezik. Ilyenkor a válogatás visszatérési értéke értelemszerűen egy üres sorozat a feltételtől függetlenül.
- A második **Válogat** függvény akkor fut le, amikor egy valódi sorozat érkezik első paraméterként. A már megismert módon a sorozatot nem egyben kezeljük, hanem külön az **első** elemét, illetve az ezt követő **többi** elemekből álló részsorozatot. A feltétel megnézi, hogy az **első** elem megfelel-e a paraméterként átadott feltételnek
  - ha igen, akkor a visszatérési érték egy sorozat lesz, amibe bekerül az **első** elem (hiszen az megfelelt a feltételnek), illetve a **többi** elemek közül a feltételnek megfelelők. Ez utóbbiakat egy rekurzív hívás válogatja ki, amit a **[ : ]** művelettel összefűzünk az első elemmel.
  - ha nem, akkor a visszatérési érték sorozatába nem kerül bele a **első** elem, annak tartalma a rekurzív hívás eredménye lesz a **többi** elemekre.

Látható, hogy a feltételtől függetlenül minden egyre rövidebb részsorozatra hívja meg magát a kiválogatás, előbb utóbb eljut az egy elemű sorozathoz, ahol már egy üres sorozatot ad át a rekurzív hívás paramétereiként. Ilyenkor már az erre vonatkozó másik **Válogat** függvény fog lefutni.

- Az első **Rendez** a triviális esetet képviseli: üres sorozat rendezése esetén egy üres sorozatot ad vissza.
- A második **Rendez** függvény általános esete. A paraméterként kapott sorozatot ez a függvény is egy **első** és azt ezt követő **többi** elemekként kezeli. A rendezést visszavezeti az előbb megismert kiválogatásra és a rekurzív rendezés hívásokra. A végeredményt a **[ : ]** művelettel fűzi össze az alábbi részsorozatokból:
  - Az elsőt követő **többi** elemei közül kiválogatja az **első** elemnél kisebbeket, majd ezeket rendezzi egy rekurzív hívással.
  - Maga az **első** elem.

- Az elsőt követő *többi* elemei közül kiválogatja az *első* elemnél nagyobb (esetleg egyenlő) elemeket, majd ezeket is rendezи egy rekurzív hívással.

A pszeudokód alján látható egy lehetséges meghívás. Vegyük észre, hogy a rendezés során egy változónak se változott meg az értéke, illetve ciklusokat se használtunk. Az így elkészített rendezés mégis jól működik, ráadásul nagyon jól párhuzamosítható.

### 2.2.5. $\lambda$ -kalkulus\*

#### $\lambda$ -kifejezések

A funkcionális nyelvek alapvetően a  $\lambda$ -kalkulusnak nevezett elméletben alapulnak, amivel csak érintőlegesen foglalkozunk. Az elmélet kidolgozásának még nem volt köze a későbbi programozáshoz, Church célja az volt, hogy a matematika formális leírására adjon meg egy rendszert, amivel a függvények tulajdonságai jól vizsgálhatók. Később erre az elméletre épült a funkcionális programozás paradigmája.

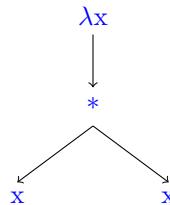
Az elmélet alapját az úgynevezett  $\lambda$ -kifejezések jelentik, amelyből három félével találkozhatunk [15]:

- A kifejezés lehet egy egyszerű változó.
- A  $\lambda$ -absztrakció egy olyan  $\lambda$ -kifejezés, amely egy névtelen függvény definiálására szolgál. Ennek jelölése:  $\lambda x.E$ , ahol az  $x$  egy változó (az absztrakció változója vagy formális paramétere), az  $E$  pedig egy  $\lambda$ -kifejezés (az absztrakció törzse). Mivel ezek a kifejezések kimondottan komplexek is lehetnek, ezért használható egy gráf alapú jelölése is (2.4. ábra)



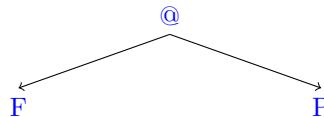
2.4. ábra. A  $\lambda x.E$  absztrakció ábrázolása gráf segítségével.

Például a négyzetre emelés:  $\lambda x.x * x$  (2.5. ábra).



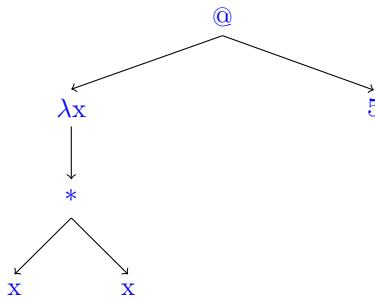
2.5. ábra. A  $\lambda x.E$  absztrakció ábrázolása gráf segítségével.

- Egy függvényből és annak paraméteréből álló párt pedig applikációt nevezünk, ennek jelölése:  $FP$  (a kettő között egy szóköz található), ahol mind az  $F$ , mind pedig a  $P$  tetszőleges  $\lambda$ -kifejezés. Ez tulajdonképpen a függvény alkalmazását jelenti, amit a gyakorlatban egyszerű változó-behelyettesítést jelent. A gráf ábrázolás esetében a „@” szimbólum jelzi az applikációt (2.6. ábra).



2.6. ábra. Az  $FP$  applikáció ábrázolása gráf segítségével.

Például a  $\lambda x.x * x 5$  (2.7. ábra) esetében az  $x$  helyére egyszerűen behelyettesítjük az 5 értéket, így az eredmény  $5 * 5$  lesz.



2.7. ábra. Az FP applikáció ábrázolása gráf segítségével.

A módszer előnye, hogy ezzel a nagyon egyszerű eszközzel tetszőleges bonyolultságú matematikai elemek is leírhatók és elemezhetők. A  $\lambda$ -kifejezésekkel minden kiszámítható függvény leírható. Az elmenetek vannak további bővítései, pl. típusokkal, de ezekkel nem foglalkozunk.

### Többváltozós függvények kezelése

Ahogy látható, egy  $\lambda$ -absztrakció csak egy változót tud kezeln. Többváltozós függvényeket szerencsére egyszerűen vissza lehet vezetni erre az egyszerű formára.

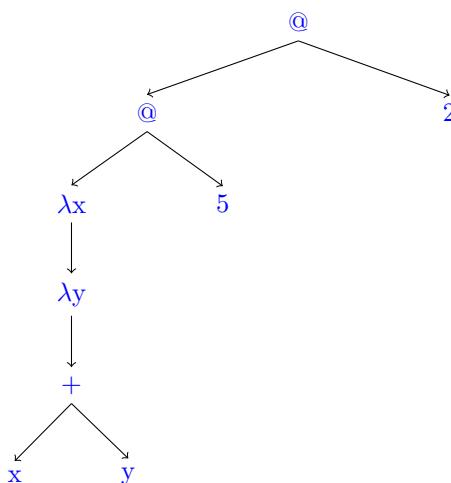
A Curry módszer lényege, hogy egy több paraméteres függvény visszatérési értéke lehet egy  $\lambda$ -kifejezés, illetve a paraméterek egy része. Ezzel tudjuk a többparaméteres függvényeket visszavezetni egyparaméteres függvények applikációjának sorozatává.

Például két szám összeadásának  $\lambda$  kifejezése az alábbi (2.8. ábra):  $(\lambda x.(\lambda y.x + y)) 5 2$

A zárójelezés azt mutatja, hogy ez olyan  $\lambda$ -absztrakció, aminek a változója  $x$ , az absztrakció törzse pedig egy újabb  $\lambda$ -absztrakció, nevezetesen  $\lambda y.x + y$ .

Az eredeti  $\lambda$  kifejezés alkalmazásán keresztül érthető meg ennek működése:

- Példaként legyen maga az applikáció:  $(\lambda x.(\lambda y.x + y)) 5 2$ .
- Elsőként alkalmazzuk a  $\lambda x.(\lambda y.x + y)$  kifejezést 5 paraméterrel, aminek következtében az  $x$  helyére behelyettesítjük az 5-öt. minden más változatlan marad, amiatt ennek eredménye  $(\lambda y.5 + y) 2$ .
- Már volt szó magasabb rendű függvényekről, tehát nem meglepő, hogy a kifejezés kiértékelésének eredménye egy másik kifejezés, ami tulajdonképpen az „öttel való növelés” kifejezése.
- Ezt a kifejezést is alkalmazzuk a 2 paraméterrel, ami egy újabb behelyettesítés, aminek eredménye:  $5 + 2$ .
- Ez pedig már egyszerű aritmetikai számítás:  $5 + 2 = 7$ .



2.8. ábra. A  $(\lambda x.(\lambda y.x + y)) 5 2$  kifejezés ábrázolása gráf segítségével.

## Változók kezelése

A  $\lambda$ -kifejezések kétféle változót tartalmazhatnak:

- Egy  $\lambda x.x + 1$  kifejezésben az  $x$  változót „kötött változónak” nevezzük. A kifejezés értéke minden  $x + 1$  lesz, tehát az  $x$  függvénye. Kötöttnek akkor nevezünk egy változót, ha tőle balra található olyan  $\lambda$ , amelyik kiterjed rá. Tehát ha ha az adott változó valamelyik  $\lambda$ -absztrakciónak a formális paramétere.
- Egy  $\lambda x.y$  kifejezésben az  $y$  változót „szabad változónak” nevezzük. Szabad változók azok, amelyeket nem köt a  $\lambda$ -kifejezés (az  $x$  az ebben az esetben is kötött), pl. látványos, hogy bármilyen  $x$  értékre alkalmazzuk ezt a kifejezést, az eredmény minden nem kötött változó szabad.

Az, hogy egy változó kötött-e, egyszerűen leolvasható a gráfról is. Amennyiben a változót (ami legyen  $x$ ) képviselő levélelemből kiindulva, felfelé haladva a fán, található egy  $\lambda x$  csomópont, akkor az a változó kötött és az a  $\lambda x$  csomópont köti, amelyik hozzá legközelebb található. Amennyiben ilyen nincs, akkor a változó szabad. A változók közti különbségek a különféle átalakítási szabályok során kapnak fontos szerepet.

## Helyettesítések, átalakítások

Egy funkcionális nyelv esetében a kezdő kifejezést különböző átalakításokkal lehet más-más, tipikusan egyszerűbb formára hozni. A  $\lambda$ -kalkulus esetén ennek alaplépése a helyettesítés. Ha egy  $E\lambda$  kifejezésben egy szabad  $x$  változót mindenütt helyettesítünk egy  $F$   $\lambda$ -kifejezéssel, akkor az így kapott  $\lambda$ -kifejezést így jelöljük:  $E[x := F]$ .

A helyettesítések segítségével különböző átalakítási szabályokat lehet megfogalmazni:[9]

- $\alpha$ -konverzió: belátható, hogy egy  $\lambda x.x$  tulajdonképpen egyenlő a  $\lambda y.y$   $\lambda$ -kifejezéssel. Ezt azt is jelenti, hogy a kötött változók átnevezése bármikor megtehető (betartva bizonyos szabályokat, pl. nyilván nem adható meg olyan változónév, ami már egyéb célra használt), ezzel ekvivalens kifejezésekhez jutunk. Az azonos változónevek kezeléséhez ez egy hasznos lépés lesz.

A helyettesítést használva:  $\lambda x.E \rightarrow_\alpha (\lambda y.E[x := y])$

Tehát  $(\lambda x.x * x) \rightarrow_\alpha (\lambda y.y * y)$

- $\beta$ -redukció: azt határozza meg, hogy egy függvény applikáció eredménye az eredeti  $\lambda$ -absztrakció olyan módosítása, ahol a formális paraméter minden előfordulása ki van cserélve az applikáció aktuális paraméterével.

A helyettesítést használva:  $(\lambda x.E)A \rightarrow_\beta E[x := A]$

Tehát  $(\lambda x.x * x) 2 \rightarrow_\beta 2 * 2$  Ez nem változtathatja meg a kifejezés értékét, emiatt figyelni például arra, hogy a szabad változók a helyettesítés után is szabad változók maradjanak. Emiatt a  $\beta$ -redukció gyakran együtt jár az  $\alpha$ -konverzióval, hogy tudja biztosítani az átalakítás helyességét.

- $\delta$ -redukció: a korábbi példákban nem foglalkoztunk vele, hogy miként lehet megvalósítani a + vagy \* műveleteket. Amennyiben ezeket külső, eleve adott műveleteknek tekintjük, akkor ezek használatát tekintjük a  $\delta$ -redukciós lépéseknek.

A különböző konverziós szabályok segítségével a  $\lambda$ -kifejezések átalakíthatók, egyszerűbb formára hozhatók. Az átalakítható kifejezéseket nevezzük „redex”-nek<sup>4</sup>. Egy funkcionális program futása ezen kifejezések lehető legegyszerűbb alakra hozását jelenti. A további már nem egyszerűsíthető  $\lambda$ -kifejezésre mondhatjuk, hogy normál formában van. De további feladatokat is meg lehet így oldani, például két kifejezés azonos, ha létezik az átalakítások egy olyan véges sorozata, amely az egyik a másikba visz át.

A  $\delta$ -redukció használata a gyakorlatban elhagyható. Messze túlmutat ezen jegyzet keretein, de a  $\lambda$ -kalkuluson alapulva be lehet vezetni a funkcionális programozáshoz szükséges alapvető elemeket: igazságértékek (igaz, hamis), feltételes kifejezések, logikai operátorok, számok, stb. Ez persze fordítva is igaz, ennek segítségével a magasabb szintű funkcionális nyelven megírt program minden eleme visszavezethető ezekre az elemi  $\lambda$ -kifejezésekre. Amelyeket a fent megismert szabályokkal lehet tovább kezelní.

<sup>4</sup>reducible expression

## 2.2.6. Értékelés

A funkcionális nyelvek nem fogják felváltani az imperatív nyelveket, de erre nincs is szükség, a két paradiigma nagyszerűen kiegészíti egymást.

A mellékhatásoktól mentes felépítés nagy segítséget jelent a hatékony végrehajtás ügyében:

- mivel egy kifejezés értéke minden időpontban ugyanaz lesz, így a kiértékelések sorrendjénél nem kell ragaszkodni a programozó által megadott sorrendhez, hanem mindenkor mindenkor lehető leghatékonyabb sorrendet lehet használni;
- ez gyakran azt is jelenti, hogy a kiértékelendő függvény egyes részeit akár párhuzamosan, több processzorral értékeli ki a végrehajtó motor;
- a mellékhatásoktól mentes programok tervezése, implementálása, tesztelése és hibakeresése is jóval egyszerűbb.

A megismert végrehajtási mód is számos előnyt jelent:

- megoldhatók vele olyan jellegű feladatok, amelyek az imperatív nyelvekkel nem (azonosság ellenőrzés, automatikus bizonyítás, stb.);
- lehetőséget nyújt a program helyesség biztosítására, ami az imperatív nyelvekkel tesztekkel való ellenőrzéséhez képest jóval nagyobb biztonságot ad.

A módszer persze sok esetben nem használható hatékonyan, például amikor intenzív kimenet/bemenet kapcsolatra vagy éppen időtől függő állapotokra van szükség. Pl. szövegszerkesztőt és játékprogramot valószínűleg nem érdemes funkcionális nyelveken írni.

## 2.3. Logikai programozás

### 2.3.1. Logikai program alapelemei

A logikai programozás a funkcionális programozás egy alágának is tekinthető. Annál jóval szűkebb területet fed le, mivel alapvetően az elsőrendű logika lehetőségeire épít. Valójában még annál is csak egy szűkebb területet fed le, az úgynevezett „Horn-logikát” képes kezelní, számos, az elsőrendű logikával megfogalmazható problémát nem lehet megoldani vele. A logikai programozás területén jóval kevesebb nyelv terjedt el, mint a funkcionális nyelveknél, ezért ebben a jegyzetben is többnyire a Prolog nyelv jellegzetességeit vesszi alapul.

A logikai programozás során a programkód valójában logikai állítások sorozata. Míg a funkcionális nyelvek a függvény fogalomból építkeztek, addig itt a reláció fogalma jelenik meg.

#### Program alapvető elépítése

A változó fogalom itt a funkcionális nyelvekben már megismert, a matematikai változó fogalomnak felel meg. Ezek értéke időtől független, tehát nincs többszörös értékadás, így mellékhatások se jelenhetnek meg. Ennek előnyei is hasonlóak, a logikai nyelven készített programok nagyon jól párhuzamosíthatók.

#### Szabályok és tények

A logikai programok fő alkotóelemei a szabályok (ezek itt többé-kevésbé megfelelnek a más paradigmákban már megismert függvényeknek.). Ez tartalmazza a szabály nevét, illetve egy hozzá tartozó kifejezést. Az elsőrendű logikával egészen összetett szabályokat is meg lehet fogalmazni, ezeket az egyszerűbb logikai nyelvek nem tudják kezelní, hanem a lehetséges állításoknak csak egy szűkített körét lehet megfogalmazni.

Ezen szűkítés eredménye, hogy a szabályokat úgynevezett „Horn-klázok” segítségével kell megfogalmazni. A Horn klázok általános alakja:

$$x \leftarrow (a_1 \wedge a_2 \wedge \dots \wedge a_n) \forall x, \forall a_1, \forall a_2, \dots \forall a_n \quad (2.1)$$

Ahol a  $\leftarrow$  jelen esetben megfelel a matematikából már ismert implikációt, az  $x$ , illetve az  $a_i$ -k pedig elemi állítások, ez utóbbiakat literáloknak is nevezik.

Egy szabály általános alakja így írható le:  $x := a_1 \wedge a_2 \wedge \dots \wedge a_n$

Ahol az  $x$  a szabály feje, a jobb oldali rész pedig pedig a szabály törzse, amely a  $a_i$  feltételeket tartalmazza. A szabály értelmezése pedig a matematikából már ismert implikációt felel meg: ha  $a_1, a_2, \dots, a_n$  feltételek közül mindegyik igaz, akkor az  $x$  is igaz.

A minél egyszerűbb kiértékelő motor miatt a szabályok alapvetően csak *logikai* és kapcsolatokat tartalmazhatnak. Erre mutat példát a 2.11. algoritmus. Az első sor meghatározza, hogy  $x$  nagyszülője  $y$ -nak, ha létezik olyan  $z$ , amelyre igaz, hogy  $x$  szülője  $z$ -nek,  $z$  pedig szülője  $y$ -nak. A szabály a matematikai implikációt felel meg, tehát nincs „csak akkor” jelentése.

Néha szükség lehet a tagadás műveletre is, mint például a testvér kapcsolatban:  $x$  és  $y$  testvérek, ha van olyan  $z$ , amelyik mindkettőnek szülője és persze senki se lehet önmaga testvére. Ez azonban nem minden esetben használható, a Prolog például nem tudja kezelní a negációt.

---

#### 2.11. Algoritmus Logikai szabályok meghatározása (vagy kapcsolat)

---

- 1: NagySzülő( $x, y$ ) := Szülő( $x, z$ )  $\wedge$  Szülő( $z, y$ )
  - 2: Testvér( $x, y$ ) := Szülő( $z, x$ )  $\wedge$  Szülő( $z, y$ )  $\wedge$   $\neg(x = y)$
- 

A *logikai vagy* kapcsolat nem szerepelhet a szabályokban, azonban a funkcionális nyelveknél megjelenő mintaillesztés segítségével ez is egyszerűen megvalósítható. Erre mutat példát a 2.12. algoritmus, amelyből látható a szülőség definíciója. Egy  $x$  szülője az  $y$ -nak akkor, ha  $x$  anyja az  $y$ -nak, illetve akkor is, ha  $x$  apja az  $y$ -nak.

A tények tulajdonképpen nulla változós szabályoknak tekinthetők. Tehát olyan szabályok, amik minden teljesülnek. Ezekre mutat néhány példát a 2.13. algoritmus.

---

## 2.12. Algoritmus Logikai szabályok meghatározása (és kapcsolat)

---

- 1:  $\text{Szülő}(x, y) := \text{Anyá}(x, y)$
  - 2:  $\text{Szülő}(x, y) := \text{Apa}(x, y)$
- 

## 2.13. Algoritmus Tények meghatározása

---

- 1:  $\text{Apa}("Laci", "Pisti")$
  - 2:  $\text{Anyá}("Mari", "Pisti")$
  - 3:  $\text{Anyá}("Mari", "Géza")$
  - 4:  $\text{Apa}("Lajos", "Mari")$
- 

Itt láthatóan nincsenek feltételek, tehát pl. az első sorban nincs annak feltétele, hogy "Laci" a "Pisti" apja lenne. Ez tehát egy tény.

A funkcionális nyelvekhez hasonlóan itt sincs szerepe a programsorok sorrendjének. Az egyes szabályok és tények teljesen egyenrangúak, közöttük semmilyen időbeli kapcsolat nincs.

### Célkifejezés

A logikai program működését egy célkifejezés megadásával lehet elindítani. Ez a célkifejezés alapvető kétfélé lehet:

- Amennyiben a célkifejezés nem tartalmaz változókat, csak konstansokat, akkor a rendszernek azt kell eldöntenie, hogy a megadott kifejezés levezethető-e a megadott tények és szabályok alapján.
- Amennyiben változók is megjelennek a célkifejezésben, akkor a rendszer megpróbál keresni olyan értékeket, amelyeket behelyettesítve a változó helyére, igaz állításhoz jut.

A fenti példa szabályok és tények ismeretében például az alábbi kérdéseket tudja megválaszolni a rendszer:

- $\text{Apa}("Laci", "Pisti")?$  ("Laci" a "Pisti" apja?  $\rightarrow \text{igen}$ )
- $\text{Anyá}("Lajos", "Pisti")?$  ("Lajos" a "Pisti" anyja?  $\rightarrow \text{nem}$ )
- $\text{Szülő}("Laci", "Pisti")?$  ("Laci" a "Pisti" szülője?  $\rightarrow \text{igen}$ )
- $\text{Nagyszülő}("Lajos", "Géza")?$  ("Lajos" a "Géza" nagyszülője?  $\rightarrow \text{igen}$ )
- $\text{Testvér}("Lajos", "Mari")?$  ("Lajos" a "Mari" testvére?  $\rightarrow \text{nem}$ )
- $\text{Szülő}("Mari", ?)$  ("Mari" ki(k)nek a szülője?  $\rightarrow "Pisti", "Géza"$ )
- $\text{Szülő}(?, "Pisti")$  (ki(k) "Pisti" szülei?  $\rightarrow "Laci", "Mari"$ )
- $\text{Testvér}(?, ?)$  (ki(k) testvérek?  $\rightarrow ("Pisti", "Géza"),$  illetve ("Géza", "Pisti"))

### 2.3.2. Értékelés

A logikai nyelvek nyilvánvalóan deklaratív eszközök, hiszen itt már teljesen elkülön egymástól a feladat meghatározása (a szabályok megadásával) és a program futásának mikéntje (amit maga a megoldó motor hajt végre a megismert módszerek egyikével). Sok egyéb hasonlóság is látható a funkcionális nyelvekkel összehasonlítva, az így készített programok hasonlóan magas absztrakciós szintet képviselnek, illetve azok futtatása jól optimalizálható párhuzamos környezetben. A két paradigma működési elve is alapvetően hasonló, hiszen nem konkrét számításokat végeznek egy megadott utasítássorozatnak megfelelően, hanem átalakítások segítségével próbálják megtalálni a végeredményt.

Lényeges különbség azonban a hatékonyan megoldható feladatok köre. A funkcionális nyelvek kimondottan egy függvény kiszámítására, optimalizálására, esetleg a függvények analízisére, azokkal való

szimbolikus bizonyításokra használhatók jól. A logikai nyelvek pedig főleg tudás reprezentációra használhatók, amelyeken a szakértői rendszerek alapulnak. Ennek alapvető koncepciója az lenne, hogy egy megadott (általában meglehetősen szűk) tudományterület szabályait felírják a már megismert szabályok segítségével, majd az összes szabály felvétele után a rendszernek fel lehet tenni kérdéseket, amelyekre az az ismert szabályok alapján megpróbál válaszolni. Az elgondolás előnyei nyilvánvalók:

- a megoldó motor képes több ezernyi szabály összefüggései alapján kikövetkeztetni a végeredményt;
- sőt, nem csak egy ténysszerű választ tud adni, hanem akár az ahhoz vezető utat is, tehát a válasz bizonyítását.

## 2.4. Hibrid megvalósítások

### 2.4.1. Megváltoztathatatlan objektumok

A funkcionális nyelveknél már láthattuk, hogy milyen sok előnnyel jár, ha a változóknak nincs mellékhatásuk. Az imperatív nyelvekben ugyan ennek tipikusan az ellenkezője a jellemző, de hasonló hatást egyszerűen el lehet érni, így pedig ennek előnyei is elérhetők.

Mégha nincsenek is erre direkt programozási nyelvi elemek, akkor is egyszerűen lehet készíteni olyan osztályokat, amelyek

- minden mezője privát, tehát kívülről nem érhető el;
- az osztály adatai csak olvasható tulajdonságokon keresztül érhetők el;
- minden mező a konstruktörhíváskor kap értéket;
- sem kívülről, sem az osztályon belül nincs olyan művelet, ami megváltoztatná a mezők értékét.

Az így elkészített objektumok számos előnnyel rendelkeznek:

- állapotuk független az időtől, tehát nem okozhatnak mellékhatásokat;
- szükség esetén több referencia is hivatkozhat rájuk, mivel nem kell attól félni, hogy az egyik referencia keresztüli változtatás problémát okozna a másik referenciát használó kódban.

Ha például a komplex számok kezelésére kell létrehozni egy új osztályt, akkor érdemes átgondolni, hogy érdemes-e a fenti szabályoknak megfelelően készíteni a megvalósítást.

Az ilyen jellegű objektumok kezelése persze újszerű kihívásokat is állít, például nyilvánvalóan nem lehet módosítani ezek értékeit. Ennek egy tipikus megoldása lehet, hogy módosítás helyett új objektumok létrehozása. Például egy komplex szám növelés során nem változik meg az objektum értéke, hanem létrejön egy új, annak megnöveült adataival.

Ez a technika egyébként a keretrendszerek által is gyakran használt, sok modern OOP nyelvben a String osztály hasonló szabályok szerint létezik. Ez egy nagyon hatékony és jól bevált módja az objektumok megosztásának.

Ugyanígy fontos a technika a különféle gyűjtemények kezelésekor. Egyszerűnek tűnik egy ciklussal végigszaladni egy lista elemein, de mi történik akkor, ha menet közben megváltozik a lista valamelyik eleme? Esetleg a változást egy másik szálban futó programkód kezdeményezi? Ezeket a problémákat mind megoldja az, ha a bejárásban egy megváltoztathatatlan listát használunk.

### 2.4.2. Rekurzió

Funkcionális nyelveknél nem volt ciklus, hiszen annak időben változó iteratív működése alapvetően elengedhetetlen a paradigma alap koncepcióinak. Helyette rekurzióval lehetett megoldani számos feladatot, ami változók (és így mellékhatások) nélkül is nagyon hatékony megoldásokhoz vezetett.

Ezek a rekurzív algoritmusok természetesen az imperatív nyelvekben is használhatók. Bár a rekurzió végrehajtása egészen másként működik (egy verem segítségével), tehát minden nem lehet egy-az-egyben átalakítani, illetve a rekurzív megoldások teljesítménye se minden megfelelő, de az esetek jelentős részében választható ez az irány.

A funkcionális nyelveknél már megismert számösszeadásra a 2.14. algoritmus mutat egy már ismert pszeudokódon alapuló imperatív megoldást.

---

#### 2.14. Algoritmus 1-től n-ig számok összeadása rekurzióval.

---

Bemenet:  $n$  - egész<sup>+</sup> (egy egész szám)

Kimenet: . - egész<sup>+</sup> (az egész számok összege 0-n között)

```
1: függvény SZUMMA(n)
2: ha $n = 0$ akkor
3: vissza 0
4: különben
5: vissza SZUMMA($n - 1$) + n
6: elágazás vége
7: függvény vége
```

---

Ez is mutatja, hogy a különböző paradigmák közti határvonal nem minden éles. Többé-kevésbé egy imperatív nyelv segítségével is lehet a funkcionális paradigmában megsokott szempontok szerinti programot írni.

#### 2.4.3. Függvény referenciák megjelenése

A funkcionális nyelvekben a legnagyobb újdonság persze a függvények újszerű kezelése volt, hiszen azok az adatokhoz hasonlóan eltárolhatók, paraméterként átadhatók, illetve lehetnek függvények visszatérési értékei. Az itt megsokott elegancia és rugalmasság az imperatív nyelvekben sokáig hiányzott, bár megoldások azért léteztek:

- Alacsonyabb szintű nyelvek kihasználják a Neumann-elvű gépek azon jellegzetességét, hogy a függvények programkódja ugyanúgy a memoriában tárolódik, mint az adatokat tároló változók. Ez viszont azt is jelenti, hogy létre lehet hozni mutatókat, amelyek nem változókra, hanem függvényekre hivatkoznak, majd pedig ezeken a változókon keresztül lehet egy függvényre való referenciát eltárolni, paraméterként átadni, visszaadni vagy éppen ezen a változón keresztül a függvényt meghívni.
- Elegáns OOP megvalósítás is létezik a kérdésre, egy függvény közvetlen eltárolása helyett készíthetünk egy interfész, ami egyetlen azonos szignatúrával rendelkező metódust ír elő. Ezt követően mindenhol, ahol ezt a függvényt kell eltárolni/paraméterként átadni elég használni ezt az interfész, amikor pedig ennek konkrét értékét kell adni, akkor létre kell hozni egy objektumot, ami megvalósítja ezt az interfész és ezen keresztül a szükséges metódust is. Az interfészen keresztül pedig meghívható maga a függvény.

A fentiek működnak, de kevésbé elegánsak és rugalmasak, ezért a modern OOP nyelveket kiegészítették funkcionális jellemzőkkel is. Tehát létre lehet hozni olyan változókat, paramétereiket, függvény visszatérési értékeit, amelyek egy-egy függvényt tudnak reprezentálni.

Ennek konkrét megvalósítása programozási nyelvtől függ, a pszeudokódokban a 2.15. algoritmusban is látható módon fogjuk ezeket a lehetőségeket használni. Ez a példa egy klasszikus módon létrehozott és elnevezett függvényre való hivatkozásokat mutatja be (változónak értékkadás, függvény visszatérési értékként visszaadás, változón át meghívás, paraméterként átadást, stb.).

Nem elmerülve az elnevezési lehetőségekbe, meg fogjuk különböztetni egymástól a függvényekre és eljárásokra vonatkozó referenciákat. A különböszég pusztán annyi, hogy az előbbieknél rendelkeznek visszatérési értékkel, az utóbbiak pedig nem. A pszeudokódokban a generikus típusokhoz hasonlóan fogjuk megadni ezek paramétereit, illetve esetleges visszatérési értéküket is. Hosszas precíz definíció helyett néhány példa arra, hogy ezt miként fogjuk használni:

- **eljárás**<egész> : egy egész szám paramétert váró, visszatérési érték nélküli eljárást tárol.
- **függvény**<szöveg→egész> : egy szöveges paramétert váró, egész számot visszaadó függvény.
- **függvény**<szöveg, egész→egész> : egy szöveges és egy egész paramétert váró, egész számot visszaadó függvény.

---

## 2.15. Algoritmus Függvények használata változóként, paraméterként és visszatérési értékként.

---

Bemenet:  $x$  - egész (a négyzetre emelendő szám)

Kimenet:  $\dots$  - egész (a szám négyzete)

1: függvény NÉGYZETREEMELÉS( $x$ )

2: vissza  $x^2$

3: függvény vége

Bemenet:  $f$  - függvény $<\text{egész} \rightarrow \text{egész}>$  (egyetlen paramétere és visszatérési értéke is egész)

4: függvény MŰVELETVÉGZÉSKETTŐVEL( $f$ )

5: vissza  $f(2)$

6: függvény vége

Kimenet:  $\dots$  - függvény $<\text{egész} \rightarrow \text{egész}>$  (egyetlen paramétere és visszatérési értéke is egész)

7: függvény MŰVELETVÁLASZTÓ()

8: vissza NÉGYZETREEMELÉS

9: függvény vége

10:  $a \leftarrow$  NÉGYZETREEMELÉS

11:  $a \leftarrow$  MŰVELETVÁLASZTÓ()

12:  $b \leftarrow a(2)$

13:  $b \leftarrow$  MŰVELETVÉGZÉSKETTŐVEL(NÉGYZETREEMELÉS)

14:  $b \leftarrow$  MŰVELETVÉGZÉSKETTŐVEL( $a$ )

---

### Felhasznált változók és függvények

- $a$  : Egy függvény $<\text{egész} \rightarrow \text{egész}>$  típusú változó (egy referencia egy függvényre, aminek egyetlen paramétere és visszatérési értéke is egész).
  - $b$  : Egy egész típusú változó.
- 

A szigorúan típusos nyelvek biztonsága persze kiterjed ezekre a függvényeket tároló változókra is. minden ilyen változónál/paraméternél a már megszokott módon meg kell határozni egy pontos típust, ami függvény típus esetén azt is tartalmazza, hogy annak milyen paraméterei illetve visszatérési értéke legyen. A fordító pedig csak olyan értékadást/paraméterátadást engedélyez, ami megfelel ezeknek a szigorú feltételeknek. A pszeudokódokban erre nem vezetünk be egy teljes jelölésrendszert, egyszerűen a megjegyzésekben jelöljük a követelményeket ha nem egyértelműek.

### 2.4.4. Névtelen függvények, $\lambda$ -kifejezések

Az imperatív nyelvekben is gyakran előfordul, hogy anélkül dolgozunk memóriabeli adatokkal, hogy azokhoz változónevet társítanánk. Például függvény híváskor ha konstans paramétert adunk át (pl.  $f(1)$ ), vagy éppen egy kifejezés értékét ( $f(a + b)$  vagy  $f(a > 2)$ ). A paraméterátadás mellett ez igaz a visszatérési értékekre is (pl. vissza  $a + 2$ ), és még számos példát lehetne hozni. Tehát ez már teljesen megszokott, hogy csak annak adunk nevet, amit megadott környezetben a későbbiekben is el akarunk érni. Ha ez nem áll fent, akkor a név is szükségtelen.

A függvényekkel való munkánál ez az igény ugyanig felmerül. Az előző példában létre kellett hozni egy metódust/függvényt csak azért, hogy ezen a néven tudjunk hivatkozni a benne lévő utasításokra. Mindez megoldható metódus/függvény nélkül is, amikor csak magukat a végrehajtandó utasításokat kell megadni.

Ennek első megvalósításai a névtelen osztályok voltak, amelyekkel az előzőekben már említett interfészeken keresztül lehetett olyan metódusokat létrehozni, amelyekhez nem tartozott névvel ellátott osztály. A modern OOP nyelvekben általában megjelenik valamelyen „ $\lambda$ -kifejezés” fogalom, amely segítsével nevessített függvények nélkül lehet utasításokkal dolgozni. A név persze nem véletlenül egyezik a 2.2.5. fejezetben már tárgyalt matematikai konstrukcióval.

Ennek lehetőségei szintén programozási nyelvekként változnak, a pszeudokódokban a 2.16. algoritmusban is látható formában fogjuk ezeket a lehetőségeket használni.

---

## 2.16. Algoritmus Lambda kifejezések használata változóként, paraméterként és visszatérési értékként.

---

**Bemenet:**  $f$  - függvény $\langle \text{egész} \rightarrow \text{egész} \rangle$  (egyetlen paramétere és visszatérési értéke is egész)

- 1: függvény MŰVELETVÉGZÉSKETTŐVEL( $f$ )
- 2: vissza  $f(2)$
- 3: függvény vége

**Kimenet:** . - függvény $\langle \text{egész} \rightarrow \text{egész} \rangle$  (egyetlen paramétere és visszatérési értéke is egész)

- 4: függvény MŰVELETVÁLASZTÓ( )
  - 5: vissza  $(x \rightarrow x * x)$
  - 6: függvény vége
  - 7:  $a \leftarrow (x \rightarrow x * x)$
  - 8:  $a \leftarrow \text{MŰVELETVÁLASZTÓ}()$
  - 9:  $b \leftarrow a(2)$
  - 10:  $b \leftarrow (x \rightarrow x * x)(2)$
  - 11:  $b \leftarrow \text{MŰVELETVÉGZÉSKETTŐVEL}(x \rightarrow x * x)$
  - 12:  $b \leftarrow \text{MŰVELETVÉGZÉSKETTŐVEL}(a)$
- 

### Felhasznált változók és függvények

- $a$  : Egy függvény $\langle \text{egész} \rightarrow \text{egész} \rangle$  típusú változó.
  - $b$  : Egy egész típusú változó.
-

## **II. rész**

# **Alapvető adatszerkezetek**

## 3. fejezet

# Adatszerkezetek jellemzői

### 3.1. Abszakt adatszerkezetek

#### 3.1.1. Adatszerkezetek

Az 1.2.1. fejezetben megismertük az egyszerű és összetett típusokat, illetve a mutatókat/referenciákat. Ezeket (a tömbökkel kiegészítve, de ezekről később lesz még bőven szó) tekintjük úgy, hogy a programozási nyelvek beépített típusai, készen kapjuk őket, nincs velük további dolgunk. Ez igaz a gépi kódktól kezdve egészen a legmodernebb programozási nyeltekig.

A gyakorlatban azonban számos, ezeknél jóval fejlettebb adatszerkezet létezik, amelyek alkalmásak nagymennyiségű adat strukturált tárolására: listák, fák, gráfok, szótárak, stb. Ezek a szerkezetek nem feltétlenül részei az alap programozási nyelvnek (habár gyakran már készen elérhetők a különféle kiegészítő könyvtárakon keresztül). Ezért ezeket használhatunk előtt el kell készítenünk a meglévő egyszerű típusok, mutatók és tömbök segítségével.

Sokféle adatszerkezet valósítható meg, amelyek jól megférnek egymás mellett, nincs közük egy, univerzálisan legjobbnak mondható sem. Van, amelyiknek a tárhelyfoglalása ideális (pl. maga a tömb, aki szintén adatszerkezetnek tekinthető), amelyikben hatékonyan tudunk módosításokat végrehajtani (pl. listák), amelyikben gyorsan tudunk keresni (pl. fák), amelyikben bizonyos speciális feltételekkel még gyorsabb hozzáférést tudunk megvalósítani (pl. szótárak), illetve van, amely képes az adatok speciális egymáshoz viszonyított kapcsolatait tárolni (pl. gráfok).

#### 3.1.2. Absztrakt adattípusok

Sokféle adatszerkezetet tudunk elkészíteni, amelyek jelentősen különböznek egymástól a felhasználási módjukban (mire jó az adatszerkezet) és megvalósítási módjukban (hogyan valósítja meg ezeket a funkciókat). A különbségek mellett azonban számos átfedés is megjelenik, ezért célszerű valamilyen hierarchiát kiépíteni ezek között. Fontos azonban, hogy felismerjük, hogy a felhasználási lehetőségek és az implementációs lehetőségek egymástól függetlenek. Egy listát például meg tudunk valósítani tömbökkel, de láncolt szerkezetekkel is. Ugyanígy egy tömbbel el tudunk tárolni listát, de akár egy fát is. Ezért az adatszerkezetek tárgyalását két irányból tesszük meg.

- Absztrakt adat típus<sup>1</sup>: meghatározza, hogy egy adat típus milyen műveletekkel rendelkezik, és ezen keresztül milyen általános jellemzőket várunk el tőle. Felhasználói oldalról elsődlegesen ez fontos számunkra, hiszen ezzel találkozik a programozó, ez alapján dönti el, hogy adott feladathoz milyen adat típusra van szüksége.
- Fizikai megvalósítás: meghatározza, hogy az adott adat típus miként képezi le az adatokat a memóriában tárolható bitek sorozatává, illetve miként valósítja meg az adott adat típuson végezhető műveleteket.

Valójában már eddig is találkoztunk ezekkel a fogalmakkal, például egy *egész* típus esetében:

---

<sup>1</sup>ADT - Abstract Data Type

- Absztrakt jellemzők: a matematikából is már ismert egész számokat (egy megadott tartományon belül) tudjuk eltárolni az ilyen típusú változókban, és azokra az alapvető aritmetikai műveleteket tudjuk elvégezni.
- Implementációs jellemzők: programozási nyelvtől függően 8-16-32-64 biten tudjuk ezeket eltárolni, tipikusan a már tanult kettes számrendszerbeli formában leírva (bár itt is lehetnek különbségek azügyben, hogy a kis vagy nagy helyiértékű biteket tároljuk el elsőként).

Még érdekesebb lehet a *szöveg* típus esetében ez a kettőség:

- Absztrakt jellemzők: a típus egy megadott hosszúságú szöveget tud eltárolni. A vele végezhető műveletek: egy karakter lekérdezése, összehasonlítás, összefűzés, stb.
- Implementációs jellemzők: programozási nyelvtől függően sokféle megvalósítás létezik: 0-ra végződő karaktersorozat (C stílus), 0. pozíció tartalmazza a szöveg hosszát (Pascal stílus), a String maga egy objektum (C#, Java), stb.

A fenti példák minden beépített típusokról szóltak, ahol nincs különösebb választási lehetőség, azt tudjuk csak használni, amit a programozási nyelv megenged. Az adatszerkezetek esetében viszont látni fogjuk, hogy sokkal rugalmasabb megoldásokat találhatunk. Például egy halmaz típus esetében:

- Absztrakt jellemzők: **T** típusú elemeket tud tárolni, alapvető műveletei: berakunk egy elemet, kiveszünk egy elemet, ellenőrizzük, hogy adott elem benne van-e a halmazban.
- Implementációs jellemzők: a halmazt megvalósíthatjuk egyszerű tömb típussal, rendezett tömbbel, láncolt listával, bináris keresőfával, stb. Az egyes műveletek megvalósításai értelemszerűen a választott fizikai adatszerkezethez kapcsolódnak.

### 3.1.3. Objektum-orientált megvalósítás

Az egyes adatszerkezetek közötti hierarchiát, illetve az előzőekben realizált kettősséget érdemes a tényleges programozási nyelvekben is megjeleníteni, ezzel is segítve az elkészült adatszerkezeteket tartalmazó könyvtár áttekinthetőségét és a kódok újrafelhasználhatóságát. Manapság már alapértelmezettnek tekinthetjük az OOP használatát, ezért adatszerkezeteinket is ennek a szemléletnek megfelelően tervezük meg.

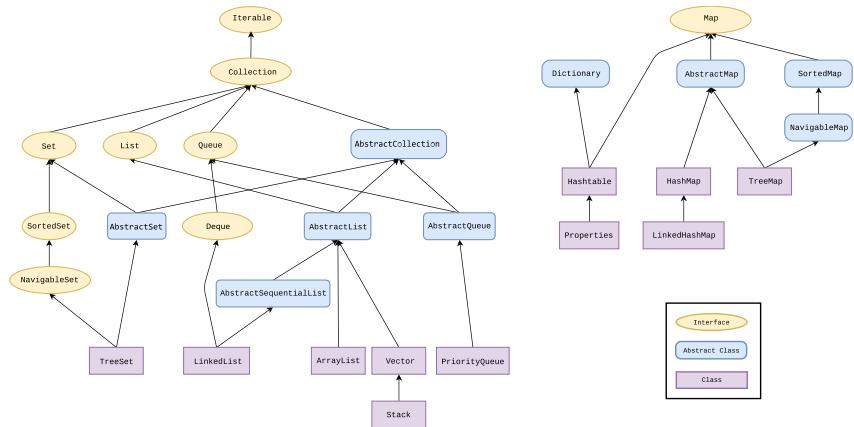
Az egyes adatszerkezetek közötti hierarchiát jól képviseli majd az OOP-ben megszokott öröklés fogalom. Az egymáshoz hasonló adatszerkezetek implementációi gyakran egy közös ős ből származnak majd, és az egyes konkrét implementációk már csak az itt megjelent újdonságokat fogják tartalmazni.

A fenti kettősséget (absztrakt és fizikai jellemzők) pedig az interfések segítségével fogjuk tudni egyszerű és elegáns módon leírni. Az absztrakt jellemzők alapvetően azt határozzák meg, hogy milyen műveleteket tudunk végrehajtani az egyes adatszerkezeteken, viszont nem szeretnénk befolyásolni vagy meghatározni annak tényleges megvalósítását. Ez tökéletesen megfelel az interfések használati esetének, ezért az alábbi felosztást fogjuk használni:

- Absztrakt jellemzők: interfések segítségével fogjuk leírni az egyes szerkezetekre vonatkozó műveleti követelményeket.
- Implementációs jellemzők: konkrét osztályokkal, illetve azok egy hierarchiájával fogjuk megvalósítani az egyes lehetséges implementációkat.

Ezzel egy rugalmas, áttekinthető, OOP szempontokból tökéletes rendszerhez fogunk jutni.

Egyébként nem meglepő, hogy az egyes programozási nyelvek általában már rendelkeznek adatszerkezeteket tartalmazó könyvtárral, és ezek felépítése általában kísértetiesen hasonlít is a fent javasolt hoz. A 3.1. ábra mutatja például a Java Collection Framework felépítését. Jól látható, hogy a legmagasabb szinteken interfések határozzák meg az elvárt műveleteket, amelyeket aztán absztrakt, illetve nem absztrakt osztályok valósítanak meg.



3.1. ábra. A Java gyűjteményeket tartalmazó osztálykönyvtára.

## 3.2. Alapvető absztrakt adatszerkezetek

### 3.2.1. Általános jellemzők

Megismerve a generikus osztályokat, célszerű az adatszerkezeteket is ilyen formában megvalósítani. A későbbiekben az alábbi típusparamétereket fogjuk használni:

- **T** típus képviseli majd minden az eltárolni kívánt adat típusát. Ezzel kapcsolatban semmilyen megkötésünk nincs, általában tetszőleges típust el tudunk tárolni.
- **K** típus képviseli majd minden az eltárolni kívánt adathoz tartozó kulcs típusát. A kulcsok olyan egyedi azonosítók, amelyek egyértelműen azonosítanak egy adatot.
- **eljárás<T>** típus egy olyan eljárást képvisel majd, ami paraméterként egy **T** paramétert vár. Ezt majd a bejárásnál<sup>2</sup> fogjuk használni, aminek az a célja, hogy az adatszerkezet minden elemét egyesével adja át paraméterként egy ilyen függvényhívásnak.

A típusparamétereknél az alábbi megszorításokat fogjuk használni szükség esetén:

- **T**; ahol **T** összehasonlítható : Ebben az esetben elvárjuk, hogy minden **x** és **y** **T** típusú változók összehasonlíthatóak legyenek, tehát két ilyen típusú adatra értelmezhetők legyenek a szokásos reláció operátorok:  $=$ ,  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ .

#### Megjegyzés

Fontos megjegyeznünk, hogy nincs szabvány az adatszerkezetek felépítésére és azok csoporthoz köthető módjára, miként az különbözik is a különböző megvalósítások során. A lenti lista ilyen szempontból önkényes, jelentős szakirodalmi kutatás, illetve a gyakorlati programozási nyelvekből nyert tapasztalatokon alapul.

Ugyanez vonatkozik az egyes adatszerkezeteknél definiált műveletekre. Ezek száma egyszerűen bővíthető, de akár el is hagyhatók belőle elemek. A **Lista** esetében például a végére fűzés művelet elhagyható, hiszen ugyanezt el lehet érni a beszúrás segítségével is (ahol indexként az utolsó elemet adjuk meg). De ugyanígy bevezethetnénk egy új elejére beszúrás műveletet is. De a fentieken túl még sokféle műveletet tudnánk kitalálni („ElemSzám”, „Üres-e”, „Benne-van-e”, „MindenTörlése”, stb.), ezek jellemzője viszont az, hogy többnyire nagyon egyszerűen megvalósíthatók, vagy visszavezethetők a már meglévő műveletekre. Ezért ezeket külön nem tárgyaljuk.

Végül megjegyeznénk, hogy az absztrakt adatszerkezetek és a fizikai megvalósítások a gyakorlatban sokkal kevésbé különülnek el egymástól, mint ahogy eddig felvezettük. Sok programban közvetlenül használjuk a tömböt mint adatszerkezet, viszont ugyanez a tömb részt vesz sok másik adatszerkezet implementációjánál. Ugyanígy a gráfot tekintetük egy absztrakt adatszerkezetnek a maga műveleteivel, de egy fizikai megvalósítást is láthatunk benne.

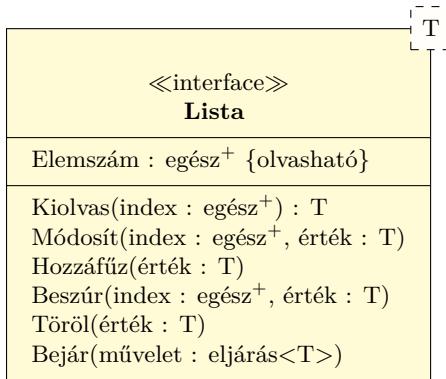
### 3.2.2. Lista

Az egyik legegyszerűbb és legsokoldalúbban használható adatszerkezet a **Lista**<sup>3</sup>. Ez véges számú adat szekvenciális, sorrendet megőrző tárolását teszi lehetővé. Egy elem többször is szerepelhet, ezért külön kulcsot itt nem is különbözünk meg. Specialitása, hogy általában támogatott a beszúrásnak megfelelő sorrendben számított *i*. elem elérése, ezt nevezzük indexelésnek.

A gyakorlatban emiatt ugyanazokban a szerepekben használható, mint a hagyományos tömb, azonban annál jóval rugalmasabban, mivel ezt változó méretű adatszerkezetnek tekintjük, így nem kell előre meghatározni a méretét.

<sup>2</sup>traversal, enumeration

<sup>3</sup>list



A **Lista** egy **T** típusparaméterrel rendelkezik, amely a benne tárolható elemek típusát határozza meg. Ezt célszerű generikus típusokkal megvalósítani, amit **Lista<T>** módon jelölünk.

Továbbá az alábbi tulajdonságot és műveleteket írja elő:

- **Elemszám**: Egy csak olvasható tulajdonság, ami visszaadja a listában lévő elemek számát.
- **KIOLVAS(index)**: Visszaadja az *index*. helyen lévő elem értékét. Amennyiben az *index* nem értelmezhető, “Hibás index” hibát jelez.
- **MÓDOSÍT(index, érték)**: Felülírja a lista *index*. helyen lévő elem értékét a második paraméterként átadott *érték*-kel. Amennyiben az *index* nem értelmezhető, “Hibás index” hibát jelez.
- **HOZZÁFŰZ(érték)**: Felveszi a paraméterként átadott *érték*-et a lista végére, ezzel eggyel növekszik a lista mérete.
- **BESZÚR(index, érték)**: Felveszi a paraméterként átadott *érték*-et a megadott helyre (a beszúrás után az *index*. elem lesz a megadott érték), ezzel eggyel növekszik a lista mérete.
- **TÖRÖL(érték)**: Megkeresi és töri a paraméterként megadott *érték*-et a listából. Amennyiben ez többször is szerepel, akkor minden előfordulást töröl.
- **BEJÁR(művelet)**: A lista minden elemére meghívja a paraméterként átadott *művelet*-et.

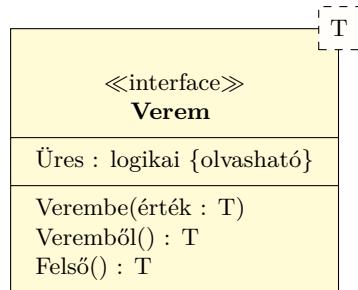
### 3.2.3. Verem

A **Verem**<sup>4</sup> egy jóval specializáltabb adatszerkezet, bizonyos esetekben viszont nagyon hatékonyan használható. A listához hasonlóan szekvenciálisan tárolja az adatokat, azokhoz viszont nem lehet közvetlenül hozzáférni. Ehelyett támogat egy verembe helyezés és kivétel műveletet. Jellegzetessége, hogy a kivétel során minden elemet kapjuk vissza a veremben lévők közül, amelyiket legutoljára raktuk be. Erre utal rövidített elnevezése is: LIFO (last-in first-out).

A működése tehát hasonló, mintha papírapokat helyeznénk egymásra egy asztalon. Ha egymás után lerakjuk az asztalra a  $p_1$ ,  $p_2$ ,  $p_3$  papírokat, akkor azokat pont fordított sorrendben tudjuk felemelni  $p_3$ ,  $p_2$ ,  $p_1$  laponként. Ahogy említettük, a **Verem** nem támogatja az elemek közvetlen elérését, tehát nem tudjuk közvetlenül kivenni a középső elemet.

Informatikához közelebbi példa a függvények hívásakor a visszatérési címek eltárolása. Ha egy függvény kódjából meghívunk egy másikat, akkor egy veremben tudjuk eltárolni, hogy a hívás után hol kell folytatni a program futását. Amennyiben az egyes függvényhívásokat egymásba ágyazzuk ( $f$  függvény meghívja  $g$ -t, ami a kódjából meghívja  $h$ -t), akkor a verem alapján könnyen nyomon követhető, hogy a visszalépésekkor hova kell visszatérni.

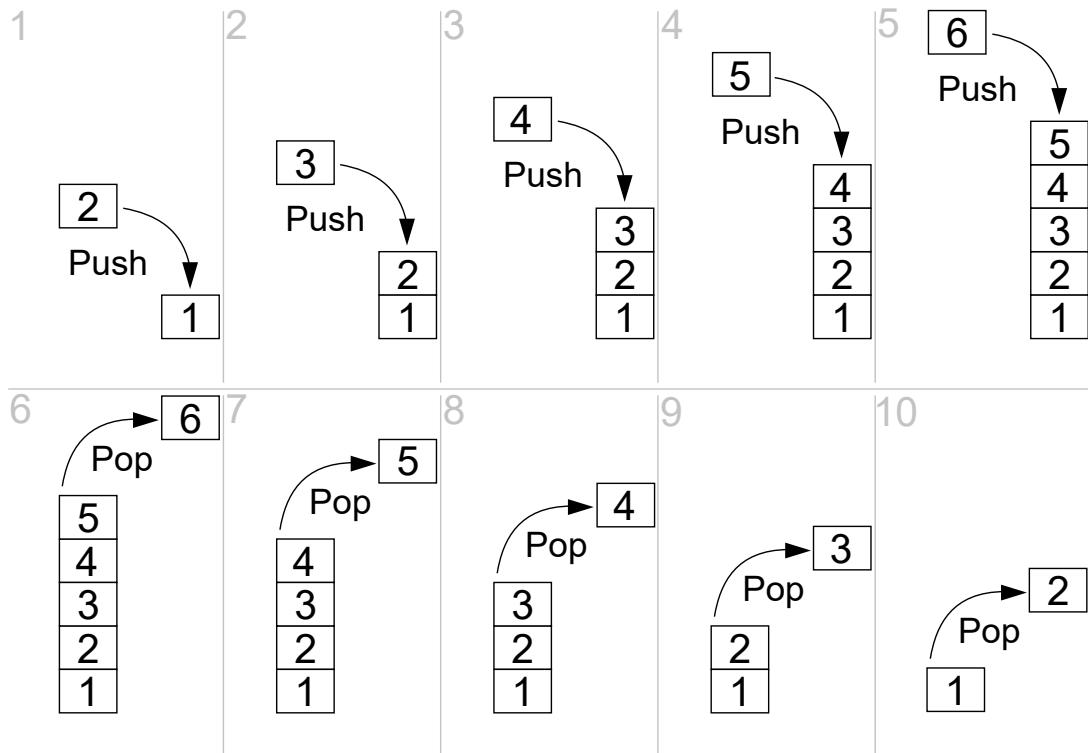
<sup>4</sup>stack, LIFO



A **Verem**<**T**> egy **T** típus paraméterrel rendelkezik, amely a benne tárolható elemek típusát határozza meg.

Továbbá az alábbi tulajdonságot és műveleteket írja elő:

- **Üres**<sup>5</sup>: Visszatérési értéke egy logikai érték: *igaz* ha üres a verem, és *hamis* ha van(nak) benne elem(ek).
- **VEREMBE**(érték)<sup>6</sup>: Berakja a paraméterként átadott *érték*-et a verembe. Amennyiben ezzel a veremben lévő elemek száma több lenne, mint a verem maximális mérete, akkor "Nincs hely" hibát jelez, és nem helyezi el az értéket az adatszerkezetben.
- **VEREMBŐL()**<sup>7</sup>: Kiveszi a veremből a legutoljára berakott *érték*-et, és ez egyben a függvény visszatérési értéke is. Amennyiben a veremben éppen nincsenek elemek, akkor "Nincs elem" hibát jelez.
- **FELSŐ()**<sup>8</sup>: Visszaadja a verem legutoljára berakott elemét, azt azonban nem törli a veremből. Amennyiben a veremben éppen nincsenek elemek, akkor "Nincs elem" hibát jelez.



3.2. ábra. A verem működésének bemutatása. A 2,3,4,5,6 elemek verembe helyezése után a kivétel sorrendje: 6,5,4,3,2. (Forrás: Wikipedia, „Stack\_(abstract\_data\_type)” szócikk)

<sup>5</sup>empty

<sup>6</sup>push

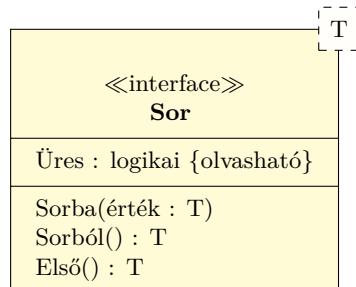
<sup>7</sup>pop

<sup>8</sup>top

### 3.2.4. Sor

A **Sor**<sup>9</sup> adatszerkezet hasonló a veremhez, ugyanis ez is szekvenciálisan tárol elemeket, amelyeket egyesével tudunk elhelyezni benne, és a veremhez hasonlóan ez sem engedélyezi a közbülső elemekhez való hozzáférést. A lényegi különbséget csak az jelenti, hogy a **Sor** esetében a kiolvasás művelet ugyanabban a sorrendben adja vissza az elemeket, amilyen sorrendben azokat beraktuk. Erre utal rövidített elnevezése is: FIFO (first-in first-out).

Erre a működési módra számtalan gyakorlati példát tudunk találni, hiszen az ehhez hasonló működést a valóságban is sorokkal tudjuk megoldani. Például egy jegypénztár előtt álló emberekből álló sor ideális esetben úgy működik, hogy aki előbb állt be a sorba, az előbb fogja megkapni az igényelt szolgáltatást. De számos kevésbé kézzelfogható sor is létezik, például ha egyetemi tárgyfelvételnél engedélyezhetjük a túljelentkezéseket. A késve jelentkező hallgatókat szintén egy sorban célszerű tárolni, hiszen ha valamilyen okból felszabadul egy hely a tárgyon, akkor a sorból elsőként kiolvasott (tehát azt, aki a benne lévők közül a legkorábban jelentkezett) hallgatót szeretnénk felvenni.



A **Sor**<**T**> egy **T** típus paraméterrel rendelkezik, amely a benne tárolható elemek típusát határozza meg.

Továbbá az alábbi tulajdonságot és műveleteket írja elő:

- **Üres**<sup>10</sup>: Visszatérési értéke *igaz* ha üres a sor és *hamis* ha van(nak) benne elem(ek).
- **SORBA**(érték)<sup>11</sup>: Berakja a paraméterként átadott *érték*-et a sorba. Amennyiben ezzel a sorban lévő elemek száma több lenne mint a sor maximális méreténél, akkor "Nincs hely" hibát jelez, és nem helyezi el az értéket az adatszerkezetben.
- **SORBÓL()**<sup>12</sup>: Kiveszi a sorból a legrégebben belerakott még benne lévő *érték*-et, és ez egyben a függvény visszatérési értéke is. Amennyiben a sorban éppen nincsenek elemek, akkor "Nincs elem" hibát jelez.
- **ELSŐ()**<sup>13</sup>: Visszaadja a sorban lévő legrégebben belehelyezett elemet, azt azonban nem törli. Amennyiben a sorban éppen nincsenek elemek, akkor "Nincs elem" hibát jelez.

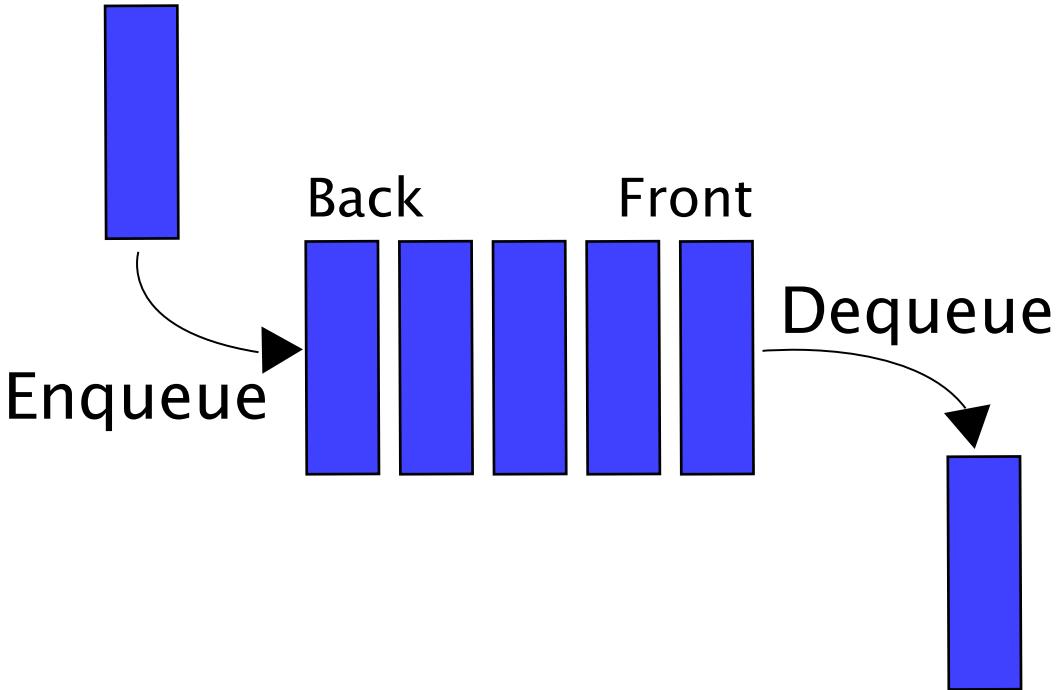
<sup>9</sup>queue, FIFO

<sup>10</sup>empty

<sup>11</sup>enqueue

<sup>12</sup>dequeue

<sup>13</sup>first



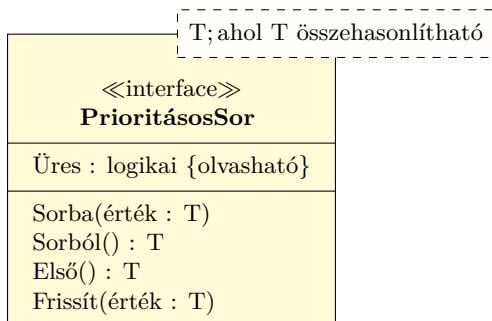
3.3. ábra. A sor működésének bemutatása. (Forrás: Wikipedia, „Queue\_(abstract\_data\_type)” szócikk)

### 3.2.5. Prioritásos sor

A **PrioritásosSor**<sup>14</sup> (más néven elsőbbségi sor) abban különbözik a hagyományos sortól, hogy az egyes elemekhez hozzárendel egy prioritás értéket is. A sorba történő új elem felvételekor tehát csak olyan **T** típusú tartalom jelenhet meg, amely tartalmazza a prioritás hozzárendeléshez szükséges műveleteket. Kiolvásás a sorhoz és veremhez hasonlóan csak egy műveleten keresztül történhet, ez pedig minden visszaadja a maximális prioritással bíró elemet, amely aktuálisan a prioritási sorban található.

A gyakorlatban ez a működés szintén megfigyelhető, például egy kórház sürgősségi osztályán. Míg egy gyorsítteremben az egyes várakozókat azonos prioritásúnak tekintjük, ezért itt a hagyományos sor célszerű használni, amely csak az érkezési sorrendet veszi figyelembe, addig egy sürgősségi osztályon a beérkező betegeket első lépésként értékelik, hogy kinek mennyire sürgős beavatkozásokra van szüksége, és a későbbiekben ez alapján történik a betegek ellátása.

De ugyanezt az adatszerkezetet használják a többszálú rendszerek bizonyos ütemezési módszertanoknál, ahol minden prioritású szálak futtatását kell támogatni a kisebb prioritásúakkal szemben.



A **PrioritásosSor**<**T**> tehát egy **T** összehasonlítható típusparaméterrel rendelkezik, amely a benne tárolható elemek típusát határozza meg. Azért adjuk meg az összehasonlíthatóság megszorítást, mivel ez alapján fogjuk meghatározni az egyes elemek prioritását. Feltételezzük, hogy az adatszerkezet minden

<sup>14</sup>priority queue

eleme összehasonlítható a többivel, és az összehasonlítás eredménye egyben azt is megadja, hogy melyiknek nagyobb a prioritása. Ami egyben azt is jelenti, hogy magát a prioritási értéket mint olyat nem is fogjuk kezelni, annak tárolása

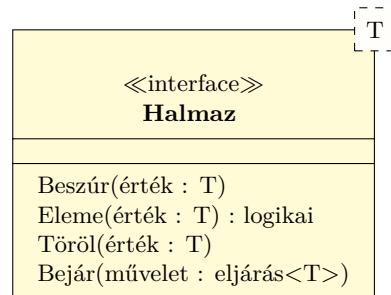
Továbbá az alábbi tulajdonságot és műveleteket írja elő:

- **Üres**<sup>15</sup>: Visszatérési értéke *igaz* ha üres a sor és *hamis* ha van(nak) benne elem(ek).
- **SORBA**(érték)<sup>16</sup>: Berakja a paraméterként átadott *érték*-et a sorba. Amennyiben ezzel a sorban lévő elemek száma több lenne, mint annak maximális mérete, akkor "Nincs hely" hibát jelez, és nem helyezi el az értéket a sorban.
- **SORBÓL()**<sup>17</sup>: Kiveszi a sorból a maximális prioritással bíró *érték*-et, és ez egyben a függvény visszatérési értéke is. Amennyiben a sorban éppen nincsenek elemek, akkor "Nincs elem" hibát jelez. Amennyiben több elem prioritása is megegyezik a maximálissal, akkor ezek közül bármelyiket visszaadhatja.
- **ELOSÓ()**<sup>18</sup>: Visszaadja a sorban lévő maximális prioritással bíró *érték*-et, azt azonban nem töri. Amennyiben a sorban éppen nincsenek elemek, akkor "Nincs hely" hibát jelez.
- **FRISSÍT**(érték)<sup>19</sup>: Előfordulhat, hogy megváltozik egy, már a sorban lévő elem prioritása, ilyenkor a sort aktualizálni kell. Erre szolgál ez a metódus, az *érték* paraméteren keresztül kell megadni a megváltozott értéket.

### 3.2.6. Halmaz

A **Halmaz**<sup>20</sup> adatszerkezet alaptulajdonságai megegyeznek a matematikából már ismert fogalommal (azzal a lényeges különbséggel, hogy itt csak véges méretű halmazokról beszélhetünk). Ennek megfelelően tetszőleges típusú diszkrét értékeket el tudunk benne tárolni az alábbi specialitásokkal: az elemek között nincs kapcsolat, így sorrendről se beszélhetünk; nincsenek ismétlődő elemek, tehát egy érték csak egyszer szerepelhet a halmazban.

Gyakorlati példa lehet a halmazok alkalmazására a hallgatók méltányossági kérelmeinek kezelése, ami ből mindenki pályafutása alatt csak egyszer részesülhet. Ennek megoldásához létrehozunk egy kezdetben üres halmazt. minden kérelem támogatásakor elhelyezzük a hallgatót ebben a halmazban. minden kérelem leadásakor ellenőrizzük, hogy a hallgató benne van-e már ebben a halmazban. Jól látszik, hogy erre a céllra tökéletesen megfelel ez az egyszerű adatszerkezet is: a hallgatók között nincs semmilyen kapcsolat, nem számít a sorrendjük, egy hallgató csak egyszer kerülhet a halmazba, stb. A cél, hogy ezt az egyszerű adatszerkezetet minél hatékonyabban tudjuk megvalósítani.



A **Halmaz**<*T*> egy *T* típus paraméterrel rendelkezik, amely a benne tárolható elemek típusát határozza meg. Később látni fogjuk, hogy a hatékony halmaz megvalósítások során gyakran azt is meg fogjuk követelni, hogy ez a típus összehasonlítható legyen.

Továbbá az alábbi műveleteket írja elő:

<sup>15</sup>empty

<sup>16</sup>enqueue

<sup>17</sup>dequeue

<sup>18</sup>first

<sup>19</sup>update

<sup>20</sup>set

- **BESZÚR**(érték)<sup>21</sup>: Berakja a paraméterként átadott *érték*-et a halmazba. Ha már volt ilyen érték a halmazban, akkor nem történik változás.
- **ELEME**(érték)<sup>22</sup>: Visszatérési értéke *igaz*, ha az átadott *érték* szerepel a halmazban és *hamis*, ha nem szerepel a halmazban.
- **TÖRÖL**(érték)<sup>23</sup>: Törli a paraméterként átadott *érték*-et a halmazból.
- **BEJÁR**(művelet): A halmaz minden elemére meghívja a paraméterként átadott *művelet*-et.

A halmaz azon változatával, amelyik megengedi, hogy ugyanaz az elem többször is szerepeljen (*multiset*) nem foglalkozunk.

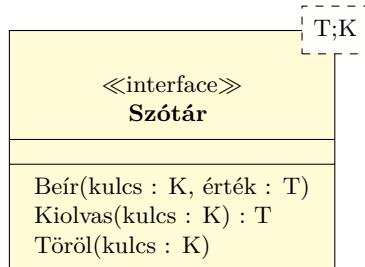
### 3.2.7. Szótár

A **Szótár**<sup>24</sup> specialitása, hogy ez kulcs-érték párokat tárol (másképpen fogalmazva, kulcsoknak és a hozzájuk tartozó értékeknek az egymáshoz rendelését). Tehát egy szótárba nem csak magát az eltárolandó értéket tároljuk el, hanem hozzárendelünk egy egyedi azonosításra alkalmas kulcsot is. A későbbiekben ezen a kulcson keresztül tudunk hozzáférni magához az eltárolt értékhez.

A **Szótár** ismerete nélkül valószínűleg mindenki egy **Lista** segítségével tárolná a hallgató objektumait, akik közül utána egy keresés ciklussal tudná kikeresni a megadott Neptun kóddal rendelkezőt. De látni fogjuk, hogy szótárak esetében van arra lehetőség, hogy (*neptun kód, hallgató*) párokat tároljunk el, és utána a Neptun kód megadása után igen hatékonyan vissza tudja adni a hozzá tartozó hallgatót.

Egy webes áruházban klasszikusan minden vásárlóhoz tartozik egy kosár (és senkinek se lehet egy-nél több kosara egy időben). Ilyen esetekben célszerű ezeket a kosarakat a vásárlóval összekapcsolva eltárolni egy szótárban. Tehát amikor bejelentkezik a felhasználó az áruházba, akkor létrehozunk egy kosarat, és eltároljuk a szótárban úgy, hogy kulcsként magát a vevőt adjuk meg. A későbbiekben, ha bármikor szükségünk van egy megadott vevőhöz tartozó kosárra, akkor ezt a szótárból egyszerűen a vevő megadásával kiolvashatjuk, nincs szükség keresésre. A példa azt is mutatja, hogy a kulcs lehet akár egy referencia is.

Itt is találhatunk számos technikai jellegű példát. A fordítóprogramokat például tipikusan úgy képzeljük el, hogy egy szótárban tárolják a változók neveinek és a hozzájuk tartozó memóriacímeknek az egymáshoz rendelését.



Mint látható, a **Szótár** $\langle K \rightarrow T \rangle$  két típus paraméterrel is rendelkezik, a **K** megadja a kulcs típusát, a **T** pedig az eltárolni kívánt érték típusát. A két típus egymástól teljesen független.

Az adatszerkezet az alábbi műveleteket írja elő:

- **BEÍR**(kulcs, érték)<sup>25</sup>: Berakja a paraméterként átadott *érték*-et a szótárba a *kulcs* kulccsal azonosítva. Ha már volt ilyen kulcsú elem, akkor azt felülírja.
- **KIOLVAS**(kulcs)<sup>26</sup> : Kiolvassa a *kulcs* által azonosított elemet a szótárból, és ez a függvény visszatérési értéke. Ha nincs ilyen kulcsú elem, akkor "Hibás kulcs" hibát jelez.

<sup>21</sup>add

<sup>22</sup>contains

<sup>23</sup>remove

<sup>24</sup>dictionary, map, lookup table, associative array

<sup>25</sup>add, put

<sup>26</sup>get

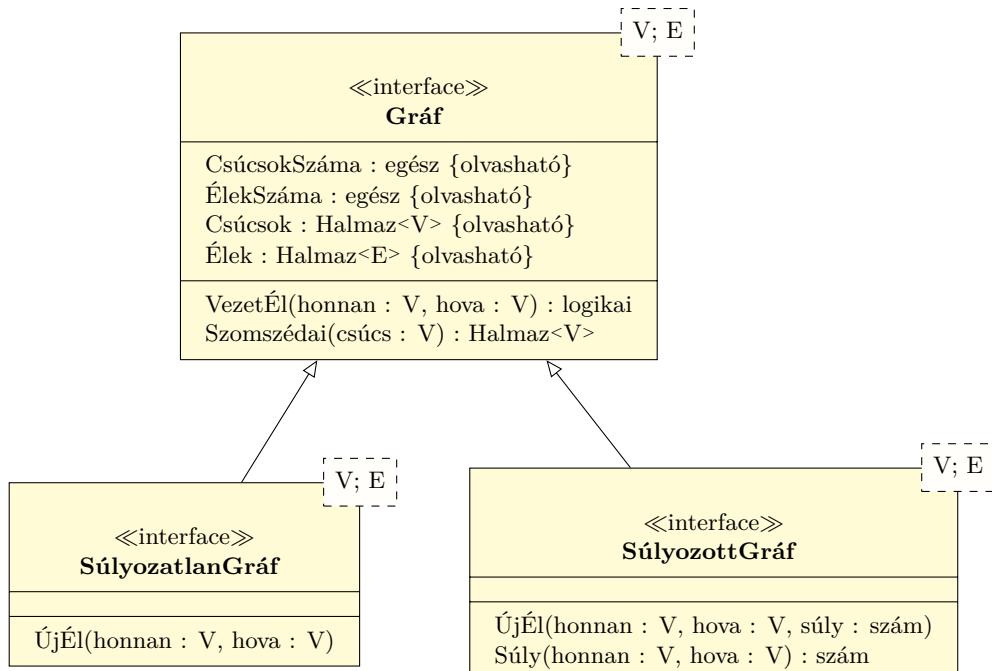
- **TÖRÖL(kulcs)<sup>27</sup>**: Törli a paraméterként átadott *kulcs* által azonosított elemet a szótárból.

### 3.2.8. Gráf

A **Gráf**<sup>28</sup> mint adatszerkezet, különböző csomópontokat, illetve azokat összekötő éleket tartalmaz. A gráfokon számos, egészen összetett műveletet tudunk meghatározni (pl. útkeresés, összefüggő komponensek meghatározása, stb.), de mint adatszerkezet, próbálunk a minimumra törekedni.

A gráfokat úgy képzeljük el, hogy azok alapvetően nem adatokat tárolnak, hanem az adatok közötti összefüggéseket. Ezt jelzi a lenti műveleteknél is, hogy nem vezettünk be külön csúcs felvétel/törlés műveleteket. A gráf létrehozásakor megadjuk, hogy hány csúcsot szeretnénk, azok kapnak valamilyen egyedi azonosítót, de további tartalmukkal a későbbiekben nem foglalkozunk, csak az őket összekötő élek az érdekesek. A műveletek így értelemszerűen nem a tartalom visszanyerésére szorítkoznak, hanem a gráf felépítésével kapcsolatos kérdésekre (persze ez senkit se akadályozzon meg abban, hogy például egy kiegészítő szótárban eltároljon az egyes csúcsokhoz egy hozzájuk tartozó tartalmat).

Egy **Gráf** tárolhatja például személyek és a köztük lévő ismerettségi kapcsolatok hálózatát. A gráf alapján le tudjuk kérdezni, hogy két személy ismeri-e egymást, illetve le tudjuk kérdezni egy megadott személy ismerőseit. Bár a lenti adatszerkezet erre nem ad még lehetőséget, de később ezt ki tudjuk egészíteni olyan műveletekkel, ami megadja, hogy két személy között van-e ismerősi lánc, és ha igen, akkor melyik a legrövidebb ilyen lánc.



A **Gráf** $<V, E>$  adatszerkezet is két típus paramétert tartalmaz: **V** képviseli majd a gráf csúcsait<sup>29</sup>, az **E** pedig az ezeket összekötő éleket<sup>30</sup>.

Az adatszerkezet az alábbi tulajdonságokat és műveleteket írja elő:

- **CsúcsokSzáma**: Visszaadja a gráf csúcsainak számát.
- **ÉlekSzáma**: Visszaadja a gráf éleinek számát.
- **Csúcsok**<sup>31</sup>: Egy halmazban visszaadja a gráf minden csúcsát.
- **Élek**<sup>32</sup>: Egy halmazban visszaadja a gráf minden élét.

<sup>27</sup>remove

<sup>28</sup>graph

<sup>29</sup>vertex

<sup>30</sup>edge

<sup>31</sup>allVertices

<sup>32</sup>allEdges

- **VEZETÉL**(honnán, hova)<sup>33</sup>: Visszatérési értéke *igaz*akkor, ha vezet él a *honnán* csúcsból a *hova* csúcsba. Ha nem, akkor a visszatérési értéke *hamis*.
- **SZOMSZÉDAI**(csúcs)<sup>34</sup>: Visszaadja a paraméterként átadott *csúcs* szomszédait, tehát azokat a csúcsokat, ahol vélez él a *csúcs*-ból. Mivel több ilyen is lehet, ezért a függvény visszatérési értéke csúcsok halmaza.

Mint látható, a fenti funkciók függetlenek attól, hogy maga a gráf súlyozott vagy sem. Ez a jellemző csak az élek felvételekor, illetve persze egy él súlyának lekérdezésekor lesz érdekes.

A *SúlyozatlanGráf* esetében az őst csak egy metódussal kell kiegészíteni:

- **ÚJÉL**(honnán, hova)<sup>35</sup>: Felvész egy új élet, ami a *honnán* csúcsból a *hova* csúcsba vezet.
- **A SúlyozottGráf** adatszerkezet esetében az egyes élekhez egy súly tulajdonságot is rendelünk. Az örökölt általános gráf műveleteket az alábbiakkal bővíti:
  - **ÚJÉL**(honnán, hova, súly)<sup>36</sup>: Felvész egy megadott *súly*-al rendelkező új élet, ami a *honnán* csúcsból a *hova* csúcsba vezet.
  - **SÚLY**(honnán, hova)<sup>37</sup>: Visszaadja a két paraméterként átadott csúcs között vezető él súlyát. Ha nincs ilyen él, akkor "Nincs él" hibát jelez.

### 3.2.9. Utólagos áttekintés

Megismerve az alapvető absztrakt adatszerkezeteket, talán érdemes őket újra áttekinteni és összehasonlítani őket.

A *Lista* alapvető jellegzetessége, hogy egy megadott sorrendben tárolja az elemeket. A sorrendet a beszúráskor lehet meghatározni, és ezen sorrend alapján lehet az adatokat elérni az indexelés segítségével.

A *Verem*, *Sor* és *PrioritásosSor* abban hasonlítanak egymásra, hogy itt szintén megjelenik egy sorrend, ez viszont adott, nem lehet a beszúráskor befolyásolni. Hozzáérni is minden csak a sorrendnek megfelelő következő elemhez lehet, ami a verem esetében az utoljára elhelyezett elem, sornál a legrégebben behelyezett elem, prioritásos sornál pedig a legnagyobb prioritással bíró elem.

A *Halmaz* ilyen szempontból a legegyszerűbb szerkezet, hiszen egyáltalán nem foglalkozik az elemek közötti kapcsolatokkal, sem sorrenddel. Csak annyit tud, hogy eltárol elemeket, és ezeket le lehet kérdezni, hogy benne vannak-e a halmazban, viszont ezt nagyon hatékony. Specialitása az, hogy egy elem csak egyszer szerepelhet benne.

A *Szótár* abban nyújt újdonságot, hogy az eltárolni kívánt tartalom mellett megjelenik egy kulcs érték is. Az adatszerkezet pedig ezen kulcs-tartalom párokat tárolja el, általában nagyon hatékony műveleteket biztosítva.

A *Gráf* esetében pedig nem a tartalmi adatok eltárolása a lényeges, hanem kimondottan az elemek közötti kapcsolatok kezelésére lett kifejlesztve. Működése is más, itt nem a benne tárolt adatokhoz akarunk hozzáérni vagy azokat karbantartani, hanem a felépített hálózat jellemzőit tudjuk kinyerni.

---

<sup>33</sup>adjacent

<sup>34</sup>neighbors

<sup>35</sup>addEdge

<sup>36</sup>addEdge

<sup>37</sup>weight

### 3.3. Leképezés fizikai adatszerkezetekre

#### 3.3.1. Implementációs rugalmasság

Az absztrakt adatszerkezetek esetében szándékossan nem foglalkoztunk a tényleges implementáció lehetőségeivel. Látnunk kell ugyanis, hogy ez sokféleképpen megvalósítható.

Példaként gondoljuk át, hogy miként lenne célszerű megvalósítani egy *Halmaz* adatszerkezetet a valóságban?

- Amíg az ember csak a tömböket ismeri, addig minden tömbökkel szeretne megvalósítani (4. fejezet). És tény, hogy erre van is lehetőség, a halmazunk úgy is működhet, hogy minden felvett elemet felvesz egy tömbbe, és az elemvizsgálatkor egy kereséssel átnézzük ezt a tömböt, hogy benne van-e a keresett elem.
- A fenti után azonnal felmerül a lehetőség, hogy a tömb legyen rendezett. Ezzel ugyanis jóval gyorsabb keresést tudunk megvalósítani (feltéve persze, hogy a tartalom rendezhető).
- Még a tömböknél maradva, az is egy megoldás, hogy létrehozunk egy akkora logikai elemeket tartalmazó tömböt, amennyi a **T** típus teljes értékkészletének mérete. Például ha hónapokat akarunk tárolni a halmazban, akkor elég egy 12 elemű logikai tömb, ahol a 3. elem értéke azt mutatja, hogy a „március” hónap benne van-e a halmazban.
- Az elemeket eltárolhatjuk egy láncolt adatszerkezetben (5. fejezet), vagy akár annak rendezett formájában.
- Később látni fogjuk, hogy egy bináris keresőfa (6. fejezet) is nagyszerű megoldás lehet a probléma megoldására.

A legérdekesebb az egészben, hogy a fentiek közül talán mindenek van létjogosultsága. A logikai tömbös megoldás futásideje minden esetben egyetlen lépés, de csak akkor valósítható meg, ha nem túl nagy a kulcs halmaz mérete. A tömbös megoldás tárigénye is optimális tud lenni, viszont számolni kell a tömb fix méretéből adódó gondokkal. Ezt a problémát megoldják a láncolt adatszerkezetek, amelyek már dinamikusak, azonban többlet mutatókat kell bennük használnunk, és a bináris keresést nem tudjuk használni a kiolvasásnál. A fák a fentiekhez képest egy egészen jó általános megoldást nyújtanak, de csak akkor, ha a kulcsok összehasonlíthatók.

A fentiek miatt nem szeretnénk egy-egy kapcsolatot létesíteni az absztrakt adatszerkezetek és az implementációs lehetőségek között. Ezzel épp ellentétesen, szeretnénk bemutatni, hogy megadott funkciójú adatszerkezet többféleképpen is implementálható, az ezek közüli választás mindenkor az adott feladattól függ.

#### Megjegyzés

Viszont ezt a döntést csak az tudja megbízhatóan meghozni, aki ismeri a különféle fizikai megvalósítások jellemzőit, előnyeit és hátrányait. Ezért is tartjuk fontosnak, hogy ezek a téma körök megjelenjenek az informatikus oktatásban. Még akkor is, ha pontosan tudjuk, hogy a tanult implementációk általában már készen rendelkezésre állnak a különböző programozási nyelvek osztálykönyvtáraiban, tehát ezek újbóli implementálása szükségtelen.

Már volt róla szó, hogy objektum-orientáltan szeretnénk elvégezni az adatszerkezetek megvalósítását. Ez ugyanis nagyon jól illeszkedik a megismert kettősséghöz:

- Interfészok segítségével írjuk le az absztrakt adatszerkezeteket, hiszen itt csak a műveleteket definiáljuk, nem foglalkozunk az implementációval.
- Osztályok segítségével készítjük el az egyes konkrét implementációkat, amelyek majd megvalósítják az interfészeken előírt műveleteket.

### **3.3.2. Szekvenciális megvalósítás**

Ez alatt tulajdonképpen a tömbökkel való megvalósítást értjük. A név onnan ered, hogy itt az adatokat közvetlenül egymást követően, szekvenciális formában tároljuk. minden adatszerkezet megvalósítható tömbök segítségével, miként azt láttuk a halmazoknál.

A módszer előnye általában egyszerűsége, illetve az, hogy nagyon hatékony tárhelykezelést tud biztosítani. Ezekben az esetekben ugyanis általában nincs semmilyen kiegészítő adat, az egyes adatszerkezetek adatait nyersen el tudjuk tárolni a memóriában, az már a műveletek feladata, hogy ebből az adatból hogyan tudja előállítani a szükséges működést.

Érdemes megjegyezni, hogy maga a tömb is felfogható egyfajta adatszerkezetnek. Sok limitációval rendelkezik, de ha ezek nem okoznak problémát, akkor egy kimondottan hatékony, gyors adatszerkezethez juthatunk vele.

### **3.3.3. Láncolt megvalósítások**

A láncolt megvalósítások tipikusan azon alapulnak, hogy a memóriában szétszórva helyezkednek el az adatszerkezet különböző darabjai, és az azok közötti kapcsolatokat mutatók/referenciák segítségével tároljuk el.

Hagyományos nyelvekben ezt még közvetlen dinamikus memóriakezeléssel oldottuk meg, ahol az adatszerkezet elemeihez memóriát kellett lefoglalni, amihez aztán mutatókon keresztül fértünk hozzá.

Ehelyett mi már egy OOP elvű megközelítést szeretnénk megvalósítani, ahol maga az adatszerkezet is egy objektum, de annak részei (pl. a lista egyes elemei) szintén külön-külön objektumokként jelennek meg, amelyek között referenciák mutatják a kapcsolatokat.

A módszer legnagyobb előnye annak dinamikus volta, hiszen objektumokat futásidőben szabadon létrehozhatunk és felszabadíthatunk, így azokból mindig pontosan annyi fogja foglalni a memóriát, amennyire éppen szükség van.

Mindez hátrányokkal is jár, az így kapott adatszerkezetek helyfoglalása általában meghaladja a tömbökét. Az egyes elemeket összekapcsoló mutatók ugyanis többlet memóriát igényelnek, ami nagy mennyiségi adatnál már látványos különbséget okoz.

## 4. fejezet

# Tömb alapú implementációk

### 4.1. Adatok reprezentációja

#### 4.1.1. Alapvető műveletek

Az egyszerű típusok mellett a tömb típus lesz az, amit eleve a programozási nyelv által adottnak tekintünk a későbbiekben. Ez a legegyszerűbb és legintuitívabb adatszerkezet, amely azon alapul, hogy több azonos típusú elemet úgy tárolunk el, hogy egyszerűen egymás után helyezzük őket a memóriában, majd pedig egy egész szám azonosító segítségével férünk hozzá az egyes elemekhez.

Az egyértelműség kedvéért a tömb változókat mindenkor nagybetűvel jelöljük ( $A$ ), a tömb egyes elemeihez való hozzáférést pedig a programozási nyelvekben már megsokott indexeléssel érjük el ( $A[i]$  adjá vissza a tömb  $i$ . elemét). Bár technikai értelemben a 0-tól való indexelés egyszerűbbnek tűnik, a későbbiekben mi a tömb elemeit 1-től indexeljük, mivel ez jóval közelebb áll az emberi gondolkodáshoz.

Egy tömbnek bármikor le tudjuk kérdezni a méretét az alábbi operátorral:  $A.\text{méret}$ , ami visszaad egy egész számot, ami az  $A$  tömbben eltárolható elemek számát.

#### 4.1.2. Statikus megvalósítás

Tömbök esetében beszélhetünk statikus és dinamikus megvalósításról. Statikus esetben már a fordítás idejében meg kell mondanunk a tömb méretét, a fordító pedig automatikusan létrehozza az ennek megfelelő memóriaterületet. Ezt az állapotot mutatja a 4.1. ábra. Jól látható, hogy az egyes elemek közvetlenül egymás mögött helyezkednek el a memóriában, az  $A$  változót pedig úgy tekintjük, hogy az egész szerkezetre vonatkozik. Tehát pl. ha paraméterként átadjuk a tömböt, akkor annak egy teljes másolata adódik át.

A tömböket is generikus típusként fogjuk kezelni, tehát egy  $\mathbf{T}$  típusú elemeket tartalmazó tömb jelölése:  $\mathbf{tömb} < \mathbf{T} >$ .

---

**4.1. Algoritmus** Műveletek statikus tömbbel.

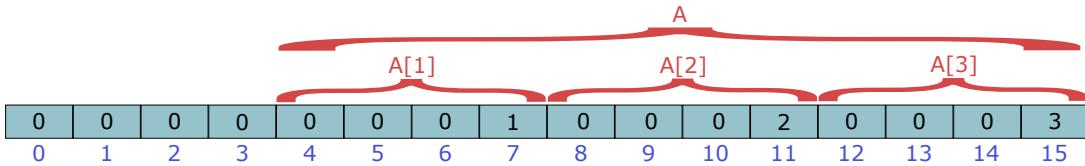
---

```
1: $A[1] \leftarrow 1$
2: $A[2] \leftarrow 2$
3: $A[3] \leftarrow 3$
```

---

**Felhasznált változók és függvények**

- $A$  : Egy statikus  $\mathbf{tömb} < \mathbf{egész} > [3]$  típusú változó, ahol  $A.\text{méret}=3$ .
-



4.1. ábra. A 4.1. algoritmus eredménye a memóriában.

Egy tömb tartalmazhat egyszerű típusokat, de akár összetett típusokat is. Ez utóbbi esetben felmerül a kérdés, hogy magában a tömbben már direkt módon az adatokat tároljuk, vagy ott csak egy hivatkozást a tényleges adatokra. Az előző megegyezésünk alapján (összetett típusokat minden dinamikus módon kezeljük) ezt minden úgy tekintjük, hogy az összetett típusú tömbök valójában csak mutatókat tartalmaznak, amelyek az egyes összetett típusú objektumokra hivatkoznak.

#### 4.1.3. Dinamikus megvalósítás

Modern programozási nyelvek esetében a tömbök már alapvetően dinamikus típusként jelennek meg. Ennek megfelelően futásidőben kell őket létrehozni (itt kell megadni a méretet is), és a létrehozáskor csak egy referenciát kapunk a tömb első elemére. Az elemekhez való hozzáférés azonos módon történik az indexelés segítségével. Az indexelés művelet során értelemszerűen nem a mutatót próbáljuk meg indexelni, hanem az általa hivatkozott tömb elemeit érjük így el.

Azonban a teljes tömbbel való műveletek egészen másképp működnek, mint a statikus megvalósításnál. Egy értékadás során itt már nem a teljes tömbről jön létre másolat, csak a referenciáról. Ezt mutatja be a 4.2. algoritmus, aminek a kimenetele látható a 4.2. ábrán. Ugyanígy, egy paraméteradásnál is csak a referenciát adjuk át paraméterként, nem a tömb egy másolatát.

---

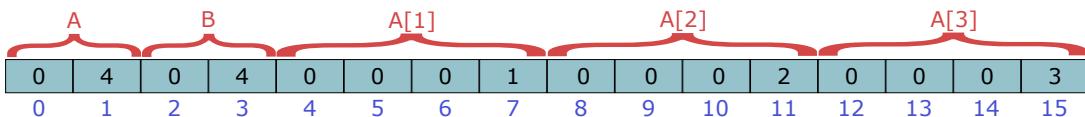
#### 4.2. Algoritmus Műveletek dinamikus tömbbel

---

- 1:  $A \leftarrow \text{LÉTREHOZ}(\text{tömb}<\text{egész}>/3)$
  - 2:  $A[1] \leftarrow 1$
  - 3:  $A[2] \leftarrow 2$
  - 4:  $A[3] \leftarrow 3$
  - 5:  $B \leftarrow A$
  - 6:  $\text{FELSZABADÍT}(A)$
- 

#### Felhasznált változók és függvények

- $A$  : Egy dinamikus  $\text{tömb}<\text{egész}>$  típusú változó.
  - $B$  : Egy dinamikus  $\text{tömb}<\text{egész}>$  típusú változó.
  - $\text{LÉTREHOZ}(\text{tömb}< T >[\text{méret}])$  : Létrehoz egy  $T$  típusú,  $\text{méret}$  méretű tömböt.
  - $\text{FELSZABADÍT}(A)$  : Felszabadítja az  $A$  által hivatkozott tömböt.
- 



4.2. ábra. A 4.2. algoritmus állapota a felszabadítás előtti állapotban.

Mint látható, a használat szempontjából (elemek tárolása és kiolvasása) nincs különbség a két módszer között, ezért pszeudokód szinten alapvetően lényegtelen, hogy melyik megvalósítást használjuk. Néha azonban fontos ezt tudunk, hiszen pl. tömbök közötti értékadásnál, vagy tömbök paraméterként átadásánál másképp viselkedik a két implementáció. A minél könnyebben olvasható pszeudokódok érdekében:

- ha a pszeudokódban megjelenik a direkt létrehozás/felszabadítás művelet, akkor dinamikus,
- egyébként pedig statikus megvalósításra gondolunk.

Előbbi esetben a tömb mérete a pszeudokódból látható, mivel az futásidőben jön létre. Statikus esetben pedig minden változó leírása fogja ezt tartalmazni.

#### Implementáció

A tömbök megvalósítási módja szintén programozási nyelvektől függ. Java és C# esetében csak a dinamikus megvalósítással találkozhatunk, hiszen itt a tömbök objektumokhoz hasonló módon működnek. C++ esetében minden két fajta megvalósítás használható, hasznos nyelveknél (pl. Pascal) pedig a tömb létrehozása alapvetően statikus, bár a dinamikushoz hasonló formát is el lehet érni direkt memória foglalással és mutatókkal.

#### Megjegyzés

A 4.2. ábrán az is jól látható, hogy miért alakult ki a 0-val való indexelés a gyakorlatban. Ha az  $A$ -t egy mutatónak tekintjük, ami egy memóriacímet tárol (jelen esetben 4), akkor a tömb  $i$ . elemének memóriabeli helye egyszerűen számolható az alábbi képlettel:  

$$A + (i \cdot \text{ElemMéret})$$

#### 4.1.4. Többdimenziós tömbök

A gyakorlatban szükség lehet kettő, vagy akár többdimenziós tömbökre is. Egy kétdimenziós tömböt mutat a 4.3. ábra. A tömb egyes elemeihez az  $A[i,j]$  formában férünk hozzá, ahol  $i$  a sort határozza meg, a  $j$  pedig az oszlopot (mindkét érték 1-től indexelt).

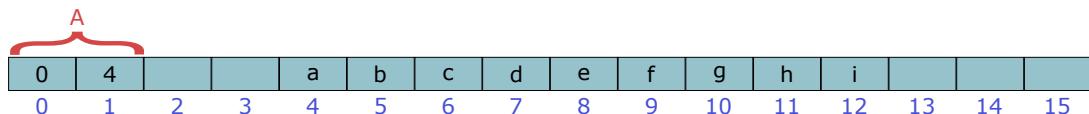
|   |   |   |
|---|---|---|
| a | b | c |
| d | e | f |
| g | h | i |

4.3. ábra. Egy minta kétdimenziós tömb.

Felmerülhet a kérdés, hogy ezek milyen formában tárolódnak a memóriában. Az nyilvánvaló, hogy a kétdimenziós elemeket valahogyan egy szekvenciális sorozattal kell leírni, ami már egyszerűen tárolható, a kérdés csak ennek a sorozatnak a felépítési módja.

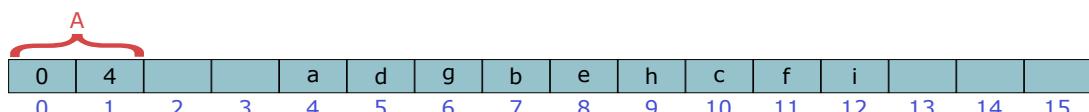
Ennek két fő megvalósítása ismert:

- Soronkénti (row-major) tárolás: ebben az esetben elsőként a mátrix első sora kerül eltárolásra, majd pedig az azt követő sor elemei, és így tovább az utolsó sorig. Ezt mutatja a 4.4. ábra.



4.4. ábra. A 4.3. ábrán látható tömb tárolása row-major formában.

- Oszloponkénti (column-major) tárolás: ebben az esetben elsőként a mátrix első oszlop kerül eltárolásra, majd pedig az azt követő oszlopainak elemei, és így tovább az utolsó oszlopig. Ezt mutatja a 4.5. ábra.



4.5. ábra. A 4.3. ábrán látható tömb tárolása column-major formában.

Ameddig nem szeretnénk alacsonyabb szintű programokat írni, ami közvetlenül fér hozzá a tömb elemeihez, addig ez a különbség nem okoz nekünk problémát, így a pszeudokódjainkban se kell definiálni a pontos tárolási módot.

#### Implementáció

A gyakorlatban ettől függetlenül érdemes tisztában lennünk ezekkel a különbségekkel, ugyanis gyakori, hogy egy nagyobb rendszer különböző moduljait más-más programozási nyelven írják, és esetleg ezen nyelvekben különbözik a mátrixelemek tárolási sorrendje. Ez egy könnyen korrigálható probléma, ha tudjuk, hogy mi okozza.

#### Implementáció

Érdemes megjegyezni, hogy bizonyos nyelvek egyáltalán nem támogatják a többdimenziós tömböket, ilyen pl. a Java is. A gyakorlatban ez nem okoz problémát, ugyanis hasonló hatást érhetünk el tömbök tömbjével.

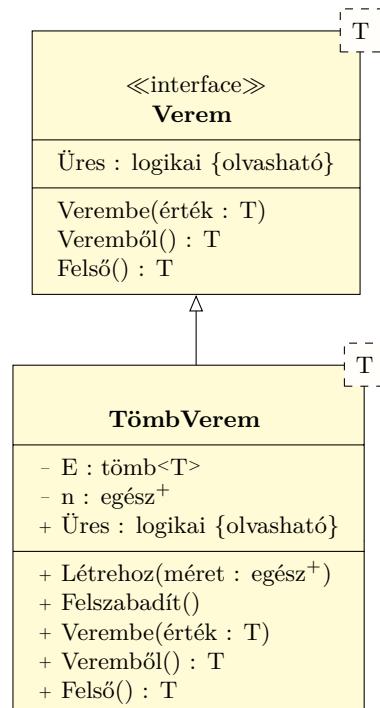
## 4.2. Verem megvalósítása tömbbel

### 4.2.1. Alapvető felépítés

**Verem** adatszerkezet megvalósításra klasszikus példa szokott lenni a tömb használata. Az adatszerkezet nem különösebben bonyolult, ezért csak két változóra van szükség: egy *E* tömb fogja tárolni a veremben lévő értékeket, illetve egy *n* mező tárolja a veremben aktuálisan lévő értékek számát.

Ez utóbbi változó értékét minden behelyezéskor növeljük, kivételkor pedig csökkentjük. A verembe egymást követően elhelyezett elemeket az *E* tömb 1, 2, ... indexű helyein fogjuk elhelyezni. Vegyük észre, hogy mind behelyezéskor, mind pedig kivételkor az *n* éppen az utolsóként elhelyezett elem indexét tartalmazza, ennek megfelelően a darabszám tárolásán túl ezt a változót fogjuk használni a tömb indexelésére is.

A műveletek megfelelnek a **Verem** interfészénél már megismertekkel, kiegészítve egy létrehozás és felszabadítás művelettel. A létrehozásnál egy paramétert várunk, ez az értékeket tároló belső tömb méretét határozza meg.



### 4.2.2. Létrehozás és megszüntetés

A **LÉTREHOZ** művelet során létrehozzuk az *E* tömböt, illetve beállítjuk az *n* értékét nulla-re, ezzel is jelezve, hogy még nincsen a veremben egy elem sem. A 4.3. algoritmus azt mutatja be, hogy az inicializáláskor megadhatunk egy maximális méretet, és ennek megfelelő méretű dinamikus tömböt hozunk létre. OOP megvalósítás esetében az inicializálás tipikusan a konstruktur híváskor történhet meg.

A létrehozás utáni állapotot a 4.6. ábrán látható módon ábrázolhatjuk. A példában egy 5 maximális elemszámmal bíró vermet hoztunk létre, ennek megfelelően az *E* egy 5 elemű tömb. Az *n* értéke kezdetben 0, tehát még nem hivatkozik a tömb egyik elemére sem.

A verem objektum felszabadítása során nincs különösebb teendőnk, csak a létrehozott tömböt megszüntetni. Mivel jelen esetben dinamikus memóriafoglalást használunk, ezért a létrehozott tömböt fel is kell szabadítani, hogy ne foglalja többé a memóriát feleslegesen (4.4. algoritmus).

---

### 4.3. Algoritmus Tömb alapú verem létrehozása

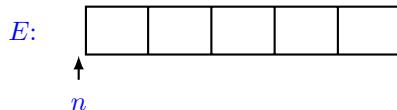
---

Bemenet: méret - egész<sup>+</sup> (a tömb mérete, ami tárolni fogja az értékeket)

- 1: eljárás TÖMBVEREM.LÉTREHOZ(méret)
  - 2:       $E \leftarrow \text{LÉTREHOZ(tömb} < T > / \text{méret})$
  - 3:       $n \leftarrow 0$
  - 4: eljárás vége
- 

#### Felhasznált változók és függvények

- $E$  : A tömb, ami tárolni fogja az értékeket.
  - $n$  : A veremben lévő elemek aktuális száma.
  - LÉTREHOZ( $tömb < T > / \text{méret}$ ) : Létrehoz egy  $T$  típusú,  $\text{méret}$  méretű tömböt.
- 



4.6. ábra. Inicializálás utáni állapot a tömb alapú verem megvalósításkor. Az  $n$  értéke 0, ennek megfelelően itt még nem címzi a tömb egyik elemét se.

### 4.2.3. Műveletek megvalósítása

#### Új elem beszúrása a verembe

A VEREMBE művelet egy új értéket vesz fel a verembe, ezt mutatja a 4.5. algoritmus. Első lépésként ellenőrizzük, hogy belefér-e még az új elem a verembe. Amennyiben nem, akkor hibaüzenetet küldünk (tipikusan egy kivétel eldobásával). Amennyiben mégis, akkor növeljük az  $n$  értékét. Vegyük észre, hogy a megnövelt  $n$  változó az  $E$  következő üres helyére mutat, tehát ebbe a pozícióba el is helyezhetjük a paraméterként kapott értéket.

#### Elem kivétele a veremből

A VEREMBŐL művelet az előző fordítottja (4.6. algoritmus). Célja, hogy kivegye a verem legutoljára berakott elemét. Ehhez elsőként ellenőrzi, hogy van-e egyáltalán elem a veremben ( $n > 0$ ), és hibát jelez, ha nincs. Ha igen, akkor az érték segédváltozóban eltárolja az  $E[n]$ , tehát az utoljára elhelyezett elem értékét. Csökkenti az  $n$  értékét, mivel innentől úgy tekintjük, hogy már egyel kevesebb elem van a veremben, végül visszaadjuk az előbb elmentett érték változó tartalmát. Érdemes megfigyelni, hogy magát az értéket valójában nem töröljük ki tömbből, csak az  $n$  csökkentésével jelezzük, hogy az utolsó tömbben lévő elemet már nem tekintjük a verem részének.

#### A verem felső elemének lekérdezése

A FELSŐ művelet megvalósítása alapvetően azonos a veremből való kivétellel, csak annyi a különbség, hogy ez nem változtatja az  $n$  értékét, ezért nem veszi ki az utolsó elemet a veremből (4.7. algoritmus).

A fenti műveletekre mutat példákat a 4.7. ábra. Az adatszerkezet egyszerűsége miatt ezek az állapotok magukért beszélnek, további magyarázatot nem igényelnek.

---

### 4.4. Algoritmus Tömb alapú verem felszabadítása

---

- 1: eljárás TÖMBVEREM.FELSZABADÍT( )
  - 2:      FELSZABADÍT( $E$ )
  - 3: eljárás vége
- 

#### Felhasznált változók és függvények

- FELSZABADÍT( $E$ ) : Felszabadítja az  $E$  tömböt.
-

---

#### 4.5. Algoritmus Tömb alapú verembe új elem felvétele

---

**Bemenet:** érték - T (a beszúrandó új érték)

```
1: eljárás TÖMBVEREM.VEREMBE(érték)
2: ha $n < E.méret$ akkor
3: $n \leftarrow n + 1$
4: $E[n] \leftarrow$ érték
5: különben
6: hiba "Nincs hely"
7: elágazás vége
8: eljárás vége
```

---

**Felhasznált változók és függvények**

- **E** : A tömb, ami tárolja az értékeket.
  - **n** : A veremben lévő elemek aktuális száma.
- 

---

#### 4.6. Algoritmus Tömb alapú veremből elem kivétele

---

**Kimenet:** . - T (a verembe utoljára felvett érték)

```
1: függvény TÖMBVEREM.VEREMBŐL()
2: ha $n > 0$ akkor
3: érték $\leftarrow E[n]$
4: $n \leftarrow n - 1$
5: vissza érték
6: különben
7: hiba "Nincs elem"
8: elágazás vége
9: függvény vége
```

---

**Felhasznált változók és függvények**

- **E** : A tömb, ami tárolja az értékeket.
  - **n** : A veremben lévő elemek aktuális száma.
- 

---

#### 4.7. Algoritmus Tömb alapú veremben a felső elem lekérdezése

---

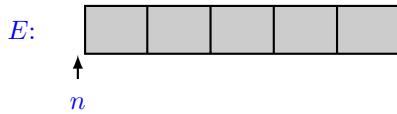
**Kimenet:** . - T (a verembe utoljára felvett érték)

```
1: függvény TÖMBVEREM.FELSŐ()
2: ha $n > 0$ akkor
3: vissza E[n]
4: különben
5: hiba "Nincs elem"
6: elágazás vége
7: függvény vége
```

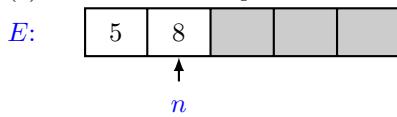
---

**Felhasznált változók és függvények**

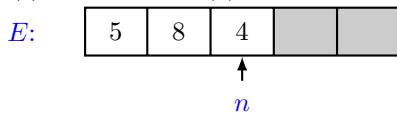
- **E** : A tömb, ami tárolja az értékeket.
  - **n** : A veremben lévő elemek aktuális száma.
-



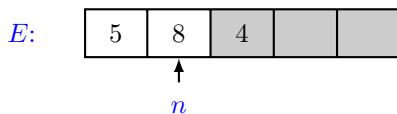
(a) Az üres verem alapértelmezett állapota.



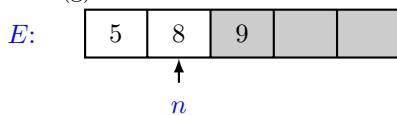
(c) Második elem (8) beszúrása a verembe.



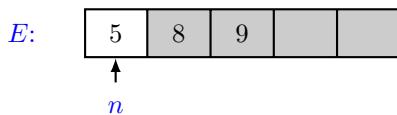
(e) A **FELSŐ** művelet ilyen esetben a 4-et adja vissza.



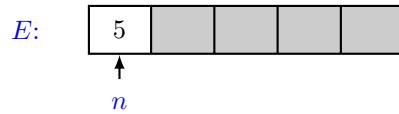
(g) A verem felső eleme most a 8.



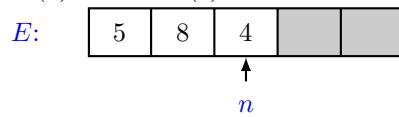
(i) **VEREMBŐL** művelet meghívása. Visszatérési értéke: 9.



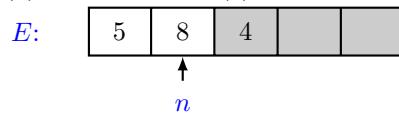
(k) A **FELSŐ** művelet ilyen esetben az 5-öt adja vissza.



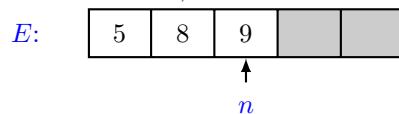
(b) Első elem (5) beszúrása a verembe.



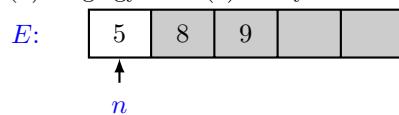
(d) Következő elem (4) beszúrása a verembe.



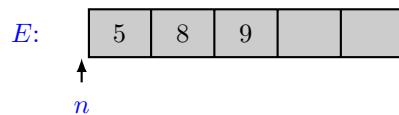
(f) Elem kivétele (visszatérési értéke: 4). A 4 a tömbben marad, de már nem a verem része.



(h) Még egy elem (9) elhelyezése a verembe.



(j) **VEREMBŐL** művelet meghívása. Visszatérési értéke: 8.



(l) **VEREMBŐL** művelet meghívása. Visszatérési értéke: 5. A verem visszaáll az alapértelmezett állapotába.)

4.7. ábra. A verem alapvető műveleteinek bemutatása.

## A verem ürességének ellenőrzése

A *Üres* tulajdonság kiolvasó művelete nagyon egyszerű, hiszen az *n* minden a veremben lévő elemek számát mutatja. Amennyiben ennek értéke 0, akkor a verem üres, egyébként pedig nem (4.8. algoritmus).

---

### 4.8. Algoritmus Tömb alapú verem ürességének lekérdezése

---

**Kimenet:** . - logikai (*üres a verem?*)

- 1: **tulajdonság** TÖMBVEREM.ÜRES
  - 2:     **vissza** *n* = 0
  - 3: **tulajdonság vége**
- 

#### Felhasznált változók és függvények

- *n* : A veremben lévő elemek aktuális száma.
-

## 4.3. Sor megvalósítása tömbbel

### 4.3.1. Alapvető felépítés

A **Sor** a veremhez hasonlóan egyszerűen megvalósítható egy tömb segítségével. Ez persze itt is ugyanazzal a hátránnal jár, hogy előre meg kell adni az adatszerkezet méretét, és ez a későbbiekben nem változtatható. Az eltárolt értékeket itt is egy **E** tömbbe fogjuk elhelyezni, az indexek kezelése viszont már kicsit összetettebb, ugyanis az utoljára beszúrt elem itt már nem biztos, hogy azonos a legkorábban beszúrt értékkel, nekünk viszont mindenki pozíciót el kell tárolnunk. Ezért két változót vezetünk be:

- **u** változó fogja tárolni az utoljára elhelyezett érték pozíóját. Ennek az a szerepe, hogy innen fogjuk tudni, hogy hova kell majd a tömbbe helyezni a következő elemet.
- **e** változó fogja tárolni a legkorábban elhelyezett, még sorban lévő érték indexét. Ennek az a szerepe, hogy innen fogjuk tudni, hogy honnan kell a tömbből kivenni a következő elemet.

A sor tömb alapú megvalósításának specialitása, hogy ezen változók értéke minden csakis növekedni fog. Amikor pedig már túlcímezünk a tömböt, akkor visszaléptetjük őket az 1-es indexű helyre. Amennyiben a tömb mérete **E.méret**, akkor az alábbi maradékos osztással nagyon egyszerűen végre lehet ezt hajtani az **e** változó esetében:

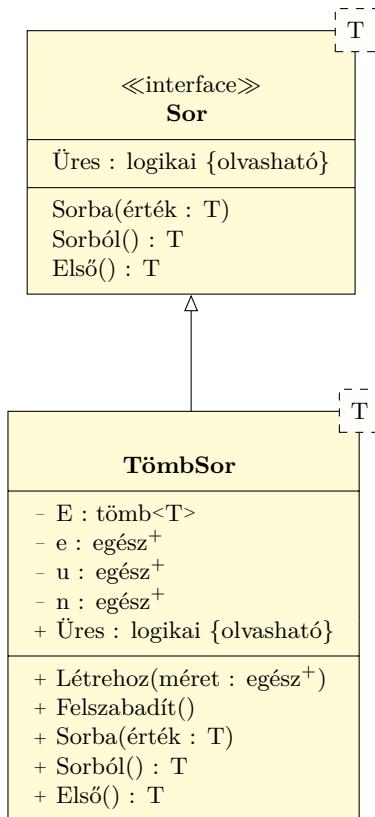
$$e \leftarrow (e \bmod E.\text{méret}) + 1 \quad (4.1)$$

Érdemes egy példán végignézni az **e** változó értékének változását egy 5 elemű **E** tömb esetén. A 0-s kezdőértéket követően az egyes növelések során a változó az alábbi értékeket veszi fel: 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, ...

Az **u** változót hasonlóan kezelve elérhetjük, hogy minden változó körbe fussen a tömb elemein, így egy tömbbe annak méreténél többször is felvehetünk és kivehetünk elemeket, csak arra kell ügyelni, hogy egy időben nem lehet benne a méretnél több érték.

Ez a jellemző viszont felvet egy érdekes kérdést, hogy miként tudjuk meghatározni, hogy van-e elem az adatszerkezetben, vagy nincs. Önmagában az, hogy az **e** vagy az **u** változó értéke a nagyobb, nem ad erről információt, hiszen minden változó előfordulhat. A legelső elemek behelyezésénél például az **u** lesz nagyobb, de amikor az „körbeér”, akkor értéke már kisebb lehet, mint az **e** változó értéke, pedig van még elem a tömbben. Az **e = u** állapot is két esetben állhat elő: vagy üres a sor, vagy éppen teljesen tele van. Ezen problémák egyszerű megoldását adja, ha bevezetünk egy **n** változót, amely minden sorban lévő elemek számát jelzi. Ha ez 0, akkor üres a sor, ha pedig értéke egyenlő a tömb méretével, akkor az minden tele van.

A műveletek megfelelnek a **Sor** interféznél már megismertekkel, kiegészítve egy **LÉTREHOZ** és **FELSZABADÍT** művelettel. A létrehozásnál egy paramétert várunk, ez az értékeket tároló belső tömb méretét határozza meg.



### 4.3.2. Létrehozás és megszüntetés

A **LÉTREHOZ** művelet során létrehozzuk az **E** tömböt, illetve beállítjuk az **n** értékét nullára, ezzel is jelezve, hogy még nincsen a sorban egy elem sem. A tömb méretét többféleképpen megadhatjuk, a 4.3. algoritmus azt mutatja be, hogy az inicializáláskor megadhatunk egy maximális méretet, és ennek megfelelő méretű dinamikus tömböt hozunk létre. OOP megvalósítás esetében az inicializálás tipikusan a konstruktor híváskor történhet meg.

---

#### 4.9. Algoritmus Tömb alapú sor létrehozása

---

**Bemenet:** *méret* - **egész<sup>+</sup>** (a tömb mérete, ami tárolni fogja az értékeket)

- 1: **eljárás** TÖMBSOR.LÉTREHOZ(*méret*)
  - 2:    *E*  $\leftarrow$  LÉTREHOZ(*tömb<T>/méret*)
  - 3:    *e*  $\leftarrow$  0
  - 4:    *u*  $\leftarrow$  0
  - 5:    *n*  $\leftarrow$  0
  - 6: **eljárás vége**
- 

#### Felhasznált változók és függvények

- **E** : A tömb, ami tárolni fogja az értékeket.
  - **e** : A legutóbb kiolvasott elem indexe (0, mivel még nem volt ilyen).
  - **u** : Az utolsó beszúrt elem indexe (0, mivel még nem volt ilyen).
  - **n** : A sorban lévő elemek aktuális száma.
  - **LÉTREHOZ(*tömb<T>/méret*)** : Létrehoz egy **T** típusú, *méret* méretű tömböt.
- 

A sor objektum felszabadítása során nincs különösebb teendőnk, csak a létrehozott tömböt megszüntetni, hogy ne foglalja többé a memóriát feleslegesen (4.10. algoritmus).

---

#### 4.10. Algoritmus Tömb alapú sor felszabadítása

---

- 1: eljárás TÖMBSOR.FELSZABADÍT( )
- 2:     FELSZABADÍT( $E$ )
- 3: eljárás vége

---

##### Felhasznált változók és függvények

- FELSZABADÍT( $E$ ) : Felszabadítja az  $E$  tömböt.
- 

### 4.3.3. Műveletek megvalósítása

#### Új elem felvétele a sorba

A sorba egy új érték felvételét mutatja a 4.11. algoritmus. Első lépésként ellenőrizzük, hogy belefér-e még az új elem a sorba. Amennyiben nem, akkor valamilyen formában hibaüzenetet küldünk (tipikusan egy kivétel eldobásával). Amennyiben mégis, akkor növeljük az  $n$  értékét. Ezt követően növeljük az  $u$  változó értékét is, amelyik az eltárolandó elem helyét fogja mutatni a növelés után. A növeléskor ellenőrizzük, hogy ha az  $u$  értéke meghaladná a tömb méretét, akkor azt újra 1-re állítjuk.

---

#### 4.11. Algoritmus Tömb alapú sorba elem felvétele

---

**Bemenet:** érték -  $T$  (a sorba beszúrandó új érték)

- 1: eljárás TÖMBSOR.SORBA(érték)
- 2:     ha  $n < E.\text{méret}$  akkor
- 3:          $n \leftarrow n + 1$
- 4:          $u \leftarrow (u \bmod E.\text{méret}) + 1$
- 5:          $E[u] \leftarrow \text{érték}$
- 6:     különben
- 7:         hiba "Nincs hely"
- 8:     elágazás vége
- 9: eljárás vége

---

##### Felhasznált változók és függvények

- $E$  : A tömb, ami tárolja az értékeket.
  - $u$  : Az utolsó beszúrt elem indexe.
  - $n$  : A sorban lévő elemek aktuális száma.
- 

#### Elem kivétele a sorból

A sorból való kivétel az előző fordítottja (4.12. algoritmus). Célja, hogy kivegye a sor legrégebben berakott elemét. Ehhez elsőként ellenőrzi, hogy van-e egyáltalán elem a veremben ( $n > 0$ ), és hibát jelez, ha nincs. Ha igen, akkor csökkentjük az  $n$  értékét, illetve az  $u$ -hoz hasonlóan növeljük az  $e$  értékét is. A növelés után ennek értéke a visszaadandó elem indexére hivatkozik, aminek értékét vissza is adjuk.

#### Sor első elemének lekérdezése

A **Első** művelet megvalósítása alapvetően azonos a sorból való kivétellel, csak annyi a különbség, hogy ez nem változtatja a  $e$  értékét, így nem veszi ki az elemet a sorból.

A fenti műveletekre mutat példákat a 4.8. ábra. Az adatszerkezet egyszerűsége miatt ezek az állapotok magukért beszélnek, további magyarázatot nem igényelnek.

#### A sor ürességének ellenőrzése

A **Üres** tulajdonság kiolvasó művelete teljesen megegyezik a veremnél látottal, hiszen az  $n$  itt is minden a sorban lévő elemek számát mutatja. Amennyiben ennek értéke 0, akkor a sor üres, egyébként pedig nem (4.14. algoritmus).

---

#### 4.12. Algoritmus Tömb alapú sorból elem kivétele

---

**Kimenet:** . - **T** (*a sorba legrégebben beszűrt érték*)

```
1: függvény TÖMBSOR.SORBÓL()
2: ha n > 0 akkor
3: n ← n – 1
4: e ← (e mod E.méret) + 1
5: érték ← E[e]
6: vissza érték
7: különben
8: hiba "Nincs elem"
9: elágazás vége
10: függvény vége
```

---

**Felhasznált változók és függvények**

- **E** : A tömb, ami tárolja az elemeket.
  - **e** : Az legutóbb olvasott elem indexe.
  - **n** : A sorban lévő elemek aktuális száma.
- 

---

#### 4.13. Algoritmus Tömb alapú sorból a legrégebbi elem lekérdezése módosítás nélkül

---

**Kimenet:** . - **T** (*a sorba legrégebben beszűrt érték*)

```
1: függvény TÖMBSOR.ELSŐ()
2: ha n > 0 akkor
3: vissza E[(e mod E.méret) + 1]
4: különben
5: hiba "Nincs elem"
6: elágazás vége
7: függvény vége
```

---

**Felhasznált változók és függvények**

- **E** : A tömb, ami tárolja az elemeket.
  - **e** : Az legutóbb olvasott elem indexe.
  - **n** : A sorban lévő elemek aktuális száma.
- 

---

#### 4.14. Algoritmus Tömb alapú sor ürességének lekérdezése

---

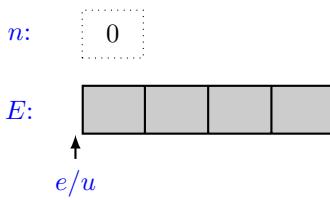
**Kimenet:** . - logikai (*üres a verem?*)

```
1: tulajdonság TÖMBSOR.ÜRES
2: vissza n = 0
3: tulajdonság vége
```

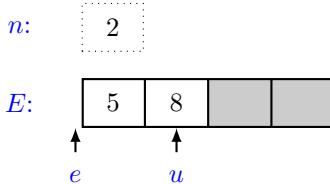
---

**Felhasznált változók és függvények**

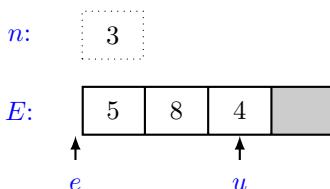
- **n** : A sorban lévő elemek aktuális száma.
-



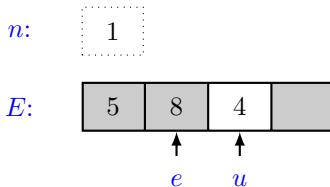
(a) Az üres sor alapértelmezett állapota (*n*=0). Az *e* és *u* változók értéke egyaránt 0.



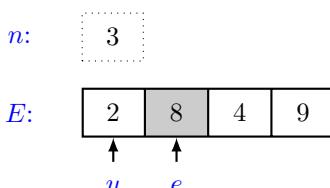
(c) Második elem (8) beszúrása a sorba.



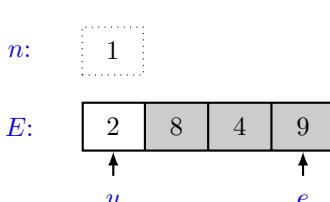
(e) Az *ELSŐ* művelet ilyen esetben az 5-öt adja vissza.



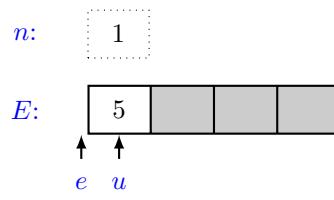
(g) Újabb elem kivétele a sorból (visszatérési értéke: 8).



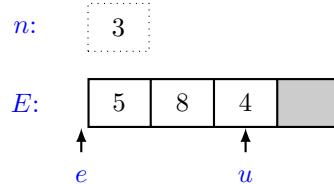
(i) A 2-es elem felvétele a sorba.



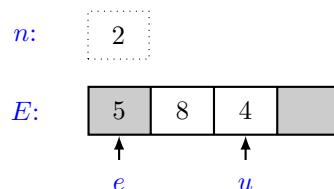
(k) Az *SORBÓL* művelet ilyen esetben a 9-et adja vissza.



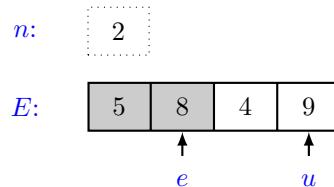
(b) Első elem (5) beszúrása a sorba.



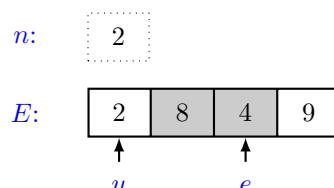
(d) Következő elem (4) beszúrása a sorba.



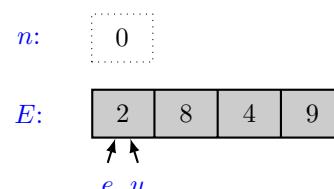
(f) Elem kivétele a sorból (visszatérési értéke: 5). Az érték ott marad, csak már nem része a sornak.



(h) Még egy elem (9) elhelyezése a sorba.



(j) *SORBÓL* művelet meghívása. Visszatérési értéke: 4.



(l) *SORBÓL* művelet meghívása. Visszatérési értéke: 2. Az *e* is visszatér az első elemre.

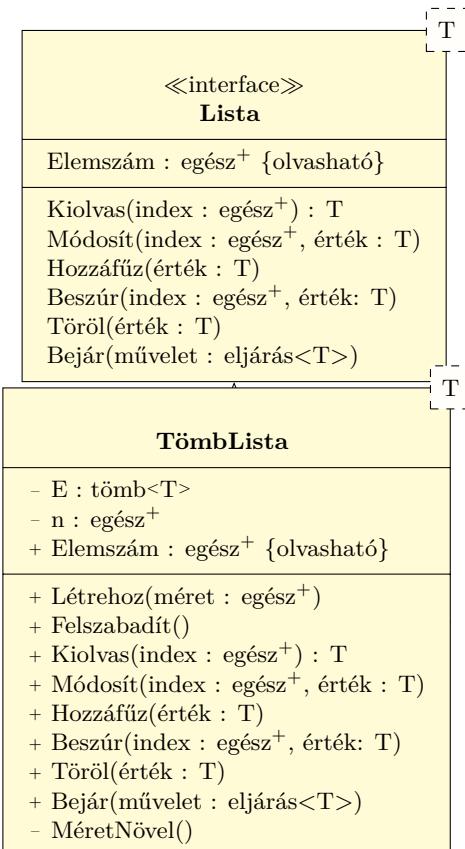
4.8. ábra. *SORBA* és *SORBÓL* műveletek bemutatása

## 4.4. Lista megvalósítása tömbbel

### 4.4.1. Alapvető felépítés

Habár a tömböt alapvetően egy statikus méretű adatszerkezetként képzeljük el, meg lehet vele oldani a **Lista** adatszerkezet műveleteit. Ahogy a 4.9. ábrán is látható, ehhez szükségünk lesz egy **E** tömbre, ami majd az elemeket tárolja, illetve egy **n** változóra, ami az elemek darabszámát. Egy tömb mérete a létrehozás után már nem változtatható, ami okoz némi problémát, az elemek számához való igazodást nekünk kell manuálisan kezelnie.

A műveletek megfelelnek a **Lista** interfésznel már megismertekkel, kiegészítve egy létrehozás és felszabadítás művelettel. A tömb alapú megvalósításhoz képest itt is látható az egyik előny, hogy a létrehozásnál ugyan itt is várunk egy paramétert, de ez nem az adatszerkezet maximális mérete, hanem csak az induló mérete. Több elem beszúrásakor az adatszerkezet mérete dinamikusan fog növekedni.



4.9. ábra. Lista megvalósítása tömb segítségével

Ennek egy megvalósítása lehet az, hogy az elemeket tartalmazó tömböt egy alapértelmezett vagy inicializáció során megkapott mérettel hozzuk létre. Ebbe elkezdhetjük elhelyezni az új elemeket, amikor pedig ezek már nem térnek el, akkor egy új, nagyobb tömböt hozunk létre. Ennek mérete többféleképpen meghatározható, mi azt a megoldást választjuk, hogy minden duplázzuk a már meglévő méretet. A létrehozás után át kell másolni a már meglévő adatainkat az új tömbbe, és ezt követően ebbe helyezzük el a későbbi elemeket. Mindez a működés a **Lista** használója számára teljesen láthatatlan lesz.

### 4.4.2. Létrehozás és megszüntetés

A létrehozás alapvetően nem különbözik az eddig megoldásoktól. Létrehozzuk a paraméterként megadott méretű **E** tömböt, ami **T** típusú elemeket tud tárolni. Az **n** értékét pedig 0-ra állítjuk, ezzel is jelölve, hogy nincs még a listában egy elem se (4.15. algoritmus).

---

#### 4.15. Algoritmus Tömb alapú lista létrehozása

---

Bemenet: méret - egész<sup>+</sup> (a lista kezdő kapacitása)

- 1: eljárás TÖMBLISTA.LÉTREHOZ(méret)
  - 2:     $E \leftarrow \text{LÉTREHOZ}(\text{tömb}\langle T \rangle[\text{méret}])$
  - 3:     $n \leftarrow 0$
  - 4: eljárás vége
- 

#### Felhasznált változók és függvények

- $E$  : A tömb, ami tárolni fogja az elemeket.
  - $n$  : A listában lévő elemek aktuális száma.
  - $\text{LÉTREHOZ}(\text{tömb}\langle T \rangle[\text{méret}])$  : Létrehoz egy  $T$  típusú,  $méret$  méretű tömböt.
- 

Különleges művelet viszont a növelés. Mivel a tömböt úgy tekintjük, hogy létrehozása után már nincs lehetőség a méretének a megváltoztatására, ezért ezt csak úgy tudjuk megtenni, hogy létrehozunk egy új tömböt (miközben az  $E'$  referenciával eltároljuk a régi tömböt) a régi méretének duplájával, majd pedig ide átmásoljuk a már meglévő  $n$  darab elemet (4.16. algoritmus). Mivel nincs rá többé szükségünk, ezért felszabadítjuk az eredeti tömb által foglalt helyet.

---

#### 4.16. Algoritmus Tömb alapú lista kapacitásának növelése

---

- 1: eljárás TÖMBLISTA.MÉRETNÖVEL( )
  - 2:     $E' \leftarrow E$
  - 3:     $E \leftarrow \text{LÉTREHOZ}(\text{tömb}\langle T \rangle[E.\text{méret}*2])$
  - 4:    ciklus  $i \leftarrow 1$ -től  $n$ -ig
  - 5:      $E[i] \leftarrow E'[i]$
  - 6:    ciklus vége
  - 7:    FELSZABADÍT( $E'$ )
  - 8: eljárás vége
- 

#### Felhasznált változók és függvények

- $E'$  : Tömb referencia, amelyik ideiglenesen eltárolja az eredeti tömb címét.
  - $E$  : A tömb, ami tárolja az elemeket.
  - $n$  : A listában lévő elemek aktuális száma.
  - $\text{LÉTREHOZ}(\text{tömb}\langle T \rangle[\text{méret}])$  : Létrehoz egy  $T$  típusú,  $méret$  méretű tömböt.
  - $\text{FELSZABADÍT}(E')$  : Felszabadítja az  $E'$  referencia által hivatkozott tömböt.
- 

A teljes adatszerkezet megszüntetése során csak a benne lévő tömböt kell felszabadítanunk, miként az látható is a 4.17. algoritmuson).

---

#### 4.17. Algoritmus Tömb alapú lista felszabadítása

---

- 1: eljárás TÖMBLISTA.FELSZABADÍT( )
  - 2:    FELSZABADÍT( $E$ )
  - 3: eljárás vége
- 

#### Felhasznált változók és függvények

- $\text{FELSZABADÍT}(E)$  : Felszabadítja az  $E$  tömböt.
- 

### 4.4.3. Műveletek megvalósítása

#### Elemszám meghatározása

A *Elemszám* tulajdonság egyszerűen megvalósítható, hiszen az  $n$  változó mindenkorban a listában lévő elemek számát tartalmazza. A tulajdonságnak egyszerűen ezt az értéket kell visszaadnia.

## Beszúrás megadott helyre

A lista sajátossága, hogy az elemeit egy megadott sorrendben tartja, azokhoz az indexeken keresztül lehet hozzáérni. Ehhez az is hozzátarozik, hogy megadott indexű helyre be tudjuk szűrni egy új elemet. Ezt mutatja be a 4.18. algoritmus.

---

### 4.18. Algoritmus Tömb alapú listába elem beszírása

---

**Bemenet:**  $index$  - egész<sup>+</sup> (a beszúrandó új érték helye a listában)

**Bemenet:** érték -  $T$  (a beszúrandó új érték)

```
1: eljárás TÖMBLISTA.BESZÚR(index, érték)
2: ha $index \leq n + 1$ akkor
3: ha $n = E.méret$ akkor
4: MÉRETNÖVELÉS()
5: elágazás vége
6: $n \leftarrow n + 1$
7: ciklus $i \leftarrow n$ -től $index + 1$ -ig visszafelé
8: $E[i] \leftarrow E[i - 1]$
9: ciklus vége
10: $E[index] \leftarrow$ érték
11: különben
12: hiba "Hibás index"
13: elágazás vége
14: eljárás vége
```

---

#### Felhasznált változók és függvények

- $E$  : A tömb, ami tárolja az elemeket.
  - $n$  : A listában lévő elemek aktuális száma.
  - MÉRETNÖVELÉS( ) : Létrehoz egy új, nagyobb  $E$  tömböt (4.16. algoritmus).
- 

Elsőként megvizsgáljuk, hogy a paraméterként átadott  $index$  megfelelő-e. Ennek értéke nem lehet nagyobb, mint  $n + 1$ , ilyenkor hibát kell jelezni. A paraméter 1-es értéke azt jelzi, hogy a legelső elem elő szeretnénk beszűrni, az  $n + 1$ -es érték pedig azt, hogy a legutolsó után. A kettő közötti értékek esetén minden két elem közé szűrünk be.

A beszúrásnak nincsenek további feltételei, ezért tudható, hogy minden híváskor növekedni fog a tömbben lévő elemek száma. Ezért a következő lépésként megnézzük, hogy elfér-e az új elem a tömbben. Ha az  $n$  értéke már egyenlő az  $E$  tömb méretével, akkor nem. Ilyenkor meghívjuk az előzőleg megismert MÉRETNÖVELÉS eljárást.

A 6. sorban emiatt már tudható, hogy van szabad hely a tömbben. Ezért növeljük az  $n$  értékét, ezzel jelölve, hogy egyelőre tömb elemünk lesz az adatszerkezetben.

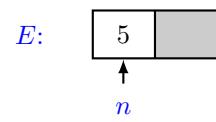
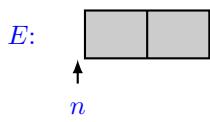
Amennyiben nem a tömb végére szúrjuk be az új elemet, akkor létre kell hozni számára egy üres helyet. Ezt végzi el a következő ciklus, ami minden, a beszúráskor megadott index utáni elemet egyelőre másol. Vegyük észre, hogy a másolást hátulról előrefelé kell végrehajtanunk, hogy ne írjuk felül a később másolandó elemeket.

Miután ez is elkészült, elhelyezhetjük a paraméterként átadott értéket a tömbben a szintén paraméterként átadott helyen.

#### Lista végére hozzáfűzés

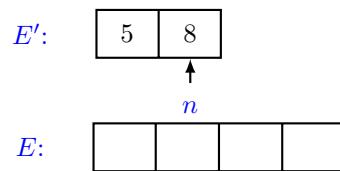
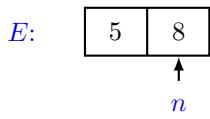
Külön műveletként határoztuk meg a hozzáfűzést, ami a lista végére helyezi el az eltárolni kívánt elemet. Maga az eltárolás egyszerű, hiszen csak növelni kell az  $n$  értékét, és elhelyezni ezen a helyen az új értéket. Előtte viszont itt is ellenőriznünk kell, hogy van-e még hely, és ha nincs, akkor növelni kell a tömböt.

Hogy elkerüljük a kódok felesleges duplikálását, ezért ezt a műveletet egyszerűen visszavezetjük az előzőre, ugyanis vegyük észre, hogy  $n + 1$ -et adva át paraméterként, az tökéletesen elvégzi ezt a feladatot (4.19. algoritmus).



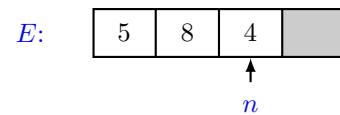
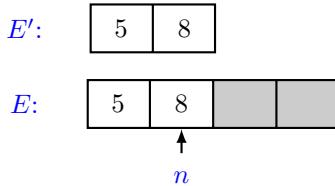
(a) Az üres lista alapértelmezett állapota ( $n=0$ ). Jelen esetben a létrehozáskor megadott méret 2, ezért ekkora tömb jött létre.

(b) Első elem (5) beszúrása a listába az első pozícióba.



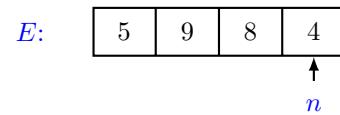
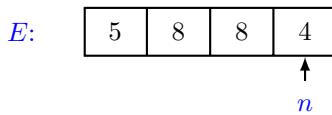
(c) Második elem (8) beszúrása a listába a második pozícióba.

(d) Következő elem (4) beszúrása a listába a harmadik pozícióba. Az eddigi tömb már nem elég, ezért létrehozunk egy újat. A régire hivatkozik az  $E'$ , az új tömb lesz az  $E$ .



(e) Átmásoljuk az  $E'$  tömb  $n$  darab elemét az  $E$  tömbbe.

(f) Az új tömb létrehozása után lefut a már megszokott beszúrás. Az  $E'$  pedig megszűnik.



(g) Beszúrjuk a következő elemet (9) a második indexet megadva. Elsőként növeljük az  $n$  értékét eggyel, majd a ciklus eggyel hátrébb másolja a második, vagy annál nagyobb indexű elemeket.

(h) Ezt követi az új érték bemásolása a megadott indexű helyre.

4.10. ábra. Listába beszúrás művelet működésének bemutatása.

#### 4.19. Algoritmus Tömb alapú listához elem hozzáfűzése

**Bemenet:** érték -  $\mathbf{T}$  (az új hozzáfűzendő érték)

1: **eljárás** TÖMBLISTA.HOZZÁFŰZ(értek)

2:      BESZÚR( $n + 1$ , érték)

3: **eljárás vége**

#### Felhasznált változók és függvények

- $n$ : A listában lévő elemek aktuális száma.
- **BESZÚR( $i, x$ )**: Beszúrja a lista  $i$ . helyére az  $x$  értéket (4.18. algoritmus).

## Hozzáférés az elemekhez

A tömb sajátossága, hogy az elemeihez az indexeken keresztül lehet hozzáférni. Vegyük észre, hogy az egyes elemek listabeli indexei tulajdonképpen megegyeznek a tömbbeli pozícióikkal. Emiatt a kiolvasás és a módosítás nagyon egyszerűen megoldhatók (4.20. és 4.21. algoritmusok). Ezknél csak arra kell figyelnünk, hogy ne lehessen „túlcímezni” a tömböt, ilyenkor hibát jelzünk.

---

### 4.20. Algoritmus Tömb alapú listából elem kiolvasása

---

**Bemenet:**  $index - \text{egész}^+$  (a kiolvasandó elem indexe)

**Kimenet:**  $. - T$  (a megadott indexű elem)

- 1: függvény TÖMLISTA.KIOLVAS( $index$ )
  - 2:   ha  $index \leq n$  akkor
  - 3:     vissza  $E[index]$
  - 4:   különben
  - 5:     hiba "Hibás index"
  - 6:   elágazás vége
  - 7: függvény vége
- 

#### Felhasznált változók és függvények

- $E$  : A tömb, ami tárolja az elemeket.
  - $n$  : A listában lévő elemek aktuális száma.
- 

---

### 4.21. Algoritmus Tömb alapú listában egy érték módosítása

---

**Bemenet:**  $index - \text{egész}^+$  (a módosítandó érték indexe)

**Bemenet:**  $\text{érték} - T$  (az új érték)

- 1: eljárás TÖMLISTA.MÓDOSÍT( $index$ ,  $\text{érték}$ )
  - 2:   ha  $index \leq n$  akkor
  - 3:      $E[index] \leftarrow \text{érték}$
  - 4:   különben
  - 5:     hiba "Hibás index"
  - 6:   elágazás vége
  - 7: eljárás vége
- 

#### Felhasznált változók és függvények

- $E$  : A tömb, ami tárolja az elemeket.
  - $n$  : A listában lévő elemek aktuális száma.
- 

## Törlés érték alapján

A **Lista** interfészen a törlés azon változatát valósítottuk meg, amelyik egy értéket vár paraméterként, és ezt törli ki az adatszerkezetből. Amennyiben a megadott érték többször is szerepel a listában, akkor annak minden előfordulását törölni szeretnénk. Természetesen elképzelhető index szerinti törlés is, és még számos egyéb megvalósítás (pl. megadott feltételnek megfelelő elemek törlése), ezeket hely hiányában nem tárgyaljuk.

A 4.22. algoritmus alapötlete az, hogy egy  $i$  számlálós ciklussal végignézzük a tömb elemeit az elsőtől az utolsóig, miközben egy  $db$  nevű változóban tároljuk a megtalált törlendő elemek számát (ennek kezdőértéke 0). Amennyiben az  $E[i]$  értéke egyenlő a törlendővel, akkor növeljük a  $db$  változó értékét. Amennyiben egy nem törlendő elemet találunk, akkor azt a tömbben előrébb helyezzük  $db$  pozícióval, tehát az  $i - db$  indexű helyre.

Miután a ciklus végetért, az ábrán is látható módon, a megmaradó elemekkel felülírtuk a törlendő elemeket. Hátra van még az  $n$  értéknek csökkentése. Mivel tudjuk, hogy  $db$  darab törlendő elemet találtunk, így ezzel az értékkel végezzük el a csökkentést is.

A beszúráskor minden új tömböt hoztunk létre, amikor az elemek már nem fértek el a meglévő  $E$ -ben. Itt is felmerülhet, hogy a törlés során, ahogy egyre kisebb lesz a tömb, úgy szüntessük meg a meglévőt, és hozzunk létre helyette egyre kisebbeket. A gyakorlatban azonban ezt általában nem használjuk,

---

#### 4.22. Algoritmus Tömb alapú listából a megadott érték törlése

---

**Bemenet:** érték -  $T$  (a törlendő érték)

```
1: eljárás TÖMBLISTA.TÖRÖL(érték)
2: db ← 0
3: ciklus $i \leftarrow 1$ -től n -ig
4: ha $E[i] = \text{érték}$ akkor
5: db ← db + 1
6: különben
7: $E[i - db] \leftarrow E[i]$
8: elágazás vége
9: ciklus vége
10: $n \leftarrow n - db$
11: eljárás vége
```

---

#### Felhasznált változók és függvények

- $db$  : A ciklus futása közben az érték-kel megegyező tartalmú elemek száma.
  - $E$  : A tömb, ami tárolja az elemeket.
  - $n$  : A listában lévő elemek aktuális száma.
- 

feltételezzük, hogy a későbbiekben még bármikor kerülhet ugyanannyi elem a listába, ezért megtartjuk a nagyobb (bár aktuálisan kihasználatlan) tömb méretét.

#### Lista bejárása

A bejárás művelete nem igényel különösebb magyarázatot. Paraméterként kapunk egy függvényt, amit minden elemre meg kell hívnunk. Ennek megfelelően egy ciklussal végigmegyünk a tömbben lévő eltárolt értékeken, minden iterációban meghívjuk a paraméterként átadott függvényt, paraméterként átadva a tömb megfelelő elemét.

---

#### 4.23. Algoritmus Tömb alapú lista bejárása

---

**Bemenet:** művelet - eljárás $<T>$  (az elemeken végrehajtandó művelet)

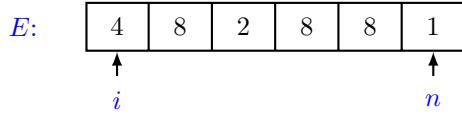
```
1: eljárás TÖMBLISTA.BEJÁRÁS(művelet)
2: ciklus $i \leftarrow 1$ -től n -ig
3: művelet($E[i]$)
4: ciklus vége
5: eljárás vége
```

---

#### Felhasznált változók és függvények

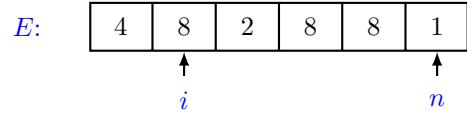
- $E$  : A tömb, ami tárolja az elemeket.
  - $n$  : A listában lévő elemek aktuális száma.
-

*db:* 



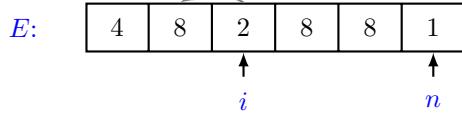
(a) A listában mint látható, 6 darab érték található. Az *i* ciklus 1-től indul. Mivel az első elem nem törlendő, ezért a *db* értéke 0, így Az *E[1]* helyre ugyanazt az *E[1]*-et másolja.

*db:* 



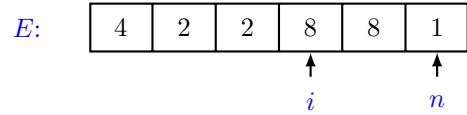
(b) A következő ciklus iterációnál az *E[i]* egy törlendő elemre mutat. Ezért itt nincs másolás, csak növeljük a *db* értékét 1-re.

*db:* 



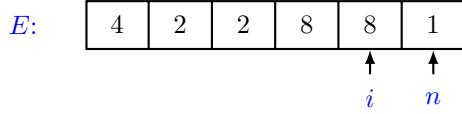
(c) A harmadik elem nem törlendő, ezért ezt előre másoljuk. A *db* értékének megfelelően 1-el kerül előrébb ez az elem.

*db:* 



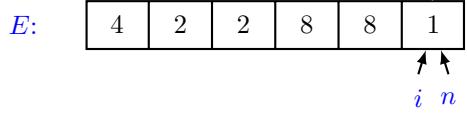
(d) A negyedik elem szintén törlendő, ezért nincs másolás, csak a *db*-t növeljük 2-re.

*db:* 



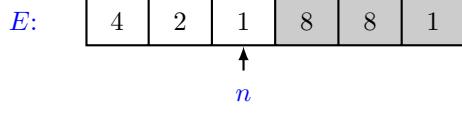
(e) Az ötödik elem szintén törlendő, így itt is növeljük a *db*-t 3-ra.

*db:* 



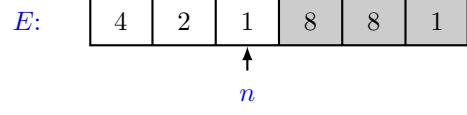
(f) Az utolsó elem nem törlendő, ezért előre másoljuk a *db* értékének megfelelően 3 pozícióval.

*db:* 



(g) Végül csökkentjük az *n* értékét *db*-vel, mivel ennyi elemet törlöttünk a listából.

*db:* 



(h) A tömbben ugyan benne marad a többi elem is, de ezeket már nem tekintjük a lista részének, csak az *n*-ig terjedő részt.

4.11. ábra. Listából törlés művelet működésének bemutatása. A listából a 8-as elemet töröljük.

## 5. fejezet

# Egyszeresen láncolt adatszerkezetek

### 5.1. Láncolt szerkezetek felépítése

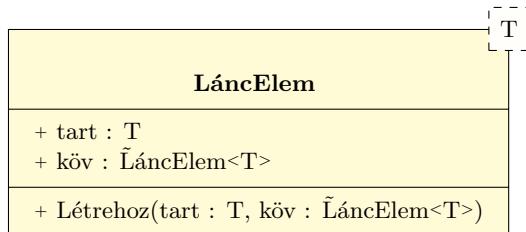
#### 5.1.1. Alapvető felépítés

Az előzőleg megismert tömb alapú implementációk után megkezdjük a dinamikus adatszerkezetek vizsgálatát. A dinamikus jelleg arra vonatkozik, hogy az adatszerkezet mérete futás közben folyamatosan változhat, új elemek beszúrása során növekszik, törlés esetén pedig csökken. Ugyan a tömb alapú listánál is el tudtuk érni ezt a működést, de ott az elemek másolása jelentősen lerontotta a beszúrás teljesítményét.

A legegyszerűbb dinamikus adatszerkezeteket egyszerű láncolással tudjuk felépíteni<sup>1</sup>. Ennek eredménye egy olyan sorozat, amelyen az első elemtől kezdődően sorban lehet haladni [8] egy, minden a következő elemre mutató hivatkozáson keresztül. Alapelve az, hogy az elemeket a memoriában nem közvetlenül egymás mellett tároljuk (mint a tömbök esetében), hanem azok tetszőleges helyen helyezkedhetnek el, a köztük lévő kapcsolatot pedig úgy valósítjuk meg, hogy mindegyik elem egy hivatkozást tárol az öt követő elemre. A legelső elem ilyen szempontból speciálisnak tekinthető, erre egy külső hivatkozás mutat.

#### 5.1.2. Egy elem felépítése

A láncelemeket egy összetett típussal tudjuk megvalósítani, amely az 5.1. ábrán látható.

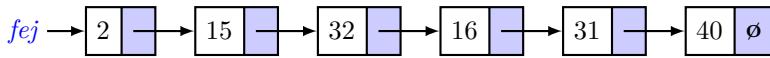


5.1. ábra. Láncolt szerkezet egy elemének felépítése.

Az adatszerkezet egy eleme az alábbi mezőkkel rendelkezik:

- Tartalmi rész: ez a rész tartalmazza magát, a láncelemben eltárolni kívánt értéket. Nincs semmiféle megkötésünk ennek típusára, lehet szám, szöveg, vagy akár tetszőleges objektum típus is. Azt általában kikötjük, hogy az egyes láncelemelek minden azonos típusú adatokat tartalmazzanak, de OOP környezetben persze erre sincs feltétlenül szükség. A példákban ezt a mezőt mindig *tart* néven fogjuk használni, típusát pedig **T**-vel fogjuk jelölni, igazodva a generikus osztályoknál látottakhoz.
- Következő elemre hivatkozó rész: a lánc minden eleme tartalmaz egy hivatkozást a listában rakkövetkező elemre. A legelső elemtől elindulva így (közvetve, a közbülső elemeken keresztül) elérhetjük

<sup>1</sup>Ezt általában egyszerűen „láncolt listának” szoktuk nevezni, viszont ez nem teljesen precíz, mivel ezzel a technikával nem csak listát, hanem sort, vermet, halmazt, stb. is lehet készíteni. Ezért próbálunk inkább a „láncolt szerkezet” vagy egyszerűen csak „lánc” elnevezésnél maradni.



5.2. ábra. Láncolt lista példa. Az eltárolt elemek: 2, 15, 32, 16, 31, 40.

a lánc bármelyik elemét. A mező típusa a lánc megvalósításától függően többféle lehet, erre még a későbbi, implementációs résznél visszatérünk (addig is a legegyszerűbb a lánc elemeit objektumként, a következő mezőt pedig objektum referenciaként felfogni). A példákból ezt a mezőt minden *köv* néven fogjuk használni, típusa pedig egy hivatkozás egy ugyanilyen típusú másik láncelemre. Ennek jelölése: *LáncElem<T>*.

### 5.1.3. Lánc felépítése

A legelső elemre is kell rendelkeznünk egy hivatkozással, ez egy külső hivatkozás kell, hogy legyen, a példákban ezt minden *fej* változóval fogjuk jelölni. A lánc legutolsó eleme szintén speciálisnak tekinthető, hiszen a *köv* hivatkozás itt már nem tud egy valóban létező elemre mutatni. Emiatt ebbe a mezőbe minden egy *ø*-al jelölt speciális lezáró értéket helyezünk.

Külön érdemes foglalkozni az üres listákkal. Ezeket minden úgy fogjuk értelmezni, hogy a *fej* változó tartalma már eleve a *ø* lezáró elem. Ezt nem tekintjük hibának, ezzel csak azt jelöljük, hogy a lánc ugyan létezik, de éppen nincsenek benne elemek.

Az 5.2. ábra mutat egy minta láncolt listát. Látható, hogy a *fej* mutató a 2-es tartalmú elemre mutat, ezt tekintjük a lánc első elemének. Ennek a tartalma "2", a *köv* mezője pedig a 15-ös tartalmú elemre hivatkozik. Amennyiben a *köv* mezőket követjük, akkor kialakul a láncolt listában tárolt elemek sorrendje: 2, 15, 32, 16, 31, 40. A 40-es tartalommal bíró láncelem következő mezőjének értéke *ø*, ezért tudható, hogy itt véget ért a láncolt szerkezet.

### 5.1.4. Értékelése

A láncolt szerkezetek számos előnyvel rendelkeznek a hagyományos tömb alapú megoldásokkal szemben:

- Mérete dinamikusan változik: a tömb egyik legnagyobb hátránya az volt, hogy már létrehozásakor meg kellett határozni a méretét, és a későbbiekben ez nem változtatható. A láncolt szerkezet esetében ilyen problémánk nincs, mivel a dinamikus megvalósításokban az egyes elemek egymástól független objektumokként léteznek, így bármikor felvehetünk egy újat, vagy akár törölhetünk is egyet, csak arra kell ügyelni, hogy a *köv* mutatókon keresztül végigjárt út minden pontosan a lánc elemeit tartalmazza.
- Nem igényel összefüggő memóriaterületet: bár napjainkban ez kevésbé okoz problémákat, de érdekes megjegyezni, hogy a tömb (felépítéséből adódóan) minden egy összefüggő memóriaterületen helyezkedik el. A lánc elemei a memóriában bárhol lehetnek szétszórva, azok egymáshoz való közelisége, illetve sorrendje lényegtelen, hiszen úgyis csak a *köv* mutatók határozzák meg az elemek logikai sorrendjét. Ezzel elkerüljük azt a futásidőben jelentkező problémát, amikor van még ugyan elég szabad hely a memóriában, de ez nem összefüggő, így nem tudunk létrehozni megfelelő méretű tömböt.
- Gyorsabb módosítás: a tömbök esetén az új elem beszúrása és az elemek törlése meglehetősen erőforrás-igényes művelet. Az előbbi esetben ugyanis (ha a tömb közepébe szúrunk be), akkor hátrébb kell tolni a mögötte lévő elemeket, és ehhez hasonlóan a törléskor is előre kell húzni a törölt mögötti elemeket. A későbbiekben részletesen megvizsgáljuk a lánc módosítási műveleteket, de talán a felépítéséből is látható, hogy ezek a műveletek jóval egyszerűbbek lesznek, hiszen az elemek tényleges mozgatása nélkül, csak a *köv* mezők lokális módosításával könnyen tudjuk az új elemeket beláncolni, illetve a törlendő elemeket kiláncolni a listából. Mivel ezek a műveletek nem érintik a teljes listát, csak a módosítandó elem közvetlen környezetét, így ezek lépésszáma független az elemszámtól, konstans időben megvalósíthatók.

A láncolt szerkezet persze néhány hátránnal is jár:

- Nagyobb helyfoglalás: minden láncelemben a tényleges eltárolt adattartalom mellett tárolnunk kell egy újabb mutatót is, ami a következő elemre mutat. Amennyiben az adattartalom csak egy egyszerű szám, akkor ez akár a tárigény duplázódását is jelentheti.
- Közvetlen elérés (véletlen elérés) hiánya: a tömb elemei mindig közvetlenül egymás mellett helyezkednek el a memóriában, így bármikor kiszámítható egy elem kezdőcíme. A láncolt szerkezet elemei azonban tetszőleges helyeken szétszórva lehetnek a memóriában, így itt lehetőség nincs. Az  $n$ . elemet csak úgy tudjuk elérni, ha az első elemről elindulva ( $n - 1$ ) darab lépésen keresztül követjük a *köv* mezők által megadott útvonalat. Ez jelentős hátrányt jelent azoknál az algoritmusoknál, amelyeknél közvetlenül szeretnénk elérni az adatszerkezet belső elemeit (pl. bináris keresés), ezek a láncolt listával csak nagyon nehézkesen valósíthatók meg (és ezzel gyakran az egész algoritmus értelmét veszti).

Az előnyök és hátrányok figyelembevételével elmondható, hogy a láncolt listák a gyakorlatban jól használhatók olyan környezetekben, ahol nagy mennyiségű adatot kell eltárolnunk, és az adatszerkezetet gyakran kell módosítanunk (elemek törlése, új elemek felvétele), viszont nincs szükségünk a belső elemek közvetlen elérésére.

Ha valaki a tömb alapú programozási tételeken keresztül ismerkedett meg a programozással, akkor ez utóbbi meglehetősen hátrányos korlátnak tűnik, hiszen ott folyamatosan tömb indexeléssel dolgoztunk. Azonban érdemes átgondolni, hogy ezen algoritmusok jelentős része (tipikusan azok, amelyeknél a ciklus az elemeket egyesével vizsgálja meg 1-től az elemszámig) egyszerűen módosítható úgy, hogy láncolt listákkal is hatékonyan működjön.

A láncolt szerkezeteknek számos változata ismert, az alábbiak szerint:

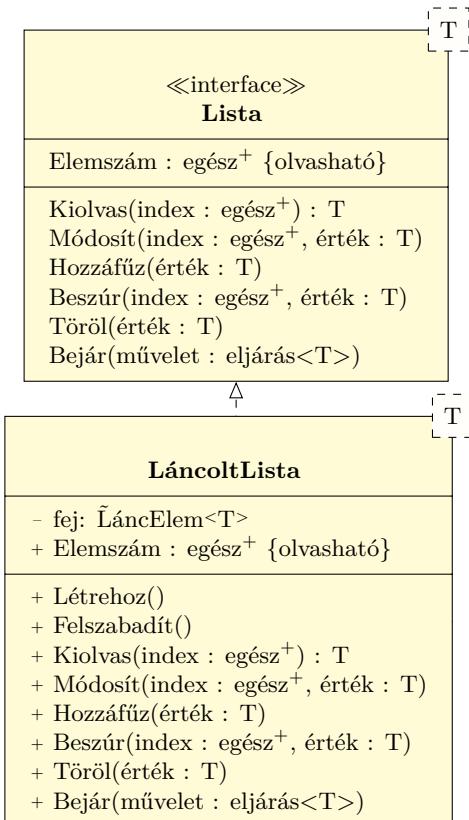
- Irányítottság alapján
  - Egyirányú láncolt szerkezet: minden láncelem csak a következő elemre tartalmaz hivatkozást. Erre mutat példát az 5.2. ábra.
  - Kétirányú láncolt szerkezet: minden láncelem a következő mellett az előző elemre is tartalmaz hivatkozást (5.6.2. alfejezet).
- Láncolások száma alapján
  - Egyszeresen láncolt (másnéven egyszerű) szerkezet: a szerkezet a benne lévő elemeknek egy sorrendjét tartalmazza (5.2. alfejezet).
  - Többszörösen láncolt szerkezet: a szerkezet nem csak egy, hanem több sorrendet is tartalmaz (5.6.3. alfejezet).

Emellett még léteznek különféle specializált listák, mint például a rendezett (5.5. alfejezet) vagy ciklikus (5.6.4. alfejezet) listák, illetve kiegészítő technikák, mint például a strázsa elemek használata (5.6.1. alfejezet).

## 5.2. Lista megvalósítása láncolt szerkezettel

### 5.2.1. Alapvető felépítés

A **Lista** adatszerkezet hatékonyan megvalósítható láncolt szerkezetekkel, mivel ezek jól kezelik a nagy mennyiségi elem tárolásával járó feladatokat. Az adatszerkezet ebben az esetben csak egy **fej** mezőt tartalmaz, amely a lánc első elemére hivatkozik.



5.3. ábra. Lista megvalósítása láncolt adatszerkezet segítségével.

Később látni fogjuk, hogy az új elem hozzáfűzése egy meglehetősen erőforrás-igényes művelet, mivel minden végig kell járnunk a listát, hogy megtaláljuk az utolsó elemet. Amennyiben erre gyakran van szükség, célszerű lehet felvenni egy kiegészítő mezőt, ami egy hivatkozást tárol a lista aktuális utolsó elemére.

Mint látható, a tervezett láncolt lista **T** típusú elemeket fog tudni eltárolni. Ez a generikus típus megjelenik mind a listánál, mind pedig a lista egyes elemeit tároló objektumoknál. Az egyszerű lista esetében nincs semmilyen megkötésünk arra, hogy abban milyen típust lehet tárolni.

#### Megjegyzés

Előzőleg említettük, hogy a láncolt megvalósítás kimondottan nehezen kezelni az indexeléssel operáló műveleteket, a **Lista** pedig elég sok ilyet tartalmaz. A valósában persze nem kötelező szigorúan tartani magunkat az ebben a jegyzetben megadott műveletekhez, ha csak adatokat tárolni és azokon műveleteket szeretnénk azok sorrendjétől függetlenül, akkor elég lehet egy lista elejére beszűrő művelet, illetve egy bejárás. Ezek pedig indexelés nélkül hatékonyan megvalósíthatók.

### 5.2.2. Létrehozás és megszüntetés

Az előzőekben láthattuk a láncolt listák alapvető felépítését. Az üres listának azt tekintettük, amikor a lista *fej* elemének értéke a speciális lezáró jellel egyenlő ( $\emptyset$ ). Egy frissen létrehozott listánál biztosítanunk kell ezt a tulajdonságot, ezt valósítja meg az 5.1. algoritmus. Ez csak annyit tesz, hogy beállítja a kezdőértéket a *fej* változonak.

A későbbiekben bemutatott algoritmusok során minden feltételezzük, hogy a láncolt lista inicializálva lett, tehát a *fej* vagy az első elemre mutat, vagy üres lista esetén a  $\emptyset$  értéket tárolja. Amennyiben az inicializálást egy osztály konstruktőrén hajtjuk végre, akkor ez természetesen adott.

---

#### 5.1. Algoritmus Láncolt lista létrehozása

---

- 1: eljárás LÁNCOLTLISTA.LÉTREHOZ( )
  - 2:     *fej*  $\leftarrow \emptyset$
  - 3: eljárás vége
- 

#### Felhasznált változók és függvények

- *fej* : Hivatkozás a láncolt lista fejére.
- 

A lista felszabadítása itt már némileg összetettebb lesz, mint amit a tömböknél láthattunk. Egy lépésből ezt nem is tudjuk megtenni, hiszen a teljes láncolt lista több, egymással csak egy hivatkozáson keresztül kapcsolódó objektumból áll. Ezért a felszabadításhoz végig kell egyesével lépegetni az összes elemen, és azokat egyesével kell felszabadítani.

A lista eleméhez rendelkezünk közvetlen referenciaival, ezért hatékony módszer az első elemek folyamatos törlése egészen addig, amíg kitöröljük az utolsó elemet (5.2. algoritmus). Ehhez csak egy ciklusra van szükség, amely addig törli a listát, amíg az nem üres (2. sor). A ciklusmag pedig folyamatosan kiláncolja a legelső elemet, tehát a *fej* mutatót átlépteti a másik elemre (4. sor), majd felszabadítja az ezzel feleslegessé vált elemet (5. sor).

Egy rövid példa látható a teljes lista törlésére az 5.4. ábrán. Ezen is látható, hogy az utolsó elem törlését követően a *fej* mutató értéke  $\emptyset$  lesz, tehát ezt nem szükséges külön beállítani.

---

#### 5.2. Algoritmus Láncolt lista felszabadítása

---

- 1: eljárás LÁNCOLTLISTA.FELSZABADÍT( )
  - 2:     ciklus amíg (*fej*  $\neq \emptyset$ )
    - 3:         *p*  $\leftarrow$  *fej*
    - 4:         *fej*  $\leftarrow$  *fej.köv*
    - 5:         FELSZABADÍT(*p*)
  - 6:     ciklus vége
  - 7: eljárás vége
- 

#### Felhasznált változók és függvények

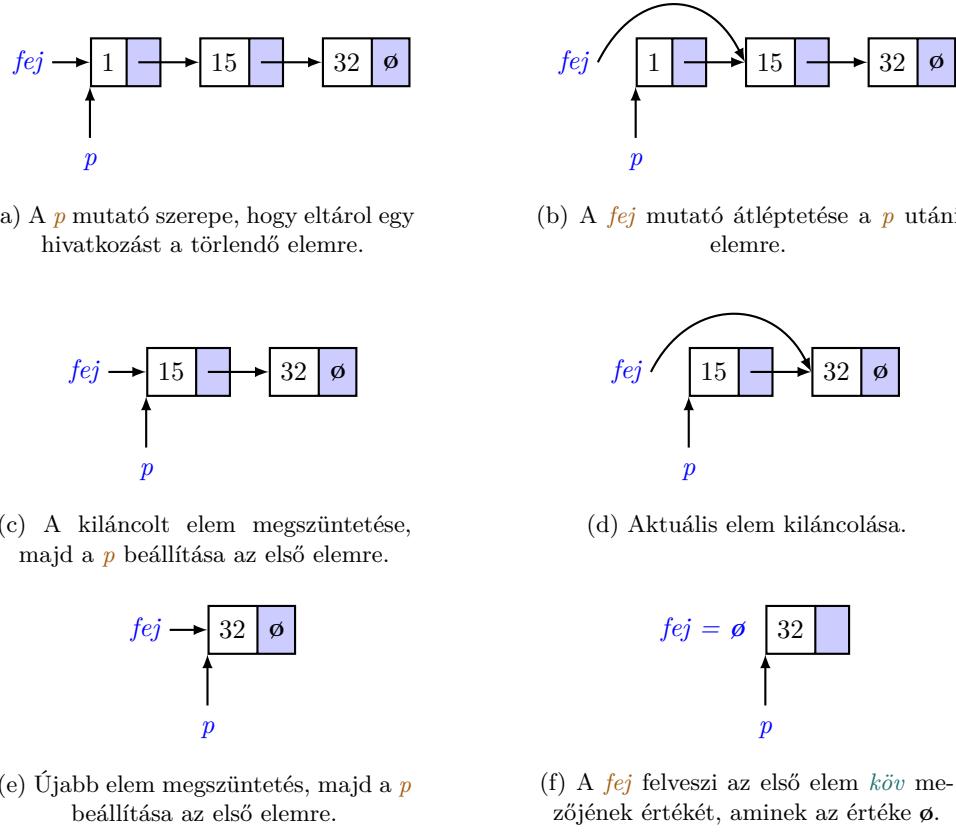
- *fej* : Hivatkozás a láncolt lista fejére. Az eljárás végén értéke  $\emptyset$  lesz.
  - *p* : Egy *LáncElem<T>* típusú segédváltozó.
  - *FELSZABADÍT(p)* : A *p* által mutatott elem felszabadítása.
- 

### 5.2.3. Új elem felvétele

#### A hozzáfűzés megvalósítása

Később látni fogjuk, hogy az első elem előtt beszűrás hatékonynak tekinthető, hiszen a láncolt lista méretétől függetlenül minden két lépést igényel az új elem beláncolása. A módszer egyik mellékhatása azonban az, hogy az elemek a listában a beszűrás sorrendjével ellentétes sorrendben tárolódnak. Amennyiben szükséges, hogy a lista pont ugyanabban a sorrendben tárolja az elemeket, mint ahogyan elhelyeztük benne, akkor mindenkor a lista végére kell felvenni az új elemet.

A lista végére beszűrás meglehetősen erőforrás-igényes művelet, ugyanis külső hivatkozással csak a lista első elemét érjük el, az ezt követőkhöz további lépések szükségesek. Az utolsó elem eléréséhez tehát egyesével végig kell lépkednünk a lista elemein, majd az utolsót megtalálva, ott beláncolhatjuk az újat.



5.4. ábra. Teljes láncolt lista törlése.

Ennek megfelelően az 5.3. algoritmusban is jól elkülöníthető ez a két fő lépés. Az 5.5. ábra pedig egy példát mutat a beszúrásra.

Az algoritmus indulásakor azonnal létrehozhatjuk az elemet, illetve kitölthetjük a *tart* és a *köv* mezők tartalmát is, hiszen az elem a lista végére fog kerülni, így ezek az értékek mindenkor megadott tartalom, illetve a  $\emptyset$  értéket kapják meg (még abban az esetben is, ha egy üres listába szűrjük be az elemet, tehát ez lesz az első és egyben utolsó láncszem is). Erre az újonnan létrehozott *LáncElem* objektumra hivatkozik az *új* változó.

A 3. sorban található ellenőrzés azt vizsgálja meg, hogy van-e már elem a listában. A beláncolás általános esete azt jelentené, hogy megkeressük a lista végét, majd az utolsó elem *köv* mezőjébe egy hivatkozást helyezünk el az újonnan létrehozott elemre. Speciális esetnek tekintethetjük azt, ha üres listába akarunk beszúrni, hiszen itt még egy elem sincs a listában, amihez hozzá lehetne fűzni. Emiatt ilyenkor a *fej* hivatkozást kell módosítanunk, az így lefutó beláncolás tulajdonképpen megegyezik a lista elejére beszúrással.

Amennyiben vannak elemek a listában, akkor meg kell keresni ezek közül az utolsót. Ehhez egy *p* segédváltozót használunk, amelynek kezdeti értéke (6. sor) egy hivatkozás lesz a lista első elemére (mivel az előbb tisztáztuk, hogy a lista nem üres, így ez biztosan egy létező elemre fog mutatni). Az ezt követő ciklus (7. sor) szerepe, hogy elkezd egyesével lépegetni a listában, egészen addig, amíg el nem jut az utolsó elemig. A keresett elem jellegzetessége, hogy a *köv* mezőjének értéke  $\emptyset$  lesz, ez a listában csak a legutolsó elemre lehet igaz. Emiatt a ciklus egészen addig fut, amíg a *p* hivatkozással nem találtuk meg ezt az elemet. Amennyiben a kilépési feltételünk nem teljesült, akkor éppen a lista valamelyik belső elemén állunk. Ilyenkor a *p* változóval egyszerűen átléphetünk a következő elemre (8. sor). Ezt a technikát a későbbiekben még sokszor fogjuk látni, mivel ez az egyetlen mód, hogy a láncolt lista belső elemeit feldolgozzuk.

Az előző ciklus előbb-utóbb véget ér, és a kilépéskor a *p* éppen a lista utolsó elemére hivatkozik (5.5d. ábra). Ezen a ponton az *új* hivatkozik az általunk létrehozott új elemre, a *p* pedig a lista eddigi utolsó elemére. Az új elem beláncolása itt már egyszerűen megoldható, csak az utolsó elem *köv* hivatkozását kell átirányítanunk az újonnan létrehozott elemre (10. sor).

---

### 5.3. Algoritmus Új érték hozzáfűzése a láncolt lista utolsó eleméhez

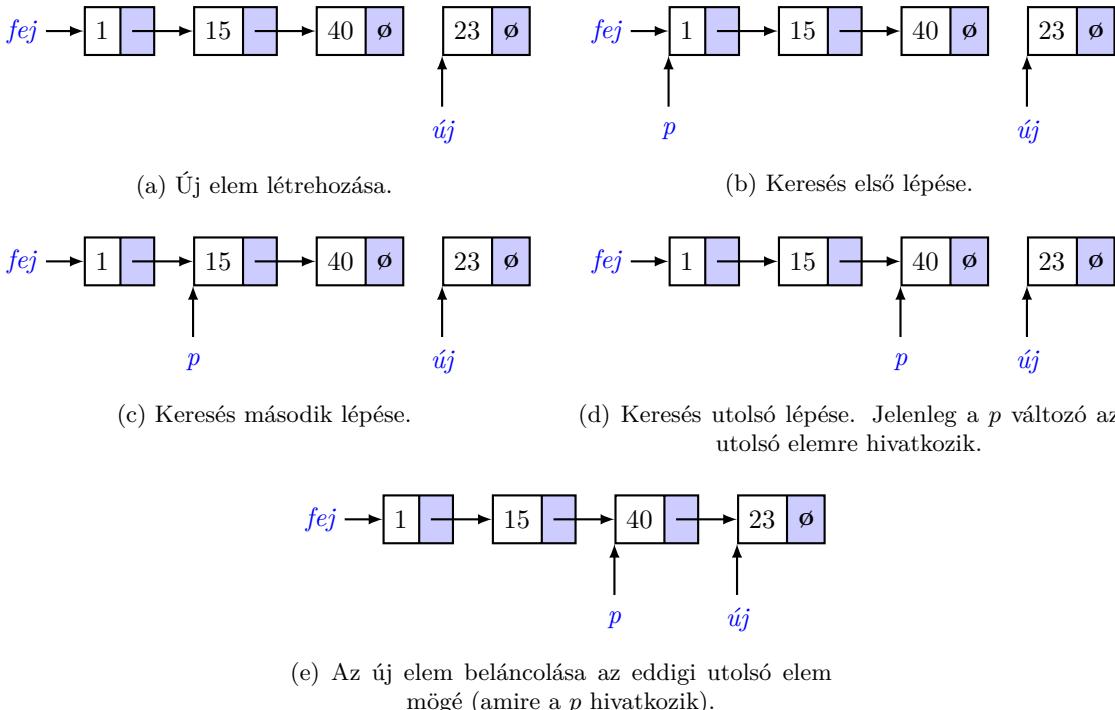
---

**Bemenet:** érték - T (a beszúrandó érték)

- 1: **eljárás** LÁNCOOLTLISTA.HOZZÁFŰZ(érték)
  - 2:     *új*  $\leftarrow$  LÉTREHOZ(*LáncElem*<T>) {*tart*  $\leftarrow$  érték, *köv*  $\leftarrow$   $\emptyset$ }
  - 3:     **ha** *fej* =  $\emptyset$  **akkor**
  - 4:         *fej*  $\leftarrow$  *új*
  - 5:     **különben**
  - 6:         *p*  $\leftarrow$  *fej*
  - 7:         **ciklus** amíg *p.köv*  $\neq$   $\emptyset$
  - 8:             *p*  $\leftarrow$  *p.köv*
  - 9:         **ciklus vége**
  - 10:        *p.köv*  $\leftarrow$  *új*
  - 11:        **elágazás vége**
  - 12: **eljárás vége**
- 

**Felhasznált változók és függvények**

- *fej* : Hivatkozás a láncolt lista fejére (ami esetleg meg is változhat).
  - *új*, *p* : *LáncElem*<T> típusú segédváltozók.
  - LÉTREHOZ(*LáncElem*<T>) : Létrehoz egy új láncelementet, és visszaad egy hivatkozást erre.
-



5.5. ábra. Láncolt lista végére fűzés. A példában a 23 értéket fűzzük fel a lista utolsó eleme mögé.

### Beszúrás művelet megvalósítása

Ezt tekinthetjük a beláncolás általános esetének, amikor a lista két, már meglévő eleme közé kell elhelyeznünk egy új elemet. A feladatot itt is két részre bonthatjuk, először meg kell keresnünk a megfelelő elemeket, amelyek közé az újat be kell majd szűrni, majd pedig ennek megfelelően el kell végezni magát a beláncolást.

Az 5.4. algoritmus ennek egy lehetséges megvalósítását mutatja (az 5.6. ábra pedig egy példát mutat ennek működésére). Az algoritmus egy ellenőrzéssel indul, mivel két különböző esetet kell megkülönbözteni:

- A lista első helyére kell beszúrnunk, ami két esetben fordulhat elő: az *index* értékének 1-et adunk meg, tehát az aktuális első elem elé kell beszúrni, vagy pedig üres a lista, így a beszúrt új elem (az *index* értékétől függetlenül) az első lesz. A két eset abban hasonlít egymáshoz, hogy minden eset a *fej* változó módosításával jár (mivel a *fej* mindenkor az első elemre mutat, így az első elem cseréje értelemszerűen ezt vonja magával).
- A lista közbülső vagy utolsó helyére kell beszúrnunk. Egy közbülső helyre beszúrás során az előző elem *köv* mezőjének az új elemre kell majd mutatnia, míg az újonnan felvett elem *köv* mutatójának az öt követő elemre. Speciális esetnek tűnik a legutolsó elem mögé beszúrás, de az algoritmus áttekintése során látni fogjuk, hogy ez nem igényel külön programát.

A 2. sor feltehetően a fent említett ellenőrzést. A *fej* változó aktuális értékéből meg tudjuk határozni, hogy a lista éppen üres-e, illetve az *index* értékéből látható, hogy az első elem előtt szeretnénk-e beszúrni. Amennyiben a kettő közül bármelyik is igaz, akkor a 3. és a 4. sorok segítségével felvesszük az új elemet a lista elejére.

Abban az esetben, ha egy belső elemet kell felvennünk, egy ciklussal meg kell keresni annak a helyét. A lista végére beszúrásnál már láthattuk, hogy hogyan tudunk egyesével lépkedni a listában. A különbség pusztán annyi, hogy ebben az esetben nem kell egészsen a lista végéig eljutni, elég megtalálni az *index*. helyet (ehhez a jelenlegi (*index* - 1). elemet kell megkeresnünk, hiszen az újonnan felvett láncelemet majd ez után kell beláncolni). A 8. sorban látható ciklusfeltételek figyelik, hogy a ciklus csak addig fússon, amíg nem léptünk az utolsó elem mögé (ez csak abban az esetben fordulhat elő, ha a paraméterként átadtott *index* érték nagyobb, mint a listában található elemek száma), vagy elérteük a szükséges elemet.

Az *i* segédváltozó minden továbblépésnél egyelőre növekszik, így minden lépésben mutatja, hogy éppen hányadik elem előtt állunk.

A ciklusból való kilépést követően megtörténik a beláncolás (13–14. sor). Ennek két lépését mutatják az 5.6d. és 5.6e. ábrák.

A legutolsó elem mögé beszúrás speciális esetnek tűnik, de valójában a megadott algoritmus ezt is kezeli. A ciklus megállításakor a *p* ilyenkor a lista utolsó elemén áll. A 13. sor ennek megfelelően átmásolja a *p.köv* mezőben található  $\emptyset$  értéket az *új* lista elembe, ami helyes, mivel valóban  $\emptyset$  lesz a lista utolsó eleme. A *p* után láncolás (14. sor) pedig már hasonlóan működik, mint az általános esetben.

A beszúrás előtt még ellenőrizzük, hogy nem lépte-e túl az *i* a láncolt lista méretét. Ha igen, akkor ezt egy hibaüzenettel jelöljük.

---

#### 5.4. Algoritmus Beszúrás a láncolt lista index-edik helyére

---

Bemenet: *index - egész<sup>+</sup>* (a beszúrandó érték helye)

Bemenet: *érték - T* (a beszúrandó érték)

1: eljárás LÁNCOLTLISTA.BESZÚRÁS(*index, érték*)

2:   ha (*fej* =  $\emptyset$ )  $\vee$  (*index* = 1) akkor

3:     *új*  $\leftarrow$  LÉTREHOZ(*LáncElem<T>*) {*tart*  $\leftarrow$  érték, *köv*  $\leftarrow$  *fej*}

4:     *fej*  $\leftarrow$  *új*

5:   különben

6:     *p*  $\leftarrow$  *fej*

7:     *i*  $\leftarrow$  2

8:     ciklus amíg (*p.köv*  $\neq$   $\emptyset$ )  $\wedge$  (*i* < *index*)

9:       *p*  $\leftarrow$  *p.köv*

10:      *i*  $\leftarrow$  *i* + 1

11:     ciklus vége

12:     ha *i*  $\leq$  *index* akkor

13:       *új*  $\leftarrow$  LÉTREHOZ(*LáncElem<T>*) {*tart*  $\leftarrow$  érték, *köv*  $\leftarrow$  *p.köv*}

14:       *p.köv*  $\leftarrow$  *új*

15:     különben

16:       hiba "Hibás index"

17:     elágazás vége

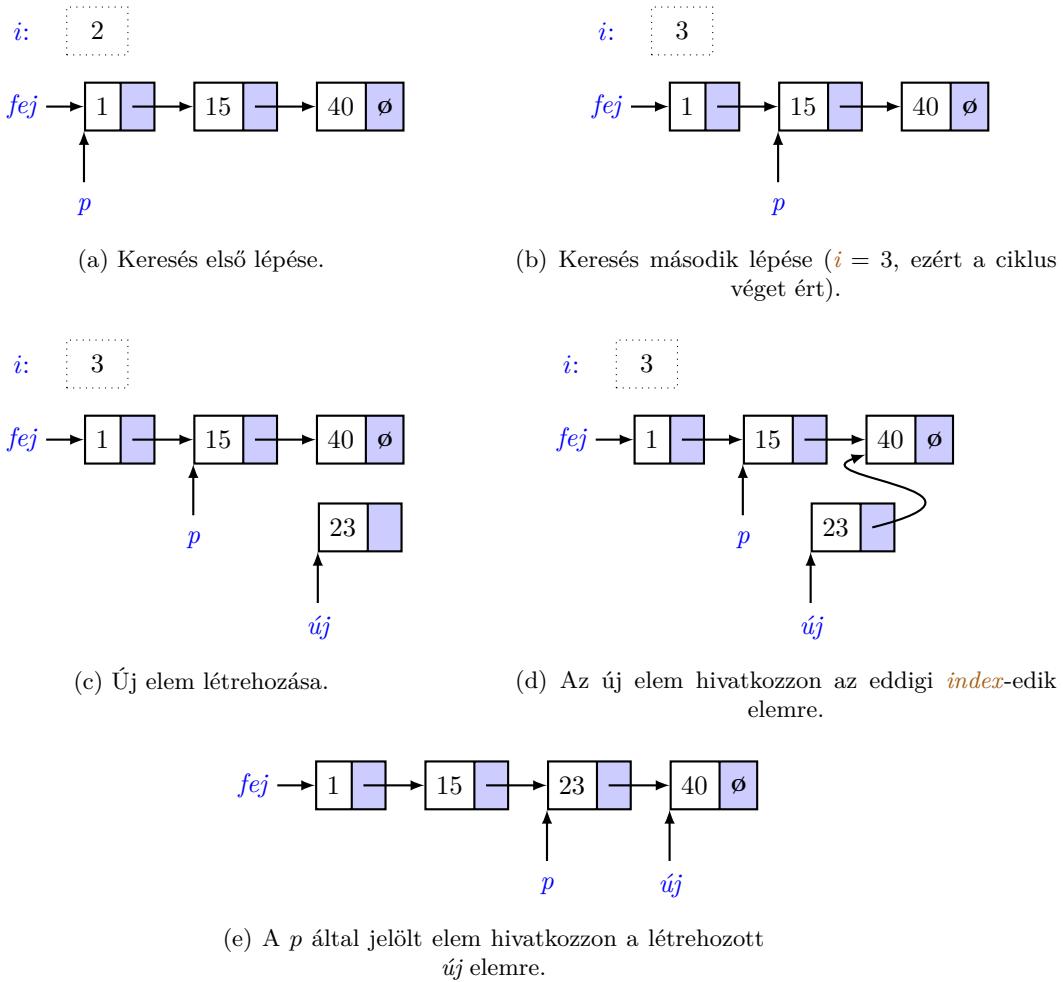
18:     elágazás vége

19: eljárás vége

---

#### Felhasznált változók és függvények

- *fej* : Hivatkozás a láncolt lista fejére (ami esetleg meg is változhat).
  - *új, p* : *LáncElem<T>* típusú segédváltozók.
  - LÉTREHOZ(*LáncElem<T>*) : Létrehoz egy új láncelementet, és visszaad egy hivatkozást erre.
-



5.6. ábra. Láncolt lista *index*. helyére beszúrás. A példában a 23 értéket szúrjuk be a 3. elem előre.

#### 5.2.4. Elemek bejárása és egyedi elérése

##### Láncolt lista bejárása

Minden általunk tárgyalt adatszerkezet esetében megvizsgáljuk annak bejárási lehetőségeit. Bejárás alatt értjük azt a műveletet, amelynek segítségével az adatszerkezet valamennyi elemét pontosan egyszer elérjük. Az elérés során természetesen tetszőleges feldolgozási műveletekre is szükség lehet, ez lehet akár az elemek listázása, akár az elemeken valamilyen egyéb művelet elvégzése (pl. elemek összegzése, legkisebb elem meghatározása, stb.). A pontos művelet számunkra lényegtelen, mivel maga a bejárás módszere minden esetben azonos lesz.

Egyszerű, egyirányú láncolt lista bejárását mutatja be az 5.5. algoritmus (ennek megértését pedig az 5.7. ábra segíti). A láncolt lista egyes elemeit nem tudjuk közvetlenül elérni, mindegyikhez csak az előző elemen keresztül rendelkezünk egy hivatkozással, kivéve persze az első elemet, amelyre egy külső hivatkozás mutat. A bejárás alapelve ennek megfelelően az, hogy elindulunk az első elemtől, majd egy ciklusba szervezve feldolgozzuk az aktuális elemet, és továbblápunk a következőre. A ciklust addig ismétljük, amíg fel nem dolgozzuk a teljes listát.

Az algoritmus egy *p* nevű segédváltozót használ az aktuális elem eltárolásához. Az algoritmus 2. sorában ennek a segédváltozónak értékül adjuk a *fej* változó értékét. Hivatkozásokról van szó, tehát itt nem a láncolt listában tárolt tartalmi adatok másolása történik meg, pusztán a *p* hivatkozás ugyanarra az elemre hivatkozik, ahol a *fej*. Amennyiben a feldolgozandó lista üres volt, akkor a *p* értéke a *fej*-hez hasonlóan  $\emptyset$  lesz, ez azonban nem okoz problémát, mivel ilyenkor a következő ciklus magja egyszer se fog lefutni, ennek megfelelően az algoritmus véget ér érdemi feldolgozás nélkül.

A ciklus feladata a fent említett lépések ismétlése (feldolgozás, következő elemre lépés) egészen addig, amíg nem jutunk a lánc végére (üres lista esetén tekinthetjük úgy, hogy már eleve ott állunk).

A ciklusmag első lépése maga a feldolgozás, ami jelen esetben egy *művelet* függvény meghívását jelenti (4. sor). A függvény egy darab, a láncolt lista tartalmának megfelelő *T* paramétert vár, és ezt a feladatnak megfelelően feldolgozza (a gyakorlatban persze elképzelhető, hogy az implementációban itt valójában nem egy függvényhívás, hanem közvetlenül a feldolgozás programkódja szerepel). Érdemes észrevenni, hogy a bejárás általános formájában nincs lehetőség az egyes elemek közötti kapcsolatok felhasználására (pl. előző, következő elemmel együtt feldolgozás).

A feldolgozást követően még a ciklusmagon belül átléptetjük a *p* hivatkozást a következő elemre (5. sor). Amíg ez egy valóban létező elemet jelent, addig a ciklus újra lefut, egészen addig, amíg el nem érünk a lánc végére (5.7c. ábra). Az utolsó elem feldolgozását követő léptetés hatására a *p* változó értéke  $\emptyset$  lesz, emiatt a ciklusfeltétel többé nem lesz igaz (3. sor), így a program kilép a ciklusból. Ez egyben a bejárás végét is jelenti.

Az 5.7. ábra egy egyszerű példát mutat egy háromelemű láncolt lista bejárására. A feldolgozás jelen esetben a tartalmak egyszerű listázását jelenti.

---

#### 5.5. Algoritmus Láncolt lista bejárása

---

**Bemenet:** *művelet - eljárás*<*T*> (az elemeken végrehajtandó művelet)

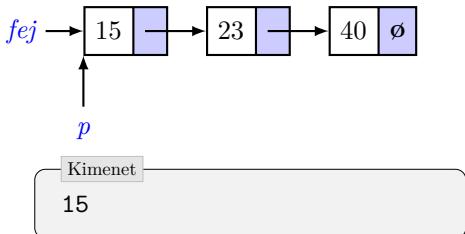
- 1: **eljárás** LÁNCOOLTLISTA.BEJÁR(*művelet*)
  - 2:   *p*  $\leftarrow$  *fej*
  - 3:   **ciklus** **amíg** *p*  $\neq$   $\emptyset$
  - 4:     *művelet*(*p.tart*)
  - 5:     *p*  $\leftarrow$  *p.köv*
  - 6:   **ciklus vége**
  - 7: **eljárás vége**
- 

##### Felhasznált változók és függvények

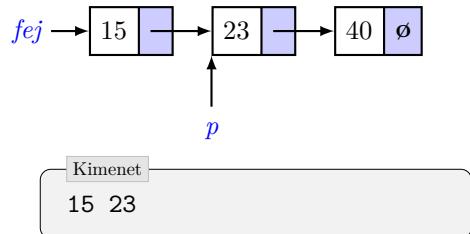
- *fej* : Hivatkozás a láncolt lista fejére.
  - *p* : Egy *LáncElem*<*T*> típusú segédváltozó.
- 

#### Elemszám tulajdonság

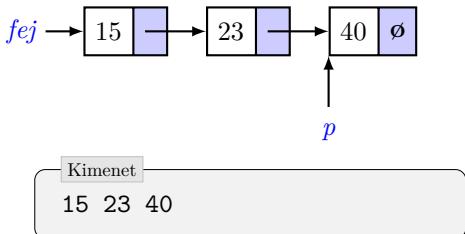
A *Lista* interfésnél meghatározott *Elemszám* tulajdonság megvalósítására két nyilvánvaló ötletünk is lehet:



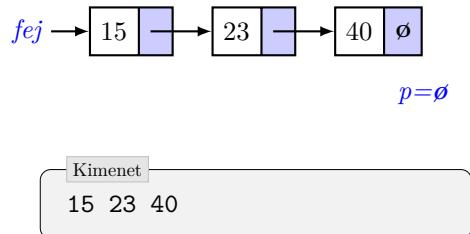
(a) Lista első elemének feldolgozása.



(b) Lista második elemének feldolgozása.



(c) Lista harmadik elemének feldolgozása.



(d) A lista végére ért a mutató.

5.7. ábra. Példa láncolt lista bejárására. A feldolgozás a láncelemek tartalmának kimenetre írását jelenti.

- A bejárásnál már látható módon végiglélhetünk az egész listán és megszámoljuk a lépések számát. Ez lesz a tulajdonság visszatérési értéke.
- Mivel ez nagyon erőforrásigényes, ezért felvehetünk egy külön mezőt az aktuális elemszám eltárolására. Ebben az esetben minden beszúrás és törlés esetén aktualizálni kell ennek az értékét.

Mivel minden két megoldás nagyon egyszerű, ezért ezeket pszeudokód szinten nem tárgyaljuk.

### Kiolvasás index alapján

A **KIOLVAS** művelet a tömb alapú megvalósítás esetében egyszerűen megoldható volt, hiszen a paraméterként átadott indexet közvetlenül lehetett használni az elemek tárolására használt tömbben is. Ebben az esetben ez nem működik, hiszen az elemek nem egymás mögött helyezkednek el a memoriában, ezért egy egyszerű memóriacím szorzással és osztással nem kapjuk meg valamelyik elem helyét. Ennek menetét mutatja be az 5.6. algoritmus.

Ehelyett a **fej** hivatkozás által mutatott első elemről kezdve lépésenként kell eljutnunk a kiolvasandó elemhez. Ez persze nem bonyolult, egy egyszerű ciklussal meg tudjuk valósítani, azonban az erőforrásigénye jelentős lehet egy nagyobb láncolt lista esetében.

Az algoritmus működése során egy **p** változó fog végiglélgetni a lista elemein, egy **i** változóban pedig azt számolja, hogy adott pillanatban hányadik láncolt lista elemen áll (kivéve, ha már túlléptük a lista utolsó elemén, ilyenkor a **p** értéke **∅**, az **i** pedig a lista méreténél egygyel nagyobb értéket tárol).

Általános esetben a **fej** mutató által hivatkozott első elemre lépünk, ennek megfelelően az **i** értékét is 1-re állítjuk. Ezt követően elindítunk egy ciklust, ami minden lépésben átlépteti a **p** hivatkozást a következő elemre, és ennek megfelelően növeli az **i** értékét is egygyel. Ezt addig ismétljük, amíg az **i** értéke eléri a paraméterként átadott **index** értéket. Amikor ez megtörtént, akkor a **p** változó éppen a keresett elemre mutat, ennek vissza tudjuk adni a tartalmát.

Vizsgálni érdemes az alábbi speciális eseteket is:

- Üres a lista: ilyenkor a **p** értéke megkapja a **fej**-ben lévő **∅** értéket. Emiatt egyszer se fog belépní a ciklusba, és az azt követő elágazás feltétele se lesz igaz. Tehát az **index** értékétől függően minden hibát adunk vissza.
- Első elem: a program ilyenkor se fog belépní a ciklusba, hiszen az **i** értéke már eleve 1-ről indul. Viszont a **p** helyesen az első elemre mutat, ezért ennek adja vissza a tartalmát.
- Túlcímezük a listát: ennek kiküszöbölése érdekében kell a ciklusban folyamatosan vizsgálni azt, hogy a **p** értéke még nem **∅**, tehát nem értünk a lista végére. Ha ez mégis bekövetkezne, akkor az

---

## 5.6. Algoritmus A láncolt lista index-edik helyén lévő elem kiolvasása

---

**Bemenet:** *index - egész<sup>+</sup>* (a kiolvasandó elem *indexe*)

```

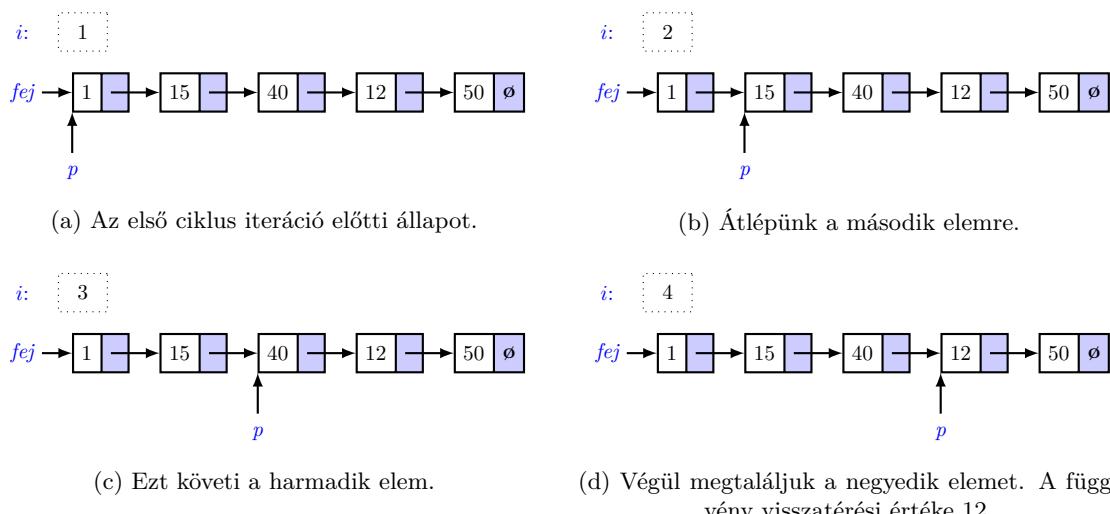
1: eljárás LÁNCOLTLISTA.KIOLVAS(index)
2: p \leftarrow fej
3: i \leftarrow 1
4: ciklus amíg (p $\neq \emptyset$) \wedge (i < index)
5: p \leftarrow p.köv
6: i \leftarrow i + 1
7: ciklus vége
8: ha p $\neq \emptyset$ akkor
9: vissza p.tart
10: különben
11: hiba "Hibás index"
12: elágazás vége
13: eljárás vége

```

---

### Felhasznált változók és függvények

- *fej* : Hivatkozás a láncolt lista fejére.
  - *p* : Egy *Láncelem<T>* típusú segédváltozó.
- 



5.8. ábra. Egy 5 elemű láncolt lista 4. elemének tartalmát kérdezzük le.

azt jelenti, hogy a paraméterként átadott *index* nagyobb, mint a lista mérete, tehát ismét hibát jelzünk.

Az általános esetre mutat példát az 5.8. ábra.

### Módosítás index alapján

A MÓDOSÍT művelet pont ugyanazokat a lépéseket követi, mint amit a kiolvasásnál láttunk, kivéve, hogy a végén nem visszaadja a tartalmat, hanem felülírja. Ennek megfelelően itt már csak a kész pszeudokódot ismertetjük (5.7. algoritmus).

---

## 5.7. Algoritmus A láncolt lista index-edik helyén lévő elem módosítása

---

Bemenet: *index - egész<sup>+</sup>* (a módosítandó elem indexe)

Bemenet: érték - *T* (az új érték)

```
1: eljárás LÁNCOLTLISTA.MÓDOSÍT(index, érték)
2: p ← fej
3: i ← 1
4: ciklus amíg ($p \neq \emptyset$) ∧ ($i < index$)
5: p ← p.köv
6: i ← i + 1
7: ciklus vége
8: ha $p \neq \emptyset$ akkor
9: p.tart ← érték
10: különben
11: hiba "Hibás index"
12: elágazás vége
13: eljárás vége
```

---

### Felhasznált változók és függvények

- *fej* : Hivatkozás a láncolt lista fejére.
  - *p* : Egy *LáncElem<T>* típusú segédváltozó.
- 

### 5.2.5. Törlés láncolt listából

A **TÖRÖL** művelet során nem csak a lánceleme(ke)n belül megtalálható tartalmat törljük (vagy jelöljük valahogy, hogy az törölt), hanem az egész láncelemet eltávolítjuk. Ezt nevezzük kiláncolásnak, hiszen önmagában a törölni kívánt elem megszüntetése még nem lenne elegendő, módosítani kell az előtte lévő elemet is (illetve a legelső elem esetében a *fej* hivatkozást), hogy megmaradjon a lista épisége.

A feladat megoldása két részből áll: meg kell keresni az első törlendő elemet (jelen esetben megadott tartalommal rendelkező elemet keresünk), majd pedig végre kell hajtani a kiláncolást (5.8. algoritmus). A kiláncolás során két alapvető esetet különböztethetünk meg:

- Középső vagy utolsó elem törlése: egyszerű egyirányú láncolt listák esetében minden elem csak a következőre hivatkozik, az előzőre nem. Tehát a kiláncolás során csak a törlendő elem előtti elem köv hivatkozását kell majd módosítani a törlendő elem utáni elemre. Ezt tekinthetjük a törlés általános esetének. Ezeket az eseteket mutatják be az 5.9. és az 5.10. ábrák.
- Első elem törlése: az első elem törlésekor ez természetesen nem valósítható meg, hiszen nincs „előző elem”, aminek a köv hivatkozását módosíthatnánk a kiláncolás során. Emiatt ilyenkor a *fej* értékét kell hasonló módon megváltoztatni, hogy az eddigi első (törrendő) elem helyett a másodikra hivatkozzon (5.11. ábra).

Az algoritmus keresés része hasonló a már megismert előre lépegetéshez. A *p* segédváltozó felveszi a *fej* értékét, majd egy ciklus segítségével addig lépeget a következő elemekre (7. sor), amíg meg nem találja a törlendő elemet.

Lényeges kiegészítés azonban, hogy egy *e* változó folyamatosan, egy elem késéssel követi a *p* értékét (6. sor), ezzel ez minden a *p* által mutatott előtti lánceleme hivatkozik. Ez amiatt szükséges, mert amikor a *p* megtalálja a törlendő elemet, akkor az előzőt kell majd módosítanunk. A láncolt listában azonban nem lehet visszafele lépni, tehát ezzel a technikával az *e* minden pont a *p* előzőjét fogja mutatni, így ez a kiláncolásnál hasznos lesz számunkra.

A ciklusból való kilépés során a *p* értékéből tudjuk meghatározni azt, hogy találtunk-e törlendő elemet (9. sor). Amennyiben ezen a ponton az értéke  $\emptyset$ , az azt jelenti, hogy nem találtunk törlendő elemet (a lista végén áll), ha pedig egy valódi elemre hivatkozik, akkor az éppen egy törlendő elem.

Amennyiben ez utóbbi feltétel teljesül, akkor a *q* hivatkozást ráállítjuk a *p* utáni elemre. Az *e* értékéből tudunk következtetni arra, hogy az első, vagy egy későbbi elemet kell törölnünk. Amennyiben az *e* értéke még mindig  $\emptyset$ , az azt jelenti, hogy az előző ciklus egy lépést sem tett, tehát a *p*-vel az első elemen állunk. Ebben az esetben a *fej* értékét kell módosítanunk, hogy az a jelenlegi második elemre hivatkozzon (12. sor). Ezt az esetet mutatja be az 5.11. ábra.

Amennyiben az  $e$  értéke sem  $\emptyset$ , akkor a változó éppen a törlendő előtti elemre hivatkozik. Ebben az esetben a kiláncolást úgy tudjuk megvalósítani, hogy ennek az előző elemnek a  $köv$  hivatkozását átállítjuk a törlendő utáni elemre (14. sor). Ezzel a törlendő elemet kiláncoltuk a listából. Érdemes megfigyelni, hogy egy közbülső, illetve az utolsó elem esetében is ugyanezeket a lépéseket hajtjuk végre, és minden esetben helyes eredményhez jutunk (5.9d. és 5.10d. ábrák).

Bármelyik eset is következik be, törlés esetében a  $p$  éppen a törlendő elemre mutat, így a kiláncolást követően ezt az elemet felszabadíthatjuk (16. sor). A  $p$ -t át kell léptetnünk a törölt elem következőjére. A törölt elemet már nem érjük el, hiszen felszabadítottuk, viszont a  $q$  mutató pont ide mutat, ezért ide irányítjuk a  $p$  mutatót.

Látható, hogy az algoritmus tartalmaz egy külső ciklust is. Ez felelős azért, hogy a lista minden törlendő elemét kiláncoljuk. Ha a 9. sorban található feltétel igaz, az azt jelenti, hogy elvégzünk egy törlést. A törlést követően a  $p$  változó a törölt elemet követőre mutat, az  $e$  pedig az előzőre (eddig köztük volt a törlendő, de azt már kiláncoltuk). Emiatt látható, hogy a törlést folytathatjuk ebből a pontból, hiszen minden esetben szükséges változó épp ideális állapotban van.

A 19. sorban található hátralétesztelős ciklus ellenőri a kilépés feltételét. Amennyiben a  $p$  változó értéke  $\emptyset$ , az két dolgot jelenthet:

- nem találtunk törlendő elemet a listában, ezért nem is töröltünk semmit,
- az utolsó elemet töröltünk, tehát már átnéztük a teljes listát.

Minden esetben leállhat az algoritmus, hiszen biztosan nincs további törlendő elem a listában.

---

## 5.8. Algoritmus Láncoolt lista megadott tartalmú elemeinek a törlése

---

**Bemenet:** érték -  $T$  (a törlendő érték)

```

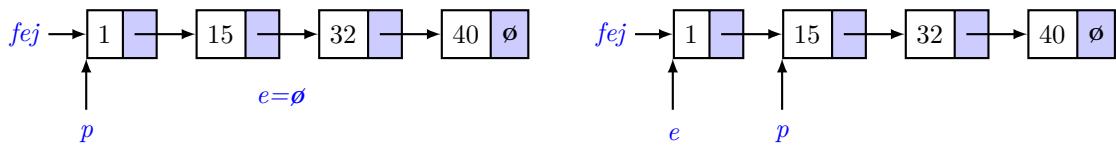
1: eljárás LÁNCOOLISTA.TÖRÖL(érték)
2: p ← fej
3: e ← ∅
4: ciklus
5: ciklus amíg ($p \neq \emptyset$) ∧ ($p.tart \neq$ érték)
6: e ← p
7: p ← p.köv
8: ciklus vége
9: ha $p = \emptyset$ akkor
10: q ← p.köv
11: ha $e = \emptyset$ akkor
12: fej ← q
13: különben
14: e.köv ← q
15: elágazás vége
16: FELSZABADÍT(p)
17: p ← q
18: elágazás vége
19: amíg $p \neq \emptyset$
20: eljárás vége

```

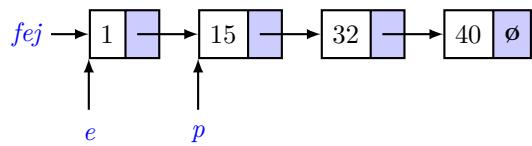
---

### Felhasznált változók és függvények

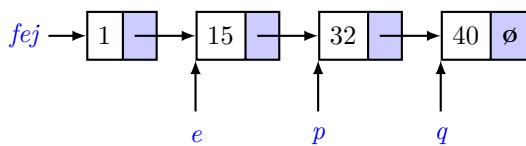
- $fej$  : Hivatkozás a láncoolt lista fejére (ami esetleg még is változhat).
  - $p, e, q$  :  $\tilde{Láncoolt\ lista}$  típusú segédváltozók.
  - $FELSZABADÍT(p)$  : A  $p$  által mutatott elem felszabadítása.
-



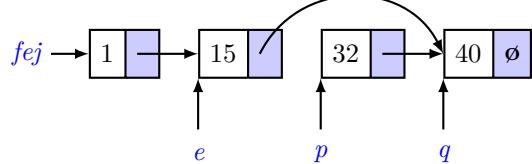
(a) Keresés első lépése. A  $p$  még nem a törlendő elemre mutat.



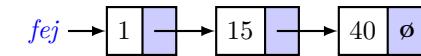
(b) Keresés következő lépése. A  $p$  még nem a törlendő elemre mutat.



(c) A következő lépés után a  $p$  a törlendő elemre mutat, az  $e$  pedig az azt megelőzőre. A  $q$  változót ráirányítjuk a törlendő utánira.

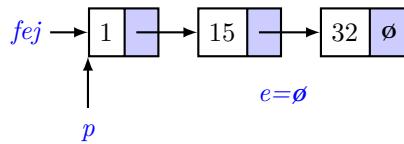


(d) Az  $e.köv$  hivatkozás módosítása a törlendő utáni elemre (ezzel a törlendő kiláncolása).

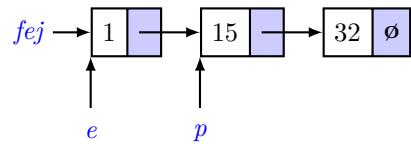


(e) A  $p$  által mutatott elem megszüntetése.

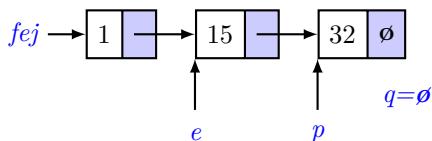
5.9. ábra. Láncolt lista középső elemének törlése.



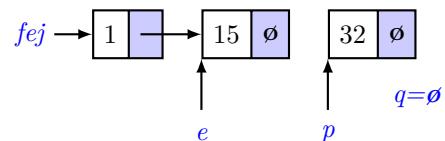
(a) Keresés első lépése. A  $p$  még nem a törlendő elemre mutat.



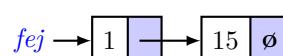
(b) Keresés következő lépése. A  $p$  még nem a törlendő elemre mutat.



(c) A következő lépés után a  $p$  a törlendő elemre mutat, az  $e$  pedig az azt megelőzőre. Mivel az utolsó elemen állunk, ezért a  $q$  értéke  $\emptyset$  lesz.

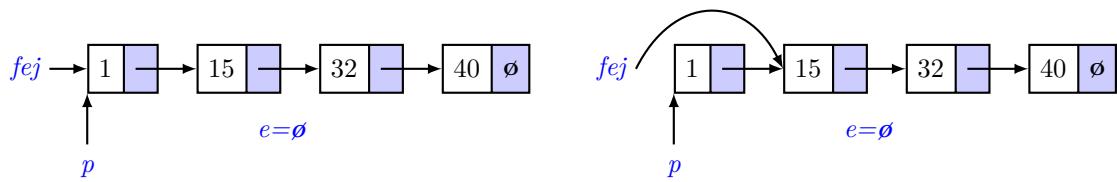


(d) Az  $e.köv$  hivatkozás értékül kapja a  $q$  hivatkozás értékét, ami jelen esetben a lezártó  $\emptyset$ .

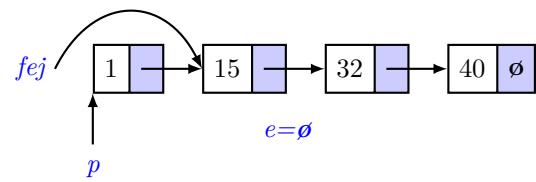


(e) A  $p$  által mutatott elem megszüntetése.

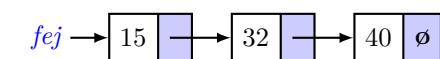
5.10. ábra. Láncolt lista utolsó elemének törlése.



(a) Keresés első lépése. A  $p$  azonnal a törlendő elemre hivatkozik.



(b) A  $fej$  mutató átléptetése a  $p$  utáni elemre.



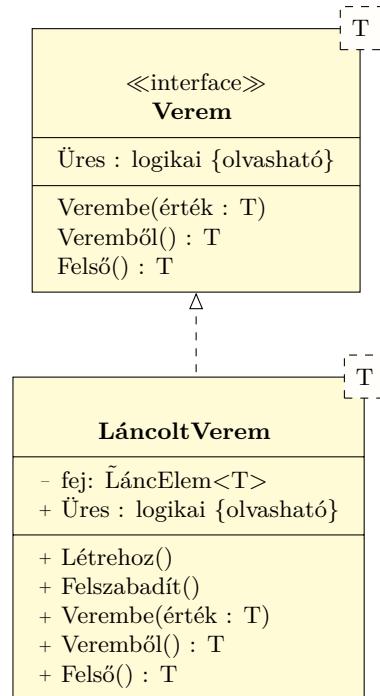
(c) A  $p$  által mutatott elem megszüntetése.

5.11. ábra. Láncolt lista első elemének törlése.

## 5.3. Verem megvalósítása láncolt szerkezzel

### 5.3.1. Alapvető felépítés

A **Verem** típust nagyon egyszerűen lehet láncolt listával megvalósítani. Ebben az esetben jól ki tudjuk használni az implementációs mód előnyeit (nem kell előre megadni a méretét, minden dinamikusan annyi helyet foglal, amennyi elem van benne), és nem okoznak problémát annak hátrányai (pl. indexelés művelet erőforrásigényes a tömbhöz képest).



### 5.3.2. Létrehozás és megszüntetés

A **LÉTREHOZ** művelet csak annyit csinál, hogy inicializálja a **fej** mező értékét (5.9. algoritmus). Elemeket itt még nem szükséges létrehoznunk.

---

#### 5.9. Algoritmus Láncolt verem létrehozása

---

- 1: eljárás LÁNCOLOTVEREM.LÉTREHOZ( )
  - 2:     `fej`  $\leftarrow \emptyset$
  - 3: eljárás vége
- 

#### Felhasznált változók és függvények

- **fej** : Hivatkozás a láncolt lista fejére.
- 

A verem objektum felszabadítása során fel kell szabadítani a lánc minden elemét. Ezt a listához hasonló módon tudjuk megtenni (5.10. algoritmus).

### 5.3.3. Műveletek megvalósítása

#### Új érték felvétele

A **VEREMBE** egy új érték felvételét végzi, ezt mutatja az 5.11. algoritmus. Mivel a lánc legelső elemét tudjuk elérni a legegyszerűbben, ezért azt a módszert válasszuk, hogy minden ide szúrjuk be az egymást követő elemeket. Ez a lánc elejére beszúrás művelet.

Érdekesség, hogy tömbök esetében az egyik leglassabban kivitelezhető művelet a tömb legelső eleme elé beszúrás, hiszen ilyenkor a tömb minden már meglévő elemét eggyel hátrébb kell tolni (feltéve, ha

---

## 5.10. Algoritmus Láncolt verem felszabadítása

---

```
1: eljárás LÁNCOLTVEREM.FELSZABADÍT()
2: ciklus amíg (fej ≠ \emptyset)
3: q ← fej
4: fej ← fej.köv
5: FELSZABADÍT(q)
6: ciklus vége
7: eljárás vége
```

---

### Felhasznált változók és függvények

- *fej* : Hivatkozás a láncolt lista fejére.
  - *q* : Egy *LáncElem<T>* típusú segédváltozó.
  - *FELSZABADÍT(p)* : A *p* által mutatott elem felszabadítása.
- 

meg szeretnénk tartani az elemek sorrendjét). Az egyirányú, egyszerű láncolt listák esetében azonban éppen ez a beszúrás valósítható meg a legegyszerűbben.

Első lépésként létrehozunk egy új, üres láncelemet, ahol a tartalom mezőnek azonnal értékül is adhatjuk a beszúrni kívánt értéket. A jelenlegi első elem a beszúrás után a második elem lesz közvetlenül a most beszúrandó elem mögött, ezért az erre hivatkozó *fej* értéket helyezzük az *új* elem *köv* mezőjébe. Ezzel a beszúrandó új elem és az eddigi első elem már megfelelő sorrendbe kerülnek.

---

## 5.11. Algoritmus Láncolt verembe új elem felvétele

---

Bemenet: érték - T (a beszúrandó új érték)

```
1: eljárás LÁNCOLTVEREM.VEREMBE(érték)
2: új ← LÉTREHOZ(LáncElem<T>) {tart ← érték, köv ← fej}
3: fej ← új
4: eljárás vége
```

---

### Felhasznált változók és függvények

- *fej* : Láncolt lista feje (ami esetleg meg is változhat).
  - *új* : Egy *LáncElem<T>* típusú segédváltozó.
  - *LÉTREHOZ(LáncElem<T>)* : Létrehoz egy új láncelemet, és visszaad egy hivatkozást erre.
- 

A lánc korábbi állapotától függően két eset lehetséges:

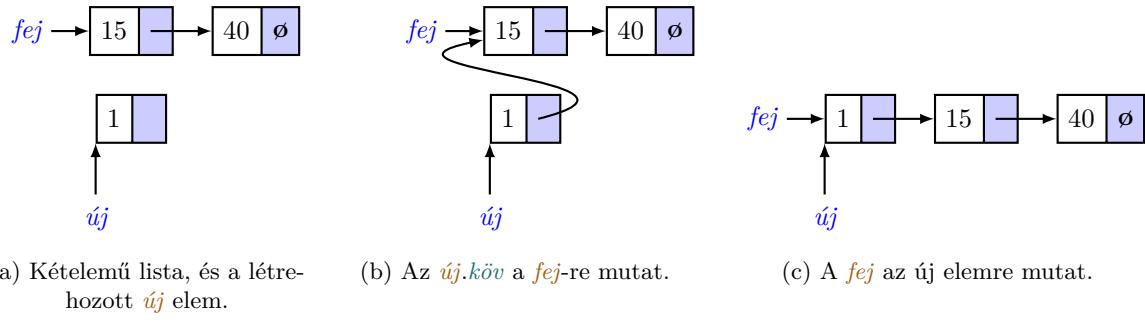
- Első elem elé beszúrás: ebben az esetben (5.12. ábra) már az elem beszúrása előtt is voltak elemek a láncolt listában. Ez tekinthető az általános esetnek, ilyenkor az új elem a fent leírt módon tárolódik el.
- Első elem felvétele: ebben az esetben a beszúrás meghívása előtt még nem volt elem a listában. Az újonnan felvett elemet tehát nem egy már meglévő elé kell felvenni, hanem ez az új elem lesz a lánc első (és egyben összes) eleme. Az algoritmus ebben az esetben is jól működik, jól látható, hogy az így felvett *új* elem *köv* mezőjébe ilyenkor a *fej* aktuális tartalma kerül, aminek (üres lánc lévén) az értéke  $\emptyset$ . Ez pedig helyes végeredmény, hiszen az első elem egyben az utolsó is, tehát a lezáró elemnek kell ide kerülnie.

### Veremből elem kivétele

A VEREMBŐL művelet az előző fordítottja (5.12. algoritmus). Célja, hogy kivegye a verem legutoljára berakott elemét. Ez egyszerű, mivel minden a legelső elem lesz az, ha egyáltalán van ilyen.

Ehhez elsőként ellenőrzi, hogy van-e egyáltalán elem a listában (tehát a *fej* ≠  $\emptyset$ ), és hibát jelziünk, ha nincs. Ha van, akkor az *érték* segédváltozóban eltárolja az első elem tartalmát. Ezt követően pedig kiláncoljuk az első elemet.

A kiláncolás előtt eltárolja a *fej* aktuális értékét a *q* változóba. Ez csak azért szükséges, mert a *fej* hátrabb léptetése után már nem lenne referencia az első elemre, amit ki akarunk törölni. Ezt a kiláncolás után a *q* változón keresztül tudjuk megtenni.



5.12. ábra. Láncolt lista elejére beszúrás. A példában az "1" értéket szúrjuk be a listába.

Az 5. sorban egyébként két eset lehetséges:

- Amennyiben a láncolt listának egynél több eleme van, akkor a *fej* mutató értéke az eddigi másodikra fog mutatni, tehát kiláncoltuk az első elemet.
- Amennyiben a láncolt listában már csak egy elem volt, akkor ennek az elemnek a *köv* mutatójának értéke  $\emptyset$ . Ez az érték fog bekerülni a *fej* mutatóba. Ez nem hiba, hanem az üres listát jelzi, ami jelen esetben helyes is, hiszen ezt követően tényleg nincs elem a veremben.

---

### 5.12. Algoritmus Láncolt veremből elem kivétele

---

**Kimenet:** *érték - T* (a verembe utoljára felvett érték)

```

1: függvény LÁCOLTVEREM.VEREMBŐL()
2: ha fej $\neq \emptyset$ akkor
3: érték \leftarrow fej.tart
4: q \leftarrow fej
5: fej \leftarrow fej.köv
6: FELSZABADÍT(q)
7: vissza érték
8: különben
9: hiba "Nincs elem"
10: elágazás vége
11: függvény vége

```

---

#### Felhasznált változók és függvények

- *fej* : Hivatkozás a láncolt lista fejére.
  - *érték* : Egy *T* típusú segédváltozó.
  - *q* : Egy *LáncElem<T>* típusú segédváltozó.
  - *FELSZABADÍT(p)* : A *p* által mutatott elem felszabadítása.
- 

#### A verem felső elemének lekérdezése

A *FELSŐ* művelet megvalósítása alapvetően azonos a veremből való kivétellel, csak annyi a különbség, hogy ez nem változtatja meg a *fej* mező értékét, ezért nem veszi ki az utolsó elemet a veremből (5.13. algoritmus).

---

### 5.13. Algoritmus Láncolt veremben felső elem lekérdezése

---

**Kimenet:** érték - T (a verembe utoljára felvett érték)

- 1: függvény LÁNCOLTVEREM.FELSŐ( )
  - 2:   ha *fej*  $\neq \emptyset$  akkor
  - 3:     vissza *fej.tart*
  - 4:   különben
  - 5:     hiba "Nincs elem"
  - 6:   elágazás vége
  - 7: függvény vége
- 

**Felhasznált változók és függvények**

- *fej* : Hivatkozás a láncolt lista fejére.
-

## 5.4. Sor megvalósítása láncolt szerkezettel

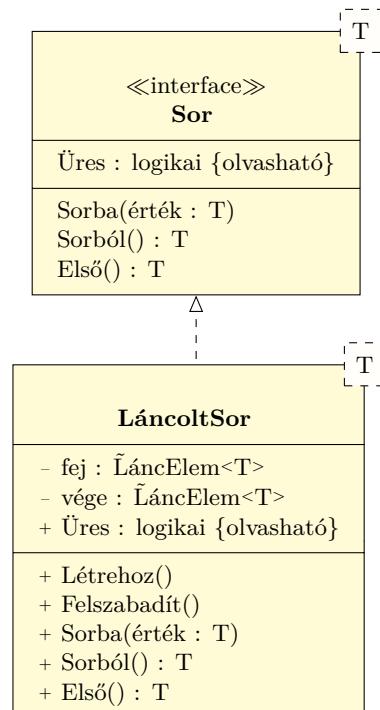
### 5.4.1. Alapvető felépítés

A **Sor** típus megvalósítása jelentősen hasonlít a veremnél látottakhoz. Lényeges különbség azonban az, hogy itt a láncolt szerkezet két különböző végén kell elvégezni a felvétel, illetve kivétel műveleteket. Mi most úgy valósítjuk ezt meg, hogy minden a lánc végére füzzük fel az új elemeket a beszúráskor, és minden a lánc első elemét fogjuk kiolvasni/törölni a sorból való kiolvasáskor.

A fenti működésből adódik, hogy gyakran lesz szükségünk a lánc végére való beszúráusra. Ez az eddig látott technikák alapján meglehetősen sok lépést igényel, hiszen minden a lánc elejétől egy ciklussal kell végigjárni az egész láncot az utolsó elem eléréséhez. Ezt kiküszöbölni lehet egy új, **vége** változót, ami minden a lánc utolsó elemére fog hivatkozni (üres listánál pedig az értéke  $\emptyset$  lesz).

#### Megjegyzés

Ezzel a kiegészítéssel azt is szeretnénk bemutatni, hogy a leírt algoritmusokat ne tekintsük véglegesnek és megváltoztathatatlanak. A megoldandó feladattól függően bátran lehet őket módosítani/kiegészíteni.



### 5.4.2. Létrehozás és megszüntetés

A **LÉTREHOZÁS** és a **FELSZABADÍTÁS** műveletek teljesen megegyeznek a veremnél látottakkal.

#### 5.14. Algoritmus Láncolt sor létrehozása

- 1: **eljárás** LÁNCOLTSOR.LÉTREHOZ( )
- 2:    $fej \leftarrow \emptyset$
- 3:    $vége \leftarrow \emptyset$
- 4: **eljárás vége**

#### Felhasznált változók és függvények

- **fej** : Hivatkozás a láncolt sor fejére.
- **vége** : Hivatkozás a láncolt sor végére.

---

### 5.15. Algoritmus Láncolt sor felszabadítása

---

```
1: eljárás LÁNCOLTSOR.FELSZABADÍT()
2: ciklus amíg (fej ≠ \emptyset)
3: q \leftarrow fej
4: fej \leftarrow fej.köv
5: FELSZABADÍT(q)
6: ciklus vége
7: vége \leftarrow \emptyset
8: eljárás vége
```

---

#### Felhasznált változók és függvények

- *fej* : Hivatkozás a láncolt sor fejére.
  - *vége* : Hivatkozás a láncolt sor végére.
  - *q* :  $\tilde{\text{LáncElem}}< T >$  típusú segédváltozó.
  - **FELSZABADÍT(*p*)** : A *p* által mutatott elem felszabadítása.
- 

### 5.4.3. Műveletek megvalósítása

#### Sorba új érték felvétele

A **SORBA** művelet megvalósítását mutatja az 5.16. algoritmus. Mivel van egy mutatónk a lánc utolsó elemére, ezért maga a beszúrás egyszerűen elvégezhető az alábbi szabályok szerint:

- *vége*  $\neq \emptyset$  esetében a listában már vannak elemek, és tudjuk, hogy ezek közül az utolsóra mutat a *vége*. Nincs más dolgunk, mint ezután fűzni az új elemet, és átirányítani az új elemre a mutatót.
- *vége* =  $\emptyset$  esetében a listában nincs egyetlen elem se. Ilyenkor az új elem a lánc elejére kerül, emiatt a *fej* változót is erre kell irányítani, és persze erre kell irányítani a *vége* mezőt is.

---

### 5.16. Algoritmus Láncolt sorba új elem felvétele

---

#### Bemenet: érték - **T** (a beszúrandó új érték)

```
1: eljárás LÁNCOLTSOR.SORBA(értek)
2: új \leftarrow LÉTREHOZ($\tilde{\text{LáncElem}}< T >$) {tart \leftarrow érték, köv \leftarrow \emptyset }
3: ha vége $\neq \emptyset$ akkor
4: vége.köv \leftarrow új
5: különben
6: fej \leftarrow új
7: elágazás vége
8: vége \leftarrow új
9: eljárás vége
```

---

#### Felhasznált változók és függvények

- *fej* : A láncolt sor feje (ami esetleg meg is változhat).
  - *vége* : A láncolt sor utolsó eleme (ami biztosan változni fog).
  - *új* : Egy  $\tilde{\text{LáncElem}}< T >$  típusú segédváltozó.
  - **LÉTREHOZ( $\tilde{\text{LáncElem}}< T >$ )** : Létrehoz egy új láncelemet, és visszaad egy hivatkozást erre.
- 

#### Sorból elem kivétele

A **SORBÓL** művelet (5.17. algoritmus) pontosan ugyanazokat a lépéseket hajtja végre, mint a veremnél látható **VEREMBŐL** művelet (5.12. algoritmus). Ez nem meglepő, hiszen itt is a lánc első elemét szeretnénk törölni, illetve ennek értékét visszaadni.

Érdekes lehet a 7. sor szerepe. Ez azért kell, mert ha kivesszük az utolsó elemet a sorból, akkor a *fej* mutató értéke ugyan  $\emptyset$  lesz, a *vége* mutató viszont még mindig az utolsó elemre hivatkozna. A kiolvasás ettől még működne, de a következő sorba való beszúráskor már hibát okozna, hiszen ott ezt a *vége* mutatót ellenőrizzük.



---

### 5.19. Algoritmus Láncolt sor ürességének ellenőrzése

---

**Kimenet:** . - logikai (értéke *igaz* ha van elem a sorban, egyébként *hamis*)

- 1: tulajdonság LÁNCOLTSOR.ÜRES
  - 2: vissza *fej*  $\neq \emptyset$
  - 3: tulajdonság vége
- 

**Felhasznált változók és függvények**

- *fej* : A láncolt sor feje.
-

## 5.5. Prioritásos sor megvalósítása láncolt szerkezettel

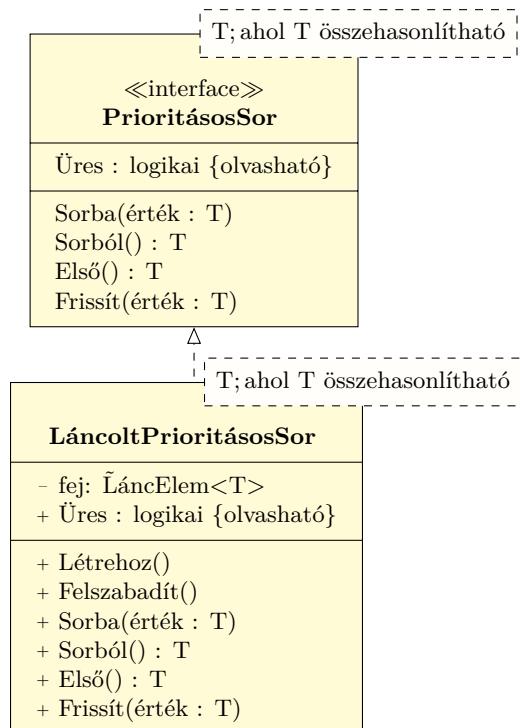
### 5.5.1. Alapvető felépítés

Prioritásos sor esetében a kivételkor visszaadott értékek függetlenek a beszúrás sorrendjétől. minden elemhez rendelünk egy prioritást, és minden valamelyik maximális prioritású elemet kapjuk vissza a SORBÓL művelet meghívásakor.

A prioritás lehetne egy új mező is a **LáncElem** összetett típuson belül, de az egyszerűség kedvéért ezt úgy fogjuk kezelní, hogy a T típusra adunk egy összehasonlíthatósági megszorítást, tehát bármely két **T** típusú elem között értelmezhető egy „kisebb” reláció. A prioritást pedig úgy értelmezzük majd, hogy bármely  $x$  és  $y$  (ahol  $x$  és  $y$  **T** típusú értékek) esetén az  $x$  nagyobb prioritású, mint  $y$ , ha  $x > y$ .

Az adatszerkezetet úgy valósítjuk meg, hogy már a beszúráskor a **T** típuson alapuló „kisebb” reláció alapján rendezetten fogja eltárolni az elemet. Így a kivétel nagyon gyors lesz, hiszen minden prioritású elem a lánc elején lesz a legnagyobb prioritású elem.

Mivel a prioritásos sor elsőként minden prioritású elemet adja vissza, ezért a listánk a T típusú értékeket tartalmazó **tart** mező alapján „csökkenő sorrendben” lesz rendezett, tehát a nagyobb prioritású elemek a lánc elején helyezkednek majd el.



### 5.5.2. Létrehozás, megszüntetés és kiolvasások

A LÉTREHOZ, FELSZABADÍT, SORBÓL, ELSŐ, ÜRES műveletek és tulajdonság teljesen megegyeznek az általános sornál látottakkal. Ezért ezeket külön nem is tárgyaljuk.

### 5.5.3. Rendezett beszúrás megvalósítása

#### Egyszerű, de hosszabb változat

Tömbök esetében többnyire azzal foglalkoztunk, hogy hogyan lehet egy már feltöltött, de rendezetlen tömböt helyben rendezni. Láncolt listák esetében is járható ez az út, de itt célszerű lehet kihasználni azt a tényt, hogy nagyon könnyen tudunk két meglévő elem közé egy újat felvenni, emiatt célszerű lehet már a lánc felépítésekor az új elemeket a rendezésnek megfelelő helyre beszúrni.

A rendezett láncolt szerkezet felépítésére két algoritmust is megvizsgálunk. Mindkét módszer kimenete teljesen azonos, vizsgálatuk oka pusztán didaktikai, az első egyszerűbben áttekinthető, a második viszont rövidebb és elegánsabb megoldás.

Az első lehetőséget mutatja az 5.20. algoritmus. Ebben négy, egymástól jól elkülöníthető programág felel a felmerülő lehetőségek kezeléséért:

- Üres listába való beszúrás: itt a beszúrandó elem lesz a lánc első és egyben egyetlen eleme.
- Első elem elé beszúrás: ebben az esetben a beszúrandó elem prioritása nagyobb, mint a listában lévő bármelyik másik prioritás, emiatt az új elem lesz a lánc első eleme.
- Két elem közé beszúrás: ezt tekinthetjük a rendezett listába beszúrás általános esetének. Meg kell keresnünk azt az elempárt, ahol az egyik prioritása nagyobb (vagy egyenlő), a másiké viszont már kisebb (vagy egyenlő), mint a beszúrandó prioritás. Ilyenkor ez a két elem közé kell elhelyezni az új elemet.
- Utolsó elem mögé beszúrás: amennyiben az újonnan beszúrandó elem prioritása kisebb a lánc minden már meglévő prioritásánál, akkor az új tartalmat tároló elemet az eddigi utolsó elem után kell láncolni.

Az algoritmus felépítése is a fenti eseteket követi.

Az első ellenőrzés (2. sor) megvizsgálja, hogy üres-e a szerkezet. Amennyiben igen, akkor a legegyszerűbb módon felveszi ezt az elemet elsőként. A *fej* mutató az új elemre hivatkozik, annak *köv* mezőjének értéke pedig  $\emptyset$  lesz, hiszen ő egyben az utolsó elem is.

A következő vizsgálatra (6. sor) már csak akkor kerül sor, ha nem üres a lánc. Ebben az esetben meg lehet vizsgálni a szerkezet első elemének prioritását, hogy az kisebb-e, mint a beszúrandó. Ha igen, akkor az új elemnek még a lánc első eleme előtt kell bekerülnie, erre jól használható a már megismert elejére beszúrás művelet.

Minden egyéb esetben egy ciklust kell indítanunk, amelyik megkeresi azt a két elemet, amelyek közé be kell láncolni az újat (12. sor). Ehhez elég megtalálnunk az első, nála kisebb prioritással rendelkezőt. Az ez előtti elem lesz az öt követlenül megelőző. Kivételes esetnek tekinthető, ha a beszúrandó elemnél nincs kisebb a listában, emiatt a ciklus egy kiegészítő feltételt is tartalmaz, hogy csak addig fussen, amíg a *p* nem jutott a lánc végére. A beszúráshoz szükségünk lesz az előző elemre, és mivel a láncolt listában nem tudunk visszafelé lépni, ezért a törlésnél már megismert módon, egy *e* nevű segédváltozóval a cikluson belül mindenkorán a *p* előző értékét.

A két kilépési feltétel közül valamelyik biztosan teljesülni fog, akkor kilépünk a ciklusból. A 16. sor szerepe, hogy megvizsgálja, melyik feltétel miatt léptünk ki. Amennyiben a *p* értéke ezen a ponton  $\emptyset$ , az azt jelenti, hogy végignéztük a listát, és nincs az újnál kisebb prioritású elem. Amennyiben egy valódi elemre mutat, akkor az nem más, mint a listában a beszúrandónál kisebb prioritást tartalmazó elemek közül az első.

Az előbbi esetben a lánc végére beszúrás algoritmussal felvesszük az új elemet a szerkezet végére (17–18. sorok).

A másik esetben pedig felvesszük az új elemet az *e* és a *p* által hivatkozott láncelemek közé (20–21. sorok).

---

## 5.20. Algoritmus Beszúrás láncolással megvalósított prioritási sorba

---

Bemenet: érték - T (a beszúrandó érték)

Megszorítás: T összehasonlítható

```
1: eljárás LÁNCOLTPRIORITÁSOSSOR.SORBA(érték)
2: ha fej = ø akkor ▷ Üres lista
3: új ← LÉTREHOZ(LáncElem<T>) {tart ← érték, köv ← ø}
4: fej ← új
5: különben
6: ha fej.tart < érték akkor ▷ Első elem elő
7: új ← LÉTREHOZ(LáncElem<T>) {tart ← érték, köv ← fej}
8: fej ← új
9: különben
10: p ← fej
11: e ← ø
12: ciklus amíg ($p \neq \emptyset$) \wedge ($p.tart > \text{érték}$)
13: e ← p
14: p ← p.köv
15: ciklus vége
16: ha p = ø akkor ▷ Utolsó elem mögé
17: új ← LÉTREHOZ(LáncElem<T>) {tart ← érték, köv ← ø}
18: e.köv ← új
19: különben
20: új ← LÉTREHOZ(LáncElem<T>) {tart ← érték, köv ← p} ▷ Két elem közé
21: e.köv ← új
22: elágazás vége
23: elágazás vége
24: elágazás vége
25: eljárás vége
```

---

### Felhasznált változók és függvények

- *fej* : A prioritás szerint rendezett láncolt lista feje (ami esetleg meg is változhat).
  - *új*, *p*, *e* : *LáncElem<T>* típusú segédváltozók.
  - *LÉTREHOZ(LáncElem<T>)* : Létrehoz egy új láncelemet, és visszaad egy hivatkozást erre.
-

## Összevont, rövidebb változat

Az előző fejezetben megadott algoritmus tökéletesen működik, helyesen kezeli az összes felmerülő lehetőséget. Azonban meglehetősen hosszú, és érdemes észrevenni, hogy bizonyos programágak gyakorlatilag ugyanazt a kódot tartalmazzák. Ezek alapján az alábbi egyszerűsítési lehetőségekkel élhetünk (a sorszámok az 5.20. algoritmusra vonatkoznak):

- Az nyilvánvaló, hogy a 4. és a 8. sorok tulajdonképpen ugyanazt hajtják végre (mivel a kettő között nem változik sem a *fej*, sem pedig az *új* értéke). Azonban megfigyelhető, hogy a felettük lévő 3. és 7. soroknak is minden ugyanaz lesz az eredményük (a *fej* =  $\emptyset$  feltétel teljesülése miatt a 3. sorban található  $\emptyset$  értéket helyettesíthetjük a *fej*-jel, így már formailag is ugyanaz a két sor).
- Hasonló módon látható, hogy a 18. és a 21. sorok is teljesen azonosak. Itt szintén megfigyelhető, hogy a felettük lévő 17. és 20. soroknak is minden ugyanaz lesz az eredményük (a *p* =  $\emptyset$  feltétel teljesülése miatt a 17. sorban található  $\emptyset$  értéket helyettesíthetjük a *p* változóval, így már formailag is ugyanaz a két sor).

Ezek a sorok tehát összevonhatók, így az előzetesen felvázolt négy esetet le tudtuk egyszerűsíteni két felmerülő esetre. A beszúrás algoritmusát átgondolva egyébként is belátható, hogy tulajdonképpen tényleg csak kétféle esetet kell megkülönböztetnünk:

- A beszúrandó elem közvetlenül a *fej* mögé kerül (tehát üres listába, vagy az első elem előtt akarunk beszúrni).
- A beszúrandó elem egy már meglévő elem mögé kerül (tehát két elem közé, vagy az utolsó elem mögé akarunk beszúrni).

Hasonló két ággal már találkozhattunk a törlés algoritmusánál is. Mivel ott is szükség volt egy előző elemre, a ciklus egy *p* és egy *e* változó értékeit módosította, majd a kilépést követően az *e* értéke alapján tudtuk eldönteni, hogy történt-e lépés, vagy sem (tehát az első elemet kell törlni, vagy egy későbbit). Ugyanezt a technikát itt is felhasználhatjuk, a ciklusból való kilépés után az *e* változó értéke megadja, hogy a *fej* változót kell módosítani, vagy pedig egy előző elem *köv* hivatkozását.

A fenti összevonásokkal nyert az 5.21. algoritmus első lépésként inicializálja a *p* és *e* változókat, majd lefut egy, a törléshez hasonló ciklus (4. sor), amely segítségével megkeressük az első, a beszúrandónál kisebb prioritással bíró elemet (közben persze ügyelve arra is, hogy a láncolt szerkezet végére jutva is kilépjünk a ciklusból).

A 8. sor szerepe, hogy meghatározza a szükséges beszúrás módját. Ha az *e* változó értéke  $\emptyset$ , akkor a ciklusmag egyszer sem futott le, tehát vagy üres a lánc (5.13. ábra), vagy már az első elem is kisebb volt a beszúrandónál (5.14. ábra). Mindkét esetben közvetlenül a *fej* után kell felvenni az új elemet, ezt végzik el a 9–10. sorok.

Ha az *e* egy valódi elemre mutat, akkor az nem más, mint a beszúrandónál közvetlenül nagyobb prioritással rendelkező elem. A beszúrást ebben az esetben a 12–13. sorok végezik el. A közvetlenül előtte lévő elem *köv* mezője az új elemre mutat, annak *köv* mezője pedig felveszi a *p* értékét. Ez utóbbi abban az esetben is jó eredményt ad vissza, ha a *p* egy létező elemre mutat (tehát két elem közé szúrunk be, ezt mutatja az 5.16. ábra), illetve akkor is, ha annak értéke  $\emptyset$  (tehát az utolsó elem mögé szúrunk be, ezt mutatja az 5.15. ábra). Ez utóbbi esetben az új elem *köv* mezőjének tartalma  $\emptyset$  lesz, ami megfelel annak, hogy ő lett a lánc utolsó eleme.

---

### 5.21. Algoritmus Rendezett láncolt listába új elem beszúrása (rövidebb változat)

---

**Bemenet:** érték -  $\mathbf{T}$  (a beszúrandó érték)

**Megszorítás:**  $\mathbf{T}$  összehasonlítható

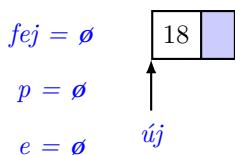
```

1: eljárás LÁNCOLTPRIORITÁSSOR.SORBA(érték)
2: $p \leftarrow fej$
3: $e \leftarrow \emptyset$
4: ciklus amíg ($p \neq \emptyset$) \wedge ($p.tart > érték$)
5: $e \leftarrow p$
6: $p \leftarrow p.köv$
7: ciklus vége
8: ha $e = \emptyset$ akkor
9: $új \leftarrow LÉTREHOZ(LáncElem<T>) \{tart \leftarrow érték, köv \leftarrow \emptyset\}$
10: $fej \leftarrow új$
11: különben
12: $új \leftarrow LÉTREHOZ(LáncElem<T>) \{tart \leftarrow érték, köv \leftarrow p\}$
13: $e.köv \leftarrow új$
14: elágazás vége
15: eljárás vége
```

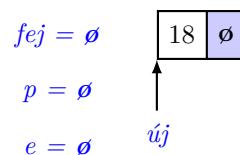
---

#### Felhasznált változók és függvények

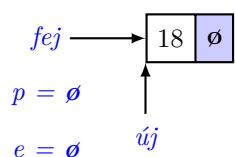
- $fej$  : A prioritás szerint rendezett láncolt lista feje (ami esetleg meg is változhat).
  - $új$ ,  $p$ ,  $e$  :  $LáncElem<T>$  típusú segédváltozók.
  - $LÉTREHOZ(LáncElem<T>)$  : Létrehoz egy új láncelemet, és visszaad egy hivatkozást erre.
- 



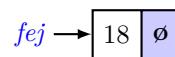
(a) Üres listába beszúrás esetén a változók állapota a ciklus után. Létrehozzuk az új elemet a megadott tartalommal.



(b) Az új elem következő mezőjeként a  $fej$  értékét adtuk meg.

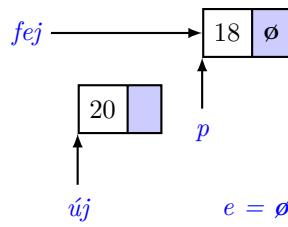


(c) A  $fej$  értékének beállítása.

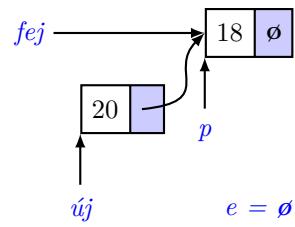


(d) A beszúrás eredménye.

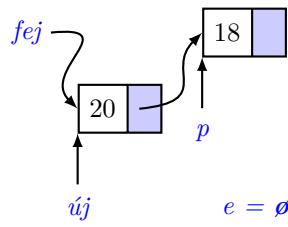
5.13. ábra. Első elem beszúrása üres rendezett láncolt listába. Az új elem tartalma: 18.



(a) Első elem elé beszúrás esetén a változók állapota a ciklus után. Létrehozzuk az új elemet a megadott tartalommal.



(b) Az új elem itt is a *fej* értékét kapja meg következő mezőként.

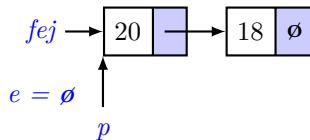


(c) A *fej* értékének beállítása.

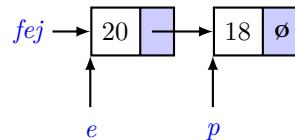


(d) A beszúrás eredménye.

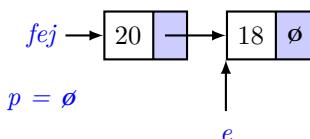
5.14. ábra. Első elem elé beszúrás rendezett láncolt listába. Az új elem tartalma: 20.



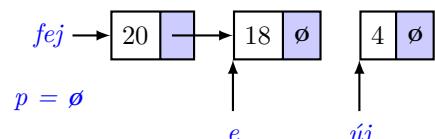
(a) A keresés első lépése, az első elem vizsgálata.



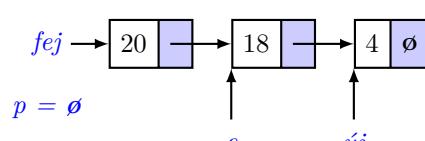
(b) Továbblépés a második elemre. Az *e* követi a *p* változót.



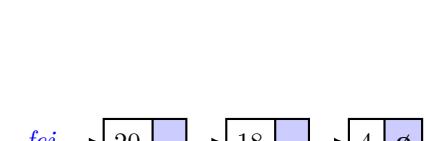
(c) A beszúrandó érték kisebb, mint az utolsó elem, emiatt újabb lépés.



(d) Az új elem létrehozása. Tartalomként a beszúrandó értéket kapja, következőként pedig a *p* értékét, ami most  $\emptyset$ .

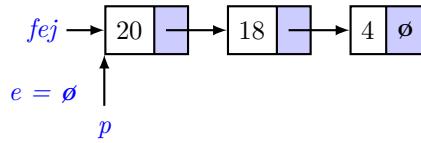


(e) Az előző elem átirányítása az újonnan beszúrt elemre.

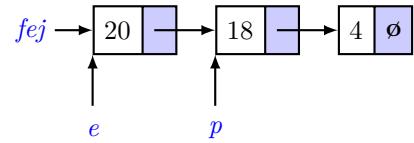


(f) A beszúrás végeredménye.

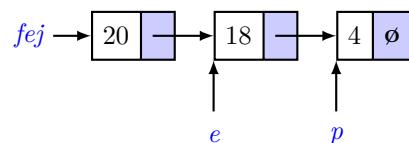
5.15. ábra. Utolsó elem mögé beszúrás rendezett láncolt listába. Az új elem tartalma: 4.



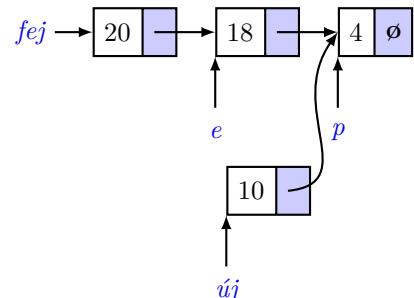
(a) A keresés első lépése, az első elem vizsgálata.



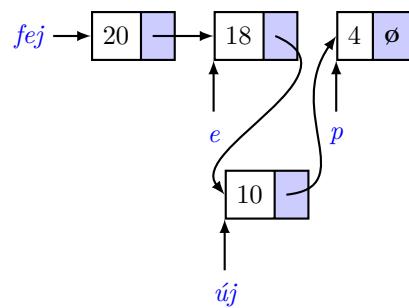
(b) Továbblépés a második elemre. Az  $e$  követi a  $p$  változót.



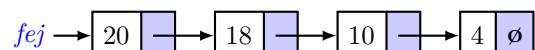
(c) A  $p$  már egy kisebb elemen áll, emiatt kilépés a ciklusból.



(d) Az új elem létrehozása a megadott tartalommal, a következő mezőbe a  $p$  értéke kerül.



(e) Az előző elem átirányítása az újnonnan beszúrt elemre.



(f) A beszúrás végeredménye.

5.16. ábra. Két elem közé beszúrás rendezett láncolt listában. Az új elem tartalma: 10.

## Prioritási sor frissítése

A **PrioritásosSor** interfész előírt még egy **FRISSÍT** műveletet is. Ezt akkor használjuk, ha feltételezhetően megváltozott a prioritási sorban lévő valamely érték prioritása, ezért újra kell rendezni a listát. Erre most egy nagyon egyszerű megoldást fogunk találni: töröljük a listából a kérdéses értéket, majd pedig újra felvesszük a rendezett beszúrás műveettel. Ezt mutatja az 5.22. algoritmus.

---

### 5.22. Algoritmus Láncolt prioritásos sor egy elemének frissítése

---

**Bemenet:** érték - **T** (az érték, aminek a prioritása megváltozhatott)

**Megszorítás:** **T** összehasonlítható

- 1: eljárás LÁNCOLTPRIORITÁSSOR.SORBA(érték)
  - 2: TÖRÖL(érték)
  - 3: SORBA(érték)
  - 4: eljárás vége
- 

#### Felhasznált változók és függvények

- **TÖRÖL(ÉRTÉK)** : Törli a láncolt listából az **érték** tartalmú elemet.
  - **SORBA(ÉRTÉK)** : Beszúrja a rendezett ráncolt listába az **érték**-et.
- 

#### Megjegyzés

Valóban, ez egy közel sem hatékony „megúszós” megoldás, ennél sokkal hatékonyabbat is készíthetnénk. A láncolt listában az elem hátrafelé mozgatása még egyszerű, az előremozgatás azonban már kicsit összetettebb kódot igényelne. Maga a művelet pedig számunkra nem igazán érdekes, csak azért foglalkozunk vele, mert a későbbiekben két gráf algoritmus is használni fogja.

## 5.6. További lehetőségek

### 5.6.1. Láncolt szerkezet strázsa elemekkel

#### Beszúrás és törlés helyi műveletekkel

A különböző láncolt műveletek áttekintése során feltűnő lehetett, hogy sok esetben maga a művelet általános esete egyszerűen kezelhető lett volna (pl. elem beszúrása két másik elem közé), azonban a különféle kivételes esetek miatt a végeredmény egy jóval összetettebb, hosszabb algoritmus lett (pl. üres lánc, első elem elé beszúrás, utolsó elem mögé beszúrás, stb.). Ez nem pusztán hosszabb programkódot eredményez, de a program futásideje szempontjából se szerencsés (pl. még több ezer elem beszúrása után is mindig meg kell vizsgálnunk, hogy a szerkezet nem üres-e véletlenül).

A kivételes esetek elhagyásával néhány nagyon egyszerű és rövid megoldást is tudunk használni, ezek közül át is tekintünk kettőt.

#### Mutatott elem elé beszúrás

Érdemes átgondolni, hogy vajon tudunk-e olyan beszúrást íni, ami nem használja a *fej* mutató értékét. Tehát paraméterként csak a beszúrás helyét kapja meg (egy hivatkozást arra az elemre, amelyik előtt szeretnénk beszúrni), és ez alapján lokálisan próbálja megoldani a beláncolást. Elsőre azt gondolhatnánk, hogy ez nem oldható meg, hiszen a lánc elemeiből nincs visszafelé mutató hivatkozás.

Az 5.17. ábrán látható módon azonban a beszúrást így is működik a legtöbb esetben. Egy elem előtt beszúrást eddig mindenkorábban úgy képzeltünk el, hogy mindenkorábban szükségünk van az azt megelőző elemre is, hiszen a beláncolásnál annak a *köv* hivatkozását is át kell majd irányítani az új elemre. Itt azonban ez nem szükséges, hiszen látható, hogy az új elemet valójában a meglévő után helyezzük el a láncba (5.17c–5.17e. ábrák), a sorrendet pusztán azzal az ötlettel állítjuk helyre, hogy az eddigi *p* által mutatott elem tartalmát átmásoljuk az új elembe, és a régi tartalmat írjuk felül az új értékkel (5.17f. ábra).

Látható, hogy a beszúrás így jóval egyszerűbben megvalósítható, habár néhány mellékhatással is számolunk kell. A legnagyobb problémánk, hogy üres listába a legelső elem beszúrásánál ez nem fog működni. Itt ugyanis nincs semmilyen elem, amit át tudnánk másolni, így ezt a speciális esetet valamilyen formában mindenkorábban kezelnünk kell.

---

#### 5.23. Algoritmus Mutatott elem előtt beszúrás

---

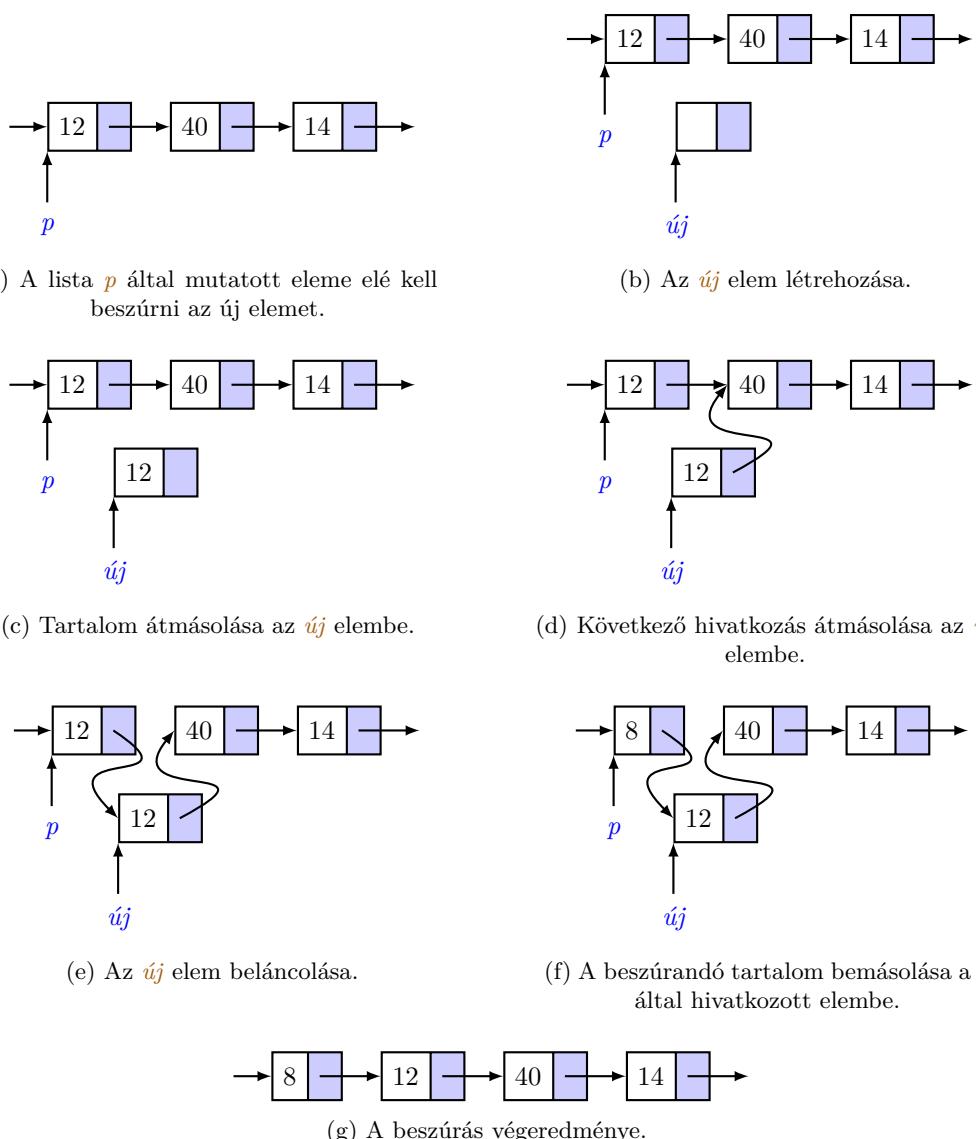
**Bemenet:** *p* - *ŁáncElem*<*T*> (*hivatkozás egy elemre, ami előtt be akarunk szúrni*)

**Bemenet:** *érték* - *T* (*a beszúrandó érték*)

- 1: **eljárás** STRÁZSALÁNC.MUTATOTTÉLEMELÉBESZÚRÁS(*p*, *érték*)
  - 2:     *új*  $\leftarrow$  LÉTREHOZ(*LáncElem*<*T*>) {*tart*  $\leftarrow$  *p.tart*, *köv*  $\leftarrow$  *p.köv*}
  - 3:     *p.köv*  $\leftarrow$  *új*
  - 4:     *p.tart*  $\leftarrow$  *érték*
  - 5: **eljárás vége**
- 

#### Felhasznált változók és függvények

- *új*, *p* : *ŁáncElem*<*T*> típusú segédváltozók.
  - *LÉTREHOZ*(*LáncElem*<*T*>) : Létrehoz egy új láncelementet, és visszaad egy hivatkozást erre.
-



5.17. ábra. Mutatott elem elő beszúrás. A példában a 8 értéket szúrjuk be a lista *p* által mutatott eleme elő.

## Mutatott elem törlése

Az előzőleg megismert ötlet alapján már könnyen kivitelezhető a mutatott elem törlése is. Ennek az algoritmusnak ugyanaz az előnye, nincs hozzá szükségünk a *fej* mutató értékére, az algoritmus csak azt a mutatót kapja meg paraméterként, amit törölünk kell. Az eddigi megoldásaink során ez azért okozott volna problémát, mert a kiláncoláshoz szükségünk volt a megelőző elemre (annak is módosítani kell a *köv* hivatkozását), ezt azonban csak a *p* ismeretében nem tudjuk megtalálni.

Látható, hogy az 5.24. algoritmus a legtöbb esetben enélkül is működik, felhasználva azt az ötletet, hogy a törlendő elem adatait felülírjuk a mögötte lévő elem adataival (3–4. sorok), majd ezt követően a mögötte lévő elemet láncoljuk ki (ez már könnyen megvalósítható, hiszen ismerjük az őt megelőző elemet, így nem kell azt külön megkeresni). A módszer nagyon hatékony és elegáns, azonban néhány kivételes esetben sajnos ez sem működik megfelelően:

- A leglátványosabb hiányossága az utolsó elem törlésekor tapasztalható. Mivel nincs már mögötte elem, ahonnan adatokat lehetne átmásolni, így a legutolsó elem törlésére ez a módszer nem használható.
- Ha a listánkban már csak egy elem van, és ezt szeretnénk törölni, akkor szintén csödöt mond ez a megoldás (bár ez felfogható az előző probléma egy speciális esetének is, hiszen az egyetlen elem egyben nyilván az utolsó is).

---

### 5.24. Algoritmus Mutatott elem törlése

---

**Bemenet:** *p* - *ŁancElem* (*hivatkozás a törlendő elemre, ami nem az utolsó*)

- 1: **eljárás** STRÁZSALÁNC.MUTATOTT ELEM TÖRLÉSE(*p*)
  - 2:     *q*  $\leftarrow$  *p.köv*
  - 3:     *p.tart*  $\leftarrow$  *q.tart*
  - 4:     *p.köv*  $\leftarrow$  *q.köv*
  - 5:     FELSZABADÍT(*q*)
  - 6: **eljárás vége**
- 

#### Felhasznált változók és függvények

- *q* : *ŁancElem*<*T*> típusú segédváltozó.
  - **FELSZABADÍT(*q*)** : A *q* által mutatott elem felszabadítása.
- 

## Strázsa elemek használata

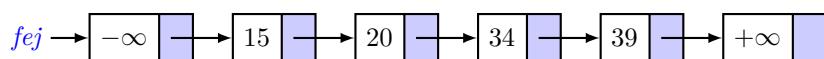
A felmerülő problémák ellenére azonban célszerű alaposabban megvizsgálnunk ezeket a módszereket. Mindkét algoritmusnál csak a kivételes esetek jelentik a problémát (első elem, egyetlen elem, utolsó elem, stb.), általános esetben (amikor egy lánc egy belső eleméről beszélünk) azonban jól használhatók. Ezen alapul a strázsa technika alapötlete, hogy próbálunk meg egy olyan láncolt szerkezet megvalósítást létrehozni, amelyben nincsenek ilyen speciális esetek.

A hagyományos láncolt listát úgy alkottuk meg, hogy a *fej* mutató közvetlenül az első elemre mutat, üres lánc esetében ennek az értéke pedig a lezáró  $\emptyset$  érték. Ugyanezt használtuk a lánc végének jelzésére, ezért az utolsó elem következő mezőjébe is ezt helyeztük.

Ehelyett készítsünk egy olyan láncolt listát, ahol minden van egy-egy láncelem az első valós adatot tároló elem előtt, illetve után is. Ezeket nevezzük a későbbiekbén *strázsa*<sup>2</sup> elemeknek. Ennek megfelelően az üres, tehát értékes adatot nem tartalmazó listát is úgy képzeljük el, hogy már van benne két láncelem (5.18. ábra).

---

<sup>2</sup>sentinel



5.18. ábra. Láncolt lista strázsa elemekkel.

A strázsa elemek felépítésüket tekintve pont ugyanolyanok, mint bármelyik másik láncelem, valahogy azonban jó lenne megkülönböztetni őket a többi elemtől. Erre használhatnánk egy új mezőt is, de az megnövelné a teljes szerkezet helyfoglalását, ezért ehelyett, (amennyiben annak értékkészlete ezt megengedi) használhatjuk a *tart* mezőt, amibe olyan értékeket töltünk, amelyek egyértelműen jelzik a két elem speciális voltát. Ha ez nem megoldható, akkor is rá lehet jönni, hogy melyik elemek a strázsák, hiszen az egyikre minden a *fej* mutat, a másiknak pedig a *köv* mezőjének értéke  $\phi$ , ezek egyike se lehet igaz más láncelem esetében.

A *tart* mezőbe tehát olyan értékeket töltünk, amelyek más elemekre biztosan nem igazak. Amennyiben rendezett a listánk, akkor az első strázsába célszerűen az értékkészlet legkisebb elemét töltjük (jelölése:  $-\infty$ ), az utolsó strázsába pedig az értékkészlet legnagyobb elemét (jelölése:  $+\infty$ ). Ezek konkrét értéke attól függ, hogy milyen típusú a tartalom, számok esetén az adott típus legkisebb/legnagyobb értéke lehet, szövegek esetén ASCII(0), stb. Ha sikerül ilyen értékeket választanunk, az azért is hatékony, mivel rendezett láncolt szerkezet esetén, a rendezettség szempontjából nem kell külön kezelnünk a strázsa elemeket, azok teljesen illeszkednek a valós tartalmat tároló elemek közé. Ennek köszönhetően pl. beszúrás esetén minden előtérrel a feltételezéssel, hogy találni fogunk egy olyan elempárt, amelyek egyike kisebb a beszúrandónál, a másik pedig nagyobb:

- Ha az eddigieknel kisebb elemet akarunk beszúrni, akkor ez az elempár az első strázsa és az eddigi legkisebb elem lesz.
- Ha az eddigieknel nagyobb elemet akarunk beszúrni, akkor ez az elempár az eddigi legnagyobb és a mögötte lévő strázsa lesz.
- Ha pedig az eddigi legkisebb és legnagyobb közötti tartalmat szúrunk be, akkor értelemszerűen találni fogunk két, a fenti feltételnek megfelelő elemet.
- Üres lánc esetén is létezik a két strázsa elem, tehát ilyenkor ezek közé fog kerülni az új érték.

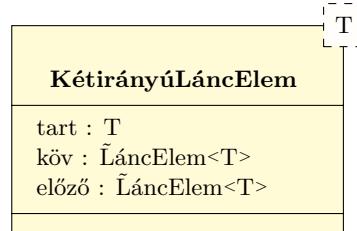
A konkrét pszeudokód bemutatása nélkül is nyilvánvaló, hogy ezzel pl. a rendezett beszúrást jelentősen leegyszerűsíthetjük, hiszen a négyféle lehetőség közül (üres lánc, első elem elé, két elem közé, utolsó elem mögé) már csak egygel kell foglalkoznunk. Ugyanez az előny látható a törlésnél is.

A strázsa technika egyetlen hátránya a nagyobb helyfoglalás, viszont ez sem tekinthető jelentősnek, hiszen csak két új elem felvételét jelenti, tehát a szerkezet méretétől függetlenül konstans lesz a helyfoglalási többlet (ami értelemszerűen sokkal jobb, mintha minden lánclelemnél növelnünk kellett volna a helyfoglalást).

### 5.6.2. Kétirányú láncolt szerkezet

Az eddigiek során minden előtérrel egyirányú listákkal foglalkoztunk, tehát minden elem a tartalmi rész mellett egy másik elemre hivatkozott, amelyik a listában a következő. Néhány algoritmusnál láttuk, hogy ez számos problémát felvet, néha hasznos lett volna egy visszalépési lehetőség, amivel egy elemből el tudunk jutni a láncban előtte lévőre.

A *kétirányú láncolt listák*<sup>3</sup> (esetenként kétszeresen láncolt listának is nevezik, bár ez félreérthető lehet [5]) egy egyszerű megoldást nyújtanak erre a problémára, az egyes elemeket ugyanis kiegészítik egy visszafelé mutató hivatkozással. Ennek megfelelően minden elem felépítése az alábbi:



5.19. ábra. Kétirányú láncolt szerkezet egy elemének felépítése.

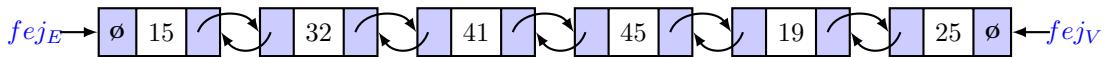
<sup>3</sup>doubly linked

- Tartalmi rész: ez a rész tartalmazza magát a láncelemben eltárolni kívánt adatot. Megfelel az egyirányú láncolt listában is tárgyalt *tart* mezőnek.
- Következő hivatkozás rész: a lánc minden eleme tartalmaz egy hivatkozást a listában rákövetkező elemre. A legelső elemtől elindulva így (közvetve, a közbülső elemeken keresztül) elérhetjük a lánc bármelyik elemét. A mező típusa a lánc megvalósításától függően többféle lehet.
- Előző hivatkozás rész: a lánc minden eleme tartalmaz egy hivatkozást a listában őt megelőző elemre. Ennek típusa ugyanaz, mint amit a következő mezőnél láthatunk.

A kétirányú láncolt szerkezet előnye nyilvánvaló, lehetővé teszi a visszafelé lépkedést. Az 5.20. ábra egy ilyen láncolt listát mutat be. Ugyan nem nélkülözhetetlen, de gyakran hasznos lehet egy új fejet is felvenni, ami a legutolsó elemre mutat, ebből indulva visszafelé is be tudjuk járni a listát. Amennyiben az általunk használt algoritmusok ezt a tulajdonságot kihasználják, akkor ezzel a módszerrel hatékony megoldásokat készíthetünk.

A hátránya a hagyományos listához képest az, hogy a két mutató miatt a lánc elemeinek a helyfoglalása nagyobb (ami már az előzőleg megismert láncolt listánál is problémát okozott). Továbbá a különféle módosító algoritmusok valamelyest bonyolultabbak, hiszen minden beszúrásnál/törlésnél be kell állítani az új elem előző mutatóját is, illetve a listában utána következő elemnél is ezt el kell végezni. Hasonló módon, két fej esetén minden két fej folyamatosan karban kell tartani az elvégzett módosításnak megfelelően.

Bár ezek a módosító műveletek bonyolultabbak, mint az egyszerű egyirányú listánál, teljesítmény szempontjából akár hatékonyabbak is lehetnek. Elég csak arra gondolunk, hogy a kétirányú láncolt listával egyszerűen megoldhatjuk a mutatott elem elé beszúrást, illetve a mutatott elem törlését, hiszen nincs szükség egy hosszadalmas keresésre a megelőző elem megtalálásához.

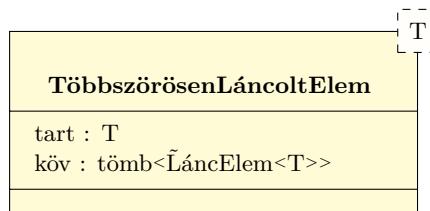


5.20. ábra. Kétirányú láncolt lista.

### 5.6.3. Többszörösen láncolt szerkezet

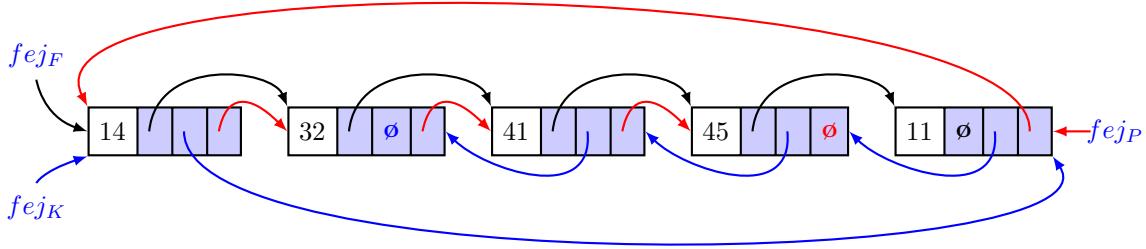
A *többszörösen láncolt szerkezet*<sup>4</sup> tekinthető a kétirányú láncolt szerkezet általánosításának abból a szempontból, hogy itt már nem csak két, hanem akár három, vagy annál több elemre is hivatkozhatnak az egyes elemek. A két struktúra közötti lényeges különbség, hogy míg a kétirányú láncolt listánál az előre és a hátra mutató hivatkozások mindig egymásnak ellenkező irányokat adtak meg, addig egy többszörösen láncolt szerkezet esetében ezek teljesen függetlenek tekintendők. Tehát elképzelhető egy kétszeresen láncolt szerkezet, ahol minden elem két következő mezőt tartalmaz, és ezek egymástól független sorrendeket adnak meg.

Egy háromszorosan láncolt listára mutat példát az 5.22. ábra. A lényeges különbség a hagyományos listához képest, hogy minden láncelem három darab következő hivatkozást tartalmaz, amelyek egymástól függetlenül mutatnak egy-egy további elemre (vagy éppen a  $\emptyset$  értékkel jelzik, hogy nincs ilyen). Célszerű úgy elképzelni ezt a szerkezetet, mintha három, egymástól teljesen független láncolt listánk lenne, amelyek a tartalom részét megosztják egymás között. Ennek megfelelően rendelkezünk három *fej* változóval is, amelyek az egyes listáknak az első elemeire mutatnak (és látható, hogy a megadott *köv* mezőt követve, minden három lánc esetében találunk egy lezártó  $\emptyset$  jelet is).



5.21. ábra. Többszörösen láncolt szerkezet egy elemének felépítése.

<sup>4</sup>multiply linked



5.22. ábra. Többszörös láncolt lista. Az példában az egyes fejekből kiinduló listák adatai: fekete: 14, 32, 41, 45, 11; kék: 14, 11, 45, 32; piros: 11, 14, 32, 41, 45.

Egy n-szeresen láncolt szerkezet felépítése tehát az alábbi:

- Tartalmi rész: ez a rész tartalmazza magát, a láncremben eltárolni kívánt adatot. Ez megfelel az egyirányú láncolt listában is tárgyalt *tart* mezőnek.
- Következői hivatkozás rész (ahol  $1 \leq i \leq n$ ): az  $i$ . lánc alapján a következő elemre való hivatkozás. Ezekre célszerű egy tömb típusú mezőt létrehozni.

A szerkezet hátránya természetesen itt is az, hogy minél több hivatkozást tárolunk, annál nagyobb tárterületet fog követelni az adatszerkezet. A módosító algoritmusok is több odafigyelést igényelnek, hiszen pl. minden törlésnél ügyelni kell arra, hogy minden sorrend szerint megfelelően láncoljuk ki az elemet (ami a lépésszámok tekintetében is meglehetősen hátrányos lehet, ha ez minden újabb keresésekkel jelent).

Bizonyos esetekben azonban mégis érdemes lehet használni ezt a megoldást, főleg olyankor, amikor ugyanazokat az adatokat többféle sorrendben is el szeretnénk tárolni. Pl. személyek adatait eltárolhatjuk életkor szerint, név szerint és igazolvány szám szerint rendezve. A feldolgozáskor pedig minden azon a lánon haladunk végig, amelyik az aktuális szempontnak a leginkább megfelel.

Szintén előnyös tulajdonsága ennek a módszernek, hogy függetlenül attól, hogy több láncot kezelünk, a tartalmi részt csak egyszer kell eltárolnunk, hiszen minden lánc ezt ugyanabban az elemben tárolja. Ez egyszerű hatékony tárhely-kihasználást tesz lehetővé, másrészről a módosításoknál ilyenkor csak egy helyen kell változtatni az adatokat, nem szükséges azt minden láncban külön-külön megtenni.

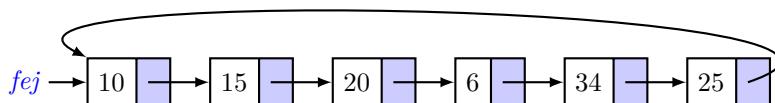
#### 5.6.4. Ciklikus láncolt szerkezet

A *ciklikus láncolt szerkezet*<sup>5</sup> [5] elemeinek a felépítése nem különbözik jelentősen az egyszerű egyirányú láncolt listánál tanultaktól. minden elem ugyanúgy egy tartalmat és egy következő elemre vonatkozó hivatkozást tárol.

A különbség pusztán az (miként az 5.23. ábrán is látható), hogy hiányzik a lánc végéről a lezáró elem. A szerkezet utolsó eleme (már ha egyáltalán utolsónak nevezhető ebben a szerkezetben) az első elemre hivatkozik, így az egész lánc egy kört alkot.

A megoldás egyik előnye az, hogy ezzel megszabadulunk néhány kivételes esettől. Néhány algoritmusnál külön kellett foglalkoznunk az első és utolsó elemekkel, mivel ezek előtt/után már nincsenek további elemek. A ciklikus láncolt szerkezet esetében ez láthatóan nem áll fenn, hiszen (ha már vannak elemeink a listában), akkor minden elemnél fogunk találni rakkövetkezőt, illetve előzőt.

<sup>5</sup>circular



5.23. ábra. Ciklikus láncolt lista.

A gyakorlatban ez a szerkezet főleg olyan feladatoknál használható ki jól, amelyek eleve hasonló, gyűrűszerű tárolást igényelnek. Ilyen lehet például a már megismert véges méretű *sor* adatszerkezet, ahol a folyamatosan behelyezett elemek egy idő után felül kell, hogy írják a legrégebben felvett elemeket. A ciklikus lánc ezt természetesen támogatja, hiszen mindenkor a következő elemre lépegetéssel az „utolsó” elem után azonnal az első elemre lépünk, így megkezdődhet a legrégebbi elemek felülírása.

Mindez persze jelentős változtatásokat igényel a megismert algoritmusok esetében is, hiszen pl. a már megismert bejárás ebben az esetben egy végtelen ciklusba kerülne.

Esetenként érdemes lehet a különféle megismert láncolt szerkezet megvalósításokat kombinálni is. Például, ha az algoritmusaink ki tudják használni annak speciális tulajdonságait, érdemes lehet használni kétirányú ciklikus listákat is akár.

## 6. fejezet

# Bináris keresőfa

### 6.1. Bináris keresőfa felépítése

#### 6.1.1. Alapvető fogalmak

A gráfelméletben fának nevezzük a körmentes, összefüggő gráfokat. Mi azonban egy kicsit más definíció alapján közelítjük meg ezeket [1]: egy *fa*<sup>1</sup> nem más, mint *csomópontok*<sup>2</sup> halmaza, amelyeket *élek*<sup>3</sup> kötnek össze, és teljesülnek az alábbi feltételek:

- létezik egy kitüntetett csomópont: a *gyökér*<sup>4</sup>,
- a gyökértől különböző minden más csomópont egy éssel van összekötve a szülőjével,
- összefüggő, tehát bármelyik nem-gyökér csomópontból kiindulva, a szülőkön keresztül eljutunk a gyökérhez.

A két definíció közötti (számunkra a megvalósítás szempontjából nagyon fontos) különbség az, hogy ez utóbbi esetben a fa egy csomópontját kitüntetékként kezeljük. A láncolt lista fejéhez hasonlóan, ez lesz a fa egyetlen kívülről is elérhető eleme, amelyen keresztül (közvetlenül vagy közvetve) hozzáérhetünk majd a többihez. Emiatt az így kapott adatszerkezetet nevezhetjük „gyökeres fa” szerkezetnek is.

Ebben a fejezetben csak úgynévezett *bináris fákra*<sup>5</sup> lesz szó, ahol minden szülőelemnek legfeljebb kettő gyereke lehet (tehát a gyerekek száma 0, 1 vagy 2). A bináris keresőfa tárolása során ezt a két gyerekekre való hivatkozást gyakran nevezzük *bal-*, illetve *jobboldali* hivatkozásnak. Ennek megfelelően beszélhetünk *bal-* illetve *jobboldali* gyerekről, és *bal-* illetve *jobboldali* részfáról is. Néhány további, a későbbiekben használni kívánt fogalom (ahol *x* és *y* csomópontok):

- Ha létezik egy *x*-ből *y*-ba vezető él, akkor *x*-et az *y szülőjének*, *y*-t pedig az *x gyerekének* nevezzük. Például a 6.1. ábrán a 31-es elem szülője a 24-es, illetve a 24-es gyereke a 31-es. A gyökérelemnek nincs szülője, minden más csomópontnak pontosan egy van.
- Az *x* és *y* elemek testvérek, ha azonos a szülőjük. A 6.1. ábrán két testvér például az 58 és a 71.
- Szükség esetén az emberi családfákhoz hasonló módon definiálhatók a nagyszülő, unokatestvér, stb. fogalmak is.
- Az út csomópontok és élek váltakozó sorozata, amely minden csúccsal kezdődik és végződik, és minden csúcsot valamelyik gyereke követi a sorozatban, illetve minden él két végpontja az előző és követő csomópont. Az út hossza az előző alkotó élek száma.
- Ha vezet út *x*-ből *y*-ba, akkor *x* az *y ōse*, az *y* pedig az *x leszármazottja*. Amennyiben  $x \neq y$ , akkor valódi ōsről, illetve valódi leszármazottról beszélünk. A 6.1. ábrán az 50-es elem (valódi) ōse a 16-osnak, míg a 16-os (valódi) leszármazottja az 50-esnek.

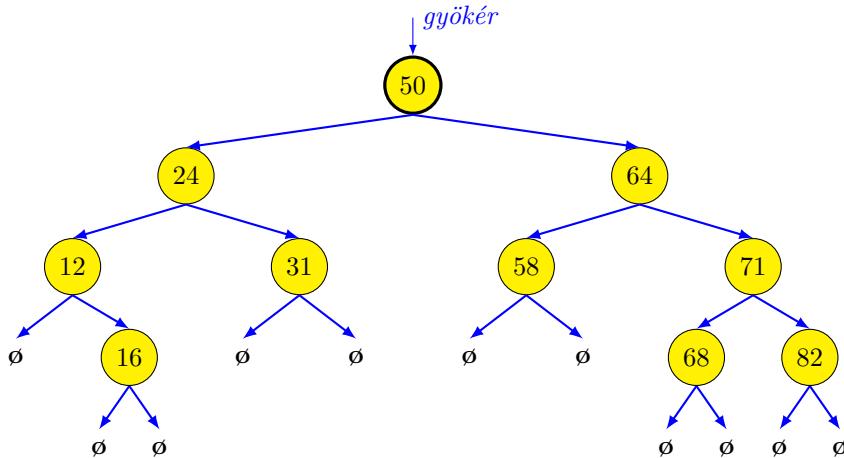
<sup>1</sup>tree

<sup>2</sup>nodes

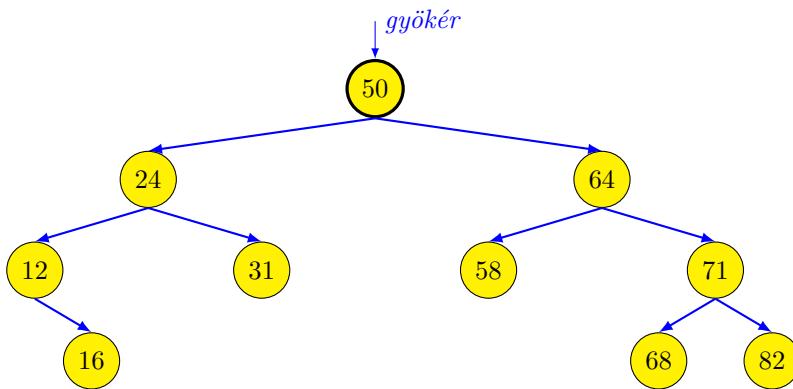
<sup>3</sup>edges

<sup>4</sup>root

<sup>5</sup>binary tree



(a) A teljes fa, ahol ábrázoljuk a levelek lezáró bal és jobb hivatkozásait.



(b) Ugyanaz a bináris fa, csak a későbbiekben is látható módon, itt már nem ábrázoljuk a levelek alatt található lezáró ø jeleket.

6.1. ábra. Egy minta bináris keresőfa, amelyben egész számokat tárolunk.

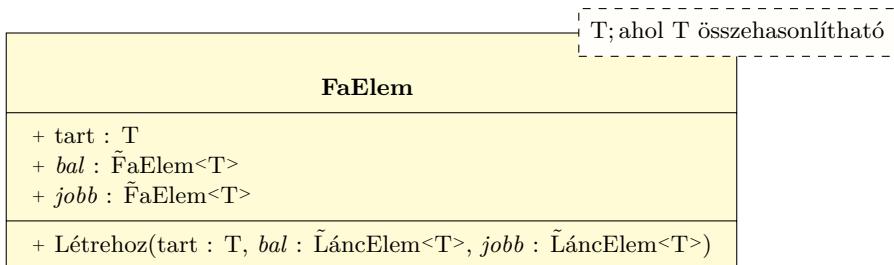
- Valódi leszármazottal nem rendelkező csomópontokat *leveleknek* nevezzük. A 6.1. ábrán a levelek az alábbiak: 12, 16, 31, 58, 68, 82.
- Egy  $x$  csomópont az összes leszármazottjával egy részfát alkot, aminek a gyökere az  $x$ . Tehát a 6.1. ábrán a 64 gyökerű részfa elemei: 64, 58, 71, 68, 82.
- Egy *csomópont mélysége* egyenlő a gyökértől hozzá elvezető út hosszával.
- Egy *csomópont magassága* a leghosszabb, innen kiinduló és a levelekig vezető út hossza.
- A *fa magassága* a gyökér magassága.

### 6.1.2. Egy elem felépítése

A bináris fák egy speciális esetének tekinthetők a bináris keresőfák. A *keresőfák* célja a hatékonyabb keresés érdekében az adatok olyan reprezentációja, amely gyors visszakeresést és dinamikus használatot tesz lehetővé [7]. Ez az igény a bináris fát annyiból specializálja, hogy a fában található tartalmak rendezve vannak az alábbiak szerint: minden egyes csomópontra igaz, hogy a baloldali részfájában lévő csomópontok nála kisebb, a jobboldali részfájában lévő csomópontok pedig nála nagyobb elemeket tartalmaznak (vagy épp fordítva). A 6.1. ábra egy példát mutat bináris keresőfára. Ezen adatszerkezet neve a *bináris keresőfa*<sup>6</sup>.

A fa elemeket egy összetett típussal tudjuk megvalósítani, amely a 6.2. ábrán látható.

<sup>6</sup>binary search tree



6.2. ábra. Bináris keresőfa egy elemének felépítése.

- Tartalmi rész: ez a rész tartalmazza magát, a fában eltárolni kívánt adatot. Nincs semmiféle megkötésünk ennek a típusára, lehet szám, szöveg, vagy akár tetszőleges osztály típus is. Azt azonban általában kikötjük, hogy az egyes csúcsok minden azonos típusú adatokat tartalmazzanak. Kihasználva a generikus típusok lehetőségeit, ezt a mezőt minden *tart* néven fogjuk használni, ami **T** típusú.

Mivel keresőfáról beszélünk, az egyes elemeknek összehasonlíthatónak kell lenniük. Ezt a rendezett láncolt listánál már látott módon úgy oldjuk meg, hogy megadunk egy összehasonlíthatósági megszorítást a **T** típusra. Két fa elem összehasonlításának eredményeként pedig azok tartalmainak összehasonlítási eredményét tekintjük majd. Persze más megoldást is választhatunk, például bevezethetünk egy új mezőt, ami alapján a rendezést megvalósítjuk. Az egyszerűség kedvéért mi az első változatot fogjuk használni (6.2. ábra).

- Bal hivatkozás rész: a fa minden eleme tartalmaz egy hivatkozást a baloldali részfájának első elemére. A mező típusa a fa megvalósításától függően többféle lehet. Az általunk használt absztrakt típus jelölések ezt a technikai részletet elfedik, de az egyszerűség kedvéért itt is a legegyszerűbb a fa elemeit objektumként, a *bal* és *jobb* mezőt pedig objektum referenciaiként felfogni. A példákban ezt a mezőt minden *bal* néven fogjuk használni. Amennyiben egy csomópontnak nincs baloldali részfája, úgy ennek a mezőnek az értéke  $\emptyset$  lesz. A mező **FaElem** típusú.
- Jobb hivatkozás rész: értelemszerűen a jobboldali részfa első elemére hivatkozik. A példákban *jobb* néven hivatkozunk erre a mezőre. Amennyiben nincs jobboldali részfa, akkor a mezőnek az értéke  $\emptyset$  lesz. A mező **FaElem** típusú.

A bináris fa egy eleme technikailag nagyon hasonlít az előző fejezetben megismert láncolt listához, azzal a kiegészítéssel, hogy a tartalmi rész mellett nem csak egy *köv*, hanem egy-egy *bal* illetve *jobb* hivatkozást tartalmaz. Látni fogjuk azonban, hogy ez a kis kiegészítés jelentősen meg változtatja a feldolgozó és módosító műveleteinket (pl. már a legegyszerűbb bejárásnál is felmerül a kérdés, hogy a gyökérelemből kiindulva mit értünk „következő” elem alatt).

### 6.1.3. Fa felépítése

A láncolt listánál szükségünk volt egy *fej* nevű külső hivatkoásra, ami a lista első elemére mutatott. A fánál erre szintén szükségünk lesz, itt azonban hagyományosan *gyökér*-nek szoktuk elnevezni, habár a szerepe teljesen azonos. Ezen keresztül tudjuk elérni a fa gyökérelemét, majd azon keresztül a többet. Üres fa esetében a *gyökér* változó értéke  $\emptyset$ .

Érdemes észrevenni a bináris keresőfa esetében annak rekurzív felépítését. A fát úgy is elképzelhetjük, hogy az áll egy gyökérelemből, illetve egy bal és jobboldali részfából. Ahol minden részfa tulajdonképpen egy gyökérelem és az ahhoz tartozó bal és jobboldali részfa, stb. A rekurzív adatszerkezeteket gyakran rekurzív algoritmusokkal tudjuk a leghatékonyabban feldolgozni, itt is erre látunk majd példákat. Érdekes megjegyezni, hogy az összes felsorolt művelet megvalósítható nem rekurzív formában (sőt, ez még hatékonyabb megoldást is adhat).

### Megjegyzés

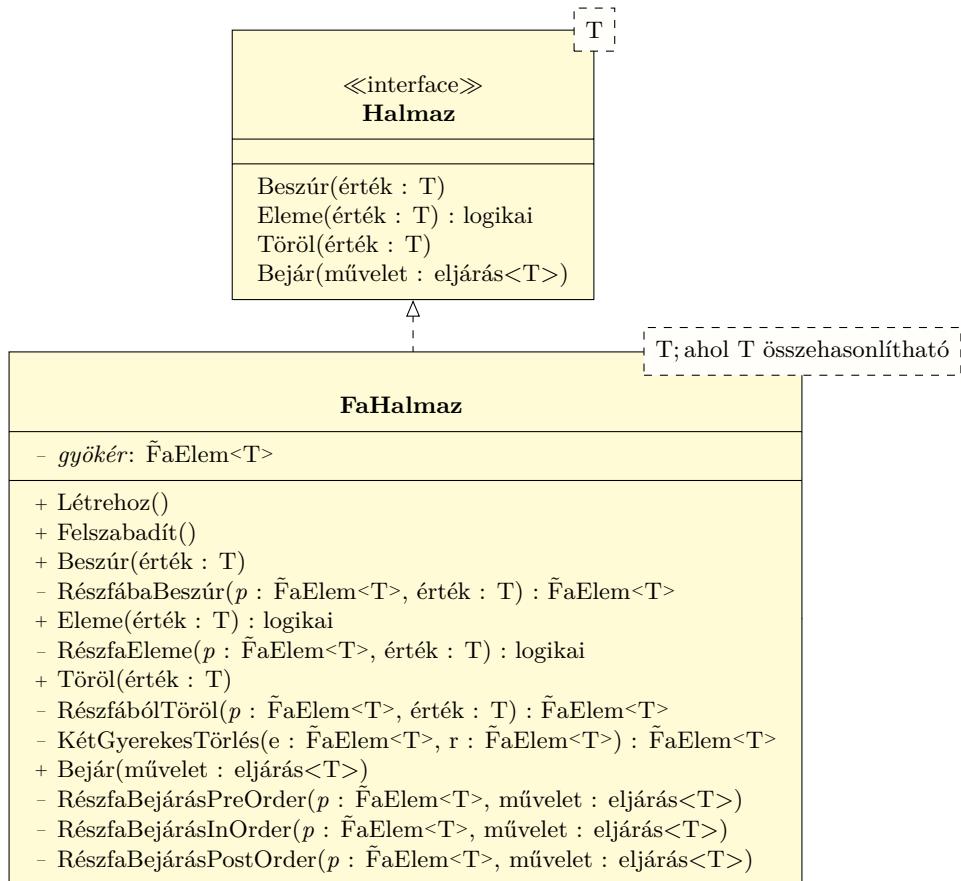
A láncolt lista esetében egyébként ugyanígy megfigyelhető egy rekurzív jellemző. Hiszen maga a lista felfogható úgy, hogy áll egy fej elemből, illetve egy ahhoz kapcsolt további láncolt listából. Ennek megfelelően a láncolt lista feldolgozások egyszerűen megadhatók rekurzív műveletekkel is. Míg azonban a fák esetében a rekurzív megoldást fogjuk gyakran egyszerűbbnek és természetesebbnek találni, addig a láncolt listák esetében ezek gyakran öncélúnak és feleslegesen bonyolultnak tűnnek.

## 6.2. Halmaz megvalósítása bináris keresőfával

### 6.2.1. Alapvető felépítés

Halmaz esetében a kiolvasó művelet csak arra igényel választ, hogy megadott elem benne van-e a már a halmazban. Nem szükséges eltárolni az egyes elemek sorrendjét (mint a listánál), sem pedig az elemek egymás utáni felvételének idejét (mint a **Verem** vagy a **Sor** esetében). Cserébe viszont minél hatékonyabb kiolvasó műveletet szeretnénk kapni.

Ennek érdekében célszerű úgy tárolni az adatokat, hogy az adatszerkezet minél gyorsabb ellenőrzést tudjon végezni. Ehhez egy jó döntés lehet, ha rendezetten tároljuk az adatokat, mivel már láthatunk, hogy ez jelentősen gyorsítja a későbbi keresést. A láncolt listánál ez nem sokat segített volna, hiszen indexelés hiányában a rendezett tömbben se tudtunk volna gyorsabban keresni. A fánál viszont látni fogjuk, hogy nem ez a helyzet, ezért ez tipikusan jó választás lehet egy **Halmaz** megvalósításra.



A fa elemeit úgy hozzuk létre, hogy a **T** típusú tartalom összehasonlítható legyen. Ez azért fontos, mert ez alapján fogunk majd tudni gyorsan keresni a fában.

### 6.2.2. Bejárások

Az eddig megszokottal ellentétben most nem a beszúrási műveletekkel kezdjük a vizsgálatot, mivel azok itt már nemileg barátságtalanabbak, mint amit az előzőekben megszokhattunk. A bejárás viszont könnyen érthető, és jól bemutatja a fa rekurzív voltából adódó specialitásokat.

A láncolt listánál már láttuk, hogy a bejárás egy olyan művelet, amely az adatszerkezet minden egyes elemét pontosan egyszer dolgozza fel. A listánál szinte nem is találhatunk más módot a bejárásra, csak amit meg is vizsgáltunk. A **fej** hivatkozásban keresztül ugyanis csak az első elemet tudjuk elérni, innen pedig csak minden a következő elemekre tudunk tovább lépegetni. Ebből adódóan az elemek bejárási sorrendje minden megfelelt azok láncbeli sorrendjének.

A fák esetében ez már nem ilyen egyértelmű. Ha megnézzük a 6.1. ábrát, akkor látható, hogy maga a kezdőpont itt is egyértelmű (hiszen a fából kívülről csak a *gyökér* hivatkozást látjuk, ami a gyökér elemre hivatkozik), azonban az innen való továbblépés már több irányba is történhet. Továbbléphetünk elsőként a 24, illetve a 64 tartalmú elemre is. Majd az ezekről való továbblépés ugyanígy több irányban is megtörténhet.

A fentieknek megfelelően a bináris keresőfánál többféle bejárás műveletet is megkülönböztetünk. Érdemes megjegyezni, hogy maga a bejárás definíciója ugyanaz maradt így is, minden egyik bejárás minden elemet pontosan egyszer fog feldolgozni. Az egyetlen különbség pusztán az, hogy milyen sorrendben érjük el a fában található csomópontokat, illetve mikor dolgozzuk fel az azokban tárolt tartalmakat.

### Preorder bejárás

Az egyik lehetséges bejárási mód a fa *preorder* bejárása. A 6.1. algoritmus mutatja a hozzá tartozó pszeudokódot.

A bejárás lépései az alábbi sorrendben dolgozzák fel a csomópont elemeit:

1. csomópont tartalmának feldolgozása,
2. csomópont baloldali részfájának feldolgozása,
3. csomópont jobboldali részfájának feldolgozása.

Az eljárás hívását követően minden ellenőrizziük, hogy a paraméterként átadott *p* hivatkozás egy valódi csomópontra mutat-e. Üres fa esetében ugyanis elképzelhető, hogy már a meghíváskor az ilyenkor szokásos  $\emptyset$  értéket kapjuk első paraméterként. Amennyiben az ellenőrzés azt mutatja, hogy a változó értéke megfelelő, akkor feldolgozhatjuk az általa hivatkozott csomópont tartalmát.

Ezt követi a csomópont részfáinak a feldolgozása. Először a csomópont baloldali részfájában lévő elemeket dolgozzuk fel, ezt egy rekurzív hívással egyszerűen meg tudunk tenni. Miután a rekurzió visszatér (tehát a baloldali részfa minden eleme fel lett dolgozva), következhet a jobboldali részfa feldolgozása egy újabb rekurzív hívással. Miután minden művelet befejeződött, az eljárás véget ér, ezzel a rekurzió visszalép egy szinttel (illetve a legelső szinten visszalép az őt meghívóhoz).

A 2. sorban található *p* ellenőrzésnek a rekurzió szempontjából is fontos szerepe van. Az egyes részfák feldolgozása során egyre mélyebbre kerülünk a fában, és mivel feltételezzük, hogy az véges számú elemet tartalmaz, így előbb-utóbb elérünk a levelekig. Ezeknek már nincsenek bal-, illetve jobboldali gyerekeik, de a rekurzió ugyanúgy meghívódik ezekre az ágakra is. Ilyenkor az eljárás következő szintje paraméterként minden  $\emptyset$  értéket kap majd (hiszen ezzel jelöltük a levelekét, hogy hivatkozásaik nem valódi elemekre mutatnak), így ezzel állítjuk meg a rekurziót. Mindez persze nem csak a levél elemek esetében, hanem az egy gyerekkel rendelkező csomópontoknál is hasonlóan működik.

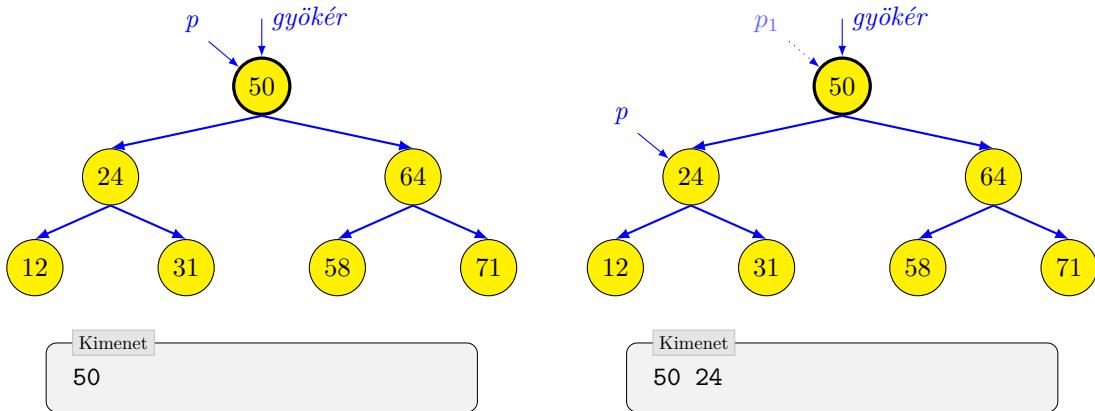
A 6.3. ábra lépésein körülöleli a rekurzió lépéseit (az alsó indexsel rendelkező *p* változók az előző rekurziós szintek értékeit mutatják). A végeredményként kapott számsor fontos szerepet kap a teljes fa elmentésekor, mivel ez alapján fel tudunk építeni egy, az eredetivel megegyező fát.

#### Megjegyzés

A számsor értelmezéséből talán már sejthető, hogy a beszúrások sorrendje befolyásolja a létrehozott fa szerkezetét. Tehát még ha ugyanazokat az elemeket is helyezzük el a fában, de különböző sorrendben, akkor is elképzelhető, hogy különböző szerkezetű fákat kapunk. Mindez persze nem törvényszerű, könnyen találhatunk olyan példákat, ahol különböző sorrendben megadott (de természetesen azonos) számokat felvéve ugyanahhoz a fához jutunk. Érdemes egy-egy példát keresni ezekre.

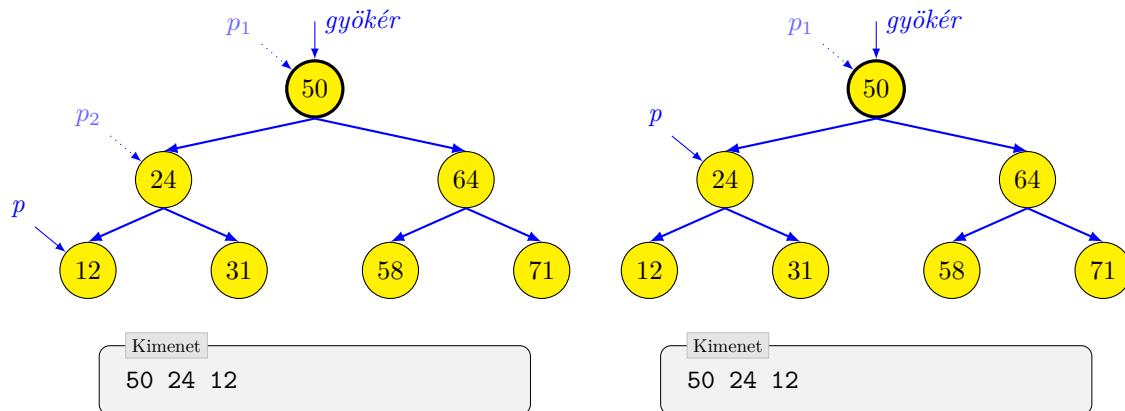
### Inorder bejárás

A bináris keresőfa *inorder* bejárása ez előzőek alapján már valószínűleg könnyen érthető lesz (6.2. algoritmus). A bejárás egészen hasonló módon járja be az elemeket, pusztán abban különbözik a megoldás a preordernél láthatótól, hogy más az egyes csomópontokban elvégzett műveletek sorrendje:



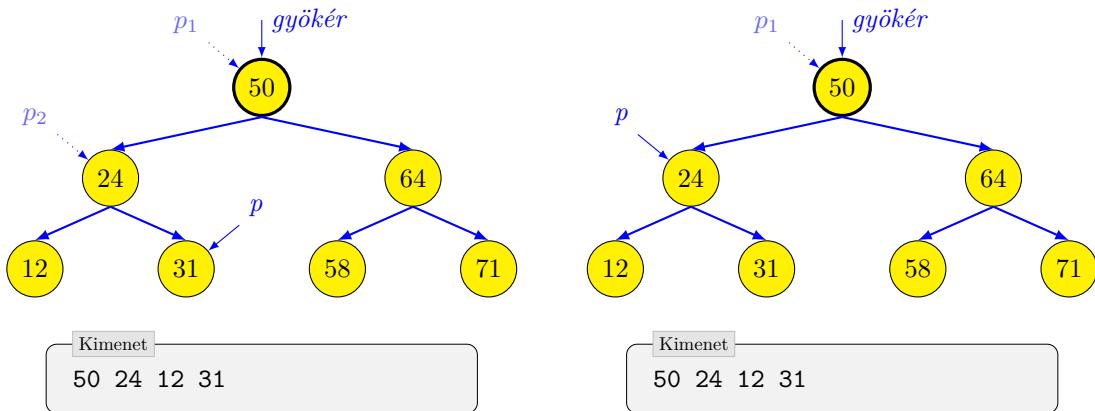
(a) A  $p$  az eljárás indításakor a gyökérelemre hivatkozik. A rekurzív algoritmus első lépéseinek megfelelően feldolgozza ezt az elemet, majd balra lép.

(b) A rekurzív algoritmus ellenőrzi, hogy az elem nem  $\varnothing$ . Mivel nem az, így ismét feldolgozza az aktuális elemet, majd balra lép.



(c) A 12-es elemre érve, azt azonnal feldolgozza. Mivel már sem balra, sem jobbra nem tud továbblépni. Így visszatér a rekurzió előző szintjére.

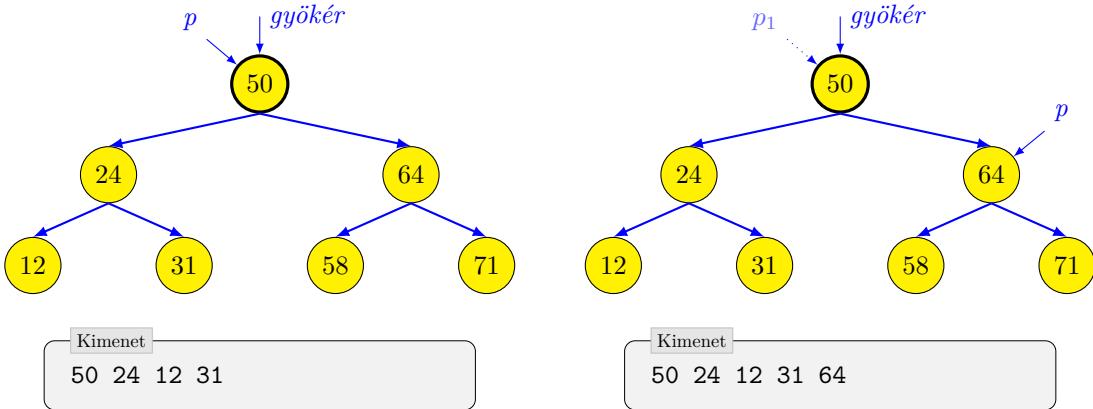
(d) Ezen a szinten a feldolgozás és a bal részfa bejárása már megtörtént. Emiatt továbblép a jobboldali részfára.



(e) A 31-es elem esetében is megtörténik a feldolgozás. Itt sincs lehetőség sem balra, sem pedig jobbra lépni. Ezért visszalépés az előző szintre.

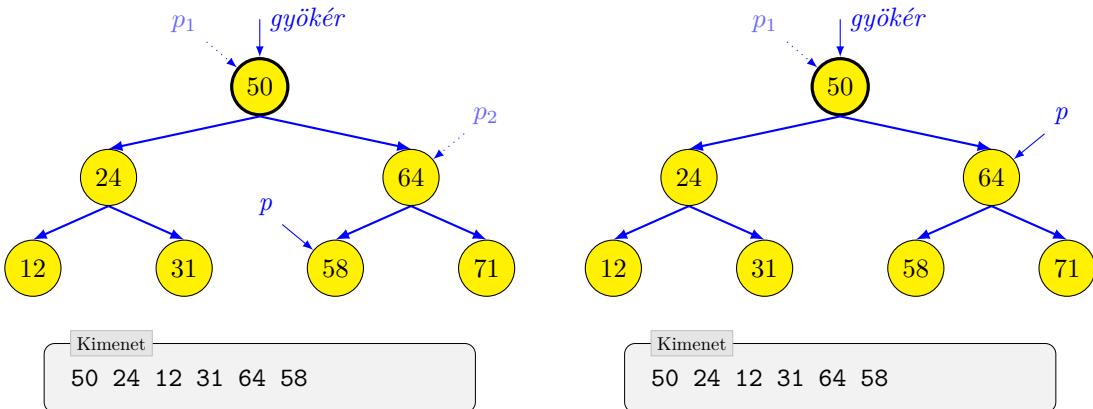
(f) Megtörtént a feldolgozás, baloldal és jobboldal bejárás, így innen is visszaébblép a rekurzió az előző szintre.

6.3. ábra. Bináris keresőfa preorder bejárása. A feldolgozás a csomópontok tartalmát kiírja a kimenetre.



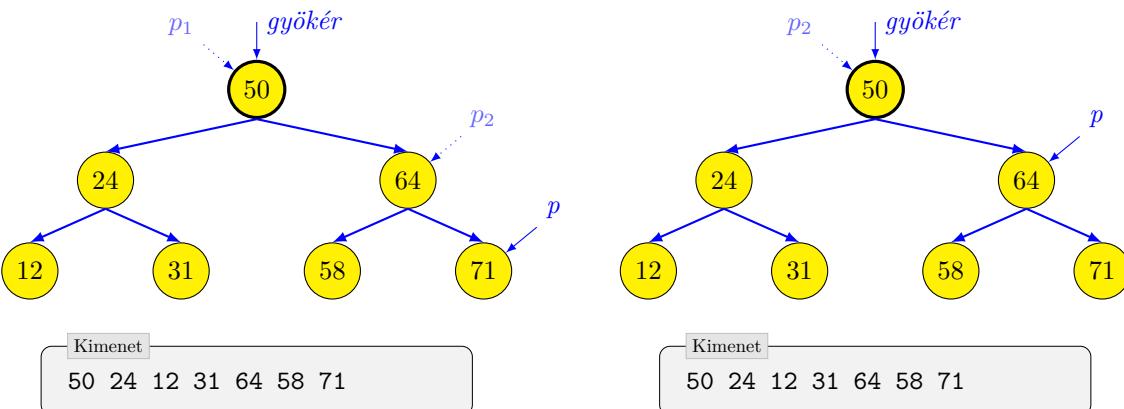
(g) A legfelső szinten a feldolgozás és a bal oldal bejárása már megtörtént. Ezt követi az a lépés, ahol az eljárás meghívja önmagát, paraméterként átadva a jobboldali részfát.

(h) A 64-es elem szintjén először megtörténik a feldolgozás, majd ellenőrzi a bal oldalt. Mivel itt vannak elemek, így azonnal tovább is lép erre.



(i) Az eljárás feldolgozza az 58-as elemet. Majd a bal részfáját, ami üres. Ezt követi a jobb oldal vizsgálata, ami szintén üres. Ezt követően a rekurzió visszábólép egy szinttel.

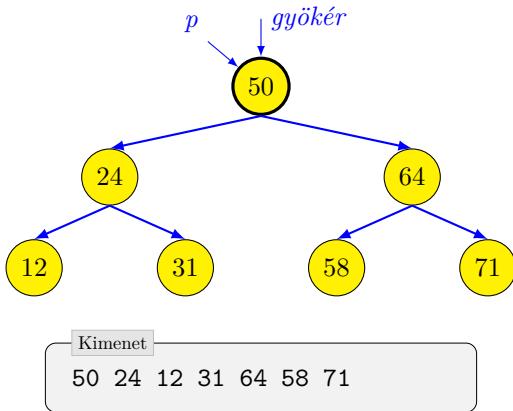
(j) A 64-es elem, és a baloldala már fel lett dolgozva, ezért az algoritmus meghívja önmagát a jobboldali részfára.



(k) Az eljárás feldolgozza az 71-es elemet. Ezt követi az elem bal részfája, majd a jobb oldal vizsgálata, ami szintén üres. Ezt követően a rekurzió visszábólép egy szinttel.

(l) Ezen a szinten már lefutott minden művelet, így a rekurzió egy szinttel visszalép.

6.3. ábra. Bináris keresőfa preorder bejárása. A feldolgozás a csomópontok tartalmát kiírja a kimenetre. (folytatás)



(m) A gyökérelemnél is megtörtént a bal és jobboldali részfa bejárása, illetve az elem feldolgozása.  
Ezzel az eljárás véget ért.

6.3. ábra. Bináris keresőfa pre-order bejárása. A feldolgozás a csomópontok tartalmát kiírja a kimenetre.  
(folytatás)

---

### 6.1. Algoritmus Részfa „preorder” bejárása

---

Bemenet:  $p$  -  $\text{FaElem} < \mathbf{T} >$  (a bejáradó részfa gyökere)

Bemenet:  $művelet$  -  $\text{eljárás} < \mathbf{T} >$  (az elemeken végrehajtandó művelet)

- 1: eljárás FAHALMAZ.RÉSZFABEJÁRÁSPREORDER( $p, művelet$ )
  - 2: ha  $p \neq \emptyset$  akkor
  - 3:      $művelet(p.tart)$
  - 4:     RÉSZFABEJÁRÁSPREORDER( $p.bal$ )
  - 5:     RÉSZFABEJÁRÁSPREORDER( $p.jobb$ )
  - 6: elágazás vége
  - 7: eljárás vége
-

1. csomópont baloldali részfájának feldolgozása,
2. csomópont tartalmának feldolgozása,
3. csomópont jobboldali részfájának feldolgozása.

Az eljárás itt is a *p* ellenőrzésével kezdődik, hiszen üres fa esetében elképzelhető, hogy már a legelső hívásnál is  $\emptyset$  értéket kaptunk paraméterként. Az eljárás hasonló módon hívja önmagát, először feldolgozza a baloldali részfában található elemeket. Miután végzett a részfával, utána dolgozza csak fel az aktuális csomópontban található tartalmat, majd ezt követően tér át a jobb oldali részfára.

A 6.4. ábrán is látható, hogy az algoritmus hasonlóan működik, ugyanaz a levélelemeknél is a  $\emptyset$  mutató szerepe, ez állítja meg a rekurzív hívások láncolatát. Érdemes megvizsgálni a feldolgozás sorrendjeként kapott számsort, bár itt jóval nyilvánvalóbb ennek gyakorlati haszna, tulajdonképpen tartalom szerint növekvő sorrendben kaptuk meg az elemeket. Jelen példánkban a fát úgy rendeztük, hogy minden csomópont baloldalán nála kisebb, jobb oldalán pedig nála nagyobb elemek vannak. Mivel az inorder bejárás alapelve az volt, hogy minden csomópont feldolgozása előtt végigjárja az ó teljes baloldali részfáját (tehát az összes nála kisebb elemet), majd pedig a csomópont feldolgozása után annak jobboldali részfáját (tehát az összes nála nagyobb elemet), így könnyen belátható, hogy miért lesz rendezett az így kapott lista.

#### Megjegyzés

Érdekes lehet a fában levő elemek tartalom szerint fordított sorrendjében való feldolgozása. Ehhez csak meg kell cserélnünk a bal- és jobboldali részfa meghívást, hiszen így minden csomópont feldolgozás előtt a nála nagyobb, utána pedig a nála kisebb elemeket fogja feldolgozni a bejárás.

---

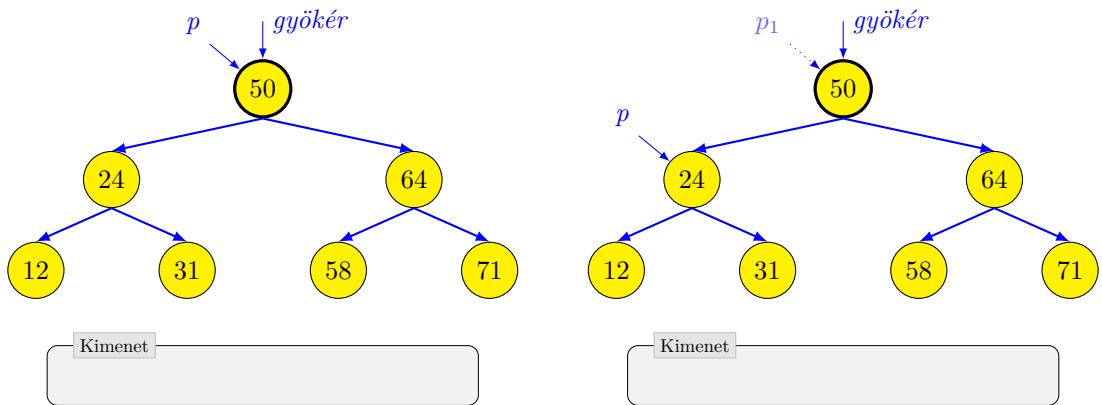
## 6.2. Algoritmus Részfa „inorder” bejárása

---

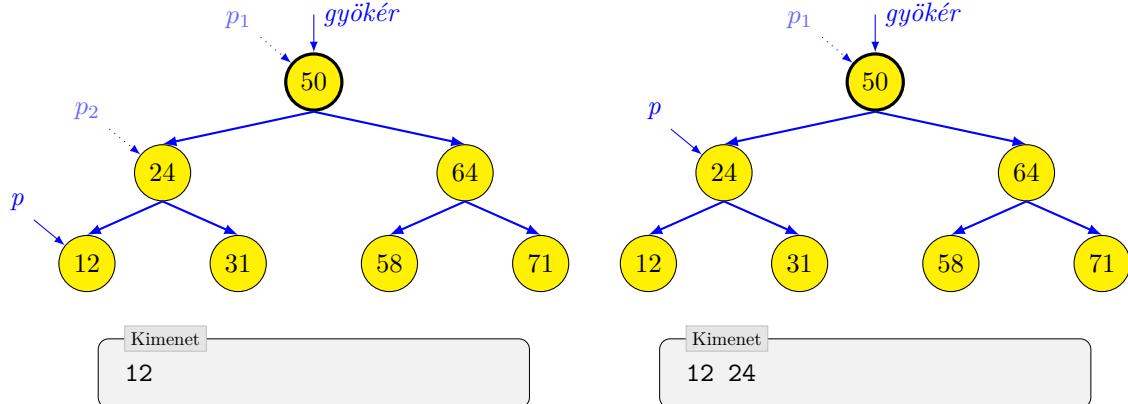
**Bemenet:** *p* -  $\tilde{F}$ aElem $<T>$  (*a* bejáradó részfa gyökere)

**Bemenet:** *művelet* - eljárás $<T>$  (*az elemeken végrehajtandó művelet*)

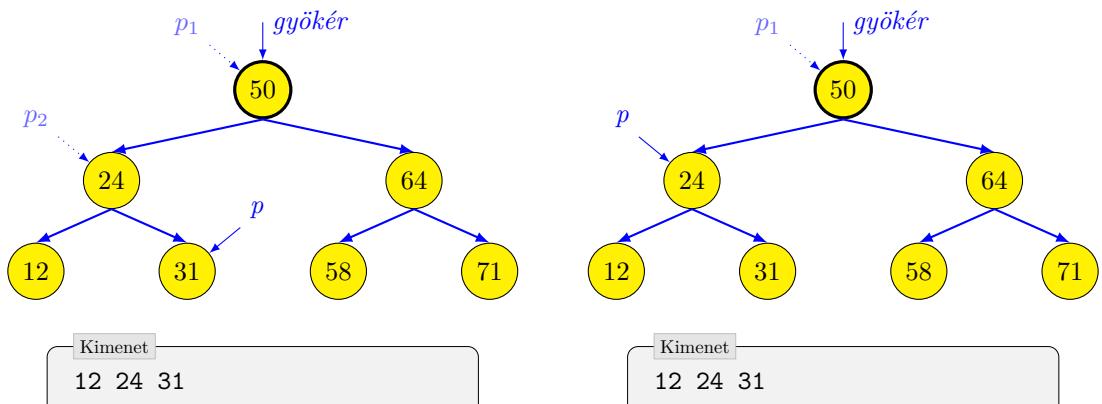
- 1: eljárás FAHALMAZ.RÉSZFABEJÁRÁSINORDER(*p, művelet*)
  - 2: ha *p*  $\neq \emptyset$  akkor
  - 3:     RÉSZFABEJÁRÁSINORDER(*p.bal*)
  - 4:     *művelet(p.tart)*
  - 5:     RÉSZFABEJÁRÁSINORDER(*p.jobb*)
  - 6:     elágazás vége
  - 7: eljárás vége
-



(a) A  $p$  az eljárás indításakor a gyökérelemre hivatkozik. A rekurzív algoritmus első lépéseinél megfelelően balra lép.

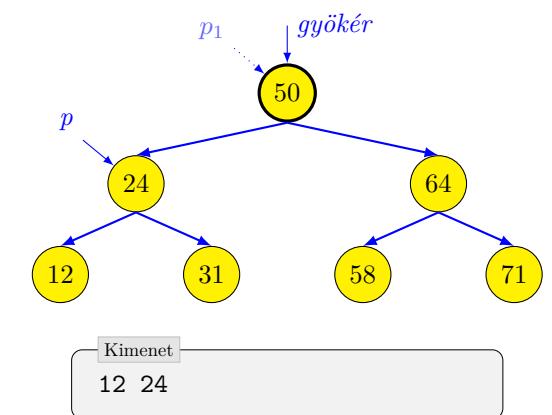


(c) Mivel balra már nem tud lépni, így feldolgozza a  $p$  által mutatott elemet. Ezt követően jobbra ellenőrzi a fát, de arra se tud továbbhaladni. Így visszalép a rekurzió előző szintjére.

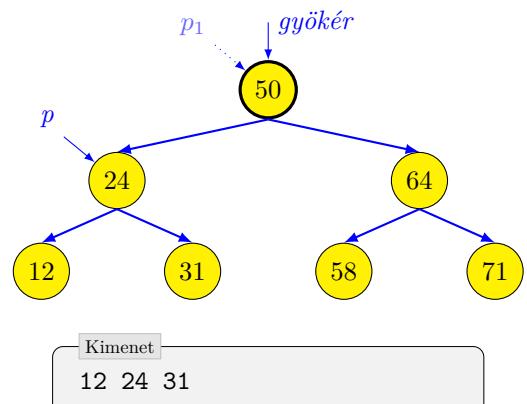


(e) A 31-es elem esetében sincs lehetőség balra lépni. Ezt követi a feldolgozás, a jobb oldal ellenőrzése, majd visszalépés az előző szintre.

(b) A rekurzív algoritmus ellenőrzi, hogy az elem nem  $\emptyset$ . Mivel nem az, így ismét azonnal balra lép.

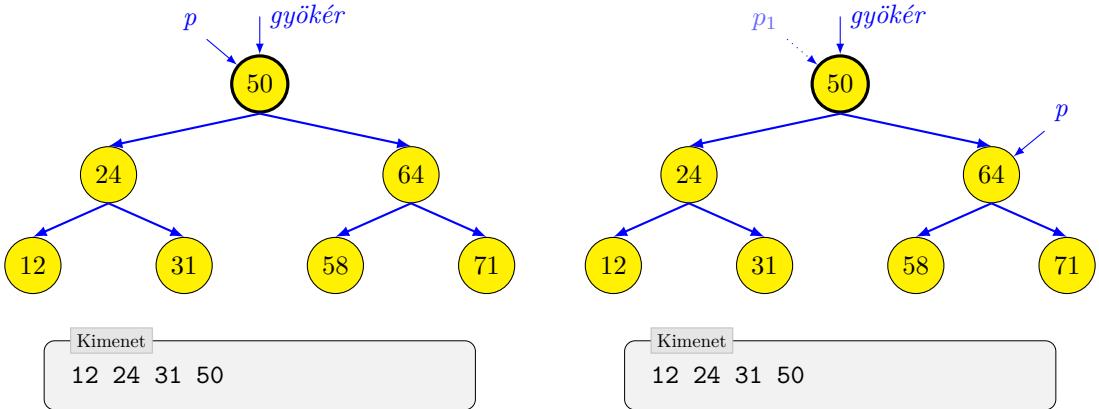


(d) Ezen a szinten a bal részfa feldolgozása már megtörtént. Emiatt feldolgozza az aktuális elemet, majd továbblép a jobboldali részfára.



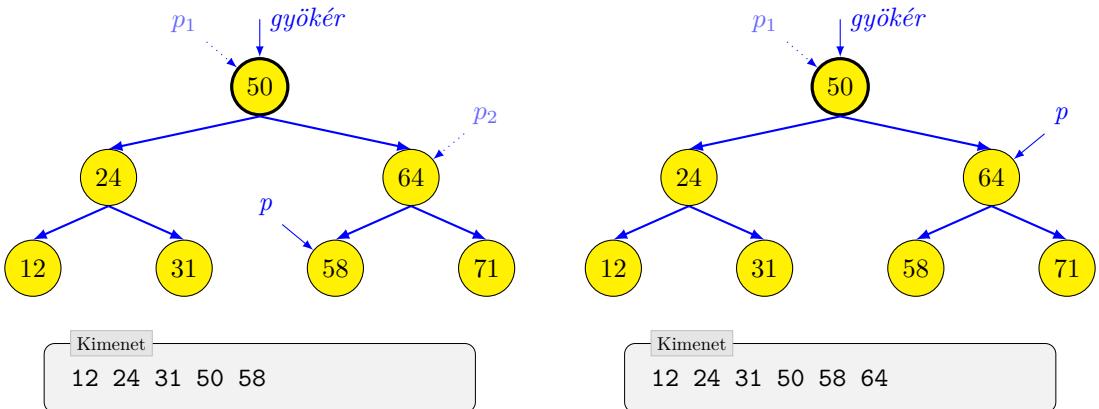
(f) Megtörtént a baloldal bejárása, a feldolgozás, a jobboldali bejárás, így innen is visszalép a rekurzió az előző szintre.

6.4. ábra. Bináris keresőfa inorder bejárása. A feldolgozás a csomópontok tartalmát kiírja a kimenetre.



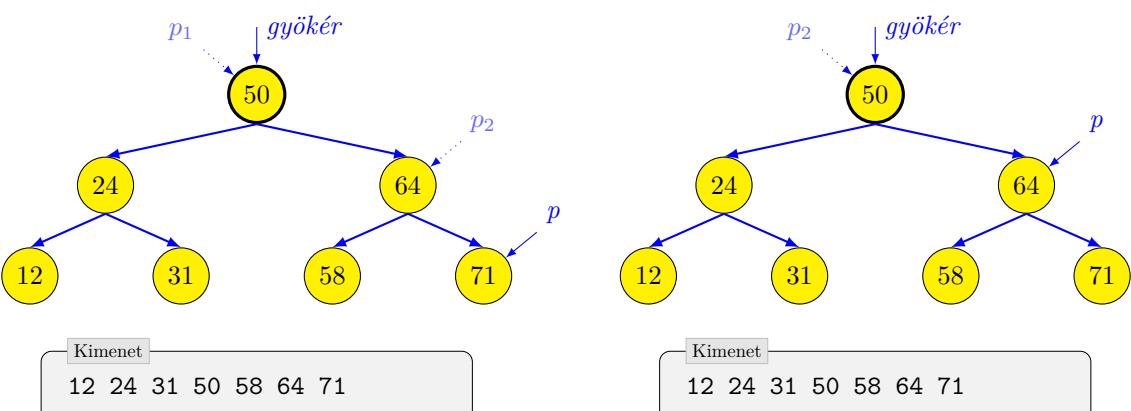
(g) A legfelső szinten a bal oldal bejárása már megtörtént. Ezt követi a feldolgozás, majd az eljárás meghívja önmagát, paraméterként átadva a jobboldali részfát.

(h) A 64-es elem szintén először ellenőrzi a bal oldalt. Mivel itt vannak elemek, így azonnal tovább is lép erre.



(i) Az eljárás feldolgozza az 58-as elem bal részfáját, ami üres. Ezt követi az 58 feldolgozása, majd a jobb oldal vizsgálata, ami szintén üres. Ezt követően a rekurzió visszabblép egy szinttel.

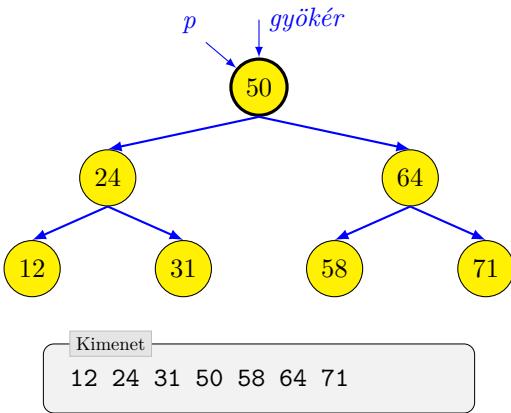
(j) A 64-es elem baloldala már fel lett dolgozva, ezt követi az elem feldolgozása, majd az algoritmus meghívja önmagát a jobboldali részfára.



(k) Az eljárás feldolgozza az 71-as elem bal részfáját, ami üres. Ezt követi a 71 feldolgozása, majd a jobb oldal vizsgálata, ami szintén üres. Ezt követően a rekurzió visszabblép egy szinttel.

(l) Ezen a szinten már lefutott minden művelet, így a rekurzió egy szinttel visszalép.

6.4. ábra. Bináris keresőfa inorder bejárása. A feldolgozás a csomópontok tartalmát kiírja a kimenetre. (folytatás)



(m) A gyökerelemnél is megtörtént a bal és jobboldali részfa bejárása, illetve az elem feldolgozása.  
Ezzel az eljárást véget ért.

6.4. ábra. Bináris keresőfa inorder bejárása. A feldolgozás a csomópontok tartalmát kiírja a kimenetre.  
(folytatás)

### Postorder bejárás

A *postorder* bejárás során is ugyanazt a három műveletet végezzük el minden egyes csomópontban, azonban a tartalom feldolgozása a sorrendben az utolsó helyre kerül (6.3. algoritmus). Ennek megfelelően minden elem esetében az alábbi lépések történnek meg:

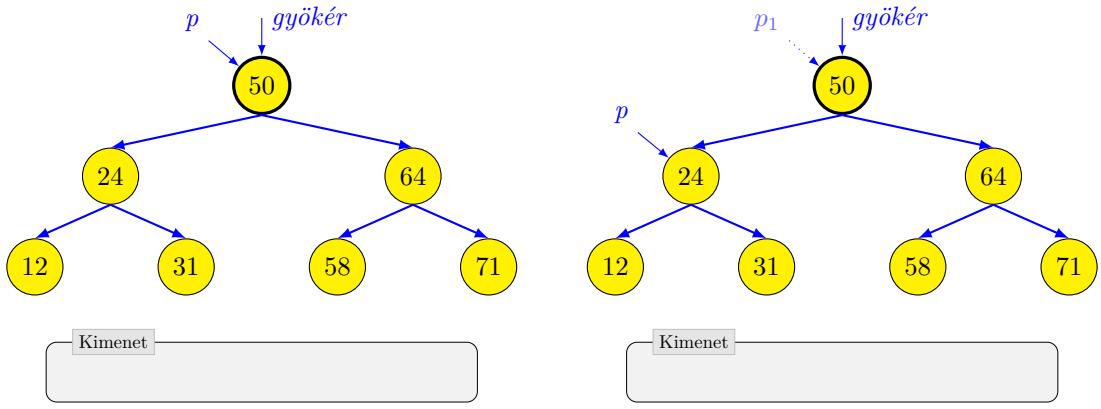
1. csomópont baloldali részfájának feldolgozása,
2. csomópont jobboldali részfájának feldolgozása,
3. csomópont tartalmának feldolgozása.

Egy lehetséges megvalósítást mutat a 6.3. algoritmus. Kezdetben ez is ellenőrzi, hogy a *p* változó értéke nem  $\emptyset$ , aminek köszönhetően a rekurzió megáll a levélelemek alatt, illetve ugyanennek köszönhetően működik az algoritmus üres fa esetében is.

A 6.5. ábra mutat egy példát a postorder bejárásra. Az algoritmus alapja, hogy először minden bejárja a két részfát, majd csak ezt követően dolgozza fel az aktuális elem tartalmát. Ennek köszönhetően a teljes fa felszabadításakor jól használható, hiszen a feldolgozás minden levélelemen fog lefutni. Bár a bináris keresőfából való törlést csak később tárgyaljuk, de sejthető, hogy a leveleket lesz a legegyszerűbb kitörölni (az ábrán is látható, hogy a 12 és a 31 feldolgozása után következik a 24 feldolgozása (ez ugyan alapesetben nem levél, de ha a 12 és 31 elemeket már töröltük, akkor már ez is levél lesz)).

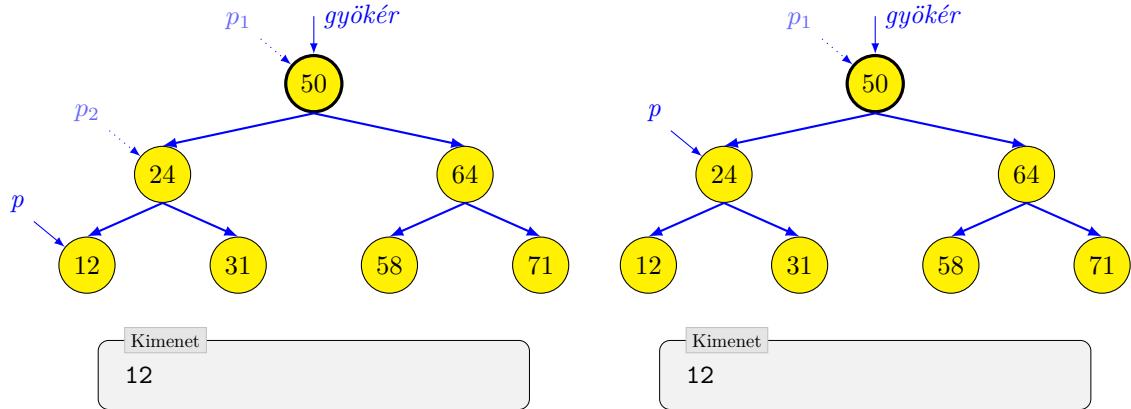
#### Megjegyzés

Érdemes lehet megfigyelni, hogy minden bejárás esetében a rekurzió ugyanolyan sorrendben érintette az elemeket. Ha a feldolgozástól függetlenül csak a *p* mutató útját figyeljük, akkor látható, hogy a pre-, in- és postorder bejárások során is ugyanazt az utat járta be, a különbség pusztán annyi, hogy maga a feldolgozás mikor történt: amikor rálépett az elemre (preorder), amikor elhagyta az elemet (postorder) vagy a két részfa feldolgozása közben (inorder).



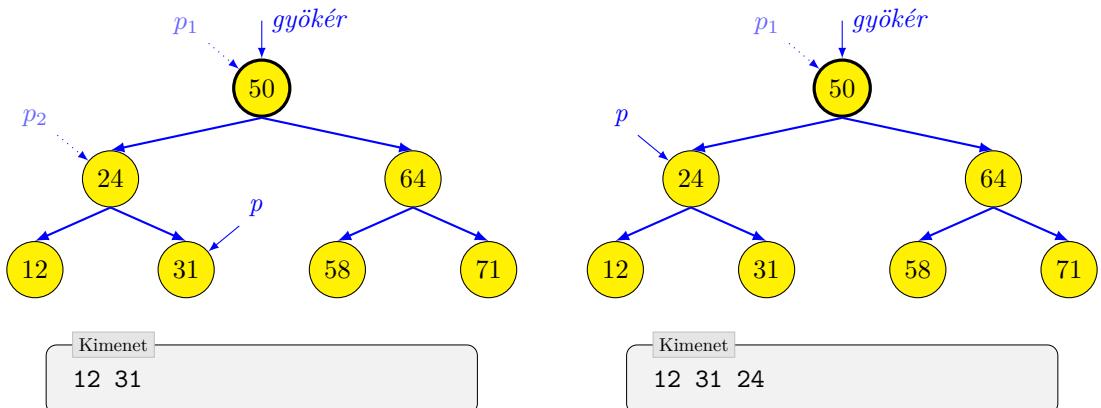
(a) A  $p$  az eljárás indításakor a gyökérelemre hivatkozik. A rekurzív algoritmus első lépésein megfelelően először balra lép.

(b) A rekurzív algoritmus ellenőrzi, hogy az elem nem  $\emptyset$ . Mivel nem az, így ismét balra lép.



(c) A 12-es elemre érve, mivel már sem balra, sem jobbra nem tud továbblépni, feldolgozza az elemet. Ezt követően visszatér a rekurzió előző szintjére.

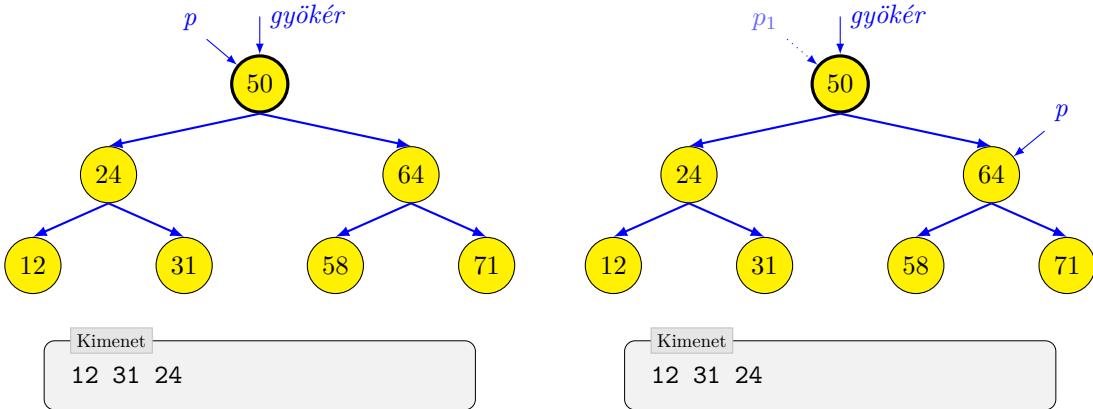
(d) A 24-re visszalépve azonnal megkezdi a jobboldali részfa feldolgozását.



(e) A 31-es elem esetében már nincs lehetőség sem balra, sem pedig jobbra lépni. Ezért ezt az elemet is feldolgozza, majd visszalépés az előző szintre.

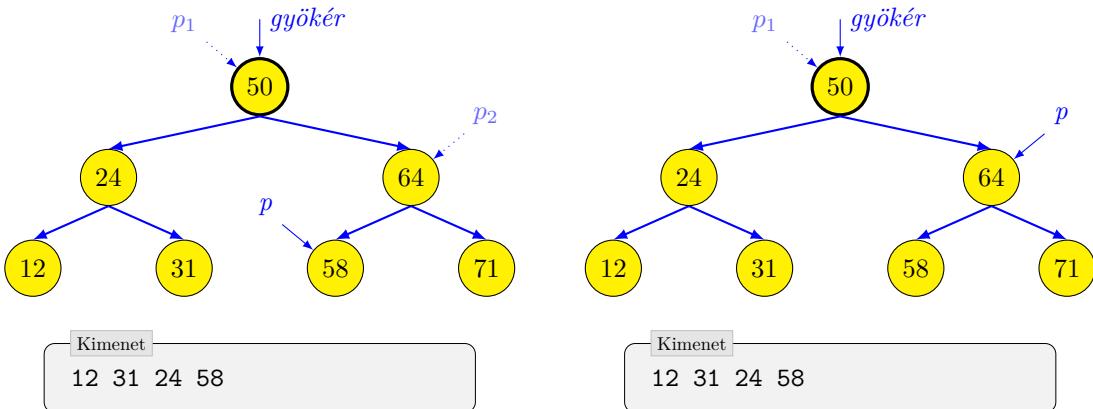
(f) Megtörtént a baloldal és jobboldal bejárása, így itt is megtörténik a feldolgozás, majd innen is visszalép a rekurzió az előző szintre.

6.5. ábra. Bináris keresőfa postorder bejárása. A feldolgozás a csomópontok tartalmát kiírja a kimenetre.



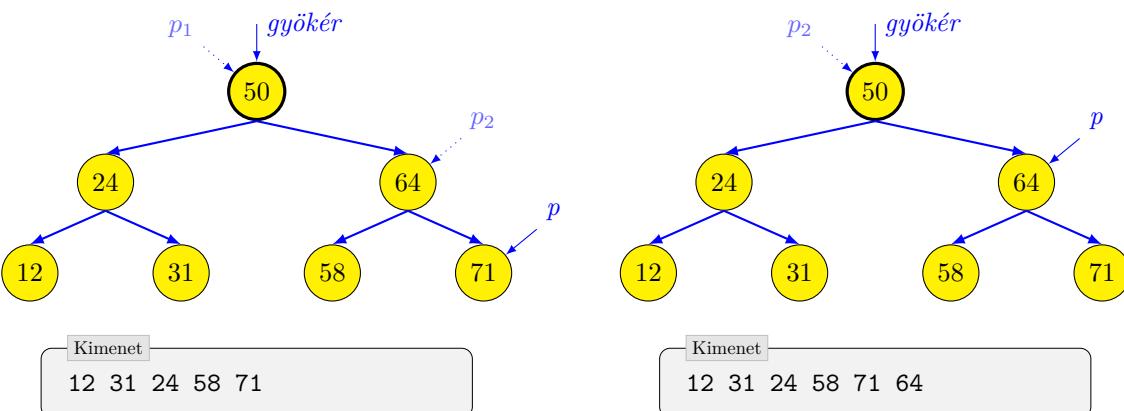
(g) A legfelső szinten a bal oldal bejárása már megtörtént. Ezt követi az a lépés, ahol az eljárás meghívja önmagát, paraméterként átadva a jobboldali részfát.

(h) A 64-es elem szintjén először ellenőrzi a bal oldalt. Mivel itt vannak elemek, így azonnal tovább is lép erre.



(i) Az eljárás ellenőrzi a bal részfáját, ami üres. Ezt követi a jobb oldal vizsgálata, ami szintén üres. Ezt követően az 58-as elemet feldolgozza, majd a rekurzió visszabblép egy szinttel.

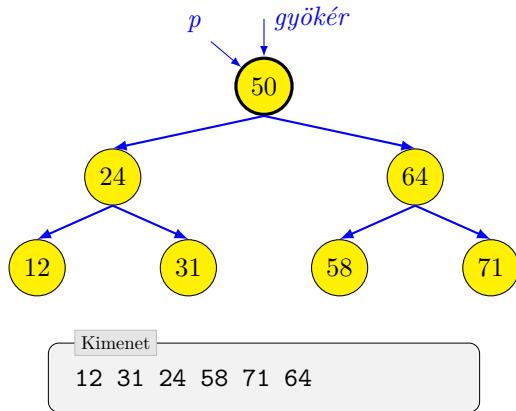
(j) A 64-es elem baloldala már fel lett dolgozva, ezért az algoritmus meghívja önmagát a jobboldali részfára.



(k) Az eljárás ellenőrzi az 71-es elemet. Első lépés a bal részfája, majd a jobb oldal vizsgálata, ami szintén üres. Végül feldolgozza az elemet, majd a rekurzió visszabblép egy szinttel.

(l) Ezen a szinten már lefutott a két részfa bejárás, ezért most jön a 64 feldolgozása. Ezután a rekurzió egy szinttel visszalép.

6.5. ábra. Bináris keresőfa postorder bejárása. A feldolgozás a csomópontok tartalmát kiírja a kimenetre. (folytatás)



(m) A gyökérelemnél is megtörtént a bal és jobboldali részfa bejárása, ezért most következik az elem feldolgozása. Ezzel az eljárás véget ért.

6.5. ábra. Bináris keresőfa postorder bejárása. A feldolgozás a csomópontok tartalmát kiírja a kimenetre.  
(folytatás)

---

### 6.3. Algoritmus Részfa „postorder” bejárása

---

Bemenet:  $p$  -  $\tilde{F}aELEM<T>$  (a bejáradó részfa gyökere)

Bemenet:  $művelet$  -  $eljárás<T>$  (az elemeken végrehajtandó művelet)

- 1:  $eljárás \text{ FAHALMAZ.RÉSZFABEJÁRÁSPOSTORDER}(p, művelet)$
  - 2:    ha  $p \neq \emptyset$  akkor
  - 3:      $RÉSZFABEJÁRÁSPOSTORDER(p.bal)$
  - 4:      $RÉSZFABEJÁRÁSPOSTORDER(p.jobb)$
  - 5:      $művelet(p.tart)$
  - 6:    elágazás vége
  - 7:  $eljárás$  vége
-

## Rekurzív algoritmusok meghívása

A fenti műveletek minden egy-egy részfa feldolgozására voltak alkalmasak. A bejárás során azonban azt várjuk az adatszerkezettől, hogy az minden adatot dolgozzon fel. Ezért az adatszerkezet publikus **BEJÁR** bejár metódusa a fent említett részfa bejárások közül meghívja valamelyiket, paraméterként átadva a **gyökér** mutató értékét.

Rekurzív algoritmusoknál gyakran van szükség egy ilyen kiegészítő műveletre, amit „belépési pontként is felfoghatunk”. Eleve nem lenne elegáns, ha a bejárásnak minden át kellene adni a hívónak a **gyökér** értékét, másrészt egy megfelelően implementált megvalósítás során a halmazt használó külső kód nem is fér hozzá ehhez a mezőhöz.

A 6.4. algoritmus a már megismert preorder bejárást használja, de értelemszerűen módosítható az eljárás bármelyik másik bejárási mód használatára.

---

### 6.4. Algoritmus Bináris keresőfával megvalósított halmaz bejárása (preorder módon)

---

**Bemenet:** *művelet - eljárás<T>* (az elemeken végrehajtandó művelet)

- 1: **eljárás FAHALMAZ.BEJÁR(*művelet*)**
  - 2:   **RÉSZFABEJÁRÁSPREORDER(*gyökér, művelet*)**
  - 3: **eljárás vége**
- 

#### Felhasznált változók és függvények

- **gyökér** : Hivatkozás a fa gyökerére.
- 

### 6.2.3. Elem meglétének vizsgálata

Ezt a műveletet is rekurzív algoritmussal lehet egyszerűen megvalósítani. Készítünk egy függvényt, ami meg tudja adni, hogy megadott részfa tartalmaz-e egy paraméterként megadott **T** típusú értéket (tehát van-e a részfának olyan eleme, amelyiknek a tartalma egyenlő ezzel az értékkel). Mivel a keresőfa a tartalom alapján lett rendezve, ezt hatékonyan meg tudjuk mondani.

A rekurzió az alábbiak alapján épül fel:

- 1. triviális eset: üres fában nem lehet benne a keresett elem. Ezért, ha a gyökerelemként  $\emptyset$  értéket kaptunk, akkor jelezhetjük, hogy nincs a keresett érték a fában.
- 2. triviális eset: ha nem üres a részfa, akkor közvetlenül elérjük, így megvizsgáljuk annak gyökerelemét. Amennyiben az a keresett értéket tartalmazza, akkor szintén végeztünk, hiszen megtaláltuk a keresett elemet.
- Általános eset: ha nem üres a fa, viszont a gyökérben nem a keresett érték található, akkor még minden lehetséges, hogy valamelyik részfában megtalálható a keresett érték. Ehhez rekurzívan folytatjuk a keresést a bal-, illetve a jobboldali részfában a keresett értéktől függően. Mivel a fa rendezett, ezért a gyökerelem tartalma a továbblépés irányára egyértelműen meghatározható. Ha a gyökérnél kisebb elemet keresünk, akkor a baloldali, ha pedig nagyobb elemet keresünk, akkor a jobboldali részfában folytatjuk a műveletet.

Mivel véges számú elemet tárolunk a fában, ezért előbb-utóbb a fenti lépések biztosan eredményhez fognak vezetni. Vagy megtaláljuk a keresett elemet, vagy eljutunk a fa valamely levélelemének lezáró  $\emptyset$  mutatójához.

A keresés algoritmusá (6.5. algoritmus) némileg hasonlít a bejárásokhoz, a lényeges különbség az, hogy a **p** nem mindenkor előre rögzített útvonalat jár be, hanem mindenkor a keresett értéknek megfelelően döntünk, hogy az egyes csomópontokból milyen irányba lépjen tovább.

Az algoritmus 2. sorában itt is megvizsgáljuk a **p** értékét, hogy az egy valódi elemre hivatkozik-e. Ha nem, akkor ez azt jelenti, hogy üres a részfa, ez volt az egyik triviális esetünk. Ilyenkor jelezhetjük, hogy nincs benne a keresett elem a részfában.

Ezt követi a részfa gyökerében lévő tartalom ellenőrzése, először megvizsgáljuk, hogy a **p** által aktuálisan hivatkozott csomópont tartalma kisebb-e mint a keresett. Amennyiben igen, akkor a rendezettségnak megfelelően a keresett elem a jobboldali részfában kell, hogy legyen, így a függvény meghívja önmagát ebbe az irányba. Elképzelhető persze, hogy nincsenek gyerekei ezen az ágon, ilyenkor maga a meghívás

még megtörténik, azonban a  $p$  értéke  $\emptyset$  lesz, így az előző bekezdésben leírtaknak megfelelően a rekurzió a következő lépéssel véget ér.

Amennyiben az előző feltétel nem teljesült, akkor megvizsgáljuk, hogy az aktuálisan vizsgált elem tartalma nagyobb-e, mint a keresett. Ha igen, akkor ez utóbbi a baloldali részfában fog elhelyezkedni (ha egyáltalán van ilyen), így a függvény ismét meghívja önmagát, paraméterként átadva a baloldali gyerek hivatkozását.

Amennyiben a program futása a 9. sorhoz jut, akkor tudjuk, hogy a  $p$  egy létező bináris fa elemre mutat, amelyiknek a tartalma nem is kisebb, illetve nem is nagyobb, mint a keresett elem, tehát megtaláltuk azt. A függvény visszatérési értéke tehát az lesz, hogy a keresett elem megtalálható a részfában.

Érdemes azonban figyelembe venni, hogy ez a visszatérési érték csak az előző rekurziós szintre kerül visszaadásra, innentől minden szintnek biztosítania kell, hogy a megtalált eredmény továbbítódjon az ōt hívó szint felé. Emiatt lényeges a 4. és 7. sorban látható módszer, ahol a rekurzívan meghívott függvénytől kapott visszatérési értéket adja vissza az aktuális szint is az ōt hívónak.

---

## 6.5. Algoritmus Bináris keresőfában keresés rekurzív megvalósítása

---

**Bemenet:**  $p$  -  $\tilde{F}$ aElem< $T$ > (*hivatkozás a vizsgálandó részfa gyökerére*)

**Bemenet:** érték -  $T$  (*a keresett érték*)

**Kimenet:** . - logikai (*a keresett elem benne van-e a  $p$  gyökerű részfában?*)

**Megszorítás:**  $T$  összehasonlítható

```
1: függvény FAHALMAZ.RÉSZFAELEME(p , érték)
2: ha $p \neq \emptyset$ akkor
3: ha $p.tart > \text{érték}$ akkor
4: vissza RÉSZFAELEME($p.bal$, érték)
5: különben
6: ha $p.tart < \text{érték}$ akkor
7: vissza RÉSZFAELEME($p.jobb$, érték)
8: különben
9: vissza igaz
10: elágazás vége
11: elágazás vége
12: különben
13: vissza hamis
14: elágazás vége
15: függvény vége
```

---

A keresés lépésszáma ideális esetben  $\mathcal{O}(\log_2 n)$ , de ez sajnos nem lesz minden esetben igaz. A bejárt út hosszát ugyanis jelentősen befolyásolja a fa szerkezete. Amennyiben ez egy, a 6.6. ábrán is látható „kiegyensúlyozott fa”, akkor az  $\mathcal{O}(\log_2 n)$ -es lépésszám valóban igaz. Nem kiegyensúlyozott esetekben ez azonban jóval több lépést igényel, pl. legrosszabb esetben, ha a fa minden csúcsának csak egy gyereke van, akkor ez a lineáris kereséshez hasonlóan  $\mathcal{O}(n)$  lesz (a fa ilyenkor sokkal inkább hasonlít egy láncolt listára, így a keresés lépésszáma is annak megfelelő lesz).

Mivel a gyökérelem nem érhető el az osztályon kívülről, ezért itt is készítünk egy, az interfész kielégítő függvényt, ezt mutatja a 6.6. ábra.

---

## 6.6. Algoritmus Bináris keresőfával megvalósított halmaz ELEME művelete

---

**Bemenet:** érték -  $T$  (*a keresett érték*)

**Kimenet:** . - logikai (*a keresett elem benne van-e teljes fában?*)

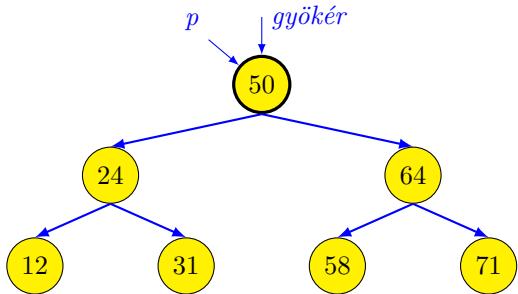
**Megszorítás:**  $T$  összehasonlítható

```
1: függvény FAHALMAZ.ELEME(érték)
2: vissza RÉSZFAELEME(gyökér, érték)
3: függvény vége
```

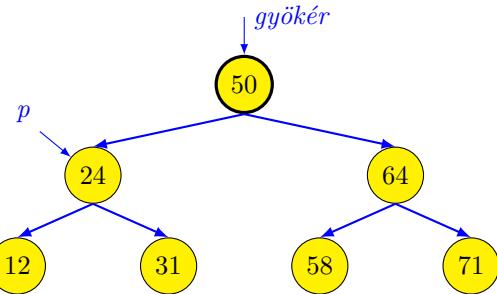
---

### Felhasznált változók és függvények

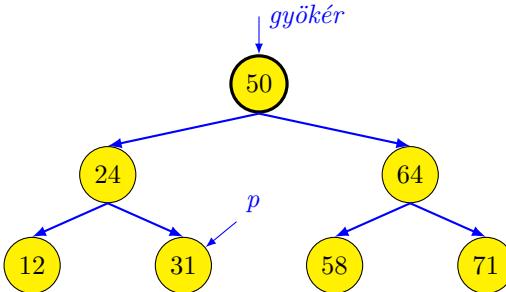
- *gyökér* : A halmaz elemeit tartalmazó fa gyökere.
-



(a) A  $p$  az eljárás indításakor a gyökérelemre hivatkozik. A keresett érték kisebb, mint 50, emiatt a függvény meghívja önmagát a baloldali részfára.



(b) A következő híváskor a  $p$  a 24-es elemre hivatkozik. Mivel ez kisebb a keresetnél, emiatt újra hívja magát a jobboldali részfára.



(c) A  $p$  által mutatott elem tartalma a keresett elem. Emiatt nem lép mélyebb szintre, hanem visszaadja ezt az értéket (a rekurzió visszatérését már nem tartalmazza az ábra).

6.6. ábra. Adott tartalmú elem keresése bináris keresőfában (a keresett érték a 31).

#### 6.2.4. Új elem felvétele

A bináris keresőfa esetében is megvizsgáljuk az új elem elhelyezésének lehetőségét. A láncolt lista esetében erre számos megoldást áttekintettünk (elejére, végére, közepére, rendezetten beszúrás), ezekre a bináris keresőfa esetében nincs szükség, hiszen a fában lévő elemeket minden rendezetten szeretnénk eltárolni. Ez persze nem jelenti azt, hogy ne lehetne többféle beszűró algoritmust kidolgozni, de a mi esetünkben olyan megoldást keresünk, amelyik az új elemek beszúrásakor nem módosítja a már meglévő elemek egymáshoz viszonyított helyzetét, hanem csak az új elemet hozzáláncolja valamelyik már meglévő elem *bal* vagy *jobb* hivatkozásához. Figyelembe véve, hogy az elemeket rendezetten szeretnénk elhelyezni, ez egyben egyértelműen meg is határozza az elem leendő helyét.

A beszúrásnál is látható a működés rekurzív jellege:

- Triviális eset: üres fába szeretnénk beszúrni az új elemet, ilyenkor az új elem lesz a fa első és egyben egyetlen eleme.
- Általános eset: nem üres fa esetén a rendezettség alapján meghívja önmagát a függvény a gyökér-elem bal-, illetve jobboldali részfájára. Ha a gyökérnél kisebb elemet akarunk felvenni, akkor balra, egyébként pedig jobbra.

A pszeudokóban külön esetként kezeljük azt, amikor a beszúrandó tartalom már megtalálható a fában. Feladattól függően erre többféle módon is lehetne reagálni, mi itt egy halmazt építünk, ezért egyszerűen figyelmen kívül hagyjuk a másodjára beszúrt elemet. A halmaz jellegzetessége ugyanis az, hogy egy elem csak egyszer szerepelhet benne.

A tényleges implementációban jól látható, hogy a fába való beszúrás két fő lépésből áll: meg kell keresni az új elem leendő helyét, majd ezt az elemet hozzá kell kapcsolni a fa eddigi csomópontjaihoz.

A 6.7. algoritmus 6–9. sorai már ismerősek lehetnek a keresésből. Itt keressük meg a beszúrandó elem helyét. Az új elem helyének megállapítása, és a keresés egymással nyilván szorosan összefügg, hiszen az új elemet olyan helyre kell elhelyeznünk, ahol az előzőleg megismert keresés majd meg fogja találni. Emiatt itt is ugyanazokat a lépéseket hajtjuk végre, attól függően, hogy az aktuálisan vizsgált elem tartalma milyen viszonyban áll a beszúrandó értékkel.

- Ha a *p* által hivatkozott csomópontban található tartalom nagyobb, mint a beszúrandó érték, akkor az új elemet az aktuálisan vizsgált elem baloldali részfájába kell majd elhelyezni (hiszen a keresés is ebbe az irányba fog majd haladni, amikor ezt vissza akarjuk olvasni). Ezért az eljárás rekurzívan meghívja önmagát, paraméterként pedig a *p* baloldali gyerekét adja át.
- Ha a *p* által hivatkozott csomópontban található tartalom kisebb, mint a beszúrandó érték, akkor az új elemet az aktuálisan vizsgált elem jobboldali részfájába kell majd elhelyezni. Ezért az eljárás rekurzívan meghívja önmagát, paraméterként pedig a *p* jobboldali gyerekét adja át.
- Külön nem kezeljük azt az esetet, amikor az aktuális elem tartalma egyenlő azzal, mint amit be szeretnénk szűrni. Itt véget ér a rekurzió a fa módosítása nélkül. Elképzelhető olyan megvalósítás is, ahol hibát jelzünk ebben az esetben.

A fenti három lehetőség persze csak akkor értelmezhető, ha a *p* egy valódi elemre mutat a fában, amelyiknek tudjuk vizsgálni a tartalmát. Amennyiben a *p* változó értéke  $\emptyset$ , az két dolgot jelenthet: a rekurzió előző szintjén olyan irányba próbáltunk meg továbblépni, amerre már nincsenek elemek; vagy eleve üres volt a fa, és már a legelső híváskor is  $\emptyset$  lett a paraméter értéke. Egyik se jelent hibát, hiszen éppen egy ilyen helyet kerestünk, ami üres, és a rendezettség szempontjából megfelel az új elemnek. Ilyenkor létrehozzuk az új elemet és feltöljük a mezőit a szükséges értékekkel.

Hogy megértsük magát a beláncolási műveletet, fontos észrevennünk, hogy a rekurziós hívások során a visszatérési értékkal minden felülírjuk annak a mezőnek az értékét, amelyet áadtunk paraméterként (tehát amelyik azt mutatja, hogy milyen irányba léptünk tovább a fában). Ezzel lehetőségünk nyílik arra, hogy az új elem felvételekor módosítsuk az új szülője rá mutató hivatkozását, illetve a legelső elem beszúrása esetén a *gyökér* változó értékét.

Két esetet célszerű megkülönböztetni:

- Amennyiben egy üres részfára hívódott meg a függvény, akkor a 4. sorban látható módon az *új* változó értékét adjuk vissza paraméterként. Ennek köszönhető, hogy megtörténik az új elem beláncolása, hiszen üres részfához két esetben juthatunk:

- Egy már létező fában eljutottunk egy alsó elemig, és az utolsó rekurziós hívás ennek valamelyik  $\emptyset$  értékű gyerekmutatóját adtuk át paraméterként. A hívás visszatérési értéke az új elemre való hivatkozás lesz, tehát a 7. sorban vagy a 10. sorban látható értékkadás felülírja az új szülő eddigi  $\emptyset$  értékű mezőjét az új elemre való hivatkozásra. Ezzel ezt az új csúcsot beláncoltuk a szerkezetbe.
- Ha a teljes fa üres volt (tehát a *gyökér* értéke  $\emptyset$ ), akkor már az első hívás során igaz lesz a  $p = \emptyset$  feltétel, tehát azonnal létrehozunk egy új elemet. Egy erre való hivatkozás lesz a függvény visszatérési értéke, amit a 6.8. algoritmus egyetlen sora értékül ad az eddigi üres *gyökér* változónak. Ezzel ismét elvégeztük a beláncolást.
- Ha megvizsgáljuk a  $p = \emptyset$  feltétel különben ágát is, akkor látható, hogy a visszatérési érték itt minden  $p$ . Tehát ugyanazt az értéket adjuk vissza, amit paraméterként kaptunk. Ennek hatására azokon a rekurziós hívási lépésekben, amikor nem történt beszúrás (csak keresés), ott ugyanazt az értéket adjuk értékül a gyökérnek, illetve a keresés során érintett elemek gyerek mutatóiba, ami már ezelőtt is ott volt.

A 6.7. ábrán is megpróbáljuk kiemelni ezt a jellegzetességet. A  $p$  címke helyzete azt próbálja jelezni, hogy a rekurzív hívásból való visszatérés során a csomópont bal vagy jobb gyerek mutatója (vagy éppen a gyökér) lesz felülírva a visszatérési értékkel.

---

### 6.7. Algoritmus Részfába új elem beszúrása

---

**Bemenet:**  $p$  -  $\tilde{F}aElem^<T>$  (*annak a részfának a gyökere, aholá be szeretnénk szűrni*)

**Bemenet:** érték -  $T$  (*a beszúrandó érték*)

**Kimenet:**  $. - \tilde{F}aElem^<T>$  (*a paraméterként átadott p (módosított) értéke*)

**Megszorítás:**  $T$  összehasonlítható

- 1: függvény FAHALMAZ.RÉSZFÁBABESZÚR( $p$ , érték)
  - 2:   ha  $p = \emptyset$  akkor
  - 3:     új  $\leftarrow$  LÉTREHOZ( $FaElem^< T >$ ) { $tart \leftarrow$  érték,  $bal \leftarrow \emptyset$ ,  $jobb \leftarrow \emptyset$ }
  - 4:     vissza új
  - 5: különben
  - 6:   ha  $p.tart >$  érték akkor
  - 7:      $p.bal \leftarrow$  RÉSZFÁBABESZÚR( $p.bal$ , érték)
  - 8:   különben
  - 9:     ha  $p.tart <$  érték akkor
  - 10:       $p.jobb \leftarrow$  RÉSZFÁBABESZÚR( $p.jobb$ , érték)
  - 11:     elágazás vége
  - 12:     elágazás vége
  - 13:     vissza  $p$
  - 14: elágazás vége
  - 15: függvény vége
- 

#### Felhasznált változók és függvények

- *új* : Egy  $\tilde{F}aElem^< T >$  típusú segédváltozó.
  - *LÉTREHOZ( $FaElem^< T >$ )* : Létrehoz egy új csúcsot, és visszaad egy hivatkozást erre.
- 

---

### 6.8. Algoritmus Bináris keresőfával megvalósított halmaz beszúrás művelete

---

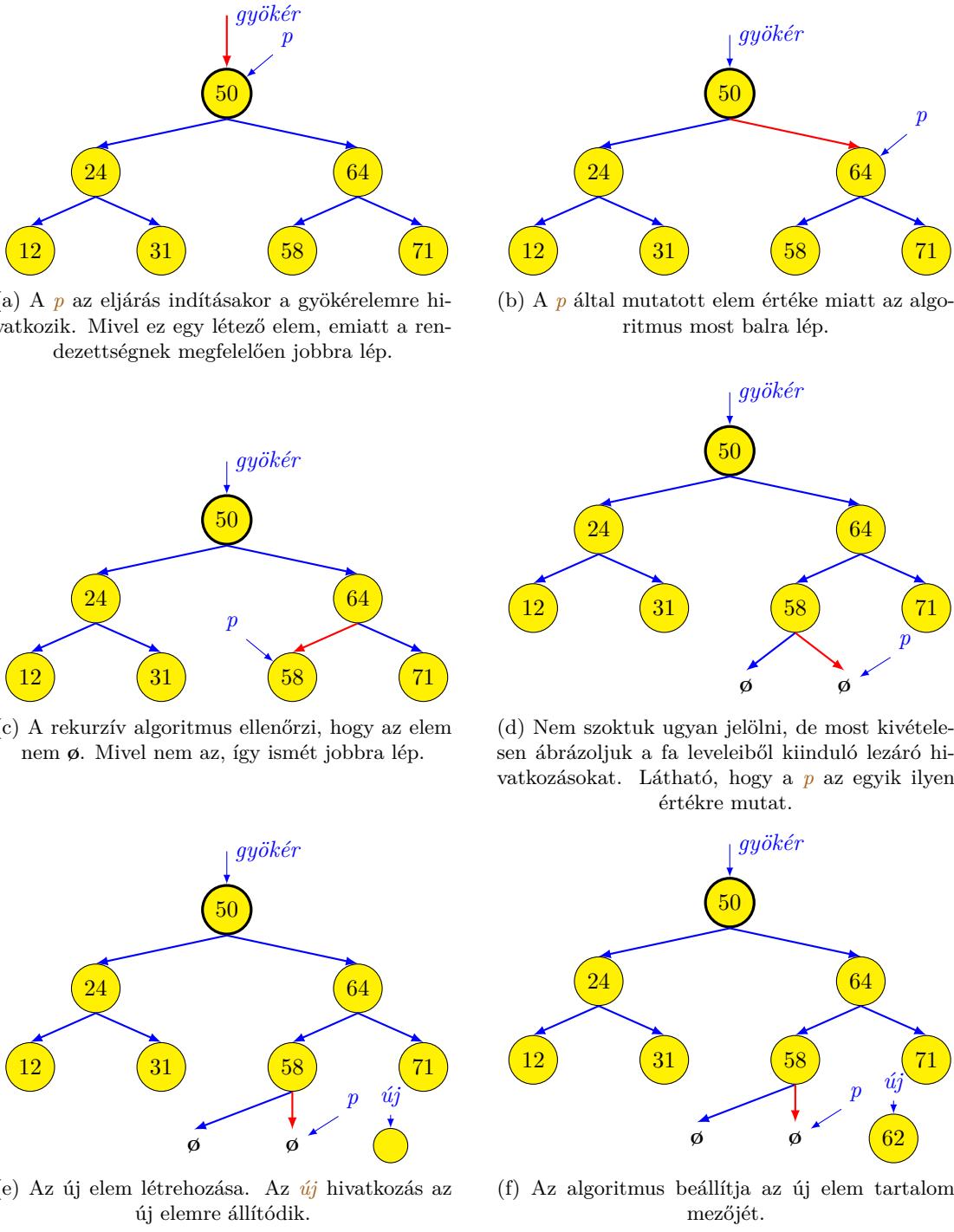
**Bemenet:** érték -  $T$  (*a beszúrandó érték*)

**Megszorítás:**  $T$  összehasonlítható

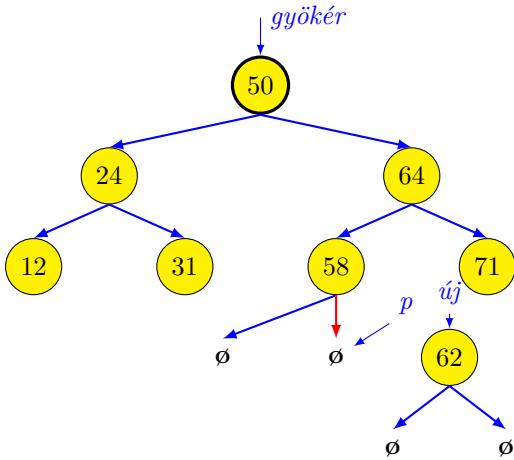
- 1: eljárás FAHALMAZ.BESZÚR(érték)
  - 2:    $gyökér \leftarrow$  RÉSZFÁBABESZÚR( $gyökér$ , érték)
  - 3: eljárás vége
- 

#### Felhasznált változók és függvények

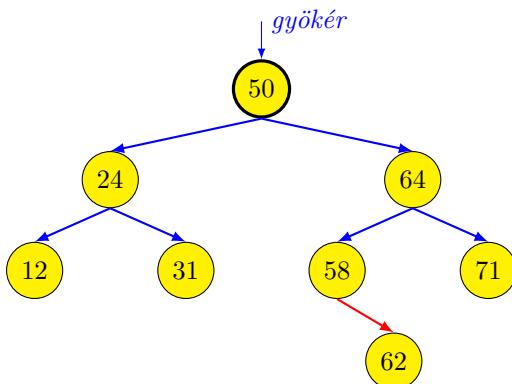
- *gyökér* : A halmazt tároló fa gyökéreleme.
-



6.7. ábra. Új elem felvétele bináris keresőfába (62-es tartalommal).



(g) Az algoritmus beállítja az új elem bal és jobb mutatóját is a lezáró elemekre.



(h) A rekurzió utolsó szintjén a függvény visszatérési értéke egy referencia az új elemre. Amikor visszalép a rekurzió egy szintet, akkor ez az érték kerül be az 58-as elem jobb oldali mutatójába. Ezzel megtörtént a beláncolás.

6.7. ábra. Új elem felvétele bináris keresőfába (62-es tartalommal). (folytatás)

Látható, hogy a fenti beszúrás műveletet használva a kialakuló fa szerkezete jelentősen függ az elemek beszúrásának sorrendjétől. Ugyanabban a sorrendben megadott elemek természetesen minden ugyanazt a fát adják, különböző sorrendek azonban más-más eredményhez vezethetnek.

A fában való keresés hatékonysága jelentősen függ a beszúrás során kialakított szerkezettől. Érdekes megfigyelni, hogy abban az esetben, ha az elemeket tartalom szerint rendezett sorrendben szúrjuk be a bináris keresőfába, akkor minden elem az előtte beszúrt (tehát addig legnagyobb) elem jobb gyerekeként kerül felfűzésre, így a lehető legrosszabb fa szerkezetet kapjuk meg.

#### Megjegyzés

A beszúrás sorrendjétől való függés meglehetősen nagy hátrányt jelent a hagyományos bináris keresőfa esetében, hiszen nem tudjuk garantálni az egyébként igen csábító  $\mathcal{O}(\log_2 n)$  lépésszámú keresést. Emiatt léteznek egyéb, az általunk tárgyalott bináris keresőfához hasonló adatszerkezetek, amelyek a beszúrás során igyekeznek úgy elhelyezni az új elemet (amit a már meglévő fa átstrukturálásával tudnak csak megoldani), hogy minél inkább megmaradjon a fa kiegyensúlyozott jellege (pl. kiegyensúlyozott fák [6], piros-fekete fák [2]).

#### 6.2.5. Elem törlése

Az elem törlése valamivel bonyolultabb művelet, mint a beszúrás, ugyanis még ott mi határozhattuk meg, hogy hová helyezzük el az új elemet (és ezt úgy tettük meg, hogy minél egyszerűbb legyen a beszúrás), addig a törlésnél már nincs hasonló szabadságunk, értelemszerűen onnan kell törölni az elemet, ahol az megtalálható. Bizonyos esetekben, pl. a levelek törlésénél ez kimondottan egyszerű lesz, de a két gyerekkel is rendelkező belső csomópontoknál már nemileg összetettebb algoritmusra lesz szükségünk.

A törlés eljárásnak paraméterként a törlendő értéket adjuk át, emiatt első lépésként meg kell találnunk magát a csomópontot, ahol ezt eltároltuk (6.9. algoritmus). Ehhez igazodva a 3–8. sorok a már megismert keresést mutatják, a *p* által aktuálisan mutatott elem tartalma alapján a rekurzió meghívja önmagát a bal, illetve a jobb oldali részfára, attól függően, hogy a törlendő elem értéke kisebb, vagy éppen nagyobb, mint az aktuálisan vizsgált. Vegyük észre, hogy a rekurzív hívások visszatérési értékével itt is minden felülírjuk a paraméterként átadott mező értékét. Ennek szerepe hasonló mint a beszúrásnál, ezzel fogjuk tudni elegánsan megoldani a kiláncolást.

Amennyiben a program eljut a 9. sorig, akkor már tudjuk, hogy a  $p$  a törlendő elemre hivatkozik. A következő lépés ennek az elemnek a kiláncolása, majd pedig felszabadítása. Attól függően, hogy a törlendő elemnek hány darab gyereke van, három lehetőséget különböztetünk meg, amelyeket külön-külön vizsgálunk meg részletesebben: nincs gyereke, egy gyereke van, két gyereke van.

### **Elem törlése egy gyerek esetén**

Elsőként célszerű ezt az esetet megvizsgálni, mivel látni fogjuk, hogy a levélelemek törlése ennek egy speciális változataként kezelhető. Vizsgáljuk meg a 6.8c. ábrát. Látható, hogy ez azt az állapotot mutatja, amikor már lefutott a keresés, és a  $p$  változó a törlendő elemre mutat. Az algoritmus ebben a pillanatban a 9. soron áll.

Ezt követi egy vizsgálat, ahol megnézzük, hogy melyik oldalon vannak gyerekei a törlendő elemet tartalmazó csúcsnak. Elsőként megnézzük, hogy van-e neki baloldali gyereke (és ezzel együtt persze baloldali részfája). Amennyiben a  $p.bal$  értéke  $\emptyset$ , akkor nincs. Az ábrán jól látható, hogy ha az egész fának csak a  $p$  közvetlen környezetét vizsgáljuk (annak szülőjével, és egyetlen jobb oldali gyerekével együtt), akkor egy láncolt listához hasonló szerkezetet találunk. Ennek megfelelően a törlés is hasonlóan történik, a szülő jobboldali hivatkozását át kell állítanunk a törlendő elem jobboldali elemére.

Ehhez szükség lenne egy referenciára a szülő elemre, amely azonban nem áll rendelkezésre, a bináris keresőfa egyes csomópontjai nem tárolnak el külön hivatkozásokat a szülőkre (miként a láncolt listánál sem volt hivatkozás az előző elemre). A listánál alkalmazott technika (egy  $e$  változóval a keresés közben mindenkorán a szülőtől a szülőtől előző elem címét) lehet az egyik lehetséges megoldás itt is, azonban itt már kicsit nehézkesebb lenne a törlés végrehajtása, hiszen tudnunk kell, hogy a szülő *bal* vagy *jobb* mezőjét kell-e átírni, továbbá azt is, hogy a törlendő elem bal-, vagy jobboldali gyerekére kell átírányítani. Az így megjelenő négy eset mellett még külön kellene foglalkoznunk a gyökérelem törlésével is, amely esetben magát a *gyökér* változót kellene módosítanunk.

Ehelyett a beszúrásnál már látott módszer használjuk, hogy a rekurzív függvények visszatérési értékével mindenkorán felülírjuk annak a mezőnek az értékét, amelyik irányban eljutottunk az adott elemhez (tehát a szülőnek a rá mutató hivatkozását). Amíg csak a keresés lépései futnak, addig ez nem változtat a fában semmit, hiszen visszatérési értékként ugyanazt a  $p$  értéket adjuk vissza, amit paraméterként kaptunk.

Viszont a képen is látható példánál látható, hogy lefut az algoritmus 11. sora, ami megváltoztatja a  $p$  értékét. Mivel a  $p$  csak egy lokális paraméter, ezért ez még önmagában nem változtat semmit a fában. Viszont amikor visszalépünk a rekurzió előző szintjére, akkor a visszatérési értékkel felülíródik a szülő megfelelő mutatója (az ábrán a 64-es elem jobb mutatója).

A felszabadításhoz szükségünk van egy hivatkozásra a törlendő elemre, emiatt szükséges a  $q$  változó. Ez még a kiláncolás előtt eltárolja a felszabadítandó elem címét, majd a 12. sorban ennek segítségével ténylegesen fel is szabadítjuk ezt (amennyiben olyan programozási nyelven implementáljuk az algoritmust, ahol van automatikus szemétygyűjtés, akkor a  $q$  változóra egyáltalán nem is lesz szükség). Ezzel végeztünk is a törléssel.

A fent vázolt módszer csak abban az esetben működik, ha a törlendő elemnek nincs baloldali gyereke. Egészen hasonló módon tudjuk kezelní az esetet is, amikor a törlendő elemnek a jobboldali gyereke (és ezzel együtt a jobboldali részfa) hiányzik. A következő sorokban alapvetően a fent már részletesen áttekintett kódot láthatjuk, pusztán az irányok megcserélésével.

Érdemes belegondolni, hogy mi történik akkor, ha a gyökér elemet szeretnénk törölni. Az eljárás ilyenkor még nem hívta meg önmagát rekurzívan, hanem már az első híváskor megtalálta a törlendő elemet, amire a  $p$  hivatkozik. A lokális  $p$  megváltoztatása (átléptetése a gyerek elemre) természetesen nem változtatja meg az ettől független *gyökér* változó értékét. Viszont amikor visszalépünk a rekurzió előző szintjére, ott a 6.10. algoritmus egyetlen sorában látható módon, felülírjuk a *gyökér* értékét a visszakapott új  $p$  értékkel.

---

## 6.9. Algoritmus Részfából elem törlése

---

Bemenet:  $p$  -  $\tilde{F}aElem^<T>$  (hivatkozás a vizsgálandó részfa gyökerére)

Bemenet: érték -  $T$  (a törlendő érték)

Kimenet:  $\dots$  -  $\tilde{F}aElem^<T>$  (a paraméterként átadott  $p$  (módosított) értéke)

Megszorítás:  $T$  összehasonlítható

```
1: függvény FAHALMAZ.RÉSZFÁBÓLTÖRÖL(p , érték)
2: ha $p \neq \emptyset$ akkor
3: ha $p.tart > \text{érték}$ akkor
4: $p.bal \leftarrow \text{RÉSZFÁBÓLTÖRÖL}(p.bal, \text{érték})$
5: különben
6: ha $p.tart < \text{érték}$ akkor
7: $p.jobb \leftarrow \text{RÉSZFÁBÓLTÖRÖL}(p.jobb, \text{érték})$
8: különben
9: ha $p.bal = \emptyset$ akkor
10: $q \leftarrow p$
11: $p \leftarrow p.jobb$
12: FELSZABADÍT(q)
13: különben
14: ha $p.jobb = \emptyset$ akkor
15: $q \leftarrow p$
16: $p \leftarrow p.bal$
17: FELSZABADÍT(q)
18: különben
19: $p.bal \leftarrow \text{KÉTGYEREK}(p, p.bal)$
20: elágazás vége
21: elágazás vége
22: elágazás vége
23: elágazás vége
24: vissza p
25: különben
26: hiba "Nincs elem"
27: elágazás vége
28: függvény vége
29: függvény KÉTGYEREKESTÖRLÉS(e, r)
30: ha $r.jobb \neq \emptyset$ akkor
31: $r.jobb \leftarrow \text{KÉTGYEREK}(e, r.jobb)$
32: vissza r
33: különben
34: $e.tart \leftarrow r.tart$
35: $q \leftarrow r$
36: $r \leftarrow r.bal$
37: FELSZABADÍT(q)
38: vissza r
39: elágazás vége
40: függvény vége
```

---

Felhasznált változók és függvények

- $e, r, q : \tilde{F}aElem^<T>$  típusú segédváltozók.
  - $\text{FELSZABADÍT}(q) : A q$  által mutatott elem felszabadítása.
-

## 6.10. Algoritmus Bináris keresőfával megvalósított halmaz törlés művelete

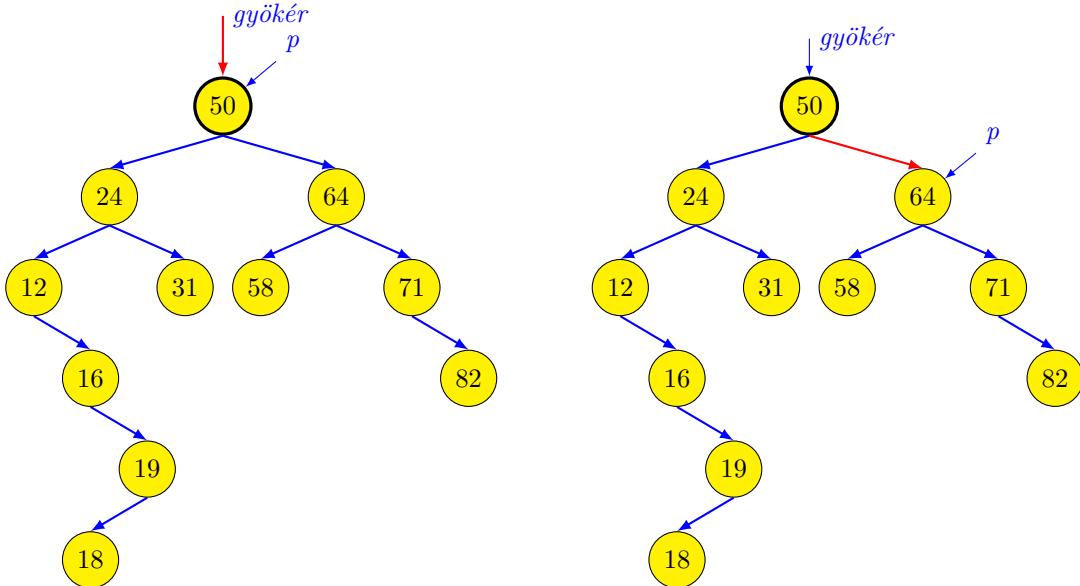
Bemenet: érték - T (a törlendő érték)

Megszorítás: T összehasonlítható

- 1: eljárás FAHALMAZ.TÖRÖL(érték)
- 2:  $gyökér \leftarrow$  RÉSZFÁBÓLTÖRÖL( $gyökér$ , érték)
- 3: eljárás vége

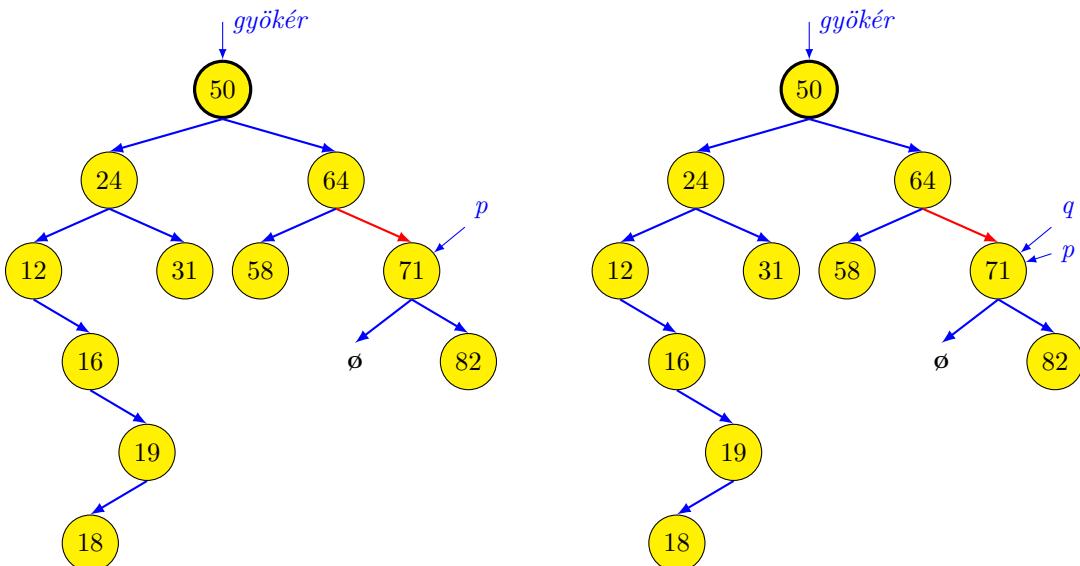
Felhasznált változók és függvények

- $gyökér$  : Hivatkozás a fa gyökérelemére.



(a) A művelet első lépése a törlendő elem megkeresése. Jelen esetben a gyökértől jobbra kell továbblépni.

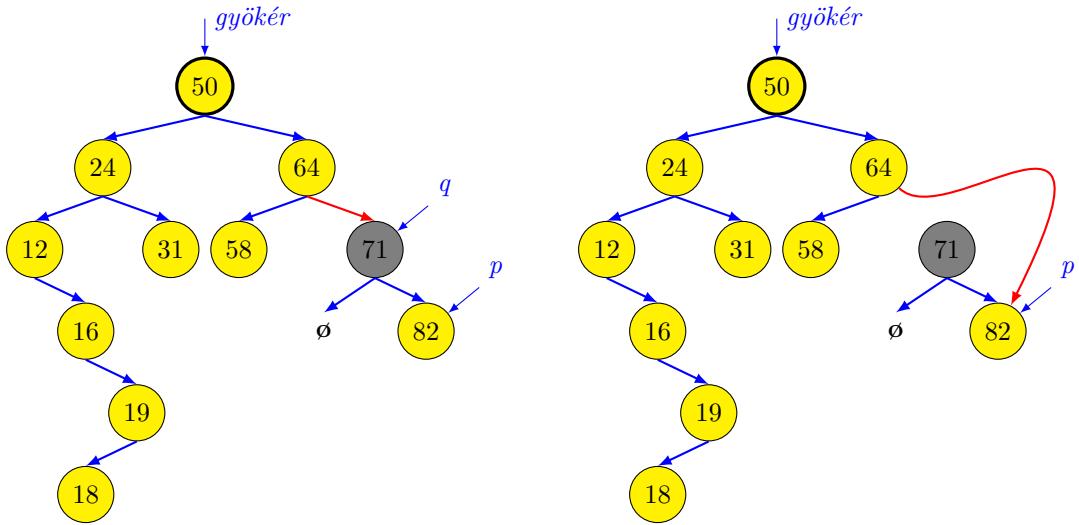
(b) Újabb lépés jobbra. Az ábra azt mutatja, hogy a függvény visszatérési értéke az 50-es elem *jobb* mezőjét fogja felülírni ezen a szinten.



(c) A lefelé haladó rekurzió megtalálta a keresett elemet. Kivételesen berajzoljuk a lezáró jelet is, így látható, hogy nincs baloldali gyereke.

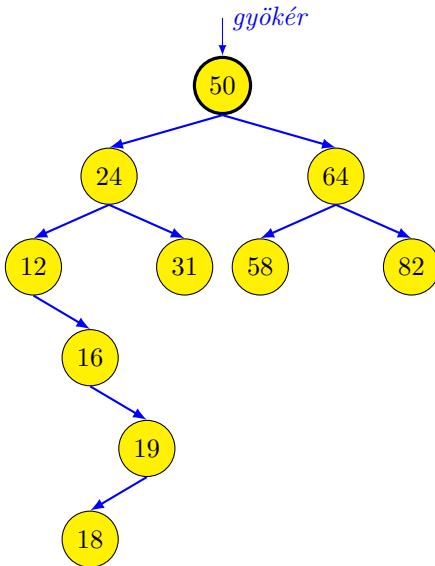
(d) A *q* hivatkozás eltárolja a törlendő elem címét, mivel a *p*-t innen tovább fogjuk léptetni.

6.8. ábra. Bináris keresőfából, egy gyerekkel rendelkező elem törlése (71-es elem). Piros szín jelzi azt a mutatót, amit felülírunk a rekurzív hívásból kapott visszatérési értékkel.



(e) A  $p$  átlép a 82-es elemre, a  $q$  által mutatott elemt felszabadítjuk.

(f) Az utolsó rekurziós szintről visszalépve a 64-es elem jobb mutatóját átirányítjuk oda, ahová a  $p$  mutat. Ezzel megtörtént a kiláncolás.



(g) A rekurzió visszalépeget a legelső szintig, de itt már nem változik a fa adattartalma.

6.8. ábra. Bináris keresőfából, egy gyerekkel rendelkező elem törlése (71-es elem). (folytatás)

### **Elem törlése levél esetén**

Bár külön esetként vizsgáljuk, de valójában ez az előző egy speciális részeseteként kezelhető. Erre mutat egy példát a 6.9. ábra, ezen belül is vizsgáljuk meg a 6.9c. ábrát. Ez ismét azt az állapotot mutatja, amikor már lefutott a keresés, és a *p* változó a törlendő elemre mutat. Az algoritmus ebben a pillanatban a 9. soron áll.

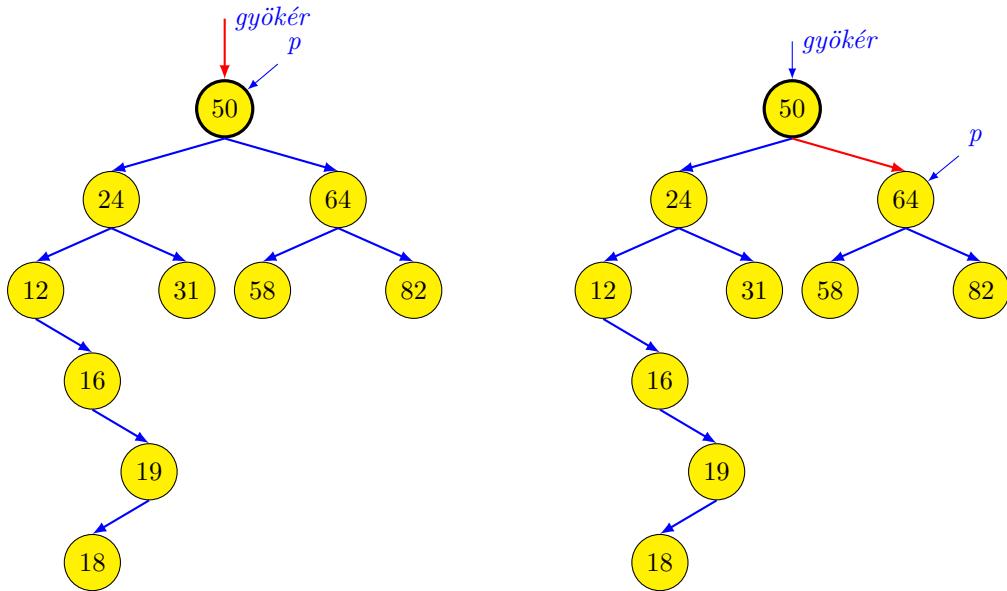
Ezt követi egy vizsgálat, ahol megnézzük, hogy hány gyereke van a törlendő elemnek. Elsőként megnézzük, hogy van-e baloldali gyereke (és ezzel együtt persze baloldali részfája). Amennyiben a *p.bal* értéke  $\emptyset$ , akkor nincs. Jelen esetben ez a feltétel igaz, tehát lefut az elágazás igaz ága.

Észrevehető, hogy az előző példánál, amikor csak egy gyereke volt a törlendő elemnek, akkor is pont ez az ág futott le. Valójában ugyanis a teendők itt is azonosak. Már az előző példánál tisztáztuk a kiláncolás módját. Jelen esetben itt is ugyanaz történik, a *p* változóba betölthjük a jobboldali gyerekre mutató hivatkozásának értékét. Míg az előző példában ez egy tényleges elemre való hivatkozás volt, addig itt ennek az értéke most  $\emptyset$ .

A *p* új értéke tehát  $\emptyset$  lesz, így ez lesz a függvény visszatérési értéke is. A rekurzió előző szintére visszalépve ezzel az értékkel írjuk felül az eddig a törlendő elemre való hivatkozást (tehát a törlendő elem szülőjének jobboldali gyerek hivatkozását), így a fának ez a része is üres lesz.

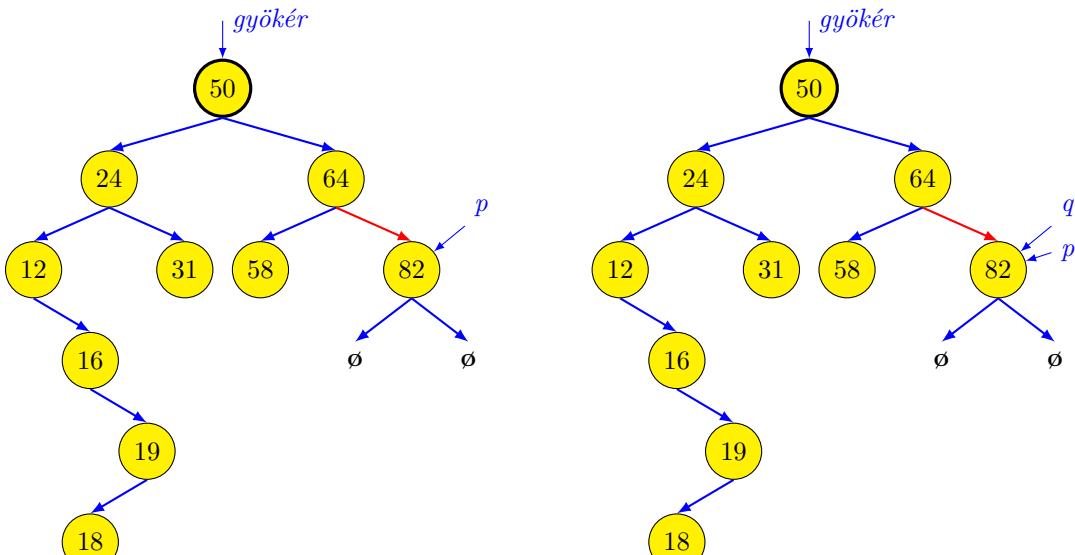
Az elem felszabadítás is hasonlóan történik, a *q* segédváltozóval eltároljuk a felszabadítandó elemet, majd a *p* átléptetése után ezt felszabadítjuk.

Itt is célszerű lehet megvizsgálni azt az állapotot, amikor a gyökérelemet kell törölnünk. A *p* ilyenkor nem más, mint maga a *gyökér* változó. Amikor a *p*-nek értékül adjuk a  $\emptyset$  értéket, és visszalépünk, akkor ezt az értéket fogjuk a rekurziót hívó eljárásban értékül adni a *gyökér* változónak. Ennek hatására az utolsó elem törlésekor beáll az elvárt állapot.



(a) Az első lépés itt a törlendő elem keresése.  
A gyökéből a rekurzió jobbra indul.

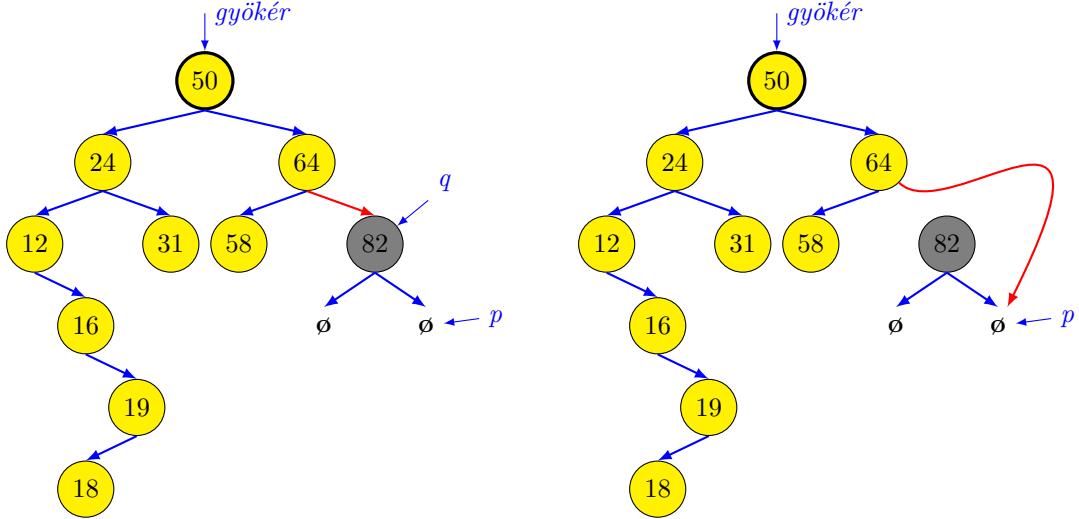
(b) Újabb lépés jobbra.



(c) Egy újabb jobbra lépés után a  $p$  a törlendő elemre hivatkozik. Kivételesen ábrázoljuk a lezáró jeleket is, így látható, hogy nincs gyereke.

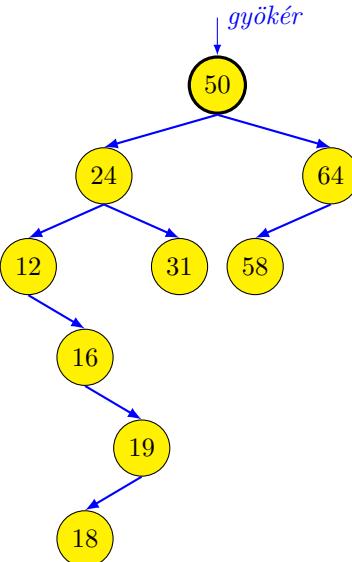
(d) A  $q$  hivatkozás eltárolja a törlendő elem címét, mivel a  $p$ -t innen tovább fogjuk léptetni.

6.9. ábra. Bináris keresőfából levélelem törlése (82-es elem).



(e) A baloldali részfa üres, a jobb oldaliban azonban vannak adatok. Emiatt a *p* átlép a jobb oldali részfára. A *q* által mutatott elemet felszabadítjuk.

(f) A rekurzióból való visszalépést követően felülírjuk a 64-es elem jobb gyerekmutatóját a visszatérési értékkel (ami ugyanaz, mint a *p*). Ezzel megtörténik a kiláncolás.



(g) A keresőfa végső állapota a törlés után.

6.9. ábra. Bináris keresőfából levélelem törlése (82-es elem). (folytatás)

### **Elem törlése két gyerek esetében**

Amennyiben a törlendő csúcsnak két gyereke is van (tehát minden két oldalra egy-egy részfa lett kapcsolva), akkor a törlés már nem oldható meg ilyen egyszerűen. Itt már nem tudjuk a kiláncolást a láncolt listánál látható módon megvalósítani, hiszen a szülőből (vagy gyökérből) csak egy hivatkozás mutat a törlendő elemre, onnan viszont két további hivatkozás halad tovább a bal-, illetve jobboldali részfára.

Emiatt a 6.10. ábrán látható módon próbáljuk megvalósítani a törlést. Ismét átugorhatjuk a keresés lépéseit, és érdemes a 6.10c. ábránál folytatni a vizsgálatokat. Az algoritmus ilyenkor már a 19. soron áll, tehát megtaláltuk a törlendő elemet, a *p* változó erre mutat, és az előző vizsgálatok eredményeképpen már látjuk, hogy minden két oldalon vagy egy-egy gyereke.

A módszer alapja az, hogy egy kisebb átrendezést végezünk el a fában, mielőtt ténylegesen törölünk belőle a szükséges elemet. Az átrendezés az alábbi lépésekkel igényli:

1. A törlendő elemnél kisebb tartalmú elemek közül a legnagyobb megkeresése. A fa rendezettségéből adódik, hogy ez az elem a törlendő csomópont baloldali részfájának legjobboldalibb eleme lesz (tehát a nála kisebbek közül a legnagyobb).
2. Ennek az elemnek a tartalmát átmásoljuk a törlendő elembe. Könnyen belátható, hogy ezt megtehetjük, mivel nem szegyük vele meg a fa rendezettségét. Érdemes a 6.10e. ábrán egy konkrét példán is megnézni ezt az állapotot. A másolás három helyen befolyásolhatja a rendezettséget, de egyik sem okoz problémát:
  - Az 50-es elem szempontjából nincs változás, hiszen eddig is minden két elem a baloldali részfájában volt, most is így maradtak.
  - A 24-es eddigi bal részfája szempontjából a rendezettség továbbra is jó lesz, hiszen a felmásolt elem a részfa legnagyobb értékét tartalmazza, tehát ezek az elemek az Ő baloldalán jó helyen vannak.
  - A 24-es eddigi jobb részfáját se kell módosítani, hiszen itt eddig is a 24-nél nagyobb elemek lehettek csak, ha a 24-et kicseréljük egy kisebbre, attól a rendezettség még biztosan megmarad.
3. Töröljük az előbb megtalált csomópontot a fából, hiszen nem szükséges, hogy az elem kétszer is benne legyen. Maga a törlés itt már biztosan megoldható egyszerűen, hiszen ennek a csomópontnak nem lehet két gyereke (akkor nem lehetne Ő a legjobboldalibb elem a részfában).

A **KÉTGYEREKES TÖRLÉS** eljárás tartalmazza ennek a módszernek a lépéseit. Mivel egy részfa legnagyobb értékű elemét kell megtalálnunk, egy rekurzióval addig lépegetünk az aktuális csomópont jobboldali gyerekére, amíg az létezik. Az eljárás tehát először addig hívogatja magát a jobboldali részfájára, amíg a jobboldali hivatkozás értéke  $\emptyset$  nem lesz. Az eljárásnak két paramétere van, az első (*e*) a törlendő elemre mutat, ez a hívások között nem is változik. A második (*r*) pedig magát az itt leírt keresést végzi.

Ezt követi a tartalom átmásolása a 34. sorban, felülírva a törlendő elem tartalmát. Az előzőekben már beláttuk, hogy ez a másolás bármikor megtehető, nem fogja befolyásolni a fa rendezettségét.

Az utolsó lépés az átmásolt tartalmú elem kiláncolása. Ehhez a törlendő elem szülőjének a rá mutató hivatkozását kell módosítani, ezt az előző törlési módokhoz hasonlóan oldjuk meg. Tudjuk, hogy a rekurzióból való visszalépéskor a függvény visszatérési értékével feltülvállalt is mutatott csúcsba. Tehát kiláncoláskor az *r* elem *bal* mezőjét adjuk vissza. Ennek két kimenetele lehet:

- Ha az *r* által mutatott csomópontnak van baloldali gyereke, akkor a szülő kiláncolandó elemre való hivatkozását erre állítjuk át (ez látható az ábrán).
- Ha az *r* által mutatott elemnek nincs baloldali gyereke, akkor a *bal* mező értéke  $\emptyset$ . Ilyenkor ez lesz a visszatérés érték, tehát a szülő megfelelő mezőjét is lezárjuk, hiszen abba az irányba már nem folytatódik a fa.

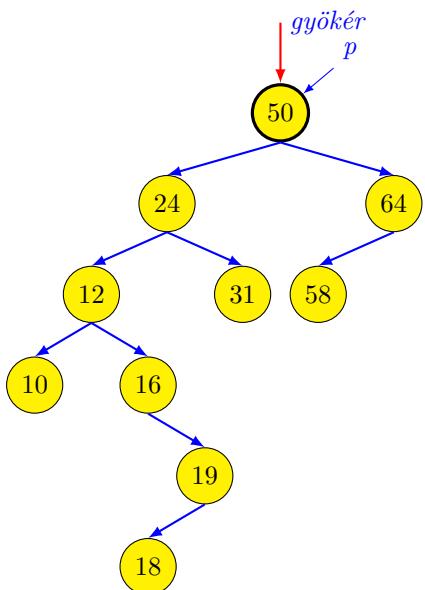
A *r* által mutatott elemet fel szeretnénk szabadítani még a kilépés előtt, viszont visszatérési értékként szükségünk van az *r.bal* mutató értékére. Mivel a felszabadítás után ez már nem érhető el, ezért be kellett vezetnünk egy új *q* változót, hogy eltárolja ezt az értéket.

### Megjegyzés

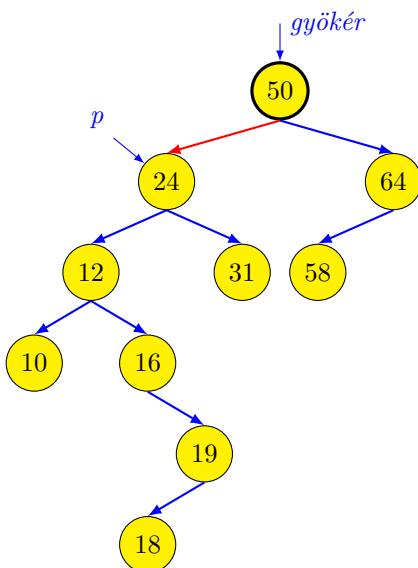
Érdemes még néhány speciális esetet figyelembe venni a két gyerekkel rendelkező elemek törlésénél (6.10. ábra):

- Hogyan működik az algoritmus akkor, ha a 12-es elemnek nincs egy jobboldali gyereke sem, tehát a részfa gyökéreleme egyben a legnagyobb elem is?
- Hogyan működik az algoritmus akkor, ha a 12-es elemnek se jobb-, sem pedig baloldali gyereke nincs. Tehát valójában ő a részfa egyetlen (így értelemszerűen legnagyobb) eleme?

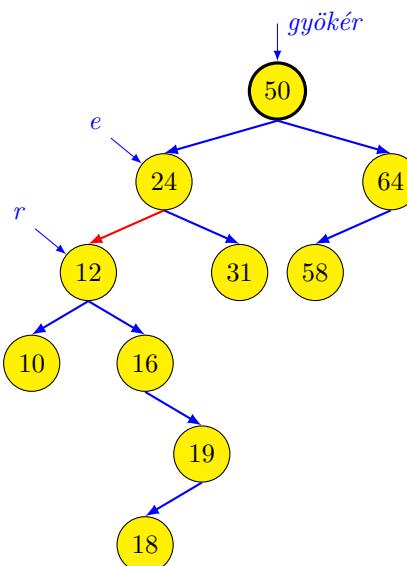
Mindkét esetben működni fog a tanult módszer, de érdemes egy-egy példán végigpróbálni, hogy pontosan milyen lépéseket hajt végre.



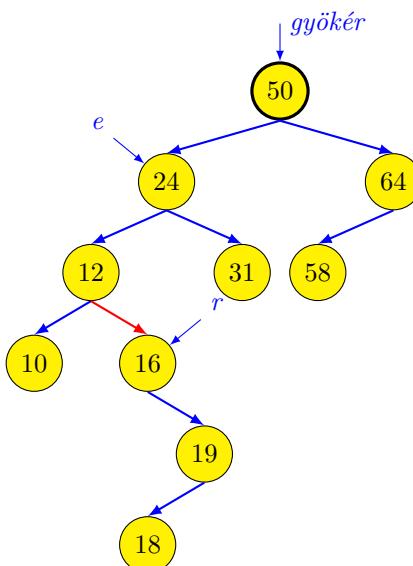
(a) Az első lépés itt is a törlendő elem keresése.  
A gyökérből a rekurzió balra indul.



(b) A következő szinten már meg is találtuk a törlendő elemet. Mint látható, minden két hivatkozás létező elemekre hivatkozik, emiatt megkeressük a baloldali részfa legjobboldalibb elemét.

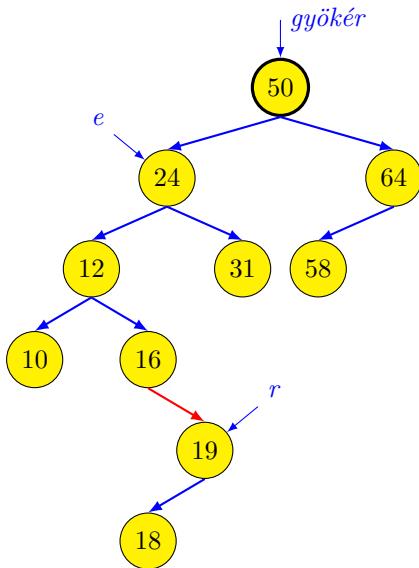


(c) A következő rekurzió során a *e* mindenkor a törlendő elemre mutat, az *r* pedig kezdetben annak a baloldali gyerekére. A piros szín mutatja, hogy melyik mező lesz felülírva a rekurzió visszatérési értékével. Mivel ez egy másik eljárás, így az előző *p* itt nem érhető el, ezért nem is ábrázoljuk. Mivel van az *r*-nek jobboldali gyereke, így átlépünk oda.

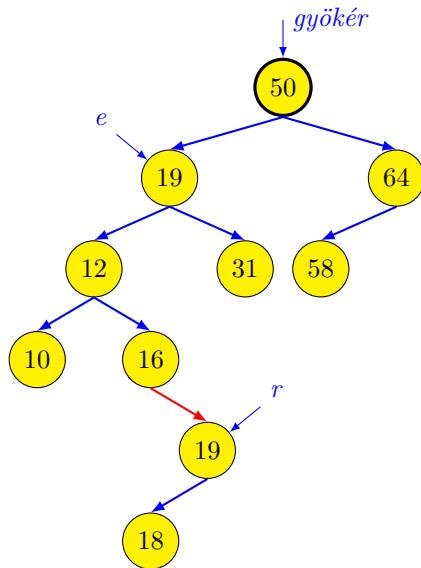


(d) Az *r* által mutatott elemek még mindenkor van jobboldali gyereke, így a függvény megírja önmagát, paraméterként átadva ezt az elemet.

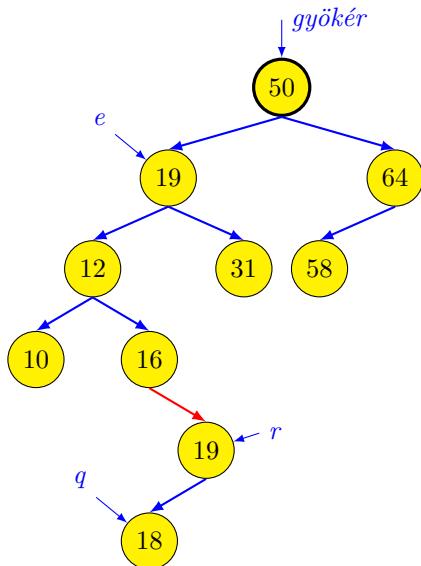
6.10. ábra. Bináris keresőfából belső csúcs törlése (24-es elem).



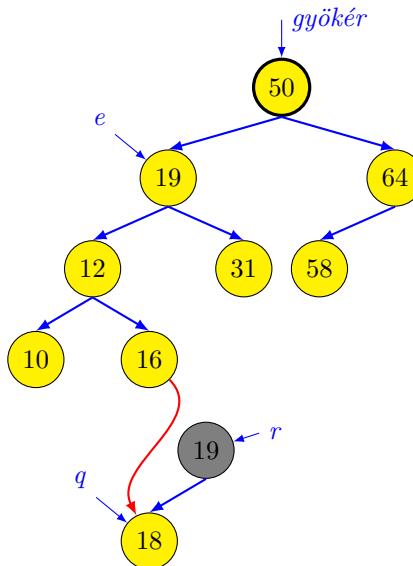
(e) A  $r$  által mutatott elemnek már nincs jobb oldali gyereke, így a rekurzió nem halad tovább, hanem folytatja a törlést.



(f) Tudjuk, hogy az  $r$  által mutatott elem tartalma a törlendő elemnél kisebbek közül a legnagyobb. Emiatt ennek értékét felmásolhatjuk a törlendő tartalom helyére.



(g) A  $q$  változót átléptetjük az  $r$  baloldali gyerekére, ide kell majd mutatnia a szülő jobb mutatójának.



(h) Felszabadítjuk az  $r$  által hivatkozott elemet. A rekurzióból való visszalépést követően pedig a 16-os elem jobb mutatóját átléptetjük a  $q$  által hivatkozott elemre (mivel ez a függvény visszatérési értéke). Ezzel kész van a törlés.

6.10. ábra. Bináris keresőfából belső csúcs törlése (24-es elem). (folytatás)

## 7. fejezet

# Szótár

### 7.1. Szótár működése

Ez a fejezet a szótárak megvalósítási lehetőségeit tárgyalja. Miként az absztrakt adatszerkezeteket bemutató fejezetben erről már szó esett, ez a konstrukció jelentősen eltér az eddig tárgyaltaktól, ugyanis az egyszerű tartalom helyett ez kulcs-érték párokkal dolgozik. Emiatt a már megszokott **T** típusú tartalom mellett bevezetünk egy **K** típusú kulcs fogalmat is. A szótár lényege, hogy tároláskor megadunk a tartalomhoz egy kulcsot, majd pedig a lekérdezéskor a kulcs alapján tudjuk kiolvasni a szükséges értéket.

Maga a kulcs egy tetszőleges típusú érték, ezért ennek típusát nem is pontosítjuk, hanem a már szokás generikus **K** típussal hivatkozunk rá. Maga a kulcs arra szolgál, hogy ez alapján tudjuk eltárolni, illetve a későbbiekben visszakeresni a tartalmat.

Fontos követelmény, hogy ez a kulcs legyen egyedi, tehát két különböző elem nem rendelkezhet ugyanazzal a kulccsal.

Ideális esetben a már egyébként is meglévő mezők közül valamelyiket közvetlenül használhatjuk kulcsként (pl. cégek neveit, stb.). Gyakran ez nem csak egy mezőt, hanem többet jelent, ilyenkor beszélünk összetett kulcsokról (pl. a gyakorlatban a „név+anyja neve+születési hely és dátum” egészen jól bevált a személyek azonosítására). Ez utóbbi esetben az összes értéket meg kell adnunk az egyértelmű azonosításhoz, de ez inkább csak implementációs kérdést jelent, a későbbiekben ezzel nem foglalkozunk.

Néha azonban a meglévő mezők nem alkalmasak az egyértelmű megkülönböztetésre, ilyenkor bevezethetünk *technikai kulcsokat* (pl. személyi szám, memóriabeli cím, stb.), amelyek már nyilvánvalóan biztosítani tudják a kívánt egyediséget.

Az adatszerkezetekben a kulcsok alapján szoktuk elvégezni a kiolvasást (pl. keressük a "998877AB" azonosítójú személy adatait), ami azonnal elvezet az egyszerű tömbben való tárolás egyik hátrányához. Tömböknél ugyanis a hozzáférés alapja az indexelés, ami az elemek memóriabeli elhelyezkedésén alapul, az pedig a kulcsoktól alapvetően független. A legtöbb amit tehetünk, hogy magát a tömböt a kulcsok alapján rendezzük, így a kulcs szerinti kereséskor (pl. adjuk vissza a megadott személyi számmal rendelkező személyt) a bináris keresést használjuk, ami  $n$  darab elem esetében várhatóan  $\mathcal{O}(\log_2 n)$  lépésből megtalálja a keresett kulcsú elemet. Később látni fogjuk, hogy ez alapvetően nem rossz, de bizonyos feltételek meglété esetében ennél sokkal hatékonyabb megoldásokat is találhatunk, akár az  $\mathcal{O}(1)$  lépésszám is elképzelhető.

## 7.2. Megvalósítás közvetlen címzéssel

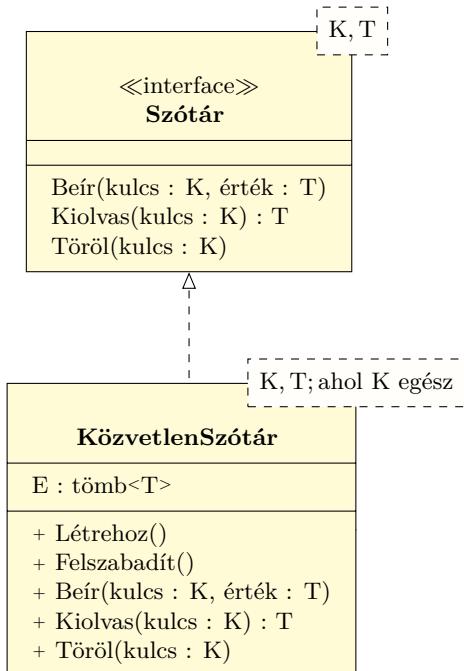
### 7.2.1. Alapvető felépítés

Az egyik ezt megvalósító tárolási módszer az úgynenezett *közvetlen címzés*. Ennek az alapja az, hogy a kulcsokat közvetlenül a tömb címzésére használjuk, aminek persze meglehetősen szigorú előfeltételei vannak:

- a kulcsoknak természetes számoknak kell lenniük (mivel tömböt csak ezekkel tudunk indexelni);
- a kulcsoknak egy megadott tartományon belül kell maradniuk (hiszen a tömb méretet előre meg kell adnunk);
- tehát  $n$  darab elem esetében a kulcsoknak az  $1..n$  tartományon belül kell lenniük.

Itt megjegyezzük, hogy a kulcsoknak egyediknek is kell lenniük, de ezt már eleve beleértettük a kules fogalmába, tehát ezt természetesen vesszük.

A  $Szótár <K \rightarrow T>$  adatszerkezet hatékonyan megvalósítható közvetlen címzéssel, de csak abban az esetben, ha a kulcsok felépítése követi a fenti követelményeket. Az adatszerkezet egy belső  $E$  tömböt tartalmaz, aminek mérete megegyezik a maximális kulcs mérettel.

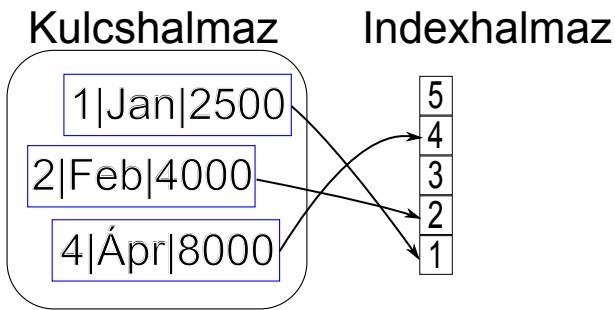


7.1. ábra. Szótár megvalósítása közvetlen címzés segítségével.

Mint látható, a tervezett szótár  $T$  elemeket fog tudni eltárolni, erre nincs semmilyen megkötésünk. A kulcsokat a  $K$  generikus típussal jelöljük, habár itt nincs túl nagy szabadságunk, ezek csak a természetes számok lehetnek egy előre megadott felső korláttal.

Ha teljesülnek a fenti feltételek, akkor a különböző műveletek már nagyon egyszerűen implementálhatóak.

- **BEÍR(kulcs, érték):** Egy elem elhelyezése a tömbben csak annyit igényel, hogy a paraméterként átadott  $\text{érték}$  elemet eltároljuk a tömb  $\text{kulcs}$  indexű cellájában.
- **KIOLVAS(kulcs):** A keresés hasonlóan egyszerű, egy megadott kulcsú elemet azonnal megtalálunk, csak ki kell olvasni a tömb  $\text{kulcs}$  indexű elemét. Ha van ott valami, akkor azt kerestük, ha nincs, akkor pedig nincs ilyen kulcsú elemünk.



7.2. ábra. Példa a közvetlen címzésre. A címzés során a hónap sorszámát használjuk kulcsként, tehát beszúráskor a januári érték az 1. cellába kerül, stb. Kiolvasáskor ehhez hasonlóan, ha pl. az áprilisi adatra van szükség, akkor azt a 4. cellából tudjuk kivenni.

- **TÖRÖL(kulcs):** Az előzőleg megismert kereséssel eldöntjük, hogy van-e megadott kulcsú elem, és ha igen, akkor töröljük/törölnek jelöljük a tömb *kulcs* indexű elemét.

Közvetlen címzésre mutat egy egyszerű példát a 7.2. ábra.

### 7.2.2. Műveletek megvalósítása

A **LÉTREHOZÁS** művelet során létrehozzuk az *E* tömböt. Feltételezzük, hogy a tömb eleve üres állapotában jön létre, tehát minden elemnek értéke  $\emptyset$ . A 7.1. algoritmus azt mutatja be, hogy a tömb méretének a *K* típusú kulcsok maximális értékét tekintjük ( $|K|$ ). OOP megvalósítás esetében az inicializálás tipikusan a konstruktur hívásban történhet meg.

---

#### 7.1. Algoritmus Közvetlen címzésen alapuló szótár létrehozása

---

- 1: **eljárás** KÖZVETLENszótár.LÉTREHOZ( )
  - 2:     *E*  $\leftarrow$  LÉTREHOZ(*tömb*<*T*>/ $|K|$ )
  - 3: **eljárás vége**
- 

#### Felhasznált változók és függvények

- *E* : A tömb ami tárolni fogja az értékeket.
  - $|K|$  : A *K* lehetséges értékeinek száma (jelen esetben ez a legnagyobb értékkel egyenlő).
  - LÉTREHOZ(*tömb*<*T*>/*méret*) : Létrehoz egy *T* típusú, *méret* nagyságú tömböt.
- 

A megszűntetés 7.2. algoritmus még egyszerűbb, csak felszabadítjuk az előzőleg lefoglalt tömböt.

---

#### 7.2. Algoritmus Közvetlen címzés alapú szótár felszabadítása

---

- 1: **eljárás** KÖZVETLENszótár.FELSZABADÍT( )
  - 2:     FELSZABADÍT(*E*)
  - 3: **eljárás vége**
- 

#### Felhasznált változók és függvények

- *E* : Az elemeket tartalmazó tömb.
  - FELSZABADÍT(*E*) : Felszabadítja az *E* tömböt.
- 

A szótártól elvárható műveletek megvalósításai se bonyolultabbak. Egy új elem felvételekor (7.3. algoritmus) egyszerűen felvesszük az elemet a tömbbe a kulcs által megadott indexű helyre.

---

### 7.3. Algoritmus Közvetlen címzés alapú szótárba elem beírása

---

**Bemenet:** *kulcs* - **K** (a tároláshoz használatos kulcs)

**Bemenet:** érték - **T** (a tárolandó érték)

**Megszorítás:** **K** egész

- 1: eljárás KÖZVETLENszótár.BEÍR(*kulcs*, érték)
  - 2:  $E[kulcs] \leftarrow$  érték
  - 3: eljárás vége
- 

#### Felhasznált változók és függvények

- **E** : Az elemeket tartalmazó tömb.
- 

Kiolvasásnál megnézzük, hogy van-e bármilyen elem a tömb *kulcs* által meghatározott pozíciójában (7.4. algoritmus). Ha nincs, akkor jelezzük a hibát, ha pedig van, akkor ez lesz a függvény visszatérési értéke.

---

### 7.4. Algoritmus Közvetlen címzés alapú szótárból elem kiolvasása

---

**Bemenet:** *kulcs* - **K** (a kiolvasandó elem kulcsa)

**Kimenet:** . - **T** (a kulcsnak tartozó érték)

**Megszorítás:** **K** egész

- 1: függvény KÖZVETLENszótár.KIOLVAS(*kulcs*)
  - 2: ha  $E[kulcs] \neq \emptyset$  akkor
  - 3: vissza  $E[kulcs]$
  - 4: különben
  - 5: hiba "Hibás kulcs"
  - 6: elágazás vége
  - 7: függvény vége
- 

#### Felhasznált változók és függvények

- **E** : Az elemeket tartalmazó tömb.
- 

A törlés is egy sorból megoldható (7.5. algoritmus), hiszen csak felül kell írnunk a megadott kulcsú elemet az ürességet jelző jellet.

---

### 7.5. Algoritmus Közvetlen címzés alapú szótárból elem törlése

---

**Bemenet:** *kulcs* - **K** (a törlendő elem kulcsa)

**Megszorítás:** **K** egész

- 1: eljárás KÖZVETLENszótár.TÖRÖL(*kulcs*)
  - 2:  $E[kulcs] \leftarrow \emptyset$
  - 3: eljárás vége
- 

#### Felhasznált változók és függvények

- **E** : Az elemeket tartalmazó tömb.
- 

### 7.2.3. Értékelés

Ezzel sikerült is implementálnunk egy olyan adatszerkezetet, ami  $\mathcal{O}(1)$  darab lépéssel tudja megvalósítani az alapműveleteket (sőt, valójában az  $\mathcal{O}$  is szükségtelen, hiszen konkrétan egy lépést igényel csak). Ez valóban igaz, de sajnos ez is számos hátránnal bír:

- A kulcsokra nagyon szigorú feltétel vonatkozik, ez a legtöbb gyakorlati esetben nem kivitelezhető (mert a kulcsok pl. szövegek).
- A használhatóságot nagyban befolyásolja a lehetséges kulcsok számának, illetve a ténylegesen eltárolni kívánt elemek számának aránya. Ha kulcsként pl. születési dátumot szeretnénk használni (az egyszerűség kedvéért év utolsó két számjegyétől kezdve, tehát pl. 781218,120228, 931211, stb.) akkor egy 991231 méretű tömböt kell létrehoznunk. Amennyiben az eltárolandó személyek száma csak néhány száz, akkor jól látható, hogy tárterület szempontjából nagyon pazarlóan működik

az adatszerkezetünk (és a születési dátumot még nem is tekinthetjük egyedinek, a tökéletességre törekedve jobb lenne a 11 jegyű személyi szám).

A fenti hátrányuktól eltekintve a közvetlen címzés bizonyos esetekben nagyon jól használható, főleg olyankor, amikor a kulcs halmaz meglehetősen kicsi (pl. a havi fizetéseket akarjuk eltárolni közvetlen címzéssel, akkor elég egy 12 méretű tömböt létrehozni), vagy legalább jól körülhatárolt (pl. mi magunk adunk olyan technikai kulcsokat, hogy azok egy kisebb tartományon belül maradjanak).

A használhatóság mindenkor az egyedi feladatok esetében dönthető el, a tárolt adatok mennyisége és mérete, a kulcsok milyensége és a várható keresések számának függvényében.

## 7.3. Megvalósítás hasító táblázattal (kulcsütközés kezelés nélkül)

### 7.3.1. Hasítás alapelve

A közvetlen címzés gondolatából kiindulva egy olyan adatszerkezetet készítünk, ami megpróbálja kiküszöbölni a 7.2. alfejezetben megfogalmazott problémákat. A közvetlen címzés legnagyobb gyengesége éppen annak „közvetlen” volta, tehát, hogy a kulcsot közvetlenül akarjuk felhasználni a tömb címzésére, ami gyakran problémákat vet fel (hiszen a kulcsok típusát általában nem mi határozzuk meg, hanem az termézszerűleg adódik). Érdemes tehát mindezt közvetett módon megvalósítani, hogy legyen lehetőség a kulcsokra először valamelyen transzformációt végezni, mielőtt azokat címzésre használnánk.

Ezt valósítják meg a hasító táblázatok. Ebben az esetben bevezetünk egy függvényt (az úgynevezett *hasítófüggvényt*, vagy szébb nevén *kulcstranszformációs függvényt*), aminek az a szerepe, hogy a beszúrás/kiolvasás/törlés során nem a kulcsot használjuk közvetlenül a címzésre, hanem először a kulcsra alkalmazzuk ezt a függvényt, és annak a visszatérési értékével fogunk indexelni. Az adatok tárolását a háttérben általában egy egyszerű tömb végzi, amit *E*-vel jelölünk. A hasító táblázatra mutat egy egész szám példát a 7.3. ábra.

Ez nagyrészt megoldja a közvetlen címzés problémáit, mivel:

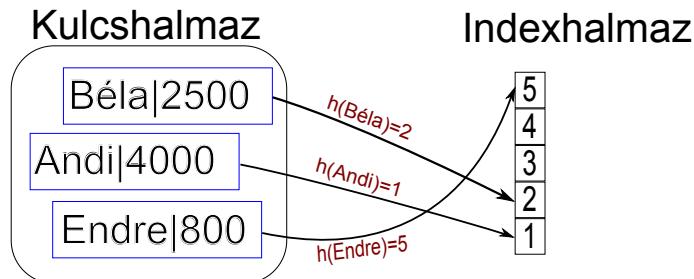
- A kulcsok lehetnek tetszőleges típusúak, hiszen végül nem azokkal fogunk címezni. Lehet a kulcs akár szöveg is, csak akkor olyan hasítófüggvényt kell találnunk, ami a szöveges bemenethez egy egész szám kimenetet rendel.
- A kulcsoknak nem kell egy megadott tartományon belül maradniuk, elég ha úgy választjuk meg a hasítófüggvényt, hogy a visszatérési értéke egy megfelelő tartományon belül maradjon.

Néhány alapvető fogalom:

- Kulcs halmaz ( $K$ ): A lehetséges kulcsok halmaza. Ez lehet tetszőleges típusú és méretű. Mérete legyen  $|K|$  (tehát a  $K$  halmaz lehetséges elemeinek a száma).
- Indexhalmaz ( $I$ ): A lehetséges indexek halmaza. Mivel a hasító táblázatok adatait általában egy tömbben tároljuk, így az indexhalmaz mérete ( $|I|$ ) összhangban kell legyen a tömb méretével. A tömb lehetséges indexei:  $1..|I|$ .
- Hasítófüggvény ( $h : K \rightarrow I$ ): Egy leképezés, amelyik a kulcs alapján megadja a keresett indexet.

#### Megjegyzés

Fontos, hogy a halmazok „lehetséges értékei”-nek számáról beszélünk. Tehát mindenkor, hogy hány elemet akarunk éppen eltárolni egy hasító táblázatban, ha a kulcs halmaz a 2 jegyű egész számoknak felel meg, akkor  $|K| = 90$ .



7.3. ábra. Példa hasító táblázatra. A lényeges különbség a direkt címzéshez képest az, hogy a kulcsmezőt (ami jelen esetben a név) nem közvetlenül használjuk indexelésre, hanem előtte meghívjuk a kulcsra a hasítófüggvényt, és annak a visszatérési értéke lesz az index. Ugyanígy működik a kiolvasás is a kulcs alapján. A hasítófüggvény ezen a példán visszaadja a név első betűjének magyar ABC szerinti pozícióját.

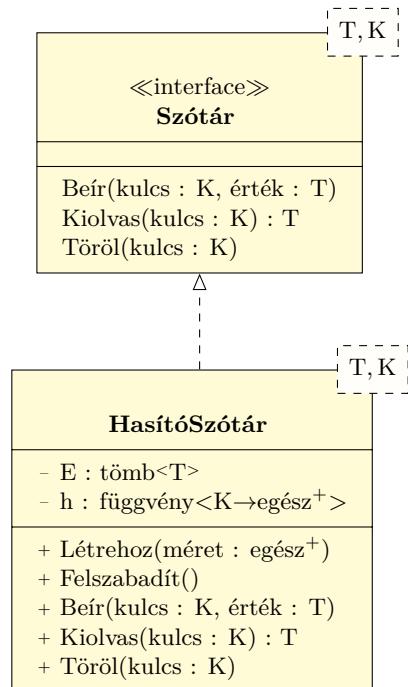
A hasító táblázatok legszembetűnöbb előnye a közvetlen címzéshez képest, hogy ezzel lehetőségünk nyílik arra, hogy egy meglehetősen nagy kulcs halmaz esetén (pl. ha a 11 jegyű személyi számot tekintünk kulcsnak, akkor a lehetséges kulcsok halmaza meglehetősen nagy) is elég lehet egy nagyságrendekkel kisebb indexhalmaz is (pl. 100 elemű), csak arra kell ügyelnünk, hogy a hasítófüggvény megfelelően képezzent le ebbe a kisebb halmazba (pl.  $h(k) = k \bmod 100 + 1$ ).

Egy tökéletes (tehát ütközéseket nem okozó) hasítófüggvény esetében, ha egy  $k$  kulcsú (ahol  $k \in K$ ) elemet szeretnénk eltárolni, akkor a tömb  $h(k)$  indexű helyére bemásoljuk ezt az elemet. Ugyanígy, egy  $k$  kulcsú elem kiolvasásakor megvizsgáljuk a tömb  $h(k)$  indexű elemét, ha az üres, akkor nincs ilyen elemünk, ha nem üres, akkor pedig az ott lévőt kell visszaadnunk. Törölés esetén pedig a tömb  $h(k)$  indexű elemét megjelöljük töröltként.

Fontos megjegyezni, hogy mindenek csakis egy tökéletes hasítófüggvény esetében igaz, a gyakorlatban ilyennel nem nagyon fogunk találkozni. A hasító táblázat említtett előnye, hogy kisebb indexhalmazba képezzük le a sokkal nagyobb méretű kulcs halmazt, törvényszerűen azzal jár együtt, hogy felmerül a kulcsütközések problémája. A fenti példánál maradva elköpzelhető, hogy két ember személyi számának megegyezik az utolsó két számjegye, így őket a hasítófüggvény ugyanarra a tömb indexre képezi le. Belátható általánosan is, hogy bármelyik esetben, ha a kulcs halmaz mérete nagyobb, mint az indexhalmaz mérete ( $|K| > |I|$ ), akkor számolnunk kell az ütközésekkel. Ez nem lesz akkora probléma, mint amilyennek elsőre tűnik, és a 7.5. fejezetben számos módszert fogunk megismerni ennek a kezelésére, de érdemes már most megjegyezni, hogy a kiváló egy lépésből álló beírás/kiolvasás/törölés műveleteket sajnos nem fogjuk tudni a gyakorlatban is megtartani.

### 7.3.2. Alapvető felépítés

A megismert hasító táblázat kiválóan alkalmas a **Szótár**  $\langle K \rightarrow T \rangle$  adatszerkezet megvalósítására. A felépítés nagyban hasonlít a közvetlen címzésnél látott hozzá, az adatszerkezet egy belső **E** nevű, **T** típusú elemeket tartalmazó tömböt tartalmaz. Ennek mérete azonban itt már az inicializáláskor meghatározandó, hiszen nincs semmilyen feltételezésünk a **K** típusossal kapcsolatban (még azt se várjuk el, hogy véges halmazt írjon le).



7.4. ábra. Szótár megvalósítása hasító táblázat.

Mint látható, a tervezett szótár **T** típusú elemeket fog tudni eltárolni, kulcsként pedig **K** típusú értékeket várunk. Egyik típusra sincs további megkötésünk.

A pszeudokódokban használni fogunk egy `h` hasítófüggvényt, felmerülhet kérdésként, hogy ez hol kerül megvalósításra. Erre több megoldás is létezik:

- Legyen magának a hasító táblázatnak egy `h` függvény típusú mezője, ami például a konstruktorban kap értéket. A későbbiekben ezt a függvényt használjuk a hasításra.
- Amennyiben az egyes eltárolandó kulcsok objektumok, akkor akár ezeknek is lehet egy `h` nevű metódusuk, ami visszaadja az adott kulcsértékhez tartozó hasítófüggvény értékét.

A gyakorlatban bármelyik megoldás használható, de akár a kettő kombinációja is (ha van a hasító táblázatnak saját függvénye, akkor azt használja, ha nincs, akkor pedig a beírandó elemekét). A pszeudokódok szintjén nemes egyszerűséggel nem foglalkozunk ezzel a kérdéssel, egyszerűen úgy tekintjük, hogy egy ilyen `h` elérhető.

A megvalósítás helye mellett persze az is kérdés, hogy miként lehet elkészíteni egy ilyen hasító függvényt. Ezt részletesebben megvizsgáljuk a 7.4. fejezetben. Egyelőre elég annyit tudnunk, hogy ez egy olyan függvény, ami paraméterként egy kulcsot kap, visszatérési értéke pedig egy szám, amivel indexelni tudjuk az `E` tömböt.

### 7.3.3. Műveletek megvalósítása

A `LÉTREHOZ(méret)` művelet során létrehozzuk a paraméterként átadott méretű `E` tömböt. Feltételezzük, hogy a tömb eleve üres állapotában jön létre, tehát minden elemnének értéke  $\emptyset$ . A 7.6. algoritmus azt mutatja be, hogy a tömb méretének a paraméterként átadott értéket válasszuk. OOP megvalósítás esetében az inicializálás tipikusan a konstruktor hívásban történhet meg.

---

#### 7.6. Algoritmus Hasító táblázaton alapuló szótár létrehozása

**Bemenet:** `méret - egész+` (a tömb mérete, ami tárolni fogja az értékeket)

- 1: `eljárás HASÍTÓSZÓTÁR.LÉTREHOZ(méret)`
  - 2:    $E \leftarrow \text{LÉTREHOZ(tömb<}T\text{>/[méret]})$
  - 3: `eljárás vége`
- 

#### Felhasznált változók és függvények

- `E` : A tömb ami tárolni fogja az értékeket.
  - `LÉTREHOZ(tömb<}T\text{>/[méret])` : Létrehoz egy `T` típusú, `méret` nagyságú tömböt.
- 

A felszabadítás (7.7. algoritmus) során pedig itt is csak felszabadítjuk az előzőleg lefoglalt tömböt.

---

#### 7.7. Algoritmus Hasító táblázat alapú szótár felszabadítása

- 1: `eljárás HASÍTÓSZÓTÁR.FELSZABADÍT( )`
  - 2:   `FELSZABADÍT(E)`
  - 3: `eljárás vége`
- 

#### Felhasznált változók és függvények

- `E` : Az elemeket tartalmazó tömb.
  - `FELSZABADÍT(E)` : Felszabadítja az `E` tömböt.
- 

A szótártól elvárható műveletek megvalósításai abban különböznek a közvetlen címzéstől, hogy nem magát a kulcsot használjuk a tömb indexelésére, hanem azt elsőként átadjuk paraméterként a `h` hasító függvénynek, és annak visszatérési értékét használjuk. Ez jól látható a beírás művelet során (7.8. algoritmus).

Kiolvasásnál megnézzük, hogy van-e bármilyen tömb `kulcs` által meghatározott pozíciójában (7.9. algoritmus). Itt se a kulcsot használjuk közvetlenül, hanem annak a hasítófüggvény által módosított értékét. Ha nincs, akkor jelezzük a hibát, ha pedig van, akkor ez lesz a függvény visszatérési értéke.

A törlés a fentiek alapján már remélhetőleg nem okoz meglepetéseket (7.10. algoritmus).

---

### 7.8. Algoritmus Hasító táblázat alapú szótárba elem beírása

---

Bemenet: *kulcs* - **K** (a tároláshoz használatos kulcs)

Bemenet: érték - **T** (a tárolandó érték)

- 1: eljárás HASÍTÓSZÓTÁR.BEÍR(*kulcs*, érték)
  - 2:  $E[h(kulcs)] \leftarrow$  érték
  - 3: eljárás vége
- 

Felhasznált változók és függvények

- **E** : Az elemeket tartalmazó tömb.
  - **h** : Tetszőleges hasítófüggvény (ahol  $h : K \rightarrow I$ ).
- 

---

### 7.9. Algoritmus Hasító táblázat alapú szótárból elem kiolvasása

---

Bemenet: *kulcs* - **K** (a kiolvasandó elem kulcsa)

Kimenet: . - **T** (az átadott kulcshoz tartozó tartalom)

- 1: függvény HASÍTÓSZÓTÁR.KIOLVAS(*kulcs*)
  - 2: ha  $E[h(kulcs)] \neq \emptyset$  akkor
  - 3: vissza  $E[h(kulcs)]$
  - 4: különben
  - 5: hiba "Hibás kulcs"
  - 6: elágazás vége
  - 7: függvény vége
- 

Felhasznált változók és függvények

- **E** : Az elemeket tartalmazó tömb.
  - **h** : Tetszőleges hasítófüggvény (ahol  $h : K \rightarrow I$ ).
- 

---

### 7.10. Algoritmus Hasító függvény alapú szótárból elem törlése

---

Bemenet: *kulcs* - **K** (a törlendő elem kulcsa)

- 1: eljárás HASÍTÓSZÓTÁR.TÖRÖL(*kulcs*)

2:  $E[h(kulcs)] \leftarrow \emptyset$

- 3: eljárás vége
- 

Felhasznált változók és függvények

- **E** : Az elemeket tartalmazó tömb.
  - **h** : Tetszőleges hasítófüggvény (ahol  $h : K \rightarrow I$ ).
-

#### 7.3.4. Értékelés

A fenti műveleteknél ismét megjegyezzük, hogy ezek csak akkor működnek, ha nincsenek kulcsütközések. Két különböző, ám azonos kulcsú elem biztos nem létezik a kulcs definíciója miatt. Az azonban előfordulhat, hogy a hasítófüggvény két különböző kulcshoz is azonos indexet rendel hozzá.

Ezekben az esetekben a fenti műveletek sajnos **nem** működnek helyesen, tehát ezeket inkább csak mint egy teoretikus, ideális hasító táblázat alapú megvalósításnak tekintsük! A gyakorlatban ezek az algoritmusok nem használhatók, viszont megvalósításuk egy részét fel fogjuk tudni használni a későbbi, a kulcsütközés már támogató, fejlettebb adatszerkezeteknél.

## 7.4. Hasítófüggvény megvalósítások

### 7.4.1. Az ideális hasítófüggvény

Ha megértettük a hasító táblázatok működésének alapjait, akkor már csak az lehet a kérdés, hogy miként válasszuk meg a hasítófüggvényt. Általában jó válasz sajnos nincs erre a kérdésre, a megfelelő függvény megtalálása mindenkor mindenkor megadott feladattól függ. Egyszerűen technikai paraméterek korlátozzák a választható hasítófüggvények körét (pl. a kulcsok egész számok, szövegek, lebegőpontos számok, stb.), amelyekhez mindenkor mindenkor igazodni kell. Másrészről gyakran optimalizációval kapcsolatos döntéseket kell hoznunk a függvények, vagy azok paramétereit illetően (pl. egy kisméretű tömb használata jelentősen megnöveli az ütközések valószínűségét, túl nagy tömb azonban a tárhelyigényt fogja drasztikusan megnövelni). Tehát hasítófüggvényt csak a konkrét feladat ismeretében tudunk ajánlani.

Mindenkor mindenkor van néhány általábanos szabály, amit érdemes figyelembe venni. Ezek alapján az ideális hasítófüggvény jellemzői:

- Egyszerűen kiszámítható legyen: bár a hasítófüggvény számítása általában nagyságrendekkel kisebb erőforrásigénytelivel bír, mint a tárhelyhez való hozzáférés, de célszerű itt is optimalizálni az időigényt.
- Kevés ütközést produkáljon: tudjuk majd kezelni az ütközéseket, de ezek mindenkor mindenkor lépések, esetenkor pedig mindenkor mindenkor további tárhelyhasználatot igényelnek. Emiatt célszerű már a hasítófüggvény megválasztással minél kevesebb ütközést előidézni.
- Egyenletesen ossza szét a kulcsokat: bár nem biztos, de várhatóan kevesebb ütközést fog produkálni egy olyan függvény, ami a kulcsokat egyenletesen osztja szét az indextartományon, mint az, amelyik bizonyos tartományokba több kulcsot képez le.
- Hasonló kulcsokat véletlenszerűen, egymástól távolra szórja szét: szintén nem biztos, de általában mindenkor mindenkor kevesebb ütközésre számíthatunk, ha az egymástól csak kicsit különböző kulcsokat egymástól távolra szórja szét a hasítófüggvény. A gyakorlatban ugyanis a kulcsok eloszlása sokszor nem egyenletes, így azonban a kulcsok sűrűsödése nem okoz majd ütközéseket az indexeknél.

Hasítófüggvények esetében szokás beszélni a *lavina-hatásról* (avalanche effect) [2], ami azt jelenti, hogy egy jól megválasztott hasítófüggvény esetében, ha a bemenetet kicsit megváltoztatjuk (egy bit változás), akkor attól a kimenet jelentősen megváltozik (legalább a kimeneti bites fele változik). Ennek a tulajdonságnak köszönhetően a hasítófüggvények gyakran megjelennek egyéb feladatak megoldásaiban is (pl. kriptográfia, ellenőrző összegek generálása, stb.).

### 7.4.2. Néhány hasítófüggvény megvalósítás

Az alábbiakban néhány hasítófüggvény megvalósítást vizsgálunk meg. Ismét megjegyezzük, hogy nincs egyetemes szabály a hasítófüggvények kialakítására, az alábbiak pusztán ötletadónak tekintendők. Magát a tényleges függvényt mindenkor mindenkor a feladathoz alkalmazkodva kell elkészíteni, akár a lentiek közül választva egyet, azokat kombinálva, akár egy teljesen újszerű megoldás alkalmazásával.

A három minta esetében mindenkor mindenkor feltételezzük, hogy a kulcs egy pozitív egész szám.

#### Osztó módszer

A hasítófüggvények fontos funkciója az átalakításon és szétszóráson túl, az intervallumba leképezés. Mivel általában a kulcsok halmaza jóval nagyobb, mint a lehetséges indexek halmaza, így a hasítófüggvénynek garantálnia kell, hogy a kulcs méretétől függetlenül a kimenete mindenkor mindenkor az indexhalmaz tartományon belülre essen. Ennek a legegyszerűbb megvalósítása egy egyszerű maradékos osztás, amivel biztosítjuk, hogy a kimenet  $1..|I|$  közötti tartományba essen.

Ennek megfelelően az osztó módszer általábanos alakja:

$$h(k) : k \bmod |I| + 1 \quad (7.1)$$

Ahol

- $k$ : a kulcs, ami mindenkor mindenkor csak pozitív egész szám lehet;

- $|I|$ : az indexhalmaz mérete.

A pszeudokódok jelöléseit használva, az  $|I|$  értéke megegyezik létrehozott  $E$  tömb méretével, így biztosítható, hogy a visszaadott érték minden 1..( $E.méret$ ) tartományba essen.

Az  $I$  megválasztása alapvetően tetszőleges, nyilvánvaló, hogy nagyobb halmaz méretek általában kevesebb kulcsütözést jelentenek nagyobb tárhelyigény árán, míg kisebb  $I$  halmazok több kulcsütöközéssel járnak, viszont csökken a szükséges tömb mérete.

A lavina hatásnál láttuk, hogy a gyakorlatban olyan hasítófüggvényeket célszerű választani, amelyek a bemenet minden kisebb változtatására lehetőleg nagy kimeneti változást eredményeznek. Az osztó módszer esetében ezért gyakran nem célszerű  $I$  méretének 2 vagy 10 hatványait választani, mivel a 10 esetében a decimális számrendszerben felírt számoknak valójában csak az utolsó néhány számjegyét fogja figyelembe venni a függvény, a 2-re pedig ugyanez igaz, csak bináris számrendszerben. Ajánlásként célszerű a 2 hatványaihoz nem túl közeli prímeket választani [2].

#### Implementáció

Amennyiben olyan nyelvet használunk az implementációhoz, ahol a tömbök indexelése 0-val kezdődik, akkor értelemszerűen elhagyható a függvényben látható +1 hozzáadás. Ez igaz lesz a későbbi hasítófüggvény megvalósításokra is.

### Szorzó módszer

Véletlenszerűhöz hasonló eloszlást érhetünk el a szorzó módszer használatával. Ennek alapja:

$$h(k) : \text{KÖZEPE}_m(y * k) + 1 \quad (7.2)$$

Ahol

- $k$  : a kulcs, ami jelen esetben csak pozitív egész szám lehet;
- $y$  : egy tetszőleges konstans (szintén pozitív egész szám),
- $\text{KÖZEPE}_m(x)$ : Függvény, ami visszaadja az  $x$  szám tízes számrendszerbeli alakjának középső  $m$  darab számjegyét (pl.  $\text{KÖZEPE}_3(24356) = 435$ ).

Az  $m$  paraméterrel tudjuk meghatározni, hogy mekkora tömböt szeretnénk a háttérben használni, mivel ennek értékből adódik, hogy a függvény mekkora számokat ad vissza (1 és  $10^m$  közöttiek). Az  $y$  paraméter egy tetszőleges, általában nagy értékű konstans. Nyilvánakkora számot célszerű választani, hogy a vele való szorzás legalább  $m$  jegyű számokat eredményezzen. A speciális esetek kivédése érdekében itt is célszerű prímet választani.

Amennyiben az index halmaz méretének nem 10 hatványát szeretnénk felhasználni, akkor az így visszakapott értéket egy maradékos osztással tudjuk egy intervallumba leképezni:

$$h(k) : \text{KÖZEPE}_m(y * k) \bmod |I| + 1 \quad (7.3)$$

A módszer egy speciális alváltozata a négyzetközép módszer (a paraméterek ugyanazok, mint az előző leírásban, csak már nincs szükség az  $y$ -ra):

$$h(k) : \text{KÖZEPE}_m(k * k) \bmod |I| + 1 \quad (7.4)$$

#### Megjegyzés

A négyzetközép módszer egy olyan jó véletlenszerű eloszlást eredményez, hogy kezdetleges véletlenszám-generátoroknál is lehet használni egymás után többszöri hívással (és persze egy adott kezdőértékkel):

$$rnd_i = \text{KÖZEPE}_m(rnd_{i-1} * rnd_{i-1}) \quad (7.5)$$

## Számjegyes módszer

Szintén jó eloszlást kaphatunk, ha a kulcsokat 10-es (vagy akár más) számrendszerben leírva, azok számjegyei segítségével próbálunk meg egy indexet kiszámítani.

Erre egy egyszerű lehetőség:

$$h(k) : \sum_{i=1}^{|k|} \text{HELYIÉRTÉK}_i(k) \bmod |I| + 1 \quad (7.6)$$

Ahol

- $k$ : a kulcs, ami jelen esetben csak pozitív egész szám lehet;
- $|k|$ : a  $k$  szám számjegyeinek a száma;
- $|I|$ : a szokásos maradékos osztás paramétere, ezzel biztosítjuk, hogy a kimenet egy megadott tartományon belüli lesz;
- $\text{HELYIÉRTÉK}_i(k)$ : Visszaadja a  $k$  paraméterként átadott egész szám  $i$ . számjegyét.

A gyakorlatban ez a megoldás szintén egy véletlenszerűhöz közeli eloszlást ad. Mivel érzéketlen arra, hogy melyik szám melyik helyiértéken szerepel (ami miatt pl. számjegycserék esetén ugyanoda képez), emiatt használható az alábbi módosított változat is:

$$h(k) : \sum_{i=1}^{|k|} w_i * \text{HELYIÉRTÉK}_i(k) \bmod m + 1 \quad (7.7)$$

Ahol

- $k$ : a kulcs, ami jelen esetben csak pozitív egész szám lehet;
- $|k|$ : a  $k$  szám számjegyeinek a száma;
- $|I|$ : a szokásos maradékos osztás paramétere, ezzel biztosítjuk, hogy a kimenet egy megadott tartományon belüli lesz;
- $w_i$ : megadja az  $i$ . számjegyhez tartozó súlyozó tényezőt;
- $\text{HELYIÉRTÉK}_i(k)$ : visszaadja a  $k$  paraméter  $i$ . számjegyét.

Az alapötlet itt is ugyanaz, csak az egyes számjegyeket különböző súlyval vesszük figyelembe, attól függően, hogy melyik helyiértéken állnak.

### 7.4.3. További megfontolások

#### Szöveges típusú kulcsok

Az előző mintákban azt feltételeztük, hogy a kulcsok egész számok, a gyakorlatban azonban ez nem feltétlenül lesz mindenig igaz. A személyi számra éppen teljesül, de pl. a személyi igazolvány számra, rendszámra már nem. És persze használhatunk kulcsként neveket is, ha tudjuk, hogy azok egyediek lesznek.

Azokban az esetekben, amikor egy karaktert kell használnunk kulcsként, ehhez egy egész számot kell hozzárendelnünk, amelyet majd az előző hasítófüggvény megvalósítások közül valamelyikkel (vagy akár egy teljesen más megvalósítással) leképezzük az indexhalmazra. Maga az átalakítás sokféleképpen megoldható, egy lehetséges megoldás (*jel-kód módszer* [8]), ha az egyes szimbólumokhoz (betűkhöz) számokat rendelünk. Ilyen táblázatok már eleve léteznek (ASCII, Unicode, stb.), de a kulcs és jellemzői (pl. a használt nyelv) alapján egy saját fordítótábla is készíthető.

Amennyiben nem csak egy karakter, hanem egy hosszabb szöveg a kulcs, akkor a karakterenként kapott kódokból kell valamilyen módszerrel egy számot előállítani. Ez jelentheti az így kapott számok összeadását, helyiértékenkénti „összeragasztását”, helyiértéktől függően súlyozott összeadását, stb.

## **Univerzális hasítási technika**

Az előzőleg megismert módszerek gyakorlati használhatósága (kulcsütközések gyakorisága) nagyban függ a paraméterek megválasztásának módjától. Bár a hasító táblázatok általános esetben nagyon gyorsan tudják végrehajtani az alapműveleket, a helytelenül megválasztott paraméterekkel ez jelentősen leromlik. Ez a legrosszabb eset fordulhat elő például akkor, ha az osztó módszernél olyan  $I$  méretet határozunk meg, amivel minden kulcs osztható.

Ez ellen egy hatásos módszer az univerzális hasítási technika, ami azt jelenti, hogy a hasítófüggvényt a kulcsoktól függetlenül véletlenül választjuk ki, például az osztó módszernél az  $I$  méretét egy intervallumból választjuk ki véletlenszerűen. Az átlagos esetet ez nem befolyásolja, viszont segíthet elkerülni a legrosszabb esetet, ami előfordulhat egyszerűen véletlen folytán is, de akár tudatos támadásként is (ismerve a hasítófüggvényünk paramétereit, valaki szándékasan olyan kulcsokat adhat, amelyek sok kulcsütközet generálnak). Néha persze előfordulhat, hogy a véletlen szám pont „eltalálja” a legrosszabb esetet, de a program következő futtatásakor az már másik értékkel, jól fog működni.

## **Összetett kulcsok**

Bizonyos esetekben az eltárolni kívánt típus összetett kulcsokkal rendelkezik (pl. egy háziállat esetében a kulcs lehet a gazda személyi igazolvány száma, az állat neve és születési dátuma). Az, hogy az ilyen összetett kulcsokat hogyan lehet felhasználni egy hasító táblázatnál, mindenkor megfontolást igényel. Azt minden esetre célszerű betartani, hogy az összetett kulcs minden eleme jelenjen meg a kulcsgenerálás során.

Az egyik lehetőség a kulcs komponensek külön-külön hasítása (akár különböző függvényekkel), majd az így kapott értékek összegzése.

Egy másik jó használható módszer, ha az összetett kulcs elemeit egyszerűen egymás mögé írjuk, és az így kapott adatsort egy nagyméretű bináris számként fogjuk fel, amire már egyszerűen alkalmazhatók az eddig megismert hasítófüggvény implementációk.

## 7.5. Megvalósítás hasító táblázattal (kulcsütközések kezelésével)

### 7.5.1. Kulcsütközés kezelése

A valóságban a hasítófüggvények helyes megválasztásának módjához hasonlóan minden hasító táblázatnál ki kell alakítani egy módszert a kulcsütközések kezelésére. A *kulcsütközés* alatt azt értjük, ha két különböző kulcshoz a hasítófüggvény ugyanazt az indexet rendeli, tehát a tároláshoz használt tömb egy cellájában két elemet is el kellene tudnunk tárolni, ami nem lehetséges. Az előzőekben már beláttuk, hogy addig, amíg a kulcs halmaz nagyobb, mint az indexhalmaz, addig kulcsütközésekkel minden számolnunk kell.

Habár úgy próbáljuk megválasztani a hasítófüggvényt, hogy minél kevesebb ütközést okozzon, de ezek bármilyen ritkán fordulnak is elő, mi egyetlen ütközés miatti adatvesztést se engedhetünk meg, így mindenkorral foglalkoznunk kell ezzel a kérdéssel. A kulcsütközést tehát ne „hibaként” képzeljük el, hanem ez egyszerűen a hasító tábla adatszerkezet velejárója, amit valamilyen formában kezelünk kell.

#### Megjegyzés

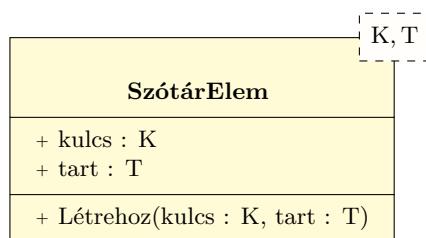
Ne zavarjon meg minket a némileg félreérthető „kulcsütközés” megnevezés. Ez nem azt jelenti, hogy vannak azonos kulcsú elemek (hiszen ezt már a kulcsok definíciójánál kizártuk), hanem azt jelenti, hogy két különböző kulcsú elemet a hasítófüggvény ugyanahoz az indexhez rendel.

Természetesen a kulcsütközést is tetszőleges módszerekkel lehet kezelni, az alábbiakban csupán ötletadóként mutatunk be néhány nevezetességes módszert.

### 7.5.2. Egy elem felépítése

Amennyiben kezelni akarjuk a kulcsütközéseket, akkor az *E* tömbben már nem csak a *tartalom*, hanem (*kulcs*, *tartalom*) értékpárokat kell tárolnunk. A közvetlen címzésnél, illetve a kulcsütközések nélküli megoldásnál ez szükségtelen volt, hiszen ha a kiolasáskor a hasító függvény által megadott helyen egy elem volt, akkor az biztos, hogy az, amit kerestünk. Most viszont már előfordulhat az is, hogy a kiolasáskor a hasító függvény által megadott helyen van ugyan elem, azonban az nem az amit keresünk (hanem egy másik kulcsú elem, amit a hasítófüggvény ugyanarra a cellára képezett le).

A szótár egyes elemeit egy összetett típussal tudjuk megvalósítani, amely a 7.5. ábrán látható.



7.5. ábra. Kulcs-tartalom párokat tartalmazó szótárelem.

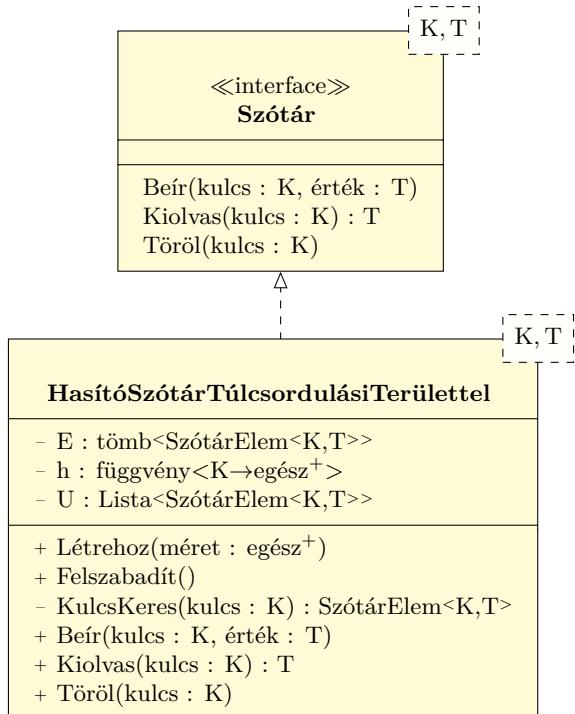
### 7.5.3. Túlcordulási terület

#### Alapvető felépítés

Az egyik legegyszerűbb megvalósítás, amely a hasító táblázatot tartalmazó tömbön túl egy további adatszerkezetet is használ. Működése megegyezik az eddig megismerttel, azonban azokban az esetekben, ahol egy hasítás kulcsütközéssel jár (tehát egy új elem beszűrása esetén a hasítófüggvény által kiszámolt indexű résben már van egy elem), az új elemet egyszerűen elhelyezi ebben a túlcordulási területben.

Értelemszerűen ennek megfelelően módosítani kell a kiolasást is: ha a hasító függvény által megadott pozícióban nincs a keresett kulcsú elem, akkor azt meg kell nézni a túlcordulási területen is.

És persze a törlésnél is figyelni kell, hogy a törlendő elem elképzelhető, hogy a túlcordulási területen van. Ilyenkor itt kell elvégezni a törlést.



7.6. ábra. Szótár megvalósítása hasító táblázattal és túlcordulási területtel.

Maga a túlcordulási terület elképzelhető egy egyszerű listaként, a fenti diagramon ez az *U* mező. Azt már az előzőekben láttuk, hogy maga a lista csak egy absztrakt fogalom, annak megvalósítása többféleképpen képzelhető el (dinamikus tömbökkel, láncolt listával, stb.).

### Kulcs alapján keresés

A szótár interfészben ugyan nem szerepel, de többször is szükségünk lesz egy olyan műveletre, ami megadott *kulcs* alapján megkeresi, hogy van-e már az adatszerkezetben olyan *SzotarElem* objektum, ami ezt a kulcsot tartalmazza. A keresésnél meg kell vizsgálnunk a tömböt, illetve a listát is. Egyik se bonyolult, de a könnyebb áttekinthetőség kedvéért ezt egy külön metódusban elhelyezni.

A keresés során figyelembe kell vennünk a túlcordulási területen lévő elemeket (7.11. algoritmus). A keresett elem itt már nem biztos, hogy benne lesz az *E* tömbben, ezért többféle lehetőséget kell kezelnünk. Látható, hogy az algoritmus a következő lehetőségeket vizsgálja át:

1. Elsöként a tömbben keressük az elemet:
  - (a) Megvizsgáljuk a a tömb *h(kulcs)* indexű elemét.
  - (b) Ha ez  $\emptyset$ , akkor biztos nincs itt a keresett kulcs.
  - (c) Ha nem  $\emptyset$ , attól még lehet, hogy egy másik kulcsú elem van itt, amit a hasító függvény ugyanide képezett le. Ezért megnézzük a kulcsot is.
  - (d) Ha a kulcs is azonos, akkor megtaláltuk a keresett elemet, ez a visszatérési érték.
2. minden egyéb esetben (ha a *h(kulcs)* indexű hely üres, vagy ott éppen egy másik elem van), akkor meg kell néznünk a túlcordulási területet is:
  - (a) Egy bejárással végignézzük a láncolt lista minden elemét.
  - (b) Ha bármelyik elem kulcsa megegyezett a keresettel, akkor a *e* értékét erre állítjuk.
  - (c) Ha a listában volt a keresett kulcs, akkor az *e* most az azt tartalmazó elemre mutat, tehát ez a visszatérési érték.
  - (d) Ha a listában nem volt benne a kulcs, akkor a *e* értéke  $\emptyset$  maradt, ez szintén helyes visszatérési érték.

Ez a bejárás így nem túl hatékony, ha készítettünk volna a láncolt listának egy keresés műveletet, akkor itt most jól tudnánk használni (de nem).

Vegyük észre az alábbiakat:

- A függvény visszatérési értéke nem csak a tartalom, hanem maga a teljes **SzótárElem** objektum. Ez azért lényeges, mert később ezt lehet, hogy módosítani is fogjuk.
- A függvény visszatérési értéke lehet  $\emptyset$  is, ha még nincs ilyen kulcs. Ez általában nem szerencsés, mivel hibákhoz vezethet, de ez egy privát metódus, csak a saját műveleteinkben fogjuk használni, ott pedig minden odafigyelünk erre a lehetőségre.

---

### 7.11. Algoritmus Kulcs keresése túlcordulási területtel rendelkező szótárban

---

Bemenet: *kulcs - K* (a keresendő kulcs)

Kimenet: . - SzótárElem<K,T> (a szótár elem ami tartalmazza a kulcsot, vagy  $\emptyset$  ha még nincs ilyen)

- 1: eljárás HASÍTÓSZÓTÁRTÚLCORDULÁSITERÜLETTEL.KULCSKERES(*kulcs*)
- 2:   ha  $E[h(\text{kulcs})] \neq \emptyset \wedge E[h(\text{kulcs})].\text{kulcs} = \text{kulcs}$  akkor
- 3:     vissza  $E[h(\text{kulcs})]$
- 4:   különben
- 5:      $e \leftarrow \emptyset$
- 6:      $U.\text{BEJÁR}(x \rightarrow \text{ha } x.\text{kulcs} = \text{kulcs} \text{ akkor } e \leftarrow x)$
- 7:     vissza  $e$
- 8:   elágazás vége
- 9: eljárás vége

---

#### Felhasznált változók és függvények

- $e$  : Egy **SzótárElem<K,T>** típusú segédváltozó.
- $E$  : Az elemeket tartalmazó tömb.
- $U$  : Egy lista adatszerkezet, ami a túlcordulási terület.
- $h$  : Tetszőleges hasítófüggvény (ahol  $h : K \rightarrow I$ ).

---

### Beírás művelet

A beszűrást mutatja be a 7.12. algoritmus. Elsőként a már elkészített kulcs alapú kereséssel megvizsgáljuk, hogy van-e már ilyen kulcsú elem az adatszerkezetben. Ha van, akkor a függvény visszatérési értéke az ezt tároló **SzótárElem** objektum lesz (mindegy, hogy ez a tömbben, vagy a láncolt listában van). Ebben az esetben ennek a tartalmi részét átírjuk, és ezzel végeztünk is.

Ha nincs, akkor keresünk egy új helyet az új elemnek. Elsőként megpróbáljuk a tömbben: a paraméterként kapott beszűrandó elem kulcsára meghívjuk a hasítófüggvényt, ez alapján megkapjuk az indexet, hogy hova kellene elhelyezni azt a tömbben. A következő sorban következik az ellenőrzés, hogy ez a hely még szabad-e. Ha igen, akkor ide bemásoljuk az új elemet, ha pedig nem, akkor beszúrjuk a túlcordulási területre.

Mivel a túlcordulási területet listaként képzeljük el, ezért ott egyszerűen használjuk a már tárgyalt műveleteket. Mivel nincs megkötés az elemek sorrendjére, célszerűnek tűnik egy láncolt listát létrehozni és minden az elejére beszúrni az elemeket.

### Kulcs alapján kiolvasás

A kulcs alapján kiolvasás nagyon egyszerű, mivel a keresést már egyszer megvalósítottuk:

1. Amennyiben a keresés egy valódi **SzótárElem** objektumot ad vissza, akkor ennek a tartalma lesz a kiolvasás visszatérési értéke.
2. Amennyiben a keresés egy  $\emptyset$  értéket ad vissza, akkor kivételt dobunk, mivel a megadott kulcsú elem nem szerepel az adatszerkezetben (sem a tömbben, sem pedig a láncolt listában).

---

## 7.12. Algoritmus Beírás túlcordulási területtel rendelkező szótárba

---

Bemenet: *kulcs* - **K** (a tároláshoz használatos kulcs)

Bemenet: érték - **T** (a tárolandó érték)

```
1: eljárás HASÍTÓSZÓTÁRTÚLCORDULÁSITERÜLETTEL.BEÍR(kulcs, érték)
2: meglévő ← KULCSKERES(kulcs)
3: ha meglévő ≠ ø akkor
4: meglévő.tart ← érték
5: különben
6: új ← LÉTREHOZ(SzótárElem<K, T>) {kulcs ← kulcs, tart ← érték}
7: ha E[h(kulcs)] = ø akkor
8: E[h(kulcs)] ← új
9: különben
10: U.HOZZÁFÚZ(új)
11: elágazás vége
12: elágazás vége
13: eljárás vége
```

---

Felhasznált változók és függvények

- *meglévő*, *új* : Egy SzótárElem<*K*, *T*> típusú segédváltozók.
  - *E* : Az elemeket tartalmazó tömb.
  - *U* : Egy lista adatszerkezet, ami a túlcordulási terület.
  - *h* : Tetszőleges hasítófüggvény (ahol  $h : K \rightarrow I$ ).
- 

---

## 7.13. Algoritmus Kiolvasás túlcordulási területtel rendelkező szótárban

---

Bemenet: *kulcs* - **K** (a kiolvasandó tartalomhoz tartozó kulcs)

Kimenet: . - **T** (a megadott kulcshoz tartozó tartalom)

```
1: függvény HASÍTÓSZÓTÁRTÚLCORDULÁSITERÜLETTEL.KIOLVAS(kulcs)
2: meglévő ← KULCSKERES(kulcs)
3: ha meglévő ≠ ø akkor
4: vissza meglévő.tart
5: különben
6: hiba "Hibás kulcs"
7: elágazás vége
8: függvény vége
```

---

Felhasznált változók és függvények

- *meglévő* : Egy SzótárElem<*K*, *T*> típusú segédváltozó.
  - *KULCSKERES(*kulcs*)* : Egy segédművelet, ami megkeresi a megadott kulcsot tartalmazó elemet.
-

## Kulcs alapján törlés

A törlés az elem helyétől függően történhet a tömbből, vagy a túlcsordulási területből is. Látható, hogy maga a törlés kódja nagyrészt megegyezik a keresés kódjával, hiszen itt is azt kell először tisztáznunk, hogy a keresett kulcs benne van-e az adatszerkezetben, és ha igen, akkor hol? Ezt követően már csak a megfelelő helyről törölünk kell az elemet.

A törlést mutatja be a 7.14. algoritmus.

---

### 7.14. Algoritmus Törlés túlcsordulási területtel rendelkező szótárból

---

**Bemenet:** *kulcs - K* (a törlendő elem *kulcsa*)

- 1: eljárás HASÍTÓSZÓTÁRTÚLCSORDULÁSITERÜLETTEL.TÖRÖL(*kulcs*)
  - 2:   ha  $E[h(\text{kulcs})] \neq \emptyset \wedge E[h(\text{kulcs})].\text{kulcs} = \text{kulcs}$  akkor
  - 3:     FELSZABADÍT( $E[h(\text{kulcs})]$ )
  - 4:      $E[h(\text{kulcs})] \leftarrow \emptyset$
  - 5:   különben
  - 6:      $e \leftarrow \emptyset$
  - 7:      $U.\text{BEJÁR}(x \rightarrow \text{ha } x.\text{kulcs} = \text{kulcs} \text{ akkor } e \leftarrow x)$
  - 8:     ha  $e \neq \emptyset$  akkor
  - 9:        $U.\text{TÖRÖL}(e)$
  - 10:      FELSZABADÍT( $e$ )
  - 11:     elágazás vége
  - 12:   elágazás vége
  - 13: eljárás vége
- 

**Felhasznált változók és függvények**

- *e* : Egy *SzótárElem<K, T>* típusú segédváltozó.
  - *E* : Az elemeket tartalmazó tömb.
  - *U* : Egy lista adatszerkezet, ami a túlcsordulási terület.
  - *h* : Tetszőleges hasítófüggvény (ahol  $h : K \rightarrow I$ ).
- 

Megjegyzés

A törlés megvalósítható úgy is, hogy ha az *E* tömbből törölünk, akkor átmásolja ide a túlcsordulási területen esetleg meglévő első olyan elemet, amelyiknek szintén itt lenne a helye. Ezzel gyorsítható lenne a keresés, hiszen az  $E[i] = \emptyset$  esetben már nem is kell megnéznünk a túlcsordulási területet, hiszen biztos, hogy nincs adott kulcsú elemünk.

## A módszer értékelése

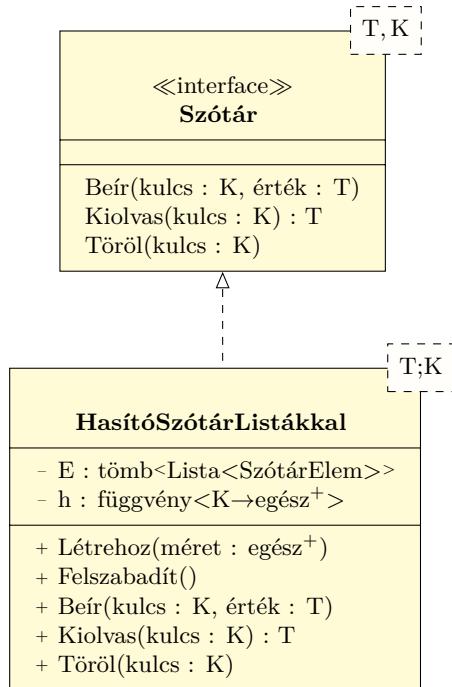
A módszer első ránézésre nem tűnik hatékonynak, hiszen a túlcsordulási területen való munka meglehetősen erőforrás-igényes. Érdemes azonban figyelembe venni, hogy ideális esetben egy jól megválasztott hasítófüggvény már eleve kevés kulcsütközést fog produkálni, így remélhetőleg kevés elemmel kell majd a túlcsordulási területen dolgozni. Nagy adatmennyiség esetén itt is célszerű valamilyen optimalizált megoldást keresni.

### 7.5.4. Listák használata

#### Alapvető felépítés

Ebben az esetben nem egy, hanem több lista adatszerkezetet használhatunk az adatok tárolására (lásd 5. fejezet). Az *E* tömböt úgy tekintjük, hogy minden eleme egy *Lista<SzótárElem<K, T>>*. A hasítófüggvény kiszámítása azonos az eddig tanultakkal, ennek eredményeképpen kapunk egy indexet. Az elemeket azonban nem közvetlenül a tömbbe helyezzük el az *E[i]* résbe, ott ugyanis csak egy lista adatszerkezet van, ebbe kell felvennünk az új elemet.

A lista megvalósítását tekintve választhatjuk bármelyik már ismert implementációt: dinamikus tömb (4.4. fejezet) vagy láncolt lista (5.2. fejezet).



7.7. ábra. Szótár megvalósítása hasító táblázattal és listákkal.

Ennek megfelelően a különféle műveletek.

- **BEÍR(kulcs,érték):** Az  $E[h(kulcs)]$  helyen található listába beírjuk az új elemet.
- **KIOLVAS(kulcs):** Az  $E[h(kulcs)]$  helyen található listából megkeressük az adott kulcsú elemet.
- **TÖRÖL(kulcs):** Az  $E[h(kulcs)]$  helyen található listából töröljük az adott kulcsú elemet.

A műveletek egyszerűsége miatt ezeket nem részletezzük a továbbiakban.

## A módszer értékelése

Ez a megoldás sok szempontból ideális lehet, főleg azért, mivel így ki tudjuk használni a listák dinamikus mivoltát (vagyis a méretük futásidőben változhat, attól függően, hogy éppen hány elemet kell ténylegesen eltárolnunk). Amennyiben a kulcsok egyenletesen helyezkednek el, akkor a keresés se lesz drasztikusan rosszabb, hiszen ideális esetben az egyes láncolt listákban csak néhány lépést kell megtennünk, ellentétben a túlcordulási terüettel, ahol egy helyen gyűjtjük az összes kulcsüközést okozó elemet.

A túlcordulási területtel összehasonlítva az a fő különbség, hogy ott egy nagy listában tároltuk az összes el nem helyezett elemet, itt pedig ezeket több kisebb listába helyezzük el. Mivel a listákból való keresés a benne lévő elemek számával lineárisan növekszik, így látható, hogy ez utóbbival jobb futásidőt érhetünk el. Ennek persze az a feltétele, hogy az elemeket egyenletesen tudjuk szétszórni az egyes listák között.

### 7.5.5. Nyílt címzés

#### Alapvető felépítés

A nyílt címzés ötlete az, hogy abban az esetben, ha látjuk, hogy a beszúrandó elem kulcsához tartozó index terület már foglalt a tömbben, akkor valamilyen szabályos módon elkezdünk egy új területet keresni. Például, ha a hasítófüggvény által visszaadott  $k$ . rés már foglalt, akkor megpróbálhatjuk a  $k+1, k+2, \dots$  cellákat is megvizsgálni (ügyelve persze arra, hogy a tömb túlcímzése esetén az első indextől folytatjuk a keresést). Mindezt addig folytatjuk, amíg nem találunk egy üres (vagy törlt, erre még visszatérünk) pozíciót, ide elhelyezzük a beszúrandó elemet (7.16. algoritmus). Másik lehetőség, ha a tömb minden elemét átnéztük, és nincs szabad hely, ilyenkor hibát jelzünk.

Ehhez bevezethetünk egy második paramétert a hasító függvénynél, amely a próbálkozások számát mutatja. Erre mutat példát a 7.8. egyenlet, ahol az egyparaméteres  $h$  függvény egy hagyományos hasító függvény, a kétparaméteres a kulcsütközést is kezelő kibővített változat, a  $k$  a kulcs, a  $j$  a próbálkozás száma, az  $|I|$  pedig a tömb mérete.

$$h(k, j) : (h(k) + j) \bmod |I| + 1 \quad (7.8)$$

A megoldás hátránya, hogy néha az egyébként kulcsütközést nem okozó elemeket se tudjuk berakni a helyükre, mert azt már előzőleg elfoglalta egy, a kulcsütközés miatt az eredeti helyéről elcsúsztatott elem.

Előnye viszont az, hogy az adatszerkezet nem igényel semmilyen kiegészítő tárhelyet, sem listákat, sem túlcordulási területet. Az elemek minden a már ismert  $E$  tömbben kerülnek eltárolásra.

### Kulcs alapján keresés

A túlcordulási területhez hasonlóan, itt is készítünk egy privát segédfüggvényt a kulcsok kereséséhez. A 7.15. algoritmus ciklusa elkezdi keresni a kulcsot a tömbben. A hasító függvényt addig hívjuk meg egyre nagyobb  $j$  értékű paraméterekkel, amíg:

- megtaláljuk a keresett kulcsot tartalmazó elemet a tömbben;
- vagy a  $j$  értéke eléri a tömb méretét, ami azt jelzi, hogy már minden lehetőséget végignéztük, tehát nincs meg a keresett kulcs;
- vagy találunk egy üres elemet, ami szintén azt jelzi, hogy nem lehet benne a kulcs a tömbben (a törölt elemekre ez viszont nem vonatkozik!).

Előző esetben a függvény visszatérési értéke mag a **SzótárElem** objektum, utóbbi esetekben pedig egy  $\emptyset$  érték. Ez utóbbit sem hiba, ezért nem dobunk kivételt, ezzel jelezük, hogy az adott kulcs nem szerepel a szótárban.

---

#### 7.15. Algoritmus Keresés nyílt címzéssel kezelt szótárból

---

**Bemenet:**  $kulcs - K$  (a kiolasztandó elem kulcsa)

**Kimenet:** . - **SzótárElem** $<K, T>$  (a szótár elem ami tartalmazza a kulcsot, vagy  $\emptyset$  ha még nincs ilyen)

1: **függvény** HASÍTÓSZÓTÁRNÝÍLT CÍMZÉSSEL.KULCSKERES( $kulcs$ )

```

2: $j \leftarrow 0$
3: ciklus amíg $j < E.méret \wedge E[h(kulcs, j)] \neq \emptyset \wedge E[h(kulcs, j)].kulcs \neq kulcs$
4: $j \leftarrow j + 1$
5: ciklus vége
6: ha $j < E.méret \wedge E[h(kulcs, j)].kulcs \neq \emptyset$ akkor
7: vissza $E[h(kulcs, j)]$
8: különben
9: vissza \emptyset
10: elágazás vége
11: függvény vége
```

---

#### Felhasznált változók és függvények

- $j$  : Egy **egész<sup>+</sup>** segédváltozó, ami a próbálkozásokat számolja.
  - $E$  : Az elemeket tartalmazó tömb.
  - $h$  : Tetszőleges hasítófüggvény (ahol  $h : K \rightarrow I$ ).
- 

### Új elem felvétele

Elsőként megnézzük az előzőleg elkészített kereséssel, hogy van-e már ilyen kulcsú elem a tömbben. Ha igen, akkor csak átírjuk annak a tartalmát a paraméterként kapott értékre.

Ha nincs, akkor a keresésnél már látott módon megkeressük az első üres helyet. Fontos különböszég azonban, hogy itt nem csak az üres, hanem a törölt helyeknél is meg kell állni, hiszen minden két lehetőség jó helyet ad az új elem felvételére.

---

### 7.16. Algoritmus Beírás nyílt címzéssel használt szótárba

---

Bemenet: *kulcs* - **K** (a tároláshoz használatos *kulcs*)

Bemenet: érték - **T** (a tárolandó érték)

```
1: eljárás HASÍTÓSZÓTÁRNÝÍLT CÍMZÉSSEL. BEÍR(kulcs, érték)
2: meglévő ← KULCSKERES(kulcs)
3: ha meglévő ≠ ø akkor
4: meglévő.tart ← érték
5: különben
6: j ← 0
7: ciklus amíg $j < E.\text{méret} \wedge E[h(\text{kulcs}, j)] \neq \emptyset \wedge E[h(\text{kulcs}, j)] \neq \otimes$
8: j ← j + 1
9: ciklus vége
10: ha $j < E.\text{méret}$ akkor
11: új ← LÉTREHOZ(SzótárElem<K, T>) {kulcs ← kulcs, tart ← érték}
12: $E[h(\text{kulcs}, j)] \leftarrow \text{új}$
13: különben
14: hiba "Nincs hely"
15: elágazás vége
16: elágazás vége
17: eljárás vége
```

---

#### Felhasznált változók és függvények

- *j* : Egy *egész*<sup>+</sup> segédváltozó, ami a próbálkozásokat számolja.
  - *új* : Egy *SzótárElem<K, T>* típusú segédváltozó.
  - *E* : Az elemeket tartalmazó tömb.
  - *h* : Tetszőleges hasítófüggvény (ahol  $h : K \rightarrow I$ ).
  - $\otimes$  : Speciális jel, amivel a törölt állapotot jelöljük.
  - *KULCSKERES(kulcs)* : Egy segédművelet, ami megkeresi a megadott kulcsot tartalmazó elemet.
-

## Kulcs alapján kiolvasás

A kiolvasás itt is nagyon egyszerű, hiszen az ehhez szükséges keresést már elkészítettük. A 7.17. algoritmus alapján csak annyit kell megvizsgálnunk, hogy mit adott vissza a keresés.

---

### 7.17. Algoritmus Keresés nyílt címzéssel kezelt szótárból

---

**Bemenet:** *kulcs - K* (a kiolvasandó elem *kulcsa*)

**Kimenet:** . - *T* (a *kulcshoz tartozó tartalom*)

```
1: függvény HASÍTÓSZÓTÁRNÝÍLT CÍMZÉSSEL.KIOLVAS(kulcs)
2: meglévő ← KULCSKERES(kulcs)
3: ha meglévő ≠ ø akkor
4: vissza meglévő.tart
5: különben
6: hiba "Hibás kulcs"
7: elágazás vége
8: függvény vége
```

---

#### Felhasznált változók és függvények

- *meglévő* : Egy *SzótárElem<K, T>* típusú segédváltozó.
  - *KULCSKERES(kulcs)* : Egy segédművelet, ami megkeresi a megadott kulcsot tartalmazó elemet.
- 

## Kulcs alapján törlés

A pszeudokódban is látható, hogy be kellett vezetnünk a már meglévő üres ( $\emptyset$ ) mellett egy új, törölt ( $\otimes$ ) jelölést. Ennek szerepe a törlés alapján érhető meg. A törlés egyébként egészen hasonló, mint a keresés, csak a végén nem visszaadja a megtalált elemet, hanem beállítja azt töröltként (7.18. algoritmus).

Azért lényeges, hogy ne üres értéket írjon be, mivel elképzelhető, hogy előzőleg volt olyan beszúrás, aminek elvileg a most törölt elem helyére kellett volna elhelyeznie az elemet, de a foglalt részek miatt csak a törölt után tudta azt elmenteni. Viszont ha most a törölt cellát üresnek állítanánk be, akkor a keresés itt minden megállna, így nem találná meg ezt az elemet.

---

### 7.18. Algoritmus Törlés nyílt címzéssel használt szótárból

---

**Bemenet:** *kulcs - K* (a törlendő elem *kulcsa*)

```
1: eljárás HASÍTÓSZÓTÁRNÝÍLT CÍMZÉSSEL.TÖRÖL(kulcs)
2: j ← 0
3: ciklus amíg j < E.méret ∧ E[h(kulcs, j)] ≠ ø ∧ E[h(kulcs, j)] ≠ kulcs
4: j ← j + 1
5: ciklus vége
6: ha j < E.méret ∧ E[h(kulcs, j)] ≠ ø akkor
7: FELSZABADÍT(E[h(kulcs, j)])
8: E[h(kulcs, j)] ← \otimes
9: különben
10: hiba "Hibás kulcs"
11: elágazás vége
12: eljárás vége
```

---

#### Felhasznált változók és függvények

- *j* : Egy *egész<sup>+</sup>* segédváltozó, ami a próbálkozásokat számolja.
  - *E* : Az elemeket tartalmazó tömb.
  - *h* : Tetszőleges hasítófüggvény (ahol  $h : K \rightarrow I$ ).
  - $\otimes$  : Speciális jel, amivel a törölt állapotot jelöljük.
-

### Megjegyzés

Nem biztos, hogy a törölt jelet az elemekkel azonos típussal meg tudjuk valósítani. Ilyenkor egyszerűen felvethetünk egy logikai tömböt a hasító táblázatban, ami azt jelzi, hogy megadott rés törölt-e.

## Kipróbálási stratégiák

Az előbb említett módszer, hogy az egymás utáni elemeket vizsgáljuk, meglehetősen egyszerű volt. Emellett azonban többféle stratégiát alkalmazhatunk arra, hogy hogyan határozzuk meg az egymás után vizsgálandó cellákat:

- Lineáris próba:

$$h(k, j) : (h(k) + zj) \bmod |I| + 1 \quad (7.9)$$

Ahol  $z$  egy tetszőleges konstans, a  $j$  pedig a probálkozások száma (0-tól kezdve). Ezt mutatja be a bevezetőben szereplő példa is  $z = 1$  választással (bár magát a képletet ott nem láthatjuk így egyben, hiszen egyesével növelgetés esetén felesleges lenne minden esetben újra kiszámolni a  $h(k)$  értéket), tehát a szabad helyek keresése közben minden a következő réseket kezdjük el nézegetni. Érdemes észrevenni a maradékos osztás szerepét, ezzel biztosítjuk azt, hogy ha esetleg már túlcímezünk a tömböt, akkor a függvény visszatérési értéke vissza fog ugrani 0-ra. Hátránya, hogy gyakran csoportosuláshoz vezet<sup>1</sup>, amikor egyre hosszabb telített szakaszok jönnek létre a tömbben, és ide egyre nehezebb lesz beilleszteni elemeket. Tovább nehezíti a helyzetet, hogy a törlések sem segítenek a sebesség növekedésben, hiszen a keresés a törölt elemeken is minden túl fog lépni. Természetesen nem csak 1-el növelhetjük az értéket, de az eredmény más esetekben is hasonló lesz.

- Négyzetes próba:

$$h(k, j) : (h(k) + z_1 j + z_2 j^2) \bmod |I| + 1 \quad (7.10)$$

Ahol  $z_1$  és  $z_2$  tetszőleges konstansok, a  $j$  pedig a probálkozások száma (0-tól kezdve). Tehát minden próbálkozással egyre nagyobb távolságokban keresünk új szabad rést, ami jó hatással van a csoportosulások elkerülésére (ebben az esetben kevésbé, illetve más jelleggel jelentkezik).

Ezeken túl természetesen tetszőleges függvény használható a következő elem indexének meghatározására.

### A módszer értékelése

A módszer egyik nagy előnye az egyszerűsége, illetve az, hogy nem igényel különböző külső adatszerkezeteket a kulcsütöközésben érintett elemek tárolásához. Ez azt is eredményezi, hogy a hasító táblázatunk tárhelyét jobban ki tudjuk használni még kulcsütöközések esetén is. A hátránya az, hogy nagyszámú kulcsütöközés során az elemek elkezdenek eltávolodni a nekik szánt helyéről és ez lelassíthatja a keresési műveletet.

#### 7.5.6. Többszörös hasítás

##### Alapvető felépítés

A többszörös hasítás egy általábanosan jól használható technika. Lényege az, hogy nem egy darab, hanem kettő, vagy akár tetszőleges számú hasító függvényteljesítője gondolni, egyszerűen megoldható az is, hogy minden elemet a hasítófüggvényt használjuk, csak különböző paraméterekkel. Tehát pl. a szorzó módszer esetében az előzőleg használt  $y$  szorzó konstans lehet a probálkozások számának ( $j$ ) a függvénye:

$$h_j(k) : \text{KÖZEPE}_m(y * j * j * k) + 1 \quad (7.11)$$

<sup>1</sup>clustering

A keresésnek nyilvánvalóan ugyanezt az utat kell bezárnia, tehát az egymást követő hasítófüggvényeket addig kell egymás után alkalmazni, amíg valamelyikkel meg nem találjuk a keresett elemet (vagy el nem tudjuk dönten, hogy az biztosan nincs benne a hasító táblázatban).

### **A módszer értékelése**

A beírás/kiolvasás/törles pszeudokódokat külön nem tekintjük át, az alaplev teljesen megegyezik a nyílt címzésnél látottal. Sőt, magát a többszörös hasítást tekinthetjük akár a nyílt címzés általánosításának is. Csak annyi a különbség, hogy nem egy egyszerű összeadással változtatjuk folyamatosan a vizsgált indexet, hanem minden az újabb hasítófüggvény meghívásával kapjuk meg a következő vizsgálandó címét.

## **III. rész**

# **Megoldási stratégiák**

## 8. fejezet

# Alapvető Optimalizációs Stratégiák

### 8.1. Alapfogalmak

#### 8.1.1. Optimalizáció

Számos módszert ismerünk már különféle jellegű feladatok megoldására. Eddig általában megelégedtünk azzal, hogy egy olyan algoritmust készítsünk, ami megoldást ad egy megadott problémára (pl. megkeresi egy tömb legkisebb elemét, stb.). Az optimalizációs feladatok esetében azonban feltételezhetjük, hogy több megoldás is létezik (persze az is lehet, hogy csak egy, vagy akár egy sincs), és ezek közül kell kiválasztanunk egy optimálisat. Ezzel kapcsolatban néhány megjegyzés:

- Feladattól függően nekünk kell meghatározni, hogy mit értünk optimális alatt (pl. tárgyak közüli választás során célunk lehet a minél olcsóbb megoldás keresése, de akár a minél jobb, minél könyebb, stb. is). Továbbra is szeretnénk minél általánosabb algoritmusokat készíteni, így az optimális fogalmát nem szeretnénk belekódolni az algoritmusokba, ezt inkább egy külső jóság/fitnessz függvény segítségével határozzuk meg. Ez a függvény minden lehetséges megoldáshoz hozzárendel egy szám értéket, ami minél nagyobb, annál jobb a megoldás.
- Bizonyos feladatoknál nem jóság, hanem költség függvényt határozzunk meg. Ebben az esetben a cél a minél kisebb költségű megoldás megkeresése, ez általában csak apró módosításokat igényel az algoritmusokban.
- Szándékosa nem „az optimálisról” beszélünk, hiszen gyakori, hogy a sok lehetséges megoldás közül nem emelhető ki egyetlen darab legjobb, hanem akár több megoldás is található, amelyek jósága maximális.

#### 8.1.2. A 0-1 hátizsák probléma

A fejezetben többféle módszerrel próbáljuk megoldani ugyanazt a feladatot. Ez az úgynevezett hátizsák probléma<sup>1</sup> lesz, amelynek bemenete:

- $n$  darab tárgy, amelyek jellemzői:
  - $w_i$  - az  $i$ . tárgy súlya ( $1 \leq i \leq n$ ), típusa  $\text{egész}^+$ ,
  - $p_i$  - az  $i$ . tárgy értéke ( $1 \leq i \leq n$ ), típusa  $\text{szám}^+$ ;
- egy hátizsák, aminek maximális kapacitása  $W_{\max}$ , tehát a benne elhelyezett tárgyak összsúlya nem lehet ennél több (szintén egy  $\text{szám}^+$  konstans).

Pakolásnak nevezzük annak meghatározását, hogy mely tárgyak kerülnek a hátizsákba, és melyek nem. Konkrét programozási nyelven a pakolást többféle adatszerkezet reprezentálhatja, pl. az elhelyezett tárgyak halmaza, stb. Mi egy  $n$  elemű logikai tömböt fogunk használni, amelynek  $i$ . eleme ( $1 \leq i \leq n$ )

---

<sup>1</sup>knapsack problem

azt mutatja, hogy az  $i$ . elem a megadott pakolás szerint bekerült-e a zsákba vagy sem (*igaz* érték esetén bekerült, *hamis* érték esetén nem).

Egy pakolás összsúlya az általa a zsákba helyezett tárgyak súlyának az összege. Hasonlóan egy pakolás összértéke az általa zsákba helyezett tárgyak értékének az összege.

Érvényes pakolásnak nevezzük azokat a pakolásokat, amelyek megfelelnek a maximális kapacitás feltételnek. Tehát a pakolás összsúlya nem nagyobb, mint a hátizsák kapacitása ( $W_{\max}$ ).

Optimális pakolásnak nevezzük azokat az érvényes pakolásokat, amelyeknél nagyobb összértékkal bíró érvényes pakolás nem állítható elő.

Feladatunk annak meghatározása, hogy mekkora egy optimális pakolásnak az összértéke ( $p_{opt}$ ). Tehát a pakolás által a zsákba helyezendő tárgyak értékének az összege. Ez értelemszerűen minden optimális pakolásnál ugyanannyi.

#### Megjegyzés

Tehát most nem ragaszkodunk ahhoz, hogy az elhelyezendő tárgyak listáját is megkapjuk eredményként. Ez persze bizonyos esetekben szükséges lehet, de az egyszerűség kedvéért mi csak az optimális összértékre összpontosítunk.

A hátizsák problémának két al változatával foglalkozunk majd:

- 0-1 hátizsák probléma: a tárgyat itt atominak tekintjük (tipikusan pl. pohár, villa, stb.). minden tárgyról azt kell tehát eldöntenünk, hogy az bekerül-e a zsákba vagy sem.
- Töredékes<sup>2</sup> hátizsák probléma: ebben az esetben arra is van lehetőségünk, hogy az egyes tárgyaknak csak egy részét helyezzük a hátizsákba (tipikusan pl. víz, liszt, stb.).

Mi alapvetően a 0-1 hátizsák problémát szeretnénk megoldani.

Két segédműveletet bevezetünk már a feladat megoldása előtt, ezek kiszámítják egy pakolás összértékét (8.1. algoritmus), illetve összsúlyát (8.2. algoritmus).

### 8.1. Algoritmus Egy pakolás összértékének kiszámítása

Bemenet:  $X$  - tömb<logikai> (pakolás adatai, ahol  $X[i]$  mutatja, hogy az  $i$ . tárgy benne van-e)

Kimenet:  $s$  - szám<sup>+</sup> (a pakolás összértéke)

```
1: függvény ÖSSZÉRTÉK(X)
2: $s \leftarrow 0$
3: ciklus $i \leftarrow 1$ -től n -ig
4: ha $X[i] = \text{igaz}$ akkor
5: $s \leftarrow s + p_i$
6: elágazás vége
7: ciklus vége
8: vissza s
9: függvény vége
```

#### Felhasznált változók és függvények

- $s$  : Egy szám<sup>+</sup> típusú segédváltozó.
- $i$  : Egy egész<sup>+</sup> típusú segédváltozó.
- $n$  : A hátizsákba pakolható tárgyak száma.
- $p_i$  : A hátizsákba pakolható  $i$ . tárgy hasznossága ( $1 \leq i \leq n$ ).

<sup>2</sup>fractional

---

## 8.2. Algoritmus Egy pakolás összsúlyának kiszámítása

---

**Bemenet:**  $X$  - tömb<logikai> (pakolás adatai, ahol  $X[i]$  mutatja, hogy az  $i$ . tárgy benne van-e)

**Kimenet:** . - szám<sup>+</sup> (a pakolás összsúlya)

```
1: függvény ÖSSZSÚLY(X)
2: $s \leftarrow 0$
3: ciklus $i \leftarrow 1$ -től n -ig
4: ha $X[i] = \text{igaz}$ akkor
5: $s \leftarrow s + w_i$
6: elágazás vége
7: ciklus vége
8: vissza s
9: függvény vége
```

---

### Felhasznált változók és függvények

- $s$  : Egy szám<sup>+</sup> típusú segédváltozó.
  - $i$  : Egy egész<sup>+</sup> típusú segédváltozó.
  - $n$  : A hátizsákba pakolható tárgyak száma.
  - $w_i$  : A hátizsákba pakolható  $i$ . tárgy súlya ( $1 \leq i \leq n$ ).
-

## 8.2. Nyers erő módszere

### A módszer bemutatása

A nyers erő módszere<sup>3</sup> számos egyéb néven ismert: „teljes kipróbálás módszere”, „generálj és tesztelj”, „leszámlálási eljárás” stb. Ezek a nevek mind jól jellemzik a módszert, ami a maga tökéletlenségében nagyon könnyen alkalmazható, bizonyos feladat típusoknál talán egyedülálló módon, azonban az esetek túlnyomó többségében drasztikus erőforrásigényel bír.

A módszer használatának tulajdonképpen csak egy alapkövetelménye van: a lehetséges megoldásjelöltek<sup>4</sup> száma legyen véges. Maga az alapelt pedig ez alapján már egyszerű: szisztematikusan próbálgassuk végig az összes megoldásjelöltet. Amennyiben csak egy megoldásra van szükségünk, akkor annak megtalálásáig, amikor viszont optimalizálni szeretnénk, akkor minden (így már érhető, hogy miért ragaszkodunk ahoz, hogy ezek száma véges legyen).

A módszer általában egy ciklust feltételez, ami az alábbi lépéseket hajtja végre:

1. Következő megoldásjelölt kiválasztása vagy generálása.
2. Ha szükséges, akkor a megoldásjelölt vizsgálata, hogy elfogadható-e (ez elhagyható ha minden csak megfelelő megoldásokat választunk ki az előző lépésben).
3. Amennyiben ez valódi megoldása a problémának, akkor
  - keresés esetén kilépünk a ciklusból, hiszen megvan a keresett megoldás;
  - optimalizáció esetén ellenőrzük, hogy ennek a jósa a nagyobb-e, mint az eddig talált legjobb megoldás: ha igen, (vagy ez volt az első megoldás) akkor ezt követően ezt tekintjük legjobbnak.
4. Amennyiben van még meg nem vizsgált megoldásjelölt, akkor folytatjuk a ciklust.

Ezt a működést írja le a 8.3. algoritmus.

Mivel egy általános feladatmegoldási módszerről van szó, ezért a feladat specifikációját külső paramétereken/függvényeken keresztül kell megadnunk. Egyrészt feltételezzük, hogy a megoldásjelöltek minden  $T$  típusúak. Ezekhez pedig kapcsolódnak az alábbiak:

- **m**: Az összes lehetséges megoldásjelölt száma. Mint volt róla szó, a módszer csak olyan esetekben használható optimalizációra, ahol a megoldások száma véges. Ez lehet például egy jelszó törésnél az összes lehetséges jelszó száma (ha tudjuk, hogy az maximálisan milyen hosszú lehet).
- **GENERÁTOR(*i*) : T** : Megadja az *i*. lehetséges megoldást. Ez kétféleképpen lehetséges:
  - Egy előre felsorolt listában vannak a lehetséges megoldások, pl. egy HR rendszerben egy adatbázisban látjuk az összes jelentkező adatát, és ezek közül kell kiválasztani valakit, aki a legalkalmasabb a feladatra.
  - Ugyan véges számú lehetséges megoldásunk van, de ezeket futásidőben generáljuk, pl. egy jelszó feltörő rendszerben felesleges lenne egy adatbázisban eltárolni minden lehetséges jelszót. Egyszerűbb ezeket futásidőben generálni az *i*. ismeretében.

A nyers erő módszeren alapuló megoldások során egy kritikus lépés, hogy létrehozzunk egy leképezést, ami a természetes számokhoz hozzá tudja rendelni lehetséges megoldásokat.

- Egy jelszó törés esetén ez lehet például az alábbi: **GENERÁTOR(1) = "a"**, **GENERÁTOR(2) = "b"**, ... , **GENERÁTOR(26) = "z"**, **GENERÁTOR(27) = "ab"**, **GENERÁTOR(28) = "ac"**, ...
- Ha az utazó ügynök problémát akarjuk megoldani 4 város esetén (A, B, C, D), akkor egy lehetséges megoldás: **GENERÁTOR(1) = "ABCD"**, **GENERÁTOR(2) = "ABDC"**, **GENERÁTOR(3) = "ACBD"**, **GENERÁTOR(4) = "ACDB"**, ...
- A 0-1 hátzsák problémára a lehetséges megoldások generálását hamarosan részleteiben is tárgyaljuk.

<sup>3</sup>brute force

<sup>4</sup>candidate solution

8.1. táblázat. Lehetséges pakolások három tárgy esetén.

| i | pakolás |       |       | bináris számként |
|---|---------|-------|-------|------------------|
|   | $t_1$   | $t_2$ | $t_3$ |                  |
| 0 | h       | h     | h     | 000              |
| 1 | h       | h     | i     | 001              |
| 2 | h       | i     | h     | 010              |
| 3 | h       | i     | i     | 011              |
| 4 | i       | h     | h     | 100              |
| 5 | i       | h     | i     | 101              |
| 6 | i       | i     | h     | 110              |
| 7 | i       | i     | i     | 111              |

#### Megjegyzés

Maga a számlálós ciklus persze elhagyható és bevezethetünk a **GENERÁTOR** helyett egy olyan függvényt, ami egy megadott megoldás jelölt alapján generálja a következő megoldás jelöltet.

### 8.3. Algoritmus A „nyers erő” módszer általános alakja

**Kimenet:** . - T (*egy optimális megoldás*)

- 1: **függvény** BRUTEFORCE( )
- 2:      $o \leftarrow \text{GENERÁTOR}(1)$
- 3:     **ciklus**  $i \leftarrow 2$ -től  $m$ -ig
  - 4:          $x \leftarrow \text{GENERÁTOR}(i)$
  - 5:         **ha**  $\text{JÓSÁG}(x) > \text{JÓSÁG}(o)$  **akkor**
  - 6:              $o \leftarrow x$
  - 7:         **elágazás vége**
- 8:     **ciklus vége**
- 9:     **vissza** o
- 10: **függvény vége**

#### Felhasznált változók és függvények

- $m$  : Lehetséges megoldások száma.
- $o$  : Az eddig talált legjobb megoldást tartalmazza, ez a függvény visszatérési értéke.
- $x$  : Az  $i$ . lehetséges megoldás.
- **GENERÁTOR**( $i$ ) : Visszaadja az  $i$ . lehetséges megoldást (**GENERÁTOR**:  $\mathbb{N}^+ \rightarrow T$ ).
- **JÓSÁG**( $x$ ) : Megadja az  $x$  megoldás jóságát (**JÓSÁG**:  $T \rightarrow \mathbb{R}$ ).

### A 0-1 hátizsák probléma megoldása

A nyers erő alapú megoldáshoz tehát meg kell határoznunk az összes lehetséges megoldásjelölt számát ( $m$ ), illetve az ezeket a megoldásokat előállító generátor függvényt (**GENERÁTOR**).

A megoldásjelöltek száma egyszerűen kiszámolható, ha az egyes tárgyak hátizsákba helyezését egymástól függetlennek tekintjük. minden döntés 2 lehetőséget jelent, tehát egy tárgy esetén összesen 2 lehetőségünk van, két tárgy esetében  $2 * 2 = 4$ , három tárgy esetében  $2 * 2 * 2 = 8$ . Általános esetben tehát  $m = 2^n$ , ahol  $n$  a tárgyak száma,  $m$  pedig a lehetséges pakolások száma.

Következő lépésként a **GENERÁTOR** függvényt kell definiálnunk, ehhez érdemes valamilyen formában sorbarendezni a lehetséges megoldásokat. A 8.1. táblázat bemutatja a három tárgy esetén felmerülő lehetséges pakolásokat (az előbb tisztáztuk, hogy  $2^3 = 8$  ilyen van).

A táblázat első  $i$  oszlopa mutatja a megoldás azonosítóját, ez a **GENERÁTOR** függvénynek átadott paraméter. Apró eltérés, hogy a **GENERÁTOR** függvényt most nem 1 és  $m$  között értelmezzük, hanem 0 és ( $m - 1$ ) között, mivel így látványosabb lesz az eredmény.

A következő három oszlop azt mutatja, hogy az adott számú pakolás esetében a  $t_1$ ,  $t_2$ ,  $t_3$  tárgyakat berakjuk-e a hártsákba („h” - *hamis*, tehát nem, „i” - *igaz*, tehát igen). Az utolsó oszlopba írjuk fel az így kapott pakolás adatait egy 3 hosszú bináris számként, ahol az egyes bitek annak felelnek meg, hogy az adott tárgyat beraktuk-e a hártsákba vagy pedig sem.

Vegyük észre, hogy az utolsó oszloban valójában az  $i$  értéke szerepel kettes számrendszerbeli alakban. A **GENERÁTOR** függvény működési elve tehát az alábbi lesz: a paraméterként kapott  $i$  számot átalakítjuk kettes számrendszerbeli alakra, és ahol 1-es bit szerepel, azt a tárgyat úgy tekintjük, hogy a pakolás része, a többiek pedig úgy, hogy nem.

Az egyes pakolásokat visszaadó **GENERÁTOR** függvény megvalósítását mutatja a 8.4. algoritmus.

---

#### 8.4. Algoritmus A lehetséges pakolásokat visszaadó generátor függvény

---

**Bemenet:**  $i$  - egész<sup>+</sup> (az igényelt pakolás száma)

**Kimenet:**  $\dots$  - tömb<logikai> (az  $i$ . lehetséges pakolás)

```

1: függvény GENERÁTOR(i)
2: szám $\leftarrow i - 1$
3: ciklus $j \leftarrow 1$ -től n -ig
4: $K[j] \leftarrow \lfloor \text{szám}/2^{j-1} \rfloor \bmod 2 = 1$
5: ciklus vége
6: vissza K
7: függvény vége
```

---

#### Felhasznált változók és függvények

- $n$  : A hártsákba pakolható tárgyak száma.
  - $\text{szám}$  : Egy egész<sup>+</sup> típusú segédváltozó, ami a bináris formára átalakítandó számot tárolja.
  - $K$  : Az  $i$ . pakolás adatai, ami egy logikai tömb, ahol  $K[j]$  azt mutatja, hogy az  $j$ . tárgy benne van-e a zsákban ( $1 \leq j \leq n$ ).
- 

A megoldásjelöltek generálását a 3. sorban kezdődő ciklus végzi. A 0-tól  $2^n - 1$ -ig tartományban minden egész számhoz egy megoldásjelöltet rendelünk. A megoldásjelölt generálása azon alapul, hogy megvizsgáljuk az adott egész szám kettes számrendszerbeli alakját: minden bináris számjegyhez egy tárgyat rendelünk, és ha az adott szám  $j$ . bitjének értéke 1, akkor a  $j$ . tárgyat elhelyezzük a pakolásba, ha a bit értéke 0, akkor pedig nem. A  $j$ . bit értékét az egész szám  $2^j$ -nel való osztásának maradékából tudjuk meghatározni. Az aktuálisan vizsgált pakolást a  $K$  logikai tömb tartalmazza.

Az alapelv megismérését követően már egyszerűen megírható a hártsák problémát megoldó algoritmus (8.5. algoritmus). Miként a fejezet többi megoldásánál is, a feladat bemenetét ( $n$ ,  $w$ ,  $p$ ,  $W_{max}$  értékek) ismertnek tekintjük, a könnyebb áttekinthetőség miatt ezeket nem tüntetjük fel a paraméterlisztában. A függvény visszatérési értéke egy optimális pakolás összértéke.

Az áttekinthetőség kedvéért néhány apró módosítást teszünk az általános algoritmushoz képest: a megoldásokat pedig nem 1-től hanem 0-tól számozzuk, mert így egyszerűbb formához jutunk; illetve, a kérdés nem az optimális pakolás, hanem csak annak az összértéke, ezért ez a függvény visszatérési értéke is.

---

## 8.5. Algoritmus A 0-1 hátizsákprobléma megoldása „nyers erő” algoritmussal

---

**Kimenet:** . - szám<sup>+</sup> (egy optimális pakolás összértéke)

```
1: függvény HÁTIZSÁKBF()
2: $O \leftarrow \text{GENERÁTOR}(1)$
3: ciklus $i \leftarrow 2$ -től 2^n -ig
4: $K \leftarrow \text{GENERÁTOR}(i)$
5: ha ($\text{ÖSSZSÚLY}(K) \leq W_{max}$) \wedge ($\text{ÖSSZÉRTÉK}(K) > \text{ÖSSZÉRTÉK}(O)$) akkor
6: $O \leftarrow K$
7: elágazás vége
8: ciklus vége
9: vissza $\text{ÖSSZÉRTÉK}(O)$
10: függvény vége
```

---

### Felhasznált változók és függvények

- $n$  : A hátizsákba pakolható tárgyak száma.
  - $W_{max}$  : A hátizsák méréte (a belepakolható tárgyak maximális összsúlya).
  - $K$  : Az aktuálisan vizsgált pakolás adatai (*tömb<logikai>*).
  - $O$  : Az eddig talált legjobb pakolás adatai (*tömb<logikai>*).
  - $\text{ÖSSZÉRTÉK}(X)$  : Megadja az  $X$  pakolás összértékét (8.1. algoritmus).
  - $\text{ÖSSZSÚLY}(X)$  : Megadja az  $X$  pakolás összsúlyát (8.2. algoritmus).
- 

A függvény első sorában inicializáljuk az  $O$  változót. Ez nem minden optimalizálációban lehetséges, de itt egyszerűen, bármiféle keresés nélkül tudunk adni egy lehetséges megoldást: ha nem rakunk semmit a zsákba, akkor az biztosan érvényes pakolás lesz (ez egyébként  $\text{GENERÁTOR}(0)$  értéke). Az ezt követő ciklusban arra törekszünk, hogy találunk egy ennél jobbat.

Tudjuk, hogy  $n$  darab független tárgyról kell igen vagy nem döntést hoznunk, ennek megfelelően összesen  $2^n$  darab lehetséges megoldásjelöltünk van. Ezek egy része lesz csak érvényes pakolás, és ezek közül is csak néhány lesz optimális.

A generátor függvény visszaadja a következő megoldásjelöltet, amit még ellenőriznünk kell. A 5. sorban látható feltétel első tagja azt figyeli, hogy egy érvényes pakolásról van-e szó (összsúlya nem nagyobb, mint a hátizsák kapacitása). Amennyiben igen, csak akkor értékelődik ki a második feltétel, ami azt vizsgálja, hogy az aktuálisan vizsgált megoldás jósága nagyobb-e, mint az eddig talált legjobb megoldás. Ha igen, akkor ezt követően ezt tekintjük optimálisnak.

A fenti ciklust tehát az 1-től  $2^n - 1$ -ig tartó tartomány minden egész számát végigjárja (a 0-t nem kell, hiszen az pont a nincs a zsákban semmi állapotot képviseli, ami az  $O$  kezdeti értéke). A megoldás tulajdonképpen megfelel a jól ismert maximumkiválasztás tételnek, ennek megfelelően a visszatérési érték az optimális összértéket tartalmazza. Amennyiben szükségünk van magára a pakolás részletes adataira, akkor azt az  $O$  tömbből kiolvashatjuk az eljárás végén.

### A módszer értékelése

Látható, hogy a módszer tárhelyigénye nem jelentős, mindig csak két pakolást kell eltárolnunk. A futásidő azonban extrém magas, a segédműveletektől eltekintve is látható, hogy a próbálkozásokat irányító ciklus  $2^n - 1$  iterációt igényel.

A módszer előnyei:

- Egyszerűen megérthető és nagyon gyorsan implementálható.
- Elvileg minden esetben használható, ahol véges a lehetséges megoldások száma. Garantáltan megtalálja az optimális megoldás(okat).
- Amennyiben nincs semmilyen támpontunk, akkor gyakran csak ez használható (pl. jelszavak feltörése, stb.)
- Tárhelyigénye minimális.

A módszer hátrányai:

- Futásideje nagyon magas. Emiatt nagyobb problémák megoldása során a gyakorlatban nem használható.

### 8.3. Mohó algoritmusok

#### Egy bevezető példa

A „mohó algoritmusok”<sup>5</sup> bemutatására nézzük egy nevezetes problémát, ami egy megadott összeg felbontása a lehető legkevesebb adott címletű bankjegy segítségével. Tehát előre meghatározott címletű pénzjegyeink vannak (pl. 1, 5, 10, 20, 50, 100, 200, 500), és a feladat annak megállapítása, hogy melyek felhasználásával tudunk a lehető legkevesebb pénzjegyből kifizetni egy megadott összeget.

A megoldás hétköznapi ismereteink alapján már ismert: a kifizetendő összegnél nem nagyobb címletek közül a legnagyobból vegyük el annyit, amennyi szükséges, majd a maradék összeget fizessük a nála kisebb pénzjegyekkel (ezt megtethetjük egy rekurzív hívással, vagy akár egy ciklussal is).

#### 8.6. Algoritmus Pénzváltás legkevesebb címlettel

Bemenet:  $a$  - egész<sup>+</sup> (a felváltandó összeg)

Bemenet:  $n$  - egész<sup>+</sup> (a lehetséges címletek száma)

Bemenet:  $C$  - tömb<egész<sup>+</sup>> (a lehetséges címleteket tartalmazza növekvő sorrendben)

Kimenet:  $O$  - tömb<egész<sup>+</sup>> (a szükséges címletek darabszáma)

- 1: függvény PÉNZVÁLTÁS( $a, n, C$ )
- 2:   ciklus  $i \leftarrow n$ -től 1-ig
- 3:      $O[i] \leftarrow \lfloor a/C_i \rfloor$
- 4:      $a \leftarrow a \bmod C_i$
- 5:   ciklus vége
- 6:   vissza  $O$
- 7: függvény vége

#### Felhasznált változók és függvények

- $O$  : Egy tömb<egész<sup>+</sup>> tömb, ami az egyes címletek szükséges darabszámát tartalmazza.

Látható, hogy a fenti címletek használatával a program jól működik, pl. 345 Ft felbontására a következő eredményt adja: [0, 1, 0, 2, 0, 1, 1, 0].

Azonban mielőtt elbíznánk magunkat, nézzük egy másik bemenetet, ahol 10 Ft-ot kell felbontani, és a címletek [1, 5, 7]. Az algoritmus a 7 Ft-os címlethez 1-et rendel, majd a maradék 3 Ft-ot próbálja tovább osztani, azonban ez már eleve kisebb, mint a hátralévő 5 Ft-os címlet, így három darab 1 Ft-ost fog keresni, így az eredmény [3, 0, 1] lesz. Ez nyilvánvalóan hibás, hiszen azonnal tudunk mondani egy kevesebb címletet igénylő megoldást: [0, 2, 0] az optimális megoldás.

A fenti példa nagyon jól jellemzi a mohó algoritmusokat:

- Ezek általában nagyon egyszerű algoritmusok, amelyek mindenkorban minden címletet kihasználva kiszámítják a legjobbnak tűnő irányba haladnak.
- Ezek gyakran nem találják meg a tényleges optimumot, csak egy közelítő értéket. Ez persze nem a véletlenben működik, egy megfelelő szabályrendszerrel meg lehet határozni, hogy mely esetekben fog tökéletesen működni az algoritmusunk.

#### Megjegyzés

A második pontot ne értsük félre: vannak olyan mohó algoritmusok, amelyek mindenkorban minden címletet kihasználva kiszámítják a legjobbnak tűnő irányba haladnak. A gyakorlatban viszont sokszor megjelennek olyanok, amelyek esetében ez nem garantálható, és ez valamelyest összefonódott a mohó algoritmusok jellemzőivel.

Például a címleteket meghatározó algoritmus bizonyíthatóan mindenkorban minden címletet kihasználva kiszámítja a legjobbnak tűnő irányba haladnak. Ez persze nem mindenkorban minden címletet kihasználva kiszámítja a legjobbnak tűnő irányba haladnak. A gyakorlatban viszont sokszor megjelennek olyanok, amelyek esetében ez nem garantálható, és ez valamelyest összefonódott a mohó algoritmusok jellemzőivel.

<sup>5</sup>greedy algorithm

## A módszer bemutatása

Azokat az algoritmusokat nevezzük mohónak, amelyek a feladat megoldása során felmerülő részproblémák megoldásakor mindig az aktuálisan legjobbnak tűnő részmegoldást választják. Legjobbnak tűnő alatt azt értjük, hogy valamelyen egyszerű, gyorsan kiszámítható kritérium alapján kell dönten. Ez természetesen nagyon kedvező futásidőhöz vezet (nincs szétbontás, rekurzió, visszalépés, stb.), azonban a problémák csak szűk körénél adja vissza a teljes probléma optimális megoldását. Érdemes azonban megjegyezni, hogy azoknál a problémáknál, ahol nincs optimális megoldást visszaadó mohó algoritmus, gyakran létezik valamilyen közel optimális megoldást adó változat. Amennyiben ez is megfelelő, akkor tudjuk használni ezt a stratégiát.

### A 0-1 hátizsák probléma megoldása

Előrebocsátjuk, hogy a 0-1 hátizsák probléma maradéktalan megoldására *nem létezik* mohó algoritmus, ami visszaadná az optimális pakolás értékét. Olyan azonban van, ami egy közel optimális megoldást ad, ezért a továbbiakban ezt vizsgáljuk (ezzel megtartva azt az előnyt, hogy minden módszerrel ugyanazt a feladatot próbáljuk megoldani).

A megvalósítás meglehetősen hasonló lesz a pénzváltási feladathoz. A cél itt is az lenne, hogy a rendelkezésre álló tárgyakat valamilyen formában sorbarakjuk, majd ebben a sorrendben próbáljuk őket berakni a hátizsákba. Amely elemek beférnek, azokat berakjuk, ami nem, az pedig kimarad.

A megoldás kulcsa persze az, hogy milyen módon határozzuk meg ezt a sorrendet. Néhány ötlet:

- Rendezhetjük a tárgyakat érték szerint csökkenő sorrendben. Így tehát először a legnagyobb értékű tárgy kerül a zsákba (már persze ha belefér), és ha van még hely, akkor sorba próbáljuk az egyre kisebbeket. Mivel a nagy értékű tárgyakat rakjuk a zsákba, így azt várhatjuk, hogy az összérték is magas lesz.
- Rendezhetjük a tárgyakat méret szerint növekvő sorrendben. Tehát először a legkisebb tárgyat próbáljuk berakni, majd a második legkisebbet, stb. Ettől a megoldástól azt várjuk, hogy sok tárgy fog beleférni a zsákba, hiszen a kisebbekkel kezdünk, és ennek hatására ezek összértéke is magas lesz.
- Rendezhetjük a tárgyakat az „érték/súly” arányuk alapján is csökkenő sorrendbe. Ez a fenti két megoldás ötvözeteitől is felfogható, hiszen a nagyobb értékű de mégis kis helyet igénylő tárgyakat választjuk először. Így megintcsak azt várhatjuk, hogy az eredmény lehetőleg elfogadható lesz.

A fentiek közül bármelyik módszert is választjuk, a hátizsákba pakoló algoritmus ugyanaz lesz (8.7. algoritmus). A gyakorlatban egyik megoldás se ad garantáltan optimális eredményt, mindegyik valamelyik szempont szerint ad egy közel optimális kimenetet.

Az algoritmus első sora a tárgyak rendezését végzi el a fent megadott valamelyik stratégia szerint. Amennyiben vannak részletesebb információink a tárgyakról, akkor a rendezés lehet persze ezeknél is összetettebb. Minél jobb sorrendet tudunk felállítani, annál jobban fogja közelíteni a kimenet az optimálisat.

Az algoritmus alapelve tehát hasonló a pénzváltási feladatéhoz. Mivel itt nem biztos, hogy minden tárgyat meg fogunk vizsgálni, ezért előre kitöljtük az  $O$  vektor mezőit hamis értékekkel (3. sor), majd az  $t$  változó beállítása után elindul a belső ciklus.

A ciklusfeltétel (5. sor) azt vizsgálja, hogy

- nincs még teli a zsák, hiszen csak ebben az esetben van értelme további vizsgálatoknak;
- vannak még meg nem vizsgált tárgyak, ellenkező esetben ki lehet lépni a ciklusból.

Amennyiben belépünk a ciklusba, akkor megvizsgáljuk a  $t$ . tárgyat. A hasznossággal itt nem is foglalkozunk, hiszen már a rendezéskor igyekeztünk az elemeket olyan sorrendbe rakni, hogy a minél előbb lévő elemeket próbáljuk berakni a zsákba. Itt már csak azt nézzük, hogy a  $t$ . tárgy a már beválasztottak mellett befér-e a zsákba (6. sor). Ha igen, akkor őt is berakjuk (7. sor), ha nem, akkor kihagyjuk.

Miután a ciklus végetért, az  $O$  változó tartalmazza az általunk talált legjobb pakolás adatait. Ennek összértéke lesz a függvény visszatérési értéke (7. sor).

Érdemes néhány teszt bemenettel kipróbálni az algoritmust, és tudatosan keresni olyan értékeket, amelyekre megtalálja az optimumot, és olyat is, amire csak egy közelítő értéket tud adni.

## 8.7. Algoritmus A 0-1 hátizsákprobléma közelítő megoldása „mohó algoritmussal”

**Kimenet:** . - szám<sup>+</sup> (egy közel optimális pakolás összértéke)

- 1: **függvény** HÁTIZSÁKMOHÓ( )
- 2:    TÁRGYAKRENDEZÉSE( $i \rightarrow -p_i/w_i$ )
- 3:     $O \leftarrow [\text{hamis}, \text{hamis}, \dots, \text{hamis}]$
- 4:     $t \leftarrow 1$
- 5:    **ciklus amíg** ( $\text{ÖSSZSÚLY}(O) < W_{max}$ )  $\wedge$  ( $t \leq n$ )
- 6:     ha  $\text{ÖSSZSÚLY}(O) + w_t \leq W_{max}$  **akkor**
- 7:        $O[t] \leftarrow \text{igaz}$
- 8:     **elágazás vége**
- 9:      $t \leftarrow t + 1$
- 10:    **ciklus vége**
- 11:    **vissza**  $\text{ÖSSZÉRTÉK}(O)$
- 12: **függvény vége**

### Felhasznált változók és függvények

- $n$  : A hátizsákba pakolható tárgyak száma.
- $O$  : Az optimális megoldást tárolja (*tömb<logikai>*).
- $W_{max}$  : A hátizsák mérete (a belepakolható tárgyak maximális összsúlya).
- $w_i$  : A hátizsákba pakolható  $i$ . tárgy súlya ( $1 \leq i \leq n$ ).
- $p_i$  : A hátizsákba pakolható  $i$ . tárgy hasznossága ( $1 \leq i \leq n$ ).
- **TÁRGYAKRENDEZÉSE(feltétel)** : A megadott három módszer közül valamelyikkkel rendezzi a hátizsákba pakolandó tárgyakat. Ennek megfelelően módosítja a paraméterként átadott súly és érték listákat.

### Megjegyzés

A 0-1 hátizsák problémát ugyan nem tudta tökéletesen megoldani az algoritmus, de egy kis módosítással a töredékes hátizsák problémára maradéktalanul alkalmazható. Ez utóbbi egy jó példa arra, amikor a mohó algoritmus hatékonyan és hibátlanul megoldja a feladatot.

### A módszer értékelése

A mohó stratégia tehát azon alapult, hogy minden lépésben csak az ott ismert információk alapján próbálunk döntést hozni a továbblépés irányáról. Ez kétségtelenül gyors, azonban miként az láttuk, nem vezet mindig optimális megoldáshoz.

#### A módszer előnyei

- Egyértelműen az erőforrásigényt jelölhetjük meg első sorban. Mind futásidő, mind pedig tárhely szempontjából meglehetősen hatékony megoldásokat tud adni a technika.
- Ha nem is adja vissza az optimális megoldást, gyakran ad egy jól használható közelítést. Ez kiindulási alapja lehet egyéb kiegészítő módszereknek.

#### A módszer hátrányai

- Csak a feladatok egy nagyon szűk körén alkalmazható.

Számos példát lehet hozni arra, amikor a mohó algoritmus bizonyíthatóan optimális megoldást ad vissza. Ezek közül néhánnyal még később foglalkozunk, tipikusan a különféle gráf algoritmusok vizsgálata során (13.2. fejezet, 13.3.2. fejezet, 13.3.3. fejezet).

## 8.4. Heurisztikus módszerek

### 8.4.1. A módszer bemutatása

A heurisztika szó rátalálást jelent, ami jelen esetben egy újabb feladat megoldási stratégiát takar. Az elnevezés onnan ered, hogy a megoldást nem jól meghatározott (vagy akár bizonyított) következtetések útján találjuk meg, hanem valamiféle intuitív, gyakran véletlenszerű lépések eredményeképpen. Mérnöki tudományokban általában heurisztikusnak nevezzük azokat a megoldásokat, amelyeknek nem minden ismert a pontos fizikai, matematikai alapja (vagy ismert, de az túl körülmenyes a gyakorlati használathoz), de a gyakorlatban mégis használhatók, hiszen a tapasztalatok alapján jó eredményeket érnek el.

Az algoritmusok terén a heurisztikákat általában akkor használjuk, ha a megoldást egy túl nagy keresési térben kell megkeresni és az egyébként már jól bevált és működő módszereink nem használhatók a gyakorlatban. Erre jó példa lehet a 0-1 hátziszák probléma is, amire számos megoldási stratégia létezik. A „nyers erő” módszerét vizsgálva láttuk, hogy a lehetséges megoldások száma  $n$  darab tárgy esetén  $2^n$ . Ha egy hátziszák pakolás értékét 1 ezredmásodperc alatt tudjuk kiszámolni, akkor könnyen látható, hogy míg 10 tárgyra az algoritmus futásideje 1 másodperc körüli, addig 20 tárgynál már 17 perc, 30 tárgynál már 12 nap, 40 tárgynál pedig 34 év. A gyakorlatban pedig előfordulhatnak még nagyobb problématerek is, pl. rendelkezésre áll több száz alkatrész, ezek közül kell kiválasztani azt a kombinációt, amelyekkel hatékonyan meg lehet oldani egy felmerülő problémát.

Az „oszd meg és uralkodj” és a „visszalépéses keresés” elvű megoldások esetében a futásidő jóval kisebb lehet, és ezek is garantáltan megtalálják az optimális megoldást, azonban a tárgyak és a hátziszák méretének növelésével itt is drasztikusan növekszik a futásidő. Bár valamivel később, de előbb-utóbb itt is elérjük azt a határt, ami már elfogadhatatlan a gyakorlatban.

A gyakorlatban ezért elterjedten használunk úgynevezett metaheurisztikákat. Ezek olyan általános stratégiák, amelyek általában egymást követő véletlen döntések segítségével próbálnak megadni egy közelítő megoldást a felmerülő problémákra. A már ismert módszerekhez képest ezek számos hátránya bírnak:

- Általában nem várható el tőlük, hogy tökéletes megoldást adjanak. Többnyire egy közel optimális megoldás várható el.
- Valójában arra sincs garancia, hogy ezt a közelítő megoldást megkapjuk. Amennyiben az algoritmust véletlen döntései rossz irányba viszik, akkor elképzelhető, hogy nem kapunk használható eredményt (persze a „közel optimális” meglehetősen tágán értelmezhető).

Viszont egy kétségtelen előnyük van:

- Általában működnek. Olyan esetekben is, ahol a többi stratégia nem tud semmilyen eredményt adni, ezek legalább egy közelítő megoldást meg tudnak határozni.

### 8.4.2. Hegymászó algoritmus általános alakja

A hegymászó algoritmus az egyik legegyszerűbb metaheurisztika, ezért példaként ezt vizsgáljuk meg. Az elnevezés onnan adódik, hogy milyen stratégiával tud egy túrázó feljutni egy hegy tetejére, ha nincs térképe és nem ismeri a környéket: körülözött ott, ahol éppen áll, és ha valamelyik irányba emelkedőt lát, akkor arrafele lép egyet, majd ezt ismételgeti addig, amíg már nem tud feljebb menni. Könnyen belátható, hogy ez nem minden vezet optimális megoldáshoz (pl. lehet, hogy felmászik egy dombra a legmagasabb hegy mellett), de ezt nem is várhatjuk el a metaheurisztikáktól.

Az algoritmus tulajdonképpen ugyanezt az elvet követi, csak úgy tekintjük, hogy a lehetséges megoldások halmozában (a keresési térben) mozgunk. Tehát első lépéseként fog egy kezdőpontot, ahonnan elindul majd a keresés. Mivel nincs semmilyen információnk az optimumról, ez általában egy véletlenül kiválasztott/generált lehetséges megoldás szokott lenni. Kezdetben ezt tekintjük az eddig elért legjobb megoldásnak.

Ezt követően pedig egy ciklusban megvizsgálja az ehhez a megoldáshoz hasonlító további megoldásjelölteket (ezek a keresési térben a közelí elemek, ezért szomszédoknak nevezzük őket). Itt többféle stratégia is létezik, mi véletlenszerűen előállítunk egy, az aktuális legjobbhoz „hasonló” megoldást, majd megvizsgáljuk, hogy az jobb-e mint az eddig feltételezett legjobb. Ha igen, akkor átlépünk ebbe a pontba, ha nem, akkor maradunk az aktuálisan legjobbnak tűnő helyen.

A ciklust egy előre meghatározott feltétel szerint ismételjük. Amikor véget ért, akkor a függvény visszatérési értéke az eddig talált legjobb hely lesz.

A 8.8 bemutatja a hegymászó módszer általános alakját.

---

### 8.8. Algoritmus A „hegymászó” lokális keresési módszer általános alakja

---

**Kimenet:** . -  $T$  (egy optimálisnak tűnő megoldás)

```
1: függvény HEGYMÁSZÓ()
2: $o \leftarrow \text{RND}(T)$
3: ciklus $i \leftarrow 1$ -től m -ig
4: $x \leftarrow \text{SZOMSZÉD}(o)$
5: ha $\text{JÓSÁG}(x) > \text{JÓSÁG}(o)$ akkor
6: $o \leftarrow x$
7: elágazás vége
8: ciklus vége
9: vissza o
10: függvény vége
```

---

#### Felhasznált változók és függvények

- $m$  : A végrehajtandó iterációk száma.
  - $o$  : Az eddig talált legjobb megoldást tartalmazza, ez a függvény visszatérési értéke.
  - $x$  : Az eddig talált legjobb megoldás módosított értéke.
  - $\text{RND}(T)$  : Visszaad egy véletlen megoldásjelöltet.
  - $\text{SZOMSZÉD}(x)$  : Visszatérési értéke a paraméterként átadott  $x$  megoldás véletlen módosított változata ( $\text{SZOMSZÉD} : T \rightarrow T$ ).
  - $\text{JÓSÁG}(x)$  : Megadja az  $x$  megoldás jóságát ( $\text{JÓSÁG} : T \rightarrow \mathbb{R}$ ).
- 

Az algoritmusban jól láthatók a módszer alapvető lépései:

- Első lépésként létrehoz egy véletlen megoldásjelöltet. Mivel semmit se tudunk az ideális megoldásról, ezért itt nincs semmilyen támponk, a legtöbb amit tehetünk, az a véletlen inicializálás. A  $o$  változó végig az eddig talált legjobb megoldást fogja tárolni, induláskor ez a kezdőpont, hiszen más pontot még nem vizsgáltunk.
- Ezt követi egy ciklus, amelynek futása során folyamatosan megpróbáljuk finomítani az eddig talált legjobb megoldást. A metaheurisztikák egyik nagy hátránya, hogy nehéz megmondani, hogy meddig érdemes őket futtatni. Véletlen keresésről lévén szó, bármikor elköpzelhető, hogy egy következő lépésben tudunk javítani az eddig talált optimális megoldáson. Praktikus okokból viszont célszerű valamilyen korlátot adni a futásidőre, jelen esetben azt választjuk, hogy összesen  $m$  darab iterációt fogunk lefuttatni.
- Az iteráció első lépéseként keresünk egy, az aktuálisan legjobb  $o$  elemhez hasonló másik megoldásjelöltet. Tehát az  $o$  valamelyik szomszédját a keresési térben. A keresést nem kell feltétlenül szó szerint érteni, az is járható út, ha az  $o$  ismeretében előállítunk egy hozzá hasonló jelöltet.
- Mognézzük, hogy az így kapott  $x$  jobb megoldás-e, mint az eddig feltételezett legjobb. Ha igen, akkor ennek értékét átmásoljuk a  $o$  változóba.
- Az iterációt összesen  $m$ -szer lefuttatjuk, majd visszaadjuk a megtalált legjobb eredményt.

#### 8.4.3. A 0-1 hátízsák probléma megoldása

A hegymászó algoritmussal megpróbálhatjuk megoldani a már ismert 0-1 hátízsák problémát is. Ehhez meg kell határoznunk az alábbi műveleteket:

- Véletlen elem előállítása: az egyes pakolásokat egy  $n$  elemű logikai tömbben tároltuk el. A véletlen inicializálás során ezt a tömböt feltöltjük véletlen *igaz* és *hamis* értékekkel.
- Szomszéd kiválasztás: fogjuk az aktuális pakolás valamelyik logikai értékét és egyszerűen negáljuk. Tehát fogunk egy véletlen  $j$  számot 1 és a tárgyak száma ( $n$ ) között, és ha a  $j$ . tárgy benne van a hátízsákba, akkor kiveszük, ellenkező esetben pedig beletesszük.

- Jóságfüggvény: a jóságfüggvény eddigi megoldásainkban minden visszaadta a pakolás értékét, hiszen ezt kell maximalizálnunk. Jelen esetben annyival bonyolódik az algoritmus, hogy a véletlen inicializálás és módosítások miatt gyakran fogunk érvénytelen pakolásokhoz jutni, tehát a jóságfüggvénynek ezekre is valamilyen eredményt kell adnia.

$$\text{JÓSÁG}(x) = \begin{cases} \text{ÖSSZÉRTÉK}(x), & \text{ha } \text{ÖSSZSÚLY}(x) \leq W_{max} \\ W_{max} - \text{ÖSSZSÚLY}(x), & \text{ha } \text{ÖSSZSÚLY}(x) > W_{max} \end{cases} \quad (8.1)$$

A jóságfüggvény értéke tehát lehet

- pozitív: érvényes pakolás esetén egyenlő a elhelyezett tárgyak összértékével;
- 0: a pakolás ugyan érvényes, de a tárgyak összértéke 0;
- negatív: a pakolás érvénytelen, a súlytöbbletet kapjuk vissza negatív számként.

#### Megjegyzés

Egyszerűbb lett volna, ha minden érvénytelen pakolás esetén 0 jóságot adunk vissza. A fenti megoldás viszont azért praktikusabb, mert az érvénytelen pakolások közül is rosszabb eredményt kap az „érvénytelenebb” tehát ahol nagyobb súlyval léptük túl a hátizsák kapacitását. Ez azért fontos, mert a heurisztika így a rossz megoldások közül is tud a kevésbé rossz felé haladni, amíg meg nem találja a jó megoldásokat.

### 8.9. Algoritmus A 0-1 hátizsák probléma közelítő megoldása hegymászó stratégiával

**Kimenet:** . - szám<sup>+</sup> (egy közel optimális pakolás összértéke)

```

1: függvény HÁTIZSÁKHC()
2: ciklus i \leftarrow 1-től n-ig
3: O[i] \leftarrow RND(logikai) \triangleright igaz vagy hamis érték véletlenszerűen
4: ciklus vége
5: ciklus i \leftarrow 1-től m-ig
6: X \leftarrow O
7: j \leftarrow RND(egész+ [1..n]) \triangleright Véletlen egész szám 1 és n között
8: X[j] \leftarrow \neg X[j] \triangleright Az X vektor j. elemének negálása
9: ha JÓSÁG(X) > JÓSÁG(O) akkor
10: O \leftarrow X
11: elágazás vége
12: ciklus vége
13: vissza ÖSSZÉRTÉK(O)
14: függvény vége

```

#### Felhasznált változók és függvények

- *i*, *i* : egész<sup>+</sup> típusú segédváltozók.
- *m* : A végrehajtandó iterációk száma.
- *O* : Az eddig talált legjobb pakolás adatai (*tömb*<logikai>).
- *X* : Az *O* egy pontban módosított változata (*tömb*<logikai>).
- RND(...) : Visszaad egy véletlen értéket a megadott típusnak/tartománynak megfelelően.
- JÓSÁG(*X*) : Megadja az *X* megoldás jóságát az (8.1) egyenlet alapján (JÓSÁG: *T*  $\rightarrow$   $\mathbb{R}$ ).

A hátizsák problémát megoldó hegymászó algoritmus alapvetően megegyezik az általános alakkal, helyettesítve a feladat függő részeket a tényleges implementációval.

Az első ciklus létrehoz egy vélelen megoldásjelöltet. Mivel egy pakolás tulajdonképpen *n* darab logikai értékből álló tömb, ezért egy ciklus segítségével feltöltünk egy tömböt *n* darab véletlen *igaz* vagy *hamis* értékkel. Az így kapott pakolásról egyelőre semmi információt nincs, de mivel ez az egyetlen, ezért ezt tekintjük a legjobbnak. Az is lehet, hogy nem is érvényes pakolás.

A következő ciklus a már látott módon *m* darab iterációt fog végrehajtani, ezek a hegymászó algoritmus lépései.

Minden lépés elején előállítjuk az aktuálisan legjobbnak gondolt  $O$  elem egy szomszédját. Ehhez átmásoljuk a vektort egy másik  $X$  vektorba, mivel gondolunk egy esetleges sikertelen lépésre is, ami miatt meg kell őriznünk az aktuális  $O$  értéket is. Ezt követően megtörténik maga a változtatás: a  $j$  változóba kerül egy véletlen szám 1 és a tárgyak száma ( $n$ ) között, majd pedig negáljuk az  $X$  tömb  $j$ -edik elemét. Tehát ha az eddig *igaz* volt, akkor *hamis* lesz, ha pedig *hamis* volt, akkor *igaz* lesz. Tehát, ha a  $j$ -edik tárgy benne volt a hátizsákban, akkor kivesszük, ellenkező esetben pedig belerakjuk.

Ezt követi annak a vizsgálata, hogy az így kapott elem jobb-e, mint az eddig feltételezett legnagyobb. Ha igen, akkor innentől ezt tekintjük optimálisnak, ellenkező esetben az  $O$  értékét nem változtatjuk.

Bármi is volt a vizsgálat eredménye, ha még nem értünk el az  $m$ . iterációhoz, akkor folytatódik az újabb irányok vizsgálata. A ciklus végeztével visszaadjuk az eddig feltételezett legjobb ( $O$ ) eredmény összértékét.

Az algoritmus működési módjából adódóan nem tudunk igérni ezügyben semmit. Lehet, hogy valóban egy optimális megoldást találtunk, de az is lehet (és minél nagyobb a probléma, annál nagyobb ennek a tényleges esélye), hogy a megoldás távol áll az optimálistól.

## A módszer értékelése

Már a véletlen számok megjelenéséből is látható, hogy ez a módszer egészen másképp működik, mint a hagyományos kereső/optimalizáló algoritmusok. Néhány specialitást érdemes külön is kiemelni:

- A metaheurisztikák általában nem determinisztikusak, tehát az algoritmusba beépített véletlen döntések miatt egymást követő futások más-más eredményeket adhatnak azonos bemenetre is.
- A nem determinisztikus létük miatt nehéz következtetést levonni a pontosságról, hiszen egymást követő futások adhatnak tökéletes, vagy akár használhatatlan eredményeket is. A pontosság persze ettől még vizsgálható, de főleg statisztikai módszerekkel.

### A módszer előnyei

- Képesek lehetünk vele (közelítő) megoldást találni olyan feladatokra, amelyeket a hagyományos módszerekkel nem tudunk megoldani a nagy számításigény miatt.
- Futásidéjük általában nagy, de futás közben egyre jobb megoldásokat találnak, tehát már menet közben is kaphatunk egyre pontosabb eredményeket.

### A módszer hátrányai

- Általában nem adnak pontos megoldást, de ha igen, akkor se tudjuk, hogy tényleg megtaláltuk-e az optimális megoldást.
- Bár általában jó eredményekre számíthatunk, de semmi se garantálja egy-egy futás jó eredményét.
- Általában felmerülnek az implementációval kapcsolatos további, nehezen megválaszolható kérdések:
  - Mi legyen az  $m$  értéke? Illetve biztos ez a legjobb megoldás, hogy előre fixáljuk a lépések számát?
  - Mekkora legyen a lépésenkénti változtatás mértéke? Most minden egy tárgyat változtattunk, de érdemes lenne egyszerre akár többet is?

A heurisztikák egy óriási téma, aminek a hegymászó algoritmus talán a legegyszerűbb képviselője. Szinte mindenikben megjelenik valamilyen szinten a véletlen, de fontos észrevenni, hogy ez nem azt jelenti, hogy teljesen céltalanul véletlen számokat dobálva bolyongunk a keresési térben. Éppen azért van sokféle metaheurisztika, mivel különböző stratégiákkal próbálják meg azt elérni, hogy az egyébként véletlen lépések hosszú távon mégis egy jó megoldás felé közelítsenek.

- A hegymászó algoritmus során ezt az biztosította, hogy bár véletlen környezetet vizsgálunk, de minden csak akkor lépünk át, ha jobb megoldást találtunk. Így monoton növekedni fog a jóság értéke. Vannak megoldások, amelyek hasonlóan működnek, de néha megengedik a rossz irányba lépést (Szimulált Lehűtés), vagy éppen megjegyzik a már vizsgált állapotokat és azokat nem vizsgálják újra (Tabu keresés).

- Ezeken túl vannak egészen eltérő alapú megoldások, mint a populáció alapú módszerek, amelyek legismertebb tagja a Genetikus Algoritmus. Ebben az esetben nem csak egy, hanem egyszerre több (száz/ezer) véletlen megoldást generálunk, ezek közül kiválasztjuk a legjobbakat, majd ezekből próbálunk meg újabb megoldásokat szerkeszteni. A véletlenű túlmutató stratégia alapja az, hogy ha a legjobbakat választjuk ki és ezekből kreálunk újabb megoldásokat, akkor remélhetőleg ezek is egyre jobbak lesznek (akár csak a természetben látható evolúció esetében).
- Számos egyéb, biológiai inspirált módszer van, amelyek például a hal/madár rajok (Particle Swarm Optimization), vagy akár a méhek/farkasok viselkedését utánozva próbálják a részben véletlen döntések alapján úgy alakítani a keresést, hogy az az optimum felé konvergáljon.

A további metaheurisztikák tárgyalása azonban messze túlmutat ezen jegyzet keretein, itt pusztán a véletlenű alapuló megoldási stratégiák létezését szerettük volna bevezetni.

## 9. fejezet

# Oszd meg és uralkodj

### 9.1. Oszd meg és uralkodj stratégia

#### 9.1.1. A módszer bemutatása

Az oszd meg és uralkodj stratégia az egyik legismertebb és leggyakrabban használt algoritmus tervezési módszer. Alapelve az, hogy egy bonyolult problémát felbontunk kettő vagy több valamelyest egyszerűbb részproblémára, majd ezekre újra elvégezzük ezt a felbontást egészen addig, amíg el nem jutunk olyan egyszerű részproblémáig, amelyek megoldása már nem okoz gondot (ezeket nevezzük „triviális megoldásoknak”).

A kisebb részproblémákra való szétbontást általában úgy értjük, hogy az így kapott részproblémák az eredetihez hasonlóak, csak például kisebb méretűek (kevesebb elem feldolgozását igénylik). Ilyenkor egy rekurzív algoritmushoz jutunk, hiszen a probléma felbontását végző függvény önmagát hívja meg a kisebb részproblémák megoldására.

Ezt az általános elvet mutatja be az alábbi egyszerű rekurzív algoritmus:

1. Amennyiben a megoldandó probléma egyszerű, akkor
  - (a) megoldjuk és visszaadjuk az eredményt.
2. Amennyiben a megoldandó probléma ehhez túl bonyolult, akkor
  - (a) felosztjuk több kisebb részproblémára,
  - (b) az így kapott kisebb részproblémákat egyesével megoldjuk (rekurzív hívásokkal),
  - (c) az így kapott részeredményeket egyesítjük, azok alapján az eredeti megoldandó probléma eredményét előállítjuk.

Néhány konkrét példa, amelyek remélhetőleg már ismertek, ezért részletesen nem is tárgyaljuk őket:

- Bináris keresés rendezett tömbben:
  - Alapötlet: fele annyi elem közül könnyebb megkeresni a keresendő elemet, mint az eredeti sorozatból.
  - Triviális megoldások:
    - \* Ha a vizsgált rendezett sorozat középső eleme a keresett, akkor megtaláltuk.
    - \* Üres sorozatban biztosan nincs benne a keresett elem.
  - Kisebb részproblémákra bontás, megoldás, egyesítés:
    - \* Ha a keresett elem nagyobb, mint a vizsgált rendezett sorozat középső eleme, akkor a keresést folytatjuk a középső utáni elemekre.
    - \* Ha a keresett elem kisebb, mint a vizsgált rendezett sorozat középső eleme, akkor a keresést folytatjuk a középső előtti elemekre.
- Részmegoldások egyesítése:

- \* Ha a részsorozat keresés megtalálta a keresett elemet, akkor az megtalálható az eredeti tömbben. Ha nem, akkor biztosan nincs.
- Összefésülő rendezés:
  - Alapötlet: fele annyi elemet könnyebb rendezni, mint a teljes sorozatot. Ezek rendezése után a két rendezett részsorozatot már egyszerűen össze tudjuk fésülni.
  - Triviális megoldások:
    - \* Ha a vizsgált részsorozat üres, vagy csak egy eleme van, akkor nincs teendőnk.
    - \* Ha a vizsgált részsorozatnak csak két eleme van, és ezek megfelelő sorrendben vannak, akkor nincs teendőnk.
    - \* Ha a vizsgált részsorozatnak csak két eleme van, és ezek fordított sorrendben vannak, akkor a két elemet meg kell cserélni.
  - Kisebb részproblémákra bontás, megoldás, egyesítés:
    - \* A vizsgált (rész)sorozatot két részre osztjuk.
    - \* A két részsorozatot egy-egy rekurzív hívással rendezzük.
  - Részmegoldások egyesítése
    - \* Az így kapott két rendezett részsorozatot összefésüljük.

De a fentiek mellett számos más megoldás alapul ezen az alapelven, mint pl. a gyorsrendezés, k-adik legkisebb tömbbeli elem kiválasztása, stb.

#### Megjegyzés

A fenti stratégia persze önmagát meghívó függvények nélkül is implementálható. A rekurzív szót olyan értelemben használjuk, hogy a kisebb részproblémákra azonos megoldási stratégiát használunk.

A módszer használható optimalizációs feladatok esetében is. Az alapelt hasonló: ha az optimális megoldás meghatározása túl bonyolult, akkor próbáljuk meg visszavezetni egyszerűbb részfeladatokra. Gyakran használható az a technika, hogy a keresési teret több kisebb részre bontjuk, és ezeken végezzük el az optimalizációt. Majd ezen rögzítések egyesítéséből próbáljuk meg meghatározni a teljes optimumot.

### 9.1.2. A 0-1 hátzsák probléma megoldása

A 0-1 hátzsák probléma szerencsére hatékonyan megoldható ezzel a stratégiával. A részproblémákra bontást elvégezhetjük úgy, hogy nem az összes tárgyra próbáljuk meg egyszerre keresni az optimális állapotot, hanem egyszerre csak egyet, az utolsóval foglalkozunk egy lépésen belül:

Tehát ha van  $n$  darab tárgyunk, akkor két esetet kell megvizsgálnunk:

- az  $n$ . tárgyat nem rakjuk a hátzsákba,
- az  $n$ . tárgyat berakjuk a hátzsákba (ez persze csak akkor érdekes, ha egyáltalán belefér).

Természetesen minden esetben meg kell vizsgálnunk a többi  $n-1$  darab tárgy által elérhető optimális pakolás értékét. Ezt egy rekurzív hívással megtehetjük, mivel feltételezzük, hogy egyet kevesebb tárgyra könnyebb lesz megtalálni a megoldást. Amennyiben  $n-1 = 0$ , tehát nincsenek további tárgyaink, akkor pedig eljutottunk egy triviális megoldáshoz (0 darab tárggyal értelemszerűen 0 értékű pakolás elérhető csak el).

A fenti fő lépésekre támaszkodva az előzőeket alapján képzelhetjük el az így előálló részproblémák megoldását. minden ilyen részprobléma esetén adott a még vizsgálandó tárgyak listája, illetve az, hogy még mennyi hely van a zsákban (ez nem azonos a zsák teljes kapacitását jelző  $W_{max}$  értékkel, lehet annál kisebb is):

- A triviális esetek

- 1. triviális eset: egyértelműnek tekinthetjük azt az állapotot, amikor a vizsgálandó tárgyak listája üres (9.1. egyenlet első feltétele). Ilyenkor nyilván nem tudunk berakni semmit a hárításákba, így az optimális pakolás összértéke biztosan 0 lesz.
- 2. triviális eset: szintén egyértelmű az az állapot, amikor a szabad hely 0 (9.1. egyenlet második feltétele). Ilyenkor szintén nem tudunk semmit elhelyezni a zsákban, az optimális pakolás összértéke tehát ilyenkor is 0.
- Nem triviális eset: amennyiben van még egy vagy több vizsgálandó tárgyunk, akkor az alábbiak szerint járunk el:
  - Amennyiben az utolsó még nem vizsgált tárgy nem fér bele a zsákba, akkor számoljuk ki, hogy a maradék elemek milyen optimális összértékkel helyezhetők el a meglévő szabad helyre. Ez célszerűen egy rekurzív hívás lesz, ahol már csak az utolsótól különböző tárgyakat adjuk tovább változatlan szabad hely mellett (9.1. egyenlet harmadik feltétele).
  - Amennyiben az utolsó még nem vizsgált tárgy befér a zsákba, akkor vizsgáljuk meg azt is, hogy vele együtt milyen optimális érték érhető el. Ez egy újabb rekurzív hívás lesz, ahol újra csak az utolsó előtti tárgyakat adjuk tovább, viszont itt már a megmaradt szabad helyet is csökkenteni kell az utolsó tárgy súlyával (hiszen ennyivel kevesebb hely marad a többinek). A rekurzív függvényhívás visszatérési értékét ilyenkor még növelnünk kell az utolsó tárgy értékével, hiszen az is bekerült a zsákba.  
Persze attól, hogy befér, még nem kötelező beraknunk a zsákba, tehát ebben az esetben az optimális érték az előzőleg kiszámolt vele vagy nélküle számított értékek maximuma (9.1. egyenlet negyedik feltétele).

A feladat megoldáshoz bevezetünk egy  $f(t, h)$  függvényt, amely visszaadja, hogy  $h$  rendelkezésre álló szabad hely esetén, ha csak az első  $t$  darab tárgyat akarjuk elhelyezni, akkor mi a maximális pakolás összértéke? Fontos észrevennünk, hogy a függvény nem csak a maximális hárításak kapacitás esetén értelmezhető (hanem  $0 \leq h \leq W_{\max}$ ), ugyanígy, nem csak az összes tárgy esetén van értelme (hanem  $0 \leq t \leq n$ ). Persze ha jól működik a függvény, akkor ebből ki tudjuk majd olvasni a teljes feladat megoldását is, ami  $f(n, W_{\max})$ , teljes a teljes hárításak kapacitást és az összes tárgyat figyelembe véve, mi az optimális pakolás összérték?

Az előzőleg leírt szabályrendszer szerint a függvényt fel tudjuk írni az alábbiak szerint:

$$f(t, h) = \begin{cases} 0, & \text{ha } h = 0 \\ 0, & \text{ha } t = 0 \\ f(t - 1, h), & \text{ha } h > 0 \wedge t > 0 \wedge h < w_t \\ \max\{f(t - 1, h), f(t - 1, h - w_t) + p_t\}, & \text{ha } h > 0 \wedge t > 0 \wedge h \geq w_t \end{cases} \quad (9.1)$$

Ezt a működést valósítja meg a 9.1. algoritmus.

---

## 9.1. Algoritmus A 0-1 hátizsákprobléma megoldása az „oszd meg és uralkodj” elv segítségével

---

**Bemenet:**  $t$  - egész<sup>+</sup> (a részmegoldás kereséskor a tárgyak darabszáma)

**Bemenet:**  $h$  - szám<sup>+</sup> (a részmegoldás kereséskor a rendelkezésre álló szabad hely)

**Kimenet:** . - szám<sup>+</sup> (optimális pakoláshoz tartozó összérték)

```
1: függvény LEGJOBBRÉSZMEGOLDÁS(t, h)
2: ha ($t = 0$) \vee ($h = 0$) akkor
3: vissza 0
4: különben
5: $o_{nem} \leftarrow$ LEGJOBBRÉSZMEGOLDÁS($t - 1, h$)
6: ha $h \geq w_t$ akkor
7: $o_{igen} \leftarrow p_t +$ LEGJOBBRÉSZMEGOLDÁS($t - 1, h - w_t$)
8: $o \leftarrow \max(o_{igen}, o_{nem})$
9: különben
10: $o \leftarrow o_{nem}$
11: elágazás vége
12: vissza o
13: elágazás vége
14: függvény vége
```

**Kimenet:**  $o$  - szám<sup>+</sup> (egy optimális pakolás összértéke)

15: függvény HÁTIZSÁKDNC()

16: vissza LEGJOBBRÉSZMEGOLDÁS( $n, W_{max}$ )

17: függvény vége

---

### Felhasznált változók és függvények

- $n$  : A hátizsákba pakolható tárgyak száma.
  - $W_{max}$  : A hátizsák méréte (a belepakolható tárgyak maximális összsúlya).
  - $w_i$  : A hátizsákba pakolható  $i$ . tárgy súlya ( $1 \leq i \leq n$ ).
  - $p_i$  : A hátizsákba pakolható  $i$ . tárgy hasznossága ( $1 \leq i \leq n$ ).
  - $o_{nem}$  : Az optimális pakolás értéke, ha nem rakjuk be a  $t$ . tárgyat a hátizsákba.
  - $o_{igen}$  : Az optimális pakolás értéke, ha berakjuk a  $t$ . tárgyat a hátizsákba (ha ez lehetséges).
  - $o$  : Egy optimális pakolás összértéke, figyelembevéve minden két fenti lehetőséget.
- 

A rekurzív függvény első paramétere, a  $t$  változó azt mutatja, hogy hány tárgyat vizsgálunk (ami egyben azt is meghatározza, hogy az előzőtől a  $t$ -edik tárgyig foglalkozunk velük, a többi tárgyat nem létezőnek tekintjük). Az ezt követő tárgyak állapotát a rekurzió előző szintjei már beállították, így velük már nem foglalkozunk. Valójában a még vizsgálandó első  $t$  darab tárgyból is csak az utolsót vizsgáljuk közvetlenül. Az ezt megelőző tárgyak állapotát pedig majd a következő rekurziós szintek fogják vizsgálni.

A második  $h$  nevű paraméter pedig azt mutatja, hogy a még elhelyezendő tárgyak (tehát a  $t$ -edik és az ez előttiek) számára mekkora szabad hely maradt. Ez az érték mindenkor kisebb vagy egyenlő, mint  $W_{max}$ , értéke attól függ, hogy az előző rekurziós szinteken elhelyezett tárgyak mennyivel csökkentették a még rendelkezésre álló szabad helyet.

A rekurzív függvény megvalósításában jól elkülöníthetők a fent leírt ágak. Az első sor végzi el a triviális esetek ellenőrzését. Amennyiben a  $t$  értéke nullaval egyenlő (tehát nincs több vizsgálandó tárgyunk), vagy a  $h$  értéke nulla (tehát nincs szabad hely), akkor az optimum nyilvánvalóan az üres zsák, aminek összértéke 0 (3. sor).

Amennyiben vannak még elemek, akkor elkezdjük megvizsgálni a  $t$ -ediket. Nézzük meg, hogy mennyi lenne a vizsgálandó tárgyakhoz tartozó optimum a maradék szabad hely mellett, ha nem rakjuk bele ezt az elemet a zsákba. A megoldás ilyenkor egy egyszerű rekurziós hívással megadható (5. sor), hiszen ennek az értéknek pont ugyanannyinak kell lennie, mint az előtte lévő tárgyakra vonatkozó optimum változatlan szabad hely mellett.

Ezt követően nézzük meg, hogy mi az optimum akkor, ha berakjuk a  $t$ -edik tárgyat a zsákba. Először ellenőrizzük, hogy egyáltalán belefér-e (6. sor). Ha igen, akkor kiszámoljuk az őt megelőző elemek által elérhető optimális összértéket (7. sor). Ez hasonló az előző híváshoz, azonban fontos változás, hogy a maradék elemek számára rendelkezésre álló szabad helyet csökkentettük a  $t$ -edik elem súlyával ( $w_t$ ),

hiszen már az is benne van a zsákban. Ennek megfelelően a rekurzió által visszaadott értéket növeljük a  $p_t$  értékkel, hiszen a  $t$ -edik elem értékével növelnünk kell részmegoldás értékét.

Amennyiben a vizsgált elem befért a zsákba, akkor megvizsgáljuk, hogy vele, vagy nélküle kaptunk jobb lokális optimumot (8. sor). Ez már könnyű, hiszen minden esetben kiszámoltuk az optimális pakolás értéket, ezeket tartalmazzák a  $O_{igen}$  és  $O_{nem}$  változók. Ha nem fér be, akkor nyilván a nélküle értéket kell figyelembe vennünk (10. sor). Bármelyik eset is lesz igaz, az  $o$  változó az optimális részeredményt fogja mutatni ami elérhető a megadott paraméterek alapján. Ez a függvény visszatérési értéke (12. sor).

A fentiek ismeretében a teljes pakolást optimalizáló függvény már könnyen érhető. Egyszerűen meghívja az előbb bemutatott rekurziót  $n$ , illetve  $W_{max}$  értékkel. Tehát a rekurzív függvény első hívásakor az utolsó tárgyat kezdi el vizsgálni, és még rendelkezésére áll a hátizsák teljes kapacitása.

### 9.1.3. A módszer értékelése

A módszer tárhelyigénye itt sem jelentős, a rekurzív hívások miatt ugyan több, mint a nyers erő alkalmazásánál, azonban valószínűleg így sem okoz problémákat. A futásidőt itt már nem tudjuk olyan egyszerűen meghatározni, ugyanis az már függ a bemenő adatoktól. Érdemes észrevenni, hogy önmagában a rekurzív megvalósítás nemhogy javítana, inkább csak ront a helyzetet, hiszen a függvény több úton is eljuthat ugyanazokhoz a részproblémákhoz, és ilyenkor minden esetben újra és újra megoldja azokat.

Az viszont javít a helyzetet, hogy maga az eljárás jóval körültekintőbben működik, mint a nyers erő egyszerű próbálgatása, hiszen a 6. sorban található feltétel miatt az eleve esélytelen részproblémák megoldásával nem foglalkozik (pl. ha már az első tárgy olyan nagy, hogy nem fér bele a zsákba, akkor nem kezdi el feleslegesen vizsgálgatni az ebből adódóan eleve reménytelen  $2^{n-1}$  darab részmegoldást).

A módszer előnyei:

- Van lehetőség valamelyen szinten irányítani a keresést, hogy ne vizsgáljon garantáltan rossz útvonalakat.
- Ennek megfelelően lépésszáma alacsonyabb lehet, mint a nyers erő technikánál látott.

A módszer hátrányai:

- Csak bizonyos szerkezetű megoldások esetén használható hatékonyan a technika.
- Esetenként sok felesleges számítást végezhet, amikor ugyanahoz a részproblémához több ágon is eljut.

Ez utóbbi probléma megoldására szolgálnak a különféle dinamikus programozási módszerek. Ezen belül két fő stratégiát különböztetünk meg: a feljegyzéses (9.2. fejezet) és a táblázatos (9.3. fejezet) módszert.

## 9.2. Feljegyzéses módszer

### 9.2.1. A módszer bemutatása

Az „oszd meg és uralkodj” módszer működése során gyakran lehet belefutni abba a hibába, hogy az algoritmus a részproblémákra bontás során ugyanazt a részproblémát többször is előállítja, és ennek megfelelően többször meg is oldja. Néhány példa:

- Fibonacci számok: klasszikus példa a rekurzióra, amikor az *i*. Fibonacci számot próbáljuk meg egy rekurzív képlettel kiszámítani. Érdemes egyszer lerajzolni a rekurzió által előállított részproblémákat, és jól látszik, hogy nagyobb kezdőparaméterek esetén a többszörösen kiszámított részproblémák száma nagyon magas.
- 0-1 hátzsák probléma: hasonlót tapasztalhatunk a 9.1 fejezetben megismert algoritmus esetében is. Érdemes megvizsgálni az implementált program futását, és itt is jól látható, hogy a rekurziós függvény általában többször is meghívódik ugyanazokkal a paraméterekkel, és ilyenkor értelemszerűen többször is kiszámítja ugyanazokat az értékeket.

Ez a probléma sokkal súlyosabb, mint ahogy elsőre gondolnánk. A rosszul megírt „oszd meg és uralkodj” elvű algoritmusok futásidejének szinte elhanyagolható része az, ami tényleges értékes munkát végez, a számítási kapacitás túlnyomó része feleslegesen számol újra és újra már ismert részeredményeket. Ez a probléma bizonyos feladatok esetében az egész módszer gyakorlati használhatóságát kérdőjelez meg.

Amennyiben szeretnénk mégis ezt az alapvetőt használni a megoldás során, akkor is van lehetőségünk a fenti probléma (teljes vagy részleges) kiküszöbölésére. A fentiek alapján nyilvánvalóan adja magát az ötlet, hogy minden egyes részprobléma kiszámítása után tároljuk el ezt az eredményt valahol, és ha ezt követően a függvény ismét ugyanolyan paraméterekkel hívódna meg, akkor elég ebből a tárolóból elővenni a már kiszámított részeredményt.

Ezt nevezzük „feljegyzéses módszernek”<sup>1</sup>. Maga a tároló tetszőleges típus lehet, egy halmaz, tömb, esetleg szótár (7. fejezet). A tárolás során kulcsként célszerű a rekurzív függvény által kapott paramétereket használni (ez persze csak akkor helyes választás, ha a részeredmény csak ezektől a paraméterektől függ). Így minden függvényhíváskor először azt ellenőrizzük, hogy ilyen paraméterekkel van-e már részeredményünk. Ha igen, akkor ezt visszaadjuk. Ha még nincs, akkor a hagyományos módon kiszámítjuk a részeredményt, és eltároljuk a tárolóban.

### 9.2.2. A 0-1 hátzsák probléma megoldása

A fenti elveknek megfelelő megoldást mutatja a 9.2. algoritmus. Az algoritmus alapvető működése szándékoltan azonos az előző fejezetben megismert 9.1. algoritmuséval. Emiatt a feladat megoldásának részleteire nem is térnénk ki, csak a feljegyzéses módszer jellegzetességeire.

Az implementáció csak két helyen tér el az előző fejezetben megismerttől:

- Részeredmény feljegyzése: alapesetben az algoritmus pontosan ugyanazzal a módszerrel számolja ki a részeredmény értékét, mint a 9.1. algoritmus. Az egyik fontos különbség azonban az, hogy miután megvan a részeredmény pontos értéke, még annak visszaadása (és a függvényből való kilépés) előtt azt eltároljuk. A pszeudokódban ezt a **RÉSZMEGOLDÁSTÁROLÓBAFELJEGYEZ** eljárás meghívásával tettük meg (15. sor). Ennek első paramétere a részfeladat azonosítása (ez jelen esetben nem más, mint a rekurzív függvény hívási paramétereiből alkotott számpár), második paramétere pedig az ehhez tartozó részmegoldás.
- Részeredmény keresése: már a rekurzív függvény elején (5. sor), mielőtt bármit is számolnánk, először megvizsgáljuk, hogy a kiszámítandó részprobléma megoldása szerepel-e már a tárolóban. Ehhez használjuk a **RÉSZMEGOLDÁSTÁROLÓBANKERES** függvényt. Ennek egyetlen paramétere a részprobléma azonosítója (jelen esetben ez a tároláshoz hasonlóan a *t* és a *h* egészek által alkotott számpár). Amennyiben van már ilyen részprobléma a tárolóban, akkor az ahhoz tartozó eredményt azonnal vissza is adja a függvény, így nem végez felesleges számításokat. Amennyiben nincs (a függvény visszatérési értéke ilyenkor  $\emptyset$ ), akkor pedig megindul a már előzőleg megismert kiszámítási folyamat (a végén a tárolással).

<sup>1</sup>memoization

---

## 9.2. Algoritmus A 0-1 hátizsákprobléma megoldása a „feljegyzéses módszer” segítségével

---

**Bemenet:**  $t$  - egész<sup>+</sup> (a részmegoldás kereséskor a tárgyak darabszáma)

**Bemenet:**  $h$  - szám<sup>+</sup> (a részmegoldás kereséskor a rendelkezésre álló szabad hely)

**Kimenet:** . - szám<sup>+</sup> (optimális pakoláshoz tartozó összérték)

```
1: függvény LEGJOBBRÉSZMEGOLDÁS(t, h)
2: ha ($t = 0$) \vee ($h = 0$) akkor
3: vissza 0
4: különben
5: ha RÉSZMEGOLDÁSTÁROLÓBANKERES([t, h]) $\neq \emptyset$ akkor
6: vissza RÉSZMEGOLDÁSTÁROLÓBANKERES([t, h])
7: különben
8: $o_{nem} \leftarrow$ LEGJOBBRÉSZMEGOLDÁS($t - 1, h$)
9: ha $h \geq w_n$ akkor
10: $o_{igen} \leftarrow p_n +$ LEGJOBBRÉSZMEGOLDÁS($t - 1, h - w_t$)
11: $o \leftarrow \max(o_{igen}, o_{nem})$
12: különben
13: $o \leftarrow o_{nem}$
14: elágazás vége
15: RÉSZMEGOLDÁSTÁROLÓBAFELJEGYEZ([t, h], o)
16: vissza o
17: elágazás vége
18: elágazás vége
19: függvény vége
```

**Kimenet:** . - szám<sup>+</sup> (egy optimális pakolás összértéke)

20: függvény HÁTIZSÁKMEMO()

21: vissza LEGJOBBRÉSZMEGOLDÁS( $n, W_{max}$ )

22: függvény vége

---

### Felhasznált változók és függvények

- $n$  : A hátizsákba pakolható tárgyak száma.
  - $W_{max}$  : A hátizsák mérete (a belepakolható tárgyak maximális összsúlya).
  - $w_i$  : A hátizsákba pakolható  $i$ . tárgy súlya ( $1 \leq i \leq n$ ).
  - $p_i$  : A hátizsákba pakolható  $i$ . tárgy hasznossága ( $1 \leq i \leq n$ ).
  - $o_{nem}$  : Az optimális pakolás értéke, ha nem rakjuk be a  $t$ . tárgyat a hátizsákba.
  - $o_{igen}$  : Az optimális pakolás értéke, ha berakjuk a  $t$ . tárgyat a hátizsákba (ha ez lehetséges).
  - $o$  : Egy optimális pakolás összértéke, figyelembevéve minden fenti lehetőséget.
  - RÉSZMEGOLDÁSTÁROLÓBAESZÚR(*kulcs, érték*) : Egy tetszőleges adatszerkezetbe (pl. hasító táblázatba) eltárolja a megadott kulcshoz (amely jelen esetben a  $t$  és  $h$  változók párosa), tartozó egész szám értéket (ami a részmegoldás-keresés eredménye).
  - RÉSZMEGOLDÁSTÁROLÓBAFELJEGYEZ(*kulcs*) : Visszaadja a paraméterként átadott kulcshoz (amely jelen esetben a  $t$  és  $h$  változók párosa) tartozó egész szám értéket (ami egy előzőleg eltárolt részmegoldás-keresés eredménye). Amennyiben ilyen nincs, akkor visszatérési értéke  $\emptyset$ .
-

### 9.2.3. A módszer értékelése

A módszer jelentősen növelte az algoritmus tárígényét, hiszen ezt követően minden vizsgálandó részprob-léma azonosítóját és eredményét el kell tárolni (a rekurzió miatt ez persze a részproblémák részproblémáira is igaz). Klasszikus példája ez annak a kódoptimalizációnak, amikor a futásidő javításáért cserébe gyengébb tárhelykihasználással fizetünk.

A módszer előnyei:

- Jelentősen tudja csökkenteni az egyébként nagy futásidőjű algoritmusok lépésszámát.
- Nagyon egyszerűen implementálható. Ha be tudjuk látni, hogy a gyorsítandó függvény kimenete csak a bemenő paramétereiktől függ, akkor tetszőleges függvénybe beépíthető ez a gyorsítótár, annak működésének pontos ismerete nélkül.

A módszer hátrányai:

- Csak bizonyos szerkezetű megoldások esetén használható hatékonyan a technika (ha sokszor kerül ugyanannak a részfeladatnak a megoldására a vezérlés).
- Jelentős többlet tárhelyigényt jelent.

Érdemes megjegyezni, hogy bár mi a „feljegyzéses módszert” az „oszd meg és uralkodj” technika kiegészítéseként vizsgáltunk, de természetesen attól függetlenül is egy értékes alapelvéről van szó, amivel tetszőleges algoritmus gyorsítása megvalósítható.

#### Megjegyzés

A feljegyzéses módszer a dinamikus programozási módszerek egyik lehetséges változata. A másik gyakori megoldás a táblázat alapú módszer, amellyel a következő fejezet foglalkozik.

## 9.3. Táblázatos módszer

### 9.3.1. Egy bevezető példa

Az előző alfejezetekben láttunk példát az „oszd meg és uralkodj” elvű optimalizációra, illetve ennek gyorsítására, a „feljegyzéses módszerre”. Most megnézünk egy újabb gyorsítási lehetőséget.

Hogy egy könnyen érhető példán lássuk ennek működési mechanizmusát, egy pillanatra tértünk vissza a már jól ismert Fibonacci számokat kiszámító példához.

A feladat ismert [10], az n-edik Fibonacci számot keressük, ami az alábbiak szerint definiálható:

$$f(n) = \begin{cases} 1, & \text{ha } n \leq 1 \\ f(n-2) + f(n-1), & \text{ha } n \geq 2 \end{cases} \quad (9.2)$$

Ez egy klasszikus „oszd meg és uralkodj” algoritmus, egyszerűen implementálható (9.3. algoritmus).

---

### 9.3. Algoritmus A Fibonacci sorozat n-edik elemének előállítása

---

**Bemenet:**  $n$  - egész<sup>+</sup> (hányadik Fibonacci számot keressük?)

**Kimenet:** . - egész<sup>+</sup> (az n-edik Fibonacci szám)

- 1: függvény FIBONACCI(DNC( $n$ ))
  - 2: ha  $n \leq 1$  akkor
  - 3:     vissza 1
  - 4: különben
  - 5:     vissza FIBONACCI(DNC( $n-1$ ) + FIBONACCI(DNC( $n-2$ )))
  - 6: elágazás vége
  - 7: függvény vége
- 

Azonban tudjuk, hogy működése kimondottan pazarló, hiszen ugyanazokat a részeredményeket többször is kiszámolja. Érdemes egyszer átgondolni, hogy a rekurzív hívások miként építenek fel a megoldandó részfeladatokból egy hatalmas fát, amelynek sok azonos részfája van. Ezért alkalmaztuk a „feljegyzéses módszer” technikát, ami azon alapult, hogy eltároltuk a már egyszer kiszámolt részeredményeket. Így azokat nem kellett minden újra kiszámítani, ez pedig jelentősen csökkentette a futásidőt, cserébe egy kicsivel több tárigénnel járt. Ezt a megoldást mutatja a 9.4. algoritmus (az egyszerűség kedvéért az átmeneti tárolóba való mentést és keresést nem írtuk külön függvényekbe, egyszerűen egy **M** tömbben tároljuk a már kiszámolt részeredményeket).

Az előző fejezetben már elemeztük ennek a technikának az előnyeit és a hátrányait, látjuk, hogy jóval hatékonyabb megoldást kaptunk a segítségével. Itt is érdemes egyszer végiggondolni, hogy miként hívja magát a függvény. Tulajdonképpen ugyanazt a fát kezdi el építeni, mint az előző algoritmus, a lényeges különbség azonban az, hogy nincsenek benne azonos részfeladatot megoldó részfák, azokat azonnal a már egyszer kiszámított eredménnyel helyettesítjük (ezt nevezik vágásnak).

Ha a feldolgozást, mint dinamikus folyamat tekintjük, akkor az is jól látszik, hogy a rekurzív függvény először csak hívogatja magát egészen addig, amíg el nem jut a már előre megadott triviális megoldásokig. Ezt követően kiszámolja az **M**[2] értéket, majd egy szinttel visszalép a rekurzióban, és nekiáll kiszámolni az **M**[3] értéket (ez már gyorsan megy, hiszen addigra a 2. és az 1. elem is ismert), ezt követi az **M**[4] számítása, és így tovább... Tehát kezdve azzal, hogy az inicializáláskor mi magunk adtunk értéket az **M**[0] és **M**[1] változóknak, jól látható, hogy az algoritmus tulajdonképpen egyesével végigszámolhatja a **M** tömb elemeit 0-tól  $n$ -ig.

Ez adja a nagyon egyszerű ötletet, hogy hagyjuk el az egész rekurziót, és csak erre a tömb feltöltésre összpontosítunk.

A végeredmény (9.5. algoritmus) látványosan egyszerű az előzőekhez képest. Alapvetően azonos lépésekkel végez, mint az előző megoldás, és a tárhelyigénye is ugyanaz. Azonban itt már nincs szükség felesleges rekurziós lépésekre, illetve maga a rekurzió elhagyása is javíthatja a teljesítményt.

---

#### 9.4. Algoritmus A Fibonacci sorozat n-edik elemének előállítása feljegyzéses módszerrel

---

Bemenet:  $n$  - egész<sup>+</sup> (hányadik Fibonacci számot keressük?)

Kimenet:  $\dots$  - egész<sup>+</sup> (az  $n$ -edik Fibonacci szám)

- 1: függvény FIBONACCI MEMOREK( $n$ )
- 2:   ha  $M[n] = \emptyset$  akkor
- 3:      $M[n] \leftarrow \text{FIBONACCI MEMOREK}(n-1) + \text{FIBONACCI MEMOREK}(n-2)$
- 4:   elágazás vége
- 5:   vissza  $M[n]$
- 6: függvény vége

Bemenet:  $n$  - egész<sup>+</sup> (hányadik Fibonacci számot keressük?)

Kimenet:  $\dots$  - egész<sup>+</sup> (az  $n$ -edik Fibonacci szám)

- 7: függvény FIBONACCI MEMO( $n$ )
- 8:    $M[0..1] \leftarrow 1$
- 9:    $M[2..n] \leftarrow \emptyset$
- 10:   vissza FIBONACCI MEMOREK( $n$ )
- 11: függvény vége

---

#### Felhasznált változók és függvények

- $M$  - Részeredmény tároló.  $n+1$  elemű, egész<sup>+</sup> típusú tömb (kivételesen 0-tól kezdjük az indexelést). Olyan formában kell létrehozni (pl. egy objektum mezőjeként, vagy globális változóként), hogy az egyes függvény hívások között megőrizze az értékét.
- 

---

#### 9.5. Algoritmus A Fibonacci sorozat n-edik elemének előállítása táblázatos módszerrel

---

Bemenet:  $n$  - egész<sup>+</sup> (hányadik Fibonacci számot keressük?)

Kimenet:  $\dots$  - egész<sup>+</sup> (az  $n$ -edik Fibonacci szám)

- 1: függvény FIBONACCI MEMO( $n$ )
- 2:    $M[0..1] \leftarrow 1$
- 3:   ciklus  $i \leftarrow 2$ -től  $n$ -ig
- 4:      $M[i] \leftarrow M[i-1] + M[i-2]$
- 5:   ciklus vége
- 6:   vissza  $M[n]$
- 7: függvény vége

---

#### Felhasznált változók és függvények

- $i$  - Egy egész<sup>+</sup> típusú segédváltozó.
  - $M$  - Részeredmény tároló.  $n+1$  elemű, egész<sup>+</sup> típusú tömb (kivételesen 0-tól kezdjük az indexelést). Olyan formában kell létrehozni (pl. egy objektum mezőjeként, vagy globális változóként), hogy az egyes függvény hívások között megőrizze az értékét.
-

### Megjegyzés

Érdemes megfigyelni, az  $M$  tömb következő elemének kiszámítása mindenkor csak az előző két elem kiszámítását igényli. Tehát a teljes  $n+1$  elemű tömb helyett elég lenne egy összesen két elemű, ahol mindenkor csak az utolsó két értéket tároljuk el. Ezzel a tárhelyigény már konstans értékre csökkenthető.

Vegyük észre, hogy nem egy teljesen új megoldást alkottunk, hanem a kezdetben megadott „oszd meg és uralkodj” algoritmust alakítottuk át két lépésben erre a nagyon egyszerű és hatékony formára a tényleges rekurziós hívások kiküszöbölésével. A szemléletbeli különbséget talán azzal tudnánk a legegyszerűbben szemléltetni, ha rámutatunk, hogy az „oszd meg és uralkodj” jellegű módszerek mindenkor *felülről lefelé* építkeznek, tehát az összetett problémából indulnak ki, és próbálják azt több egyszerűbb részproblémává alakítani. Addig az utolsó megoldásunk éppen fordítva működött, ez *alulról felfelé* építkezik, tehát egyből a kis részproblémák megoldásával kezd, és ezek összeépítésével fokozatosan halad a bonyolultabb, majd végül a végső probléma megoldása felé.

### 9.3.2. A módszer bemutatása

Elsőként vizsgáljuk meg, hogy mikor lehet/érdes használni a dinamikus programozást? Alapvetően két követelményt határozhatunk meg (később, a konkrét példánál ezekre még visszatérünk):

- A feladat legyen *optimális részstruktúrájú*: a feladat optimális megoldása önmagán belül a részfeladatok optimális megoldásait is tartalmazza.
- Legyenek a *részfeladatok átfedők*: a részfeladatokra bontás során többször is merüljön fel ugyanannak a részfeladatnak a megoldása.

Az első feltétel csak azt határozza meg, hogy maga az optimalizálási feladat a kisebb részfeladatok optimumából épüljön fel. A második feltétel is könnyen érthető, ezt már a feljegyzéses módszernél is láttuk. Amennyiben ugyanazt a részfeladatot sokszor kell megoldani, akkor az egy jelentős gyorsítási lehetőséget ad a kezünkbe, ha meg tudjuk oldani, hogy ezeket csak egyszer kelljen kiszámolni (miként erre az imént láttunk két példát is, felülről lefelé és alulról felfelé építkezve is).

### 9.3.3. A 0-1 hátizsák probléma megoldása (optimális érték)

A fentiek szerint próbáljuk megoldani a már ismert hátizsák problémákat dinamikus programozás segítségével. Elsőként vizsgáljuk meg, hogy a feladat megfelel-e a szükséges követelményeknek:

- A feladat optimális részstruktúrájú. A rekurzív hívásokból is jól látható, hogy a teljes hátizsákra és összes tárgyra vonatkozó optimumot visszavezetünk az egyes kevesebb tárgyat és esetenként valamivel kevesebb szabad helyet tartalmazó részproblémák optimális megoldásainak a megkeresésére.
- A részfeladatok gyakran átfedők. Épp ez jelentette a problémát az „oszd meg és uralkodj” módszer esetén, amit jelentősen tudtunk enyhíteni a „feljegyzéses technika” segítségével. Most egy másik módszerrel próbáljuk ezt kezelní.

Ennek megfelelően a megoldás menete:

1. Az optimális megoldást az eddigiekhez hasonlóan egy  $n$  elemű logikai tömbként képzeljük el, aminek az  $i$ . eleme azt mutatja, hogy egy optimális pakolás esetén az  $i$ . tárgy bekerül-e a zsákba vagy sem. A jóslágot pedig az így bekerült tárgyak összértékének tekintjük.
2. A rekurzív módon való definiáláshoz ugyanazt az elvet használjuk, amit már leírtunk az „oszd meg és uralkodj” módszernél. Tehát a teljes feladat megoldását visszavezetjük arra, hogy megvizsgáljuk, hogy az utolsó elemmel vagy anélkül kapunk-e jobb eredményt. Ehhez pedig mindenkor esetben (már ha egyáltalán elfér az utolsó tárgy) egy rekurzív hívással számítjuk ki a többi tárgy optimális pakolásának értékét a részükre maradt hely alapján. A rekurzív megoldás tehát ugyanaz, mint amit a 9.1. egyenletnél már beláttunk.

3. Az így megadott rekurzív megoldást megpróbáljuk kiszámítani alulról felfelé történő módon. Cél-szerű újra rátekinteni a Fibonacci számoknál alkalmazott módszerre, ahogy a rekurziót egy egyszerű táblázat kitöltéssel tudtuk helyettesíteni. Itt valójában ugyanez történik, csak a táblázat nem egy, hanem két dimenziós. A feltöltés során tehát a táblázat elemeit az alábbi szabály alapján tudjuk majd kitölteni:

$$F[t, h] = \begin{cases} 0, & \text{ha } h = 0 \\ 0, & \text{ha } t = 0 \\ F[t - 1, h], & \text{ha } h > 0 \wedge t > 0 \wedge h < w_t \\ \max\{F[t - 1, h], F[t - 1, h - w_t] + p_t\}, & \text{ha } h > 0 \wedge t > 0 \wedge h \geq w_t \end{cases} \quad (9.3)$$

A feltöltés során persze arra figyelnünk kell, hogy hivatkozunk az előző sor és az előző oszlopok elemeire, tehát ennek megfelelő sorrendben kell elvégezni a számításokat. Ezt majd a későbbi algoritmus bemutatja.

4. Az előző lépések még csak az optimális megoldás értékét adják meg. A megoldás előállítását a következő alfejezet (220. oldal) tartalmazza.

Az eddig megszokott feladathoz képest egy apró változást kell tennünk, ezen fejezetben belül kikötjük azt is, hogy az egyes *tárgyak súlya* csak *egész számok* lehetnek, tehát:

- $w_i$  - az *i*. tárgy súlya ( $1 \leq i \leq n$ ), típusa *egész*<sup>+</sup>,

Ez azért fontos, mert így a vizsgálandó szabad hely értékek csak 0 és  $W_{max}$  közötti egészek lehetnek, aminek hamarosan fontos szerepe lesz a táblázat kitöltésekor.

Ha csak az optimális megoldás értékére vagyunk kíváncsiak, akkor a 3. lépésben megadott táblázat kitöltést kell csak implementálnunk. Ezt mutatja a 9.6. algoritmus.

## 9.6. Algoritmus A 0-1 hátzsák probléma megoldása „táblázatos” dinamikus programozás technikával

**Kimenet:** . - *egész*<sup>+</sup> (egy optimális pakolás összértéke)

```

1: függvény HÁTIZSÁKDP()
2: ciklus t ← 0-től n-ig
3: F[t, 0] ← 0
4: ciklus vége
5: ciklus h ← 1-től Wmax-ig
6: F[0, h] ← 0
7: ciklus vége
8: ciklus t ← 1-től n-ig
9: ciklus h ← 1-től Wmax-ig
10: ha h ≥ wt akkor
11: F[t, h] ← max(F[t - 1, h], F[t - 1, h - wt] + pt)
12: különben
13: F[t, h] ← F[t - 1, h]
14: elágazás vége
15: ciklus vége
16: ciklus vége
17: vissza F[n, W]
18: függvény vége

```

### Felhasznált változók és függvények

- $n$  : A hátzsákba pakolható tárgyak száma.
- $W_{max}$  : A hátzsák méréte (a belepakolható tárgyak maximális összsúlya).
- $w_i$  : A hátzsákba pakolható *i*. tárgy súlya ( $1 \leq i \leq n$ ).
- $p_i$  : A hátzsákba pakolható *i*. tárgy hasznossága ( $1 \leq i \leq n$ ).
- $F$  :  $(n+1) \times (W_{max}+1)$  méretű, *szám*<sup>+</sup> típusú tömb. Kivételesen 0-tól kezdjük a tömb indexelését mindenkét dimenzióban.

Érdemes alaposabban szemügyre venni a 9.1 és a 9.3 egyenleteket. A különbség nem elhanyagolható, hiszen az első esetben egy *f* függvényt határozunk meg, itt pedig az *F* táblázat feltöltésének menetét

### 9.1. táblázat. Dinamikus programozás példában vizsgált tárgyak súlyai és értékei

| i     | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|---|---|
| $w_i$ | 2 | 1 | 1 | 1 | 3 | 2 |
| $p_i$ | 4 | 3 | 2 | 8 | 7 | 5 |

látjuk. A mégis nyilvánvaló hasonlóság azonban nem véletlen, hiszen valójában ugyanaz az alapelv minden esetben. Itt ugyan egy táblázatot töltünk fel, de pontosan ugyanazt a képletet használjuk, ami az előző megoldás rekurzív függvényében szerepelt.

Az algoritmus tehát egy  $F$  táblázatot tölt fel. A táblázat  $t$ . sorában és  $h$ . oszlopában lévő érték ( $F[t, h]$ ) minden az  $f(t, h)$  függvény értékét fogja tárolni annak első kiszámítását követően. Tehát annak az optimális pakolásnak az összértékét, amely az első  $t$  darab tárgyat próbálja elhelyezni  $h$  szabad helyen. Mivel a tárgyak száma és a szabad hely is korlátos ( $0 \leq t \leq n, 0 \leq h \leq W_{max}$ ), így ezt a tömböt már előre létrehozhatjuk. Mérete ( $n+1) \times (W_{max}+1)$  (hogy jobban igazodunk a rekurzív definícióhoz, a tömböt most kivételelesen 0-tól indexeljük). Kezdetben a teljes táblázat üres.

Első lépésként beírjuk a táblázatba a triviális megoldások eredményeit, hiszen ezek nem igényelnek további számításokat. A program első ciklusa (2. sor) feltölti a 0 indexű oszlopot nullákkal. Ez tulajdonképpen azt jelzi, hogy a tárgyak számától és adataitól függetlenül, ha nincs már a zsákban szabad hely, akkor az optimális pakolás értéke csak 0 lehet. A következő ciklus (5. sor) feltölti a 0 indexű sort is nullákkal. Ez pedig azt a triviális megoldást képviseli, hogy a szabad helytől függetlenül, ha nincsenek tárgyaink, akkor az optimális pakolás értéke szintén csak 0 lehet. (9.1a. ábra).

Ezt követően megkezdhetjük a táblázat belső elemeinek a kitöltését. A sorrend természetesen fontos, hiszen csak azokat az értékeket tudjuk kiszámolni, amelyek már ismert részeredményeket használnak fel. A rekurzív képletből jól látható, hogy a táblázat  $t$ . sorában lévő értékekhez minden csak a  $t-1$ . sor adataira van szükség. Ugyanígy belátható, hogy a táblázat bármelyik  $h$ . oszlopában lévő értékhez csak a  $h$ -nál kisebb cellákban lévő adatokra lehet szükség. Tehát az algoritmusban megadott ciklusokat használva a feltöltés minden csak a már előzőleg kiszámolt adatokon fog alapulni.

Az egyes cellák adatait kiszámoló 10–14. sorok a fentiek alapján már nem igényelnek különösebb magyarázatot. Jól látható, hogy teljes egészében a már részletesen megvizsgált 9.3. egyenlet lett implementálva. Működése tehát annak megfelelő.

Miután a ciklus végzett, kiveszük a táblázat  $F[n, W_{max}]$  cellájában lévő értéket. A fentiek szerint itt az  $n$  darab tárgy  $W_{max}$  szabad helyen való optimális elrendezésének értékét tároljuk. Ez pedig nem más, mint a teljes feladat megoldása, ezt keressük. Emiatt ez a függvény visszatérési értéke (17. sor).

Egy gyakorlati példán jól látható, hogy mennyire egyszerűen működik a fenti algoritmus. Feltételezzük, hogy van 6 darab tárgunk a 9.1. táblázatban látható súly ( $w_i$ ) és érték ( $p_i$ ) jellemzőkkel. Továbbá azt, hogy a hátizsák kapacitása ( $W_{max}$ ) legyen 4.

Az algoritmus első lépése a táblázat feltöltése a triviális értékekkel (9.1a. ábra). Tehát a 0. sort és a 0. oszlopot feltöltjük 0 értékekkel.

Ezt követően soronként (1-től kezdődő növekvő indexeléssel), és azon belül oszloponként (1-től kezdődő növekvő indexeléssel) megkezdjük a táblázat alulról-felfelé való feltöltését. Maga a kitöltés nagyon egyszerű, mivel csak a 9.3. képletet kell használnunk minden esetben. A kitöltés irányából adódik, hogy a szükséges részeredmények már minden rendelkezésre fognak állni a szükséges pillanatban.

Egy-egy konkrét példát érdemes kiemelni az alábbi esetekre:

- Az éppen vizsgált tárgy nem fér el a rendelkezésre álló helyen. Erre mutat példát a 9.1b. ábra, vagy egy későbbi lépés esetén pl. a 9.1v. ábra. Ilyenkor nincs mérlegelési lehetőségünk, értelemszerűen csak a változatlan szabad hely mellett a vizsgált tárgy nélküli optimális pakolás értékét tudjuk figyelembevenni (és ez lesz az optimum ebben az esetben is). Ezt nem kell újra kiszámolnunk, hiszen ezt az értéket mutatja az  $F[t-1, h]$  érték.
- Az éppen vizsgált tárgy elfér a zsákban és érdemes is belerakni. Erre mutat példát a 9.1q. ábra. Ebben az esetben a  $t$  értéke 4, tehát összesen négy tárgyunk van csak, és ezek közül a negyediket vizsgáljuk. A  $h$  értéke 4, tehát a szabad hely mérete 4. A  $w_t$  értéke 1, tehát a negyedik tárgy súlya kevesebb, mint a szabad hely, így elvileg elhelyezhetjük a zsákban. Ilyenkor célszerű megvizsgálni az alábbiakat:

- Ha azt feltételezzük, hogy berakjuk a zsákba, akkor egyrészt értékként megkapjuk a  $t$ . tárgy értékét ( $p_t$ ), tehát 8-at. Ha berakjuk a 4 méretű zsákba az 1 méretű tárgyat, akkor még marad is 3 szabad helyünk ( $h - w_t$ ). Tudnunk kellene, hogy erre a 3 maradék helyre mi a maradék tárgyak optimális elrendezése, és szerencsére ezt tudjuk, hiszen már kiszámítottuk. Az  $F[3, 3]$  azt mutatja, hogy csak az első 3 tárgy figyelembevételével 3 szabad helyet feltételezve mennyi az elérhető optimális pakolás összértéke. Ez pedig (7) pontosan az, amit keresünk. Összefoglalva: ha a negyedik tárgyat elhelyezzük a zsákban, akkor az optimális érték:  $F[t, h-w_t] + p_t = 7 + 8 = 15$ .
- Ha azt feltételezzük, hogy nem rakjuk be a zsákba, akkor a tárgy nem fogja növelni a pakolás értékét, de cserébe a szabad helyet se csökkenti. Ilyenkor az elérhető optimum tehát nem más, mint az azonos szabad hely mellett, de egyetlen kevesebb tárgy pakolásánál elérhető érték. Ezt pedig az  $F[t-1, h]$  mutatja, ami a kitöltés sorrendje miatt már szintén ismert (9).

Attól függően tehát, hogy a tárgyat berakjuk-e a zsákba vagy sem, 15 illetve 9 jóság lenne elérhető. Ennek megfelelően a kettő közül a nagyobbik lesz az  $F[t, h]$  értéke.

- A harmadik lehetőség pedig az, amikor befér ugyan a következő tárgy a zsákba, azonban nem jutunk vele jobb eredményhez. Erre mutat példát a 9.11. ábra. Ebben az esetben a  $t$  értéke 3, tehát az első három tárggyal dolgozunk, és a harmadikat vizsgáljuk. A  $h$  értéke 3, tehát a szabad hely mérete 3.
  - A harmadik tárgy mérete 1, tehát nyilvánvaló, hogy önmagában elfér a zsákban ( $h \geq w_t$ ). Sőt, még marad is 2 szabad hely ( $w_t - h$ ), tehát tudnunk kell, hogy erre a maradék helyre a maradék tárgyak milyen optimális pakolási értéket tudnak elérni. Ez nem más, mint a táblázat  $F[t-1, h-w_t]$  elemének az értéke, ami már ismert, jelen esetben 4. Ebben az esetben elhez még hozzáadjuk a harmadik tárgy értékét ( $p_t$ ), így összesen 6-os összértéket érhetünk el.
  - Vizsgáljuk meg azt az esetet is, ha nem rakjuk bele a tárgyat a zsákba. Ilyenkor a nélküle elérhető optimális érték  $F[t-1, h]$ , ami már szintén ismert, hiszen előzőleg kiszámítottuk: 7.

Jól látható, hogy ugyan a tárgy belefér a zsákba, de nem érdemes belerakni. Nyerünk ugyan vele  $p_t$  értéket, de az általa elfoglalt terület miatt a többi tárgy által elérhető érték ennél többel csökkenne. Emiatt az optimális pakolás értéke (és ezzel az  $F[3, 3]$  értéke) 7 lesz.

Ugyanazokat a lépéseköt követjük végig az algoritmus futása során, beleérve a legutolsó  $F[n, W]$  érték kiszámítását is. A példában látható, hogy még az utolsó lépésben is sikerült egy jobb megoldást találnunk (9.1y. ábra). Itt már nagyon nehéz lenne fejben visszakövetni, hogy pontosan hogyan is jöttek ki az előző sorok értékei, de elég azt tudnunk, hogy az algoritmus végig jól működött, tehát az itt felhasznált  $F[5, 2]$  és  $F[5, 4]$  értékek valóban a megadott paraméterekkel elérhető optimális eredményt tartalmazzák, és ezekre építve már egyszerűen kiszámolható a végeredmény.

A teljes feladat megoldását tehát a táblázat  $F[n, W]$  cellája tartalmazza a program lefutását követően (9.1z. ábra).

### 9.3.4. A 0-1 hátizsák probléma megoldása (optimális pakolás)

Az optimalizálási feladat kiírásakor nagyvonalúan csak az optimális pakolás értékének keresését igényel-tük, ezt teljesítettük is. A mintapéldában a dinamikus algoritmussal megtudtuk, hogy egy optimális pakolás értéke 16. A gyakorlatban azonban sokszor szükség van magára a tényleges megoldásra is, erre viszont nem kaptunk még választ, az algoritmus nem adta meg az optimális pakolást (tehát azt, hogy konkrétan melyik tárgyat kell belerakni a zsákba és melyeket nem).

Önmagában a dinamikus programunk erre nem tud választ adni, egy kis kiegészítéssel azonban az általa felépített táblázatból ez is kiolvasható. A felépített  $F$  táblázat alapján tehát az optimális pakolás adatait keressük.

A 9.2. ábra a teljes megoldás visszafejtését mutatja be. Az első ábra alapján már könnyen megérthető ennek menete. Egy ciklussal elindulunk a táblázat  $F[n, W]$  elemétől. Ennek értéke a példában 16. Tehát az ideális pakolás 6 tárgy és 4 méretű hátizsák esetén 16 értékkel bír. Elsőként vizsgáljuk meg azt, hogy a hatodik tárgy része lehet-e ennek a pakolásnak (9.2a. ábra).

|         |   | hely |   |   |   |   |
|---------|---|------|---|---|---|---|
|         |   | 0    | 1 | 2 | 3 | 4 |
| tárgyak | 0 | 0    | 0 | 0 | 0 | 0 |
|         | 1 | 0    |   |   |   |   |
|         | 2 | 0    |   |   |   |   |
|         | 3 | 0    |   |   |   |   |
|         | 4 | 0    |   |   |   |   |
|         | 5 | 0    |   |   |   |   |
|         | 6 | 0    |   |   |   |   |

(a) Az inicializálás utáni üres táblázat. 6 tárgyunk van, és a zsák kapacitása 4. A 0. indexű oszlopot feltöltöttük 0-val (tehát ha nincs hely, akkor ez az optimális érték). A 0. indexű sort is feltöltöttük (ha nincsenek tárgyaink, akkor is ez az optimális érték).

|         |   | hely |   |   |   |   |
|---------|---|------|---|---|---|---|
|         |   | 0    | 1 | 2 | 3 | 4 |
| tárgyak | 0 | 0    | 0 | 0 | 0 | 0 |
|         | 1 | 0    | 0 | 0 | 0 | 0 |
|         | 2 | 0    |   |   |   |   |
|         | 3 | 0    |   |   |   |   |
|         | 4 | 0    |   |   |   |   |
|         | 5 | 0    |   |   |   |   |
|         | 6 | 0    |   |   |   |   |

(b)  $t=1, h=1, w_t=2, p_t=4$   
Mivel  $h < w_t$ , ezért a felette lévő mezőt másoljuk át (mivel a 2 súlyú tárgy nem kerülhet bele az 1 méretű zsákba).

|         |   | hely |   |   |   |   |
|---------|---|------|---|---|---|---|
|         |   | 0    | 1 | 2 | 3 | 4 |
| tárgyak | 0 | 0    | 0 | 0 | 0 | 0 |
|         | 1 | 0    | 0 | 0 | 0 | 0 |
|         | 2 | 0    |   |   |   |   |
|         | 3 | 0    |   |   |   |   |
|         | 4 | 0    |   |   |   |   |
|         | 5 | 0    |   |   |   |   |
|         | 6 | 0    |   |   |   |   |

(c)  $t=1, h=2, w_t=2, p_t=4$

Mivel  $h \geq w_t$ , ezért megvizsgáljuk az optimumot vele és nélküle is. Az  $F[t-1, h-2]+4$  a legmagasabb elérhető érték (tehát az 1. tárgy bekerül a zsákba).

|         |   | hely |   |   |   |   |
|---------|---|------|---|---|---|---|
|         |   | 0    | 1 | 2 | 3 | 4 |
| tárgyak | 0 | 0    | 0 | 0 | 0 | 0 |
|         | 1 | 0    | 0 | 4 | 0 | 0 |
|         | 2 | 0    |   |   |   |   |
|         | 3 | 0    |   |   |   |   |
|         | 4 | 0    |   |   |   |   |
|         | 5 | 0    |   |   |   |   |
|         | 6 | 0    |   |   |   |   |

(d)  $t=1, h=3, w_t=2, p_t=4$

Ugyanaz a helyzet mint az előző esetben. Itt is érdekes berakni a tárgyat a zsákba (és persze bele is fér).

9.1. ábra. A 0-1 háitzsák probléma megoldása dinamikus programozással.

| hely |   |   |   |              |          |
|------|---|---|---|--------------|----------|
|      | 0 | 1 | 2 | 3            | 4        |
| 0    | 0 | 0 | 0 | $t-1, h-w_t$ | $t-1, h$ |
| 1    | 0 | 0 | 4 | 4            | $t, h$   |
| 2    | 0 |   |   |              |          |
| 3    | 0 |   |   |              |          |
| 4    | 0 |   |   |              |          |
| 5    | 0 |   |   |              |          |
| 6    | 0 |   |   |              |          |

(e)  $t=1, h=4, w_t=2, p_t=4$

Értelemszerűen az eredmény itt is ugyanaz mint az előzőekben. Hiszen egyetlen tárgyat próbálhatunk egyre nagyobba szabad helyre.

| hely |   |   |              |          |   |
|------|---|---|--------------|----------|---|
|      | 0 | 1 | 2            | 3        | 4 |
| 0    | 0 | 0 | 0            | 0        | 0 |
| 1    | 0 | 0 | $t-1, h-w_t$ | $t-1, h$ |   |
| 2    | 0 |   |              | $t, h$   |   |
| 3    | 0 |   |              |          |   |
| 4    | 0 |   |              |          |   |
| 5    | 0 |   |              |          |   |
| 6    | 0 |   |              |          |   |

(f)  $t=2, h=1, w_t=1, p_t=3$

Egyel több tárgyat vizsgálunk a következő sorban. A második tárgy súlya 1, tehát 1 szabad hely esetén elfér ( $h \geq w_t$ ). Ezért megvizsgáljuk a vele és nélküle elérhető optimumot, ezek közül az előbbit választjuk.

| hely |   |   |              |          |   |
|------|---|---|--------------|----------|---|
|      | 0 | 1 | 2            | 3        | 4 |
| 0    | 0 | 0 | 0            | 0        | 0 |
| 1    | 0 | 0 | $t-1, h-w_t$ | $t-1, h$ |   |
| 2    | 0 | 3 | $t, h$       |          |   |
| 3    | 0 |   |              |          |   |
| 4    | 0 |   |              |          |   |
| 5    | 0 |   |              |          |   |
| 6    | 0 |   |              |          |   |

(g)  $t=2, h=2, w_t=1, p_t=3$

A következő lépésekben ugyanazt a tárgyat vizsgáljuk eggyel nagyobba szabad helyen. Ha felveszük a zsákba, akkor nem fér mellé semmi, így 3 az optimális érték. Viszont ha nem vesszük fel a zsákba, akkor nélküle 4 is elérhető, így ezt választjuk.

| hely |   |   |              |          |   |
|------|---|---|--------------|----------|---|
|      | 0 | 1 | 2            | 3        | 4 |
| 0    | 0 | 0 | 0            | 0        | 0 |
| 1    | 0 | 0 | $t-1, h-w_t$ | $t-1, h$ |   |
| 2    | 0 | 3 | $t, h$       |          |   |
| 3    | 0 |   |              |          |   |
| 4    | 0 |   |              |          |   |
| 5    | 0 |   |              |          |   |
| 6    | 0 |   |              |          |   |

(h)  $t=2, h=3, w_t=1, p_t=3$

Két tárgyat vizsgálunk 3 szabad hely esetén. Itt már látszik, hogy a második tárgy bevétele esetén érjük el az optimális pakolási összértéket.

9.1. ábra. A 0-1 hátizsák probléma megoldása dinamikus programozással. (folytatás)

|         |   | hely |   |   |   |   |
|---------|---|------|---|---|---|---|
|         |   | 0    | 1 | 2 | 3 | 4 |
| tárgyak | 0 | 0    | 0 | 0 | 0 | 0 |
|         | 1 | 0    | 0 | 4 | 4 | 4 |
|         | 2 | 0    | 3 | 4 | 7 | 7 |
|         | 3 | 0    |   |   |   |   |
|         | 4 | 0    |   |   |   |   |
|         | 5 | 0    |   |   |   |   |
|         | 6 | 0    |   |   |   |   |

(i)  $t=2, h=4, w_t=1, p_t=3$

Egyet nagyobb a hely mint az előző esetben, de az optimális érték ugyanennyi (mivel már előzőleg is elérte minden tárgy a zsákban).

|         |   | hely |   |   |   |   |
|---------|---|------|---|---|---|---|
|         |   | 0    | 1 | 2 | 3 | 4 |
| tárgyak | 0 | 0    | 0 | 0 | 0 | 0 |
|         | 1 | 0    | 0 | 4 | 4 | 4 |
|         | 2 | 0    | 3 | 4 | 7 | 7 |
|         | 3 | 0    |   |   |   |   |
|         | 4 | 0    |   |   |   |   |
|         | 5 | 0    |   |   |   |   |
|         | 6 | 0    |   |   |   |   |

(j)  $t=3, h=1, w_t=1, p_t=2$

Most már három tárgyat vizsgálunk. A harmadik elfér a rendelkezésre álló helyre, viszont nélküle jobb eredmény érhető el, így az lesz az optimum.

|         |   | hely |   |   |   |   |
|---------|---|------|---|---|---|---|
|         |   | 0    | 1 | 2 | 3 | 4 |
| tárgyak | 0 | 0    | 0 | 0 | 0 | 0 |
|         | 1 | 0    | 0 | 4 | 4 | 4 |
|         | 2 | 0    | 3 | 4 | 7 | 7 |
|         | 3 | 0    | 3 | 5 |   |   |
|         | 4 | 0    |   |   |   |   |
|         | 5 | 0    |   |   |   |   |
|         | 6 | 0    |   |   |   |   |

(k)  $t=3, h=2, w_t=1, p_t=2$

Egyet több szabad hely esetén már érdemes berakni a zsákba, mivel így érjük el a legjobb eredményt.

|         |   | hely |   |   |   |   |
|---------|---|------|---|---|---|---|
|         |   | 0    | 1 | 2 | 3 | 4 |
| tárgyak | 0 | 0    | 0 | 0 | 0 | 0 |
|         | 1 | 0    | 0 | 4 | 4 | 4 |
|         | 2 | 0    | 3 | 4 | 7 | 7 |
|         | 3 | 0    | 3 | 5 | 7 | 7 |
|         | 4 | 0    |   |   |   |   |
|         | 5 | 0    |   |   |   |   |
|         | 6 | 0    |   |   |   |   |

(l)  $t=3, h=3, w_t=1, p_t=2$

Itt már egyre nehezebb fejben áttekinteni az eredményt. De a táblázat felépítéséből adódik, hogy az új tárgy felvételével, és az így csökkentett szabad hely optimális kitöltésével rosszabb eredményt kapnánk, mintha nélküle a teljes szabad helyre keresnék optimális pakolást.

9.1. ábra. A 0-1 hátizsák probléma megoldása dinamikus programozással. (folytatás)

| hely |   |   |   |   |
|------|---|---|---|---|
| 0    | 1 | 2 | 3 | 4 |
| 0    | 0 | 0 | 0 | 0 |
| 1    | 0 | 0 | 4 | 4 |
| 2    | 0 | 3 | 4 | 7 |
| 3    | 0 | 3 | 5 | 7 |
| 4    | 0 |   |   |   |
| 5    | 0 |   |   |   |
| 6    | 0 |   |   |   |

(m)  $t=3, h=4, w_t=1, p_t=2$

Az optimális eredményt tehát úgy érjük el, ha berakjuk a zsákba.

| hely |   |   |   |   |
|------|---|---|---|---|
| 0    | 1 | 2 | 3 | 4 |
| 0    | 0 | 0 | 0 | 0 |
| 1    | 0 | 0 | 4 | 4 |
| 2    | 0 | 3 | 4 | 7 |
| 3    | 0 | 3 | 5 | 7 |
| 4    | 0 |   |   |   |
| 5    | 0 |   |   |   |
| 6    | 0 |   |   |   |

(n)  $t=4, h=1, w_t=1, p_t=8$

Vizsgáljuk a következő tárgyat. Elfér 1 helyen és jobb értéket ad mint az előzőleg elért optimumok, tehát beválasztjuk.

| hely |   |   |    |   |
|------|---|---|----|---|
| 0    | 1 | 2 | 3  | 4 |
| 0    | 0 | 0 | 0  | 0 |
| 1    | 0 | 0 | 4  | 4 |
| 2    | 0 | 3 | 4  | 7 |
| 3    | 0 | 3 | 5  | 7 |
| 4    | 0 | 8 | 11 |   |
| 5    | 0 |   |    |   |
| 6    | 0 |   |    |   |

(o)  $t=4, h=2, w_t=1, p_t=8$

Követjük a már megismert algoritmust.

| hely |   |   |    |    |
|------|---|---|----|----|
| 0    | 1 | 2 | 3  | 4  |
| 0    | 0 | 0 | 0  | 0  |
| 1    | 0 | 0 | 4  | 4  |
| 2    | 0 | 3 | 4  | 7  |
| 3    | 0 | 3 | 5  | 7  |
| 4    | 0 | 8 | 11 | 13 |
| 5    | 0 |   |    |    |
| 6    | 0 |   |    |    |

(p)  $t=4, h=3, w_t=1, p_t=8$

Követjük a már megismert algoritmust.

9.1. ábra. A 0-1 hátizsák probléma megoldása dinamikus programozással. (folytatás)

|         |   | hely |   |    |    |    |
|---------|---|------|---|----|----|----|
|         |   | 0    | 1 | 2  | 3  | 4  |
| tárgyak | 0 | 0    | 0 | 0  | 0  | 0  |
|         | 1 | 0    | 0 | 4  | 4  | 4  |
|         | 2 | 0    | 3 | 4  | 7  | 7  |
|         | 3 | 0    | 3 | 5  | 7  | 9  |
|         | 4 | 0    | 8 | 11 | 13 | 15 |
|         | 5 | 0    |   |    |    |    |
|         | 6 | 0    |   |    |    |    |

(q)  $t=4, h=4, w_t=1, p_t=8$

Követjük a már megismert algoritmust.

|         |   | hely |   |    |    |    |
|---------|---|------|---|----|----|----|
|         |   | 0    | 1 | 2  | 3  | 4  |
| tárgyak | 0 | 0    | 0 | 0  | 0  | 0  |
|         | 1 | 0    | 0 | 4  | 4  | 4  |
|         | 2 | 0    | 3 | 4  | 7  | 7  |
|         | 3 | 0    | 3 | 5  | 7  | 9  |
|         | 4 | 0    | 8 | 11 | 13 | 15 |
|         | 5 | 0    | 8 | 11 |    |    |
|         | 6 | 0    |   |    |    |    |

(r)  $t=5, h=1, w_t=3, p_t=7$

A vizsgált tárgy nem fér el a rendelkezésre álló szabad helyen, így nincs más választásunk, másoljuk a felette lévő értéket.

|         |   | hely |   |    |    |    |
|---------|---|------|---|----|----|----|
|         |   | 0    | 1 | 2  | 3  | 4  |
| tárgyak | 0 | 0    | 0 | 0  | 0  | 0  |
|         | 1 | 0    | 0 | 4  | 4  | 4  |
|         | 2 | 0    | 3 | 4  | 7  | 7  |
|         | 3 | 0    | 3 | 5  | 7  | 9  |
|         | 4 | 0    | 8 | 11 | 13 | 15 |
|         | 5 | 0    | 8 | 11 |    |    |
|         | 6 | 0    |   |    |    |    |

(s)  $t=5, h=2, w_t=3, p_t=7$

A tárgy nem fér el, így nincs más választásunk, másoljuk a felette lévő értéket.

|         |   | hely |   |    |    |    |
|---------|---|------|---|----|----|----|
|         |   | 0    | 1 | 2  | 3  | 4  |
| tárgyak | 0 | 0    | 0 | 0  | 0  | 0  |
|         | 1 | 0    | 0 | 4  | 4  | 4  |
|         | 2 | 0    | 3 | 4  | 7  | 7  |
|         | 3 | 0    | 3 | 5  | 7  | 9  |
|         | 4 | 0    | 8 | 11 | 13 | 15 |
|         | 5 | 0    | 8 | 11 | 13 |    |
|         | 6 | 0    |   |    |    |    |

(t)  $t=5, h=3, w_t=3, p_t=7$

A tárgy itt már elfér, de mellé más nem. Így pedig nem kapunk optimumot (a tárgy értéke kevesebb mint a nélküle elérhető optimális érték).

9.1. ábra. A 0-1 hátzsák probléma megoldása dinamikus programozással. (folytatás)

|         |   | hely |   |    |    |    |
|---------|---|------|---|----|----|----|
|         |   | 0    | 1 | 2  | 3  | 4  |
| tárgyak | 0 | 0    | 0 | 0  | 0  | 0  |
|         | 1 | 0    | 0 | 4  | 4  | 4  |
|         | 2 | 0    | 3 | 4  | 7  | 7  |
|         | 3 | 0    | 3 | 5  | 7  | 9  |
|         | 4 | 0    | 8 | 11 | 13 | 15 |
|         | 5 | 0    | 8 | 11 | 13 | 15 |
|         | 6 | 0    |   |    |    |    |

(u)  $t=5, h=4, w_t=3, p_t=7$

Ugyanazt az eredményt érjük el vele, mint nélküle.  
Az optimum így egyszerűen megadható.

|         |   | hely |   |    |    |    |
|---------|---|------|---|----|----|----|
|         |   | 0    | 1 | 2  | 3  | 4  |
| tárgyak | 0 | 0    | 0 | 0  | 0  | 0  |
|         | 1 | 0    | 0 | 4  | 4  | 4  |
|         | 2 | 0    | 3 | 4  | 7  | 7  |
|         | 3 | 0    | 3 | 5  | 7  | 9  |
|         | 4 | 0    | 8 | 11 | 13 | 15 |
|         | 5 | 0    | 8 | 11 | 13 | 15 |
|         | 6 | 0    | 8 |    |    |    |

(v)  $t=6, h=1, w_t=2, p_t=5$

A tárgy nem fér el a rendelkezésre álló 1 helyen, így nincs más választásunk, másoljuk a felette lévő értéket.

|         |   | hely |   |    |    |    |
|---------|---|------|---|----|----|----|
|         |   | 0    | 1 | 2  | 3  | 4  |
| tárgyak | 0 | 0    | 0 | 0  | 0  | 0  |
|         | 1 | 0    | 0 | 4  | 4  | 4  |
|         | 2 | 0    | 3 | 4  | 7  | 7  |
|         | 3 | 0    | 3 | 5  | 7  | 9  |
|         | 4 | 0    | 8 | 11 | 13 | 15 |
|         | 5 | 0    | 8 | 11 | 13 | 15 |
|         | 6 | 0    | 8 | 11 | 13 |    |

(w)  $t=6, h=2, w_t=2, p_t=5$

A tárgy itt már elfér, de nem kapunk vele jobb megoldást, mint amikor nem vettük figyelembe.

|         |   | hely |   |    |    |    |
|---------|---|------|---|----|----|----|
|         |   | 0    | 1 | 2  | 3  | 4  |
| tárgyak | 0 | 0    | 0 | 0  | 0  | 0  |
|         | 1 | 0    | 0 | 4  | 4  | 4  |
|         | 2 | 0    | 3 | 4  | 7  | 7  |
|         | 3 | 0    | 3 | 5  | 7  | 9  |
|         | 4 | 0    | 8 | 11 | 13 | 15 |
|         | 5 | 0    | 8 | 11 | 13 | 15 |
|         | 6 | 0    | 8 | 11 | 13 | 13 |

(x)  $t=6, h=3, w_t=2, p_t=5$

A helyzet hasonló, mint az előző lépésben.

9.1. ábra. A 0-1 hátizsák probléma megoldása dinamikus programozással. (folytatás)

|         |   | hely |   |    |    |    |
|---------|---|------|---|----|----|----|
|         |   | 0    | 1 | 2  | 3  | 4  |
| tárgyak | 0 | 0    | 0 | 0  | 0  | 0  |
|         | 1 | 0    | 0 | 4  | 4  | 4  |
|         | 2 | 0    | 3 | 4  | 7  | 7  |
|         | 3 | 0    | 3 | 5  | 7  | 9  |
|         | 4 | 0    | 8 | 11 | 13 | 15 |
|         | 5 | 0    | 8 | 11 | 13 | 15 |
|         | 6 | 0    | 8 | 11 | 13 | 16 |

(y)  $t=6, h=4, w_t=2, p_t=5$

A legutolsó cella vizsgálatakor még találunk egy még jobb eredményt. Ha berakjuk a zsákba a 6. elemet is, akkor egy kicsivel jobb eredményhez jutunk.

|         |   | hely |   |    |    |    |
|---------|---|------|---|----|----|----|
|         |   | 0    | 1 | 2  | 3  | 4  |
| tárgyak | 0 | 0    | 0 | 0  | 0  | 0  |
|         | 1 | 0    | 0 | 4  | 4  | 4  |
|         | 2 | 0    | 3 | 4  | 7  | 7  |
|         | 3 | 0    | 3 | 5  | 7  | 9  |
|         | 4 | 0    | 8 | 11 | 13 | 15 |
|         | 5 | 0    | 8 | 11 | 13 | 15 |
|         | 6 | 0    | 8 | 11 | 13 | 16 |

(z)

Ezzel megkaptuk a végeredményt. A táblázat utolsó cellája mutatja a optimális pakolás értékét.

9.1. ábra. A 0-1 hátizsák probléma megoldása dinamikus programozással. (folytatás)

## 9.7. Algoritmus A 0-1 hátizsákprobléma egyik optimális pakolásának előállítása

**Bemenet:**  $F$  - tömb<szám<sup>+</sup>> (a feltöltött táblázat)  
**Kimenet:** . - tömb<logikai> (egy optimális pakolás adatai)

```

1: függvény HÁTIZSÁKDPEREDMÉNY(F)
2: $O \leftarrow [hamis, hamis, \dots, hamis]$
3: $t \leftarrow n$
4: $h \leftarrow W_{max}$
5: ciklus amíg ($t > 0 \wedge h > 0$)
6: ha $F[t, h] \neq F[t - 1, h]$ akkor
7: $O[t] \leftarrow igaz$
8: $h \leftarrow h - w_t$
9: elágazás vége
10: $t \leftarrow t - 1$
11: ciklus vége
12: vissza O
13: függvény vége

```

### Felhasznált változók és függvények

- $n$  : A hátizsákba pakolható tárgyak száma.
- $W_{max}$  : A hátizsák mérete (a belepakolható tárgyak maximális összsúlya).
- $w_i$  : A hátizsákba pakolható  $i$ . tárgy súlya ( $1 \leq i \leq n$ ).
- $p_i$  : A hátizsákba pakolható  $i$ . tárgy hasznossága ( $1 \leq i \leq n$ ).
- $O$  : Egy optimális pakolás adatai. Logikai tömb, ahol  $O[t]$  azt mutatja, hogy a  $t$ . tárgy benne van-e a zsákban ( $1 \leq t \leq n$ ).
- $F$  :  $(n+1) \times (W_{max}+1)$  méretű, egész számokat tartalmazó tömb, amit kivételesen 0-tól indexelünk. Az előzőleg megismert HÁTIZSÁKDP állítja elő (9.6. algoritmus).

| hely |   |   |    |    |    |
|------|---|---|----|----|----|
|      | 0 | 1 | 2  | 3  | 4  |
| 0    | 0 | 0 | 0  | 0  | 0  |
| 1    | 0 | 0 | 4  | 4  | 4  |
| 2    | 0 | 3 | 4  | 7  | 7  |
| 3    | 0 | 3 | 5  | 7  | 9  |
| 4    | 0 | 8 | 11 | 13 | 15 |
| 5    | 0 | 8 | 11 | 13 | 15 |
| 6    | 0 | 8 | 11 | 13 | 16 |

o: 

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| H | H | H | H | H | I |
|---|---|---|---|---|---|

(a)  $t = 6, h = 4, w_t = 2, p_t = 5$

Ezt az optimális eredményt csak a 6. tárgy zsákba helyezésével érhettük el (mivel  $F[t, h] \neq F[t-1, h]$ ). Ezért eltároljuk, hogy a 6. elem a zsákban van, és átlépünk az  $F[t-1, h-w_t]$  cellába.

| hely |   |   |    |    |    |
|------|---|---|----|----|----|
|      | 0 | 1 | 2  | 3  | 4  |
| 0    | 0 | 0 | 0  | 0  | 0  |
| 1    | 0 | 0 | 4  | 4  | 4  |
| 2    | 0 | 3 | 4  | 7  | 7  |
| 3    | 0 | 3 | 5  | 7  | 9  |
| 4    | 0 | 8 | 11 | 13 | 15 |
| 5    | 0 | 8 | 11 | 13 | 15 |
| 6    | 0 | 8 | 11 | 13 | 16 |

o: 

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| H | H | H | I | H | I |
|---|---|---|---|---|---|

(c)  $t = 4, h = 2, w_t = 1, p_t = 8$

Mivel  $F[t, h] \neq F[t-1, h]$ , ezért a 4. tárgy is benne van a zsákban.

| hely |   |   |    |    |    |
|------|---|---|----|----|----|
|      | 0 | 1 | 2  | 3  | 4  |
| 0    | 0 | 0 | 0  | 0  | 0  |
| 1    | 0 | 0 | 4  | 4  | 4  |
| 2    | 0 | 3 | 4  | 7  | 7  |
| 3    | 0 | 3 | 5  | 7  | 9  |
| 4    | 0 | 8 | 11 | 13 | 15 |
| 5    | 0 | 8 | 11 | 13 | 15 |
| 6    | 0 | 8 | 11 | 13 | 16 |

o: 

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| H | H | H | H | H | I |
|---|---|---|---|---|---|

(b)  $t = 5, h = 2, w_t = 3, p_t = 7$

Mivel  $F[t, h] = F[t-1, h]$ , ezért az 5. tárgy nélkül kaptuk meg az optimális megoldás értékét. Ezért ezt az utat követjük. Az  $O[5]$  emiatt hamis marad.

| hely |   |   |    |    |    |
|------|---|---|----|----|----|
|      | 0 | 1 | 2  | 3  | 4  |
| 0    | 0 | 0 | 0  | 0  | 0  |
| 1    | 0 | 0 | 4  | 4  | 4  |
| 2    | 0 | 3 | 4  | 7  | 7  |
| 3    | 0 | 3 | 5  | 7  | 9  |
| 4    | 0 | 8 | 11 | 13 | 15 |
| 5    | 0 | 8 | 11 | 13 | 15 |
| 6    | 0 | 8 | 11 | 13 | 16 |

o: 

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| H | H | H | I | H | I |
|---|---|---|---|---|---|

(d)  $t = 3, h = 1, w_t = 1, p_t = 2$

Mivel  $F[t, h] = F[t-1, h]$ , ezért a 3. tárgy nélkül kaptuk meg az optimális megoldás értékét ( $O[3]$  hamis).

9.2. ábra. A 0-1 hátizsák problémának megoldásának előállítása dinamikus programozással.

(a)

|   | hely |   |    |    |    |
|---|------|---|----|----|----|
|   | 0    | 1 | 2  | 3  | 4  |
| 0 | 0    | 0 | 0  | 0  | 0  |
| 1 | 0    | 0 | 4  | 4  | 4  |
| 2 | 0    | 3 | 4  | 7  | 7  |
| 3 | 0    | 3 | 5  | 7  | 9  |
| 4 | 0    | 8 | 11 | 13 | 15 |
| 5 | 0    | 8 | 11 | 13 | 15 |
| 6 | 0    | 8 | 11 | 13 | 16 |

o:

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| H | I | H | I | H | I |
|---|---|---|---|---|---|

|   | hely |   |    |    |    |
|---|------|---|----|----|----|
|   | 0    | 1 | 2  | 3  | 4  |
| 0 | 0    | 0 | 0  | 0  | 0  |
| 1 | 0    | 0 | 4  | 4  | 4  |
| 2 | 0    | 3 | 4  | 7  | 7  |
| 3 | 0    | 3 | 5  | 7  | 9  |
| 4 | 0    | 8 | 11 | 13 | 15 |
| 5 | 0    | 8 | 11 | 13 | 15 |
| 6 | 0    | 8 | 11 | 13 | 16 |

o:

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| H | I | H | I | H | I |
|---|---|---|---|---|---|

(a)  $t = 2, h = 1, w_t = 1, p_t = 3$

Most  $F[t, h] \neq F[t-1, h]$ , ezért a 2. tárgyat is berakjuk a hátizsákba. Átlépünk a következő helyre.

(b)

Mivel  $h$  értéke 0 lett, így nem kell tovább vizsgálnunk a táblázatot. Az  $O$  tömb tartalmazza a tárgyak adatát. Az ábra a teljes bejárt útvonalat mutatja.

Elsőként tegyük fel, hogy nem része. Ekkor a 6. tárgy által adott 16 összértéket kell adnia az első 5 tárgynak is ugyanilyen hátizsákméret mellett. Ezért ellenőrizzük az  $F[t-1, h]$  értéket. Mivel ez nem 16, ez azt mutatja, hogy a 6. tárgy nélkül nem lehet ugyanezt az eredményt elérni. Ezért biztos, hogy a 6. tárgy része az optimális pakolásnak, ezt jelöljük az  $O$  tömb utolsó elemének igazra állításával.

Tehát már tudjuk, hogy a hatodik része a pakolásnak, akkor vizsgáljuk meg a többit. Ha összesen 16 volt az optimális összérték, akkor nyilvánvaló, hogy a 6. elem nélkül a többi elemmel is előállítható egy 11 ( $16 - 5$ ) összértékű pakolás. A kérdés csak az, hogy miként. Ez a lépés még szintén a 9.2a. ábra alapján látható, átlépünk a 5. sor 2. oszlopába.

A 9.2b. ábra a következő tárgy meghatározását mutatja, ami egy másik féle lépésre mutat példát. Az  $F[5, 2]$  értéke 11, tehát 5 tárggyal 2 szabad helyet feltételezve legfeljebb 11-es összérték érhető el. Látszik, hogy az  $F[4, 2]$  értéke is ugyanez, tehát ugyanekkor szabad helyen csak az első 4 tárggyal is elérhető ez a 11-es összérték. Tehát az 5. elem nem része az optimális pakolásnak, itt hagyjuk az  $O$  tömb értékét hamisnak.

Könnyen belátható, hogy minden esetben a fenti két lépés egyikét kell alkalmaznunk, és így sorban kideríthetjük, hogy a  $t$ . elem része-e egy megadott optimális megoldásnak, vagy sem. Mindezt addig folytatjuk, amíg a  $t$  vagy a  $h$  értéke 0 lesz. Ebben az esetben leállítjuk a ciklust, és az  $O$  aktuális értéke egy optimális pakolást fog mutatni.

A fentieket általánosítva adhatjuk a 9.7. algoritmust: első körben inicializáljuk az segédváltozókat (2–4. sorok), ehhez az  $O$  tömböt feltöljük hamis értékekkel, majd a  $t = n, h = W_{max}$  pontból kiindulva az alábbi ciklust követjük:

- Ha  $t = 0$  vagy  $h = 0$ , akkor az algoritmust leállítjuk (5. sor).
- Ha  $F[t, h] = F[t-1, h]$ , akkor a  $t$ . tárgy nem része az optimális pakolásnak. Tehát az  $O$  tömböt nem módosítjuk, a  $t$  értékét eggyel csökkentjük (10. sor).
- Ha  $F[t, h] \neq F[t-1, h]$  akkor a  $t$ . tárgy része az optimális pakolásnak. Tehát az  $O$  tömb  $t$ . elemét igazra állítjuk (7. sor). Csökkentjük eggyel a  $t$  értékét (7. sor), és  $w_t$ -vel a  $h$  értékét (8. sor).

A függvény visszatérési értéke az  $O$  tömb, ami egy optimális pakolás adatait tartalmazza.

### 9.3.5. A módszer értékelése

A módszer tehát pontosan ugyanazt az eredményt adja, mint az előzőek, ami nem meglepő, hiszen valójában pontosan ugyanazokat a részeredményeket számolja ki, mint az „oszd meg és uralkodj” illetve a „feljegyzéses módszer”. A lényeges különbség csak az, hogy ebben az esetben a feldolgozás nem felülről lefelé, hanem alulról felfelé halad. Tehát először a kisebb részproblémákat oldjuk meg, ezeket valamilyen formában eltároljuk, majd ezek alapján építjük fel a komplexebb (rész)problémák megoldását.

A módszer előnyei

- A „nyers erő” és a rekurzív módszerekkel összehasonlítva a futásidő általában jóval kedvezőbb. Jó tervezés esetén nincs szükség egy részprobléma többszöri megoldására.
- A megoldás nem tartalmaz rekurziót. Ez növeli a teljesítményt, illetve az algoritmust is egyszerűbbé teszi.
- Bár elsőre talán mágikusnak tűnik, hogy miként működnek a dinamikus programozás alapú megoldások, valójában nagyon egyszerűek. Ha valaki megérte a módszer alapelveit, akkor szinte tetszőleges (a bevezetőben említett feltételeknek megfelelő) rekurzív algoritmust át tud alakítani erre a formára.

A módszer hátrányai

- A „feljegyzéses módszerhez” viszonyítva bizonyos feladatoknál magasabb lehet a lépésszáma. Itt ugyanis az alulról felfelé való építkezés miatt nem minden tudjuk finomhangolni, hogy mely részproblémák kiszámítására van szükség és melyekre nincs.
- A többi megoldáshoz viszonyítva nagyobb tárterületet igényel. Egy nagyobb feladat megoldása során a részmegoldások száma meglehetősen nagy lehet, ezeket valahol minden kell tárolnunk. Leegyszerűsítve azt is mondhatjuk, hogy a dinamikus programozás során általában tárhelyet adunk hatékonyabb futásidőért cserébe.

# 10. fejezet

## Visszalépéses keresés

### 10.1. Visszalépéses keresés stratégia

#### 10.1.1. A módszer bemutatása

##### Visszalépések szerepe a keresésben

A visszalépéses keresés<sup>1</sup> egy olyan problémamegoldási módszer, amely az alábbi esetekben használható a leghatékonyabban:

- a megoldandó feladat több, egymástól csak közvetve függő részfeladat megoldásából áll;
- már a részfeladatok egy részéből is lehet arra következtetni, hogy az azokra adott részmegoldásokkal biztosan nem érhető el a teljes megoldás.

A visszalépéses keresés alapelve, hogy egyesével dolgozzuk fel a megoldandó részfeladatokat. Az első részfeladathoz választunk egy lehetséges részmegoldást, majd nézzük a következőt. Amennyiben találunk olyan részmegoldást, ami kielégíti a második részfeladatot, és nincs ellentmondásban az első részfeladathoz választottal, akkor ezt is rögzítjük, és továbblépünk a következő részfeladathoz. Amennyiben valamelyik részfeladatnál nem találunk egy, az előző feltételeknek megfelelő részmegoldást sem, akkor visszalépünk az előző részfeladathoz, és ott keresünk egy másik lehetőséget.

Amennyiben a részfeladatok és a lehetséges megoldások száma véges, akkor véges számú lépést követően vagy találunk olyan részmegoldásokat, amelyek egyenként kielégítik az összes részfeladatot (tehát találtunk egy teljes megoldást), vagy előáll egy olyan helyzet, hogy az első részfeladatról is vissza kellene lépnünk, mivel az ott lévő minden egyes részmegoldásra beláttuk, hogy egyik se lehet egy teljes megoldás része (tehát a feladatnak nincs megoldása).

A visszalépéses keresést gyakran rekurzív algoritmussal valósítjuk meg, ahol a rekurzió alapelve:

- Visszavezetés: *n* darab részfeladat megoldását úgy keressük meg, hogy rögzítünk egy, az első részfeladatot kielégítő, és az előzőleg talált részmegoldásokat nem kizáró részmegoldást, majd újra meghívjuk a rekurzív eljárást a maradék  $n - 1$  darab részfeladatra. Amennyiben a rekurzió azzal az eredménnyel tért vissza, hogy nem sikerült végleges megoldást találnia, akkor az adott szinten keresünk egy másik lehetséges részmegoldást, és újrahívjuk a rekurziót. Amennyiben nem találunk több lehetőséget, akkor visszalépünk az előző rekurzió szintre.
- 1. triviális eset: a fenti hívás során ha már csak egy darab megoldandó részfeladat maradt, és arra találtunk a fenti feltételeknek megfelelő megoldást, akkor megoldottuk a teljes feladatot.
- 2. triviális eset: ha a legelső szinten nem találunk a fentieknek megfelelő részmegoldást, akkor nincs megoldása a feladatnak.

A módszer működésének ugyan nem előfeltétele, de csak akkor működik hatékonyan, ha már néhány részeredményből el tudjuk dönten, hogy az nem vezethet jó teljes megoldáshoz. Ilyenkor ugyanis nem kell folytatnunk a még hátralévő részfeladatok megoldását, ehelyett azonnal megpróbálhatunk egy másik

---

<sup>1</sup>backtracking

utat keresni az aktuális szinten. Ezzel ideális esetben jelentősen leszűkíthetjük a keresés során megvizsgált tér méretét, ami számottevő sebességnövekedést eredményezhet.

A mohó algoritmuskallal ellentétben egy megadott szinten a továbblépéskor soha nem tudhatjuk, hogy valóban jó lépést tettünk-e. A kizárási feltétel csak abban segít, hogy megadott szinten el tudjuk dönten, hogy az előző eredmények kizárták-e egy részmegoldás kiválasztását. Azt azonban itt még nem tudjuk, hogy a választott megoldáshoz a még hátralévő részfeladatokhoz is találunk-e majd nem kizárt részmegoldásokat.

### Példa visszalépéses keresésre

Példaként vegyük az alábbi feladatot: Egy építkezésen 6 különböző feladatra kell embert találnunk: „tervezés”, „irányítás”, „beszerzés”, „ellenőrzés”, „engedélyezés”, „értékesítés”, ahol az egyes feladatokra az alábbi személyek alkalmasak:

- „tervezés” feladatra alkalmas: „Miklós”, „Klaudia”;
- „irányítás” feladatra alkalmas: „Zsolt”, „Miklós”;
- „beszerzés” feladatra alkalmas: „András”;
- „ellenőrzés” feladatra alkalmas: „András”, „Pál”, „Zsolt”;
- „engedélyezés” feladatra alkalmas: „András”, „Géza”;
- „értékesítés” feladatra alkalmas: „Géza”, „Miklós”.

A feladatunk az, hogy rendeljünk az egyes feladatokhoz egy-egy személyt az arra alkalmasak közül, olyan módon, hogy egy személy csak egy feladatot vállalhat.

A 10.1. ábra alapján érdemes végignézni, hogy pontosan milyen lépések követik egymást, ez alapján az alapelvek már jól érthetők. Az előző elvi leírásban is megjelenő lépések itt is jól láthatóak, pl.:

- Részmegoldás rögzítése és továbblépés: ez látható már az első lépéskor is (10.1a. ábra). Amennyiben az adott szinten találunk olyan részmegoldást, amelyet nem zárnak ki az előző eredmények, akkor feltételezzük, hogy ez része lehet egy teljes megoldásnak. Ezért ezt rögzítjük, és folytatjuk a keresést a következő szinten (szintek alatt a részfeladatokat értjük).
- Szinten belül új megoldás keresése: a 10.1d. ábrán is látható, hogy az adott szinten az elsőként megvizsgált részeredmény nem választható, mivel azt már kizárta az egyik előző választásunk. Ilyenkor még ugyanezen a szinten maradva megpróbálunk más részmegoldást keresni.
- Visszalépés: az első visszalépés a 10.1i. ábrán látható. Mindezt az okozza, hogy az adott szinten belül nem találtunk egy részmegoldást sem, amelyet ne zártak volna ki az előző választásaink, így nem tudtunk továbblépni a következő szintekre. Mivel az adott szinten belül se tudunk már új részmegoldást keresni, így nyilvánvaló, hogy az eddig követett út nem vezethet jó végeredményhez, emiatt visszalépünk egy szintet. Az pusztán véletlen egybeesés, hogy éppen az utolsó szintről fordultunk vissza, a példában számos eset látható, amikor már egy köztes szintről is vissza kellett lépnünk (pl. 10.1j. ábra, 10.1l. ábra).
- Teljes megoldás: a 10.1u. ábrán látható, hogy találtunk megfelelő részmegoldást az utolsó szinten. Mivel ide csak úgy juthattunk, hogy már az összes előző szintet megoldottuk, így ez a teljes feladat megoldását is jelenti.

Egyedül arra nem láttunk példát, hogy mi történik akkor, ha nincs megoldása a feladatnak. De belátható, hogy amennyiben újabb visszalépések után ismét visszajutnánk a legelső szintre, akkor az azt jelentené, hogy sem a „Miklós”, sem pedig a „Klaudia” nevű jelentkezővel nem sikerült a teljes feladatra megoldást találni. Más lehetőségünk pedig nincs, így nyilván nincs megoldása a feladatnak.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
| Zsolt    |           |           |            |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(a) Első szinten választjuk az első lehetséges részmegoldást, Miklóst. Mivel megfelel a feltételeknek, továbblárnunk a következő szintre.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
| Zsolt    |           |           |            |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(b) Következő szinten választjuk az elsőt, Zsoltot. Miklóssal nem zárják ki egymást, így elfogadható. Továbblárnunk a következő szintre.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
| Zsolt    |           |           |            |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(c) A harmadik szinten választhatjuk Andrászt, mivel nem zárják ki az előző választások. Továbblárnunk a következő szintre.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
| Zsolt    |           |           |            |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(d) A negyedik szinten András lenne az első lehetőség, de őt nem választhatjuk, mivel az előző szinten már adtunk neki munkát. Ezért keresünk újabb részmegoldást a szinten belül.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
| Zsolt    |           |           |            |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(e) Ugyanezen a szinten a következő lehetőség Pál, őt választhatjuk. Továbblárnunk a következő szintre.

10.1. ábra. Visszalépéses keresés mintapélda (feladatok kiosztása).

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(f) Az ötödik szinten András ismét nem választható, hiszen ő már foglalt. Keresünk itt másik megoldást.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(g) A következő elem (Géza), azonban már jó, emiatt ezt rögzítjük, és folytatjuk a következő szinttel.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(h) Az utolsó szint első lehetséges részmegoldása nem megfelelő, hiszen az előző szinten is Gézát választottuk.  
Így a szinten belül keresünk tovább.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(i) Az utolsó szint második lehetséges részmegoldása sem megfelelő. Mivel így nem tudunk előrelépni, illetve a szinten belül sincs már további lehetőség, ezért egy szinttel visszalépünk.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(j) Azonban már itt sincs további választási lehetőség, ezért itt is egy szinttel visszalépünk.

10.1. ábra. Visszalépéses keresés mintapélda (feladatok kiosztása). (folytatás)

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(k) A negyedik szintre visszatérve, itt még van egy újabb lehetőség. Zsoltot azonban már választottuk, így visszalépünk az előző szintre.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(l) Az előző szinten nincs más választási lehetőség, így innen is visszalépünk.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(m) A második szinten megpróbálhatnánk Miklóst választani, azonban az első szint aktuális részeredménye ezt kizárja. Így innen is visszalépünk.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(n) A legelső szinten járunk ismét. Itt még van másik lehetőség, így itt megpróbáljuk Klaudiát választani.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(o) A második feladatra Zsolt megfelelő lehet. Lépünk tovább.

10.1. ábra. Visszalépéses keresés mintapélda (feladatok kiosztása). (folytatás)

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(p) Harmadik szinten értelemszerűen Andrást választjuk (ezt nem ábrázoltuk). A negyedik szinten András emiatt már nem választható. Így szinten belül keresünk mást.

|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(q) Pál viszont itt megfelelő, így léphetünk tovább.

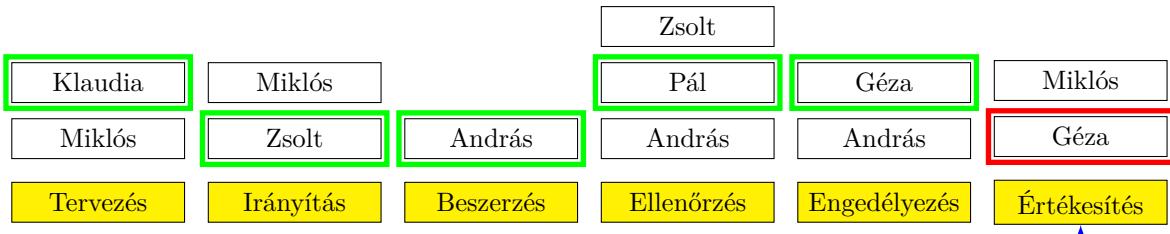
|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(r) András itt sem jó, így keresünk a szinten belül valaki mást.

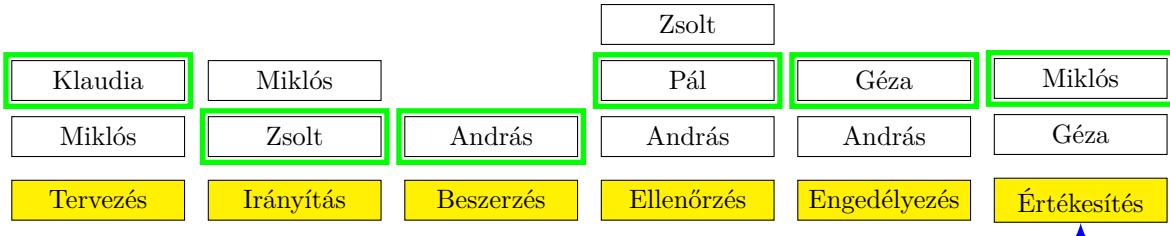
|          |           |           |            |              |             |
|----------|-----------|-----------|------------|--------------|-------------|
|          |           |           | Zsolt      |              |             |
| Klaudia  | Miklós    |           | Pál        | Géza         | Miklós      |
| Miklós   | Zsolt     | András    | András     | András       | Géza        |
| Tervezés | Irányítás | Beszerzés | Ellenőrzés | Engedélyezés | Értékesítés |

(s) Géza még szabad, így őt választhatjuk.

10.1. ábra. Visszalépéses keresés mintapélda (feladatok kiosztása). (folytatás)



(t) Az utolsó részfeladatra Géza így már nem választható. Így a szinten belül keresünk megoldást.



(u) Miklóst viszont nem zárják még ki az előző választások. Ezzel megoldást találtunk az utolsó részfeladatra is, így befejezhetjük a keresést, megvan egy lehetséges teljes megoldás.

10.1. ábra. Visszalépéses keresés mintapélda (feladatok kiosztása). (folytatás)

A 10.1. ábrán látható lépésekkel lefutó algoritmus végeredménye az alábbi feladatkiosztás:

- „tervezés” feladatra: „Klaudia”,
- „irányítás” feladatra: „Zsolt”,
- „beszerzés” feladatra: „András”,
- „ellenőrzés” feladatra: „Pál”,
- „engedélyezés” feladatra: „Géza”,
- „értékesítés” feladatra: „Miklós”.

Az eredmény nem tartalmaz arra vonatkozó információt, hogy vannak-e még további megoldások (akár lehetnek is), az első helyes eredmény megtalálása után ugyanis leállt az algoritmus futása.

### 10.1.2. Visszalépéses keresés általános alakja

#### Bemenet és kimenet

Az alapely tehát az, hogy folyamatosan építsünk egy utat a végső megoldás felé, menet közben elhagyva azokat a lehetséges leágazásokat, amelyek biztosan nem vezetnek jó eredményhez.

A példán láthattuk egy konkrét feladat megoldásának lépéseit, ez alapján próbáljuk meg leírni a módszer általános alakját. Ehhez első lépésként a bemutatott példában látható bemeneteket (feladatok, alkalmazások, személyek), és kimeneteket (van-e megoldás, és ha igen, akkor kit melyik munkához rendeltünk), illetve a végrehajtandó lépéseket próbáljuk meg általánosítani!

Egy visszalépéses keresési feladat bemenetét az alábbiakkal adhatjuk meg:

- $n$ : a megoldandó részfeladatok száma;
- $M_{szint}$ : a  $szint$ -edik részfeladat esetében rendelkezésre álló lehetséges részmegoldások száma (ahol  $1 \leq szint \leq n$ );
- $R_{szint,i}$ : a  $szint$ -edik részfeladat  $i$ . lehetséges megoldása ( $1 \leq szint \leq n$  és  $1 \leq i \leq M_{szint}$ ). A lehetséges megoldások tetszőleges típusúak lehetnek, pszeudokódokban  $\mathbf{T}$  típussal fogjuk ezt jelölni.

A visszalépéses keresés kimenete:

- *van*: logikai érték, ami azt mutatja, hogy találtunk-e teljes megoldást;
- *E*: az eredményeket tartalmazó vektor, ahol *E<sub>i</sub>* az *i*. részfeladat esetében az algoritmus által talált részmegoldás ( $1 \leq i \leq n$ ).

Fontos észrevennünk, hogy maga az alapely sokféle feladat megoldására alkalmassá tehető. Hasonló a helyzet, mint a már ismert lineáris keresésnél, ahol maga az algoritmus a tényleges keresési feltételtől függetlenül minden ugyanaz volt, pusztán egy függvény paraméterrel jelöltük, hogy pontosan milyen feltételnek kell megfelelnie a keresett elemek. A visszalépéses keresésnél is hasonló technikát alkalmazunk annyi különbséggel, hogy itt a keresési feltételt két különböző függvényel fogjuk meghatározni [12]:

- *f<sub>t</sub>(szint, r)*: egy függvény, ami azt határozza meg, hogy a *szint*-edik részfeladat esetében lehetséges megoldás-e az *r*?
- *f<sub>k</sub>(szint, r, E)*: azt határozza meg, hogy a *szint*-edik részfeladat esetében választhatjuk-e az *r* részmegoldást, amennyiben az előző szinteken az *E* vektorban található részmegoldásokat választottuk.

A két függvény közötti lényeges különbség tehát az, hogy az *f<sub>t</sub>* az egyes részfeladatokat önmagukban kezeli, pusztán azt vizsgálja, hogy a többi megoldástól függetlenül egy részeredmény megoldhatja-e az adott részfeladatot. Az *f<sub>k</sub>* függvény azonban már figyelembe veszi azt is, hogy a visszalépéses keresésnek több részfeladatra is megoldást kell találnia, így ezzel már azt vizsgálhatjuk meg, hogy a vizsgált részmegoldást választhatjuk-e az eddig választott részmegoldások ismeretében?

#### Megjegyzés

A visszalépéses keresés megvalósításától függően az *f<sub>t</sub>* függvény sokszor el is hagyható. Amennyiben úgy választjuk meg a bemeneteket (*M<sub>szint</sub>*, illetve *R<sub>szint,j</sub>*), hogy eleve csak azokat a részmegoldásokat soroljuk fel, amelyek bármilyen esetben kiválaszthatók a hozzájuk tartozó részfeladat megoldásához, akkor az *f<sub>t</sub>* azonosan igaz függvény lesz, és így értelemszerűen szükségtelen. Bizonyos feladatoknál azonban célszerű lehet egy külön függvényel vizsgálni ezt a kérdést, emiatt a mi megvalósításunkban megtartjuk az *f<sub>t</sub>* függvényt.

### Tetszőleges megoldás keresése

A fenti bemeneteken dolgozó, visszalépéses keresésen alapuló általános megoldást mutatja be a 10.1. algoritmus.

Mint látható, maga az alapely meglehetősen egyszerűen leírható. Rekurzív formában próbáljuk megoldani a problémát, ahol a rekurzió alapfeltételeit az előzőekben már ismertetett módon határoztuk meg. A rekurzió minden egyes végrehajtása a paraméterként megadott szintre próbál egy megoldást keresni.

A lokális *i* változó szerepe, hogy ez fogja végigvizsgálni a paraméterként átadott *szint*-edig részfeladat lehetséges részmegoldásait. A 2. sorban adunk ennek kezdőértékét, tehát az első lehetséges megoldással fog először próbálkozni.

Egy ciklus végignézni az egyes lehetséges részmegoldásokat. A 3. sorban látható ciklusfeltétel felelős azért, hogy a ciklus csak addig fussen, amíg:

- nem találtunk-e egy teljes megoldást, hiszen ezt követően már felesleges lenne a további lehetőségek vizsgálata;
- van-e még az adott szinten belül lehetséges részmegoldás (ha nincs, akkor már nincs mit vizsgálni).

Amennyiben lefut a ciklusmag, akkor első körben növeljük az *i* változó értékét (4. sor), hogy az valóban a következő, megvizsgálandó részmegoldás sorszámát tartalmazza.

Ezt követően ellenőriznünk kell, hogy a vizsgált részmegoldás megfelelő lehet-e a megadott szinten. Ez két függvény hívását jelenti. Az 5. sorban látható az előzőleg bemutatott *f<sub>t</sub>* függvényé, ami azt vizsgálja, hogy a megadott szinten (*szint*) az éppen vizsgált részmegoldás (*R<sub>szint,i</sub>*) kiválasztható-e.

---

### 10.1. Algoritmus Visszalépéses keresés általános esete

---

Bemenet:  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált részprobléma száma)

Bemenet:  $E$  - tömb<T> (az eddig választott részmegoldásokat tárolja)

Bemenet:  $van$  - logikai (eddig talált-e már teljes megoldást)

Kimenet:  $E$  - tömb<T> (a részmegoldások kiegészítve az aktuális szint esetleges megoldásával)

Kimenet:  $van$  - logikai (talált-e már teljes megoldást, kiegészítve az aktuális szint vizsgálatával)

1: eljárás VISSZALÉPÉSESKERESÉS( $szint$ , címszerint  $E$ , címszerint  $van$ )

2:      $i \leftarrow 0$

3:     ciklus amíg  $\neg van \wedge i < M_{szint}$

4:          $i \leftarrow i + 1$

5:         ha  $f_t(szint, R_{szint,i})$  akkor

6:             ha  $f_k(szint, R_{szint,i}, E)$  akkor

7:                  $E_{szint} \leftarrow R_{szint,i}$

8:                 ha  $szint = n$  akkor

9:                      $van \leftarrow igaz$

10:                 különben

11:                 VISSZALÉPÉSESKERESÉS( $szint+1$ ,  $E$ ,  $van$ )

12:                 elágazás vége

13:                 elágazás vége

14:                 elágazás vége

15:             ciklus vége

16:     eljárás vége

---

#### Felhasznált változók és függvények

- $i$  : Egy egész<sup>+</sup> típusú segédváltozó.
  - $n$  : A megoldandó részfeladatok száma.
  - $M_{szint}$  : A lehetséges részmegoldások száma a megadott szinten.
  - $R_{szint,i}$  : Az  $i$  részmegoldás a megadott szinten.
  - $f_t$ ,  $f_k$  : A keresési feltételeket meghatározó függvények.
-

Amennyiben a válasz igen, akkor következik annak a vizsgálata, hogy a vizsgált megoldást nem zárja-e ki egy előzőleg már rögzített részmegoldás. Erre használható a már bemutatott  $f_k$  függvény, amelynek paraméterei a szint, amelyiken dolgozunk ( $szint$ ), a részmegoldás, amit éppen vizsgálunk ( $R_{szint,i}$ ), illetve az eddig rögzített eredmények sorozata ( $E$ ).

Ha minden függvény igaz értékkel tért vissza, az azt jelenti, hogy az  $i$ . részmegoldás az adott szinten, az előző részmegoldások ismeretében jónak tűnik, ezért elmentjük ezt az értéket az  $E$ -be (7. sor).

Ezt követi egy vizsgálat (8. sor), hogy már az utolsó szinten járunk-e. Ha igen, akkor a  $van$  változó értékét igazra állítjuk, ezzel jelölve, hogy találtunk egy teljes megoldást (amit éppen az  $E$  tartalmaz). Érdemes megfigyelni, hogy a  $van$  változó új értékének hatására kilépünk az utolsó szinten lévő ciklusból, és ugyanez fog történni az előző rekurziós szinteken is, tehát az első teljes megoldás megtalálása után a lehető leggyorsabban kilépünk a visszalépéses keresésből.

Egy másik lehetőség, hogy találtunk ugyan egy részmegoldást, azonban a 8. sorban található vizsgálat alapján arra jutunk, hogy még nem oldottuk meg az egész feladatot. Ilyenkor folytatnunk kell a keresést a következő részfeladattal. A rekurzió szabályainak megfelelően ezt a 11. sorban látható rekurzív hívással tesszük meg. Itt az eljárás meghívja önmagát, paraméterként átadja a következő szint számát ( $szint+1$ ), illetve az eddig talált részeredményeket.

Az eljáráshívásból előbb-utóbb visszatér a vezérlés. Ilyenkor a  $van$  változó értékéből tudhatjuk meg, hogy mi volt ennek a visszatérésnek az oka. Ha a változó értéke *igaz*, az azt jelenti, hogy egy későbbi szinten találtunk egy teljes megoldást, tehát ezen a szinten is kiléphetünk. Ha azonban a változó értéke *hamis*, az csak annyit jelent, hogy a következő szinteken nem sikerült eljutni egy teljes megoldáshoz, tehát az itt feltételezett  $R_{szint,i}$  eredményünk mégse lesz jó választás, az nem vezet végleges megoldáshoz. Ezért első körben próbálkozunk egy újabb részeredmény választással (amíg a ciklusfeltétel igaz), illetve ha ez nem vezet eredményre, akkor kilépünk az eljárásból, ezzel visszaadva a vezérlést az előző szintnek, hogy az tudjon más részmegoldást választani (a  $van$  változó alapértelmezetben *hamis*, tehát ilyenkor ezt külön nem kell beállítanunk).

A fenti leírás talán elsőre összetettnek tűnik, de mindez annak köszönhető, hogy egy meglehetősen általános formát választottunk. A későbbiekben látni fogjuk, hogy ugyanezt az algoritmust fogjuk felhasználni különféle, esetenként egymástól jelentősen különböző feladatok megoldásához. Az alapelv ugyanis végig ugyanaz marad, de a bemenő paraméterek, illetve az  $f_t$  és  $f_k$  függvények helyes megválasztásával teljesen eltérő feladatokat is meg tudunk oldani (feladat kiosztás, Sudoku feladat megoldás, 8 királynő elhelyezése sakktáblán, stb.).

## Egymást kölcsönösen kizáró részfeladatok

Az előzőleg megismert algoritmus és változatai meglehetősen általánosnak tekinthetőek, azok implementációja nehézségekbe ütközhet. Érdemes lehet specializálni a megoldást, ennek megfelelően két fő csoportra bontjuk a visszalépéses kereséssel megoldható feladatokat, majd ezekre adunk egy-egy, már egyszerűbben implementálható megoldást.

Elsődlegesen, a már bemutatott  $f_k$  függvény formája alapján (tehát, hogy miként viszonyulnak egymáshoz az egyes részfeladatok) elkülöníthetjük egymástól az alábbi feladattípusokat:

- Kölcsönösen kizáró feltétel: ebben az esetben egy újabb részfeladat vizsgálatakor annak eldöntéséhez, hogy egy ottani részmegoldást kizárnak-e előző részmegoldások, elég az előzőleg már kiválasztott részmegoldás jelöléssel páronkénti összehasonlítást végezni. Tipikusan ilyen volt a példaként bemutatott feladat is, hiszen az  $i$ . részmegoldás vizsgálatához elég külön-külön megvizsgálni az előző ( $i-1$ ) részmegoldást, hogy a vizsgált személyt már hozzárendeltük-e valamelyik előző részfeladathoz.
- Összetett kizárási feltétel: ebben az esetben nem elég a páronkénti összehasonlítás, mivel a részmegoldások egy nagyobb halmazáról tudjuk csak eldönteni, hogy azokkal folytatható-e a keresés vagy sem. Az elfogadási szabály tetszőleges lehet, pl. a háitzsákba pakolás feladatnál (lásd 10.1.3. alfejezet), ahol egy új tárgy vizsgálatakor nem elég az előzőleg választott tárgyakkal egyesével összehasonlítani, hanem ki kell számolni az előzőleg kiszemelt tárgyak össztömegét, és az alapján hozni meg a döntést. Az előzőleg megismert algoritmusok valójában ezt a módszert követik.

Az első esetben tudjuk, hogy a feltétel mindenkor minden részfeladat vizsgálatát igényli, így magát az  $f_k$  függvényt is jóval egyszerűbben adhatjuk meg. Hiszen nincs szükség az összes eddig rögzített részered-

mény átadására, elég paraméterként megadni két szintet, illetve az ott vizsgálandó két részeredményt, és ez alapján már el lehet dönten, hogy ezek egymást kizárták-e vagy sem.

Ezt mutatja be a 10.2. algoritmus.

---

## 10.2. Algoritmus Visszalépéses keresés kölcsönösen kizáró esetben

---

**Bemenet:**  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált részprobléma száma)

**Bemenet:**  $E$  - tömb<T> (az eddig választott részmegoldásokat tárolja)

**Bemenet:**  $van$  - logikai (eddig talált-e már teljes megoldást)

**Kimenet:**  $E$  - tömb<T> (a részmegoldások kiegészítve az aktuális szint esetleges megoldásával)

**Kimenet:**  $van$  - logikai (talált-e már teljes megoldást, kiegészítve az aktuális szint vizsgálatával)

```
1: eljárás VISSZALÉPÉSESKERESÉS($szint$, címszerint E , címszerint van)
2: $i \leftarrow 0$
3: ciklus amíg $\neg van \wedge i < M_{szint}$
4: $i \leftarrow i + 1$
5: ha $f_t(szint, R_{szint,i})$ akkor
6: $k \leftarrow 1$
7: ciklus amíg $k < szint \wedge f_k(szint, R_{szint,i}, k, E_k)$
8: $k \leftarrow k + 1$
9: ciklus vége
10: ha $k = szint$ akkor
11: $E_{szint} \leftarrow R_{szint,i}$
12: ha $szint = n$ akkor
13: $van \leftarrow igaz$
14: különben
15: VISSZALÉPÉSESKERESÉS($szint+1$, E , van)
16: elágazás vége
17: elágazás vége
18: elágazás vége
19: ciklus vége
20: eljárás vége
```

---

### Felhasznált változók és függvények

- $i$  : Egy egész<sup>+</sup> típusú segédváltozó.
  - $n$  : A megoldandó részfeladatok száma.
  - $M_{szint}$  : A lehetséges részmegoldások száma a megadott szinten.
  - $R_{szint,i}$  : Az  $i$  részmegoldás a megadott szinten.
  - $f_t$ ,  $f_k$  : A keresési feltételeket meghatározó függvények.
- 

Az alapváltozathoz képest két lényeges változás látható. Egyszerű némileg megváltozott az  $f_k$  függvény paramétere az alábbiak szerint:

- $f_k(szint_a, r_a, szint_b, r_b)$ : visszatérési értéke akkor igaz, ha a  $szint_a$  szinten az  $r_a$  részmegoldás választása nem zárja ki a  $szint_b$  szinten az  $r_b$  részmegoldás választását.

Mivel ez csak két szintet tud összehasonlítani, így ezt egy ciklusba kellett építeni, hogy az összes eddig választott részeredményt ellenőrizze az algoritmus.

A 7. sorban található ciklus megfelel a lineáris keresés programozási tételeiben megismert ciklusnak. A  $k$  változó segítségével végigszaladunk az előző szintek részmegoldásain (már ha van ilyen egyáltalán). minden szinten ellenőrizzük, hogy az ott rögzített részmegoldás nem zárja-e ki azt, hogy az adott szinten az aktuálisan vizsgáltat válasszuk.

A lineáris kereséshez hasonlóan itt is ellenőriznünk kell, hogy miért léptünk ki a ciklusból (10. sor). Ha a  $k$  változó értéke kisebb, mint az előző szintek száma, akkor a ciklus futása során az  $f_k$  valahol hamis értéket adott vissza, tehát ez az eredmény nem lehet egy jó út része. Ellenkező esetben azonban megállapíthatjuk, hogy az előzőleg választott részeredmények nem zárják ki az aktuálisan vizsgáltat, tehát ezt rögzíthetjük.

Az összes, illetve az optimális megoldás keresése természetesen ebben a formában is megadható, ehhez ugyanazokat a módosításokat kell megtenni, mint amit az eredeti algoritmusnál is láthattunk.

### 10.1.3. Visszalépéses keresés példák

#### Megoldható feladatok osztályozása

Az előzőekben láttuk, hogy a visszalépéses kereséssel megoldható feladatokat célszerű lehet két fő csoportba osztani: az egyik esetben az egyes részfeladatok közötti kizárási pusztán a két részmegoldás vizsgálatával ellenőrizhetők, míg a másik esetben ehhez valamelyen komplex feltételt kell megadnunk. Ennek megfelelően két alapvető pszeudokódot írtunk fel, ezekre nézünk most külön-külön néhány gyakorlati példát.

A jobb áttekinthetőség kedvéért a két fő csoporton belül még további három osztályt határozhatunk meg, attól függően, hogy az egyes részfeladatok során milyen és hány darab részmegoldás közül választhatunk.

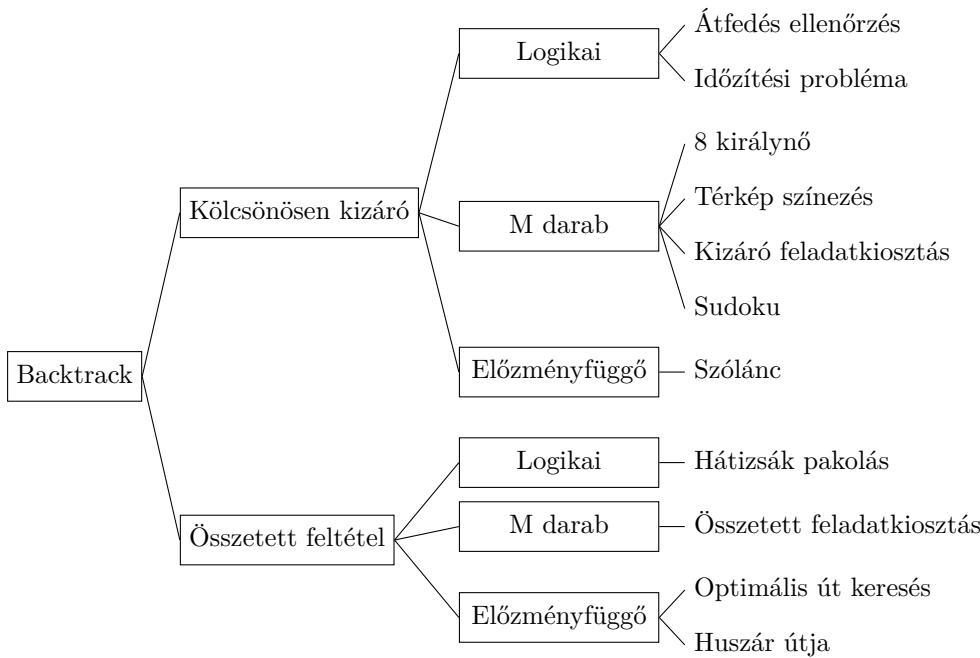
- Logikai érték: ezek a legegyszerűbb feladatok, ahol minden részfeladatnál csak egy *igaz/hamis* érték közül választhatunk. Például ha több időpont halmazából kell kiválasztanunk egy olyan részhalmazt, ahol egyik időpont sem fedi át a másikat, akkor ez felfogható egy olyan visszalépéses keresésnek, ahol a szintek száma egyenlő a időpontok számával, és minden szinten a megoldás két érték közül választható: igaz, ha az időpont bekerül a részhalmazba, és hamis, ha nem. Ehhez hasonlóan a hátizsákba pakolásnál is minden tárgyról el kell dönten, hogy berakjuk-e a zsákba vagy sem.
- M darab érték: ebben az esetben minden részfeladat esetén már tetszőleges számú (tehát 1, 2, vagy több) részmegoldás közül kell kiválasztani egyet. Itt azonban még feltételezzük, hogy az egyes részhalmazok (és persze ezzel együtt azok mérete is) már a keresés kezdetekor is ismertek. Bizonyos feladatoknál minden szinten ugyanazok lehetnek ezek a részmegoldások (pl. térkép/gráf színezés [4] esetén minden szinten ugyanazon színek közül választhatunk), bizonyos esetben viszont szintenként különbözőek lehetnek (pl. a feladatok kiosztásakor láthattuk, hogy minden feladatnál külön-külön meg volt határozva, hogy azt kik tudják elvégezni).
- Előzményfüggő értékek: általában ezek tekinthetők a legösszetettebb feladatoknak, itt minden részfeladat megoldásakor a lehetséges részmegoldások halmaza függ az előző részfeladatok megoldásaitól. Pl. a huszár útja a sakktáblán feladat (10.1.3. alfejezet) esetén a  $k$ . részfeladat azt jelenti, hogy a  $k$ . ugrásnál milyen irányba lépjünk tovább. Mivel a tábla szélén nem tudunk minden irányba továbbhaladni (és ez nem keverendő össze azzal a korláttal, hogy egy megadott helyen már jártunk), így az egyes lépések során a választható irányok száma változik, az a huszár aktuális pozíciójától függ. Az pedig, hogy a  $k$ . lépésnél pontosan hol vagyunk, az előző ( $k - 1$ ) lépés alapján határozható meg.

Számtalan példát találhatunk a visszalépéses kereséssel megoldható feladatokra (kezdve a klasszikus példákkal: 8 királynő, hátizsák pakolási probléma, stb.). Mivel ezek szerkezetileg hasonlóak, célszerű őket a fenti osztályokba rendezni. Néhány példa látható erre a 10.2. ábrán.

#### Megjegyzés

Érdekes lehet még bevezetni egy harmadik osztályozást is, amely azt határozná meg, hogy az egyes részfeladatok száma ( $n$ ) adott, vagy pedig a többi bemenő adattól függően változó. Például érdemes megvizsgálni a huszár útja a sakktáblán feladatot, illetve egy tetszőleges labirintusban útkereső feladatot. A kettő egymáshoz nagyon hasonlít (minden részfeladat egy irányt ad meg, hogy merre lépjünk tovább). Lényeges különbség azonban, hogy a huszár feladat esetén már a keresés indításakor is pontosan tudjuk, hogy összesen 63 darab lépést kell megtenni, míg egy útkeresés esetében ez előre nem ismert (esetleg egy maximális lépésszámot tudunk meghatározni a labirintus ismeretében, de lehet, hogy annál jóval kevesebb lépésből is meg tudjuk oldani a feladatot). A gyakorlatban ez a 10.1. algoritmus esetében csak kisebb módosítást igényel, mivel csak a megállási feltételt kell ennek megfelelően átalakítani (8. sor).

A példákkal is szeretnénk hangsúlyozni a visszalépéses keresés alapelvénél általános voltát. A példák áttekintése során mindenkor érdemes észrevenni a tényt, hogy magát a megoldó algoritmust nem változtatjuk, pusztán a bemenetet, illetve az  $f_t$  és  $f_k$  függvényeket.



10.2. ábra. Visszalépéses kereséssel (backtrack) megoldható problémák osztályozása.

### Logikai, egymást kizáró részmegoldás - Időzítési probléma

Ennek a feladat típusnak a jellegzetessége, hogy adott számos részfeladatunk, és mindegyik esetben egy igen/nem döntést kell hozni. Egy tipikus példa: egy tanásteremre érkezett megadott számú foglalási igény (kezdő és befejező időpontpárok halmaza), ezek közül válogassunk ki egy olyan részhalmazt, amelyek között nincs átfedő időintervallum, a foglalási idők összege pedig maximális. A feladat a visszalépéses kereséssel egyszerűen megoldható, amennyiben az alábbi értékeket választjuk:

- $n$  - egyenlő a foglalási igények számával. Tehát minden szint megfelel egy kérésnek.
- $M_{szint}$  - értéke minden esetben 2 (ez természetesen így már be is helyettesíthető az algoritmusba).
- $R_{szint,i}$  - értéke minden szinten  $R_{szint,1} = \text{igaz}$ ,  $R_{szint,2} = \text{hamis}$ . Tehát minden szinten azt kell eldöntenünk, hogy a hozzá tartozó időpont foglalást teljesítjük-e vagy sem.
- $f_t(szint, r)$  megvalósítása: azonosan igaz függvény, mivel az időpontok mindegyike választható és elhagyható is.
- $f_k(szint_a, r_a, szint_b, r_b)$  megvalósítása: a paraméterként átadott két részeredmény (jelen esetben idő intervallum) nem fedheti át egymást (az, hogy melyik szinten vannak, nem lényeges).

A függvények megvalósítását mutatja a 10.3. algoritmus. Mivel optimumot kell keresni, emiatt az esetlegesen megtalált eredményeket az ott leírt módon célszerű kezelní.

Hasonló alapelvek szerint nem csak egydimenziós esetben tudjuk az átfedéseket ellenőrizni, hanem az  $f_k$  függvény megfelelő megvalósításával akár síkbeli vagy térbeli elemekre is működik ez a megoldás (pl. válogassuk ki azokat a síkidomokat egy halmazból, amelyek között nincs átfedés, összterületük pedig maximális).

#### Megjegyzés

Az időzítési problémára létezik mohó algoritmus is, ami az itt megadottnál jóval hatékonyabban működik. A visszalépéses keresést azonban célszerű lehet megjegyezni, mivel a feltételrendszer változását a mohó algoritmus már nem biztos, hogy tudja követni, addig itt tetszőleges kizártó okokat tudunk kezelni (pl. több tárgyalónk van, prioritás az alkalmazottak között, stb.).

---

### 10.3. Algoritmus Időzítési probléma megoldása

---

Bemenet:  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált szint)

Bemenet:  $r$  - T (az adott szinten vizsgált részmegoldás)

Bemenet:  $E$  - tömb<T> (az előzőleg választott részmegoldások)

Kimenet: . - logikai (a részmegoldás elfogadható-e a megadott szinten)

1: függvény  $f_t(szint, r)$

2: vissza igaz

3: függvény vége

Bemenet:  $szint_A$  - egész<sup>+</sup> (az egyik ellenőrizendő szint)

Bemenet:  $r_A$  - T (az egyik szinten választott részmegoldás)

Bemenet:  $szint_B$  - egész<sup>+</sup> (a másik ellenőrizendő szint)

Bemenet:  $r_B$  - T (a másik szinten választott részmegoldás)

Kimenet: . - logikai (a két részmegoldás a megadott szinteken nem zárja ki egymást)

4: függvény  $f_k(szint_A, r_A, szint_B, r_B)$

5: vissza ( $r_A.kezdete > r_B.vége \vee r_A.vége < r_B.kezdete$ )

6: függvény vége

---

### M darab, egymást kizáró részmegoldás - Sudoku feladat

Ebben az esetben rendelkezésünkre áll egy  $9 \times 9$  méretű tábla (9 darab  $3 \times 3$  méretű blokkra osztva), ahol néhány szám már előre be van írva, a feladatunk pedig az, hogy az üresen maradt helyekre helyezzük számokat 1..9 között a Sudoku játék szabályainak megfelelően (egy sorba, oszlopba, blokkba nem kerülhetnek azonos számok). Az üres mezőkbe a megfelelő szám beírását tekintjük részfeladatoknak, tehát a részfeladatok száma megegyezik az üresen hagyott mezők számával. minden mezőbe 9 lehetséges részmegoldásunk van, tehát a feladat valóban ebbe az osztályba tartozik.

Elsőre nehéz lehet felismerni a visszalépéses keresés használatának lehetőségét, mivel a Sudoku tábla egy kétdimenziós adatszerkezet, az eddigi algoritmusaink azonban minden csakis egydimenziós feladatlistával működtek. Természetesen lehetne módosítani az algoritmust, de ennél egyszerűbb megoldás az, ha a feladatot próbáljuk meg átalakítani egydimenziós formára. Hozzunk létre egy listát, és ebbe helyezzük el a táblázat üres celláit: legyen ez az  $U$ , ahol  $U[i]$  visszaadja az  $i$ . üres cella koordinátait (sor, oszlop). Ezt követően ennek a sorozatnak az elemei képviselik a megoldandó részfeladatokat, és már a megszokott szinteket tudjuk használni.

Ebben az esetben a backtrack megvalósítás az alábbi:

- $n$  - egyenlő a kitöltendő cellák számával.
- $M_{szint}$  - értéke minden esetben 9 ( minden helyre ennyiféle részmegoldás lehetséges).
- $R_{szint,i}$  - értéke minden esetben az  $i$ , ahol  $1 \leq i \leq 9$ .
- $f_t(szint, r)$  megvalósítása: akkor ad vissza igazat, hogy ha az előre beírt számok közül egyik sem akadályozza meg a megadott szinten rögzítését. Tehát vele egy sorban, oszlopban, vagy  $3 \times 3$ -as blokkban nincs ugyanilyen szám.
- $f_k(szint_a, r_a, szint_b, r_b)$  megvalósítása: azt ellenőrzi, hogy a két szinten megadott két szám nem zárja-e ki egymást. Tehát csak akkor ad vissza igaz értéket, ha a két szintnek megfelelő mező azonos sorban, azonos oszlopban, vagy azonos  $3 \times 3$ -as blokkban van, és a paraméterként átadott két részeredmény (tehát a beírandó számok) azonosak.

A két függvény tényleges implementációját mutatja be a 10.4. algoritmus.

#### 10.4. Algoritmus Sudoku feladvány megoldása

Bemenet:  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált szint)

Bemenet:  $r$  - T (az adott szinten vizsgált részmegoldás)

Bemenet:  $E$  - tömb<T> (az előzőleg választott részmegoldások)

Kimenet: . - logikai (a részmegoldás elfogadható-e a megadott szinten)

1: függvény  $f_t(szint, r)$

2: vissza  $\nexists i \in \mathbb{N}, j \in \mathbb{N} | (1 \leq i \leq 9) \wedge (1 \leq j \leq 9) \wedge (S = P[i, j]) \wedge (i = U[szint].sor \vee j = U[szint].oszlop \vee (i \bmod 3 = U[szint].sor \bmod 3 \wedge j \bmod 3 = U[szint].oszlop \bmod 3))$

3: függvény vége

Bemenet:  $szint_A$  - egész<sup>+</sup> (az egyik ellenőrizendő szint)

Bemenet:  $r_A$  - T (az egyik szinten választott részmegoldás)

Bemenet:  $szint_B$  - egész<sup>+</sup> (a másik ellenőrizendő szint)

Bemenet:  $r_B$  - T (a másik szinten választott részmegoldás)

Kimenet: . - logikai (a két részmegoldás a megadott szinteken nem zárja ki egymást)

4: függvény  $f_k(szint_A, r_A, szint_B, r_B)$

5: vissza  $(r_A \neq r_B) \vee \neg(U[szint_A].sor = U[szint_B].sor \vee U[szint_A].oszlop = U[szint_B].oszlop \vee (U[szint_A].sor \bmod 3 = U[szint_B].sor \bmod 3 \wedge U[szint_A].oszlop \bmod 3 = U[szint_B].oszlop \bmod 3))$

6: függvény vége

#### Felhasznált változók és függvények

- $P[i,j]$  : Megadja az  $i$ . sor,  $j$ . oszlopba előre beírt számot (ha van ilyen).
- $U[i]$  : Visszaadja az  $i$ . üres cella koordinátáit.

#### Megjegyzés

Ebbe a kategóriába tartozik a klasszikus „8 királynő elhelyezése a sakktáblán” feladat is [14], ahol úgy kell elhelyezni 8 királynőt egy szabványos sakktáblán, hogy azok ne üssék egymást. Mivel nyilvánvalóan minden királynőt más-más oszlopba kell helyezni, így már könnyen át tudjuk alakítani a feladatot a visszalépéses keresés által is használható formára. 8 részfeladatunk van, minden részfeladatban azt kell előntení, hogy adott oszlopon belül melyik sorba helyezzük a királynőt. Az  $f_t$  azonosan igaz függvény, az  $f_k$  pedig azt figyeli, hogy az újonnan vizsgált királynőt ne üsse egyik, az előző oszlopokba lerakott társa se.

#### Előzményfüggő, egymást kizáró részmegoldás - Szólánc játék

Tegyük fel, hogy van egy szavakból álló készletünk, és ezeket úgy kell sorba rendeznünk, hogy (egy tetszőleges első szót követően) minden szó első betűje megegyezzen az előző szó utolsó betűjével. Több azonos kezdetű és végű szó esetén a lehetőségek száma meglehetősen nagy lehet, emiatt a visszalépéses keresés itt is jól használható.

Az egyes szavak választása nem független az előzőektől, emiatt itt feltételezzük, hogy az  $f_t$  hozzáfér az előző eredményekhez is (bár valójában csak az utolsóra van szüksége), ez az alap algoritmus kisebb módosítását igényli.

Ebben az esetben a backtrack megvalósítás az alábbi:

- $n$  - a rendelkezésre álló szavak száma, mivel minden fel akarjuk használni.
- $M_{szint}$  - értéke minden esetben egyenlő a szavak számával, mivel bármelyik szó szerepelhet bárminelyik helyen.
- $R_{szint,i}$  - értéke minden esetben az i. szó.
- $f_t(szint, r)$  megvalósítása: azt vizsgálja, hogy minden szinten csak olyan szavakat lehessen választani, amelynek első karaktere megegyezik az előző részmegoldás utolsó karakterével (kivéve persze

az első szót, ahol bármi szerepelhet). Látható, hogy paraméterként megkapja a  $E$  vektort, emiatt módosítani kell a kereső algoritmust, hogy ezt is adja át.

- $f_k(\text{szint}_a, r_a, \text{szint}_b, r_b)$  megvalósítása: ebben az esetben ez csak egy egyszerű kizárást jelent, hogy ne lehessen ugyanazt a szót kétszer kiválasztani. Tegyük fel, hogy egy szó csak egyszer szerepel a bemenetben.

---

## 10.5. Algoritmus Szólánc játék

---

**Bemenet:**  $\text{szint}$  - egész<sup>+</sup> (az aktuálisan vizsgált szint)

**Bemenet:**  $r$  -  $\mathbf{T}$  (az adott szinten vizsgált részmegoldás)

**Bemenet:**  $E$  - tömb< $\mathbf{T}$ > (az előzőleg választott részmegoldások)

**Kimenet:** . - logikai (a részmegoldás elfogadható-e a megadott szinten)

1: függvény  $f_t(\text{szint}, r)$

2: vissza ( $\text{szint} = 1 \vee E_{\text{szint}-1}.\text{vége} = S.\text{eleje}$ )

3: függvény vége

**Bemenet:**  $\text{szint}_A$  - egész<sup>+</sup> (az egyik ellenőrizendő szint)

**Bemenet:**  $r_A$  -  $\mathbf{T}$  (az egyik szinten választott részmegoldás)

**Bemenet:**  $\text{szint}_B$  - egész<sup>+</sup> (a másik ellenőrizendő szint)

**Bemenet:**  $r_B$  -  $\mathbf{T}$  (a másik szinten választott részmegoldás)

**Kimenet:** . - logikai (a két részmegoldás a megadott szinteken nem zárja ki egymást)

4: függvény  $f_k(\text{szint}_A, r_A, \text{szint}_B, r_B)$

5: vissza ( $r_A \neq r_B$ )

6: függvény vége

---

### Felhasznált változók és függvények

- $x.\text{eleje}$  : Az  $x$  szó első betűje.
  - $x.\text{vége}$  : Az  $x$  szó utolsó betűje.
- 

## Logikai, nem kizáró részmegoldás - Hátzsák-pakolás

A hátzsák-pakolási feladatnál van  $n$  darab tárgyunk, illetve ismerjük azok súlyát is ( $r_i$  legyen az  $i$ . tárgy súlya). Úgy kell kiválogatnunk közülük tetszőleges számút, hogy az összsúly ne haladja meg a *SúlyKorlát* konstans értékét. Az egyszerűség kedvéért most nem optimumot keresünk, mint a „0-1 hátzsák problémánál”, hanem megelégszünk egy tetszőleges megoldással.

A feladat hasonló az átfedések ellenőrzéséhez, itt is célszerű úgy megválasztani a részfeladatokat, hogy minden esetben azt kelljen csak eldöntení, hogy egy megadott tárgyat berakunk-e a hátzsákba, vagy pedig sem. A lényeges különbség az, hogy míg az időpont átfedésekénél az egyes részfeladatok páronként kizárták egymást, addig itt az eddig rögzített részeredmények összességeből dönthető csak el, hogy azok kizárták-e az új megoldást.

A feladat megoldása a visszalépéses keresésre egyszerűen visszavezethető (10.6. algoritmus), amennyiben az alábbi bemenetet választjuk:

- $n$  - egyenlő a tárgyak számával. minden részfeladat azt jelenti, hogy az adott tárgyat elhelyezzük-e a hátzsákba vagy sem.
- $M_{\text{szint}}$  - értéke minden esetben 2 (ez természetesen így már be is helyettesíthető az algoritmusba).
- $R_{\text{szint}, i}$  - értéke minden szinten  $R_{\text{szint}, 1} = \text{igaz}$ ,  $R_{\text{szint}, 2} = \text{hamis}$ . Tehát minden szinten azt kell eldöntenünk, hogy az adott tárgyat berakjuk-e a hátzsákba, vagy sem.
- $f_t(\text{szint}, r)$  megvalósítása: azonosan *igaz* függvény, hiszen minden tárgy be is rakható, illetve ki is hagyható.
- $f_k(\text{szint}, r, E)$  megvalósítása: az eljárás paraméterként megkapja az előzőleg rögzített részmegoldásokat (tehát mely tárgyakat raktuk be eddig a zsákba, és melyeket nem). Ez alapján ki tudja számolni az eddigi összes súlyt, és így már el tudja döntení, hogy a  $\text{szint}$ -edik tárgy belefér-e a hátzsákba.

---

## 10.6. Algoritmus 0-1 hátizsák probléma megoldása visszalépéses kereséssel

---

Bemenet:  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált szint)

Bemenet:  $r$  - T (az adott szinten vizsgált részmegoldás)

Bemenet:  $E$  - tömb<T> (az előzőleg választott részmegoldások)

Kimenet: . - logikai (a részmegoldás elfogadható-e a megadott szinten)

1: függvény  $f_t(szint, r)$

2: vissza igaz

3: függvény vége

Bemenet:  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált szint)

Bemenet:  $r$  - T (az adott szinten vizsgált részmegoldás)

Bemenet:  $E$  - tömb<T> (az előzőleg választott részmegoldások)

Kimenet: . - logikai (a részmegoldást nem zártja ki egy előző választás sem)

4: függvény  $f_k(szint, r, E)$

5: ha  $r = hamis$  akkor vissza igaz

6:  $s \leftarrow R_{szint}$

7:  $i \leftarrow 1$

8: ciklus amíg  $i < szint \wedge s \leq W_{max}$

9:     ha  $E_i = igaz$  akkor

10:          $s \leftarrow s + R_i$

11:         elágazás vége

12:          $i \leftarrow i + 1$

13:     ciklus vége

14:     vissza ( $s \leq W_{max}$ )

15: függvény vége

---

### Felhasznált változók és függvények

- $i$  : Egy egész<sup>+</sup> típusú segédváltozó.
  - $R_i$  : Megadja az  $i$ . térgy súlyát.
  - $W_{max}$  : A hátizsák mérete (maximális összsúly).
- 

### M darab, nem kizáró részmegoldás - Feladatok kiosztása

Ebben az esetben adott számos részfeladat, és mindegyikre rendelkezünk különféle potenciális megoldásokkal, az egyes megoldások azonban páronként nem zárják ki egymást. A feladat ezzel nagyon hasonlít a fejezet elején bemutatott példához, azonban egy lényeges különbség, hogy nem zárjuk ki azt, hogy két feladatot ugyanaz a személy végezzen el. Tehát általánosabban fogalmazzuk meg a feladatot az alábbiak szerint: adott  $n$  darab feladat, és  $k$  darab személy. A  $J$  mátrix tartalmazza azt, hogy kik tudják megoldani az egyes feladatokat, pl.  $J[i, j]$  értéke igaz, ha az  $i$ . feladatot a  $j$ . személy meg tudja oldani, egyébként pedig hamis. Adjunk egy olyan feladatkiosztást, hogy minden feladathoz rendelünk valakit, de egy személy se vállalhat egyszerre  $z$  darabnál több részfeladatot.

Ebben az esetben a visszalépéses keresés megvalósítása az alábbi (10.7. algoritmus):

- $n$  - egyenlő a feladatok számával.
- $M_{szint}$  - azt mutatja, hogy hányan jelentkeztek a  $szint$ -edik munkára.
- $R_{szint,i}$  - értéke a  $szint$ -edik munkára  $i$ -edikként jelentkezett munkás.
- $f_t(szint, r)$  megvalósítása: a  $J$  mátrix ellenőrzését jelenti, hiszen minden feladatnál meg kell néznünk, hogy az adott személy alkalmas-e rá.
- $f_k(szint, r, E)$  megvalósítása: a függvény megszámolja, hogy a megadott személyt hány munkára jelöltük már. Amennyiben az aktuálissal együtt meghaladná a korlátot ( $z$ ), akkor *hamis* a visszatérési értéke, egyébként pedig *igaz*.

---

## 10.7. Algoritmus Feladat kiosztási probléma megoldása

---

Bemenet:  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált szint)

Bemenet:  $r$  - T (az adott szinten vizsgált részmegoldás)

Bemenet:  $E$  - tömb<T> (az előzőleg választott részmegoldások)

Kimenet: . - logikai (a részmegoldás elfogadható-e a megadott szinten)

1: függvény  $f_t(szint, r)$

2: vissza  $J[szint, r]$

3: függvény vége

Bemenet:  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált szint)

Bemenet:  $r$  - T (az adott szinten vizsgált részmegoldás)

Bemenet:  $E$  - tömb<T> (az előzőleg választott részmegoldások)

Kimenet: . - logikai (a részmegoldást nem zártja ki egy előző választás sem)

4: függvény  $f_k(szint, r, E)$

5:  $db \leftarrow 0$

6:  $i \leftarrow 1$

7: ciklus amíg  $i < szint \wedge db < z$

8:     ha  $E_i = r$  akkor

9:          $db \leftarrow db + 1$

10:     elágazás vége

11:      $i \leftarrow i + 1$

12: ciklus vége

13: vissza ( $db < z$ )

14: függvény vége

---

### Felhasznált változók és függvények

- $db$  : Egy egész<sup>+</sup> típusú segédváltozó.
  - $J_{i,j}$  : A  $j$ . személy jelentkezett-e az  $i$ . feladatra.
  - $z$  : Egy személy egyszerre hány feladatot végezhet el.
-

## Előzményfüggő, nem kizáró részmegoldás - Huszár útja a sakktáblán

A jól ismert feladatunk az, hogy egy szabványos sakktábla egyik sarkából elindulva egy huszárral a szabályos lépésekkel érinteni kell a tábla minden egyes mezőjét úgy, hogy minden mezőre pontosan egyszer lépjünk.

Ez egyben ismét egy jó példa arra is, hogy a kétdimenziós problémát hogyan tudjuk egydimenzióssá alakítani. Ugyanis ha zavar minket a sakktábla kétdimenziós volta, akkor a feladatot úgy is átfogalmazhatjuk, hogy 63-at kell lépnünk, ennek megfelelően van 63 részfeladatunk, ahol minden esetben a lehetséges lépések közül kell egyet kiválasztanunk.

A lehetséges részmegoldások száma azonban itt már előzményfüggő, mivel a huszár alapvetően 8 irányba tud lépni, de a tábla szélein nem biztos, hogy az összes irány válaszható. Azt pedig, hogy aktuálisan hol áll a huszár, csak az előző lépések ismeretében tudjuk meghatározni.

- $n$  - 63, hiszen ennyi lépést kell tennünk.
- $M_{szint}$  - értéke minden esetben 8, mivel 8 irányba lehet lépni.
- $R_{szint,i}$  - a  $szint$  értékétől függetlenül, visszaadja az i. huszár lépés irány relatív sor és oszlop koordinátáit.
- $f_t(szint, r)$  megvalósítása: azt vizsgálja, hogy megadott szinten a megadott irányú lépés a táblán belülre vezet, vagy sem. Mivel ez attól függ, hogy előzőleg milyen lépéseket hajtottunk végre, emiatt ehhez végignézzük az előző eredményeket, mintegy lejátszva az előző lépéseket.
- $f_k(szint, r, E)$  megvalósítása: az előző alapján már érthető ennek a működése is. A kérdés az, hogy az új lépés olyan helyre vezet-e, ahol még nem jártunk. Ehhez végigkövetjük az előző lépéseket, és ha azok egyike se vezetett a vizsgálandó pozícióba, akkor a visszatérési értéke *igaz*, egyébként pedig *hamis* lesz.

A függvények megvalósítását mutatja a 10.8. algoritmus.

---

## 10.8. Algoritmus Huszár útja a sakktáblán

---

**Bemenet:**  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált szint)  
**Bemenet:**  $r$  - T (az adott szinten vizsgált részmegoldás)  
**Bemenet:**  $E$  - tömb<T> (az előzőleg választott részmegoldások)  
**Kimenet:** . - logikai (a részmegoldás elfogadható-e a megadott szinten)

```
1: függvény $f_t(szint, r)$
2: sor $\leftarrow 1$; oszlop $\leftarrow 1$
3: ciklus $k \leftarrow 1$ -től ($szint - 1$)-ig
4: sor $\leftarrow sor + R_{szint,E_k}.sor$
5: oszlop $\leftarrow oszlop + R_{szint,E_k}.oszlop$
6: ciklus vége
7: sor $\leftarrow sor + R_{szint,r}.sor$
8: oszlop $\leftarrow oszlop + R_{szint,r}.oszlop$
9: vissza ($1 \leq sor \leq 8 \wedge 1 \leq oszlop \leq 8$)
10: függvény vége
```

**Bemenet:**  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált szint)  
**Bemenet:**  $r$  - T (az adott szinten vizsgált részmegoldás)  
**Bemenet:**  $E$  - tömb<T> (az előzőleg választott részmegoldások)  
**Kimenet:** . - logikai (a részmegoldást nem zártja ki egy előző választás sem)

```
11: függvény $f_k(szint, r, E)$
12: posA.sor $\leftarrow 1$; posA.oszlop $\leftarrow 1$; posB.sor $\leftarrow 1$; posB.oszlop $\leftarrow 1$
13: ciklus $k \leftarrow 1$ -től ($szint_A - 1$)-ig
14: posA.sor $\leftarrow posA.sor + R_{szint,E_k}.sor$
15: posA.oszlop $\leftarrow posA.oszlop + R_{szint,E_k}.oszlop$
16: ha $k \leq szint_B$ akkor
17: posB.sor $\leftarrow posB.sor + R_{szint,E_k}.sor$
18: posB.oszlop $\leftarrow posB.oszlop + R_{szint,E_k}.oszlop$
19: elágazás vége
20: ciklus vége
21: posA.sor $\leftarrow posA.sor + R_{szint,r}.sor$
22: posA.oszlop $\leftarrow posA.oszlop + R_{szint,r}.oszlop$
23: vissza ($posA.sor \neq posB.sor \vee posA.oszlop \neq posB.oszlop$)
24: függvény vége
```

---

### Felhasznált változók és függvények

- **sor** : Az egyes pozíciókat tartalmazó értékek sor komponense.
  - **oszlop** : Az egyes pozíciókat tartalmazó értékek oszlop komponense.
- 

#### Megjegyzés

A bemutatott mintapéldák jelentős része megoldható egyszerűbben, illetve gyakran jóval hatékonyabban is (pl. a huszár útja feladatnál egy köztes tömb segítségével eltárolhatjuk azokat a pozíciókat, ahol már jártunk, így nem kell minden vizsgálatnál újrajátszani az előzményeket). A példák célja azonban az egyes feladatok leghatékonyabb megoldása helyett jelen esetben az volt, hogy bemutassuk, hogy az eredeti visszalépéses keresés módosítása nélkül, pusztán az  $f_t$  és  $f_k$  függvények helyes megválasztásával is meg lehet oldani egészen különböző feladatokat.

## 10.2. Optimalizálás visszalépéses kereséssel

### 10.2.1. A módszer bemutatása

#### Összes megoldás keresése

Az előzőleg megismert algoritmus minden megoldást adja vissza. A gyakorlatban sokszor előfordul, hogy nem csak egy tetszőleges, hanem az összes megoldást keressük (ezt mutatja be a 10.9. algoritmus).

---

#### 10.9. Algoritmus Minden megoldást visszaadó visszalépéses keresés

---

**Bemenet:**  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált részprobléma száma)

**Bemenet:**  $E$  - tömb< $T$ > (az eddig választott részmegoldásokat tárolja)

**Bemenet:**  $van$  - logikai (eddig talált-e már teljes megoldást)

**Kimenet:**  $E$  - tömb< $T$ > (a részmegoldások kiegészítve az aktuális szint esetleges megoldásával)

**Kimenet:**  $van$  - logikai (talált-e már megoldást teljes, kiegészítve az aktuális szint vizsgálatával)

**Kimenet:**  $M$  - Halmaz< $tömb$ < $T$ >> (a megtalált eredmények halmaza)

```
1: eljárás VISSZALÉPÉSESKERESÉS($szint$, címszerint E , címszerint van , címszerint M)
2: $i \leftarrow 0$
3: ciklus amíg $i < M_{szint}$
4: $i \leftarrow i + 1$
5: ha $f_t(szint, R_{szint,i})$ akkor
6: ha $f_k(szint, R_{szint,i}, E)$ akkor
7: $E_{szint} \leftarrow R_{szint,i}$
8: ha $szint = n$ akkor
9: $van \leftarrow igaz$
10: $M \leftarrow M \cup \{E\}$
11: különben
12: VISSZALÉPÉSESKERESÉS($szint+1$, E , van , M)
13: elágazás vége
14: elágazás vége
15: elágazás vége
16: ciklus vége
17: eljárás vége
```

---

#### Felhasznált változók és függvények

- $i$  : Egy egész<sup>+</sup> típusú segédváltozó.
  - $n$  : A megoldandó részfeladatok száma.
  - $M_{szint}$  : A lehetséges részmegoldások száma a megadott szinten.
  - $R_{szint,i}$  : Az  $i$  részmegoldás a megadott szinten.
  - $f_t$ ,  $f_k$  : A keresési feltételeket meghatározó függvények.
- 

Az algoritmust csak minimálisan kellett megváltoztatni ahhoz, hogy az megtalálja az összes megoldást. Megjelent egy új paraméter, az  $M$ , ami egy halmaz, amiben folyamatosan gyűjtjük a menet közben megtalált megoldásokat (a rekurzió kezdő hívásakor egy üres halmazt kell átadni címszerinti paraméterként). Amennyiben a  $van$  változó értéke a futás végén *igaz*, akkor ebből a halmazból lehet kiolvasni a talált teljes megoldásokat.

Az alapesetet bemutató pszeudokódot két helyen kellett módosítani: a 10. sorban látható kiegészítés célja, hogy egy új teljes eredmény megtalálása esetében mentse el ezt a paraméterként átadott  $M$  halmazba. Ezen túlmenően módosítanunk kellett a ciklust is, mivel az alapváltozatban az úgy működött, hogy az első megoldás után megpróbált minél gyorsabban kilépni a rekurzióból. Ha az összes megoldást keressük, akkor azonban erre nincs szükség. Sőt, egy teljes megoldás megtalálása nem befolyásolja a további vizsgálatokat, a keresésnek ugyanúgy kell folytatódnia, mint eddig. Ennek megfelelően változott a 3. sor.

## Optimális megoldás keresése

Szintén gyakori módosítás, hogy nem az összes, hanem csak a legjobb megoldást keressük. A módosítás hasonló az előzőhez, itt is el kell érnünk a ciklusfeltétel módosításával, hogy az első találat után ne álljon le a keresés, hanem lépjön tovább a következő lehetséges részmegoldáshoz. Ezt mutatja be a 10.10. algoritmus.

---

### 10.10. Algoritmus Optimális megoldást visszaadó visszalépéses keresés

---

Bemenet:  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált részprobléma száma)  
Bemenet:  $E$  - tömb<T> (az eddig választott részmegoldásokat tárolja)  
Bemenet:  $van$  - logikai (eddig talált-e már teljes megoldást)  
Bemenet:  $O$  - tömb<T> (az eddig talált legjobb optimális eredmény)  
Kimenet:  $E$  - tömb<T> (a részmegoldások kiegészítve az aktuális szint esetleges megoldásával)  
Kimenet:  $van$  - logikai (talált-e már teljes megoldást, kiegészítve az aktuális szint vizsgálatával)  
Kimenet:  $O$  - tömb<T> (az optimális eredmény a szint vizsgálata után)

1: eljárás VISSZALÉPÉSESKERESÉS( $szint$ , címszerint  $E$ , címszerint  $van$ , címszerint  $O$ )  
2:      $i \leftarrow 0$   
3:     ciklus amíg  $i < M_{szint}$   
4:          $i \leftarrow i + 1$   
5:         ha  $f_t(szint, R_{szint,i})$  akkor  
6:             ha  $f_k(szint, R_{szint,i}, E)$  akkor  
7:                  $E_{szint} \leftarrow R_{szint,i}$   
8:                 ha  $szint = n$  akkor  
9:                     ha  $\neg van \vee JÓSÁG(E) > JÓSÁG(O)$  akkor  
10:                          $O \leftarrow E$   
11:                     elágazás vége  
12:                  $van \leftarrow igaz$   
13:             különben  
14:                 VISSZALÉPÉSESKERESÉS( $szint+1$ ,  $E$ ,  $van$ ,  $O$ )  
15:                 elágazás vége  
16:                 elágazás vége  
17:                 elágazás vége  
18:     ciklus vége  
19: eljárás vége

---

#### Felhasznált változók és függvények

- $i$  : Egy egész<sup>+</sup> típusú segédváltozó.
  - $n$  : A megoldandó részfeladatok száma.
  - $M_{szint}$  : A lehetséges részmegoldások száma a megadott szinten.
  - $R_{szint,i}$  : Az  $i$  részmegoldás a megadott szinten.
  - $f_t$ ,  $f_k$  : A keresési feltételeket meghatározó függvények.
  - $JÓSÁG(E)$  : Egy függvény, ami megadja a paraméterként átadott  $E$  végeredmény értékét.
- 

A lényeges különbség pusztán annyi, hogy az új, teljes megoldások megtalálása esetén nem tároljuk el azokat egy halmazban, hanem minden új megoldás esetében ellenőrizzük, hogy az jobb-e mint az eddig talált legjobb. Ha igen, akkor mostantól ezt tekintjük a legjobbnak.

A változás tehát a 9. során látható. Amikor találunk egy új megoldást, akkor ellenőrizzük, hogy volt-e már megoldásunk (ezt egyszerűen megtehetjük, hiszen ezt mutatja a  $van$  változó értéke). Ha nem, akkor nyilvánvalóan ez az első lesz az eddigi legjobb. Ha pedig már volt, akkor ahhoz hasonlítjuk a most találtat, és csak akkor írjuk ezzel felül a régit, ha annál jobbat találtunk (10. sor). Ez utóbbi művelet azonban csak akkor hajtható végre, ha az egyes végeredmények összehasonlíthatóak. Emiatt bevezettünk egy kiegészítő  $JÓSÁG$  nevű függvényt, aminek a paramétere egy végeredmény, visszatérési értéke pedig egy szám, ami két eredmény közül mindenkorán lesz nagyobb, amelyiket jobbnak tekintjük.

### 10.2.2. A 0-1 hátizsák probléma megoldása

A fentieknek megfelelően a hátizsák problémát is egyszerűen megoldhatjuk a visszalépéses kereséssel. Részfeladatnak itt azt tekinthetjük, hogy az egyes tárgyak bekerülnek-e a zsákba vagy sem.  $n$  darab tárgy esetén tehát lesz  $n$  darab részfeladat, mindenek lehetséges részmegoldásai az *igaz* vagy *hamis*. A 10.11. algoritmus bemutatja a feladat megoldását visszalépéses kereséssel. A megoldó algoritmus nemileg eltér a 10. fejezetben bemutatott általános formától, de itt a cél nem a módszer általánosságának bemutatása, hanem az előző problémamegoldó technikákkal való összehasonlítás.

---

#### 10.11. Algoritmus A 0-1 hátizsákprobléma megoldása „visszalépéses keresés” segítségével

---

**Bemenet:** *szint* - egész<sup>+</sup> (hányadik szintet vizsgáljuk)

**Bemenet:** *E* - tömb<logikai> (az eddigi részmegoldások)

**Kimenet:** . - logikai (a részmegoldások nem zárják ki egymást)

- 1: függvény  $f_k(\text{szint}, E)$
- 2: vissza ÖsszSÚLY(*E*)  $\leq W_{max}$
- 3: függvény vége

**Bemenet:** *szint* - egész<sup>+</sup> (hányadik szintet vizsgáljuk)

**Bemenet:** *E* - tömb<logikai> (az aktuálisan vizsgált részmegoldás, aminek megváltozhat az értéke)

- 4: eljárás BACKTRACK(*szint*, címszerint *E*, címszerint *O*)
- 5: ciklus *i*  $\leftarrow 0$ -től 1-ig
- 6:  $E[\text{szint}] \leftarrow (i = 0)$  ▷ ha *i*=0 akkor legyen igaz, különben hamis
- 7: ha  $f_k(\text{szint}, E)$  akkor
- 8: ha *szint* = *n* akkor
- 9: ha ÖSSZÉRTÉK(*E*) > ÖSSZÉRTÉK(*O*) akkor
- 10: *O*  $\leftarrow E$
- 11: elágazás vége
- 12: különben
- 13: BACKTRACK(*szint*+1, *E*, *O*)
- 14: elágazás vége
- 15: elágazás vége
- 16: ciklus vége
- 17: eljárás vége ▷ visszalépéskor *E*[*szint*] értéke minden hamis

**Kimenet:** . - szám<sup>+</sup> (egy optimális pakolás összértéke)

- 18: függvény HÁTIZSÁKBT
  - 19:  $E \leftarrow [\text{hamis}, \text{hamis}, \dots, \text{hamis}]$
  - 20:  $O \leftarrow [\text{hamis}, \text{hamis}, \dots, \text{hamis}]$
  - 21: BACKTRACK(1, *E*, *O*)
  - 22: vissza ÖSSZÉRTÉK(*O*)
  - 23: függvény vége
- 

#### Felhasznált változók és függvények

- $n$  : A hátizsákba pakolható tárgyak száma.
  - $W_{max}$  : A hátizsák méréte (a belepakolható tárgyak maximális összsúlya).
  - $w_i$  : A hátizsákba pakolható *i*. tárgy súlya ( $1 \leq i \leq n$ ).
  - $p_i$  : A hátizsákba pakolható *i*. tárgy hasznossága ( $1 \leq i \leq n$ ).
  - *E*, *O* : Az aktuálisan vizsgált/eddi legjobb pakolás adatai egy *tömb<logikai>* adatszerkezetben tárolva.
- 

Egyeszerűsítésnek tekinthető, hogy nincs külön lehetséges részmegoldások mátrixunk, hanem minden szinten két lehetőséget vizsgál meg az algoritmus (5. sor). Amennyiben az *i* értéke 0, akkor úgy tekinti, hogy a *szint*-edik tárgyat berakjuk a hátizsákba, amennyiben az *i* értéke 1, akkor pedig nem. A következő sorban ennek megfelelően tölti ki az *E* vektor megfelelő elemét.

Az  $f_k$  feltétel ellenőrzése jelen esetben a hátizsákba eddig bepakolandó elemek összsúlyának vizsgálatát jelenti. Amennyiben ez kevesebb, mint a rendelkezésre álló teljes kapacitás ( $W_{max}$ ), akkor a

részmegoldást jónak tekinti. Amennyiben nem, akkor pedig nem. Ez utóbbi esetben próbálkozik az *i* ciklus következő lehetséges értékével, illetve ha már minden változatot megnézte, akkor engedi vissza lépni a rekurziót.

Amennyiben találtunk egy teljes megoldást (8. sor), akkor megvizsgáljuk, hogy ez jobb-e, mint az eddig talált legjobb (9. sor). Ha igen, akkor ezt követően ezt tekinti optimális megoldásnak.

Amennyiben még csak egy részmegoldást rögzítettünk, akkor a tanult visszalépések keresés elvét követve meghívja önmagát (13. sor), és egy szinttel feljebb próbál megoldást találni.

#### Megjegyzés

Szintén egyszerűsítés, hogy nincs *van* változó az algoritmusban. Bár az eredeti visszalépések keresésben a változónak több szerepe is volt, itt egyikre sincs szükség. Az algoritmust nem kell leállítanunk az első találat esetén, tehát emiatt nem kell. Azt se kell vizsgálnunk, hogy találtunk-e legalább egy elfogadható megoldást, hiszen tudjuk, hogy ilyen biztosan van (a teljesen üres hártsák). Ennek köszönhetően az új eredmény vizsgálatakor (hogy az jobb-e, mint az eddig megtalált) sem kell külön foglalkoznunk azzal a kérdéssel, hogy mi van akkor, ha még nem is találtunk egy jó megoldást sem.

#### Megjegyzés

Ezt a pszeudokódnál is kiemeltük, hogy lényeges körülmény, hogy a belső *i* ciklus először az *igaz* és utána a *hamis* értéket vizsgálja. Ez azért fontos, mert így az eljárásból való kilépésnél az *E[szint]* értékében mindig *hamis* fog maradni. Tehát egy előrelépkedés utáni visszalépkedés után az *E* tömb *szint*-edik utáni elemei mindig *hamis* értéket fognak tartalmazni. Ez azért fontos, mert különben az *ÖsszÉrték* függvény hívása hibás eredményt adna.

### 10.2.3. A megoldás értékelése

A módszer előnyei:

- Jól áttekinthető, általánosan használható módszer.
- Az „oszd meg és uralkodj” módszerhez hasonlóan a nyilvánvalóan felesleges részfeladatok vizsgálatát elkerüli.

A módszer hátrányai:

- A rekurzív hívások miatt erőforrásigénye nagy lehet.
- Bizonyos feladatokra találhatunk hatékonyabb megoldásokat is.

## 10.3. Szétválasztás és korlátozás technika

### 10.3.1. A módszer bemutatása

A már megismert módszerek közül a „nyers erő” a legprimitívebb, ez egyáltalán nem próbálja meg kihasználni a feladat jellegzetességeit a hatékonyság növelése érdekében. Amennyiben a feladat jellege ezt lehetővé teszi, akkor a „visszalépéses keresés” ehhez képest jóval előnyösebb lehet, mivel ez a keresés során észreveszi, ha az előző választásai eredményeképpen egy megadott úton zsákutcába jutott és onnan visszalépve próbál másik irányt keresni. A következő szintre való lépés során viszont ez se foglalkozik azzal, hogy a választott út vezethet-e teljes megoldáshoz, egyszerűen végigpróbál minden lehetőséget.

A „szétválasztás és korlátozás” annyival kifinomultabb, hogy ez a továbblépés előtt megvizsgálja, hogy a kiválasztott úton elvileg elérhető-e a teljes megoldás. Erre biztos választ persze általában nem lehet adni (ha lehetne, akkor mohó algoritmust használnánk), ezért működése azon alapul, hogy már az is segítséget jelenthet, ha egy megfelelően pontos felső becslést tud adni. Hiszen már ez alapján is gyakran kerülhetünk olyan helyzetbe, hogy egy megadott úton lehetne ugyan tovább folytatni a keresést, de az ottani legjobb eredmény felső becslése is azt mutatja, hogy nem érdemes, mert nem vezethet teljes megoldáshoz. Ezzel újabb számításigényes ágakat zárhatunk ki a keresésből.

Ahhoz, hogy a technikát jól használhassuk, az alábbi két függvényre van szükségünk:

- **Szétválasztási függvény:** ennek szerepe, hogy a meglévő (rész)feladatot további részfeladatokra bontsa (branch). Ez már ismerős lehet a többi megismert technikából („oszd meg és uralkodj”).
- **Korlátozó függvény:** ez egy korlátot ad egy részfeladat optimális megoldására. Ennek segítségével tudjuk eldöntheti, hogy érdemes-e továbblépni egy úton, vagy akár vissza is fordulhatunk, hiszen nem fogunk a már meglévőnél jobb megoldást találni. Ezzel tulajdonképpen levágjuk (bound) a keresési fának azokat az ágait, amit nem érdemes vizsgálni.

A fentiek alapján már könnyen elképzelhető a megoldó algoritmus. Az eredeti problémát a szétválasztási függvényekkel szétbontjuk kisebb részproblémára. A korlátozó függvény segítségével megvizsgáljuk, hogy az egyes részproblémák esetében mi az optimális megoldás értékének felső korlátja (persze nem biztos, hogy tényleg lesz is ilyen optimális eredményünk, csak egy felső becslést adunk). Amennyiben ez alapján érdemesnek tűnik, akkor rekurzív módon megvizsgáljuk az egyes részproblémákat (azokat is szétbontjuk, stb.).

A technikát úgy is elképzelhetjük, hogy ez egy kiegészítése a már megismert módszereknek. Mind az „oszd meg és uralkodj”, mind pedig a „visszalépéses keresés” módszer esetében a rekurzív hívás előtt elvégezhetjük ezt a becslést, és ez alapján dönthetjük el, hogy továbblépjünk-e a megadott irányba. Ezek mellett persze rugalmasan, bármilyen módszer kiegészíthető ezzel a gondolattal. A lentiekben mi a már részletesen tárgyalta „visszalépéses keresést” fogjuk továbbfejleszteni.

### 10.3.2. Visszalépéses keresés kiegészítése

A már meglévő algoritmust csak minimálisan kell kiegészítenünk, hogy alkalmas legyen a továbblépések korlátozására is. Ehhez a már meglévő  $f_t$  és  $f_k$  függvények mellé bevezetünk egy harmadik, szintén a feladat által meghatározott függvényt:  $f_b$ .

- $f_t(\text{szint}, r)$ : egy függvény, ami azt határozza meg, hogy a  $\text{szint}$ -edik részfeladat esetében lehetséges megoldás-e az  $r$ ?
- $f_k(\text{szint}, r, E)$ : azt határozza meg, hogy a  $\text{szint}$ -edik részfeladat esetében választhatjuk-e az  $r$  részmegoldást, amennyiben az előző szinteken az  $E$  vektorban található részmegoldásokat választottuk.
- $f_b(\text{szint}, E)$ : a  $\text{szint}$ -edik részfeladat megoldását követően, amennyiben az előző és az aktuális szinteken az  $E$  vektorban található részmegoldásokat választottuk, akkor egy „felső becslést” ad arra, hogy a maradék részfeladatok megoldásával megoldható-e a teljes feladat? Felső becslés alatt azt értjük, hogy ha a válasz *hamis*, akkor biztosan nincs megoldás, ha a válasz *igaz*, akkor lehet hogy van, de elfogadjuk azt is, ha tévedett és valójában nincs.

---

### 10.12. Algoritmus Visszalépéses keresés módosítása szétválasztás és korlátozással

---

Bemenet:  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált részprobléma száma)

Bemenet:  $E$  - tömb<T> (az eddig választott részmegoldásokat tárolja)

Bemenet:  $van$  - logikai (eddig talált-e már teljes megoldást)

Kimenet:  $E$  - tömb<T> (a részmegoldások kiegészítve az aktuális szint esetleges megoldásával)

Kimenet:  $van$  - logikai (talált-e már teljes megoldást, kiegészítve az aktuális szint vizsgálatával)

```
1: eljárás VISSZALÉPÉSESBNB($szint$, címszerint E , címszerint van)
2: $i \leftarrow 0$
3: ciklus amíg $\neg van \wedge i < M_{szint}$
4: $i \leftarrow i + 1$
5: ha $f_t(szint, R_{szint,i})$ akkor
6: ha $f_k(szint, R_{szint,i}, E)$ akkor
7: $E_{szint} \leftarrow R_{szint,i}$
8: ha $szint = n$ akkor
9: $van \leftarrow igaz$
10: különben
11: ha $f_b(szint, E)$ akkor
12: VISSZALÉPÉSESKERESÉS($szint+1$, E , van)
13: elágazás vége
14: elágazás vége
15: elágazás vége
16: elágazás vége
17: ciklus vége
18: eljárás vége
```

---

#### Felhasznált változók és függvények

- $i$  : Egy egész<sup>+</sup> típusú segédváltozó.
  - $n$  : A megoldandó részfeladatok száma.
  - $M_{szint}$  : A lehetséges részmegoldások száma a megadott szinten.
  - $R_{szint,i}$  : Az  $i$ . részmegoldás a megadott szinten.
  - $f_t$ ,  $f_k$ ,  $f_b$  : A keresési feltételeket meghatározó függvények.
-

A 10.12. algoritmus mutatja a módosítás eredményét. A változás minimális, a 11. sorban jelent meg egy új ellenőrzés. Amennyiben a  $f_b$  függvény hívása igazat ad vissza, az azt jelenti, hogy érdemes folytatni a keresést, van remény egy teljes megoldásra. Amennyiben hamisat, akkor felesleges belelépni ebbe az ágba.

Apróság, de vegyük észre, hogy a  $E$  vektor a  $f_b$  hívásakor már tartalmazza az aktuális szint eredményét is, tehát ezt nem kell külön paraméterként átadnunk.

### 10.3.3. Visszalépéses keresés alapú optimalizálás kiegészítése

Amennyiben a visszalépéses keresést optimalizálásra használjuk, akkor a továbblépéskori becslésnél már nem elégszünk meg azzal, hogy megadott út vezethet-e teljes megoldáshoz, hanem azt is szeretnénk megbecsülni, hogy mennyire jó lesz ez a megoldás. Tehát már azokból az irányokból is visszalépünk, amelyek ugyan teljes megoldáshoz vezetnének, de ez biztosan nem lesz jobb, mint az eddig talált legjobb.

Emiatt egy kicsit más hogyan definiáljuk a  $f_b$  függvényt, ami most nem logikai értéket, hanem egy jósági értéket ad vissza:

- $f_b(\text{szint}, E)$ : a  $\text{szint}$ -edik részfeladat megoldását követően, amennyiben az előző és az aktuális szinteken az  $E$  vektorban található részmegoldásokat választottuk, akkor egy „felső becslést” ad arra, hogy a maradék részfeladatok megoldásával mennyivel tud növekedni a már meglévő jósági érték?

A 10.13. algoritmus mutatja a módosítás eredményét. A változás itt is hasonló, a 14. sorban jelent meg egy új ellenőrzés. Amennyiben az eddig elérte jósági értékhez hozzáadjuk a  $f_b$  függvény által visszaadott értéket (ami a megadott úton még elérhető további jóság növekmény felső becslése), és így az eddig talált optimum jóságánál nagyobb értéket kapunk, akkor érdemes folytatni a keresést, van remény egy teljes megoldásra. Amennyiben az érték kisebb, akkor nincs értelme itt továbblépni, hiszen a legideálisabb esetben se jutunk jobb megoldáshoz, mint az eddig feltételezett optimális.

#### Megjegyzés

Jogos lehet a gondolat, hogy az  $f_b$  miért nem magának az elérhető legjobb megoldásnak a jóságát adja vissza, és miért csak egy „növekményt”. Ennek oka pusztán didaktikai, így jobban látszik, hogy hol válik el egymástól a már rögzített megoldásra vonatkozó jóság és a jövőbeli becslés. Bár így látványos, de ez olyan feladatoknál nem is használható, ahol az egyes részmegoldásokra nem is értelmezhető jóság, csak a teljes megoldásra.

### 10.3.4. A 0-1 hátizsák probléma megoldása

Vizsgáljuk meg a megadott módszert a már jól ismert feladat megoldásával. Elsőként nézzük meg a szétválasztás és a korlátozás megválasztását:

- Szétválasztás: a teljes feladatot a többi rekurzív megoldásnál megszokott módon bontjuk szét kisebb részfeladatokra. Az  $n$  darab tárgy elhelyezését visszavezetjük két kisebb részfeladat megoldására: feltételezzük, hogy az utolsó elem benne van a zsákban és ez alapján számolunk optimumot a maradékra, illetve feltételezzük, hogy az utolsó nincs benne a zsákban és ez alapján számolunk optimumot a maradékra.
- Korlátozás: a korlátozás során a még vizsgálatra váró részprobléma optimális pakolási értékére adunk egy felső becslést. Kérdés, hogy ezt miként tehetjük meg, de azonnal kínálja magát két lehetőség:
  - Összeadjuk az összes még rendelkezésre álló tárgy pakolási értékét. Ez nyilvánvalóan egy felső becslés a maradék helyre még bepakolható tárgyak összértékére, ennél több nem érhető el.
  - Nem sokkal bonyolultabb, de mégis precízebb megoldás, ha figyelembe vesszük a még rendelkezésre álló szabad helyet is. Végignézzük az összes még hátralévő tárgyat, amennyiben a tárgy önmagában befér a zsákban maradt szabad helyre, akkor annak értékét hozzáadjuk egy segédváltozóhoz. A segédváltozó értéke tehát azt mutatja, hogy milyen optimális pakolási értéket érhettünk el, ha az összes önmagában beférő tárgyat bele fogjuk tudni rakni a zsákba. Ez

---

**10.13. Algoritmus** Visszalépéses keresés alapú optimalizálás szétválasztás és korlátozással

---

Bemenet:  $szint$  - egész<sup>+</sup> (az aktuálisan vizsgált részprobléma száma)

Bemenet:  $E$  - tömb<T> (az eddig választott részmegoldásokat tárolja)

Bemenet:  $van$  - logikai (eddig talált-e már teljes megoldást)

Bemenet:  $O$  - tömb<T> (az eddig talált legjobb optimális eredmény)

Kimenet:  $E$  - tömb<T> (a részmegoldások kiegészítve az aktuális szint esetleges megoldásával)

Kimenet:  $van$  - logikai (talált-e már teljes megoldást, kiegészítve az aktuális szint vizsgálatával)

Kimenet:  $O$  - tömb<T> (az optimális eredmény a szint vizsgálata után)

```
1: eljárás VISSZALÉPÉSESBNB($szint$, címszerint E , címszerint van , címszerint O)
2: $i \leftarrow 0$
3: ciklus amíg $i < M_{szint}$
4: $i \leftarrow i + 1$
5: ha $f_t(szint, R_{szint,i})$ akkor
6: ha $f_k(szint, R_{szint,i}, E)$ akkor
7: $E_{szint} \leftarrow R_{szint,i}$
8: ha $szint = n$ akkor
9: ha $\neg van \vee JÓSÁG(E) > JÓSÁG(O)$ akkor
10: $O \leftarrow E$
11: elágazás vége
12: $van \leftarrow igaz$
13: különben
14: ha $JÓSÁG(E) + f_b(szint, E) > JÓSÁG(O)$ akkor
15: VISSZALÉPÉSESBNB($szint+1$, E , van , O)
16: elágazás vége
17: elágazás vége
18: elágazás vége
19: elágazás vége
20: ciklus vége
21: eljárás vége
```

---

**Felhasznált változók és függvények**

- $i$  : Egy egész<sup>+</sup> típusú segédváltozó.
  - $n$  : A megoldandó részfeladatok száma.
  - $M_{szint}$  : A lehetséges részmegoldások száma a megadott szinten.
  - $R_{szint,i}$  : Az  $i$ . részmegoldás a megadott szinten.
  - $f_t$ ,  $f_k$ ,  $f_b$  : A keresési feltételeket meghatározó függvények.
  - $JÓSÁG(E)$  : Egy függvény, ami megadja a paraméterként átadott  $E$  végeredmény értékét.
-

nyilvánvalóan egy felső becslés, hiszen valószínűleg nem fog minden beleférni, de számunkra most csak a fontos, hogy ennél jobb eredményt biztosan nem kaphatunk.

Ezt utóbbi megoldást valósítja meg az  $f_b$  függvény.

A pszeudokódnál (10.14. algoritmus) szándékoltan megtartottuk az előző visszalépéses keresésnél már megismert alapvető felépítést. Az algoritmus működése szinte teljesen ugyanaz, egy pontban történt változtatás, a 22. sorban. Itt látható, hogy a következő rekurziós hívás előtt ellenőrizzük, hogy az arra az ágra vonatkozó felső becslést figyelembe véve kaphatunk-e jobb eredményt az eddig elérnél. Ha igen, akkor a függvény meghívja önmagát, ha nem, akkor pedig nem történik rekurzív hívás.

A felső becslést az  $f_b$  függvény végzi, aminek két paramétere van ( $szint$  és  $E$ ). Ezek alapján a visszatérési értéke egy felső becslés arra vonatkozóan, hogy az  $E$  által aktuálisan tárolt (tehát 1-től  $szint$ -ig már ismert) részeredményeket figyelembe véve, a hátralévő (tehát  $szint$  utáni) tárgyak felhasználásával milyen optimális pakolást lehet elérni.

### 10.3.5. A módszer értékelése

Általában elmondható, hogy az itt megismert technika túlmutat az eddigi hasonló módszerek lehetőségein. Mindegyik hasonló módszer többé-kevésbé próbálkozással kereste meg az optimális eredményt, közülük néhány volt annyira hatékony, hogy nem folytatott olyan útvonalakat, ahol eleve nem fog elfogadható eredményt (érvényes pakolást) találni. A most megismert módszer pedig ezt annyiból továbbfejleszti, hogy próbálja kiszűrni azokat az ágakat is, ahol ugyan lehet elfogadható eredmény (érvényes pakolás), de az várhatóan nem lesz jobb, mint az eddig talált legjobb (optimális pakolás).

A módszer előnyei

- A fentieknek megfelelően egy kellőképpen finomhangolt algoritmus nagyon sok felesleges lépést el tud kerülni.
- Nem csak a már rögzített részeredményeken alapul a továbblépés, hanem a még nem feldolgozott részproblémát is figyelembe veszi.

A módszer hátrányai

- Mindezek az előnyök csak akkor használhatók ki, ha kellőképpen jó szétválasztó és korlátozó függvényeket írunk. Ezek kidolgozása gyakran meglehetősen bonyolult lehet.

---

**10.14. Algoritmus** A 0-1 hátizsákprobléma megoldása „szétválasztás és korlátozás” technikával

---

Bemenet:  $szint$  - egész<sup>+</sup> (hányadik szintet vizsgáljuk)

Bemenet:  $E$  - tömb<logikai> (az eddigi részmegoldások)

Kimenet: . - logikai (a részmegoldások nem zárják ki egymást)

1: függvény  $f_k(szint, E)$

2: vissza  $\text{ÖSSZSÚLY}(E) \leq W$

3: függvény vége

Bemenet:  $szint$  - egész<sup>+</sup> (hányadik szintet vizsgáljuk)

Bemenet:  $E$  - tömb<logikai> (az eddigi részmegoldások)

Kimenet: . - szám<sup>+</sup> (felső becslés a várható optimumra)

4: függvény  $f_b(szint, E)$

5:  $b \leftarrow 0$

6: ciklus  $i \leftarrow szint + 1$ -től  $N$ -ig

7: ha  $\text{ÖSSZSÚLY}(E) + w_i \leq W$  akkor

8:  $b \leftarrow b + p_i$

9: elágazás vége

10: ciklus vége

11: vissza  $b$

12: függvény vége

Bemenet:  $szint$  - egész<sup>+</sup> (hányadik szintet vizsgáljuk)

Bemenet:  $E$  - tömb<logikai> (az aktuálisan vizsgált részmegoldás, aminek megváltozhat az értéke)

Bemenet:  $O$  - tömb<logikai> (az eddig talált legjobb megoldás, aminek megváltozhat az értéke)

13: eljárás BACKTRACK( $szint, \text{címszerint } E, \text{címszerint } O$ )

14: ciklus  $i \leftarrow 0$ -től 1-ig

15:  $E[szint] \leftarrow (i = 0)$  ▷ ha  $i=0$  akkor legyen igaz, különben hamis

16: ha  $f_k(szint, E)$  akkor

17: ha  $(szint = n)$  akkor

18: ha  $\text{ÖSSZÉRTÉK}(E) > \text{ÖSSZÉRTÉK}(O)$  akkor

19:  $O \leftarrow E$

20: elágazás vége

21: különben

22: ha  $\text{ÖSSZÉRTÉK}(E) + f_b(szint, E) > \text{ÖSSZÉRTÉK}(O)$  akkor

23: BACKTRACK( $szint+1, E, O$ )

24: elágazás vége

25: elágazás vége

26: elágazás vége

27: ciklus vége

28: eljárás vége ▷ visszalépéskor  $E[szint]$  értéke minden hamis

Kimenet: . - szám<sup>+</sup> (egy optimális pakolás összértéke)

29: függvény HÁTIZSÁKB  
NB

30:  $E \leftarrow [\text{hamis}, \text{hamis}, \dots, \text{hamis}]$

31:  $O \leftarrow [\text{hamis}, \text{hamis}, \dots, \text{hamis}]$

32: BACKTRACK(1,  $E, O$ )

33: vissza  $\text{ÖSSZÉRTÉK}(O)$

34: függvény vége

---

**Felhasznált változók és függvények**

- $n$  : A hátizsákba pakolható tárgyak száma.
  - $W$  : A hátizsák mérete (a belepakolható tárgyak maximális összsúlya).
  - $w_i$  : A hátizsákba pakolható i. tárgy súlya ( $1 \leq i \leq n$ ).
  - $p_i$  : A hátizsákba pakolható i. tárgy hasznossága ( $1 \leq i \leq n$ ).
  - $E, O$  : Az aktuálisan vizsgált/eddig legjobb pakolás adatai (logikai tömbök, ahol  $E[i]$  és  $O[i]$  azt mutatja, hogy az i. tárgy benne van-e a zsákban ( $1 \leq i \leq n$ )).
-

## **IV. rész**

# **Speciális adatszerkezetek**

## 11. fejezet

# Részben kiegyensúlyozott fák

### 11.1. Kupac adatszerkezet

#### 11.1.1. Felépítése

Fákat nem csak dinamikus formában tudunk tárolni, hanem itt is lehetőség van a tömb alapú, statikus tárolásra. Ennek egyik lehetséges formája a *kupac*<sup>1</sup> adatszerkezet. Többféle kupac szerkezetet dolgoztak már ki, ezek közül a legegyszerűbb a *bináris kupac*, ami egy majdnem teljes bináris fa egy tömbben eltárolva. Ennek jellemzői:

- Egy  $E$  tömbben  $n$  darab  $T$  típusú elemet tárolunk ( $n \leq E.\text{méret}$ ).
- A kupacban tárolt elemek valójában egy fát reprezentálnak, az alábbiak szerint:
  - a tömb első eleme ( $E[1]$ ) a fa gyökere,
  - a tömb minden  $i$ . elemének baloldali gyerekének indexe  $2i$ ,
  - a tömb minden  $i$ . elemének jobboldali gyerekének indexe  $2i + 1$ .
- A fa rendezett, ami *maximum-kupac* esetén azt jelenti, hogy az egyes elemek gyerekei (és így részfaik) nem tartalmazhatnak nagyobb elemeket.

A 11.1. ábra egy példát mutat a kupac adatszerkezetre. Példaként ellenőrizhetők a fenti szabályok: a 24-es elem a tömb 2. pozíciójában található ( $i = 2$ ), ennek megfelelően baloldali gyereke a  $2i = 4$ . pozícióban (12), jobboldali gyereke pedig a  $2i + 1 = 5$ . pozícióban (22). A tömb első eleme a fa gyökere (50). Illetve látható a fa majdnem teljes mivolta: a felső 3 szint teljes, az utolsó szint pedig balról jobbra van feltöltve hiánytalanul.

Ez a szerkezet nagyon hatékonyan tudja eltárolni egy bináris fa csúcsait, hiszen nincs szükség mutatók használatára. Ez persze egy jelentős kötöttséggel is jár: a kupac csak bizonyos formájú fákat tud eltárolni. A bevezetőben említett „majdnem kiegyensúlyozott” alatt azt értjük, hogy a fa a legalsó szintjét kivéve teljes (tehát minden csúcsnak pontosan két gyereke van), a legalsó szint pedig balról jobbra van feltöltve elemekkel.

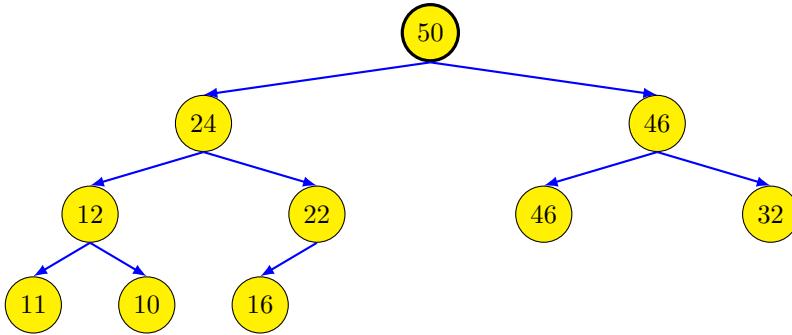
#### Megjegyzés

A kupac ugyan egy bináris fát tárol, de ennek rendezettsége nem azonos a 6. fejezetben tanult bináris keresőfáéval. A bináris keresőfa esetében a gyökér elem baloldali részfájában kisebb, a jobboldaliban pedig nagyobb elemek voltak (vagy épp fordítva). A kupac esetében a gyökér egy maximális elemet tartalmaz, minden részfában csak ugyanekkor, vagy ennél kisebb elemek lehetnek (iránytól függetlenül).

A már bevezetett műveletek egyszerűen megoldhatók a 11.1. algoritmus pszeudokódjaival. Miként a kódok között látható, egyszerűen kiszámolható egy tetszőleges gyökértől különböző elem szülője is.

<sup>1</sup>heap

|           |    |    |    |    |    |    |    |    |    |    |
|-----------|----|----|----|----|----|----|----|----|----|----|
| <i>E:</i> | 50 | 24 | 46 | 12 | 22 | 46 | 32 | 11 | 10 | 16 |
|-----------|----|----|----|----|----|----|----|----|----|----|



11.1. ábra. Egy kupac tömb és fa reprezentációja.

Mivel a fa véges, ezért meg kell tudnunk vizsgálni, hogy egy megadott csomópontnak van-e egyáltalán bal- vagy jobboldali gyereke. Ezt egyszerűen úgy tudjuk megtenni, hogy ha  $2i < n$ , akkor az  $i$ . elemnek van baloldali gyereke, egyébként nincs. Ha  $2i + 1 < n$ , akkor az  $i$ . elemnek van jobboldali gyereke, egyébként nincs. Érdemes észrevenni azt is, hogy ha egy elemnek nincs baloldali gyereke, akkor biztos nem lehet jobboldali se.

---

### 11.1. Algoritmus Bináris kupac segédműveletei

---

**Bemenet:**  $i$  - egész<sup>+</sup> (a kupac egy elemének indexe)

**Kimenet:**  $\cdot$  - egész<sup>+</sup> (az  $i$ . elem szülőjének indexe)

1: függvény KUPAC.SZÜLŐ( $i$ )

2: vissza  $\lfloor i/2 \rfloor$

3: függvény vége

**Bemenet:**  $i$  - egész<sup>+</sup> (a kupac egy elemének indexe)

**Kimenet:**  $\cdot$  - egész<sup>+</sup> (az  $i$ . elem bal gyerekének indexe)

4: függvény KUPAC.BAL( $i$ )

5: vissza  $2 * i$

6: függvény vége

**Bemenet:**  $i$  - egész<sup>+</sup> (a kupac egy elemének indexe)

**Kimenet:**  $\cdot$  - egész<sup>+</sup> (az  $i$ . elem jobb gyerekének indexe)

7: függvény KUPAC.JOBB( $i$ )

8: vissza  $2 * i + 1$

9: függvény vége

---

### 11.1.2. Kupactulajdonság fenntartása

Bármelyik tömböt elköpzelhetjük faként, de ettől még nem lesz kupac, mivel ehhez a rendezettséget is biztosítani kell. Ehhez meg kell ismernünk két segédműveletet.

Az egyik a kupactulajdonság fenntartása művelet. Ennek előfeltételei:

- van egy  $E$  tömbben tárolt  $n$  méretű bináris fa;
- ez a gyökérelem kivételével megfelel a kupac rendezettségének.

A kupac karbantartás művelet eredményeként a teljes  $n$  méretű fára biztosítjuk a rendezettséget, tehát az egész megfelel majd a kupac tulajdonságának.

A rendezés alapelve egyébként meglehetősen egyszerű: tudjuk, hogy az egész fa megfelel a kupac rendezettségének, kivéve esetleg a gyökér elemet. Ezért meg kell vizsgálni, hogy a gyökér és annak bal- és jobboldali gyerekei (ha vannak) megfelelnek-e a kupac tulajdonságának:

- ha igen, akkor nincs további teendőnk, hiszen az előfeltételek és az előző vizsgálat eredményeként biztos, hogy az egész adatszerkezet rendezett;

- ha nem, akkor lokálisan helyreállítjuk a kupactulajdonságot:
  1. a bal- és jobboldali gyerekek közül kiválasztjuk a nagyobbat;
  2. ezt kicséréljük a gyökérelemmel, amivel garantáljuk a gyökér és két gyereke között a kupactulajdonságot, hiszen a legnagyobb elem van most már a gyökérben, a másik kettő pedig alatta (ezek sorrendje nem fontos);
  3. a csere során nem érintett részfa eddig is rendezett volt, ez továbbra is fennáll;
  4. a csere által érintett irányban azonban ez még nem biztos. Az viszont tudható, hogy ennek a részfának is rendezettsnek kell lennie az új gyökér kivételével. Ezért erre a részfára rekurzívan meghívjuk a kupac karbantartás műveletét.

Ezt a módszert írja le a 11.2. algoritmus. A függvény  $i$  paramétere a kupacolandó részfa gyökerének tömbbeli indexét határozza meg.

Az első két sor beállítja a  $b$  és  $j$  indexeket, amelyek az  $i$ . elem bal- és jobboldali gyerekének indexét mutatják (már ha vannak ilyenek).

A következő sorok célja, hogy kiválasztja az  $E[i]$ ,  $E[b]$ ,  $E[j]$  elemek közül azt, amelyik még része a kupacnak, és a legnagyobb értéket tárolja.

A 4. sor elsőként ellenőrzi, hogy létezik-e egyáltalán a baloldali gyerek ( $b \leq n$ ), és ha igen, akkor megvizsgálja, hogy az nagyobb-e mint az  $i$ . elem. Ha igen, akkor ennek értéke kerül a  $\max$  változóba, egyébként a  $i$ .

A 9. sor elsőként ellenőrzi, hogy létezik-e egyáltalán a jobboldali gyerek ( $j \leq n$ ), és ha igen, akkor megnézi, hogy nagyobb-e mint az eddigi legnagyobbnak feltételezett  $\max$ . elem. Ha igen, akkor ennek értéke kerül a  $\max$  változóba.

Ezen a ponton (12. sor) már tudjuk, hogy a vizsgált elemek közül a legnagyobb indexét a  $\max$  változó tartalmazza. Amennyiben ennek az értéke egyenlő a  $i$ -vel, akkor nincs további teendőnk, hiszen a tömb már eleve teljesíti a kupac tulajdonságát.

Amennyiben ez nem igaz, akkor elvégezzük a szükséges cserét (13. sor), ezzel a vizsgált elemekre lokálisan a teljesítjük a kupac tulajdonságot.

A teljes tömbre azonban ez még nem garantálható, mivel csere esetén a lejjebb csúsztatott gyökér elem és a most alatta lévő gyerekek rendezettségéről nem tudunk semmit. Ezért a 14. sorban a függvény rekurzívan meghívja önmagát erre a pontra, és erre a részfára is elvégzi a rendezést.

Könnyen belátható, hogy legfeljebb a kupac mélységének megfelelő számú lépésünk lesz, hiszen vagy nem történik csere és véget ér az algoritmus, vagy pedig egy szinttel lejjebb fut tovább. Tehát véges számú lépés után végetér a függvény, eredményeként pedig a „majdnem” kupacból készít egy kupacot.

---

## 11.2. Algoritmus Kupac tulajdonság fenntartása

---

Bemenet:  $i$  - egész<sup>+</sup> (a vizsgálandó részfa gyökerének indexe)

```
1: eljárás KUPAC.KUPACOL(i)
2: $b \leftarrow \text{BAL}(i)$
3: $j \leftarrow \text{JOBB}(i)$
4: ha $b \leq n \wedge E[b] > E[i]$ akkor
5: $\max \leftarrow b$
6: különben
7: $\max \leftarrow i$
8: elágazás vége
9: ha $j \leq n \wedge E[j] > E[\max]$ akkor
10: $\max \leftarrow j$
11: elágazás vége
12: ha $\max \neq i$ akkor
13: $E[i] \leftrightarrow E[\max]$
14: KUPACOL(\max)
15: elágazás vége
16: eljárás vége
```

---

### Felhasznált változók és függvények

- $b, j, \max$  : egész<sup>+</sup> típusú segédváltozók.
  - $E$  : Az elemeket tartalmazó tömb.
  - $n$  : A kupacban lévő elemek száma ( $n \leq E.\text{méret}$ ).
  - $\text{BAL}(i)$  : Az  $i$ . elem bal gyerekének indexe (11.1. algoritmus).
  - $\text{JOBB}(i)$  : Az  $i$ . elem jobb gyerekének indexe (11.1. algoritmus).
- 

A fentiekre mutat példát a 11.2. ábra.

### 11.1.3. Kupac építés

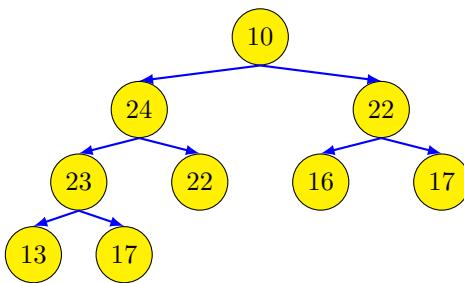
Az előző eljárás segítségével egy „majdnem kupacból” már tudunk egy teljesen rendezett kupacot építeni. Egy tetszőleges tömbre azonban még nem használható, mivel csak a gyökérelem esetében engedtük meg a rendezetlenséget.

A következő, **KUPACOTÉPÍT** eljárás célja éppen ez, hogy egy rendezett kupacot tudjuk készíteni egy tetszőleges tömbből. Működési elve, miként a pszeudokódból is látható, nagyon egyszerű:

1. nyilvánvaló, hogy a fa levelei önmagukban teljesítik a kupac tulajdonságot (mivel nincs gyerekük, ezért nincs aki megszegné a rendezettséget);
2. megvizsgáljuk a közvetlenül a levelek felett lévő belső pontokat, amelyekre már meghívható a **KUPACOL** eljárás, hiszen ezekre igaz a feltétel, hogy a gyökér kivételével a fa többi része (amik jelen esetben csak levelek lehetnek) rendezettek;
3. az esetleges kupacolások után, folytathatjuk az alulról 3. szinttel, hiszen most már ezekre is igaz lesz, hogy a gyökér kivételével az alatta lévő rész megfelel a rendezettségnek (hiszen az előző művelet során ezt garantáltuk);
4. minden egyes szint kupacolása után továbbléphetünk a felette lévő szintre és folyamatosan elvégezzük az egyes szintekre a kupacolást;
5. végül magára a gyökér elemre is elvégezzük a kupacolást. Ezt követően az egész tömb megfelel a kupac tulajdonságának.

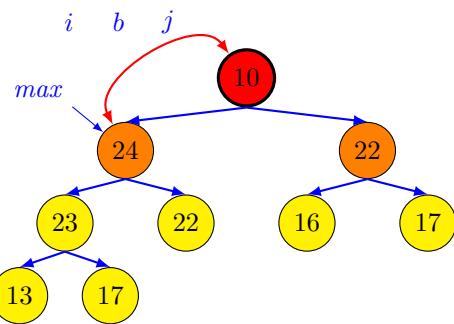
A 11.3. algoritmus mutatja a fenti műveleteket, aminek a megvalósítása meglepően egyszerű. Magyarázatot talán csak a ciklusban látható  $\lfloor n/2 \rfloor$  számítás igényel. Ennek az a szerepe, hogy ezzel a számítással minden megkapjuk a kupac utolsó (tehát a legalsó szinten lévő legjobboldalibb, így a tömbben is legnagyobb indexű) belső csúcsát. A tömbben ezt követő elemek minden levelek. Ez, illetve az előtte lévők pedig minden belső csúcsok. Ez azért praktikus számunkra, mivel tudjuk, hogy a levelek már

|           |                                              |
|-----------|----------------------------------------------|
| <i>E:</i> | [10   24   22   23   22   16   17   13   17] |
|-----------|----------------------------------------------|



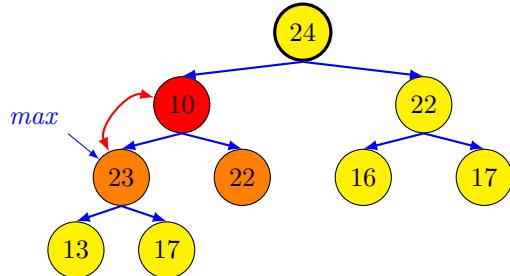
(a) A kupacolás előfeltételeként a gyökér elem kivételével a többi elem a kupactulajdonságban megadott módon rendezett.

|           |                                              |
|-----------|----------------------------------------------|
| <i>E:</i> | [10   24   22   23   22   16   17   13   17] |
|-----------|----------------------------------------------|



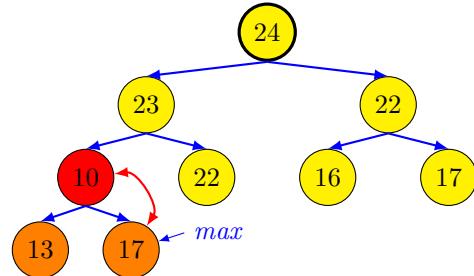
(b) Az algoritmus indulásakor  $i=1$ , ezért a gyökeret és gyerekeit vizsgáljuk ( $b = 2i$  és  $j = 2i+1$ ). Ezek közül a legnagyobb a 24-es. Ezzel cserélünk.

|           |                                              |
|-----------|----------------------------------------------|
| <i>E:</i> | [24   10   22   23   22   16   17   13   17] |
|-----------|----------------------------------------------|



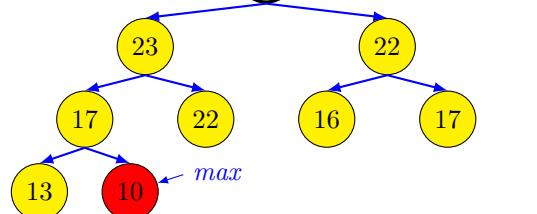
(c) A rekurzív hívás után ennek az elemnek a környezetét vizsgáljuk. Itt is cserélünk a legnagyobbal.

|           |                                              |
|-----------|----------------------------------------------|
| <i>E:</i> | [24   23   22   10   22   16   17   13   17] |
|-----------|----------------------------------------------|



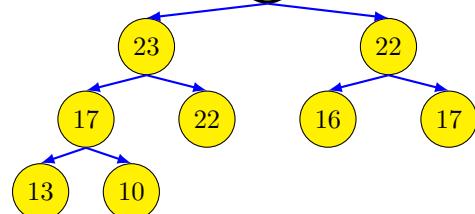
(d) A következő rekurzív hívás után a 17-es a legnagyobb elem, ezért itt is cserélünk, majd rekurzív hívás következik.

|           |                                              |
|-----------|----------------------------------------------|
| <i>E:</i> | [24   23   22   17   22   16   17   13   10] |
|-----------|----------------------------------------------|



(e) Az utolsó elem vizsgálatakor  $b > n$  és  $j > n$ , ezért a  $\text{max} = i$ . Így nincs csere, sem további vizsgálat.

|           |                                              |
|-----------|----------------------------------------------|
| <i>E:</i> | [24   23   22   17   22   16   17   13   10] |
|-----------|----------------------------------------------|



(f) Az eljárás eredményeképpen egy rendezett kúpacot kaptunk.

11.2. ábra. Kupac tulajdonság fenntartása

önmagukban is teljesítik a kupac tulajdonságot, tehát nem kell őket vizsgálnunk, a **KUPACOL** eljárást elég az utolsó belső ponttól indítani.

A ciklus pedig ugyanazokat a lépéseket hajtja végre, mint amiket az előző leírás is bemutatott, egyesével elvégzi a kupacolás műveletet alulról felfelé (és azon belül jobbról balra) az egyes csomópontokra. Mindezt a gyökér elem feldolgozásáig folytatjuk, amikor is elkészült az egész rendezett kupac.

---

### 11.3. Algoritmus Kupac építése

---

- 1: **eljárás** KUPAC.KUPACOTÉPÍT( )
  - 2:     **ciklus**  $i \leftarrow \lfloor n/2 \rfloor$ -től 1-ig
  - 3:         KUPACOL( $i$ )
  - 4:     **ciklus vége**
  - 5: **eljárás vége**
- 

#### Felhasznált változók és függvények

- $i$  : Egy **egész<sup>+</sup>** típusú segédváltozó.
  - $n$  : A kupacban lévő elemek száma ( $n \leq E.\text{méret}$ ).
  - **KUPACOL( $i$ )** : Kupacolja a tömböt az  $i$ . elemtől kezdve (11.2. algoritmus).
- 

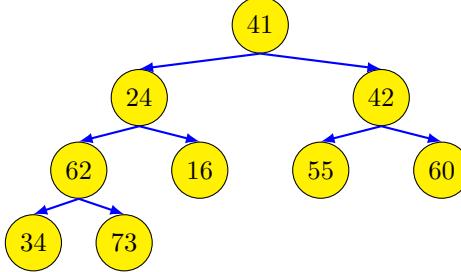
A kupac építésre mutat példát a 11.3. ábra.

#### 11.1.4. Alap műveletek összefoglalása

A fent leírt mezőket és műveleteket mutatja a 11.5. ábra. Önmagában a **Kupac** osztályt nem használható semmire, ezért ezt egy absztrakt osztályban valósítjuk meg. Ennek megfelelően van ugyan egy **E** tömbünk, de nem foglalkozunk azzal, hogy honnan kerülnek ebbe az adatok. Így szükségtelenek a konstruktor/destruktőr műveletek is.

*E:*

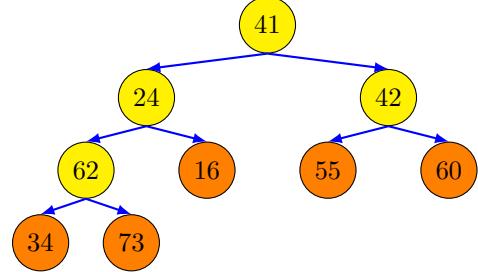
|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 41 | 24 | 42 | 62 | 16 | 55 | 60 | 34 | 73 |
|----|----|----|----|----|----|----|----|----|



(a) Egy rendezetlen *E* tömbből indulunk ki, ez még látványosan nem kupac.

*E:*

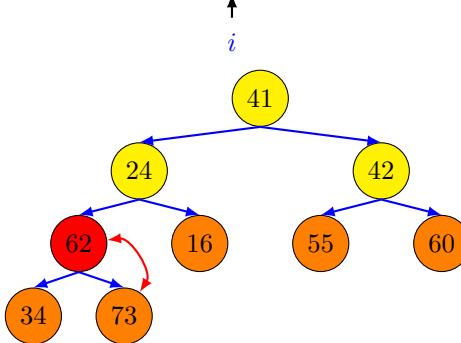
|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 41 | 24 | 42 | 62 | 16 | 55 | 60 | 34 | 73 |
|----|----|----|----|----|----|----|----|----|



(b) Látható, hogy az első  $\lfloor n/2 \rfloor = 4$  elem belső csúcs, a többi levél. Ez utóbbiak minden teljesítik a kupac tulajdonságokat önmagukban.

*E:*

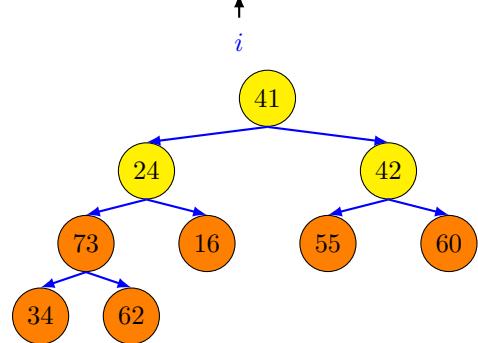
|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 41 | 24 | 42 | 62 | 16 | 55 | 60 | 34 | 73 |
|----|----|----|----|----|----|----|----|----|



(c) Elsőként *i* értéke 4, innen indul a kupacolás. Ez egy cserét fog végezni.

*E:*

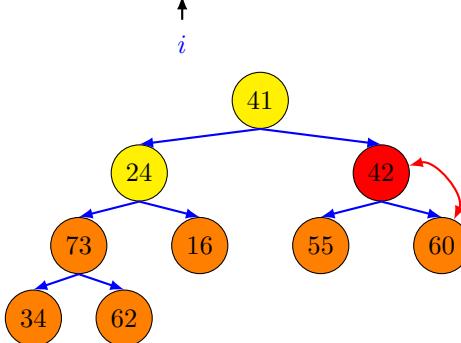
|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 41 | 24 | 42 | 73 | 16 | 55 | 60 | 34 | 62 |
|----|----|----|----|----|----|----|----|----|



(d) A kupacolás véget ért, a 4. indexű elemtől induló részfa is már kupac.

*E:*

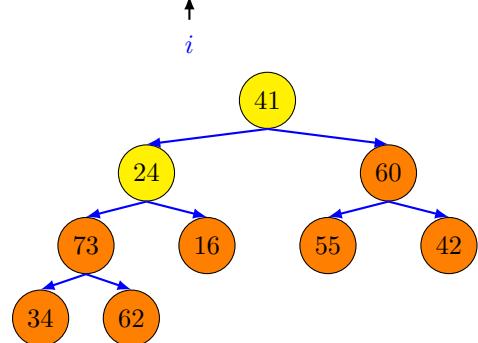
|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 41 | 24 | 60 | 73 | 16 | 55 | 42 | 34 | 62 |
|----|----|----|----|----|----|----|----|----|



(e) A *i* értékét csökkentjük, a 42-es gyökerű részfára újra indul a kupacolás. Ez megint egy cserét fog okozni.

*E:*

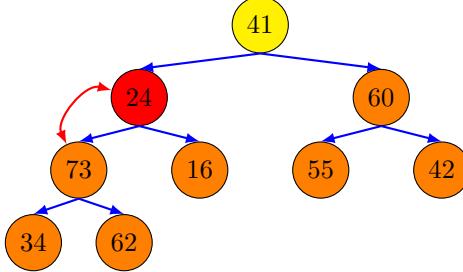
|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 41 | 24 | 60 | 73 | 16 | 55 | 42 | 34 | 62 |
|----|----|----|----|----|----|----|----|----|



(f) A kupacolás véget ért, a 3. indexű gyökérrel bíró részfa is teljesít a kupac tulajdonságokat.

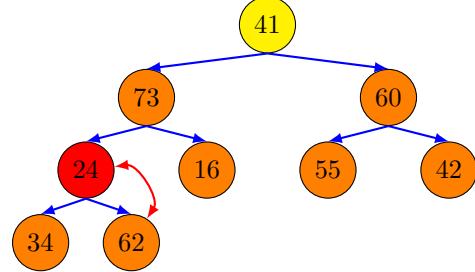
11.3. ábra. Kupac tulajdonság fenntartása

|           |                                                                                                                                          |    |    |    |    |    |    |    |    |    |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|----|----|----|----|
| <i>E:</i> | <table border="1"> <tr><td>41</td><td>24</td><td>60</td><td>73</td><td>16</td><td>55</td><td>42</td><td>34</td><td>62</td></tr> </table> | 41 | 24 | 60 | 73 | 16 | 55 | 42 | 34 | 62 |
| 41        | 24                                                                                                                                       | 60 | 73 | 16 | 55 | 42 | 34 | 62 |    |    |
| $i$       | ↑                                                                                                                                        |    |    |    |    |    |    |    |    |    |



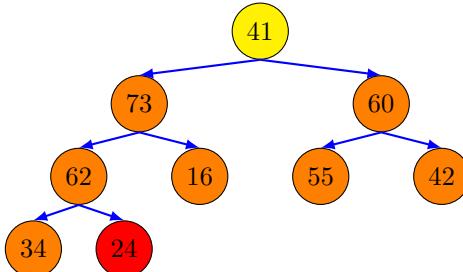
(a) A  $i$  értékét tovább csökkentjük, a 24-es csúcsból is indul egy kupacolás. Ennek első cseréjét mutatja az ábra.

|           |                                                                                                                                          |    |    |    |    |    |    |    |    |    |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|----|----|----|----|
| <i>E:</i> | <table border="1"> <tr><td>41</td><td>73</td><td>60</td><td>24</td><td>16</td><td>55</td><td>42</td><td>34</td><td>62</td></tr> </table> | 41 | 73 | 60 | 24 | 16 | 55 | 42 | 34 | 62 |
| 41        | 73                                                                                                                                       | 60 | 24 | 16 | 55 | 42 | 34 | 62 |    |    |
| $i$       | ↑                                                                                                                                        |    |    |    |    |    |    |    |    |    |



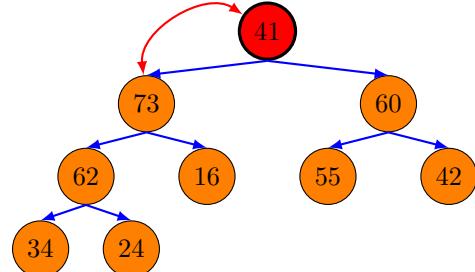
(b) Ez a kupacolás egy következő cserét is igényel.

|           |                                                                                                                                          |    |    |    |    |    |    |    |    |    |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|----|----|----|----|
| <i>E:</i> | <table border="1"> <tr><td>41</td><td>73</td><td>60</td><td>62</td><td>16</td><td>55</td><td>42</td><td>34</td><td>24</td></tr> </table> | 41 | 73 | 60 | 62 | 16 | 55 | 42 | 34 | 24 |
| 41        | 73                                                                                                                                       | 60 | 62 | 16 | 55 | 42 | 34 | 24 |    |    |
| $i$       | ↑                                                                                                                                        |    |    |    |    |    |    |    |    |    |



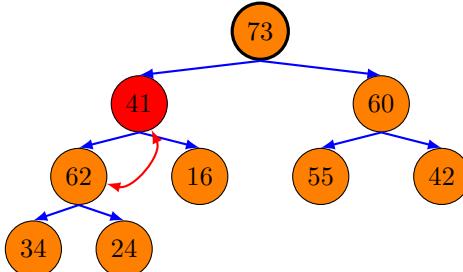
(c) Ez a kupacolás is véget ért.

|           |                                                                                                                                          |    |    |    |    |    |    |    |    |    |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|----|----|----|----|
| <i>E:</i> | <table border="1"> <tr><td>41</td><td>73</td><td>60</td><td>62</td><td>16</td><td>55</td><td>42</td><td>34</td><td>24</td></tr> </table> | 41 | 73 | 60 | 62 | 16 | 55 | 42 | 34 | 24 |
| 41        | 73                                                                                                                                       | 60 | 62 | 16 | 55 | 42 | 34 | 24 |    |    |
| $i$       | ↑                                                                                                                                        |    |    |    |    |    |    |    |    |    |



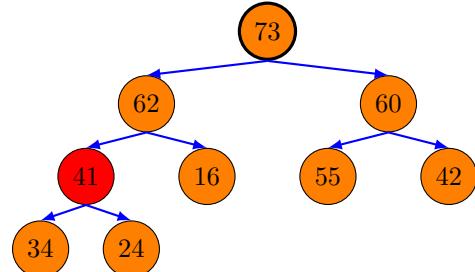
(d) A gyökérből is indítunk egy kupacolást. Az ábrán látható az első csere.

|           |                                                                                                                                          |    |    |    |    |    |    |    |    |    |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|----|----|----|----|
| <i>E:</i> | <table border="1"> <tr><td>73</td><td>41</td><td>60</td><td>62</td><td>16</td><td>55</td><td>42</td><td>34</td><td>24</td></tr> </table> | 73 | 41 | 60 | 62 | 16 | 55 | 42 | 34 | 24 |
| 73        | 41                                                                                                                                       | 60 | 62 | 16 | 55 | 42 | 34 | 24 |    |    |
| $i$       | ↑                                                                                                                                        |    |    |    |    |    |    |    |    |    |



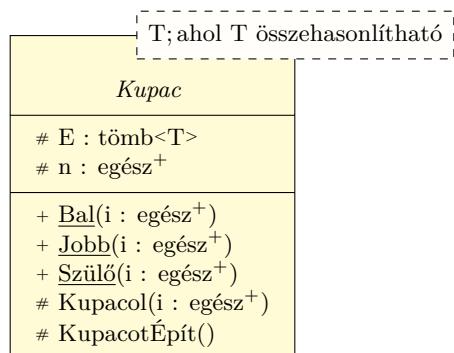
(e) A második cseré.

|           |                                                                                                                                          |    |    |    |    |    |    |    |    |    |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|----|----|----|----|
| <i>E:</i> | <table border="1"> <tr><td>73</td><td>41</td><td>60</td><td>62</td><td>16</td><td>55</td><td>42</td><td>34</td><td>24</td></tr> </table> | 73 | 41 | 60 | 62 | 16 | 55 | 42 | 34 | 24 |
| 73        | 41                                                                                                                                       | 60 | 62 | 16 | 55 | 42 | 34 | 24 |    |    |
| $i$       | ↑                                                                                                                                        |    |    |    |    |    |    |    |    |    |



(f) Ez a kupacolás is véget ért. A teljes adatszerkezet megfelel a kupac tulajdonságának.

11.4. ábra. Kupac tulajdonság fenntartása(folytatás)



11.5. ábra. Az absztrakt Kupac alapvető műveletei és mezői

## 11.2. Kupac rendezés

Eltávolodva az adatszerkezetek témakörétől, a kupac szerkezet egy hatékony rendezés alapja is lehet. Megvizsgálva egy kupac tulajdosággal bíró tömböt (11.1. ábra) látható, hogy az alapvetően nem rendezett sem növekvő, sem pedig csökkenő sorrendben. Rendezettség szempontjából egyet tudunk: a tömb legelső eleme biztos, hogy maximális értékű elem.

A kupac másik jellegzetessége, hogy ha csak a legelső elemet cseréljük ki egy kisebb elemre, akkor a **KUPACOL** eljárás hatékonyan ( $\mathcal{O}(\log_2 n)$ ) lépésben helyre tudja állítani a teljes tömbre vonatkozó kupac tulajdonságát, tehát ismét egy maximális méretű elem fog a gyökérbe kerülni.

A fentiek alapján a 11.4. algoritmus alapelve már érthető:

1. A meglévő rendezetlen tömb alapján felépítünk egy kupacot a már megismert **KUPACOTÉPÍT** művelet segítségével. Az  $E$  tömb tartalmazza az elemeket, az  $n$  értéke pedig egyenlő a kupac elemeinek számával (ami induláskor  $E.\text{méret}$ ).
2. Indítunk egy ciklust, aminek az első iterációjában a  $i$  értéke a kupacban lévő elemek számával egyenlő.
3. Tudjuk, hogy a kupac építés után a tömb első eleme maximális értékű lesz. Ezt a 4. sorban kicseréljük a tömb utolsó elemével. Ez az utolsó elem tehát már a nagyság szerinti rendezettség szerinti helyén lesz a tömbben.

Vegyük észre, hogy az  $n$  és az  $E.\text{méret}$  értékek elválnak egymástól. A statikus implementáció esetén az egyébként is általános, hogy maga az elemeket tartalmazó tömb nagyobb, mint a benne ténylegesen tárolt elemek száma. Itt azonban a rendezés egy lépése is épít arra a lehetőségre, hogy az  $E$  tömbnek csak egy részét tekintjük a kupac részeként. Kezdetben az egészet, majd minden egyes csere művelet után a tömb végére került elemet már nem számítjuk bele a kupacba. Ezt egyszerűen az  $n$  csökkentésével értjük el.

A kupac rendezés során felhasználjuk majd az előzőleg megismert karbantartási műveleteket. Ezért célszerűen a rendezést végző osztály az előzőleg megismert **Kupac** absztrakt osztály leszármazottja lesz (11.6. ábra). A **LÉTREHOZ** művelet megvalósítását itt már meg kell adni, bár ezt nem tárgyaljuk részletesen. A szerepe csak annyi, hogy a kupac belső  $E$  változójának értékül adj a paraméterként átadott  $A$  tömböt, az  $n$  értékét pedig beállítja ennek a méretére. Az értékkadást nem másolásként képzeljük el, hanem az  $E$  és az  $A$  referenciaiák ugyanarra a tömbre hivatkoznak. Tehát amikor az  $E$  elemeit módosítjuk, akkor az eredeti  $A$  is módosulni fog. Ezzel egy helyben rendezést fogunk tudni megvalósítani, hiszen amelyik tömböt áadtuk a **KupacRendezés** konstruktornak, azt fogja majd rendezni.

A rendezésre mutat egy példát a 11.7. ábra. Az algoritmus első lépése egy kupac építés, ezt már ismerjük, tehát tegyük fel, hogy a 11.3. ábrán már felépített kupacon folytatjuk a rendezés lépésein.

---

### 11.4. Algoritmus Kupac rendezés

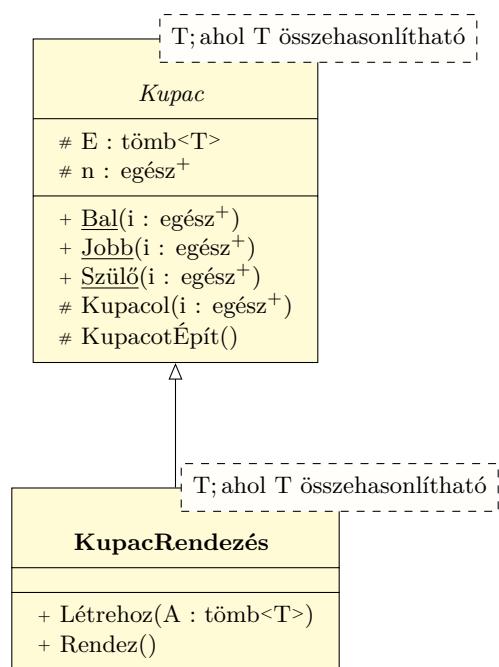
---

```
1: eljárás KUPACRENDEZÉS.RENDEZ()
2: KUPACOTÉPÍT()
3: ciklus $i \leftarrow n$ -től 2-ig
4: $E[1] \leftrightarrow E[i]$
5: $n \leftarrow n - 1$
6: KUPACOL(1)
7: ciklus vége
8: eljárás vége
```

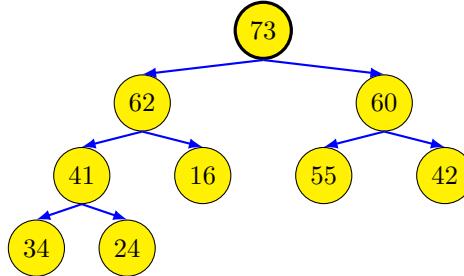
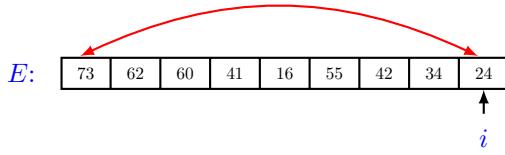
---

#### Felhasznált változók és függvények

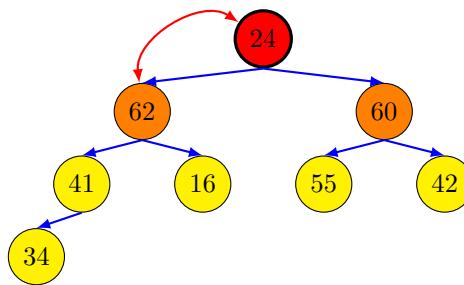
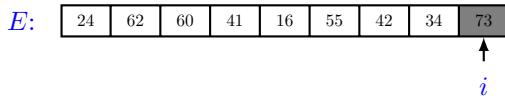
- $i$  : Egy *egész*<sup>+</sup> típusú segédváltozó.
  - $E$  : A rendezendő elemeket tartalmazó tömb, ezeket helyben rendezi.
  - $n$  : A kupacban lévő elemek száma ( $n \leq E.\text{méret}$ ).
  - **KUPACOTÉPÍT( $i$ )** : Felépít egy új kupacot (11.3. algoritmus).
  - **KUPACOL( $i$ )** : Kupacolja a tömböt a  $i$ . elemtől kezdve (11.2. algoritmus).
-



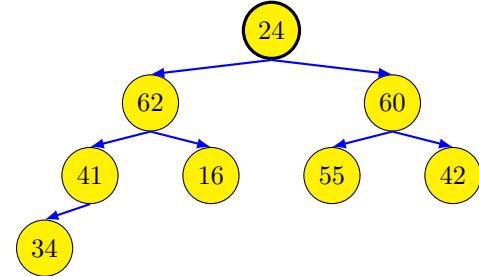
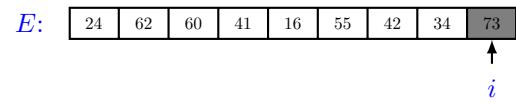
11.6. ábra. KupacRendezés osztály műveletei



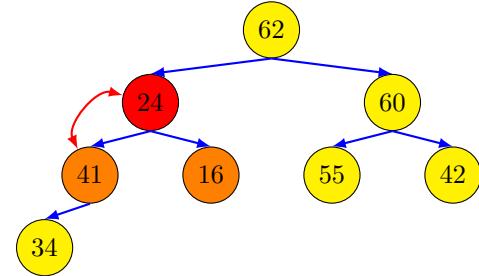
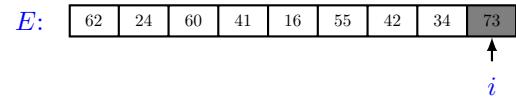
(a) Ezen a ponton állt le az előző ábrán bemutatott KUPACOTÉPÍT művelet. Ez láthatóan egy kupac, az *E* tömb pedig nem rendezett. Az *i* ciklus *n*-től indul. És egyből egy cserét végez.



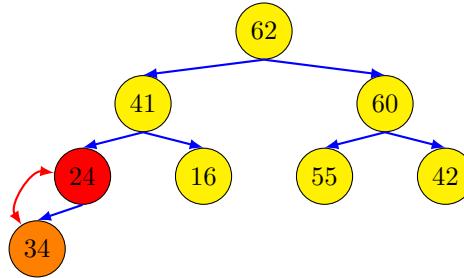
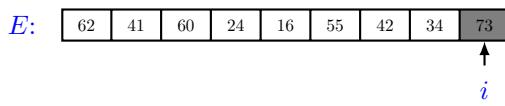
(c) Elindul a kupacolás a gyökérből.



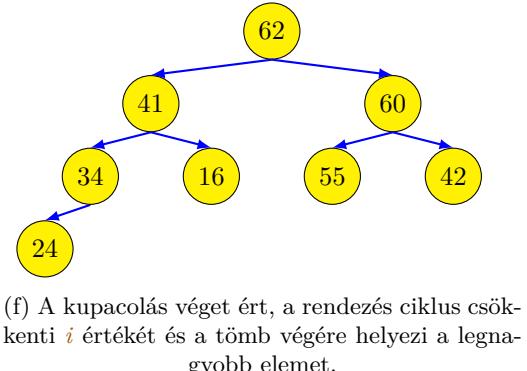
(b) A csere után csökkentjük *n* értékét, így az utolsó 73-as elem már nem része a kupacnak (hábor a tömbben még ott van).



(d) Második csere.

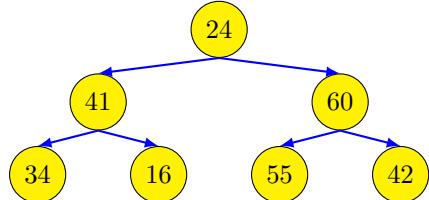
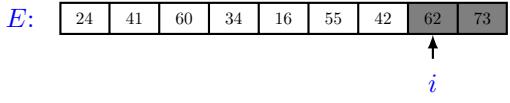


(e) Harmadik csere.

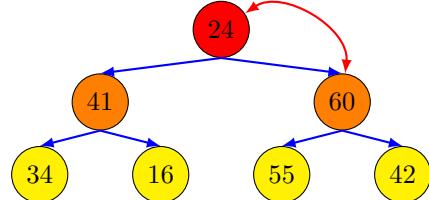
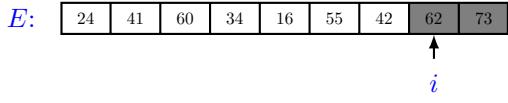


(f) A kupacolás véget ért, a rendezés ciklus csökkeneti *i* értékét és a tömb végére helyezi a legnagyobb elemet.

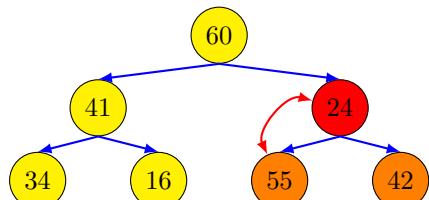
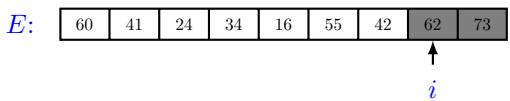
11.7. ábra. Kupac rendezés



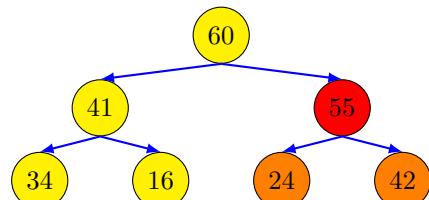
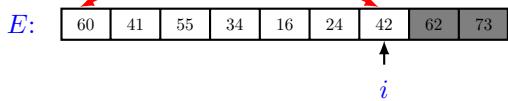
(a) A csere után csökken  $n$  értéke, így már két elem a végső helyére került, a kupac pedig ismét kisebb lett.



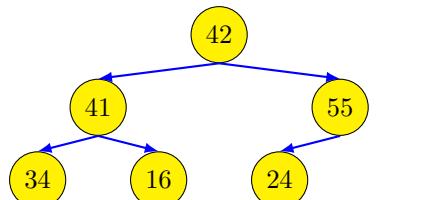
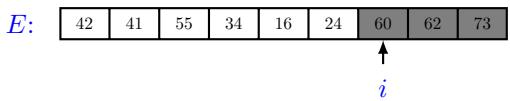
(b) Nem áll fent a kupac tulajdonság, ezért újabb kupacolás indul.



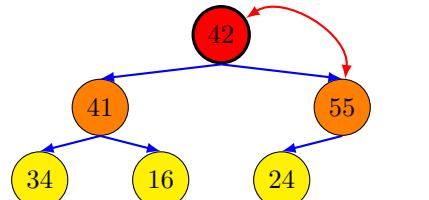
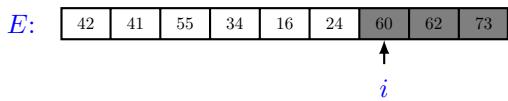
(c) A következő cserével ez is véget ért.



(d) A gyökérben van a legnagyobb elem, jöhet a rendezés következő lépése.

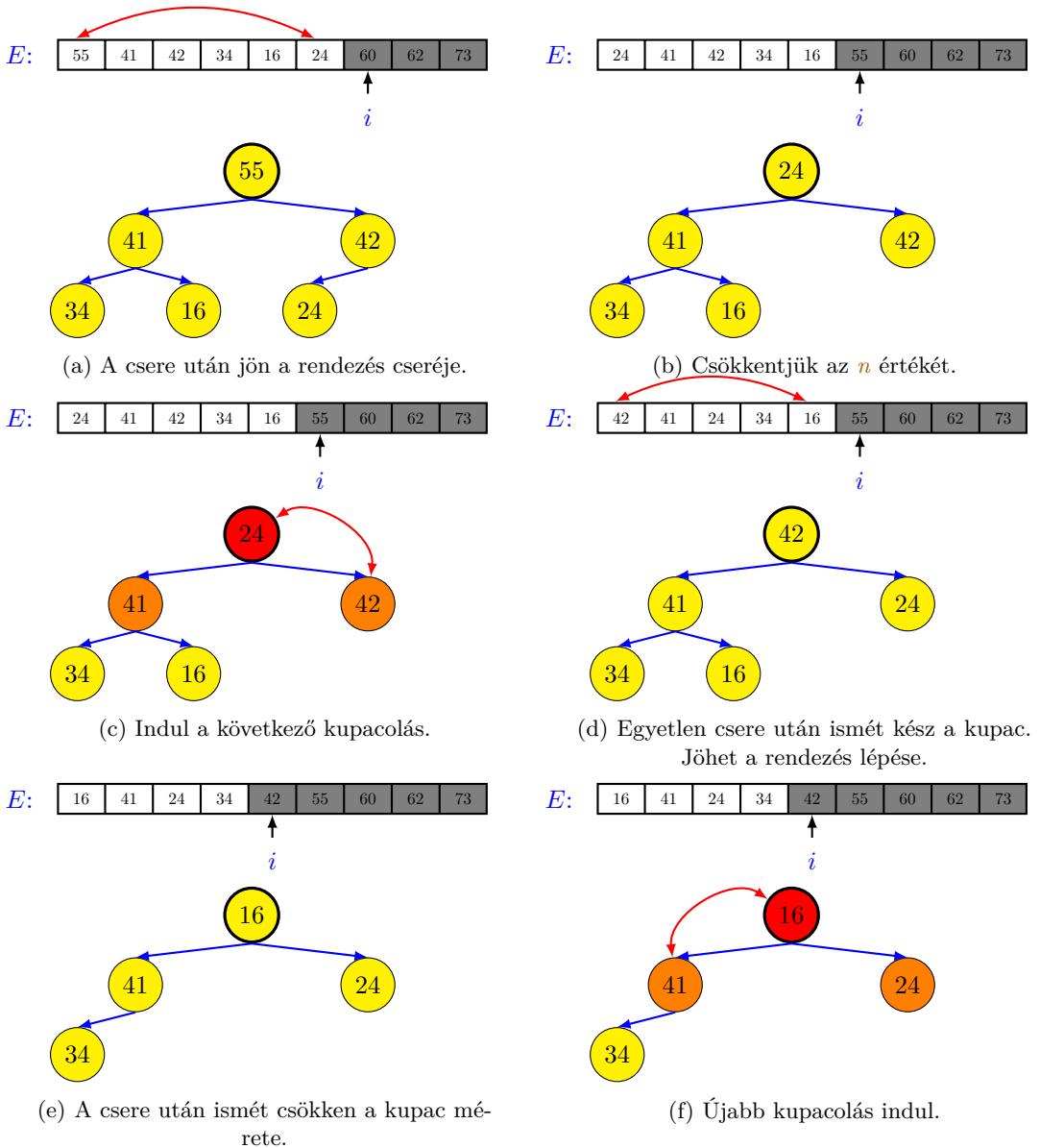


(e) A csere után ismét csökken a kupac mérete.

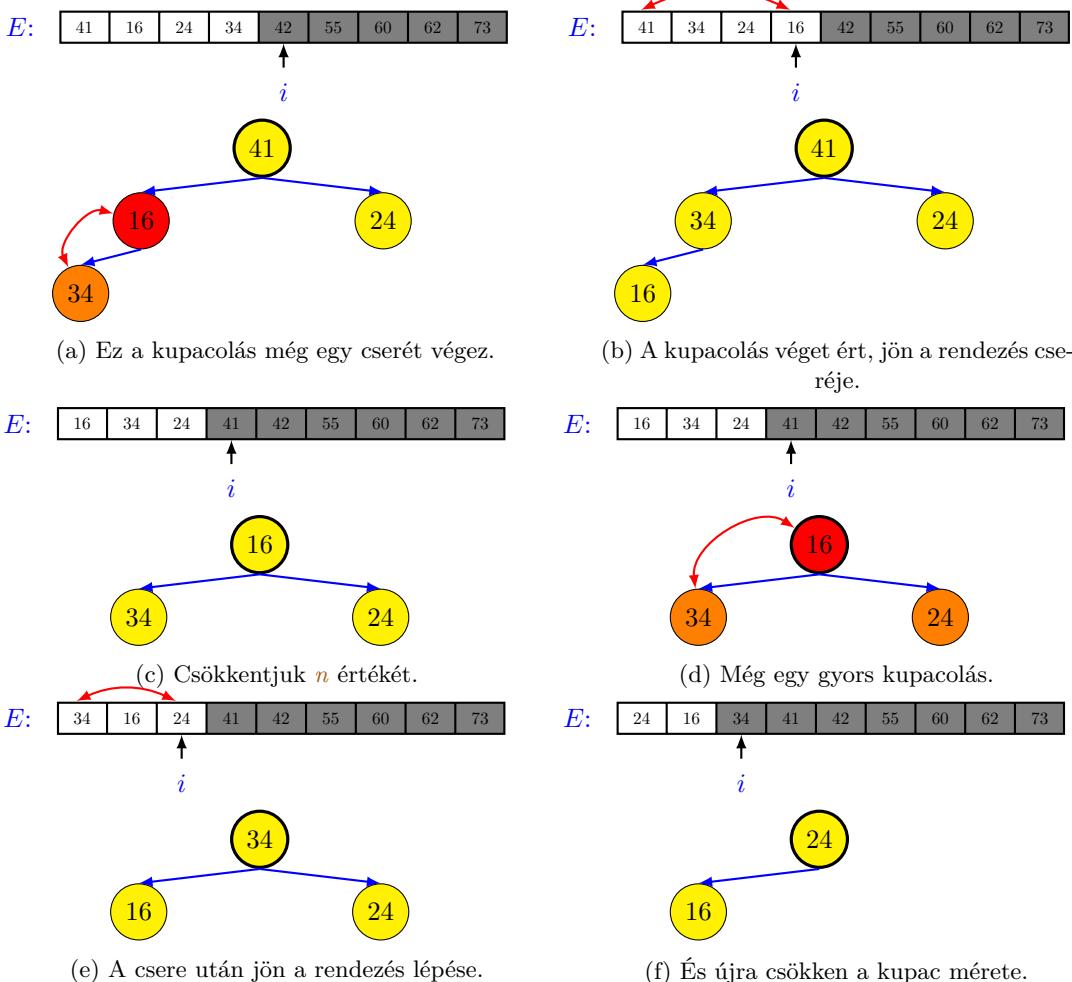


(f) Újabb kupacolás indul, ez csak egy cserét fog tenni.

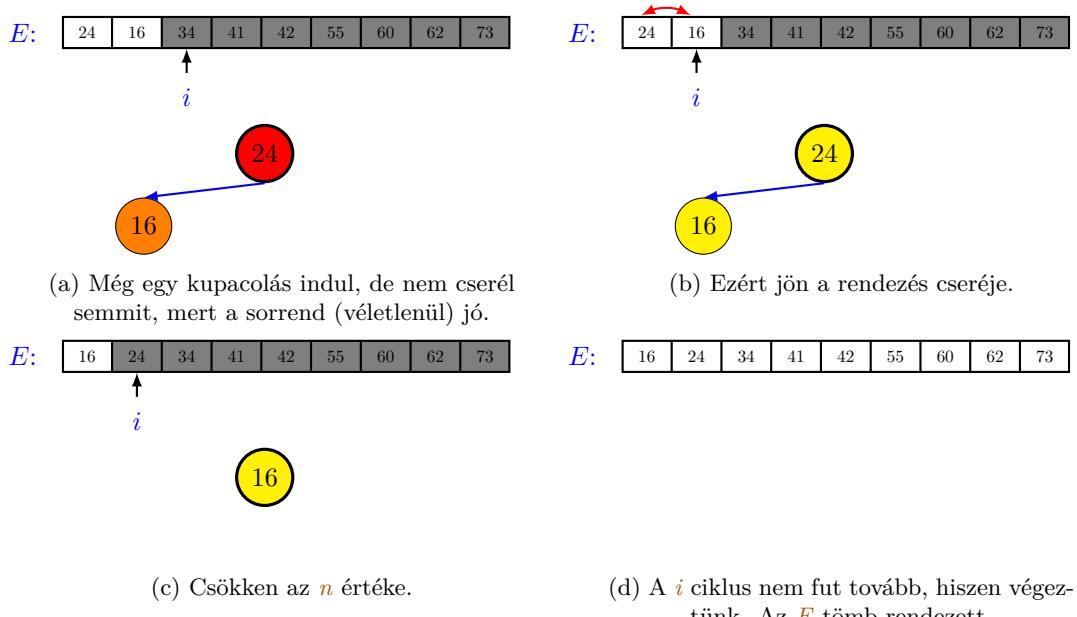
11.8. ábra. Kupac rendezés(folytatás)



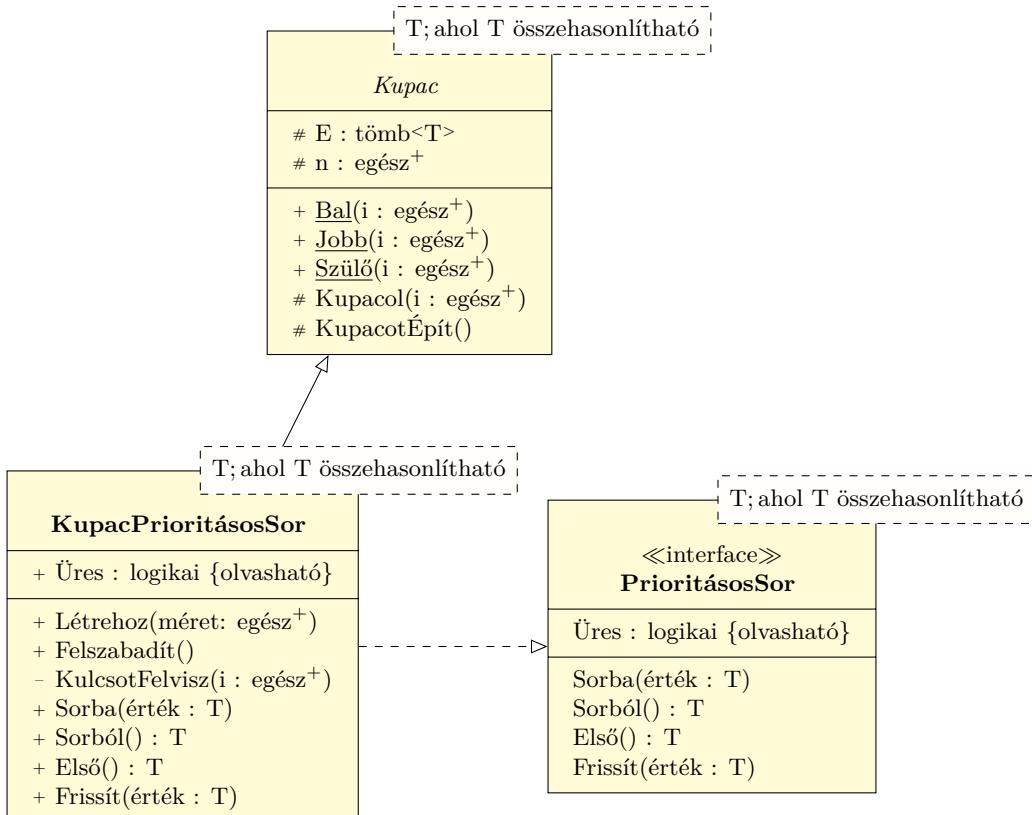
11.9. ábra. Kupac rendezés(folytatás)



11.10. ábra. Kupac rendezés(folytatás)



11.11. ábra. Kupac rendezés(folytatás)



11.12. ábra. A bináris kupac alapú prioritásos sor osztály műveletei

### 11.3. Prioritásos sor megvalósítása bináris kupaccal

A kupacok egy másik gyakori felhasználási területe a prioritási sorok készítése. A kupac tulajdonság alapján már beláttuk, hogy a kupac gyökere minden a legnagyobb elemet tartalmazza, ami egy prioritási sorban fontos, hiszen minden minden ez a következő kiolvasandó elem. Már csak a hatékony módosítás (új elem felvétele, kivétel, frissítés) műveleteket kell megvalósítanunk. A 11.12. ábra mutatja az így készítendő adatszerkezetet. Ez láthatóan megvalósítja a **PrioritásosSor** interfést, annak a metódusait fogjuk most megvalósítani. Működése során pedig sokban alapoz a már előzőleg elkészített **Kupac** absztrakt osztályra, ami készen ad számos praktikus kupac karbantartási műveletet.

Elsőként nézzük át a további magyarázatokat nem igénylő műveleteket:

- **Üres**: Az `n` mező mutatja a kupacban lévő elemek számát (ahol tudjuk, hogy  $n \leq E.méret$ ). Ez a vizsgálat tehát elég, ha azt nézi meg, hogy az `n` értéke nagyobb-e, mint 0.
- **LÉTREHOZ(méret)**: A rendezéssel ellentétben a konstruktornak itt nem egy már meglévő tömböt akarunk átadni, hanem ez a metódus fogja létrehozni a `méret` nagyságú `E` tömböt.
- **FELSZABADÍT()**: Mivel mi hoztuk létre az `E` tömböt, ezért a destrukturálásban ezt illik felszabadítani.
- **ELSŐ()**: Amennyiben nem üres a kupac, akkor tudjuk, hogy a gyökérelemnél (`E[1]`) nincs nagyobb eleme. Ezért visszaadhatjuk ennek az értékét, törlés itt még nem történik. Amennyiben a kupac üres, akkor egy kivétellel hibát jelzünk.

A többi művelet némi gondolkodást igényel, ezért ezeket részletesebben tárgyaljuk.

#### 11.3.1. Új elem felvétele

A **SORBA** művelet felvesz egy új elemet a kupacba. Mindezt persze úgy kell megtennünk, hogy ne sérüljön a kupac tulajdonság, tehát a kupac rendezett maradjon.

Ehhez elsőként bevezetünk egy segédműveletet, amely a fa aljáról felfelé tudja mozgatni az elemeket, amennyiben a rendezés ezt követeli meg. Tehát hasonló a már megismert **KUPACOL** művelethez, csak míg az a kupac tetejéről tudta lefelé vinni az elemeket, addig ez éppen fordítva, a kupac aljáról tudja felfelé vinni.

Működésének előfeltétele, hogy az  $i$ . elem és annak szülőjének (ha van) kapcsolatát kivéve minden elemre teljesüljön a kupac tulajdonság (tehát a  $i$ . elemre és annak gyerekeire is, ha azok egyáltalán vannak). A 11.5. algoritmus lépései:

1. A 2. sorban kiszámoljuk az  $i$ . elem szülőjének indexét ( $sz$ ).
2. A következő sorban ellenőrizzük, hogy létezik-e ez a szülő egyáltalán. Ha nem, akkor a fenti szabály szerint a teljes tömb kupac állapotban van, tehát nincs teendőnk.
3. Ha van szülő, akkor ellenőrizzük, hogy a szülő kulcsa és az  $i$ . elem kulcsa megfelelő sorrendben vannak-e. Ha a szülő kulcsa ( $E[sz]$ ) nagyobb, akkor megintcsak rendezett a kupacunk, így nincs tennivalónk.
4. Ha azonban van szülő, és annak kisebb a tartalma, mint a vizsgált elemnek, akkor a 4. sorban elvégezzük a két elem cseréjét. Ezzel lokálisan helyreáll a rendezettség.
5. Viszont a felvitt elem még mindig lehet, hogy megszegi a kupacrendezés szabályait, ezért rekurzívan meghívjuk az eljárást erre az elemre (5. sor). Könnyen belátható, hogy ennek a hívásnak az előfeltételei fennállnak (csak a feljebb vitt elem és annak új szülője sértheti meg a kupac tulajdonságát, minden máshol ennek teljesülnie kell).

A fenti lépéseket addig folytatjuk, ameddig nem áll helyre a  $i$ . elem és szülőjének rendezettsége, vagy eljutunk a gyökérbe, ahol nincs szülő.

---

### 11.5. Algoritmus Kupac tulajdonság fenntartása elem felvitelével

---

**Bemenet:**  $i$  - egész<sup>+</sup> (a vizsgálandó elem indexe)

- 1: **eljárás** KUPACPRIORITÁSSOR.KULCSOTFELVISZ( $i$ )
  - 2:      $sz \leftarrow SZÜLÖ(i)$
  - 3:     **ha**  $sz \geq 1 \wedge E[sz] < E[i]$  **akkor**
  - 4:          $E[sz] \leftrightarrow E[i]$
  - 5:         KULCSOTFELVISZ( $sz$ )
  - 6:     **elágazás vége**
  - 7: **eljárás vége**
- 

#### Felhasznált változók és függvények

- $sz$  : Egy egész<sup>+</sup> típusú segédváltozó.
  - $E$  : Az elemeket tartalmazó tömb.
  - **SZÜLÖ( $i$ )** : Az  $i$ . elem szülőjének indexe (11.1. algoritmus).
- 

Ez a segédművelet jól használható a kupacba új elemek felvételekor. Ezt ugyanis az alábbi lépésekkel tudjuk megtenni:

1. Fix méretű statikus tömbbel dolgozunk, ezért elsőként megvizsgáljuk, hogy elfér-e az új elem. Ha nem, akkor kivételek dobunk.
2. Ha igen, akkor növeljük az  $n$  értékét eggyel, és felvessük az új elemet az  $E[n]$  helyre. Tehát a kupac utolsó eleme ez lesz.
3. Ebben a pillanatban tudjuk, hogy a kupac első  $n-1$  eleme megfelel a kupactulajdonságnak, esetleg az utolsó elem lehet problémás. Gyerekei nincsenek, tehát ebbe az irányba is megfelel a rendezettségeknek, csak a szülővel lehet probléma. Viszont pont erre készítettük az előző segédfüggvényt, ezért ezt hívjuk meg.
4. Miután a **KULCSOTFELVISZ** lefutott, biztosak lehetünk benne, hogy az egész kupac rendezett.

---

## 11.6. Algoritmus Kupac alapú prioritásos sorba új elem felvétele

---

**Bemenet:** érték -  $T$  (a prioritásba sorba szúrandó érték)

- 1: eljárás KUPACPRIORITÁSSOR.SORBA(érték)
  - 2:   ha  $n < E.méret$  akkor
  - 3:      $n \leftarrow n + 1$
  - 4:      $E[n] \leftarrow$  érték
  - 5:     KULCSOTFELVISZ( $n$ )
  - 6:   különben
  - 7:     hiba "Nincs hely"
  - 8:   elágazás vége
  - 9: eljárás vége
- 

### Felhasznált változók és függvények

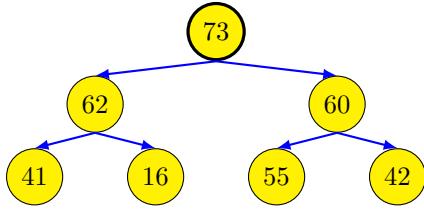
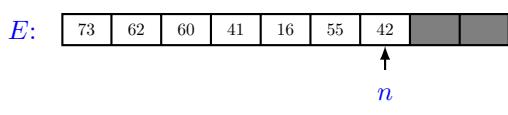
- $E$  : Az elemeket tartalmazó tömb.
  - $n$  : A kupacban lévő elemek száma ( $n \leq E.méret$ ).
  - KULCSOTFELVISZ( $i$ ) : A kupac tulajdonság fenntartása (11.5. algoritmus).
- 

### 11.3.2. Legnagyobb elem kivétele

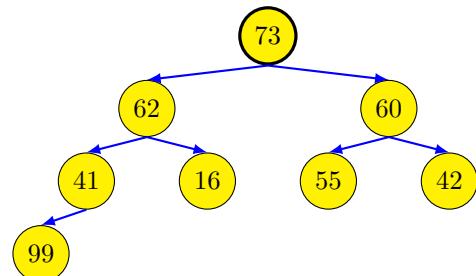
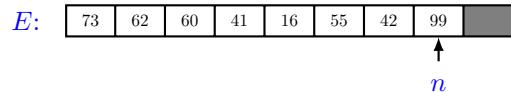
A SORBÓL művelet visszaadja a kupac gyökerében lévő elemet, hiszen tudjuk, hogy ennél biztosan nincs nagyobb az adatszerkezetben. Problémát csak az okoz, hogy törölünk is kell ezt az elemet úgy, hogy megmaradjon a rendezettség.

Ezt megoldhatjuk az alábbi lépésekkel:

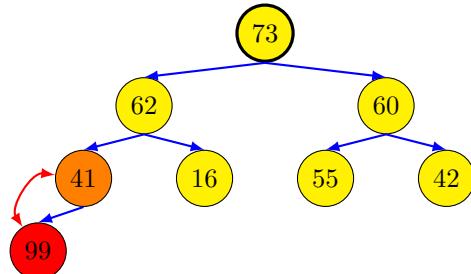
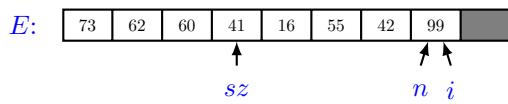
1. Elsőként ellenőrizzük, hogy van-e elem a prioritásos sorban. Ha nincs, akkor dobunk egy kivételt.
2. Ha van, akkor a gyökérelem lesz a visszatérési érték, hiszen az biztosan a(z egyik) maximális elem.
3. Törlés gyanánt a rendezésnél már megismertethet használjuk:
  - Felülírjuk a tömb első elemét a kupac utolsó elemével.
  - Csökkentjük a  $n$  értékét eggyel, tehát az utolsó elem már nincs kétszer a kupacban.
  - A gyökérbe másolt elemre nem biztos, hogy igaz a kupac tulajdonság. Viszont tudjuk, hogy minden más elemre ez igaz, emiatt erre meghívhatjuk a már megismert KUPACOL eljárást a gyökérből indítva, hiszen ott történt a változás.
4. Az eljárás végére töröltük a gyökérelemet a kupacból, az pedig újra rendezett. Visszatérési érték-ként pedig visszaadjuk az eddigi gyökeret.



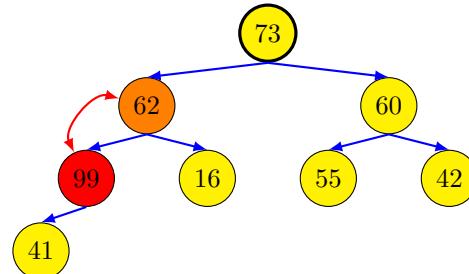
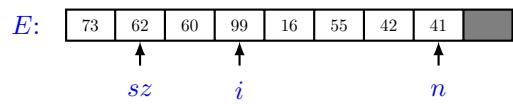
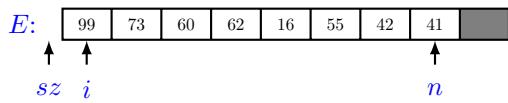
(a) Ismét ebből, a már ismert kupacból indukunk ki. Ez láthatóan megfelel a kupac tulajdonságainak.



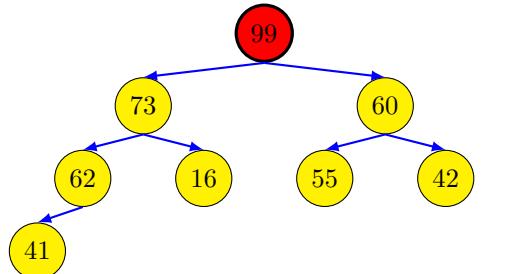
(b) Felvesszük egy új, 99 értékű elemet. Növeljük az  $n$  értékét és berakjuk a kupac végébe.



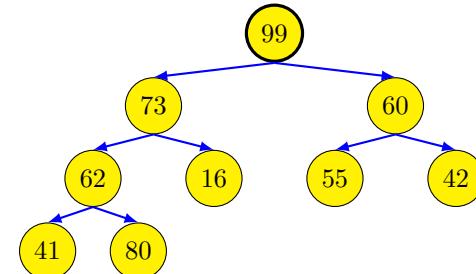
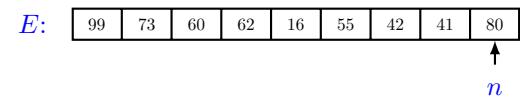
(c) Elindul az ellenőrzés, összehasonlítsa a szülővel, ezt cserélni kell.



(d) Újabb csere és rekurzív hívás következik.

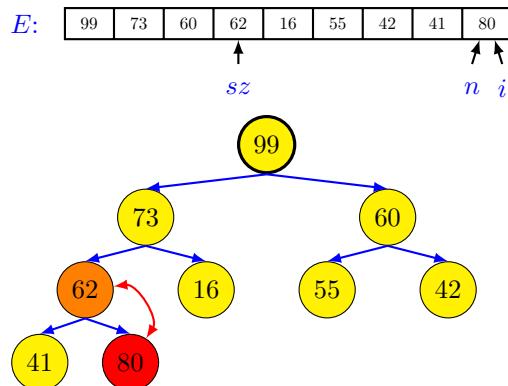


(e) A gyökérrel is cserálni kellett. Itt már nincs szülő, ezért megállunk

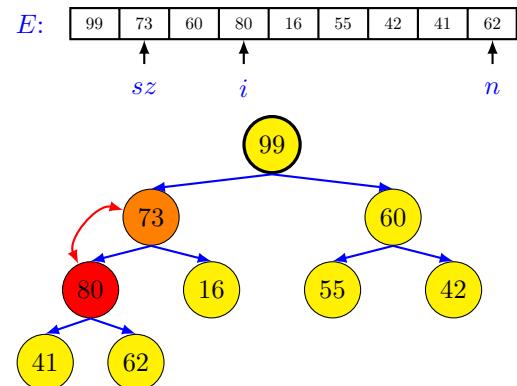


(f) Beszűrünk egy új, 80-as elemet a kupacba. Ami láthatóan nem rendezett.

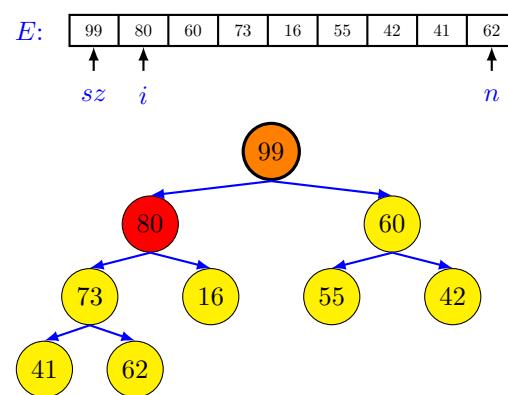
11.13. ábra. Kupacba új elem beszúrása



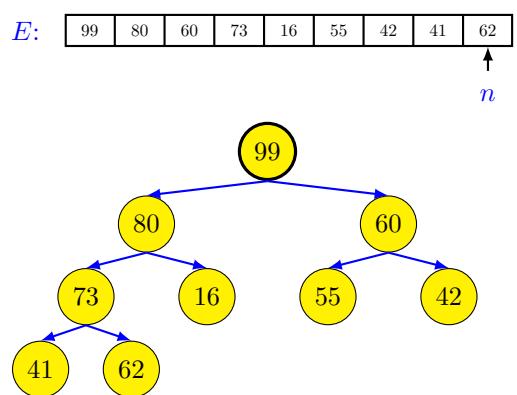
(a) Elindul a kulcsok felviteli  $i=9$  indextől.



(b) A következő rekurzív hívásnál is cserélni kell.



(c) Itt viszont már megáll a rekurzió, mivel a szülő nagyobb.



(d) Ezzel végeztünk is. A kupac rendezett és tartalmazza a két szükséges elemet.

11.14. ábra. Kupacba új elem beszúrása (folytatás)

---

## 11.7. Algoritmus Kupac alapú prioritásos sorból következő kivétele

---

**Kimenet:** . - **T** (*maximális érték*)

```
1: eljárás KUPACPRIORITÁSSOR.SORBÓL()
2: ha \neg Üres akkor
3: $max \leftarrow E[1]$
4: $E[1] \leftarrow E[n]$
5: $n \leftarrow n - 1$
6: KUPACOL(i)
7: vissza max
8: különben
9: hiba "Nincs elem"
10: elágazás vége
11: eljárás vége
```

---

**Felhasznált változók és függvények**

- **E** : Az elemeket tartalmazó tömb.
  - **n** : A kupacban lévő elemek száma ( $n \leq E.méret$ ).
  - **max** : Egy **T** típusú segédváltozó, ami eltárolja az első elem tartalmát.
  - **Üres** : Visszaadja, hogy üres-e a kupac.
  - **KUPACOL(*i*)** : Kupacolja a tömböt az *i*. elemtől kezdve (11.2. algoritmus).
- 

### 11.3.3. Prioritás növelése/csökkentése

Egy prioritási sort úgy is implementálhatunk (sőt, talán ez a gyakoribb változat), hogy fixnek tekintjük a benne lévő elemek prioritását. Tehát az egyszer már felvett elemek sorrendje a későbbiekben nem változhat.

Bizonyos algoritmusok viszont igényelhetik azt, hogy az egyszer már kupacban lévő elemek prioritását megváltozzassuk. Mivel a későbbiekben két ilyet is fogunk tanulni a gráfokkal kapcsolatban, ezért érdemes implementálni egy ilyen műveletet is. Ez lesz a **FRISSÍT(*elem*)** művelet, amit akkor hívunk meg, amikor megváltozik a paraméterként átadott *elem* prioritása, miközben az már egyszer be lett rakva a kupacba. Emiatt lokálisan, az *elem* környékén elképzelhető, hogy módosítanunk kell a kupacot.

A 11.8. algoritmus egy nem túlzottan elegáns, ámde rövid és egyszerű megoldást mutat be a probléma megoldására:

1. Az első ciklussal megkeressük az elemet a kupacban. Kihasználjuk azt, hogy a kupac valójában egy tömbben tárolja az elemeket, tehát egy egyszerű lineáris keresést futtatunk. Vegyük észre, hogy bár a kupac teljesíti a kupac tulajdonságot, de a keresés során ezt nem tudjuk kihasználni.
2. Ha nincs ilyen elem, akkor ezt egy kivétellel jelezzük
3. Ha az elem prioritása nagyobb lett, akkor elképzelhető, hogy cserélni kell majd őt a szülőjével. Ezt valósítja meg a 7. sorban lévő hívás. A **KULCSOTFELVISZ** rekurzív módon azt is garantálni fogja, hogy szükség esetén további cserékkal viszi fel a megváltozott prioritású elemet, akár a gyökérig. Ha az elem prioritása nem lett nagyobb mint a szülőé, akkor a függvény nem csinál semmit.
4. Ha az elem prioritása kisebb lett, mint valamelyik gyerekéé, akkor szintén változtatni kell a fán. Ilyenkor pedig használható a már megismert **KUPACOL** művelet meghívása az *i* gyökerű részfára (a 8. sor). Ha a gyerekekkel nem szükséges a csere, akkor a szükséges ellenőrzésekben túl ez se csinál semmit.
5. Miután ezekkel végeztünk, a kupac újra rendezett lesz.

### 11.8. Algoritmus A paraméterként átadott elem frissítése

**Bemenet:** érték - T (a vizsgálandó elem aktuális értéke)

- ```

1: eljárás KUPACPRIORITÁSSOR.FRISSÍT(érték)
2:   i ← 1
3:   ciklus amíg  $i \leq n \wedge \neg E[i] = \text{érték}$                                 ▷ Megkeressük a módosított elemet
4:     i ←  $i + 1$ 
5:   ciklus vége
6:   ha  $i \leq n$  akkor
7:     KULCSOTFELVISZ( $i$ )                                         ▷ Ha megvan, ellenőrizzük a környezetét
8:     KUPACOL( $i$ )
9:   különben
10:    hiba "Nincs elem"
11:  elágazás vége
12: eljárás vége

```

Felhasznált változók és függvények

- i : Egy **egész⁺** típusú segédváltozó.
 - E : Az elemeket tartalmazó tömb.
 - n : A kupacban lévő elemek száma ($n \leq E.\text{méret}$).
 - **KULCSOTFELVISZ(i)** : A kupac tulajdonság fenntartása (11.5. algoritmus).
 - **KUPACOL(i)** : Kupacolja a tömböt az i . elemtől kezdve (11.2. algoritmus).

11.4. Piros-fekete fa

11.4.1. Piros-fekete szabályok

A bináris keresőfa (6. fejezet) tárgyalásakor már volt róla szó, hogy az egy nagyon hatékony keresést biztosító adatszerkezet lenne, viszont ezt nagyban befolyásolja a fa kiegyensúlyozottsága. Ezügyben pedig kiszolgáltatottak voltunk az elemek beszűrási sorrendjének: ha szerencsénk volt, akkor egy kiegyensúlyozott fát kaptunk, ha pedig nem volt szerencsénk és pl. folyamatosan növekvő értékeket kellett felvenni a fába, akkor gyakorlatilag egy láncolt listához hasonló adatszerkezetet kaptunk. Ez volt az ára annak, hogy egy nagyon egyszerű beszűrási algoritmust használtunk.

A problémát megoldandó, léteznek különböző fa változatok, amelyek már bonyolultabb beszűrás/törlés algoritmusokat használnak, hogy valamelyest megpróbálják megőrizni a fa kiegyensúlyozottságát. Ezek között van olyan, amely minden kiegyensúlyozott fát ad (ilyen a 14. fejezetben bemutatott B-fa), viszont az felépítésében jelentősen különbözik a bináris keresőfától. Vannak azonban olyan bináris fákra vonatkozó kiegészítések, amelyek ugyan nem garantálják a minden kiegyensúlyozott állapotot, de valamilyen szinten törekednek erre. Ilyenek az AVL-fák, illetve a most tárgyalt piros-fekete fák.

A piros-fekete fa felépítése alapvetően megegyezik a már tárgyalt bináris keresőfával, mind az elemek felépítését, mind pedig a rendezettségét tekintve. Egyetlen kiegészítést tartalmaz, minden csúcspont esetén eltárolunk egy *szín* tulajdonságot, ami két értéket vehet fel (így egy biten is eltárolható): piros vagy fekete. A színezésnek a kereséshez és a rendezettséghez nincs köze, csak a fa kiegyensúlyozását fogja segíteni.

Önmagában ez az információ persze nem oldja meg a kiegyensúlyozást, ezért az alábbi szabályokat írjuk elő a piros-fekete fák számára:

1. A gyökér csík színe minden fekete.
2. minden piros színű csúcs minden gyereke fekete színű.
3. minden, a gyökérből a levelekbe vezető úton a fekete csúcsok száma azonos.

Megjegyzés

A szakirodalomban gyakran megjelenik feltételként az is, hogy minden levél fekete. Ez viszont abból adódik, hogy ezekben az esetekben a legalsó csúcsok alatt lévő \emptyset értékeket nevezik levélnek. Hogy elkerüljük ezeket az ellentmondásokat, mi maradunk az eddig is használt levél fogalomnál (tehát azok a csúcsok, akinek nincsenek gyerekeik).

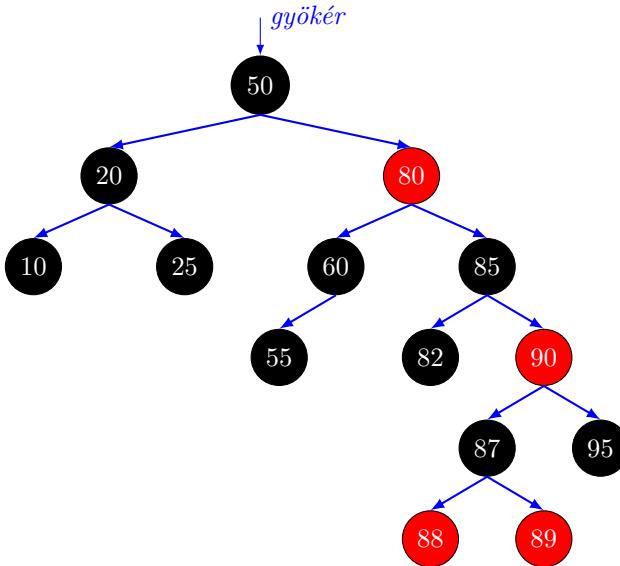
A piros-fekete fa adatszerkezet ugyan nem tudja garantálni a teljesen kiegyensúlyozott fák felépítését, viszont egy ehhez közel állapotot igen: biztosítja, hogy a leghosszabb gyökérből levélbe vezető út hossza sem lehet több, mint a legrövidebb ilyen út hossza. Ez könnyen belátható a fenti szabályokból, hiszen minden gyökérből levélbe vezető úton ugyanannyi fekete csúcsnak kell lennie (3. szabály). Legyen ez az érték k : ilyenkor a legrövidebb úton k darab fekete csúcsunk lesz; a leghosszabb úton pedig fekete és piros csúcsok fogják egymást váltogatni (mivel a 2. szabály miatt minden piros csúcsnak csak fekete gyerekei lehetnek), így a leghosszabb út legfeljebb $2k$ darab csúcsból állhat (11.15. ábra).

További apró eltérés a már megismert bináris keresőfához képest, hogy kényelmi okokból minden csúcsban tárolni fogunk egy szülőre való hivatkozást is (*szülö* mező), ami a gyökér elem esetében \emptyset érték lesz. De ez csak technikai különbség, magát az alapelvet nem érinti.

11.4.2. Forgatások

A 6. fejezetben megismert bináris keresőfa beszűrás algoritmus egyszerűsége abból fakadt, hogy az soha nem módosította a már meglévő fa szerkezetet, minden csak a már meglévő csúcsok közül valamelyik gyerekéknél, levélként vette fel az új csúcsokat. Ezt már láttuk, hogy szerencsétlen esetekben olyan fákhöz vezetett, amelyekben a későbbi keresés műveletek hatékonyisége is romlott.

Ezért bevezetünk két forgatás nevű műveletet (balra forgatás, illetve a hozzá nagyon hasonló jobbra forgatás), amelyek segítségével lokálisan meg tudjuk változtatni a fa szerkezetét. A 11.16. ábra mutatja a balra forgatás lépéseiit a 80-as csúcs körül. Maga a forgatás bármelyik bináris fában elvégezhető (még csak a rendezettség sem szükséges), mi most egy piros-fekete fa esetében fogjuk ezt áttekinteni.



11.15. ábra. Egy lehetséges piros-fekete fa. A gyökérből minden levélbe vezető úton 3 fekete csúcs van.

A művelet önmagában különösebb magyarázatot nem igényel, nézzük annak lépéseiit (11.9. algoritmus):

1. Legyen x az az elem, ami körül forgatni szeretnénk, y pedig ennek baloldali gyereke. A balra forgatáshoz szükséges, hogy ez az y egy létező csúcs legyen (11.16b. ábra).
2. Legyen x csúcs jobboldali gyereke az y baloldali gyereke. Ahogy a 11.16c. ábrán is látható, ez lehet egy valóban létező csúcs, ilyenkor annak a *szülő* hivatkozását is átállítjuk x -re. Lehetséges, hogy y -nak nincs bal gyereke, ilyenkor az x *jobb* mutatójába bekerül a \emptyset érték.
3. Az x elem helyére hozzuk fel a fában az y elemet. Ehhez elsőként be kell állítani, hogy y *szülő* hivatkozása legyen egyenlő x szülőjével (amennyiben x a gyökér volt, akkor egy \emptyset érték kerül ide). Ezt követően pedig ennek a szülőnek a gyerek hivatkozását kell ráirányítani, ami három esetet vizsgálatát igényli:
 - Ha az x a gyökér volt, akkor a *gyökér* mutatót kell az y csúcsra állítani.
 - Ha az x baloldali gyerek volt, akkor a szülője *bal* mutatóját kell átállítani y -ra.
 - Ha az x jobboldali gyerek volt, akkor a szülője *jobb* mutatóját kell átállítani y -ra.
4. A 11.16d. ábrán látható, hogy az x elem épp nincs bekötve a fába, ezért felvesszük őt y baloldali gyerekeként. Ehhez az x *szülő* mutatóját is aktualizálni kell (11.16e. ábra).

A forgatás végeredményén (11.16f. ábra) látható, hogy az átalakítások következtében nem veszítettünk el egy elemet se a fából, és persze új elem se jött létre. Az egyes átláncolásoknál egyesével belátható, hogy csak olyan műveleteket végeztünk el, amelyek garantáltan nem rontják el a fa rendezettségét. A piros-fekete tulajdonságról ez már nem mondható el, hiszen lett egy 4 fekete csúcsot tartalmazó utunk a meglévő 3 csúcsot tartalmazók mellett, de ezzel majd később foglalkozunk.

Vegyük azonban észre, hogy jelentősen meg változott a fa szerkezete. A forgatás előtt a 80-as csúcs jobboldalán látható részfa jóval mélyebb volt, mint a baloldalon látható részfa. A forgatás után ez a különbség viszont csökkent. Ez alapján már elkezdhetünk abban gondolkodni, hogy egy új elem beszúrása után megpróbáljuk úgy forgatni a fát, hogy az közelebb kerüljön a kiegensúlyozott állapothoz, miközben megtartja rendezettségét és a piros-fekete szabályokat se szegi meg. Valójában ez gyakran nem is egy, hanem több forgatást igényel, erről szól a következő beszúrás művelet.

Külön nem foglalkozunk a jobb irányba való forgatás műveletével, mivel az elvében teljesen meggyezik a bemutatott algoritmussal, csak a *bal* és a *jobb* irányokat kell felcserélnünk a pszeudokódban (11.10. algoritmus).

11.9. Algoritmus Bináris fa forgatása balra

Bemenet: x - $\tilde{\text{PFFaElem}}<\mathbf{T}>$ (a csúcs, ami körül forgatunk)

1: eljárás PIROSFEKETEFA.FORGATBALRA(x)

```
2:    $y \leftarrow x.jobb$ 
3:    $x.jobb \leftarrow y.bal$ 
4:   ha  $y.bal \neq \emptyset$  akkor
5:      $y.bal.szülő \leftarrow x$ 
6:   elágazás vége
7:    $y.szülő \leftarrow x.szülő$ 
8:   ha  $x.szülő = \emptyset$  akkor
9:     gyökér  $\leftarrow y$ 
10:  különben
11:    ha  $x = x.szülő.bal$  akkor
12:       $x.szülő.bal \leftarrow y$ 
13:    különben
14:       $x.szülő.jobb \leftarrow y$ 
15:    elágazás vége
16:  elágazás vége
17:   $y.bal \leftarrow x$ 
18:   $x.szülő \leftarrow y$ 
19: eljárás vége
```

Felhasznált változók és függvények

- *gyökér* : A fa gyökere.
 - *y* : $\tilde{\text{PFFaElem}}<\mathbf{T}>$ típusú segédváltozó.
-

11.10. Algoritmus Bináris fa forgatása jobbra

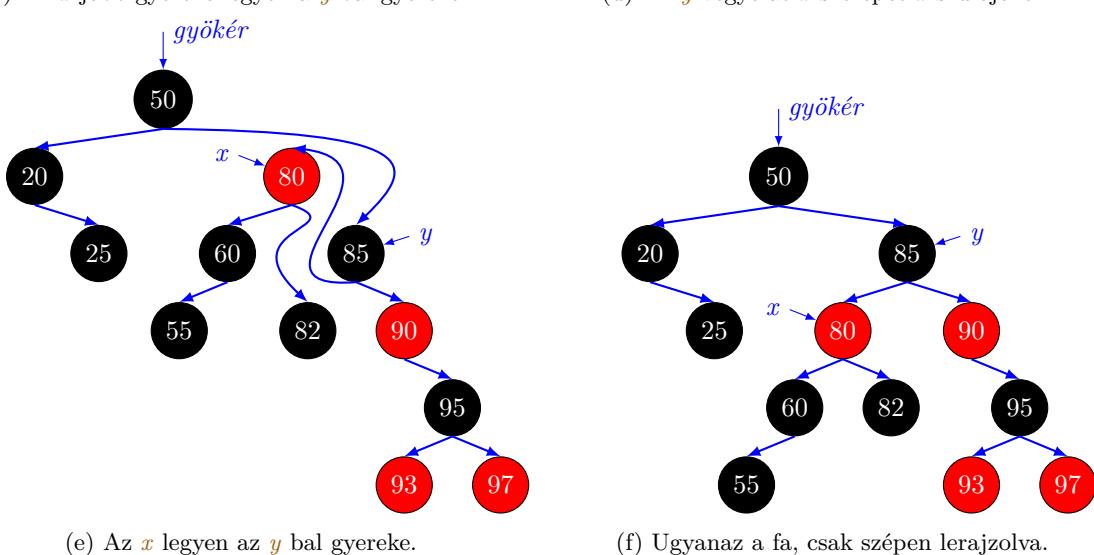
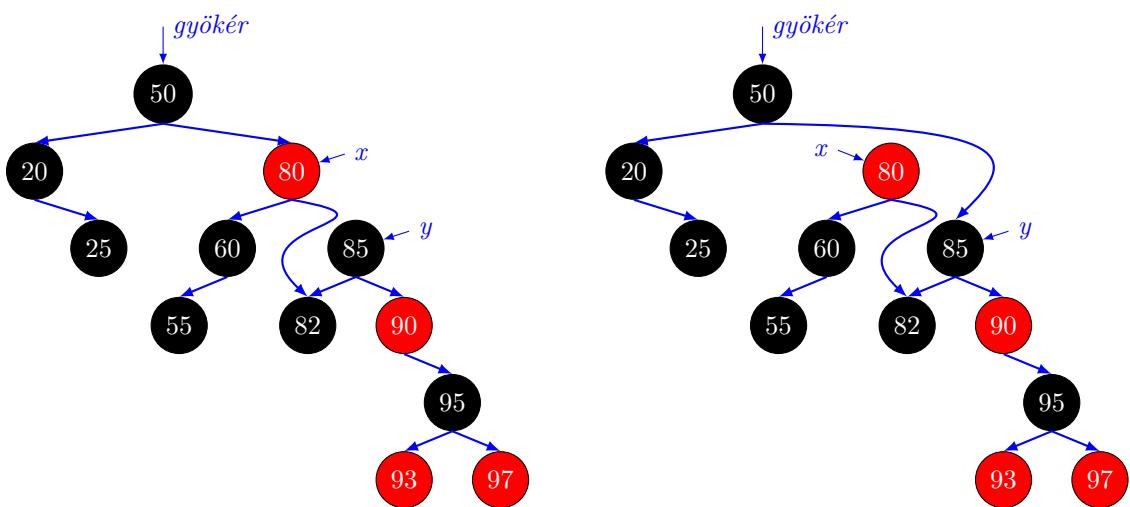
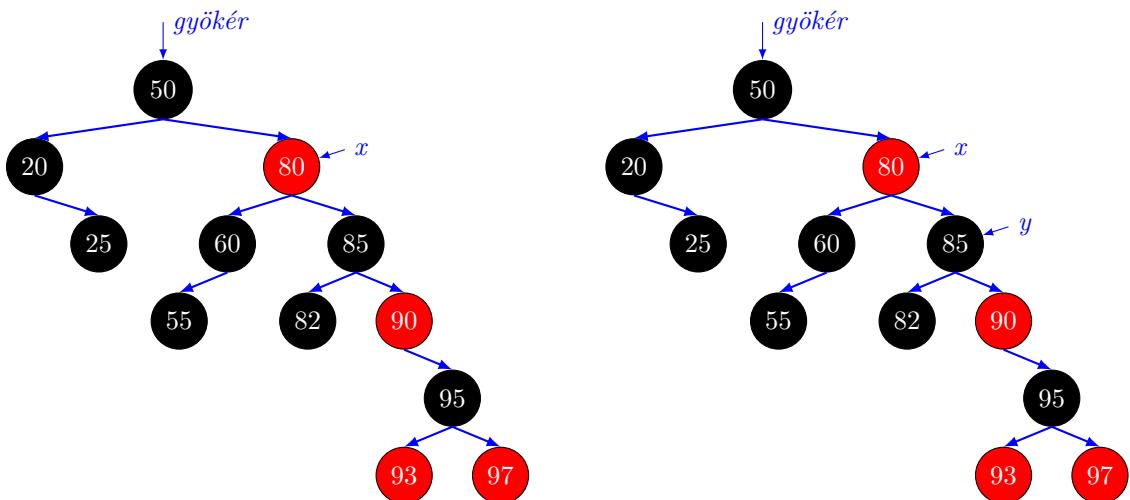
Bemenet: x - $\tilde{\text{PFFaElem}}<\mathbf{T}>$ (a csúcs, ami körül forgatunk)

1: eljárás PIROSFEKETEFA.FORGATJOBBRA(x)

```
2:    $y \leftarrow x.bal$ 
3:    $x.bal \leftarrow y.jobb$ 
4:   ha  $y.jobb \neq \emptyset$  akkor
5:      $y.jobb.szülő \leftarrow x$ 
6:   elágazás vége
7:    $y.szülő \leftarrow x.szülő$ 
8:   ha  $x.szülő = \emptyset$  akkor
9:     gyökér  $\leftarrow y$ 
10:  különben
11:    ha  $x = x.szülő.jobb$  akkor
12:       $x.szülő.jobb \leftarrow y$ 
13:    különben
14:       $x.szülő.bal \leftarrow y$ 
15:    elágazás vége
16:  elágazás vége
17:   $y.jobb \leftarrow x$ 
18:   $x.szülő \leftarrow y$ 
19: eljárás vége
```

Felhasznált változók és függvények

- *gyökér* : A fa gyökere.
 - *y* : $\tilde{\text{PFFaElem}}<\mathbf{T}>$ típusú segédváltozó.
-



11.16. ábra. Balra forgatás lépései.

11.4.3. A beszúrás művelete

A beszúrás első lépéseként fel kell vennünk az új elemet a fába. A művelet kezdetben nagyon hasonlít a 6. fejezetben már látott beszúrásra, az alábbi különbségeket vehetjük észre:

- Szembetűnő, bár igazából lényegtelen, hogy egy nem rekurzív formát használunk itt. Ennek oka, hogy a fa átalakítások miatt a rekurzió visszatérő ága már esetleg nem ugyanazt a fát látná, mint amin lefelé haladt.
- Mint említettük, kényelmi okokból egy *szülő* mutatót is felvettünk minden elemhez. Ez természetesen azt jelenti, hogy a beszúrás során ezt minden aktualizálnunk kell.
- A gyökér elem beszúrását kivéve meghívunk egy új **BESZÚRÁSJAVÍTÁS** nevű függvényt, ami elvégzi a piros-fekete szabályoknak való megfeleléshez szükséges átalakításokat. Ezeket a műveleteket később részletesen tárgyaljuk.

A 11.11. algoritmus remélhetőleg nem igényel különösebb magyarázatot, röviden a lépései:

1. Indulásként ellenőrizzük, hogy üres-e a fa. Ha igen, akkor létrehozunk egy fekete gyökérelementet, és ezzel végeztünk is. Ellenőrizhető, hogy minden szabályt teljesít az így létrejövő egyelemű fa.
2. Egyébként elkezdünk a *p* változó segítségével lefelé lépegetni a fában, hogy megkeressük az új elem helyét. Ehhez a már megszokott kereső lépéseket használjuk, csak most nem rekurzív formában. minden továbblépés előtt a *q* változóban tároljuk el azt az elemet, ahonnan továbbléptünk (tehát *q* minden a *p* szülőjére hivatkozik a lépés után).
3. Amennyiben a továbblépés során egy üres helyre lépnénk, akkor megtaláltuk az új elem helyét. Emiatt létrehozunk egy új elemet, és beláncoljuk a fába. Az új elem szülője *q* lesz, a *q* megfelelő gyerek mutatóját pedig erre az új elemre irányítjuk. Fontos, hogy az új elem színe mindenkor piros! Mivel ezzel a beszúrással már megszeghattuk a 2-es szabályt, ezért elindítjuk a beszúrás javítást végző eljárást.

A beszúrás során nem foglalkoztunkazzal az esettel, ha már meglévő kulcsot szúrunk be újra. Ilyenkor a beszúrás egyszerűen nem módosít semmit a fában.

Beszúrás utáni javítások

A beszúrást követően meg kell vizsgálnunk, hogy megszeghattuk-e valamelyik piros-fekete szabályt:

- A 1-es szabály csak egyetlen esetben lehet érdekes számunkra, amikor a legelső elemet vesszük fel a fába. Ilyenkor alapból feketére állítjuk a gyökeret, tehát ezzel teljesítjük a feltételt. minden más esetben a gyökér értéke változatlan, tehát ha a beszúrás előtt fekete volt a gyökér, akkor utána is az lesz.
- Ugyanígy az is igaz, hogy a 3-as szabályt se szeghattuk meg. Az első elem felvételekor egy darab fekete elemből áll a fa, tehát minden út hossza 1. minden más esetben pedig egy piros elemet veszünk fel, amiatt egyik úton se változhatott meg a fekete csomópontok száma. Így ha a beszúrás előtt érvényes volt a 3-as szabály, akkor biztos, hogy utána is az.
- A 2-es szabály viszont már sajnos további vizsgálódást igényel. Ezzel foglalkozunk a későbbiekbén.

Amikor elindul a beszúrás utáni javítás művelet, akkor a 11.17a. ábrán látható állapothoz hasonló helyzetet feltételezhetünk. A *p* hivatkozás mutat az újonnan beszúrt elemre, a *szülő* hivatkozás annak a szülőjére, a *nagyszülő* a szülő szülőjére, a *nagybácsi* pedig a szülő testvérére. Az ábrán látható színek is azt jelzik, hogy ezek színéről egyelőre nem tudunk semmit (sőt, még az se biztos, hogy van-e egyáltalán nagyszülő és nagybácsi elem), csak az a biztos, hogy az újonnan felvett elem piros, hiszen a beszúrást végző algoritmus minden pirosra állítja az új elemet. Az ábrán lévő felhők azt jelzik, hogy a fának csak egy kis részletét vizsgáljuk, a többi rész valójában nem lesz lényeges a beszúrás szempontjából.

Amikor a gyökér elemet szúrtuk be, akkor nem is indítottuk el a javítást végző eljárás. minden más elem beszúrásakor egy más meglévő elem gyerekeként vettük fel az új elemet, ennek következtében biztosan van *szülő* elemünk. Továbblépésként ennek a biztosan létező elemnek a színét vizsgáljuk. Érdemes

11.11. Algoritmus Piros-fekete fába beszúrás

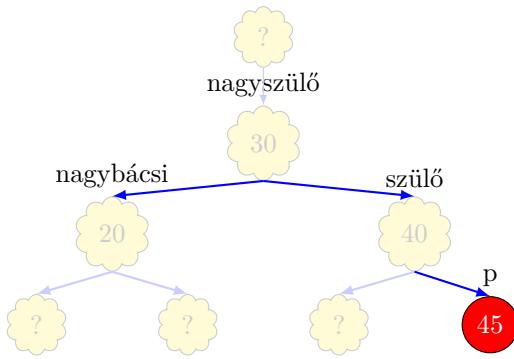
Bemenet: érték - **T** (*a* beszúrandó érték)

Megszorítás: **T** összehasonlítható

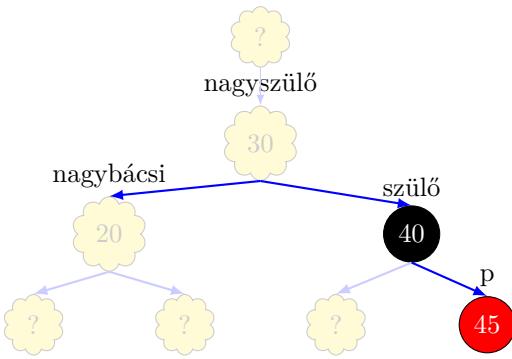
```
1: eljárás PIROSFEKETEFA.BESZÚR(érték)
2:   ha gyökér = ø akkor
3:     gyökér ← LÉTREHOZ(PFFaElem<T>) {tart ← érték, szín ← fekete, szülő ← ø}
4:   különben
5:     p ← gyökér
6:     ciklus amiíg p ≠ ø
7:       p ← p
8:       ha q.tart > érték akkor
9:         p ← q.bal
10:        ha p = ø akkor
11:          q.bal ← LÉTREHOZ(PFFaElem<T>) {tart ← érték, szín ← piros, szülő ← q}
12:          BESZÚRÁSJAVÍTÁS(q.bal)
13:        elágazás vége
14:      különben
15:        ha p.tart < érték akkor
16:          p ← p.jobb
17:          ha p = ø akkor
18:            p.jobb ← LÉTREHOZ(PFFaElem<T>) {tart ← érték, szín ← piros, szülő ← q}
19:            BESZÚRÁSJAVÍTÁS(q.jobb)
20:          elágazás vége
21:        elágazás vége
22:      elágazás vége
23:    ciklus vége
24:  elágazás vége
25: eljárás vége
```

Felhasznált változók és függvények

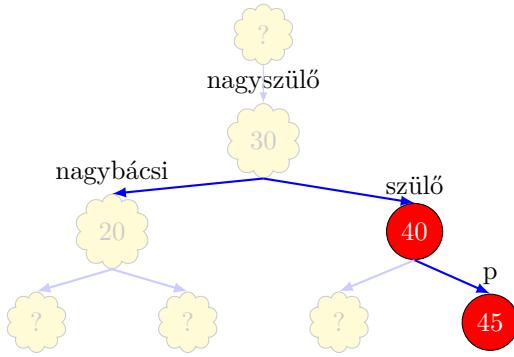
- *gyökér* : Hivatkozás a fa gyökerére, meg is változhat a metódus futása során.
 - *p, q* : *PFFaElem<T>* típusú segédváltozók.
 - *LÉTREHOZ(PFFaElem<T>)* : Létrehoz egy új piros-fekete fa csúcsot, és visszaad egy hivatkozást erre. Az új elem bal és jobb gyerek hivatkozásai legyenek ø értékkel inicializálva.
 - *BESZÚRÁSJAVÍTÁS(p)* : Szükség esetén a *p* pontból kiindulva átalakítja a fát, hogy az megfeleljen a piros-fekete szabályoknak.
-



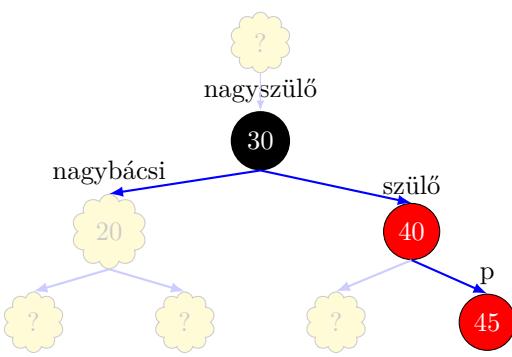
(a) Egy lehetséges beszúrás utáni állapot sok bizonytalansággal.



(b) Ha a szülő fekete, akkor nincs teendőnk.



(c) Ha a szülő piros, akkor és csak akkor viszont javítanunk kell.



(d) A javításra váró esetben biztosan van egy fekete nagyszülő.

megvizsgálni azt az esetet, amikor a szülő színe fekete (11.17b. ábra). Azt már beláttuk, hogy az 1-es és 3-as szabályt nem szeghettük meg, de ebben az esetben az is látható, hogy a 2-est sem. Mivel egy fekete elem alá szúrtunk be egy új elemet, ezért ez nem befolyásolja ennek a szabálynak a fennállását. Ebben az esetben tehát nincs további teendőnk.

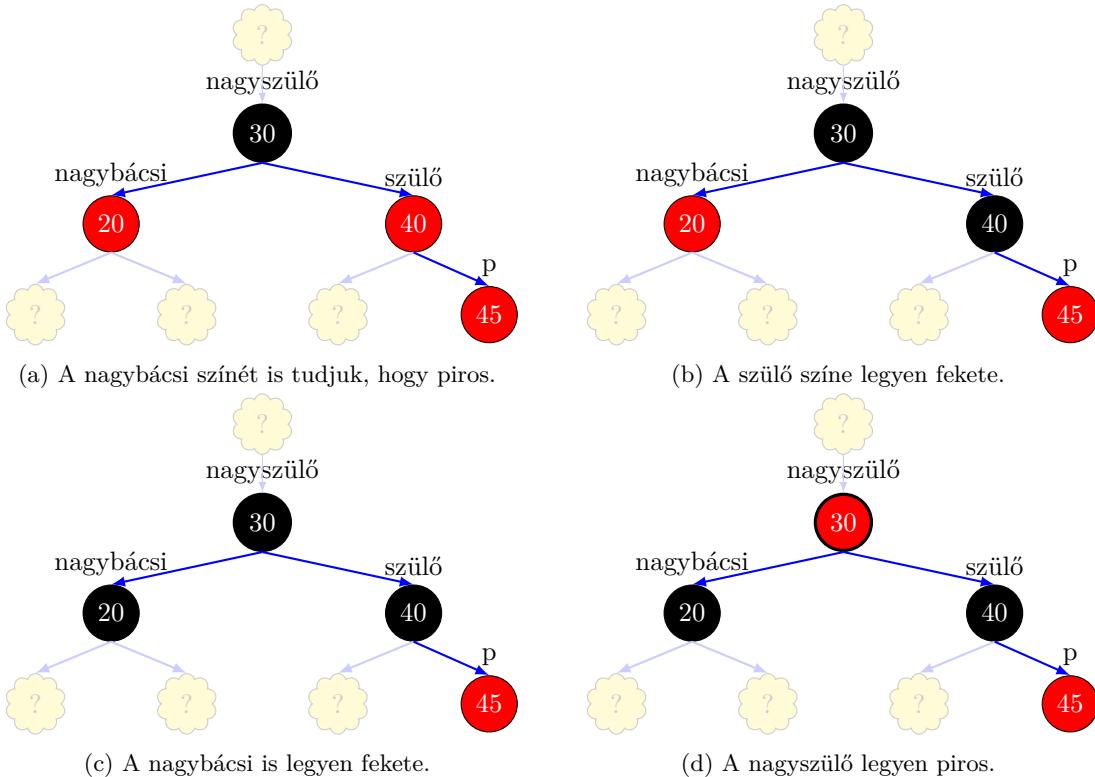
Izgalmasabb a kérdés, ha a *szülő* elem színe piros volt (11.17c. ábra). Itt már látható, hogy biztosan megszegtük a 2-es szabályt, tehát valamilyen formában javítanunk kell a fán. Ebben az esetben viszont még néhány dolgot ki tudunk következtetni a meglévő ismereteinkből: ha a *szülő* piros, akkor biztosan nem lehetett ő a gyökér, mivel az 1. szabály alapján alapján a gyökér minden fekete. Tehát biztosan léteznie kell a *nagyszülő* elemnek is. Sőt, az is tudható, hogy ez a *nagyszülő* csak fekete lehet, hiszen ha piros lenne, akkor nem lehetne piros gyereke (márpedig a *szülő* az piros). Ezekkel kiegészítve tehát elmondható, hogy a 11.17d. ábrához hasonló helyzetet van dolgunk. A helyzet persze csak „hasonló”, hiszen az új *p* elem lehet a *szülő* bal- illetve jobboldali gyereke, illetve maga a *szülő* is lehet a *nagyszülő* bal- és jobboldali gyereke. Tehát az ábrát négyféleképpen lehetne lerajzolni azonos színezés mellett.

Piros nagybácsi esete

Mielőtt külön-külön megvizsgálhnánk ezt a négy esetet, tekintsük át a *nagybácsi* szerepét. Amennyiben ez létezik, és ez a *nagybácsi* piros, akkor egy nagyon egyszerű átszínezéssel meg tudjuk oldani a problémát. Az eddigi ismereteinket kibővítvé ezzel az információval, a 11.18a. ábrán látható állapotból indulunk ki.

Ebben az esetben a fa szerkezeti átalakítása nélkül végre tudjuk hajtani az alábbi lépéseket:

1. A *szülő* színe legyen fekete (11.18b). Ő eddig piros volt, emiatt az öt érintő utakon növekedett a fekete csúcsok száma. Másképp megfogalmazva, a *nagyszülő* egyik gyerekének irányában haladó utakon eggyel nagyobb lett a fekete csúcsok száma.
2. A *nagybácsi* színe is legyen fekete (11.18c). Ő eddig piros volt, emiatt ezen az úton is eggyel növekedett a fekete csúcsok száma. Tehát a *nagyszülő* másik gyerekének irányában is eggyel több fekete csúcs van minden úton.



3. A *nagyszülő* színe legyen piros (11.18d). Mivel eddig ő fekete volt, ezért az összes rajta áthaladó úton az érintett fekete csúcsok száma eggyel csökkent. Ezzel pont ellensúlyoztuk az előző két növekedést.

Végeredményként tehát látható, hogy a teljes fában nézve az összes gyökérből levélbe vezető útra változatlan maradt a fekete csúcsok száma, tehát a 3. szabályt továbbra se szegtük meg. A színezés viszont helyre állt, hiszen a piros *nagyszülő* már valóban két fekete gyerekkel rendelkezik. Láthatóan a gyerek mutatóhoz nem nyúltunk, nem is forgattunk, tehát ha piros a *nagybácsi*, akkor mind a négy irányítottsági lehetőséget kezeltük.

Fontos azonban észrevenni, hogy ezzel nem biztos, hogy a teljes fára kijavítottuk a színezést, hiszen elképzelhető, hogy a *nagyszülő* szülője piros volt, ezért a feketéről átsínezésével ismét megszegtük a 2. szabályt. Ezért az algoritmus itt még nem állhat le, hanem folytatnuk kell a javítási műveletet úgy, hogy a *p* mutatót átállítjuk a *nagyszülő* elemre. Annyival biztos előrébb vagyunk, hogy feljebb tudtuk vinni a problémát, és belátható, hogy ennek többszöri ismétlése előbb-utóbb eljutunk a gyökér elemig (tehát amikor *nagyszülő* = *gyökér*). Az így pirosra váltott *gyökér* viszont már nyugodtan visszaszínezhető feketére, hiszen ezzel minden fekete út hossza eggyel lesz hosszabb.

Fekete vagy nem létező nagybácsi esete

Amennyiben a szabályt megszegő elemek a *nagybácsi* eleme fekete, vagy akár nincs is (ezt az állapotot jelzi az áttetsző fekete felhő a 11.19a ábrán), akkor az előző módszer sajnos nem működik. A fa legalsó szintjén ez az állapot ugyan nem állhat elő, de az előző módszernél láttnak, hogy a javítást egy ciklusban kell elvégezni, ami miatt felfelé lépegettünk a fában, így a *p* már nem biztos, hogy minden az új elemre mutat. Ilyenkor kénytelenek vagyunk a fent már említett négy (a *szülő* és a *p* elem bekötési irányától függő) esetet külön-külön kezelní.

- Elsőként vizsgáljuk meg azt az esetet, amikor a *szülő* a *nagyszülő* jobboldali gyereke, és a *p* is a *szülő* jobboldali gyereke. Ezt mutatja a 11.19a. ábra. Ilyenkor az alábbi lépésekkel megoldható a fa javítása:

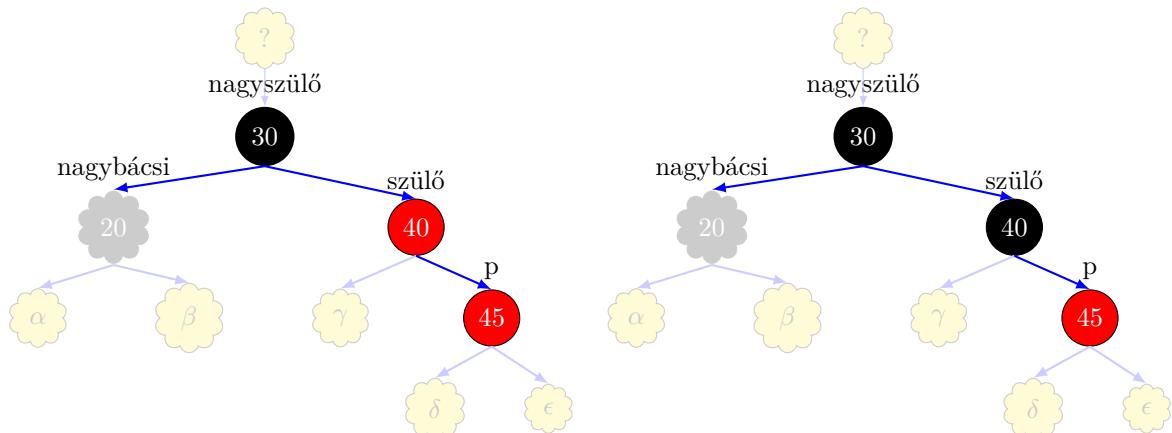
- A *szülő* színe legyen fekete (11.19b. ábra). Hasonlóan az előző megoldáshoz.

2. A *nagyszülő* színe legyen piros (11.19c. ábra). Ez a lépés is még ugyanaz.
 3. Ez az állapot még nem jó, hiszen a *nagyszülő* pirosra színezése miatt a *nagybácsi* irányába vezető utakon eggel kevesebb lett a fekete csúcsok száma. Ezt viszont egy *nagyszülő* körüli balra forgatással meg tudjuk oldani (11.19d. ábra). A *nagybácsi* felé vezető úton eggel több fekete csúcs lett, a *p* felé vezető úton pedig nem történt változás. A színek láthatóan rendben vannak, ezért ebben az esetben is végeztünk.
- Másodikként vizsgáljuk meg azt az esetet, amikor a *szülő* a *nagyszülő* jobboldali gyereke, a *p* viszont a *szülő* baloldali gyereke. Ezt mutatja a 11.20a. ábra. Ilyenkor az alábbi lépésekkel megoldható a fa javítása:
 1. Átléptetjük a *p* mutatót a szülőjére (11.20b. ábra).
 2. Elvégzünk egy *p* (tehát a korábbi *szülő*) körüli jobbra forgatást (11.20c. ábra).
 3. Vegyük észre, hogy pontosan ugyanabba az állapotba (színek, irányok) jutottunk, mint az 1. esetben. Tehát az ott lévő lépésekkel elvégezzük a teljes javítást. Tehát ezt az esetet is meg tudtuk oldani.
 - A harmadik eset az lenne, amikor a *szülő* a *nagyszülő* baloldali gyereke, és a *p* is a *szülő* baloldali gyereke. Ezt ugyanazokkal a lépésekkel oldjuk meg, mint az 1. esetet, csak értelemszerűen fel kell cserélni a *bal* és a *jobb* irányokat, illetve egy jobbra forgatásra lesz szükség a balra forgatás helyett.
 - A negyedik eset megoldása így már sejthető. Itt a *szülő* a *nagyszülő* baloldali gyereke, a *p* viszont a *szülő* jobboldali gyereke. Követjük a 2. lépést (majd az 1. lépést), csak felcseréljük az irányokat és forgatásokat.

A fenti lépéseket követve belátható, hogy minden esetre

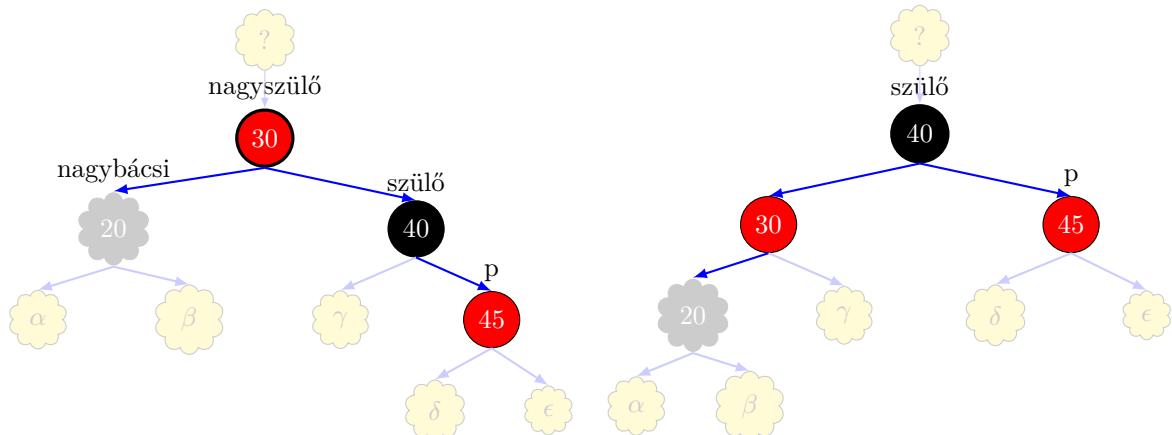
- beláttuk, hogy az nem okozhatja a piros-fekete tulajdonságok megsértését,
- vagy ha igen, akkor adtunk rá egy javítási módszert.

Ennek örömére vizsgáljuk meg a 11.12. algoritmust, ami pontosan a fent felsorolt lépéseket írja le pszeudokód formájában.



(a) Nincs nagybácsi, vagy ha van is, nem piros. A szülő és a p is bal gyerek.

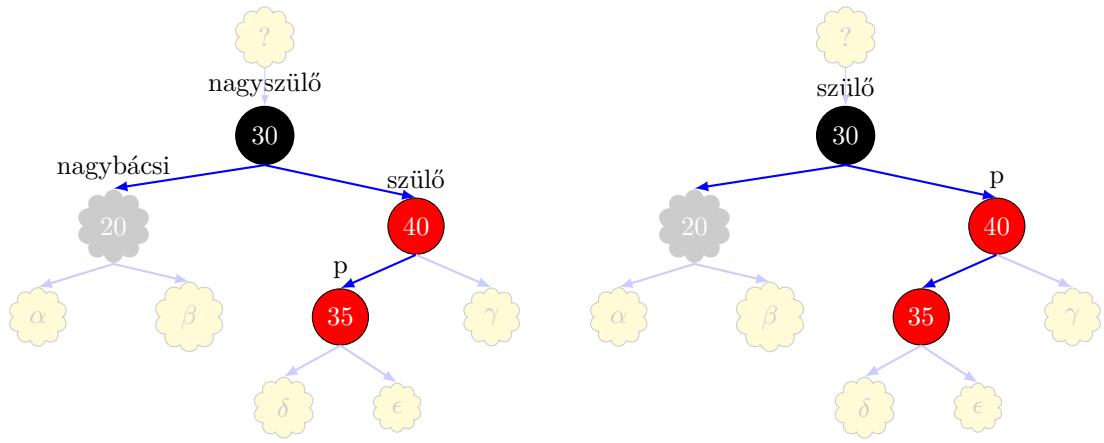
(b) A szülő színe legyen fekete.



(c) A nagyszülő színe legyen piros.

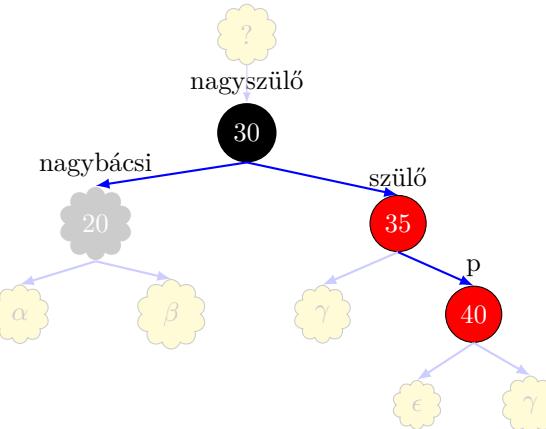
(d) A nagyszülő körüli balra forgatás eredménye. A p körül megváltozik a fa felépítése, ezért újra aktualizálni kell a hozzá igazodó mutatókat.

11.19. ábra. Fekete vagy nem létező nagybácsi esete, ha a szülőhöz képest a p ugyanolyan irányban van, mint a nagyszülőhöz képest a szülő



(a) Nincs nagybácsi, vagy ha van is, nem piros. A szülő bal gyerek, a p pedig jobb.

(b) A p átlép a szülőjére, így változik a többi hozzá viszonyított mutató is.



(c) A p (korábbi szülő) körüli jobbra forgatás. Elmozdul a p, így a hozzá viszonyított többi mutató is változik.

11.20. ábra. Fekete vagy nem létező nagybácsi esete, ha a szülőhöz képest a p ellenkező irányban van, mint a nagyszülőhöz képest a szülő

11.12. Algoritmus Piros-fekete fába beszúrás utáni javító művelet

Bemenet: x - $\tilde{\text{PFFaElem}}<\mathbf{T}>$ (a vizsgálandó/javítandó csúcs)

```
1: eljárás PIROSFEKETEFA.BESZÚRÁSJAVÍTÁS( $p$ )
2: ciklus amíg  $p.\text{szülő} \neq \emptyset \wedge p.\text{szülő}.szín = \text{piros}$ 
3:   ha  $p.\text{szülő} = p.\text{szülő}.szülő.\text{bal}$  akkor
4:      $nagybácsi \leftarrow p.\text{szülő}.szülő.\text{jobb}$ 
5:   különben
6:      $nagybácsi \leftarrow p.\text{szülő}.szülő.\text{bal}$ 
7:   elágazás vége
8:   ha  $nagybácsi \neq \emptyset \wedge nagybácsi.szín = \text{piros}$  akkor
9:      $p.\text{szülő}.szín \leftarrow \text{fekete}$ 
10:     $nagybácsi.szín \leftarrow \text{fekete}$ 
11:     $p.\text{szülő}.szülő.szín \leftarrow \text{piros}$ 
12:     $p \leftarrow p.\text{szülő}.szülő$ 
13:   különben
14:     ha  $p.\text{szülő} = p.\text{szülő}.szülő.\text{bal}$  akkor
15:       ha  $p = p.\text{szülő}.\text{jobb}$  akkor
16:          $p \leftarrow p.\text{szülő}$ 
17:         FORGATBALRA( $p$ )
18:       elágazás vége
19:        $p.\text{szülő}.szín \leftarrow \text{fekete}$ 
20:        $nagybácsi.szín \leftarrow \text{fekete}$ 
21:       FORGATJOBBRA( $p.\text{szülő}.szülő$ )
22:     különben
23:       ha  $p = p.\text{szülő}.\text{bal}$  akkor
24:          $p \leftarrow p.\text{szülő}$ 
25:         FORGATJOBBRA( $p$ )
26:       elágazás vége
27:        $p.\text{szülő}.szín \leftarrow \text{fekete}$ 
28:        $nagybácsi.szín \leftarrow \text{fekete}$ 
29:       FORGATBALRA( $p.\text{szülő}.szülő$ )
30:     elágazás vége
31:   elágazás vége
32: ciklus vége
33: gyökér.szín  $\leftarrow \text{fekete}$ 
34: eljárás vége
```

Felhasznált változók és függvények

- *gyökér* : A fa gyökere.
 - p , *nagybácsi* : $\tilde{\text{PFFaElem}}<\mathbf{T}>$ típusú segédváltozók.
 - **FORGATBALRA(p)**, **FORGATJOBBRA(p)** : Az előzőleg tárgyalt p pont körüli balra és jobbra forgató műveletek.
-

12. fejezet

Súlyozatlan gráf

12.1. Súlyozatlan gráf felépítése

12.1.1. Alapvető fogalmak

Jelen jegyzetnek nem célja az alapvető gráfelméleti fogalmak bevezetése, az egységes jelölésrendszer érdekében viszont érdemes áttekinteni az alapokat.

Egy $\text{g} = \text{Gráf}\langle V, E \rangle$ gráf csomópontok (V) és rajtuk értelmezett élek (E) halmaza, ahol E egy V -n értelmezett bináris reláció ($E \subseteq V \times V$).

A gyakorlatban megkülönböztetjük az alábbi fajtákat:

- irányított vagy irányíthatlan gráfok;
- súlyozott, vagy súlyozatlan gráfok.

Ha külön nem hangsúlyozzuk, akkor a bemutatott algoritmusok egyaránt működnek irányított és irányíthatlan gráfokra is, jelen fejezetben pedig csak a súlyozatlan változattal foglalkozunk. A súlyozott gráfok felépítése alapvetően nem sokkal bonyolultabb, viszont ezekkel jóval többféle összetett feladatot tudunk megoldani, ezért ezek egy külön fejezetben kaptak helyet.

A jegyzeten belül a csúcsokra felváltva használjuk a csúcs, csúcspont, csomópont, pont szavakat.

12.1.2. Egy elem felépítése

Az eddigi módszertant követve a következő lépés a gráf egy elemének vizsgálata lenne, ami az eddigi adatszerkezetek esetében egy osztály felépítését vonta maga után. Jelen esetben azonban ezt egyszerűbben fogjuk kezelní.

- A gráf csúcsai V típusú változók. Ezeket a legegyszerűbb egy azonosítóként elközelni, amelynek típusát maga az implementáció határozza meg. Ez lehet pl. egy egész szám.
- A gráf éleit pedig E típusú változók képviselik a pszeudokódokban, amelyek típusát szintén az implementáció határozza meg. Lehet pl. egy egész számokból álló számpár.

Az előzőleg vizsgált adatszerkezetekhez képest lényeges különbség, hogy gráfok esetében nem az egyes elemek tartalmával kapcsolatos műveletekre fókuszálunk (tárolás, visszakeresés, stb.), hanem az elemek közötti kapcsolatokra. Ugyan van rá mód, de a későbbiekben emiatt nem is foglalkozunk azokkal a kérdésekkel, hogy pl. milyen típusú adatokat tárolunk el a gráfban, hogy tudunk ezek között hatékonyan keresni, stb. A gyakorlatban persze meg kell valósítani a gráf csomópontjainak és a köztük lévő kapcsolatoknak a karbantartását végző műveleteket, azonban az igazán érdekesek nem ezek lesznek, hanem pl. összefüggő komponensek keresése, (legrövidebb) út keresése csomópontok között, stb.

12.2. Tárolási módok

12.2.1. Dinamikus tárolás

Ha nyomon követjük az eddig megismert adatszerkezeteket, akkor jól látható a hasonlóság közöttük. A láncolt lista minden eleme eltárolt egy tartalmat, illetve egy darab referenciát a következő elemre. A ciklikus láncolt lista kivételével ez a hivatkozás nem mutathatott egy előző elemre. A fák esetében technikailag csak annyi különbséget láttunk (bár már ez is jelentősen megváltoztatta a műveleteinket), hogy minden elemnek lehet több „következő” eleme is, a bináris keresőfa esetében legfeljebb kettő, de pl. a B-fánál látni fogjuk, hogy a gyerekek száma lehet ennél több is. A fánál azonban még mindig tudtuk, hogy nincsenek visszairányuló következő mutatók, tehát nem lehet egy elem bármelyik leszármazottja az ō őse is egyben, hiszen ezzel a legtöbb algoritmusunk végétlen ciklusba került volna.

Ha követjük az eddigi gondolatmenetet, akkor a gráfot elképzelhetjük egy olyan dinamikus adatszerkezetként, ahol minden egyes elemnek tetszőleges számú mutatója lehet, amelyek a szomszédaira hivatkoznak. Ezeket a hivatkozásokat eltárolhatjuk minden egyes csúcsban egy tömbben, ez azonban csak akkor valósítható meg, ha tudjuk az egy csúcsból kiinduló élek maximális számát. Amennyiben ez nincs így (mivel két csúcspont között több él is vezethet, így még a csúcsok száma se mindig kielégítő korlát), akkor esetleg egy listában is eltárolhatjuk ezeket a hivatkozásokat.

További esetleges módosítások:

- Irányított/irányítatlan gráfok: mivel a programozási nyelvekben lévő mutató típusok minden csak egyirányúak, így ezzel a módszerrel csak irányított gráfokat tudunk modellezni. Irányítatlan esetben minden él két mutatót jelent, amelyek a két irányt jelképezik.
- Súlyozott gráfok: magában a gráf csúcsban is tárolhatjuk, hogy az egyes élek milyen súlyúak, vagy akár magát az élt is elképzelhetjük külön objektumként, aminek van súly tulajdonsága.

Az így elkészült adatszerkezet szépségét némileg beárnyékolja, hogy a gráf esetén nem elégedhetünk meg egy darab „fejszerű” külső hivatkozással, hiszen a későbbi bejárásainkat bármelyik csúcsból el szeretnénk tudni indítani. Emiatt minden csúcsra célszerű felvenni egy-egy külső referenciát (amiket tárolhatunk akár egy szótárban is).

A módszer előnye, hogy dinamikus volta miatt optimális a helyfoglalása. Érdemes azonban megismerni két, a gyakorlatban elterjedt további megoldást.

12.2.2. Tárolás csúcsmátrixban

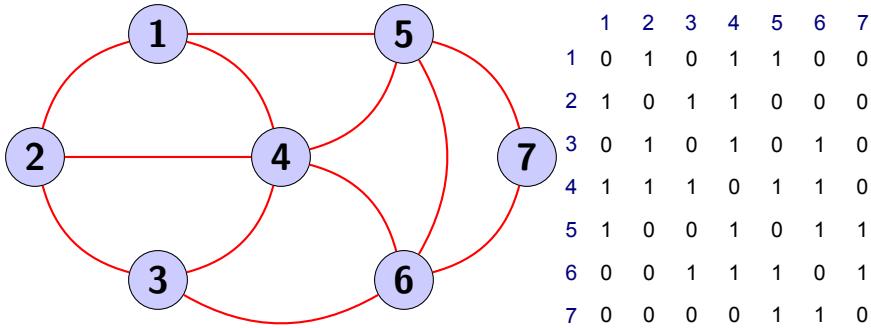
Amennyiben megengedjük azt a korlátot, hogy a gráf csúcsainak számát fixnek tekintjük, akkor jóval egyszerűbben kezelhető módszerekhez jutunk. A csúcsmátrixban való tárolás talán a legegyszerűbb tárolási módszer, még ha bizonyos esetekben nem is tekinthető hatékonynak. A gráf tárolásának alapelve, hogy létrehozunk egy kétdimenziós M nevű mátrixot, amely sorainak és oszlopainak a száma megegyezik a gráf csúcspontjainak a számával ($|V|$) (és minden gráf csúcsot hozzárendelünk egy-egy sorhoz, illetve oszlophoz).

Ezt követően az $M[i,j]$ azt határozza meg, hogy az i . csúcsból vezet-e él a j . csúcsba. A mátrix tartalmazhat *igaz*/*hamis* értékeket (vezet-e él), vagy akár természetes számokat is (hány él vezet, 0 ha nincs, 1 vagy annál nagyobb, ha van).

A 12.1. ábra egy példát mutat erre a tárolási módra. Látható, hogy pl. a $M[2,3]$ helyen lévő 1-es azt mutatja, hogy van él, ami a 2-es csúcsból vezet a 3-as csúcsba.

További esetleges módosítások:

- Irányított/irányítatlan gráfok: minden fajta gráfot egyszerűen ábrázolhatjuk. Irányítatlan gráfok esetében nyilvánvalóan minden $M[i,j]$ és $M[j,i]$ érték azonos lesz, így nem is szükséges eltárolni a teljes mátrixot, elég csak a felső háromszög mátrixot (ezzel a tárigény csaknem a felére csökken).
- Súlyozott gráfok: egy súlyozott gráfot úgy is tárolhatunk, hogy $M[i,j]$ megadja, hogy az i . csúcsból a j . csúcsba vezető élnek mennyi a súlya. Ebben az esetben valamilyen speciális módon jelölünk kell azt is, ha nem vezet él (ez alapos megfontolást igényel, hiszen pl. a 0 választása nem feltétlenül szerencsés, ha szeretnénk 0 súlyú élekkel is dolgozni). Illetve két csúcs között több él kezelése se oldható így meg.



12.1. ábra. Irányítatlan, súlyozatlan gráf és az azt reprezentáló csúcsmátrix.

Megtartjuk a generikus jelölést, de valójában a típusok itt már meglehetősen konkrétek. A **V** tulajdonképpen megfelel az **egész⁺** típusnak, azzal a korlátta, hogy értéke csak 1 és $|V|$ közötti lehet. Az **E** egy csúcspárt tárol, ami többféleképpen implementálható, lehet pl. egy **egész⁺** elemeket tartalmazó kételemű tömb, esetleg egy saját osztály. Vegyük észre, hogy a gráf valójában nem tárol **E** típusú adatokat, ezt a típust csak paraméterként/visszatérési értékként használjuk.

A gráf interfészben meghatározott alapműveletek megvalósításai során irányított súlyozatlan gráfot veszünk alapul, ahol két csúcs között maximum egy él lehet. Ennek megfelelően a tárolást egy logikai tömbbel meg tudjuk valósítani. Csúcs azonosítóként egész számokat használunk 1 és $|V|$ között.

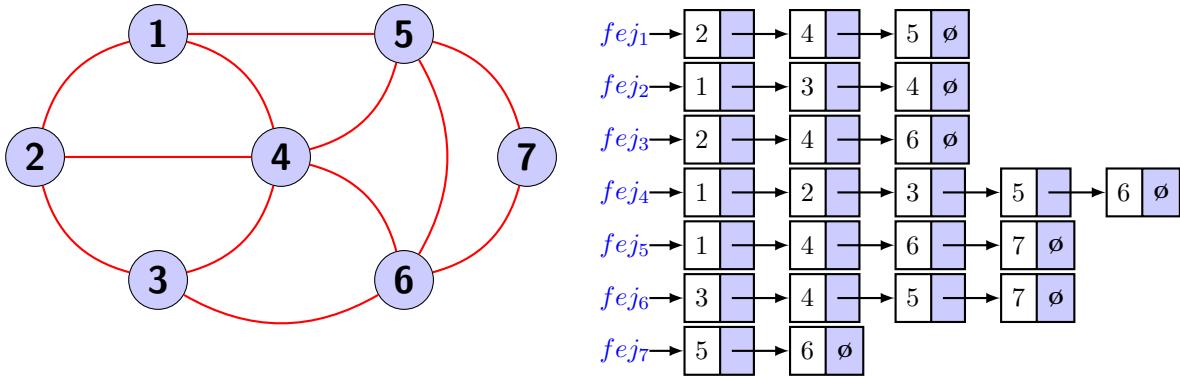
- **ÚJÉL(honnan : V, hova : V)**: Egy él felvétele a mátrix módosítását jelenti olyan módon, hogy az $M[honnan,hova]$ értéket *igaz*-ra állítjuk.
- **Csúcsok() : Halmaz<V>**: Emlékezzünk rá, hogy magukról a csúcsokról nem tárolunk el semmilyen információt. Ezért itt visszaadhatunk egy halmazt, ami tartalmazza az 1 és $|V|$ közötti egész számokat, mint csúcs azonosítókat.
- **ÉLEK() : Halmaz<E>**: Ehhez két egymásbaágazott ciklussal érdemes átnézni az **M** mátrix minden elemét. Összegyűjtjük azokat az (i, j) párokat, ahol az **M[i,j]** értéke *igaz*, és az ezeknek megfelelő élek halmaza lesz a visszatérési érték.
- **VEZETÉL(honnan : V, hova : V) : logikai**: Egy lépéssel megoldható az ellenőrzés, mivel csak a **honnan**-hoz tartozó sor, és a **hova**-hoz tartozó oszlopan lévő értéket kell megvizsgálnunk, amit az **M[honnan,hova]** mutat. Ha itt *igaz* szerepel, akkor nincs él, különben pedig van.
- **SZOMSZÉDAI(csúcs : V) : Halmaz<V>**: Ez már egy jóval erőforrás-igényesebb művelet. Végig kell néznünk az **csúcs**-hoz tartozó sort az **M** mátrixban, és ki kell válogatni azokat az oszlopokat (illetve a hozzájuk tartozó csúcsokat), amelyeknél a mátrix értéke *igaz*. Ezen csúcsok halmaza lesz a függvény visszatérési értéke.

Látható, hogy bizonyos műveletek nagyon hatékonyan megvalósíthatók ebben a tárolási formában, mások pedig kevésbé. Röviden összefoglalva a tapasztalatokat:

- Tárhely szempontjából: mivel a létrehozott tömb mérete mindenkor a csúcsok számának a négyzete lesz, így ez mindenkor ugyanakkora. A tárolás hatékonysága tehát attól függ, hogy valójában hány él van a gráfban, amennyiben ez a szám kicsi, akkor a tömb sok felesleges helyet foglal.
- Sebesség szempontjából: általában elmondható, hogy akkor célszerű ezt a tárolási módot választani, ha a gráfon végzendő műveletek főleg az élek meglétének vizsgálatát igénylik (mivel ez csak 1 lépést igényel), kevésbé a szomszédsági vizsgálatokat (ami viszont a csúcsok számának megfelelő lépést igényel).

12.2.3. Szomszédsági listában tárolás

Az előzőleg felsorolt negatívumok miatt bizonyos esetekben célszerű lehet egy másik megvalósítást alkalmazni, ez pedig a szomszédsági listában való tárolás.



12.2. ábra. Irányítatlan, súlyozatlan gráf szomszédsági listával ábrázolva.

A generikus jelölés itt is konkretizálható, a V tulajdonképpen megfelel az $egész^+$ típusnak, azzal a korláttal, hogy értéke csak 1 és $|V|$ közötti lehet. Az E egy csúcspárt tárol, amit itt is csak paraméterként/visszatérési értékként használunk.

Az alapvető jól szemlélteti a 12.2. ábra. Létre kell hoznunk egy, a csúcsok számával megegyező méretű L tömböt, amelynek típusa $tömb <Lista <V>>$. Láncolt megvalósítás esetében tehát a tömb minden eleme egy láncolt lista elem típusú hivatkozást tartalmaz, ahol a lista elemek csúcs azonosítókat tartalmaznak ($\tilde{LánCElem} <V>$).

Az éleket pedig úgy tároljuk el, hogy az i . indexhez tartozó csúcs szomszédait az $L[i]$ láncolt lista tartalmazza (tehát az összes, i . csúcsból kiinduló élt).

Az alapműveletek megvalósításai:

- **ÚJÉL(honnan : V, hova : V)**: Egy él felvétele egy új láncolt lista elem felvételét jelenti. Az $L[honnan]$ láncolt listába kell felvenni egy olyan elemet, ami tárolja a $hova$ csúcs azonosítóját.
- **CSÚCSOK() : Halmaz <V>**: Az előző módszerhez hasonlóan a csúcsokról nem tárolunk el semmilyen információt. Ezért itt is visszaadhatunk egy halmazt, ami tartalmazza az 1 és $|V|$ közötti egész számokat, mint csúcs azonosítókat.
- **ÉLEK() : Halmaz <E>**: Ez a művelet hatékonyabb lesz ebben az esetben, csak össze kell fűznünk az összes láncolt listát az eredményhez.
- **VEZETÉL(honnan : V, hova : V) : logikai**: Az előző megoldással ellentétben ez egy meglehetősen erőforrás-igényes művelet, hiszen végig kell nézni az $L[honnan]$ láncolt listát. Rendezéssel persze valamennyire lehet ezen gyorsítani; illetve a listában csak annyi elem van, ahány él kiindul az adott csúcspontból, tehát valamivel jobb a helyzet, mint a csúcsmátrix szomszédkeresésénél.
- **SZOMSZÉDAI(csúcs : V) : Halmaz <V>**: Ez viszont itt könnyen megadható, csak vissza kell adni (esetlegesen kisebb technikai átalakítások után) magát az $L[csúcs]$ láncolt listát, ami épp ezt az adatot tárolja.

Röviden itt is összefoglalhatjuk, hogy mikor érdemes használni ezt a módszert:

- Tárhely szempontjából: a láncolt listák *fej* változói és az elemek *következő* mutatói mind valamennyi tárhelyet igényelnek. Amennyiben kevés él indul ki az egyes csúcsokból, akkor általában még ezzel együtt is kevesebb helyet foglal a szomszédsági lista, mint a csúcsmátrix, az élek számának növelésével azonban ez a helyzet romlik, majd meg is fordul. Tehát ahhoz, hogy egyértelműen eldöntsük, hogy melyik foglal kevesebb helyet, tudnunk kell a csomópontok és élek számát, illetve a mutatók és a tartalom által foglalt helyet.
- Sebesség szempontjából: általában elmondható, hogy akkor fogjuk ezt a tárolási módot választani, ha a gráfon végzendő műveletek főleg a szomszédsági kapcsolatok vizsgálatát igénylik majd (mivel ez csak 1 lépést igényel), kevésbé az élek meglétének lekérdezését (ez ugyanis akár $|V|$ darab lépést is igényelhet).

12.3. Bejárások

Miként a többi adatszerkeztnél, úgy itt is a bejárások vizsgálatával kezdjük az adatszerkezet műveleteinek bemutatását. Tegyük fel, hogy már rendelkezünk egy, a fenti módszerek egyikével eltárolt gráffal, és ennek szeretnénk feldolgozni az elemeit. A bejárás célja itt is hasonló, mint az előző dinamikus adatszerkezeteknél, azonban attól nemileg különbözik. Míg a láncolt lista és a fa esetében a bejárás kiindulópontja minden a *fej* vagy *gyökér* volt, addig a gráfok esetében nem szoktunk kijelölni egy megkülönböztetett pontot, így a bejárás bármelyik csúcsból elindítható. Másik lényeges különbség, hogy míg a láncolt lista és a bináris keresőfa esetében a bejárás az adatszerkezet összes elemének a feldolgozását jelenti, addig a gráf esetében a bejárás csak azokat fogja elérni, amelyek az előbb említett kiindulópontból elérhetőek a gráf élei mentén (a feldolgozandó elemek alatt a gráf esetében általában a csúcsokat értjük, habár az élek ugyanúgy a gráf alkotóelemei, és elképzelhetők olyan bejárások is, amelyek az élekre vonatkozóan is adnak valamiféle követelményt).

Ennek megfelelően tehát mindenki ismertetésre kerülő bejárás (szélességi és mélységi bejárás) esetében az algoritmus bemenete egy csúcs lesz, amelyből elindulunk. Az algoritmus pedig futása közben az összes, ebből a csúcsból elérhető csúcsot fel fogja dolgozni. A különbség pusztán az a két bejárás között, hogy különböző sorrendben érik el az egyes csúcsokat. A bejárás szempontjából nem kell azzal foglalkoznunk, hogy a feldolgozás maga milyen jellegű művelet, lehet hogy a képernyőre kiírás, lehet hogy módosítás, lehet hogy csak összegyűjt az elérte csúcsok referenciáit egy tömbbe, stb. A többi adatszerkezethez hasonlóan az algoritmus ezektől függetlenül, általánosan értelmezhető lesz.

Az a tény, hogy a bejárás nem éri el a gráf minden csúcsát nem tekinthető hátránynak. Ez a módszer ugyanis jól használható arra, hogy feltérképezzük az egyes csomópontok összefüggéseit, pl. összefüggő komponensek keresésére, a teljes gráf összefüggőségének vizsgálatára [1, 2, 11]. Vegyük észre, hogy ha az összes csúcsra szükségünk van, akkor nem is kell bejárás, hiszen erre létezik egy **Csúcsok** nevű függvényünk.

A bejárások csak a gráf már megvalósított publikus műveleteit használják, ezért ezeket nem is a gráf egy metódusaként, hanem külső műveletként valósítjuk meg, de ez csak technikai kérdés.

12.3.1. Szélességi bejárás

A *szélességi bejárás* alapelve, hogy elsőként mindenki a kiindulóponthoz legközelebbi csúcsokat dolgozza fel. A 12.3. ábrán jól látható ez a sorrend. A kezdőpont után az ettől egy élnyi távolságra lévő csúcsokat dolgozza fel (ezek feldolgozási sorrendje már tetszőleges lehet), majd ezt követően a két élnyi távolságra lévőket, stb. A feldolgozás sorrendje egy hullám terjedéséhez hasonlítható, a közelebbi csúcsok korábban, a távolabbi csúcsok pedig később kerülnek sorra.

Működésének alapja egy sor adatszerkezet. Ez a sor tartalmazza azokat a csúcsokat, amelyeket már elérte az algoritmus, azonban még nem lettek feldolgozva. Kezdetben a kiinduló csúcsot helyezzük el ebbe a sorba, majd egy ciklus segítségével addig fut az algoritmus, amíg ez nem lesz üres. minden ciklus iterációnál kivesszük a soron következő elemet, azt feldolgozzuk, majd az elem azon szomszédjait, amelyeket eddig még nem értünk el, elhelyezzük a sorba a későbbi feldolgozás érdekében.

Ezt a működést mutatja a 12.1. algoritmus. Az algoritmus első lépéseként az üres *S* sorba elhelyezi a *start* változó által tárolt csúcs, tehát a kiinduló csúcs referenciáját. Ugyanezt a csúcsot elhelyezi az *F* halmazba is (ez lesz a halmaz egyetlen eleme), amely a már „elérte” (tehát feldolgozott, vagy feldolgozásra váró) elemeket tartalmazza.

Ezt követően elindul a ciklus (4. sor), ami egészen addig fut, ameddig van elem a sorban. Ha elfogytak az elemek, az azt jelenti, hogy nincs több olyan csúcs, amelyik a kiinduló csúcsból elérhető, viszont még nem lett feldolgozva.

Az 5. sorban látható módon kivesszük a sorból az első elemet. Kezdetben csak a kiinduló elem van itt, tehát azt vesszük azonnal ki. A 6. sorban pedig már meg is történik ennek a feldolgozása.

Az aktuális elem feldolgozását követően az algoritmus egyesével megvizsgálja az elem minden szomszédját (7. sor), hogy melyik irányokba lehet folytatni a bejárást. Az egyes szomszédok közül bármelyik irányba továbbléphetünk majd, csak arra kell ügyelnünk, hogy ugyanazt az elemet ne dolgozzuk fel egymás után többször. Ez egyrészt hibás eredményt adna, másrészt ez az ellenőrzés azt is garantálja, hogy ne kerüljünk végtelen ciklusba. A feldolgozottságot egyszerűen tudjuk vizsgálni (8. sor), csak azt kell megnézni, hogy a vizsgált szomszédot elérők-e már, tehát benne van-e az *F* halmazban?

12.1. Algoritmus Gráf szélességi bejárása

Bemenet: g - Gráf $\langle V, E \rangle$ (a bejárando gráf)

Bemenet: $start$ - V (a kiinduló csúcs)

Bemenet: művelet - eljárás $\langle T \rangle$ (az elemeken végrehajtandó művelet, paramétere egy gráf csúcs)

Kimenet: F - Halmaz $\langle V \rangle$ (a bejárás során elért csúcsok halmaza)

1: függvény SZÉLESSÉGI BEJÁRÁS(g , $start$, $művelet$)

2: $S \leftarrow start$

3: $F \leftarrow \{start\}$

4: ciklus amíg ($S \neq \emptyset$)

5: $k \leftarrow S$

6: művelet(k)

7: ciklus $\forall x \in g.\text{SZOMSZÉDAI}(k)$

8: ha $x \notin F$ akkor

9: $S \leftarrow x$

10: $F \leftarrow F \cup \{x\}$

11: elágazás vége

12: ciklus vége

13: ciklus vége

14: vissza F

15: függvény vége

Felhasznált változók és függvények

- S : Egy Sor $\langle V \rangle$ adatszerkezet, amely a feldolgozásra váró elemeket tartalmazza.
 - F : Egy Halmaz $\langle V \rangle$ adatszerkezet, amely a már elért elemeket tartalmazza.
 - művelet : Tetszőleges feldolgozási művelet, amely minden csúcsra pontosan egyszer fut le.
 - k , x : V típusú segédváltozók.
-

Ha nincs, akkor a vizsgált szomszéd felé folytathatjuk majd a keresést. Nem lépünk rá azonnal, csak elhelyezzük a sorba a későbbi feldolgozás érdekében a 9. sorban látható művelettel.

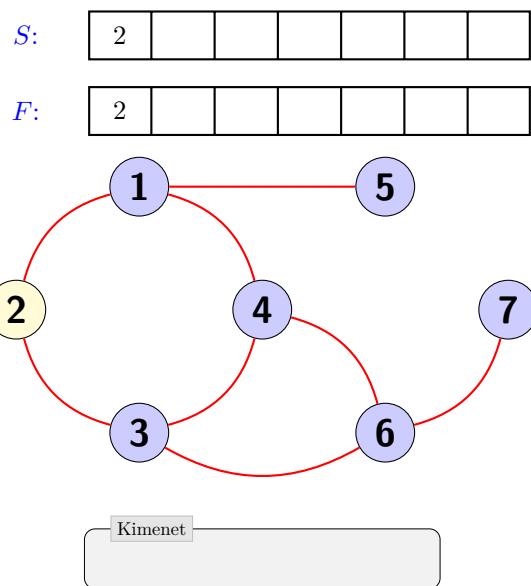
Az így elhelyezett elem feldolgozása majd a külső ciklus egy későbbi iterációjában fog megtörténni. A sor működéséből adódóan az 5. sor minden az egyik legközelebbi csúcsot fogja visszaadni, ez pedig garantálja, hogy a szélességi bejárás valóban azt az eredményt adja, amit várunk.

Véges méretű gráfok esetében a még el nem ért szomszédok idővel elfogynak, ilyenkor már nem kerül a sorba új csúcs. A benne lévő csúcsok feldolgozását követően így a sor hamarosan üres lesz, ezzel pedig a külső ciklus feltétele nem teljesül többé, így véget ér a bejárás.

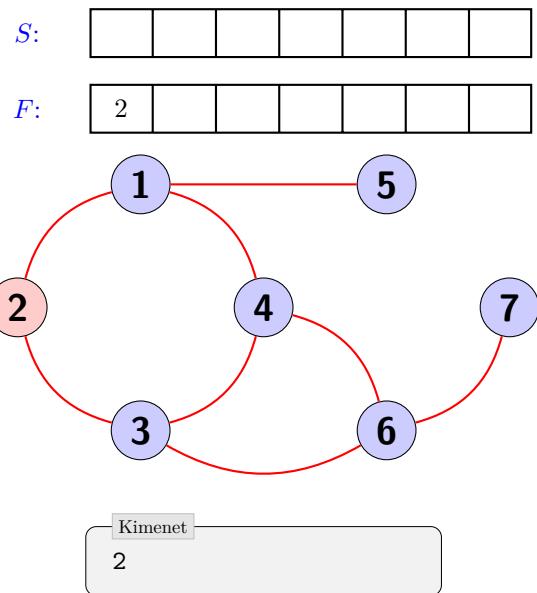
A szélességi bejárás működésére látható egy példa a 12.3. ábrán. Az ábra színezése minden egyes csúcs esetén az alábbi állapotokat különbözteti meg:

- Kék: a gráf csúcsai, amelyekhez még nem értünk el (kezdetben minden csúcsot ilyennek tekintünk). Tehát ezek még nincsenek benne az F halmazban.
- Sárga: azok a csúcsok, amelyeket már elérünk egy másik csúcson keresztül, azonban még nem dolgoztuk fel őket. Tehát éppen benne vannak az S sorban, várakoznak a későbbi feldolgozásra. Emellett benne vannak az F halmazban is.
- Szürke: azok a csúcsok, amelyek már feldolgozásra kerültek. Tehát benne vannak az F halmazban, de az S -ben már nincsenek benne.
- Piros: az aktuálisan feldolgozás alatt álló csúcs (tehát valójában már szürke, csak a jobb áttekintetőség kedvéért kapott egy külön színt). Az algoritmusban a k változó értéke.

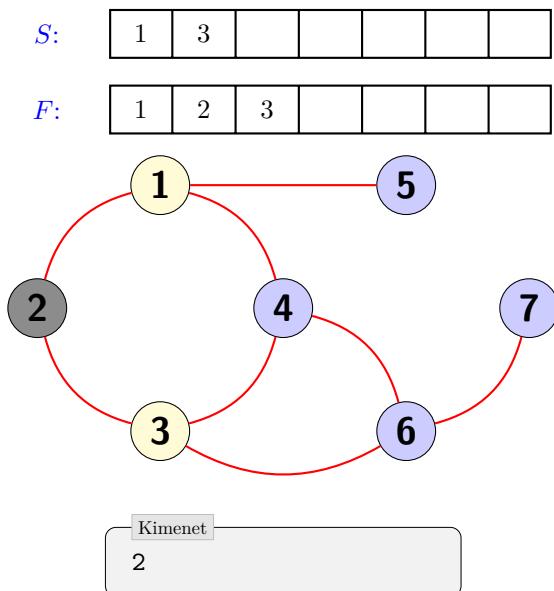
Érdemes megjegyezni, hogy a szélességi bejárás csak egy összefüggő komponens bejárására alkalmas. Amennyiben biztosítani szeretnénk, hogy a gráf minden csúcsát elérje, akkor egy újabb ciklussal a bejárást meg kell hívni minden csúccsal (illetve csak azokra, amelyeket nem dolgoztak fel az előzőleg futtatott bejárások). Az egyes hívások között tartsuk meg az F értékét, hogy ugyanazt az elemet ne dolgozzuk fel többször.



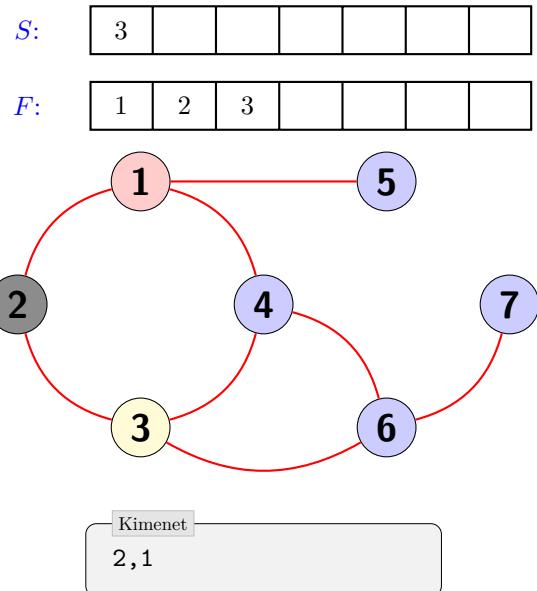
(a) Az algoritmus kezdeti lépéseként elhelyezi a kezdőcsúcsot (ami egy paraméter, jelen példában legyen 2) az *S* sorba és az *F* halmazba.



(b) A sorból kiveszi az első elemet, majd feldolgozza.



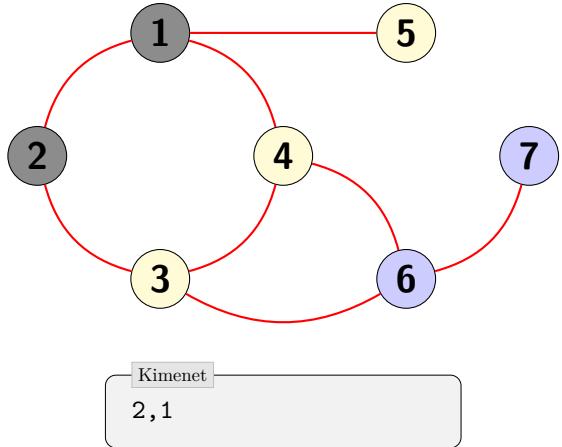
(c) A feldolgozott csúcs szomszédai bekerülnek a sorba és a halmazba (1,3).



(d) A sorból kiveszi a következő feldolgozandó 1-es csúcsot, majd feldolgozza.

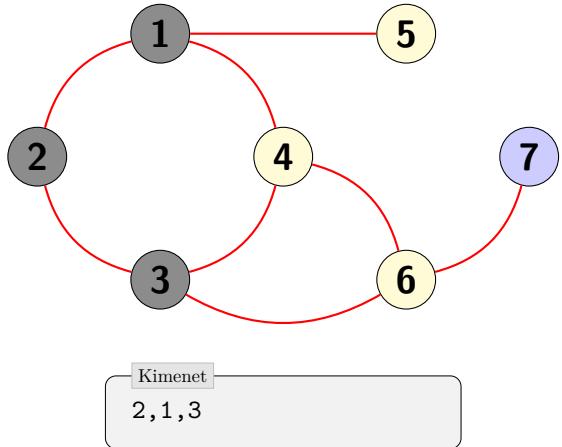
12.3. ábra. Gráf szélességi bejárása.

<i>S:</i>	3	5	4				
<i>F:</i>	1	2	3	4	5		



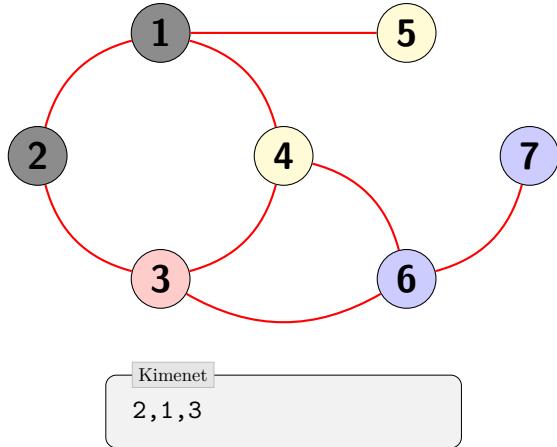
(e) Ennek a szomszédai (csak azok, amelyek még nincsenek benne) szintén bekerülnek a halmazba és a sorba (5,4).

<i>S:</i>	5	4	6				
<i>F:</i>	1	2	3	4	5	6	



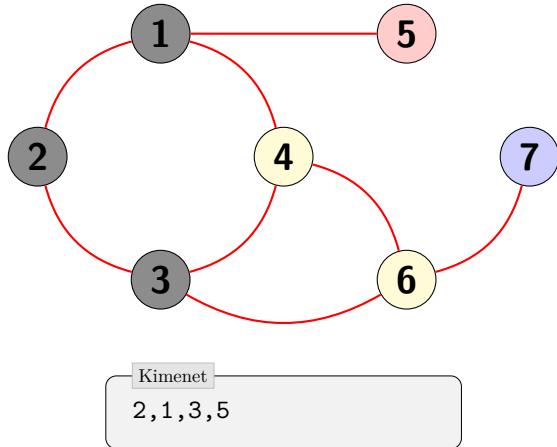
(g) Ennek a szomszédai (csak azok, amelyek még nincsenek benne) szintén bekerülnek a halmazba és a sorba (6).

<i>S:</i>	5	4					
<i>F:</i>	1	2	3	4	5		



(f) A sorból kiveszi a következő feldolgozandó 3-as csúcsot, majd feldolgozza.

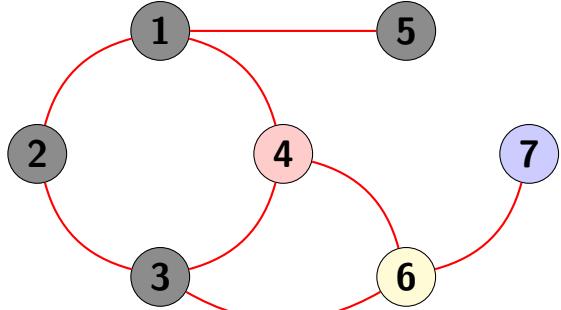
<i>S:</i>	4	6					
<i>F:</i>	1	2	3	4	5	6	



(h) A következő feldolgozandó elem az 5-ös. Ennek nincs új szomszéda, így nem bővül az *S* sor.

12.3. ábra. Gráf szélességi bejárása. (folytatás)

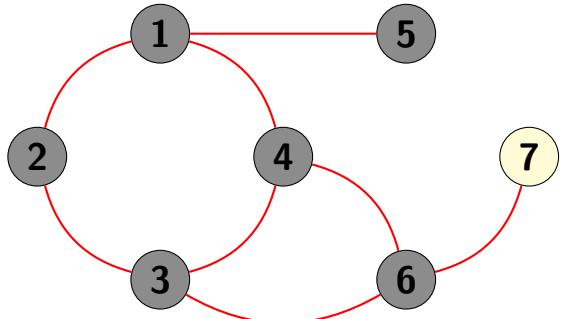
<i>S:</i>	<table border="1"><tr><td>6</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	6						
6								
<i>F:</i>	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td></td></tr></table>	1	2	3	4	5	6	
1	2	3	4	5	6			



Kimenet
2,1,3,5,4

(i) A sorban a következő a 4-es elem. Ezt is feldolgozza, és mivel nincsenek még el nem ért szomszédai, itt sem kerül új elem a sorba.

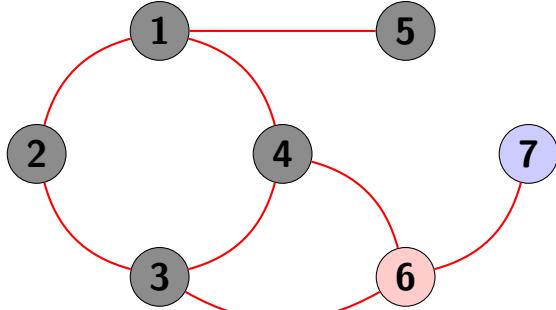
<i>S:</i>	<table border="1"><tr><td>7</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	7						
7								
<i>F:</i>	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	1	2	3	4	5	6	7
1	2	3	4	5	6	7		



Kimenet
2,1,3,5,4,6

(k) Ennek a szomszéda még belekerül a sorba (7).

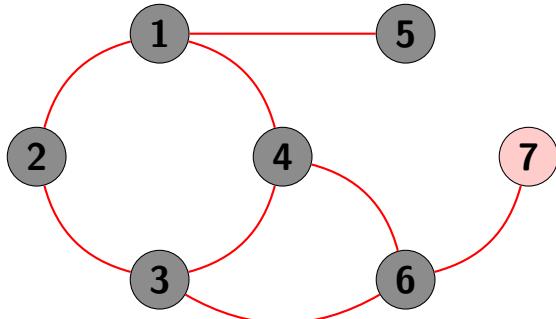
<i>S:</i>	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>							
<i>F:</i>	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td></td></tr></table>	1	2	3	4	5	6	
1	2	3	4	5	6			



Kimenet
2,1,3,5,4,6

(j) A sorban következő 6-os elem feldolgozása következik.

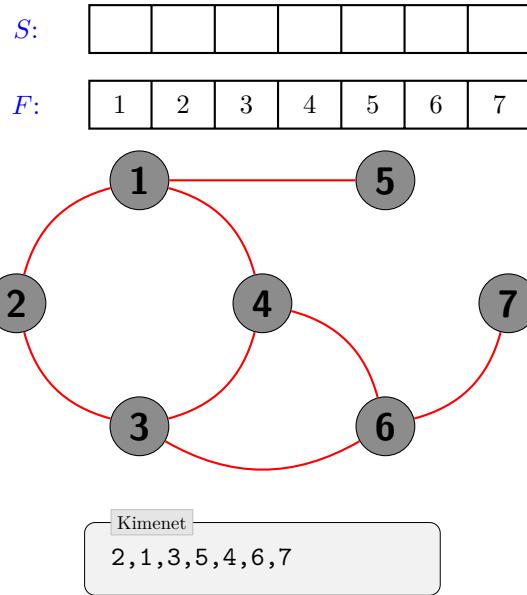
<i>S:</i>	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>							
<i>F:</i>	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	1	2	3	4	5	6	7
1	2	3	4	5	6	7		



Kimenet
2,1,3,5,4,6,7

(l) Az egyetlen 7-es elem is kikerül a sorból és feldolgozásra kerül. Feldolgozatlan szomszédai nincsenek, így nem kerül a sorba új elem.

12.3. ábra. Gráf szélességi bejárása. (folytatás)



(m) Mivel üres a sor, nem teljesül tovább a külső ciklus feltétele, emiatt az algoritmus végetér. Láthatóan bejártuk az egész gráfot.

12.3. ábra. Gráf szélességi bejárása. (folytatás)

Ezt a tulajdonságot azonban jól ki is lehet használni, ha éppen az a kérdés, hogy megadott csúcsból melyik további csúcsok érhetők el. Az algoritmus futása során ezeket pont összegyűjtöttük a *F* halmazban, ezért visszatérési értékként ezt vissza is adjuk.

Szintén gyakori módosítás, ha az algoritmust kiegészítjük néhány kiegészítő művelettel:

- minden új csomópont felvétele esetén (tehát amikor találtunk egy új, még el nem ért szomszédot) eltárolhatjuk, hogy melyik csúcsból értük el ezt a pontot. Ilyenkor egy, a kezdőpontból kiinduló fát kapunk eredményül, amely mutatja az egyes csúcsok legrövidebb elérési útjait (ahol úthossz alatt a kiindulópontból ide vezető utakon található élek számát értjük).
- minden új csomópont felvétele esetén eltárolhatjuk azt is, hogy ŏ milyen messze van a kiinduló ponttól. Ez a távolság minden esetben egyszerűen számolható, hiszen minden pont eggel több, mint annak a csomópontnak a távolsága, ahonnan elérte az új elemet (kivéve persze a kiinduló csúcsot, ahol ez 0).

Ez utóbbi módosítással a szélességi keresés használható súlyozatlan gráfok legrövidebb útjainak keresésére. Ebben az esetben úthossz alatt az adott út során érintett élek számát értjük. Egyszeri lefuttatása után megadjva a *start* pontból az összes többi csúcsba vezető legrövidebb út hosszát.

12.3.2. Mélységi bejárás

A *mélységi bejárás* ugyanazokat a csúcsokat fogja elérni, mint a szélességi, azonban más sorrendben. Az előzőleg megismert szélességi bejárás minden a legközelebbi csúcsokat dolgozta fel először, és csak akkor látott neki a távolabbi csúcsoknak, ha már az összes közelebbivel végzett. A mélységi bejárás ezzel szemben elindul egy úton, és egészen addig halad előre a szomszédsági kapcsolatokon keresztül, amíg nem jut egy olyan csúcsra, ahonnan már nem tud továbblépni. Ilyenkor egy szinttel visszább lép, és az előző csúcsnál próbál keresni egy másik, még fel nem dolgozott szomszédöt. Mindezt addig folytatja, amíg elfogynak a választható feldolgozatlan szomszédek.

Megjegyzés

Nem véletlen a hasonlóság a mélységi keresés alapelve, és a visszalépéses keresésnél (10. fejezet) látható módszer között (folyamatosan halad előre, és zsákutcába kerülve visszalép egy szintet, és új irányt keres). A visszalépéses keresés tulajdonképpen a mélységi bejárás lépéseiit végzi.

A szélességi bejárásnál egy **Sor** biztosította, hogy a csomópontok feldolgozása abban a sorrendben történjen, ahogy először találkoztunk velük (ami így megfelel a kiindulóponttól való távolságnak), a mélységi bejárásnál verem segítségével tudjuk eltárolni a keresés során bejárt utat. Magát a vermet nem mi kezeljük közvetlenül, hanem az algoritmust rekurzívan adjuk meg, így a visszalépéses kereséshez hasonló megoldáshoz jutunk. A mélységi bejárás megvalósítását mutatja a 12.2. algoritmus.

12.2. Algoritmus Gráf mélységi bejárása

Bemenet: g - Gráf $\langle V, E \rangle$ (a bejárando gráf)

Bemenet: k - V (az aktuálisan vizsgált csúcs)

Bemenet: F - Halmaz $\langle V \rangle$ (a már elérte elemek)

Bemenet: művelet - eljárás $\langle T \rangle$ (az elemeken végrehajtandó művelet, paramétere egy gráf csúcs)

Kimenet: F - Halmaz $\langle V \rangle$ (a halmaz bővítése az aktuális elemmel)

1: eljárás MÉLYSÉGI BEJÁRÁS REKURZIÓ(g, k , címszerint F , művelet)

2: $F \leftarrow F \cup \{k\}$

3: művelet(k)

4: ciklus $\forall x \in g.SZOMSZÉDAI(k)$

5: ha $x \notin F$ akkor

6: MÉLYSÉGI BEJÁRÁS REKURZIÓ($g, x, F, művelet$)

7: elágazás vége

8: ciklus vége

9: eljárás vége

Bemenet: g - Gráf $\langle V, E \rangle$ (a bejárando gráf)

Bemenet: start - V (a kiinduló csúcs)

Bemenet: művelet - eljárás $\langle T \rangle$ (az elemeken végrehajtandó művelet, paramétere egy gráf csúcs)

Kimenet: . - Halmaz $\langle V \rangle$ (a bejárás során elérte csúcsok halmaza)

10: függvény MÉLYSÉGI BEJÁRÁS($g, start, művelet$)

11: $F \leftarrow \emptyset$

12: MÉLYSÉGI BEJÁRÁS REKURZIÓ($g, start, F, művelet$)

13: vissza F

14: függvény vége

Felhasznált változók és függvények

- $k, x : V$ típusú segédváltozók.
- F : Egy **Halmaz** $\langle V \rangle$ adatszerkezet, amely a már elérte elemeket tartalmazza.
- művelet : Tetszőleges feldolgozási művelet, amely minden csúcsra pontosan egyszer fut le.

Rekurzív algoritmus lévén találunk egy indító eljárást is, ez csak inicializálja a már feldolgozott csúcsokat tartalmazó F halmaz tartalmát (11. sor), majd meghívja magát a rekurziót, első paraméterként átadva a kiinduló csúcsot. Második paraméterként minden az F halmazt fogjuk átadni címszerint a következő szintekre, hogy ebbe tudjuk gyűjteni a már feldolgozott elemeket.

Maga a rekurzió meglehetősen egyszerű, minden lefutásakor eltárolja az éppen feldolgozandó elemet az F halmazba (2. sor), majd elvégzi a paraméterként kapott csúcs feldolgozását a 3. sorban.

A továbblépéshez megvizsgálja az aktuális csúcs szomszédait egy ciklus segítségével (4. sor). A továbblépés egyetlen követelménye, hogy a kiszemelt csúcsot még nem dolgoztuk fel. Ezt egyszerűen tudjuk ellenőrizni az F halmaz alapján (5. sor), hiszen minden elérte csúcs esetében a feldolgozás előtt közvetlenül elhelyeztük a csúcsot ide.

Amennyiben a szomszéd nincs benne az F -ben, akkor rekurzív módon újrahívja önmagát a függvény (6. sor). Látható, hogy a szomszédok ellenőrzése hasonló, mint a szélességi keresésnél, a lényeges különbség, hogy a szomszédok ellenőrzése a függvényben a feldolgozás előtt történik, míg a szélességi keresésben a feldolgozás előtt történik.

ség pusztán az, hogy a mélységi keresés nem tárolja az előrendő elemeket, hanem egy új csúcs felfedezése esetén azonnal átlép arra, és ott folytatja a feldolgozást.

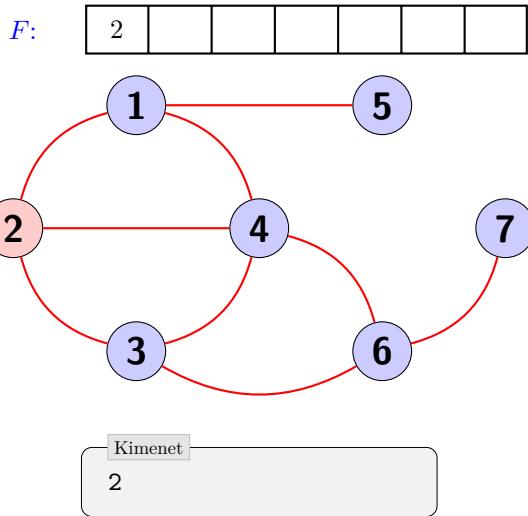
A mélységi bejárás működésére látható egy példa a 12.4. ábrán. Az ábra színezése minden egyes csúcs esetén az alábbi állapotokat különbözteti meg:

- Kék: a gráf csúcsai, amelyekhez még nem értünk el (kezdetben minden csúcsot ilyennek tekintünk). Tehát ezek még nincsenek benne az F halmazban.
- Szürke: azok a csúcsok, amelyeket már elértünk és egyben már feldolgozásra kerültek. Tehát benne vannak az F halmazban.
- Piros: az aktuálisan feldolgozás alatt álló csúcs, ami az algoritmusban a k változó értéke.

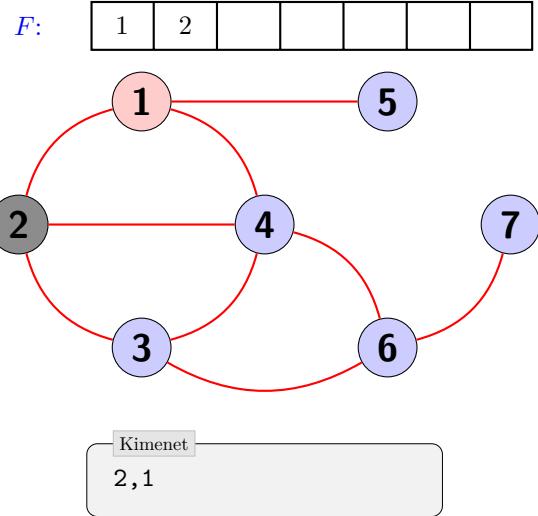
A mélységi bejárásra is igaz, hogy önmagában csak egy összefüggő komponens bejárására alkalmas. Amennyiben biztosítani szeretnénk, hogy a gráf minden csúcsát elérje, akkor egy újabb ciklussal a bejárásról meg kell hívni minden csúcsra (illetve csak azokra, amelyeket nem dolgoztak fel az előzőleg futtatott bejárások).

Az algoritmus kiegészíthetjük néhány kiegészítő művelettel:

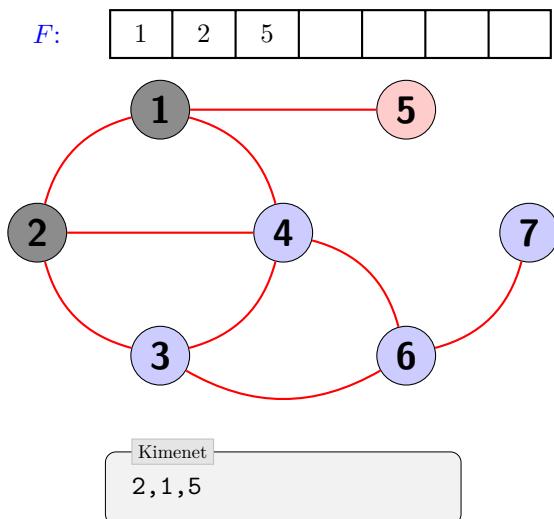
- minden új csomópont felvétele esetén (tehát amikor találtunk egy új, még el nem ért szomszédot) eltárolhatjuk, hogy melyik csúcsból értük el ezt a pontot. Ilyenkor egy, a kezdőpontból kiinduló fát kapunk eredményül, amely mutatja az egyes csúcsok egy lehetséges elérési útjait.
- Eltárolhatjuk az egyes csúcsok elérési és elhagyási idejét, amely alapján érdekes információkat tudhatunk meg a gráf felépítéséről (bővebben lásd [2]).



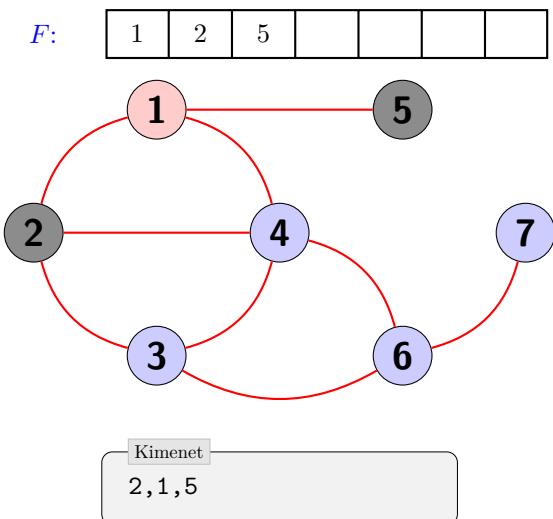
(a) Az algoritmus kezdeti lépéseként elhelyezi a kezdőcsúcsot (ami egy paraméter, jelen példában legyen 2) az *F* halmazba, és egyből fel is dolgozza azt.



(b) A 2-es elem első szomszédja az 1-es, ezért ide lép tovább.



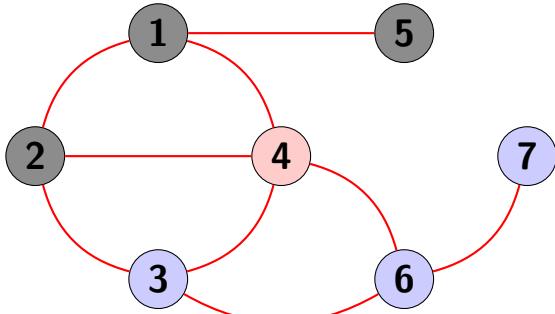
(c) Az 1-es első szomszédja az 5, ezért ide lép tovább.



(d) Az 5-ösnek csak egy szomszédja van, az 1-es, de az már benne van az *F*-ben. Ezért visszalép a rekurzió.

12.4. ábra. Gráf mélységi bejárása.

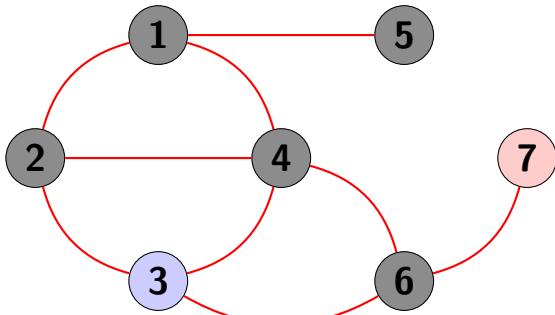
<i>F:</i>	1	2	4	5			
-----------	---	---	---	---	--	--	--



Kimenet
2,1,5,4

(e) Az 1-esről még tovább lehet lépni a 4-re, ezért újrahívja magát a függvényt.

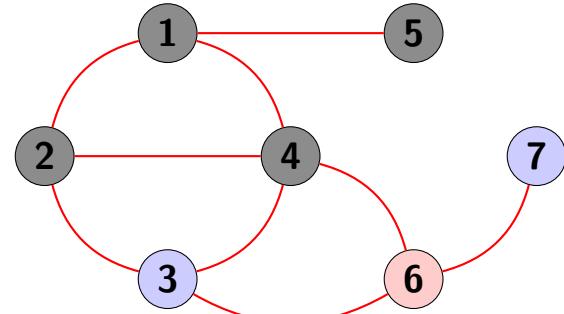
<i>F:</i>	1	2	4	5	6	7	
-----------	---	---	---	---	---	---	--



Kimenet
2,1,5,4,6,7

(g) A 6-ról pedig tovább lehet lépni a 7-re.

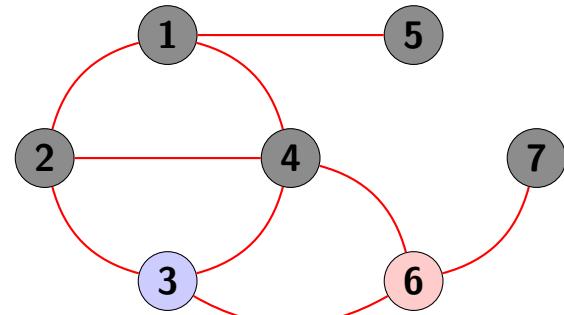
<i>F:</i>	1	2	4	5	6		
-----------	---	---	---	---	---	--	--



Kimenet
2,1,5,4,6

(f) A 4-es elem első szomszéja a 6, így továbblép ide.

<i>F:</i>	1	2	4	5	6	7	
-----------	---	---	---	---	---	---	--

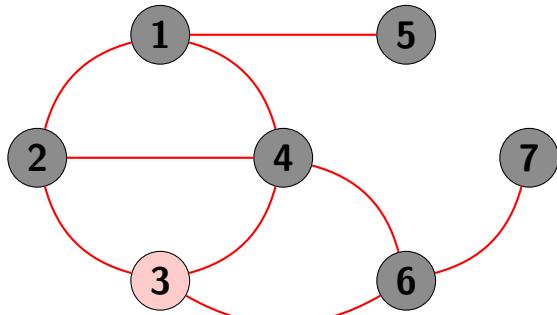


Kimenet
2,1,5,4,6,7

(h) A 7-nek már nincs olyan szomszéja, akit nem dolgoztunk fel, ezért visszalépés.

12.4. ábra. Gráf mélységi bejárása. (folytatás)

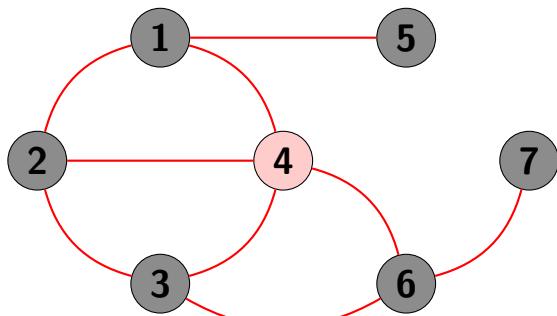
<i>F:</i>	1	2	3	4	5	6	7
-----------	---	---	---	---	---	---	---



Kimenet
2,1,5,4,6,7,3

(i) A 6-osból el tudunk jutni a 3-ba is, mivel az még nincs feldolgozva.

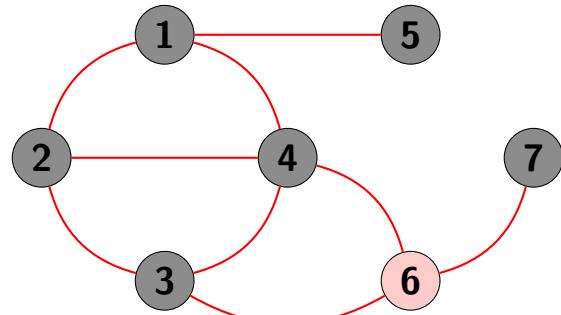
<i>F:</i>	1	2	3	4	5	6	7
-----------	---	---	---	---	---	---	---



Kimenet
2,1,5,4,6,7,3

(k) A 6-osból is visszalép a rekurzió a 4-esre.

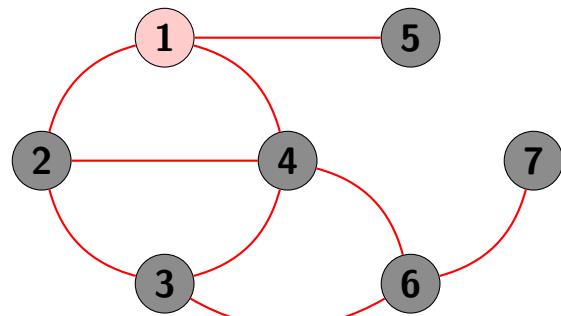
<i>F:</i>	1	2	3	4	5	6	7
-----------	---	---	---	---	---	---	---



Kimenet
2,1,5,4,6,7,3

(j) Mivel a 3-as minden szomszédja feldolgozott, ezért visszalépünk.

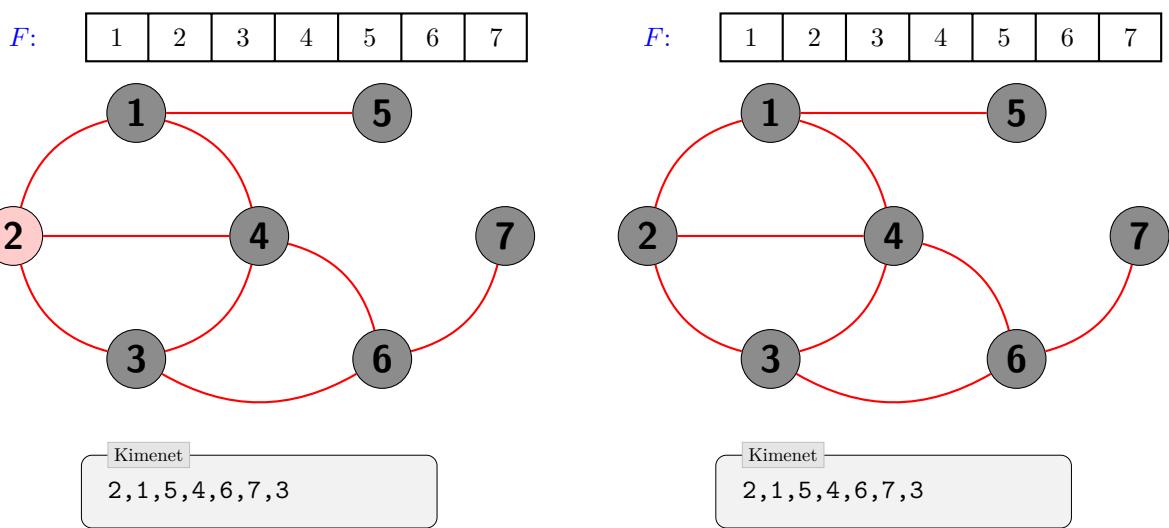
<i>F:</i>	1	2	3	4	5	6	7
-----------	---	---	---	---	---	---	---



Kimenet
2,1,5,4,6,7,3

(l) A 4-esből is visszalép az 1-esre.

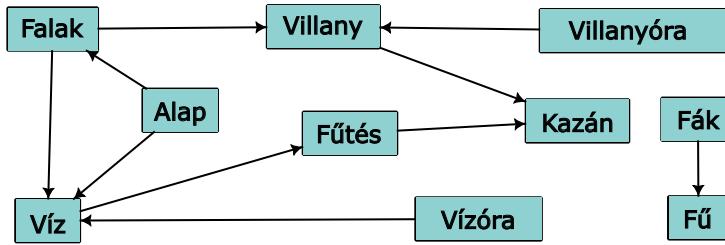
12.4. ábra. Gráf mélységi bejárása.(folytatás)



(m) Az 1-esből visszalépés a kiinduló elemre.

(n) A 2-esnek sincs már feldolgozatlan eleme, így az egész rekurzív hívás véget ér. Bejártunk minden csúcsot.

12.4. ábra. Gráf mélységi bejárása.(folytatás)



12.5. ábra

12.4. Topologikus rendezés

12.4.1. Topologikus rendezés fogalma

Egy $g = \text{Gráf}\langle V, E \rangle$ irányított gráf topologikus rendezésén a V csúcsok egy olyan sorba rendezését értjük, amelyre igaz, hogy ha E -ben szerepel egy (u, v) él, akkor a sorrendben u meg kell, hogy előzze v -t. Előfordulhat, hogy egy gráfnak többféle topologikus rendezése is van, a fenti szabálynak megfelelő sorrendek közül ugyanis bármelyik elfogadható. Amennyiben a gráf tartalmaz irányított kört, akkor topologikus rendezése nem lehet. Ezért a későbbiekben feltételezzük, hogy körmentes, irányított gráffal dolgozunk.

Példaként vegyük a 12.5. ábrán látható gráfot. Ennek néhány lehetséges topológiai rendezése:

- Alap, Falak, Vízóra, Víz, Villanyóra, Villany, Fűtés, Kazán, Fák, Fű
- Fák, Villanyóra, Vízóra, Alap, Falak, Víz, Villany, Fűtés, Fű, Kazán

A példából is jól látható, hogy ez nagyon jól használható különböző egymásra épülő folyamatok kezelésére. Például:

- Egy összetett gyártási folyamat során meg van határozva, hogy milyen lépések milyen egyéb lépések után hajthatók végre. A topologikus rendezés megad egy lehetséges sorrendet, ahogy egymás után végre lehet hajtani a lépéseket úgy, hogy az előfeltételek minden adottak legyenek.
- Szoftverfejlesztés során számos külső csomagot használnak a fejlesztők, amelyek további csomagokat igényelnek, stb. A topologikus rendezés megad egy sorrendet, ahogy ezeket le lehet fordítani a függőségek figyelembevételével.

Nagyméretű feladatok esetében egy ilyen rendezés megtalálása már meglehetősen nehéz, ilyenkor célszerű felhasználni a mélységi bejárásban alapuló algoritmusunkat. Az az ábrán is jól látható, hogy a feladat egyszerűen leírható gráfok segítségével, ahol a csomópontok az egyes megoldandó részproblémák, egy (u, v) irányított él pedig azt mutatja, hogy a v részprobléma függ a u részproblémától.

12.4.2. Topologikus rendezés algoritmusa

A mélységi keresés egy lehetséges bővítési lehetősége, hogy folyamatosan figyeljük a csúcsok elhagyásának idejét. Ehhez módosíthatjuk a 12.2. algoritmust úgy, hogy a 8. sor végére elhelyezünk egy új sort, ami eltárolja, hogy a k csúcsot mikor hagytuk el (ennek nem kell minden időnek lennie, elég egy számlálót folyamatosan növelni).

Ez az elhagyási idő meglepően sok információt nyújt a gráf felépítésével kapcsolatban, a topologikus rendezés okán nekünk most elég az alábbi következményt észrevenni: ha egy keresés során elérünk egy u és egy v elemet is, és az v elem elhagyási ideje kisebb mint a u elhagyási ideje, akkor biztos nem vezet u -ból út a v -be. Ugyanis ha lenne ilyen, akkor az u -n keresztül eljutottunk volna v -be és a mélységi bejárás működésének megfelelően elsőként a v -t hagytuk volna el, majd a rekurzió visszalépése után valamikor a u -t. Ez azonban ellent mondana az elhagyási időknek.

A topológiai rendezés szempontjából ez azért lényeges, mert ilyenkor biztos nincs u -ból v -be vezető út, tehát az u -t a topologikus sorrendben a v előtt helyezhetjük. Ebből a szempontból egyébként lényegtelen,

hogy a v -ből vezet-e út u -ba, vagy egyikből se lehet elérni a másikat, minden esetben jó választás ha u előbb van a rendezésben.

Ebből adódik az az ötlet, hogy indítsunk el egy mélységi bejárást, közben tároljuk el az egyes csúcsok elhagyási idejét, és ezt a sorrendet megfordítva megkapunk egy lehetséges topologikus rendezést. Ez majdnem jó is, gondolni kell azonban arra, hogy a gráf nem biztos, hogy összefüggő, mint ahogyan a 12.5. ábrán is látható („fű”, „fák”).

Ezt pedig úgy tudjuk feloldani, hogy egymást követően elindítjuk a mélységi keresést minden csúcsból, ügyelve arra, hogy az egyszer már feldolgozott csúcsokat nem dolgozzuk fel újra (tehát az F halmazt nem töröljük az egyes hívások között) és az elhagyási időket számoló időzítőt sem nullázzuk le, tehát az egyre növekvő számokat rendel az egyes csúcsokhoz.

Így már minden csomóponthoz fogunk kapni egy egyedi elhagyási idő értéket, és a fenti szabály továbbra is érvényes lesz. Ha egy v elem elhagyási ideje kisebb, mint az u elhagyási ideje, akkor biztos nem vezet út v -ből u -ba, tehát az u megelőzheti v -t a topologikus sorrendben. Mivel nem lehet két csúcsnak azonos az elhagyási ideje, így ez a csökkenő sorrend egyértelműen előállítható, ami megfelel a rendezés szabályainak.

A fenti módszer már jól működik és még egyszerűsíteni is tudunk rajta: nem szükséges eltárolni az időket, hiszen ez egy folyamatosan növekvő érték. Elég azt figyelnünk, hogy milyen sorrendben hagyjuk el az egyes csúcsokat, és ezt a sorrendet eltárolni. Az így kapott sorozat megfordítva már maga a topologikus rendezés lesz.

További egyszerűsítés, hogy ha elhagyáskor a csúcsokat eleve fordított sorrendben tároljuk el egy listában (amire egy láncolt lista tökéletesen alkalmas, hiszen gyorsan tudunk az elejére szúrni), akkor később azt meg se kell fordítani, hanem eleve megkapjuk a topologikus sorrendet.

12.3. Algoritmus Gráf topologikus rendezése

Bemenet: g - **Gráf** $\langle V, E \rangle$ (a gráf, aminek a topológikus rendezését keressük)

Bemenet: k - **V** (az aktuálisan vizsgált csúcs)

Bemenet: F - **Halmaz** $\langle V \rangle$ (a már elért csúcsokat tartalmazó halmaz)

Bemenet: L - **Lista** $\langle V \rangle$ (láncolt lista, ami az elhagyás sorrendben tárolja a csúcsokat)

Kimenet: F - **Halmaz** $\langle V \rangle$ (az eredeti halmaz kiegészítve k -val)

Kimenet: L - **Lista** $\langle V \rangle$ (az eredeti láncolt lista kiegészítve k -val)

```

1: eljárás TOPOLOGIKUSREKURZIÓ( $g, k$ , címszerint  $F$ , címszerint  $L$ )
2:    $F \leftarrow F \cup \{k\}$ 
3:   ciklus  $\forall x \in g.\text{SZOMSZÉDAI}(k)$ 
4:     ha  $x \notin F$  akkor
      MÉLYSÉGI BEJÁRÁS REKURZIÓ( $g, x, F, L$ )
6:   elágazás vége
7:   ciklus vége
8:   L.BESZÚR(1, k)
9: eljárás vége

```

Bemenet: g - **Gráf** (a gráf, aminek a topológikus rendezését keressük)

Kimenet: L - **Lista** $\langle V \rangle$ (a csúcsok topologikus rendezése)

10: függvény TOPOLOGIKUSRENDEZÉS(g)

```

11:    $F \leftarrow \emptyset$ 
12:    $L \leftarrow \emptyset$ 
13:   ciklus  $\forall x \in g.\text{CSÚCSOK}()$ 
14:     TOPOLOGIKUSREKURZIÓ( $g, x, F, L$ )
15:   ciklus vége
16:   vissza  $L$ 
17: függvény vége

```

Felhasznált változók és függvények

- F : Egy **Halmaz** $\langle V \rangle$ adatszerkezet, amely a már elért elemeket tartalmazza.
- k, x : **V** típusú segédváltozók.

A pszeudokód a mélységi bejárásban alapul, ezért csak a különbségeket érdemes kiemelni:

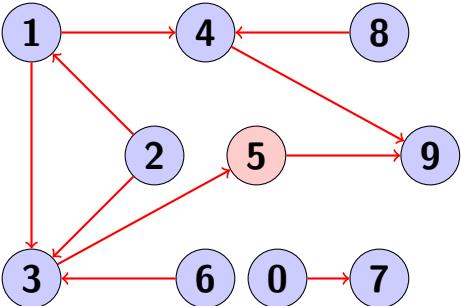
- Eltűnt a bejáráshoz szükséges végrehajtandó „művelet” paraméter, hiszen nem akarunk ilyet végrehajtani az elemeken. A kezdőcsúcs változó is eltűnt, hiszen minden pontból elindítjuk majd a rekurziót.
- Megjelent egy új L lista, amely üresen indul, majd ezt követően minden átadódik a rekurziónak. A rekurzív függvény kilépéskor (8. sor) minden csúcsot (k) ennek a listának az elejére.
- Újdonság az indító függvényben megjelenő új ciklus (13. sor), amely biztosítja, hogy minden csúcsból elindítjuk a bejárást.
- Vegyük észre, hogy az egyes rekurzív indítások között nem töröljük az F és L változók tartalmát!

F:

5									
---	--	--	--	--	--	--	--	--	--

L:

--	--	--	--	--	--	--	--	--	--



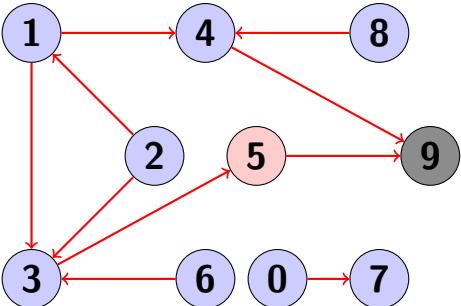
(a) Véletlenszerűen elindítjuk a rekurziót az 5-ös csúcsból. Ez bekerül az *F* halmazba, az *L*-be még nem, ide csak kilépéskor.

F:

5	9								
---	---	--	--	--	--	--	--	--	--

L:

9									
---	--	--	--	--	--	--	--	--	--



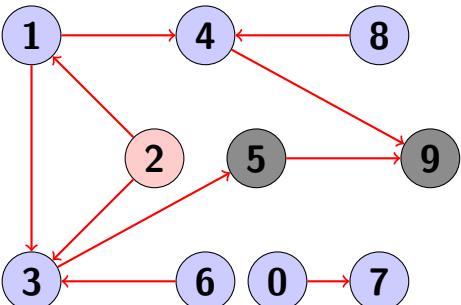
(c) 9-nek nincs szomszéda, ezért visszalép. A 9 bekerül az *L*-be és a rekurzió visszalép az 5-re.

F:

5	9	2							
---	---	---	--	--	--	--	--	--	--

L:

5	9								
---	---	--	--	--	--	--	--	--	--



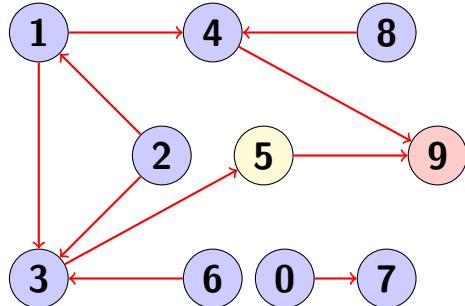
(e) Újra indítjuk a rekurziót, pl. a 2-esből.

F:

5	9								
---	---	--	--	--	--	--	--	--	--

L:

--	--	--	--	--	--	--	--	--	--



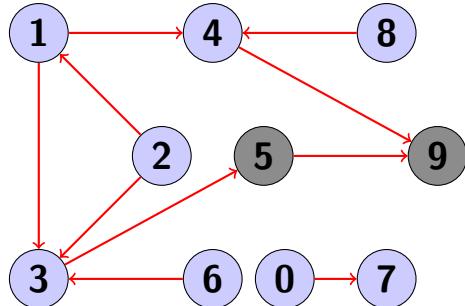
(b) Az 5-ös egyetlen szomszédja a 9-es, ezért ide lép át.

F:

5	9								
---	---	--	--	--	--	--	--	--	--

L:

5	9								
---	---	--	--	--	--	--	--	--	--



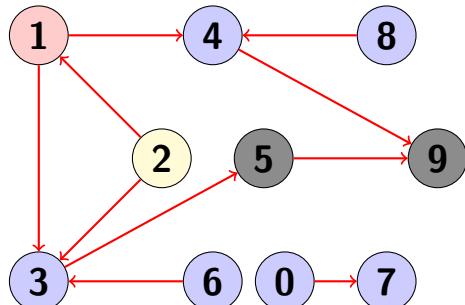
(d) 5-nek sincs további szomszéda, így bekerül az *L* elejére. A rekurzió végetér.

F:

5	9	2	1						
---	---	---	---	--	--	--	--	--	--

L:

5	9								
---	---	--	--	--	--	--	--	--	--

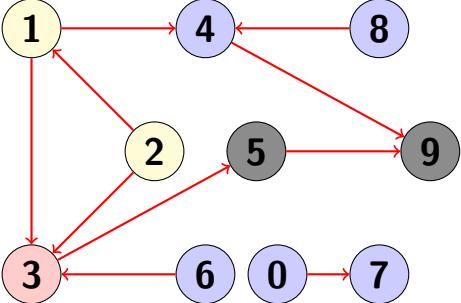


(f) A 2-es szomszéda az 1-est, így átlép ide.

12.6. ábra. Gráf topologikus rendezése.

<i>F:</i>	5 9 2 1 3	
-----------	-------------------	--

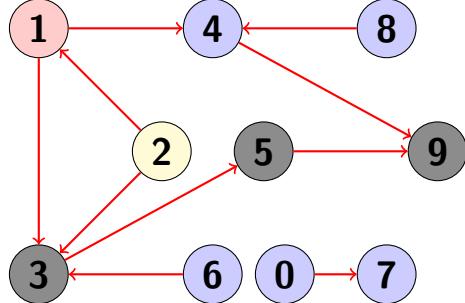
<i>L:</i>	5 9	
-----------	-------	--



(g) Majd innen továbblép a 3-as csúcsra.

<i>F:</i>	5 9 2 1 3	
-----------	-------------------	--

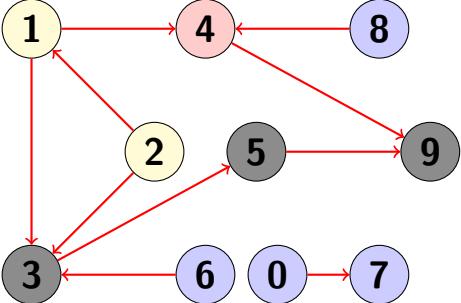
<i>L:</i>	3 5 9	
-----------	-----------	--



(h) A 3-as egyetlen szomszédja az 5, de az már benne van *F*-ben, ezért visszalép és bővíti az *L*-et.

<i>F:</i>	5 9 2 1 3 4	
-----------	-----------------------	--

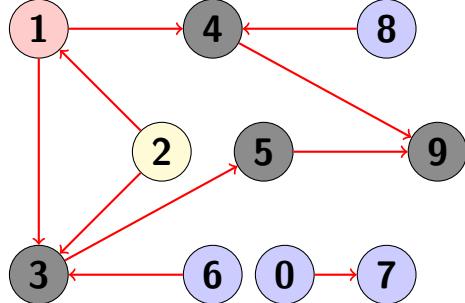
<i>L:</i>	3 5 9	
-----------	-----------	--



(i) Egyesből továbblép még a 4-esbe, hiszen az is szomszédja.

<i>F:</i>	5 9 2 1 3 4	
-----------	-----------------------	--

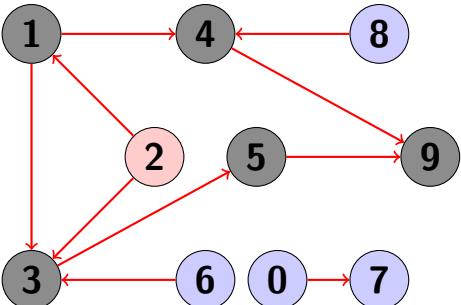
<i>L:</i>	4 3 5 9	
-----------	---------------	--



(j) 9-est már elértek, ezért visszalép a 4-esből. Ez bekerül az *L*-be.

<i>F:</i>	5 9 2 1 3 4	
-----------	-----------------------	--

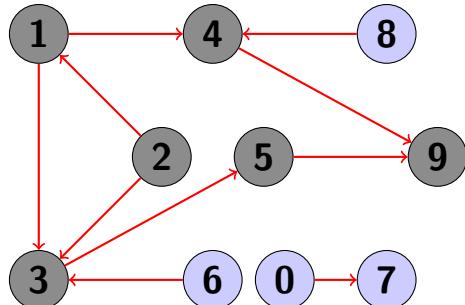
<i>L:</i>	1 4 3 5 9	
-----------	-------------------	--



(k) 1-es szomszéda is fel lették dolgozva, innen is visszalép és bővíti az *L* listát.

<i>F:</i>	5 9 2 1 3 4	
-----------	-----------------------	--

<i>L:</i>	2 1 4 3 5 9	
-----------	-----------------------	--

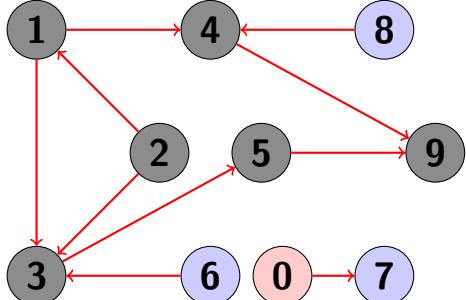


(l) 2-esből még a 3 fele lehetne, de az már benne van az *F*-ben, így ezt is elhagyja.

12.6. ábra. Gráf topologikus rendezése. (folytatás)

F:	5 9 2 1 3 4 0
----	---------------------------

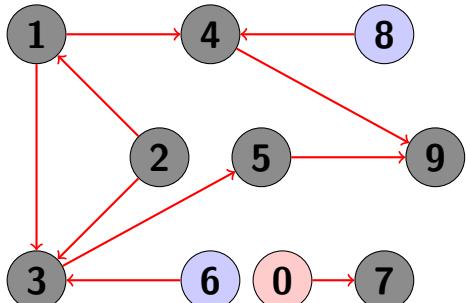
L:	2 1 4 3 5 9
----	-----------------------



(m) Újra indítjuk a rekurziót, pl. a 0-ból.

F:	5 9 2 1 3 4 0 7
----	-------------------------------

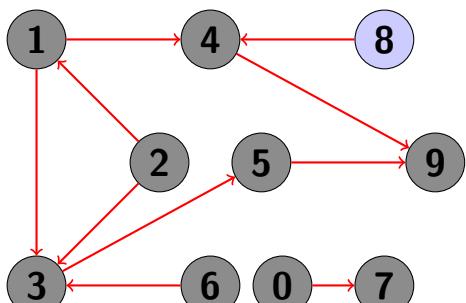
L:	7 2 1 4 3 5 9
----	---------------------------



(o) Nincs további szomszéd, ezért a 7-est elhagyja.

F:	5 9 2 1 3 4 0 7 6
----	-----------------------------------

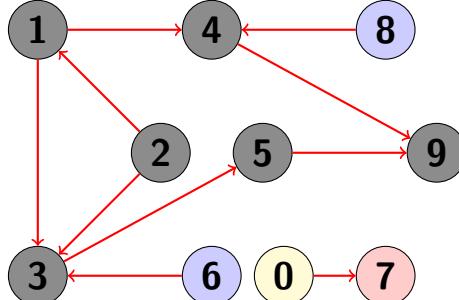
L:	6 0 7 2 1 4 3 5 9
----	-----------------------------------



(q) Elindul a rekurzió 6-ból is, de nem tud továbblépni, egyből ki is lép.

F:	5 9 2 1 3 4 0 7
----	-------------------------------

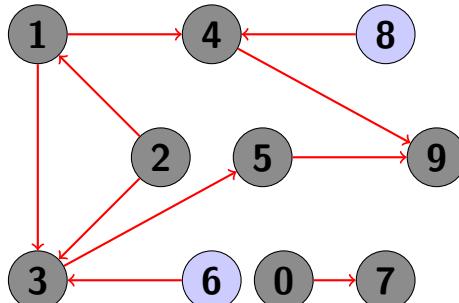
L:	2 1 4 3 5 9
----	-----------------------



(n) Innen tovább tud lépni a 7-be.

F:	5 9 2 1 3 4 0 7
----	-------------------------------

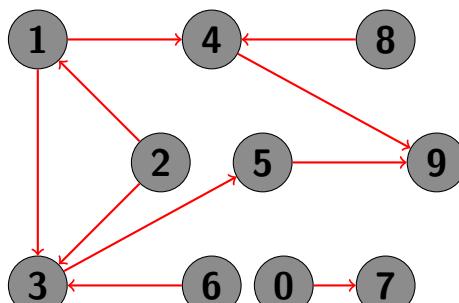
L:	0 7 2 1 4 3 5 9
----	-------------------------------



(p) A 0-nak sincs további szomszédja, ezt is elhagyja és ezt rögzíti az L-ben.

F:	5 9 2 1 3 4 0 7 6 8
----	---------------------------------------

L:	8 6 0 7 2 1 4 3 5 9
----	---------------------------------------



(r) Végül ugyanez történik a 8-as csúccsal is.

12.6. ábra. Gráf topologikus rendezése. (folytatás)

13. fejezet

Súlyozott gráf

13.1. Súlyozott gráf felépítése

Felépítésüket tekintve a súlyozott és súlyozatlan gráfok nem különböznek egymástól lényegesen, ezért a súly eltárolásának módját már a 12. fejezetben tárgyaltuk. Röviden összefoglalva:

- Dinamikus tárolás esetén célszerű magukat az éleket is objektumként létrehozni, és ennek egy mezője lehet az adott él súlya.
- Csúcsmátrixos tárolás esetén magában a mátrixban érdemes eltárolni a súlyokat. Arra persze figyelni kell, hogy a nem létező éleket olyan értékkel jelöljük, amelyek nem keverhetők össze súlyokkal (pl. ha lehetnek 0 súlyú éleink a gráfban, akkor nyilván nem jelölhetjük a mátrixban is 0-val az él hiányát).
- Szomszédsági listában tárolás esetén pedig a lista elemét célszerű kibővíteni még egy tartalmi résszel, amely a csúcspont azonosítója mellett az odavezető él súlyát tárolja.

A *SúlyozottGráf* típus a már meglévő gráf műveletek mellett az alábbiakat kell, hogy megvalósítsa:

- **ÚJÉL(honnan, hova, súly):** Ennek szerepe hasonló, mint a súlyozatlan gráfok **ÚJÉL** művelete, csak ez eltárolja a két csúcspont között létrehozott él súlyát is.
- **SÚLY(honnan, hova) : szám:** Ez pedig visszaadja a paraméterként átadott csúcsok közötti él hosszát. Amennyiben nem vezet a *honnan* csúcsból él a *hova* csúcsba, akkor egy "Nincs él" hibát jelez.

A fenti adatszerkezet és műveleteinek megvalósítását azok egyszerűsége miatt külön nem tárgyaljuk, a meglévő gráf implementáció alapján ez egyszerűen megvalósítható. Nem is magának az adatszerkezet megvalósításának okán tárgyaljuk külön a súlyozott gráfokat, hanem a rajtuk elvégezhető sokkal érdekesebb és komplexebb műveletek miatt:

- Legrövidebb utak keresése: Dijkstra algoritmusa (13.2. fejezet)
- Minimális feszítőfák keresése
 - Prim algoritmusa (13.3.2. fejezet)
 - Kruskal algoritmusa (13.3.3. fejezet)

13.2. Legrövidebb út keresése (Dijkstra algoritmus)

13.2.1. Az alapelve

A szakirodalomban számos gráfokkal kapcsolatos problémára találhatunk algoritmusokat, az egyik ismert és a gyakorlatban jól használható példa a legrövidebb út keresése két vagy akár egyszerre több csúcs között. Ennek egyik megvalósítása Edsger Wybe Dijkstra nevéhez kötődik, ezt a mohó algoritmust vizsgáljuk meg.

Feladatunk egy $g = \text{SúlyozottGráf} < V, E >$ súlyozott gráfban az egyik csúcsból kiinduló legrövidebb utak megkeresése az összes többi csúcsba. Legrövidebb alatt itt most azt az utat értjük, ahol az azt alkotó élek összsúlya minimális. Feltételezzük, hogy a gráf nem tartalmaz negatív súlyú éleket.

Az algoritmus alapelve, hogy egy szélességi bejáráshoz hasonló módszerrel végignézni a kiinduló pontból elérhető összes csúcsot, és minden csúcs esetében eltárolja az oda vezető legrövidebb út hosszát, illetve ennek az útnak az előző elemét (tehát, hogy honnan jutottunk el a kérdéses csúcshoz). A pszeudokódban ehhez két segédváltozót használunk:

- L : Ebben tároljuk el minden egyes csúcshoz a kiinduló ponttól való eddig talált legrövidebb távolságot. Ez célszerűen egy $\text{Szótár} < V \rightarrow \text{szám}^+ >$ adatszerkezet.
- P : Ebben tároljuk el azt az információt, hogy az egyes csúcsokat melyik csúcsból értük el a legrövidebb úttal. Ez lehet egy $\text{Szótár} < V \rightarrow V >$, ahol mind a kulcs, mind pedig a tartalom egy-egy V típusú elem (a kulcs megadja, hogy melyik csúcsról van szó, a tartalom pedig azt, hogy honnan értük el).

Az L leírásában fontos az „eddig talált” megjegyzés, ugyanis a program futása közben ez általában csak egy felső becslést tartalmaz. Tehát bármelyik u csúcs esetén $L[u]$ azt mutatja, hogy egy legfeljebb ilyen hosszú út biztos vezet az u csúcsba, de ez csak egy felső becslés, a későbbiekben ezt még lehet, hogy tudjuk csökkenteni.

Ez a csökkentés pedig az úgynévezett fokozatos közelítés elvén alapul, amely azt mondja ki, hogy ha találunk egy (u, x) élt a g gráfban, amelyre igaz, hogy

$$L[u] + g.\text{SÚLY}(u, x) \leq L[x] \quad (13.1)$$

akkor $L[x]$ értékét csökkenthetjük az alábbiak szerint:

$$L[x] \leftarrow L[u] + g.\text{SÚLY}(u, x) \quad (13.2)$$

Ez különösebb bizonyítás nélkül is érthető, hiszen

- Tudjuk, hogy a kiinduló pontból az u pontba el lehet jutni $L[u]$ hosszú úton, hiszen az L alapján ilyen utat már találtunk.
- Az u -ból létezik egy x -be vezető $g.\text{SÚLY}(u, x)$ súlyú él.
- A fentiekből adódik, hogy az x -be vezet $L[u] + g.\text{SÚLY}(u, x)$ hosszú út a kiinduló pontból.
- Az $L[x]$ -ben a kiinduló pontból az x -be vezető eddig feltételezett legrövidebb út hosszát tároljuk. Ha az előző pontban kiszámolt út hossza ennél kisebb, akkor ezzel felülírható az eddigi $L[x]$ érték.

A fentiekből is látszik, hogy az L futás közben csak egy felső becsléseteket tárol az egyes csúcsokba vezető utak hosszáról, ez menet közben folyamatosan csökkenhet a fenti szabály alkalmazásával.

A kérdés már csak az, hogy az egyes éleket milyen sorrendben vizsgáljuk meg. Egyszerű de számításigényes megoldás, hogy tetszőleges sorrendben végignézünk minden élt és javítjuk az eddigi becsléseteket. Egyszeri ilyen vizsgálat azonban nem elég, ezt egészen addig kell ismételni, amíg nem találunk javítási lehetőséget.

Dijkstra algoritmusa ezen a téren ad egy sokkal hatékonyabb megoldást. Bizonyítható, hogy ha megfelelő sorrendben vizsgáljuk az éleket, akkor elég egyszer átnézni őket. Működése az alábbi:

1. Inicializáláskor a kiindulópontot tekintsük úgy, hogy elérhető, és 0 távolságra van (önmagától). Az összes többi csúcs esetén végtelenre állítjuk a felső becslést. Ezt követően egy ciklusban futtatjuk a következő lépéseket.

2. Fogja a legkisebb távolsági értékkel bíró (L), még nem feldolgozott, de már elérte elemet, legyen ez az u csúcs (az algoritmus indításakor ez nyilván csak a kiinduló pont lehet). Az u -t mostantól feldolgozottnak tekintjük.
3. Megvizsgálja u szomszédait. Amennyiben valamelyik x szomszédhoz talál rövidebb utat, mint amit eddig találtunk (tehát az u -hoz vezető út hossza + u és x közötti él súlya kisebb, mint az eddig talált x -hez vezető út hossza), akkor módosítja x adatait úgy, hogy a L -be felveszi az új legrövidebb út hosszát, az P -be pedig rögzíti, hogy ezt az utat az u -n keresztül értük el.
4. Ezt követően addig ismételjük ezeket a lépéseket, amíg vannak fel nem dolgozott csúcsaink.

Miután lefutott ez a ciklus, bizonyítható [2], hogy az L valóban azt fogja tartalmazni, hogy az egyes csúcsokhoz milyen hosszúságú a legrövidebb út (tehát már nem csak felső becslésekkel beszélünk). Ha pedig magára a csúcshoz vezető útra vagyunk kíváncsiak, akkor azt P -n keresztül vissza tudjuk fejteni: innen látjuk, hogy honnan jutottunk el az utolsó csúcsig, majd megnézhetjük, hogy honnan jutottunk az előző elemig, stb. Ezt addig folytatjuk, amíg el nem jutunk a kezdőcsúcshoz.

Gyakori módosítás, hogy csak két csúcs között keresünk utat. Ilyenkor ugyanezt az algoritmust elindítjuk a kezdőcsúcsból, azonban abban a pillanatban, amikor már a cél csúcsot kapjuk vissza a minimális távolság vizsgálatánál, be is fejezhetjük a keresést. Könnyen belátható, hogy a későbbiekben már biztosan nem fogunk jobb eredményt találni, hiszen a már feldolgozott csúcsokból rövidebb út nem létezik a cél csúcsba (ha lenne, akkor ez az érték szerepelne az L -ben), a nem feldolgozottak távolsága pedig már most is nagyobb (különben nem ezt a csúcsot kaptunk volna minimumként), negatív súlyú élek pedig nincsenek, tehát azokon keresztül sem számíthatunk rövidebb útra.

13.2.2. Dijkstra algoritmus pszeudokódja

A 13.1. algoritmus bemutatja az algoritmus részletes működését. Az első ciklus a felhasznált adatszerkezetek inicializációját végzi. A L szótár fogja tárolni az egyes csúcsokhoz eddig talált legrövidebb út hosszát, ennek minden elemét végtelenre állítjuk, mivel még nincsenek ilyen útjaink. Az P szótár fogja tárolni, hogy az előbbi minimális úton az egyes csúcsokat melyik szomszédjukból értük el, kezdetben itt is minden érték \emptyset lesz, ezzel jelölve, hogy még nem értük el a csúcsokat. Az S prioritási sor kezdetben a gráf összes csúcsát kell, hogy tartalmazza, emiatt ezeket egyesével belehelyezzük.

Az inicializálást követően a 7. sorban beállítjuk a kezdőcsúcsot. Mivel az L az ettől való távolságot fogja minden mutatni, így a kezdőcsúcshoz tartozó értéket kinullázzuk. A kezdőcsúcsnak nincs előző csúcsa, emiatt az P szótárt változatlanul hagyjuk (marad a \emptyset érték).

A 8. sorban látható ciklus az előzőekben leírt módszerrel kezdi el az utak keresését. A ciklus megállási feltétele az, hogy elfogytak az S -ben lévő csúcsok. Egyelőre még a gráf összes csúcsa benne van a prioritási sorban, emiatt belépünk a ciklusba.

Minden ciklus iterációban ki kell vennünk a prioritási sorból a legkisebb távolsággal bíró elemet. Ezt a műveletet végzi el a 9. sor. Végignézi a sorban lévő csúcsokat, mindegyikhez megvizsgálja az L -ben hozzárendelt távolsági értéket, majd visszaadja az (egyk) minimális távolsággal bíró csúcsot.

Tudjuk tehát, hogy az u változóba betöltött csúcsnak már ismerjük a végleges távolságát. Vagy azért, mert a ciklus első iterációjakor az u tulajdonképpen nem más, mint a kezdőelem, így annak a távolsága 0, vagy pedig a fentiekben már ismertetett szabályok alapján. Ezt követően a belső ciklussal (10. sor) megvizsgáljuk ennek a csúcsnak a szomszédait (a ciklus futása közben ezekre minden x -el hivatkozunk), és beállítjuk azoknak is a távolsági adatait.

A 11. sorban lévő vizsgálat ennek a fő lépése, ugyanis itt ellenőrizzük, hogy az u csúcson keresztül elérve az x csúcsot rövidebb utat találunk-e, mint az eddig talált legrövidebb út. Ha még az eddigiek során nem értük el az x -et, az sem probléma, hiszen ilyenkor a $L[x]$ értéke az inicializáláskor megadott végtelen, tehát biztos, hogy ennél csak rövidebb utat találhatunk a későbbiekben.

Amennyiben azt találjuk, hogy az új út nem rövidebb, mint a régi, akkor nincs további teendőnk ezzel a szomszéddal. Azonban ha rövidebb utat találunk, akkor ezt a tény rögzítenünk kell. Egyrészt módosítjuk az L értékét (12. sor), hogy az valóban helyesen mutassa, hogy mi az eddig talált legrövidebb út hossza. Másrészt pedig módosítanunk kell a P értékét is (13. sor), hogy ezt a legrövidebb utat követve melyik csúcson keresztül értük el ezt a csúcsot.

Miután átvizsgáltuk az összes szomszédöt, akkor az u elem feldolgozása már véget is ért, a későbbiekben nem térünk vissza rá. A külső ciklus újabb iterációja betölti a következő minimális távolsággal rendelkező csúcsot, és folytatja annak a feldolgozását.

13.1. Algoritmus Dijkstra algoritmusa

Bemenet: g - SúlyozottGráf $\langle V, E \rangle$ (a gráf, amiben utakat keresünk)

Bemenet: $start$ - V (a kiinduló csúcs)

Kimenet: L - Szótár $\langle V \rightarrow szám^+ \rangle$ (a legrövidebb utak hossza)

Kimenet: P - Szótár $\langle V \rightarrow V \rangle$ (a legrövidebb utak előző eleme)

```
1: függvény DIJKSTRA( $g, start$ )
2:   ciklus  $\forall x \in g.Csúcsok$ 
3:      $L[x] \leftarrow \infty$                                 ▷ Távolság végtelen
4:      $P[x] \leftarrow \emptyset$                             ▷ Nincs még előző elem
5:      $S \leftarrow x$                                     ▷ Csúcs a prioritási sorba
6:   ciklus vége
7:    $L[start] \leftarrow 0$                                 ▷ Lehet, hogy S-t frissíteni kell
8:   ciklus amíg  $S \neq \emptyset$ 
9:      $u \xleftarrow{\min} S$                                 ▷ Legkisebb távolságú elem kivétele
10:    ciklus  $\forall x \in g.SZOMSZÉDAI(u)$ 
11:      ha  $L[u] + g.SÚLY(u, x) < L[x]$  akkor
12:         $L[x] \leftarrow L[u] + g.SÚLY(u, x)$                 ▷ Lehet, hogy S-t frissíteni kell
13:         $P[x] \leftarrow u$ 
14:    elágazás vége
15:  ciklus vége
16:  ciklus vége
17:  vissza ( $L, P$ )
18: függvény vége
```

Felhasznált változók és függvények

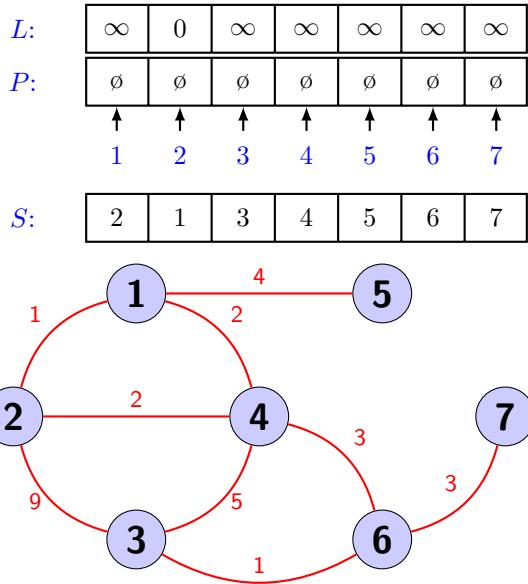
- L : Egy Szótár $\langle V \rightarrow V \rangle$, amelyik egy felső becslést tárol az egyes csúcsokba vezető legrövidebb út hosszáról.
 - P : Egy Szótár $\langle V \rightarrow szám^+ \rangle$, amelyik csúcsonként a legrövidebb talált úthoz tartozó előző csúcsot tárolja.
 - S : Egy PrioritásosSor $\langle V \rangle$, amely a gráf csúcsait tárolja, rendezettségének alapja pedig az L szótárban tárolt értékek.
 - $\xleftarrow{\min} S$: Az S prioritási sorból visszaadja azt a csúcsot, amelyhez tartozó L érték minimális. Egyben törli is ezt az elemet a sorból.
 - u, x : V típusú segédváltozók.
-

Amikor feldolgoztunk minden csúcsot, illetve azok szomszédait, az algoritmus futása véget ér. Az eredményeket a L és az P változók tartalmazzák. Ez a két táblázat a függvény visszatérési értéke, bár magához az út kereséshez elég lehet a P is. Hogy pontosan hogyan is tudjuk felhasználni az ebben lévő adatokat, arra hamarosan visszatérünk, előtte nézzük meg az algoritmus működését a gyakorlatban.

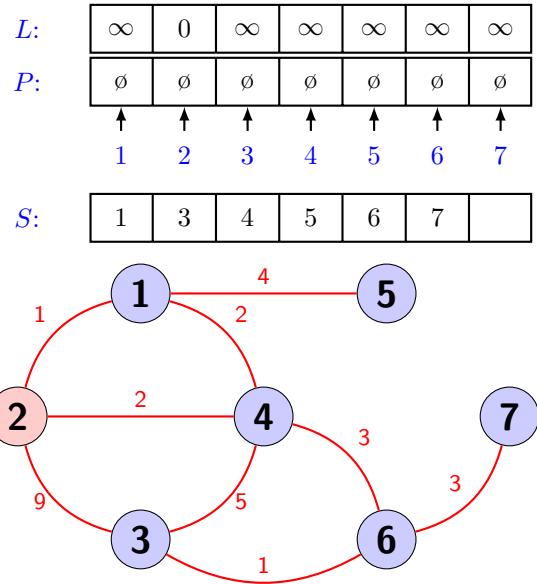
A 13.1. ábrán egy példát látunk, ahol egy irányítlan sílyozott gráfban próbálunk legrövidebb utakat keresni. A kiindulópont a 2-es csúcs.

Megjegyzés

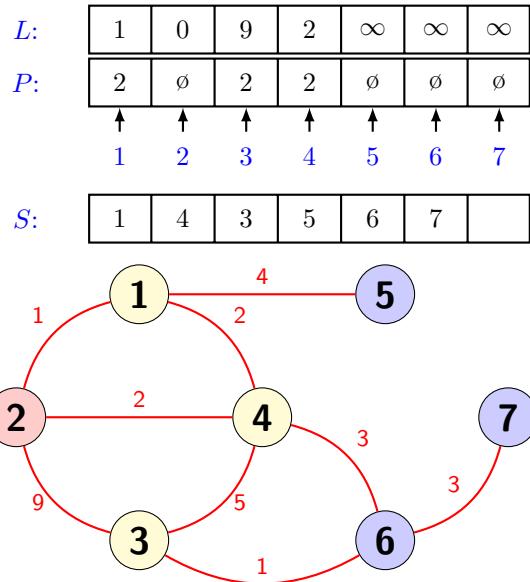
A prioritásos sor az L értékei alapján határozza meg az elemek sorrendjét, ez az érték azonban változhat a sorba való beszúrás után. Nem minden megvalósítás ad erre lehetőséget, lehet olyan megvalósítás, ahol ez gond nélkül működik, az általunk készített implementációk esetében pedig [FRISSÍT](#) metódus végzi el az ilyenkor szükséges karbantartást. Ez azonban implementációtól függő, ezért a pszeudokódba ezt nem jelöljük.



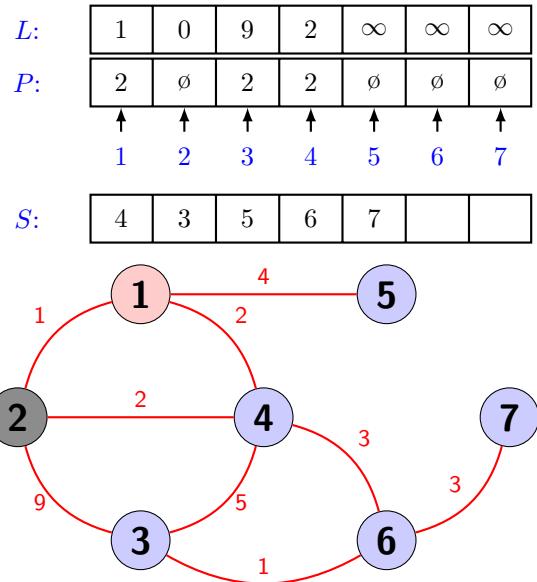
(a) Közvetlenül az inicializálást követő állapot. A kezdőcsúcs távolsága 0, megelőzője pedig \emptyset .



(b) A prioritási sorból kiveszük azt az elemet, amelyikhez a legkisebb *L*-beli érték tartozik. Ez a 2-es elem, ezzel kezdődik a feldolgozás.



(c) Megvizsgáljuk a 2-es elem három szomszédját. Távolságunk nem más, mint a 2-estől induló él súlya. Mivel ez mindenkor kisebb lesz, mint az inicializáláskor felvett végtelen érték, így ezzel felülírjuk az *L*-ben lévő értékeket, illetve beállítjuk az *P* megfelelő mezőit is, hogy honnan találtuk a legrövidebb utat.

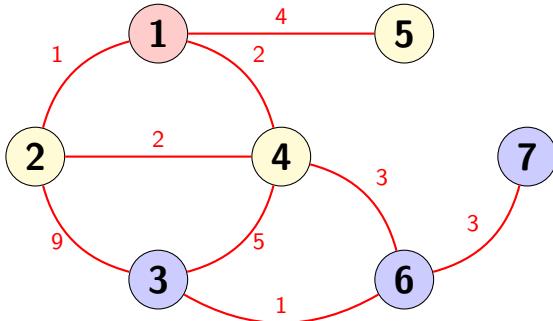


(d) A 2-es elemet ezzel feldolgozottnak tekintjük. A prioritási sorból ismét kiveszük azt, amelyik a legkisebb *L*-beli értékkal bír. Ez most az 1-es.

13.1. ábra. Dijkstra algoritmus működése.

<i>L:</i>	1	0	9	2	5	∞	∞
<i>P:</i>	2	\emptyset	2	2	1	\emptyset	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

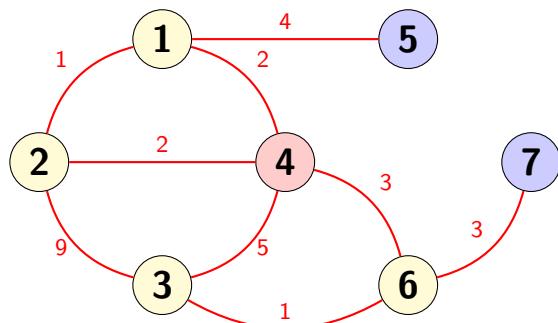
<i>S:</i>	4	5	3	6	7		
-----------	---	---	---	---	---	--	--



(e) Megvizsgáljuk ennek a szomszédait, hátha az egyeshez vezető út hossza (1) + az egyesből vezető él súlya kisebb lesz, mint az eddig talált legrövidebb út. Az 5-ös elem esetében ez igaz, hiszen oda még nem találtunk utat. A 4-es esetében ez nem igaz, illetve a 2-es esetében értelemszerűen szintén nem lehet igaz.

<i>L:</i>	1	0	7	2	5	5	∞
<i>P:</i>	2	\emptyset	4	2	1	4	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

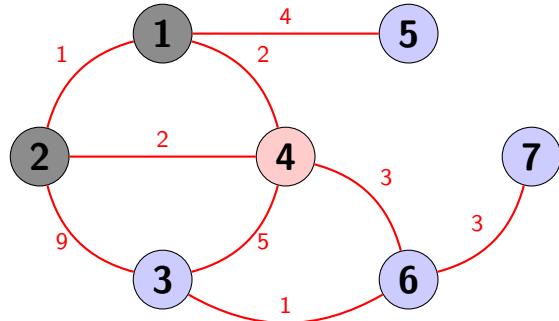
<i>S:</i>	5	6	3	7			
-----------	---	---	---	---	--	--	--



(g) A 4-es szomszédait is átvizsgáljuk, hogy találtunk-e a 4-esen keresztül haladó rövidebb utat, mint az eddigi legrövidebb. Az 1 és 2 esetében ez nincs így. A 3-as csúcsba így már 7 hosszú úton is eljuthatunk, tehát ez jobb mint az eddigi 9, emiatt cserélünk. A 6-osba pedig ez az első út amit találtunk.

<i>L:</i>	1	0	9	2	5	∞	∞
<i>P:</i>	2	\emptyset	2	2	1	\emptyset	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

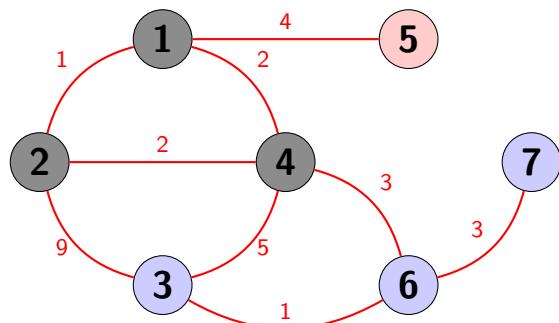
<i>S:</i>	5	3	6	7			
-----------	---	---	---	---	--	--	--



(f) Az 1-est is feldolgozzunk tekinthetjük. Ismét ki kell venni a prioritási sorból a legkisebb *L* értékkel bíró értéket. Jelenleg ez a 4-es.

<i>L:</i>	1	0	7	2	5	5	∞
<i>P:</i>	2	\emptyset	4	2	1	4	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

<i>S:</i>	6	3	7				
-----------	---	---	---	--	--	--	--

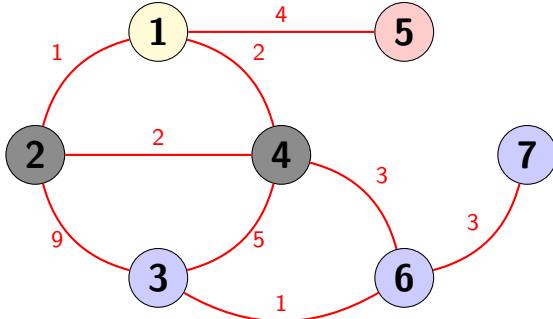


(h) A 4-es elemet ezzel feldolgozzunk tekintjük. A prioritási sorból ismét kiveszük azt, amelyik a legkisebb *L*-beli értékkel bír. Ez most az 5-ös vagy 6-os közül bármelyik lehet. Legyen az 5-ös.

13.1. ábra. Dijkstra algoritmus működése.(folytatás)

<i>L:</i>	1	0	7	2	5	5	∞
<i>P:</i>	2	\emptyset	4	2	1	4	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

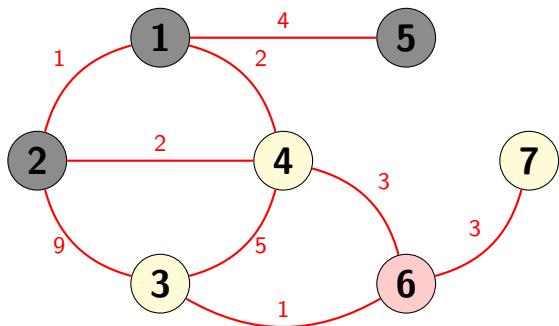
<i>S:</i>	6	3	7				
-----------	---	---	---	--	--	--	--



(i) Az 5-ösnek csak egy szomszédja van, és ide se találtunk jobb utat, így nem változik semmi.

<i>L:</i>	1	0	6	2	5	5	8
<i>P:</i>	2	\emptyset	6	2	1	4	6
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

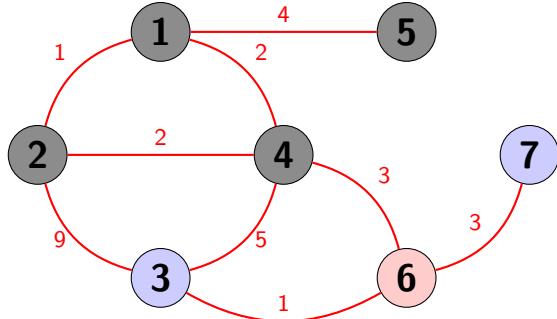
<i>S:</i>	3	7					
-----------	---	---	--	--	--	--	--



(k) Átvizsgáljuk a 6-os szomszédeit, hátha rövidebb a 6-osba eddig talált legrövidebb út (hossza 5) + a hatosból induló él hossza. A 3-as és a 7-es esetében ez igaz, ezért itt módosítjuk az *L* és az *P* értékeit.

<i>L:</i>	1	0	7	2	5	5	∞
<i>P:</i>	2	\emptyset	4	2	1	4	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

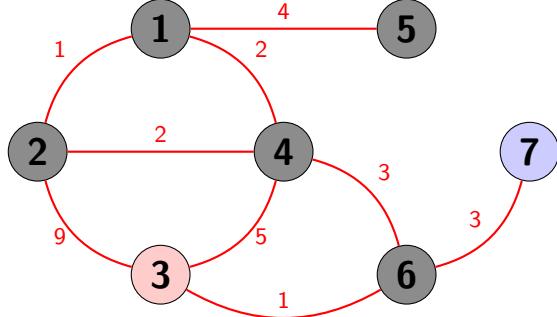
<i>S:</i>	3	7					
-----------	---	---	--	--	--	--	--



(j) Az 5-öst is feldolgozottnak tekinthetjük. Ismét ki kell venni a prioritási sorból a legkisebb *L* értékkel bíró értéket. Jelenleg ez a 6-os.

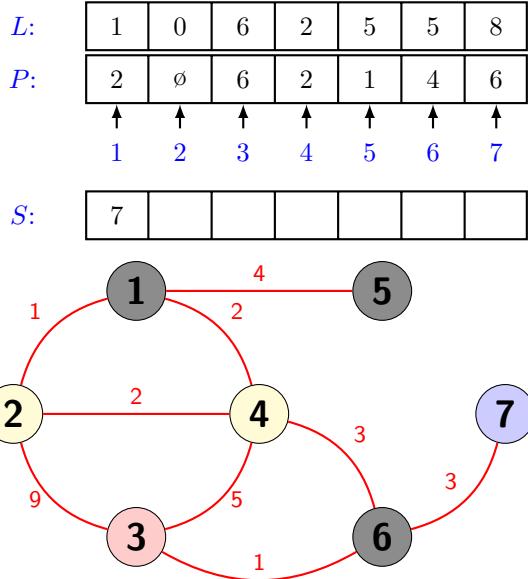
<i>L:</i>	1	0	6	2	5	5	8
<i>P:</i>	2	\emptyset	6	2	1	4	6
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

<i>S:</i>	7						
-----------	---	--	--	--	--	--	--

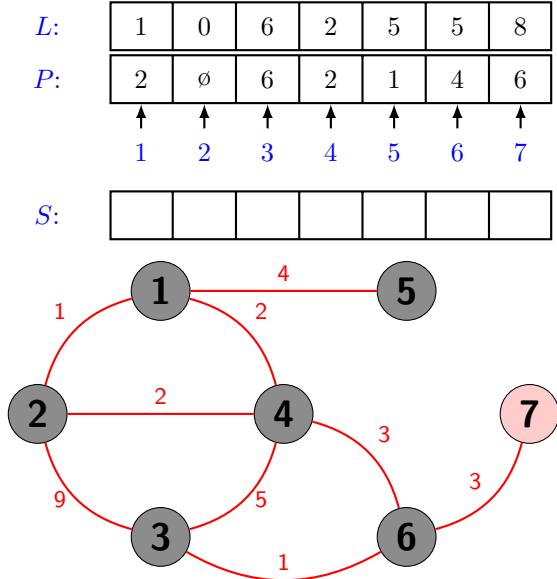


(l) A prioritási sorban már csak a 3-es és 7-es van. Ezek közül a 3-asnak kisebb az *L*-beli értéke, emiatt ez a következő.

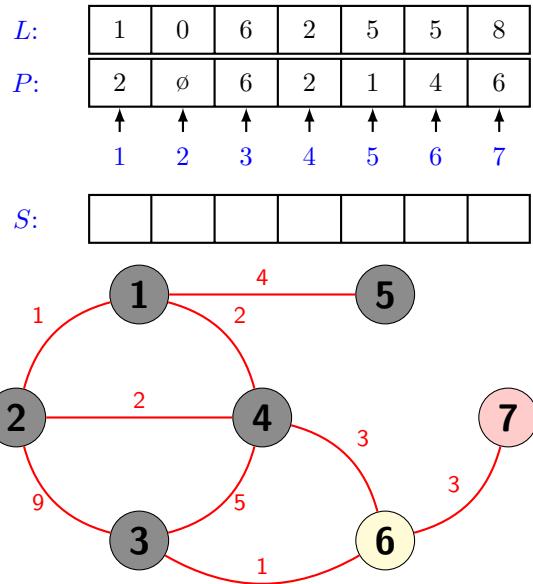
13.1. ábra. Dijkstra algoritmus működése.(folytatás)



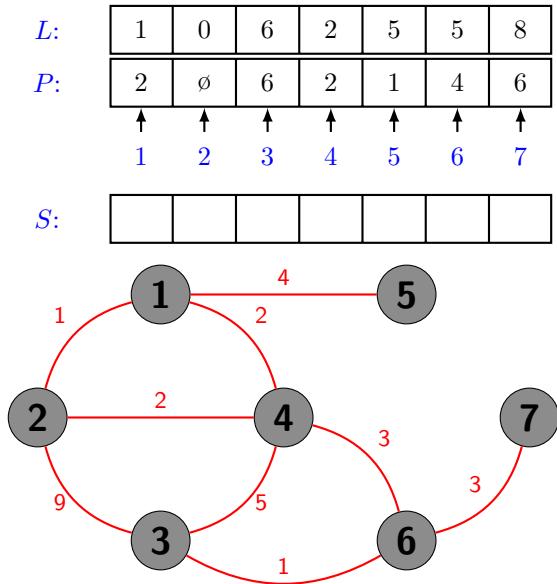
(m) Az 3-as szomszédai között nem találunk olyat, ahol javítani tudnánk a legrövidebb út hosszán.



(n) A prioritási sorban már csak a 7-es van, így ezt fogjuk feldolgozni.



(o) Mivel már mindenki mást feldolgoztunk, így értelemszerűen itt se találunk jobb értéket a szomszéd(ok) között.



(p) A sor üres, tehát az algoritmus futása végetért. A kimeneti adatok az *P* táblázatban vannak.

13.1. ábra. Dijkstra algoritmus működése.(folytatás)

13.2.3. Dijkstra algoritmus kimenetének feldolgozása

Habár a kiinduló feladatunk csúcsok közötti utak keresése volt, láthatjuk, hogy az előzőleg megismert Dijkstra algoritmus visszatérési értéke még nem pontosan ez. A visszatérési érték minden csúcsra egy szótár, ahol azt tároljuk, hogy az egyes csúcsokba honnan érkeztünk a legrövidebb úton keresztül, illetve egy másik, ami a távolságokat tárolja. Ezek segítségével viszont már egyszerűen tudunk válaszolni az időközben felmerülő kérdésekre, pl.:

- Egy csúcsból melyik a legrövidebb út egy másik csúcsba?
- Egy csúcsból melyek a legrövidebb utak az összes elérhető csúcsba?
- Milyen messze van egy csúcs a kiinduló csúcstól?
- Milyen messze vannak az egyes csúcsok a kiinduló csúcstól?

A kiinduló pont tehát az lesz, amelyiket paraméterként áadtunk a Dijkstra algoritmusnak, a kérdés csak az, hogy melyik csúcs felé és melyik információra van szükségünk (maga az út, vagy pedig csak annak a hossza érdekes). A hossz ügyében nincs további tennivalónk, egyszerűen ki kell olvasnunk a szükséges értéket az L táblázatból. Ennek értéke az alábbi lehet:

- ∞ : azok a csúcsok tartalmazzák ezt az értéket, amelyeket nem sikerült elérni az algoritmus futása során, tehát ezekbe nem vezet út a kiinduló csúcsból.
- 0: tipikusan a kezdőcsúcs értéke (persze más csúcsé is lehet, ha megengedjük a 0 súlyú éleket).
- Egyéb szám: a megadott csúcs távolsága a kiinduló csúcstól (tehát a hozzá vezető legrövidebb úton az élek súlyainak összege).

Érdekesebb kérdés az, ha magára a csúcshoz vezető útra van szükségünk. Ezt az utat az P táblázat alapján tudjuk előállítani. Ez minden csúcs esetében azt tárolja, hogy a hozzá vezető legrövidebb úton melyik volt az előző csúcs. Ennek megfelelően magát az utat is visszafelé tudjuk előállítani az alábbi lépésekkel követve (a *hova* csúcshoz vezető utat keressük):

1. Töltsük be az x változóba az $P[hova]$ értékét. Ez visszaadja azt a csúcsot, amelyből a *hova* csúcsba tudunk majd jutni.
2. Mentsük el az x értékét, majd cseréljük ki a $P[x]$ értékével. Ezzel megkapjuk az aktuális x előtti csúcsot.
3. Addig ismételjük a fentieket, amíg az x értéke nem lesz \emptyset , ez ugyanis csak a kezdőcsúcsra lehet igaz (az el nem ért csúcsoknál is ez az érték van az P -ben, de azokhoz nem juthatunk el ilyen módon).

A fenti módszerrel megkapjuk az utat alkotó csúcsokat visszafelé. Szükség esetén ezt a sorozatot meg kell fordítani, és ezzel megkapjuk a keresett utat. Ezt mutatja be a 13.2. algoritmus. Látható, hogy az utólagos megfordítás helyett egy láncolt listának az elejére beszűrását használjuk, ami ugyanazt eredményezi.

Amennyiben az összes útra van szükségünk, akkor ezt a műveletet kell megismételnünk az összes csúcsra.

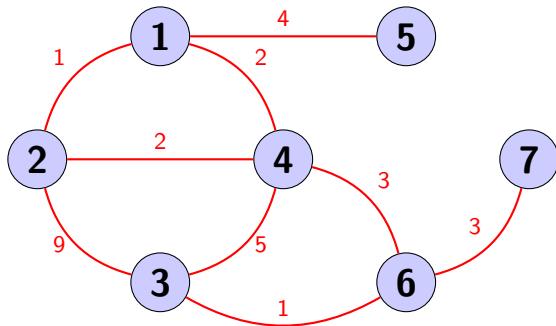
Két csúcs közötti út keresésre (az előzőleg is feldolgozott gráfban a 2-es és a 3-as csúcs között keressük a legrövidebb utat) mutat egy példát a 13.2. ábra.

1	0	6	2	5	5	8
2	\emptyset	6	2	1	4	6

↑ ↑ ↑ ↑ ↑ ↑ ↑

1 2 3 4 5 6 7

fej = \emptyset



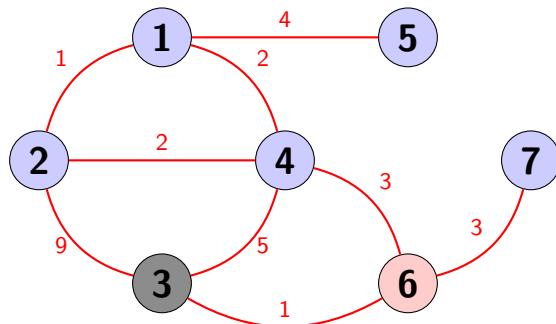
(a) Az algoritmus lefuttatja az előzőleg megismert Dijkstra függvényt. Mivel ugyanaz a gráf és a kiindulási pont, így a kimenet is ugyanaz marad. A fejmutató kezdetben egy üres láncolt listát jelez csak.

<i>L:</i>	1	0	6	2	5	5	8
<i>P:</i>	2	\emptyset	6	2	1	4	6

↑ ↑ ↑ ↑ ↑ ↑ ↑

1 2 3 4 5 6 7

fej → $[6] \rightarrow [3] \emptyset$



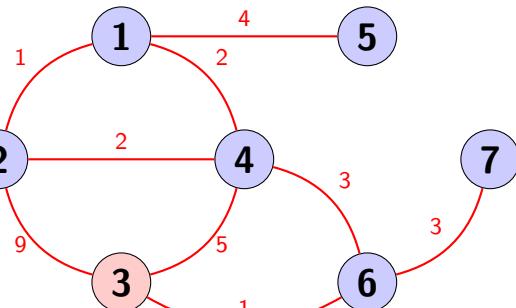
(c) Az x -be betöltjük az $P[3]$ értékét, ami jelenleg 6. Ez egyben bekerül a láncolt listába is.

<i>L:</i>	1	0	6	2	5	5	8
<i>P:</i>	2	\emptyset	6	2	1	4	6

↑ ↑ ↑ ↑ ↑ ↑ ↑

1 2 3 4 5 6 7

fej → $[3] \emptyset$



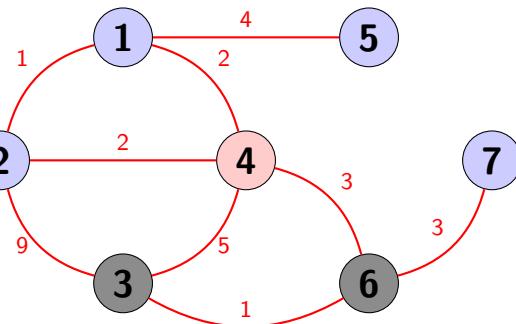
(b) Az x -be betöltjük a végcsúcson, ami jelen esetben 3-as. Az első ciklus iterációval ez azonnal bekerül a láncolt listába is.

<i>L:</i>	1	0	6	2	5	5	8
<i>P:</i>	2	\emptyset	6	2	1	4	6

↑ ↑ ↑ ↑ ↑ ↑ ↑

1 2 3 4 5 6 7

fej → $[4] \rightarrow [6] \rightarrow [3] \emptyset$



(d) Az x -be ismét bekerül az $P[x]$ értéke, ami jelenleg 4. Ez is bekerül a láncolt lista elejére.

13.2. ábra. Két csúcs közötti legrövidebb út keresése.

13.2. Algoritmus Legrövidebb út két csúcs között

Bemenet: g - SúlyozottGráf $\langle V, E \rangle$ (a gráf, amelyikben az utat keressük)

Bemenet: honnan - V (a kiinduló csúcs)

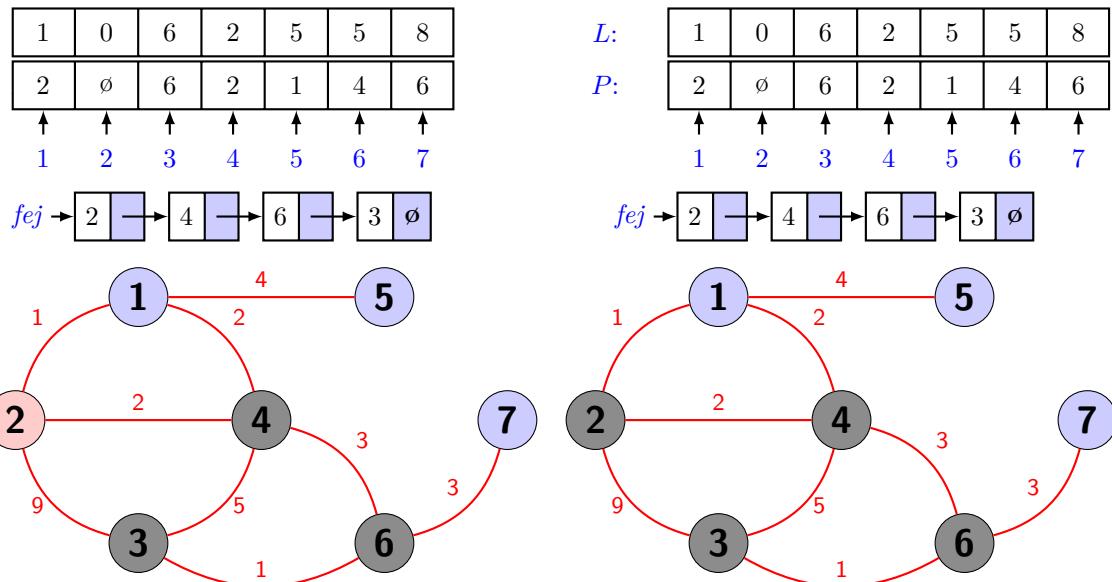
Bemenet: hova - V (a cél csúcs)

Kimenet: . - Lista $\langle V \rangle$ (tartalmazza a legrövidebb út csúcsait)

- 1: függvény LEGRÖVIDEBBÚT(g , honnan, hova)
- 2: $(L, P) \leftarrow$ DIJKSTRA(g , honnan)
- 3: $u \leftarrow \emptyset$
- 4: $x \leftarrow$ hova
- 5: ciklus amíg $x \neq \emptyset$
- 6: $u.BEZSÚR(1, x)$
- 7: $x \leftarrow P[x]$
- 8: ciklus vége
- 9: vissza u
- 10: függvény vége

Felhasznált változók és függvények

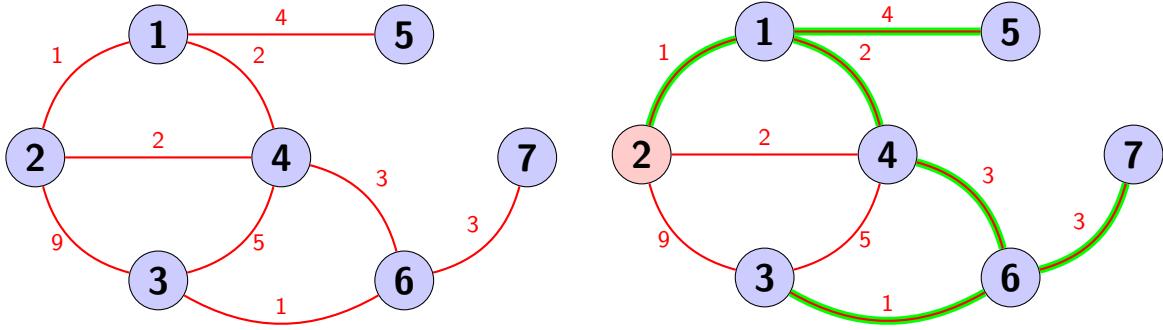
- $x : V$ típusú segédváltozó.
- $L, P : A Dijkstra algoritmus (13.1. algoritmus) által visszaadott szótárak.$
- $u : Egy$ Lista $\langle V \rangle$ ami tárolja az út pontjait. Célszerűen láncolt megvalósítással.



(e) A P segítségével ismét eggyel visszább lépünk, az x értéke most 2. Bekerül a láncolt listába.

(f) A ciklus következő iterációjánál az x értéke \emptyset lesz, emiatt véget ér az eljárás. Visszatérési értéke a fej által mutatt láncolt lista, ami valóban a keresett legrövidebb utat tartalmazza.

13.2. ábra. Két csúcs közötti legrövidebb út keresése.(folytatás)



(a) Az eredeti gráf láthatóan nem fa.

(b) A zölddel jelölt élek alkotják a minimális feszítőfát.

13.3. ábra. Egy minta gráf, és annak egyik minimális feszítőfája

13.3. Minimális feszítőfa keresése

13.3.1. Minimális feszítőfa fogalma

Egy $g = (E, V)$ gráf minimális feszítőfájának nevezzük azt a $g' = (E', V')$ gráfot, amelynek a csúcsai ugyanazok mint az eredetinek (E), éleinek halmaza az eredeti gráf éleinek halmazának egy részhalmaza ($V' \subseteq V$), emellett pedig összefüggő és körmentes (tehát fa). Ez úgy képzelhető el, hogy az eredeti összefüggő gráfnak elkezdjük elhagyni az éleit úgy, hogy az összefüggő maradjon, de ne maradjon benne kör.

Mivel egy gráfnak több ilyen feszítőfája is lehet, ezért minimális feszítőfák alatt ezen fák közül a minimális összsúlyúakat értjük (tehát ahol az E' -ben lévő élek súlyának összege minimális). Belátható, hogy minimális feszítőfából is lehet több.

A minimális feszítőfákkal megoldható problémák kevésbé kézenfekvőek mint az útkeresés esetén, azonban így is fontosak számunkra:

- Képzeljük el, hogy meg kell terveznünk egy épület számítógép-hálózatát. A probléma felírható egy gráf segítségével, ahol a csúcspontok az egyes munkaállomásokat/elosztókat képviselik, az élek pedig azt mutatják, hogy melyik pontokat tudjuk fizikailag összekötni, azok súlya pedig ennek költségét mutatja. Több száz gép esetén egy minimális feszítőfa egy jó javaslatot ad a bekötésre, hiszen az összefüggő (tehát közvetlenül vagy közvetve minden gép mindegyikkel össze van kötve), a bekötési költség viszont minimális.
- Tegyük fel, hogy egy osztály tanulói között szeretnénk biztosítani, hogy mindenki megkapja a házi feladattal kapcsolatos információkat. Ha az egyes tanulók a gráf csúcsai, az élek mutatják, hogy kik állnak egymással kapcsolatban, az él súlya mutatja, hogy mennyire kedvelik egymást, akkor egy minimális feszítőfa tud adni egy olyan hálózatot, ahol mindenki el tud juttatni információt mindenkihez a lehető legkisebb költséggel.
- A fentiek mellett számos példát lehet még hozni az áramköri elemek áramellátásától kezdve további problémáig.

A 13.3. ábra egy gráfot és annak egyik lehetséges minimális feszítőfáját mutatja. Az is látható, hogy több ilyen is létezik, hiszen az (1, 4) él helyett a (2, 4) élt választva is ugyanilyen összsúlyú fához jutunk.

A fejezetben két egyszerű algoritmust vizsgálunk meg, amelyek meg tudják adni egy összefüggő gráf minimális feszítőfáját. Mindkettő mohó algoritmus, azonban minden esetben bizonyítható, hogy helyes eredményt adnak. A bizonyítással nem foglalkozunk, az a szakirodalomban megtalálható [2]. Ezt érdemes lehet átnézni, mert érdekes, hogy a két egymástól nagyon különbözőnek tűnő módszer valójában ugyanazon az elven nyugszik.

13.3.2. Prim algoritmusa

A Prim által javasolt módszer ránézésre nagyon hasonlít a Dijkstra féle algoritmusra. A fő különbség a 11. sorban látható, ahol nem a már megismert „fokozatos közelítés elve” látható. Itt ugyanis nem az u csúcsba vezető úthoz adjuk hozzá az innen x -be vezető él hosszát, hanem csak magát az él hosszúságot figyeljük. Ez persze logikus, hiszen itt nem egy legrövidebb utat keresünk az egyes csúcsokba, csak az a cél, hogy azokat minél kisebb súlyú élen keresztül tudjuk bekötni a folyamatosan épített feszítőfához.

Az algoritmus működése során egy K szótárban tároljuk, hogy az egyes csúcsokat milyen súlyú éssel tudtuk bekötni a feszítő fába. Az algoritmus futása során ez egy felső becslés, ennek megfelelően kezdetben végtelen hosszúságot adunk meg, a későbbiekben ezt fokozatosan csökkentjük majd akkor, amikor látunk egy kedvezőbb bekötési lehetőséget. A P szótár szerepe, hogy ez tárolja azt, hogy az egyes csúcsokat melyik másik csúcson keresztül kötöttük be a leendő feszítőfába.

Az S prioritási sor pedig itt is az egyes csomópontokat fogja tárolni. A rendezés alapja az egyes csúcsokhoz tartozó K -beli érték, tehát a prioritási sor minden azzal fogja visszaadni, amelyik még benne van az S -ben, és az ott lévők közül a lehető legkisebb súlyú éssel lett bekötve a gráfba.

Kiválasztunk egy tetszőleges $start$ elemet, innen fog elindulni az algoritmus. Különböző induló elemekhez különböző eredmények tartozhatnak, de minden esetben a helyes minimális feszítőfát fogunk kapni eredményül. Ezért a $start$ választás tetszőleges.

Ezt követi az algoritmus fő ciklusa, amely az alábbi lépésekkel hajta végre:

1. Kiveszi a prioritási sorból a legkisebb súlyú éssel bekötött csúcsot (9).
2. Megnézi ennek minden szomszédját. Amennyiben az adott x szomszédra igaz, hogy
 - még nem vettük ki a sorból (tehát még nem véglegesítettük a bekötését), és
 - az u csomópontron keresztül kisebb súlyú éssel tudjuk bekötni a feszítőfába ($g.SÚLY(u,x)$) mint az eddigi felső becslésünk ($K[x]$),akkor innentől úgy tekintjük, hogy az x -en keresztül célszerűbb őt bekötni a feszítőfába. Ennek megfelelően az új felső becslést rögzítjük a K -ban, illetve a bekötési irányt a P -ben.
3. A ciklust addig folytatjuk, amíg vannak elemek a sorban, tehát minden csúcs minden szomszédját pontosan egyszer fogjuk megvizsgálni (ami pontosan megfelel az élek számának).

A módszer kulcsa, hogy megfelelő sorrendben nézi a csúcsokat, ennek köszönhetően a ciklus végére a P alapján már elő tudjuk állítani a leendő feszítőfát. Ehhez a $start$ elem kivételével minden x csúchoz elő kell venni a ($P[x]$, x) élt (ilyen biztosan van), és ezek lesznek a minimális feszítőfa élei. A fa csúcsai pedig értelemszerűen az eredeti gráf csúcsaival egyeznek meg.

13.3.3. Kruskal algoritmusa

Kruskal egy másik mohó algoritmust dolgozott ki, amelyik ugyanehhez a végeredményhez vezet, mégha a működése egészen különböző is. Ez eleve nem is a csúcsokból indul ki, hanem az éleket dolgozza fel egymás után. Alapötlete az, hogy sorbarendezni az éleket súly szerint, majd ezekkel elkezd bővíteni egy kezdetben üres gráfot. A bővítés során pedig csak arra ügyel, hogy az új él felvétele ne hozzon létre kört, hiszen azzal megszegné a fával kapcsolatos elvárásokat. Az összes él feldolgozása után könnyen belátható, hogy egy összefüggő fát fogunk kapni, az pedig bizonyítható [2], hogy ez egy minimális feszítőfa lesz.

A 13.4. algoritmus érdekessége az, hogy milyen módon tudjuk ellenőrizni, hogy egy új él behúzásával okozunk-e kört vagy pedig sem. Ha minden éssel való bővítés előtt egy kör keresést kellene futtatnunk a gráfban, az meglehetősen erőforrásigényes lenne. Az alábbi ötletet követve azonban ez nagyon egyszerűen megoldható: a folyamatosan épített feszítőfa komponenseit helyezzük el különböző halmazokban. Amikor egy új élt akarunk felvenni, akkor minden ellenőrizzük, hogy annak kezdő- és végpontja ugyanabban a halmazban van-e. Ha igen, akkor az azt jelenti, hogy egy már összefüggő gráfba akarunk felvenni még egy élt, ami biztosan kört hozna létre. Ha különböző halmazban vannak, akkor az nem okoz ilyen problémát, ezek az élek nyugodtan felvehetők a feszítőfába.

Kiinduló állapotnak egy olyan gráfot tekintünk, aminek csúcsai megegyeznek az eredeti gráf csúcsaival, és nincsenek élei. Ez biztosan a leendő minimális feszítőfa egy részgráfja. Ennek az egyes csúcsait külön-külön komponenseknek tekinthetjük, ezért ezeket elhelyezzük egy-egy különálló halmazban. Ezeket a halmazokat hozza létre az első ciklus (3. sor).

13.3. Algoritmus Prim algoritmusa

Bemenet: g - SúlyozottGráf $\langle V, E \rangle$ (a gráf, aminek a feszítőfáját keressük)

Bemenet: $start$ - V (egy tetszőleges kezdőpont)

Kimenet: P - Szótár $\langle V \rightarrow V \rangle$ (tartalmazza a minimális feszítőfa élét)

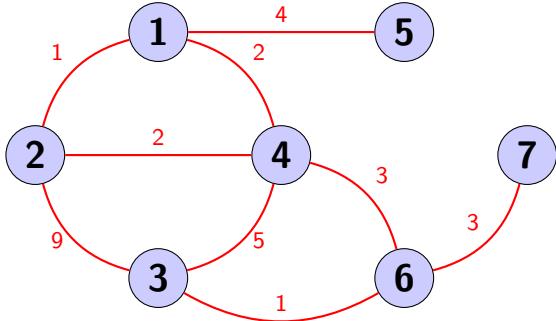
```
1: függvény PRIMFESZÍTŐFAKERESÉS( $g, start$ )
2:   ciklus  $\forall x \in g.Csúcsok$ 
3:      $K[x] \leftarrow \infty$                                 ▷ Bekötő él végtelen
4:      $P[x] \leftarrow \emptyset$                             ▷ Nincs még előző elem
5:      $S \leftarrow x$                                     ▷ Csúcs a prioritási sorba
6:   ciklus vége
7:    $K[start] \leftarrow 0$                                 ▷ Lehet, hogy  $S$ -t frissíteni kell
8:   ciklus amíg  $S \neq \emptyset$ 
9:      $u \xleftarrow{\min} S$                                 ▷ Legkisebb távolságú elem kivétele
10:    ciklus  $\forall x \in g.SZOMSZÉDAI(u)$ 
11:      ha  $x \in S \wedge g.SÚLY(u, x) < K[x]$  akkor
12:         $K[x] \leftarrow g.SÚLY(u, x)$                       ▷ Lehet, hogy  $S$ -t frissíteni kell
13:         $P[x] \leftarrow u$ 
14:    elágazás vége
15:    ciklus vége
16:  ciklus vége
17:  vissza  $P$ 
18: függvény vége
```

Felhasznált változók és függvények

- K : Egy Szótár $\langle V \rightarrow V \rangle$, amelyik egy felső becslést tárol arra, hogy az egyes csúcsokat milyen súlyú élen át lehet bekötni az épülő feszítőfába.
 - P : Egy Szótár $\langle V \rightarrow V \rangle$, amelyik csúcsonként megadja, hogy melyik másik csúcson át lehetően bekötni a feszítőfába.
 - S : Egy PrioritásosSor $\langle V \rangle$, amely a gráf csúcsait tárolja, rendezettségének alapja pedig a K szótárban tárolt értékek.
 - $\xleftarrow{\min} S$: Az S prioritási sorból visszaadja azt a csúcsot, amelyhez tartozó L érték minimális. Egyben törli is ezt az elemet a sorból.
 - u, x : V típusú segédváltozók.
-

<i>K:</i>	∞	0	∞	∞	∞	∞	∞
<i>P:</i>	\emptyset						
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

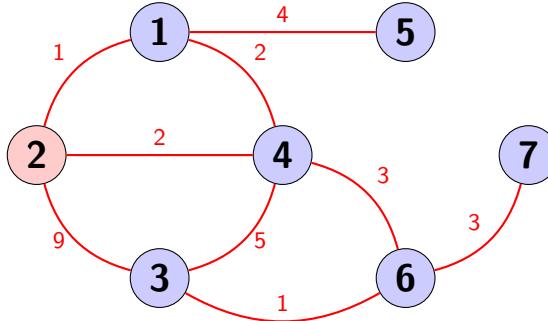
<i>S:</i>	2	1	3	4	5	6	7
-----------	---	---	---	---	---	---	---



(a) Közvetlenül az inicializálást követő állapot. A kezdőcsúcs *K* értéke 0, megelőzője pedig \emptyset .

<i>K:</i>	∞	0	∞	∞	∞	∞	∞
<i>P:</i>	\emptyset						
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

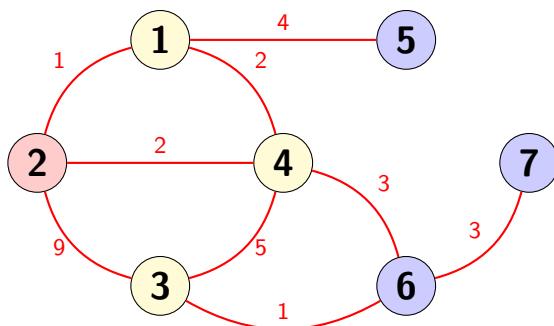
<i>S:</i>	1	3	4	5	6	7	
-----------	---	---	---	---	---	---	--



(b) A prioritási sorból kiveszük azt az elemet, amelyikhez a legkisebb *K*-beli érték tartozik. Ez a 2-es elem, ezzel kezdődik a feldolgozás.

<i>K:</i>	1	0	9	2	∞	∞	∞
<i>P:</i>	2	\emptyset	2	2	\emptyset	\emptyset	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

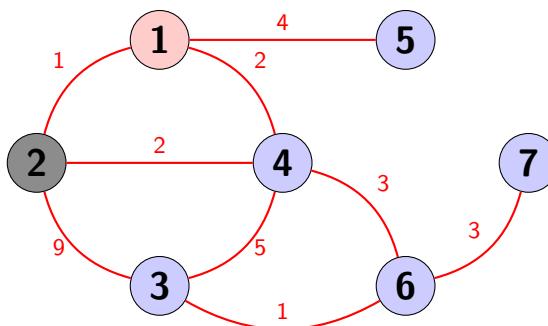
<i>S:</i>	1	4	3	5	6	7	
-----------	---	---	---	---	---	---	--



(c) Megvizsgáljuk a 2-es elem három szomszédját. Mindháromnál beállítjuk az új felső becsléseket az 2-esből vezető él súlyokra. Ennek megfelelően módosítjuk a *P* megfelelő mezőit is, hogy honnan feltételezzük a bekötést, és átrendezzük az *S*-t.

<i>K:</i>	1	0	9	2	∞	∞	∞
<i>P:</i>	2	\emptyset	2	2	\emptyset	\emptyset	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

<i>S:</i>	4	3	5	6	7		
-----------	---	---	---	---	---	--	--

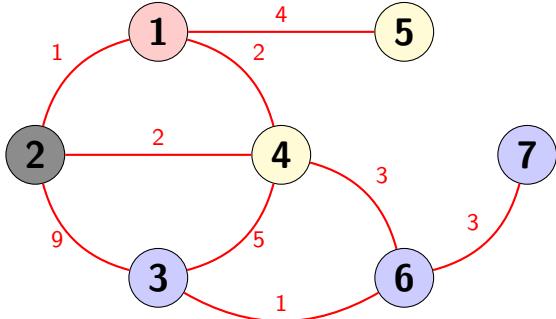


(d) A 2-es elem ezzel feldolgozottnak tekintjük. A prioritási sorból ismét kiveszük azt, amelyik a legkisebb *K*-beli értékkel bír. Ez most az 1-es.

13.4. ábra. Prim algoritmus működése.

<i>K:</i>	1	0	9	2	4	∞	∞
<i>P:</i>	2	\emptyset	2	2	1	\emptyset	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

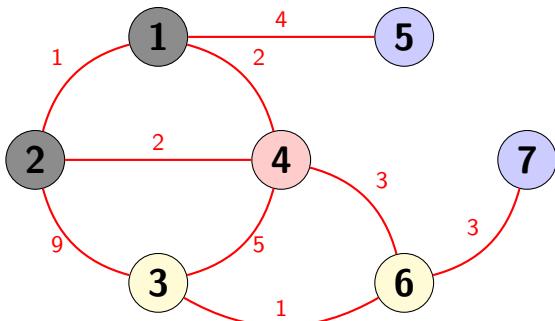
<i>S:</i>	4	5	3	6	7		
-----------	---	---	---	---	---	--	--



(e) Megvizsgáljuk ennek a szomszédait, az egyesből vezető él súlya kisebb lesz, mint az eddig talált legrövidebb bekötés. Az 5-ös elem esetében ez igaz, a 4-esnél nem, a 2-es elem pedig már nincs az *S*-ben.

<i>K:</i>	1	0	5	2	4	3	∞
<i>P:</i>	2	\emptyset	4	2	1	4	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

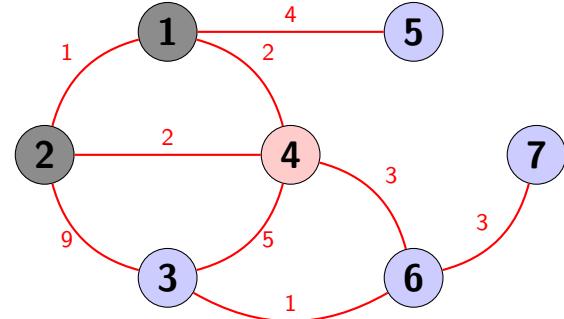
<i>S:</i>	6	5	3	7			
-----------	---	---	---	---	--	--	--



(g) A 4-es szomszédait is átvizsgáljuk, hogy találtunk-e a 4-esből induló rövidebb élet a még nem lezárt csúcsokhoz. Az 1 és 2 már nincs az *S*-ben. A 3-as csúcsba találtunk egy 5 súlyú élet, tehát ez jobb mint az eddig 9-, emiatt cserélünk. A 6-osba pedig ez az első bekötés amit találtunk.

<i>K:</i>	1	0	9	2	4	∞	∞
<i>P:</i>	2	\emptyset	2	2	1	\emptyset	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

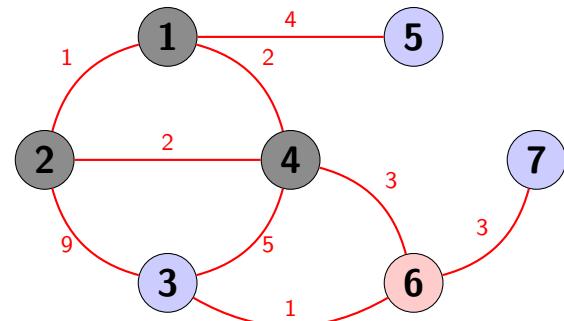
<i>S:</i>	5	3	6	7			
-----------	---	---	---	---	--	--	--



(f) Az 1-est is feldolgoztnak tekinthetjük. Ismét ki kell venni a prioritási sorból a legkisebb *K* értékkel bíró értéket. Jelenleg ez a 4-es.

<i>K:</i>	1	0	5	2	4	3	∞
<i>P:</i>	2	\emptyset	4	2	1	4	\emptyset
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

<i>S:</i>	5	3	7				
-----------	---	---	---	--	--	--	--

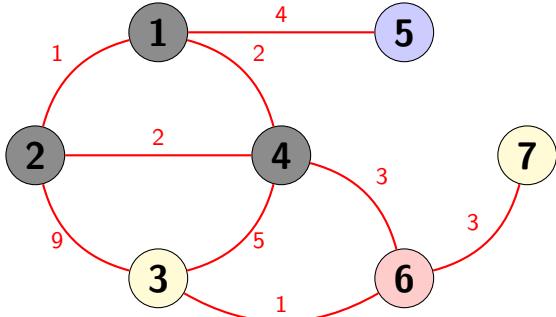


(h) A 4-es elemet ezzel feldolgozottnak tekintjük. A prioritási sorból ismét kiveszük azt, amelyik a legkisebb *K*-beli értékkal bír. Ez most az 6-os.

13.4. ábra. Prim algoritmus működése.(folytatás)

<i>K:</i>	1	0	1	2	4	3	3
<i>P:</i>	2	\emptyset	6	2	1	4	6
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

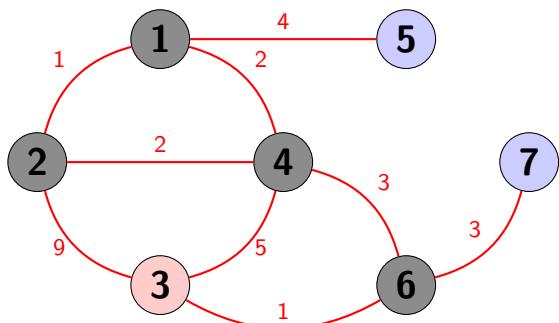
<i>S:</i>	3	7	5				
-----------	---	---	---	--	--	--	--



(i) Az 6-os szomszédai a 3 és a 7. Mindkét irányba jobbnak tűnik az innen induló él, mint az eddigi feltételezés.

<i>K:</i>	1	0	1	2	4	3	3
<i>P:</i>	2	\emptyset	6	2	1	4	6
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

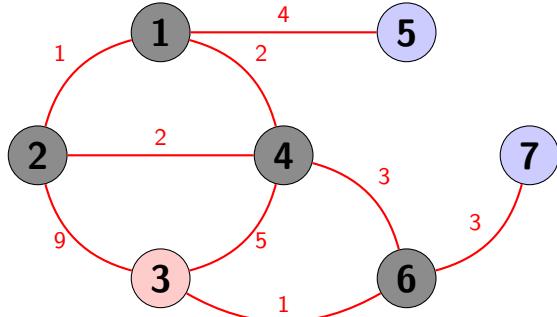
<i>S:</i>	7	5					
-----------	---	---	--	--	--	--	--



(k) Egyik szomszéda sincs már bent az *S*-ben, ezért nem módosítunk semmit.

<i>K:</i>	1	0	1	2	4	3	3
<i>P:</i>	2	\emptyset	6	2	1	4	6
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

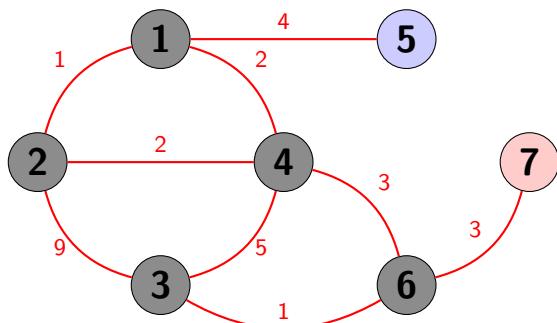
<i>S:</i>	7	5					
-----------	---	---	--	--	--	--	--



(j) Az 6-ost is feldolgoztnak tekinthetjük. Ismét ki kell venni a prioritási sorból a legkisebb *K* értékkel bíró értéket. Jelenleg ez a 3-as.

<i>K:</i>	1	0	1	2	4	3	3
<i>P:</i>	2	\emptyset	6	2	1	4	6
	↑	↑	↑	↑	↑	↑	↑
	1	2	3	4	5	6	7

<i>S:</i>	5						
-----------	---	--	--	--	--	--	--



(l) A prioritási sorban már csak a 5-ös és 7-es van. Ezek közül a 7-esnek kisebb a *K*-beli értéke, emiatt ez a következő.

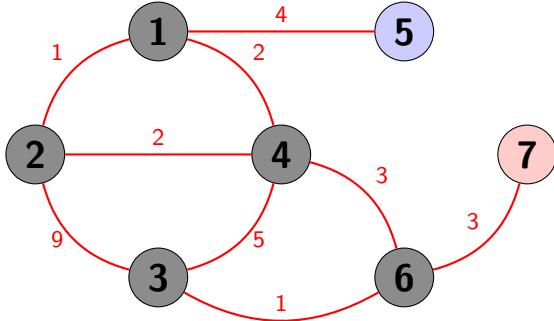
13.4. ábra. Prim algoritmus működése.(folytatás)

<i>K:</i>	1	0	1	2	4	3	3
<i>P:</i>	2	\emptyset	6	2	1	4	6

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

1 2 3 4 5 6 7

<i>S:</i>	5						
-----------	---	--	--	--	--	--	--



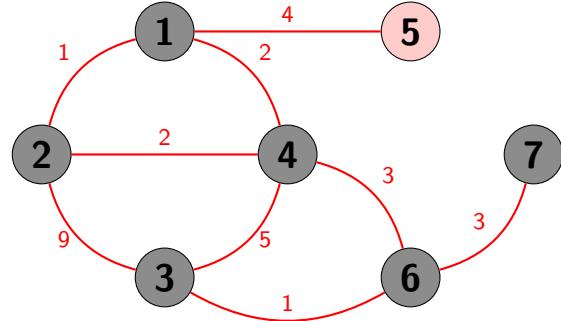
(m) A 7-es egyetlen szomszéda már nincs benne az *S*-ben, ezért nem csinál semmit.

<i>K:</i>	1	0	1	2	4	3	3
<i>P:</i>	2	\emptyset	6	2	1	4	6

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

1 2 3 4 5 6 7

<i>S:</i>							
-----------	--	--	--	--	--	--	--



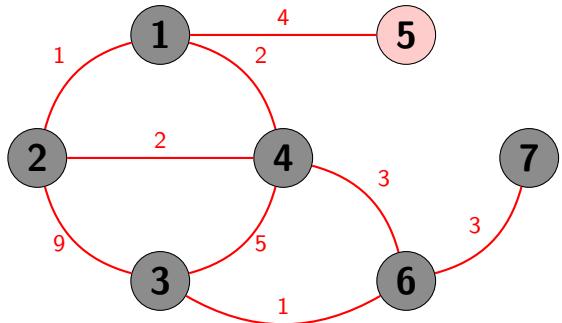
(n) A prioritási sorban már csak az 5-ös van, így ezt fogjuk feldolgozni.

<i>K:</i>	1	0	1	2	4	3	3
<i>P:</i>	2	\emptyset	6	2	1	4	6

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

1 2 3 4 5 6 7

<i>S:</i>							
-----------	--	--	--	--	--	--	--



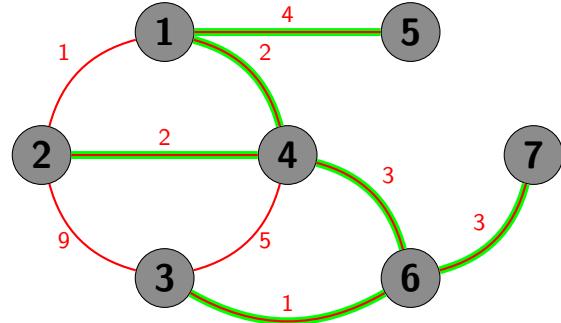
(o) Mivel már mindenki mást feldolgoztunk, így értelmezés szerűen itt se találunk jobb értéket a szomszéd(ok) között.

<i>K:</i>	1	0	1	2	4	3	3
<i>P:</i>	2	\emptyset	6	2	1	4	6

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

1 2 3 4 5 6 7

<i>S:</i>							
-----------	--	--	--	--	--	--	--



(p) A sor üres, tehát az algoritmus futása végetért. A *P* alapján felrajzolható a minimális feszítőfa.

13.4. ábra. Prim algoritmus működése.(folytatás)

Mivel az egyes éleket növekvő sorrendben szeretnénk majd feldolgozni, ezért ezeket egyszerűen elhelyezzük egy prioritási sorba a súly szerinti rendezést használva. Ezt végzi el a második ciklus (6. sor).

Végül a fő ciklus elvégzi a fent már leírt műveleket:

1. Kiveszi a következő legkisebb élt a prioritási sorból (9. sor).
2. Ellenőrzi, hogy az él két végpontja különböző halmazokban van-e (10. sor).
3. Ha igen, akkor ezzel az éssel bővíti az A halmazban lévő éleket, amely a leendő feszítőfa éleit tartalmazzák (11. sor).
4. Mivel a bővítés során az új él összeköti a két végpontjánál lévő gráf komponenseket, ezek mostantól egy komponenst alkotnak. Emiatt össze kell vonni a két végponjánál lévő halmazt (12. sor).

Az algoritmus által használt halmazműveletek elsőre bonyolultnak tűnnek, de valójában nem azok. Nincs csökkenthetően szükségünk külön halmaz adatszerkezetre a megvalósításhoz, elég ha az egyes csúcsoknál (magában a csúcsban, vagy pl. egy szótárban) eltároljuk, hogy adott csúcs hányadik halmazban van.

- **HALMAZ-LÉTREHOZÁS(x):** Így már egyszerű, hiszen nem kell ténylegesen halmazokat létrehozni, csak az egyes x csomópontokhoz kell hozzárendelni egy egyedi „halmaz azonosítót”, ami lehet akár egy egész szám is.
- **TARTALMAZÓ-HALMAZ(u):** Esetén nincs szükség keresésre, csak vissza kell adni a csúcshoz tárolt halmaz azonosítót. Ha a vizsgált él végpontjainál ez ugyanaz, akkor azzal nem szabad bővíteni a fát.
- **HALMAZ-ÖSSZEVONÁS(a,b):** Ez szintén egyszerűen megoldható. Végig kell nézni minden csúcsot, és minden b halmaz azonosítót át kell írni a -ra (vagy épp fordítva).

Az ábrán látható példánál is ezt a módszert követjük, egyszerűen a csúcsok mellett ábrázoljuk, hogy ōk melyik halmazba tartoznak.

13.4. Algoritmus Kruskal algoritmusa

Bemenet: g - SúlyozottGráf $\langle V, E \rangle$ (a gráf, aminek a feszítőfáját keressük)

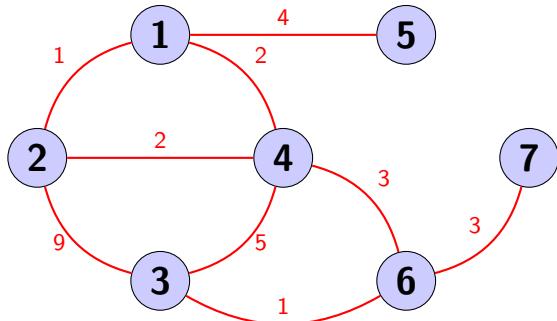
Kimenet: A - Halmaz $\langle E \rangle$ (tartalmazza a minimális feszítőfa éleit)

- 1: **függvény** KRUSKALFESZÍTŐFAKERESÉS(g)
 - 2: **ciklus** $\forall x \in g.Csúcsok$
 - 3: HALMAZ-LÉTREHOZÁS(x) ▷ minden csúcs egy saját halmazba kerül
 - 4: **ciklus vége**
 - 5: **ciklus** $\forall (u, v) \in g.Élek$
 - 6: $S \Leftarrow (u, v)$ ▷ minden él bekerül a prioritási sorba
 - 7: **ciklus vége**
 - 8: **ciklus amíg** $S \neq \emptyset$
 - 9: $(u, v) \xleftarrow{\min} S$ ▷ legkisebb súlyú él kivétele
 - 10: **ha** TARTALMAZÓ-HALMAZ(u) \neq TARTALMAZÓ-HALMAZ(v) **akkor**
 - 11: $A \leftarrow A \cup \{(u, v)\}$
 - 12: HALMAZ-ÖSSZEVONÁS(TARTALMAZÓ-HALMAZ(u), TARTALMAZÓ-HALMAZ(v))
 - 13: **elágazás vége**
 - 14: **ciklus vége**
 - 15: **vissza** A
 - 16: **függvény vége**
-

Felhasznált változók és függvények

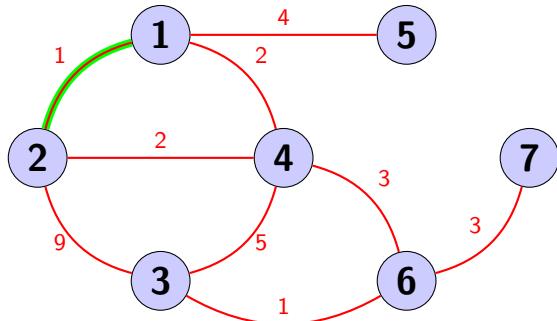
- S : Egy **PrioritásosSor** $\langle E \rangle$, amely a gráf éleit tárolja súly szerint növekvő sorrendben.
 - A : Egy **Halmaz** $\langle E \rangle$, amely gyűjtíti a minimális feszítőfa éleit.
 - $\xleftarrow{\min} S$: Az S prioritási sorból visszaadja a legkisebb súlyú élet, és törli is azt a sorból.
 - (u, v) : Egy E típusú élt képvisel, amely az u és v csúcsokat köti össze.
 - HALMAZ-LÉTREHOZÁS(x) : Létrehoz egy új halmazt, aminek egyetlen eleme a x lesz.
 - TARTALMAZÓ-HALMAZ(u) : Visszaadja a u csúcsot tartalmazó halmaz azonosítóját.
 - HALMAZ-ÖSSZEVONÁS(a, b) : Összevonja az a és b halmazokat.
-

<i>H:</i>	<table border="1"> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> </table>	1	2	3	4	5	6	7			
1	2	3	4	5	6	7					
	\uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow										
<i>S:</i>	<table border="1"> <tr><td>(1,2)</td><td>(3,6)</td><td>(1,4)</td><td>(2,4)</td><td>(6,7)</td><td>(4,6)</td><td>(1,5)</td><td>(3,4)</td><td>(2,3)</td><td></td></tr> </table>	(1,2)	(3,6)	(1,4)	(2,4)	(6,7)	(4,6)	(1,5)	(3,4)	(2,3)	
(1,2)	(3,6)	(1,4)	(2,4)	(6,7)	(4,6)	(1,5)	(3,4)	(2,3)			



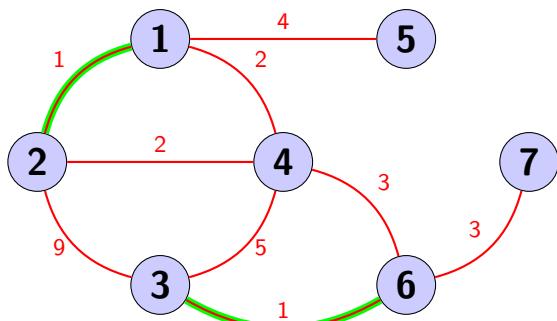
(a) Közvetlenül az inicializálást követő állapot. A *H* tartalmazza, hogy a csúcsok melyik halmazban vannak. A prioritási sor pedig fel van töltve az élekkel súly szerint rendezve.

<i>H:</i>	<table border="1"> <tr><td>1</td><td>1</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> </table>	1	1	3	4	5	6	7		
1	1	3	4	5	6	7				
	\uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow									
<i>S:</i>	<table border="1"> <tr><td>(3,6)</td><td>(1,4)</td><td>(2,4)</td><td>(6,7)</td><td>(4,6)</td><td>(1,5)</td><td>(3,4)</td><td>(2,3)</td><td></td></tr> </table>	(3,6)	(1,4)	(2,4)	(6,7)	(4,6)	(1,5)	(3,4)	(2,3)	
(3,6)	(1,4)	(2,4)	(6,7)	(4,6)	(1,5)	(3,4)	(2,3)			



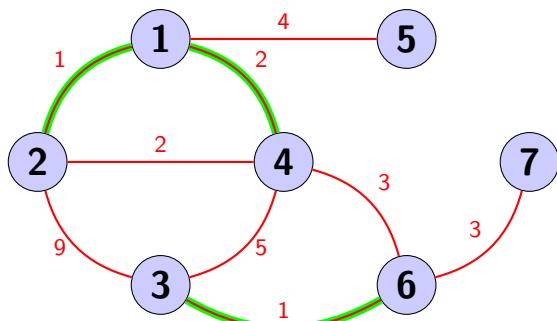
(b) A sorból az első él az (1, 2). A halmaz azonosítók (1 és 2) különbözök, ezért ezzel bővítjük a feszítőfát. A két halmazt összevonjuk.

<i>H:</i>	<table border="1"> <tr><td>1</td><td>1</td><td>3</td><td>4</td><td>5</td><td>3</td><td>7</td></tr> </table>	1	1	3	4	5	3	7	
1	1	3	4	5	3	7			
	\uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow								
<i>S:</i>	<table border="1"> <tr><td>(1,4)</td><td>(2,4)</td><td>(6,7)</td><td>(4,6)</td><td>(1,5)</td><td>(3,4)</td><td>(2,3)</td><td></td></tr> </table>	(1,4)	(2,4)	(6,7)	(4,6)	(1,5)	(3,4)	(2,3)	
(1,4)	(2,4)	(6,7)	(4,6)	(1,5)	(3,4)	(2,3)			



(c) Következő él a (3, 6). Szintén különböző halmazokban vannak, ezért ezzel is bővíthető a fa.

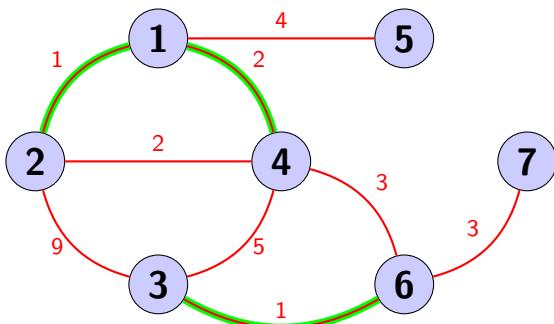
<i>H:</i>	<table border="1"> <tr><td>1</td><td>1</td><td>3</td><td>1</td><td>5</td><td>3</td><td>7</td></tr> </table>	1	1	3	1	5	3	7
1	1	3	1	5	3	7		
	\uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow							
<i>S:</i>	<table border="1"> <tr><td>(2,4)</td><td>(6,7)</td><td>(4,6)</td><td>(1,5)</td><td>(3,4)</td><td>(2,3)</td><td></td></tr> </table>	(2,4)	(6,7)	(4,6)	(1,5)	(3,4)	(2,3)	
(2,4)	(6,7)	(4,6)	(1,5)	(3,4)	(2,3)			



(d) Következik az (1, 4). Ezzel is bővíthetünk, így az 1-es halmazban már három csúcs van.

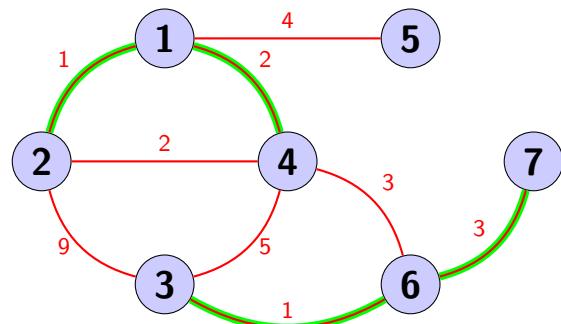
13.5. ábra. Kruskal algoritmus működése.

<i>H:</i>	<table border="1"> <tr><td>1</td><td>1</td><td>3</td><td>1</td><td>5</td><td>3</td><td>7</td></tr> </table>	1	1	3	1	5	3	7	
1	1	3	1	5	3	7			
<i>S:</i>	<table border="1"> <tr><td>(6,7)</td><td>(4,6)</td><td>(1,5)</td><td>(3,4)</td><td>(2,3)</td><td></td><td></td><td></td></tr> </table>	(6,7)	(4,6)	(1,5)	(3,4)	(2,3)			
(6,7)	(4,6)	(1,5)	(3,4)	(2,3)					



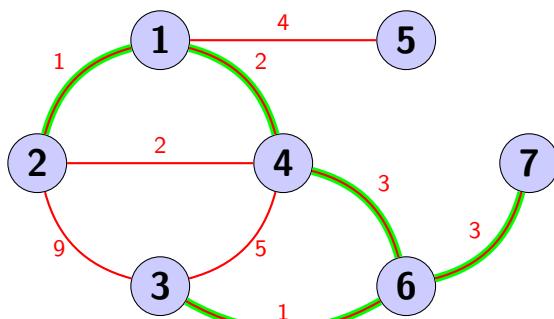
(e) A $(2, 4)$ él a következő. Mindkét csúcs az 1-es halmazban van, ezért ezt nem húzhatjuk be. Látható is, hogy egy kört okozna.

<i>H:</i>	<table border="1"> <tr><td>1</td><td>1</td><td>3</td><td>1</td><td>5</td><td>3</td><td>3</td></tr> </table>	1	1	3	1	5	3	3
1	1	3	1	5	3	3		
<i>S:</i>	<table border="1"> <tr><td>(4,6)</td><td>(1,5)</td><td>(3,4)</td><td>(2,3)</td><td></td><td></td><td></td></tr> </table>	(4,6)	(1,5)	(3,4)	(2,3)			
(4,6)	(1,5)	(3,4)	(2,3)					



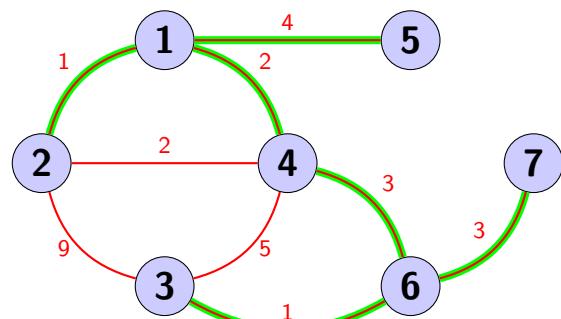
(f) A $(6, 7)$ él következik, ezzel bővíthető a feszítő fa. A 7-es pont is a 3-as halmazba került.

<i>H:</i>	<table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>5</td><td>1</td><td>1</td></tr> </table>	1	1	1	1	5	1	1
1	1	1	1	5	1	1		
<i>S:</i>	<table border="1"> <tr><td>(1,5)</td><td>(3,4)</td><td>(2,3)</td><td></td><td></td><td></td><td></td></tr> </table>	(1,5)	(3,4)	(2,3)				
(1,5)	(3,4)	(2,3)						



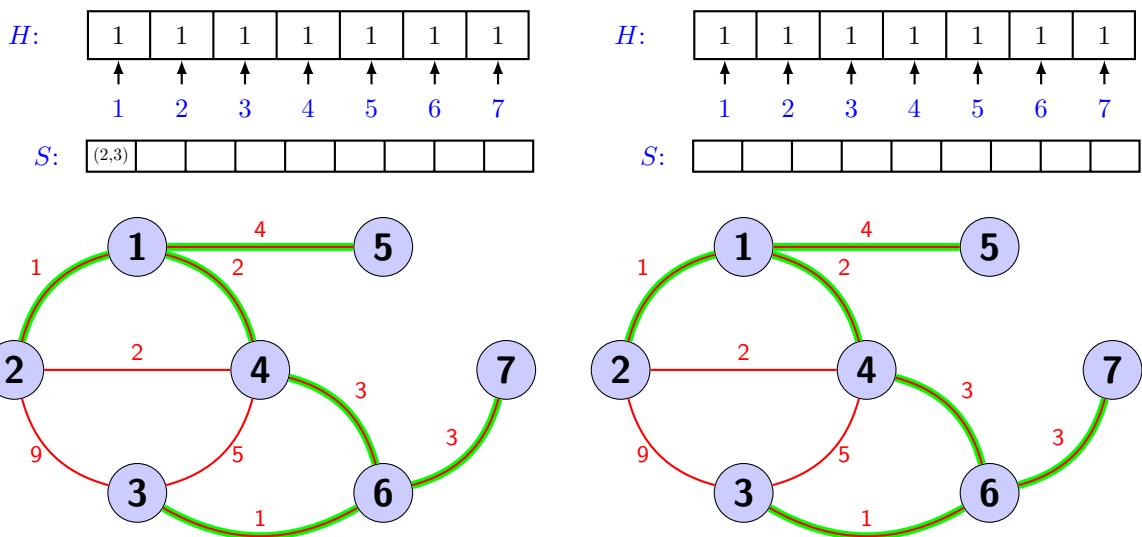
(g) Következik a $(4, 6)$. Különböző halmazokban vannak. Látható, hogy ezzel a lépéssel összekapcsoljuk a két nagy komponenst.

<i>H:</i>	<table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </table>	1	1	1	1	1	1	1
1	1	1	1	1	1	1		
<i>S:</i>	<table border="1"> <tr><td>(3,4)</td><td>(2,3)</td><td></td><td></td><td></td><td></td><td></td></tr> </table>	(3,4)	(2,3)					
(3,4)	(2,3)							



(h) Az $(1, 5)$ gond nélkül behúzható.

13.5. ábra. Kruskal algoritmus működése.(folytatás)



(i) Már minden csúcs egy komponensben van, ezért az (1,3) élt eldobjuk.

(j) Az utolsó (2,3) él is kört okozna, emiatt nem húzzuk be. Üres a sor, ezért az algoritmus véget ért.

13.5. ábra. Kruskal algoritmus működése.(folytatás)

14. fejezet

B-fa

14.1. Fák kiegyensúlyozottsága

A 6. fejezetben megismert bináris keresőfa esetében előnyként hoztuk fel a gyors keresési lehetőséget, aminek a várható lépésszáma n darab csúcs és megfelelően felépített fa esetében $\mathcal{O}(\log_2 n)$. Ugyanakkor az is könnyen belátható, hogy amennyiben a fa szerkezete kevésbé ideális, tehát az egyes levelek mélysége egymástól jelentősen különbözik, akkor ez a várható átlagos lépésszám is jóval nagyobb lesz, legrosszabb esetben akár $\mathcal{O}(n)$ lépéstre is szükség lehet. Annak érdekében, hogy valamilyen formában pontosítsuk a fák „jóságát”, vezessük be az alábbi fogalmakat:

- Kiegyensúlyozott fa: az azonos szinten található részfák mélységei között legfeljebb egy szintnyi különbség lehet.
- Teljesen kiegyensúlyozott fa: minden csúcsra igaz, hogy az abból kiinduló részfák csúcsainak száma legfeljebb egyetlen különbözik egymáshoz képest.
- Teljes fa: minden csúcsra igaz, hogy az abból kiinduló részfák csúcsainak száma azonos.

Célunk természetesen a minél inkább kiegyensúlyozott fák felépítése, bár a fenti feltételeket nem is tudjuk minden esetben teljesíteni (pl. két csúcsot tartalmazó bináris keresőfa nyilvánvalóan nem lehet teljes). A megismert bináris keresőfa beszűró és törlő algoritmusok nem foglalkoztak a fa kiegyensúlyozottságával, így ezekben az esetekben a fa felépítésében csak a beérkező tartalmak sorrendje játszott szerepet. Ettől függően (szerencsés esetben) a fa lehetett akár teljes is, de (szerencsétlen esetben) elfajult is, ami jelentősen lerontotta a keresés hatékonyságát. A bináris keresőfának is léteznek továbbfejlesztései (pl. piros-fekete fa [2]), amelyek a beszúrás és a törlés során igyekeznek olyan formába átalakítani a fát, hogy az minél inkább közelítsen a kiegyensúlyozott állapothoz, ebben a fejezetben azonban egy másik lehetséges megoldással, a B-fák használatával foglalkozunk.

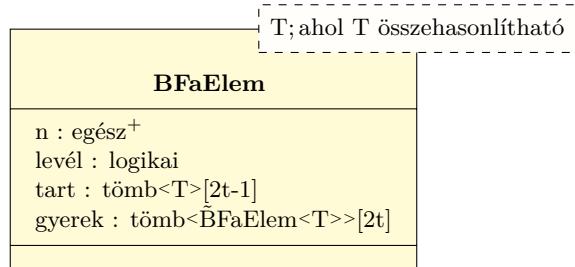
14.2. B-fa felépítése

A B-fa tulajdonképpen a bináris keresőfa általánosításaként is tekinthető, azt annyiból egészíti ki, hogy egy csomópont egynél több tartalmat is tartalmazhat, és ezzel együtt a gyerekek száma is lehet kettőnél több (14.1. ábra). Mindez persze jelentősen meg változtatja a fa felépítését, illetve a rajta különböző műveleteket végző algoritmusok működését.

A B-fa minden egyes csomópontja az alábbi mezőket tartalmazza (minden csomópont $\text{BFaElem} < T >$ típusú):

- n : egy egész szám, ami a csomópontban aktuálisan eltárolt tartalmak darabszámát mutatja (később látni fogjuk, hogy erre különféle korlátok fognak majd vonatkozni).
- $levél$: logikai érték, ami azt mutatja, hogy az adott csúcs levél (nincs gyereke), vagy pedig nem levél (vannak gyerekei). Értelemszerűen el is tárolható ez az érték egy külön mezőben, de akár futás közben is számolható.
- $tart[1], tart[2], \dots, tart[n]$: a csomópontban eltárolt tartalmi elemek. A bináris keresőfához hasonlóan tetszőleges típusú lehet, csak ahhoz ragaszkodunk, hogy összehasonlítható legyen. Itt is típusparamétereket fogunk használni, ezért a tartalom mező típusa T . Ezeket célszerűen egy tömbben tárolhatjuk, aminek a mérete $2t - 1$ (a t értékét később definiáljuk).
- $gyerek[1], gyerek[2], \dots, gyerek[n+1]$: a belső csúcsok hivatkozásai a gyerekekre. A gyerekmutatók száma mindenkorral egygyel több, mint a tartalmak száma. A gyerek hivatkozások típusa a kódokban: $\tilde{\text{BFaElem}} < T >$. Ezeket célszerűen egy tömbben tárolhatjuk, aminek a mérete $2t$ (a t értékét később definiáljuk).

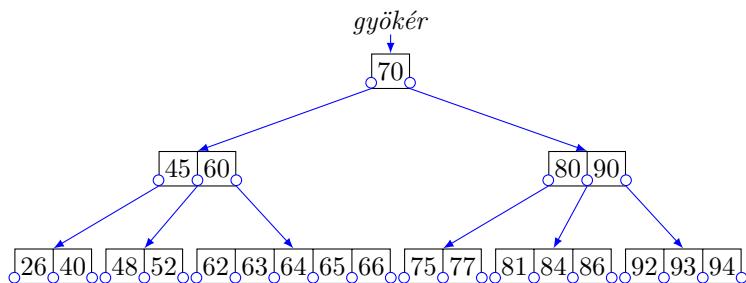
Egy objektumorientált megvalósításban ennek a 14.2. ábrán megfelelő osztály felel meg.



14.2. ábra. B-fa egy elemének felépítése.

A 14.1. ábra alapján is láthatók a B-fa jellemzői:

- Egyediség: nem lehet két azonos tartalmú elem a fában.



14.1. ábra. B-fa példa. A számok a fában tárolt tartalmak. Az egyes kulcsok előtt/között/mögött látható körök a gyerekekre hivatkozó mutatók (a mutatók száma minden csomópontban egygyel több, mint a tartalmak száma). A fa legsalsó szintjén a gyerekmutatók értéke \emptyset , ezt nem ábrázoljuk.

- Rendezettség: minden csúcs esetében igaz, ha x_i egy adat a *gyerek*[i] gyökerű részfában, akkor:

$$x_1 \leq \text{tart}[1] \leq x_2 \leq \text{tart}[2] \leq \dots \leq x_n \leq \text{tart}[n] \leq x_{n+1} \quad (14.1)$$

- kiegyensúlyozott: minden levél mélysége azonos.
- korlát a tartalmak darabszámára: Az egyes csúcsokban található tartalmak darabszáma alulról és felülről is korlátos, ezt előre rögzített t számmal jelöljük (B-fa minimális fokszáma). Ez alapján:
 - minden csúcsnak legfeljebb $(2t - 1)$ darab tartalma lehet (tehát legfeljebb $2t$ darab gyereke). *Telítettnek* nevezzük azt a csúcsot, amelyiknek már $(2t - 1)$ tartalma van.
 - minden nemgyökér csúcsnak legalább $(t - 1)$ darab tartalma kell, hogy legyen (tehát legalább t darab gyereke).

Az általunk használt beszúrás és törlés művelet garantálni fogja, hogy a B-fa minden módosítás után teljesíteni fogja mindegyik feltételét. Érdemes megjegyezni, hogy a minimumfeltétel a gyökérelemre nem vonatkozik (ami logikus, hiszen egy olyan fa esetében, ami csak egy elemet tárol, nem tudnánk azt teljesíteni).

Megjegyzés

A szakirodalomban a tartalom mezőt gyakran kulcsnak nevezik, ami az egyediség alapján indokolt, mi viszont nem szeretnék eltérni az listáknál/bináris keresőfáknál már megismert elnevezésektől. A szövegben a tartalom/kulcs/érték/adat szavak mind ugyanarra a mezőre vonatkoznak majd, ezeket felváltva használjuk (ennek oka pusztán stilisztikai, hogy elkerüljük a „tartalmú tartalmat tartalmazó”-hoz hasonló kifejezésekkel). A csúcs/csomópont alatt pedig egy egész B-fa csomópontra gondolunk.

Másodlagos tároló használata

Az ábrán is feltűnő, hogy a bináris keresőfához képest a legnagyobb különbséget az jelenti, hogy a B-fa adatszerkezet esetében egy csomópontban nem csak egy, hanem egyszerre több tartalmat is tárolunk, ami értelemszerűen nem csak kettő, hanem annál több gyerek hivatkozással is jár. Ennek köszönhetően azonos kulcsszám mellett csökken a fa magassága (a gyökérből a levelekig vezető leghosszabb út hossza). A kiegyensúlyozott bináris fa esetében n adat tárolásához $\log_2 n$ magas fára volt szükség, a B-fánál a képlet hasonló, azonban a csomópontok nagyobb „elágazási tényezőjének” hatására a logaritmus alapja 2-nél nagyobb lehet. Bizonyítható, hogy a fa magasságára igaz az alábbi összefüggés [2]:

$$h \leq \log_t \frac{n + 1}{2} \quad (14.2)$$

Ahol

- n : a B-fában lévő értékek száma ($n \geq 1$),
- t : a fent említett minimális fokszám ($t \geq 2$),
- h : a fa magassága.

Tévedés lenne azonban azt gondolni, hogy ez a tény önmagában indokolná a B-fák létezését. Míg a bináris keresőfa esetében a keresés lépésszáma a gyökérből a levelekig vezető út hosszával volt jellemző, addig a B-fa esetében ez már nem igaz. Itt ugyanis az egyes csomópontokon belül már nem tudjuk egy egyszerű összehasonlítással eldönteni, hogy melyik irányba folytassuk a keresést, hiszen nem csak egy, hanem akár $(2t - 1)$ darab tartalmunk is lehet. A tartalmakat rendezve tároljuk, tehát használhatjuk a logaritmikus keresést, de belátható, hogy a t értékének növelésével (ami a fa magasságának csökkenését jelenti) arányosan növekszik a csomópontokon belüli műveletek lépésszáma.

A B-fa hatékonyságának megértéséhez tudnunk kell azt is, hogy ezt az adatszerkezetet tipikusan a másodlagos tárolókon való működésre terveztek. A bináris keresőfa esetében eddig minden úgy képzeltük, hogy a teljes fa a memóriában helyezkedik el, így meglehetősen gyorsan tudunk az egyes csomópontok

között mozogni. Nagy mennyiségű adat esetében már kénytelenek vagyunk a fát a másodlagos háttér-táron elhelyezni, ez esetben viszont minden egyes csomópont hozzáférés egy-egy háttértár írási/olvasási műveletet is jelent, ami drasztikusan lecsökkenti a teljesítményt.

Ez az a pont, ahol megjelenik a B-fa igazi erőssége. Napjainkban másodlagos tárolóként még mindig elsősorban merevlemezeket használunk, amelyek az adatokat sávokra, illetve azon belül blokkokra bontva tárolják. Egy írási/olvasási műveletnél a legtöbb időt a fej mozgatása, illetve a lemezek megfelelő irányba állítása igényli, maga az írás/olvasás ehhez képest már rövid ideig tart. Emiatt a technika sajátossága, hogy egy teljes blokk kiírása/beolvasása közel ugyanannyi ideig tart, mint egy byte írása/olvasása. A B-fa pedig nagyon jól kihasználja ezt a lehetőséget, ugyanis ha célszerűen úgy választjuk meg a t paramétert, hogy egy B-fa csomópont egy merevlemez blokkot foglaljon el, akkor egy hozzáféréssel minden egy teljes csomópontot tudunk írni/olvasni. Tehát a B-fa egy $(2t - 1)$ darab kulcsot tartalmazó csomópont-jának a kiírása/beolvasása ugyanannyi ideig tart, mint a bináris keresőfa egyetlen kulcsot tartalmazó csomópontjáé.

Tehát az valóban igaz, hogy a B-fa keresés elvi lépésszáma hasonló, mint a bináris keresőfáé, azonban ha megkülönböztetjük az új csomópontokra lépés műveletét (ami merevlemez műveletet igényel) és a csomóponton belüli keresést (ami már jóval gyorsabb memóriaművelet), akkor látható, hogy ez előbbi jóval hatékonyabban működik a jelenleg használt másodlagos tárolók esetében.

Megjegyzés

Éppen ez a megkülönböztetés miatt a szakirodalomban gyakran külön jelzik a B-fa algoritmusok esetében a háttértár olvasási és írási műveleteket. Az könnyebb olvashatóság kedvéért mi ezeket elhagyjuk, de értelemszerűen ezeket odaérthetjük minden csomópont váltás elő és módosítás mögé.

14.3. Beszúrás B-fába

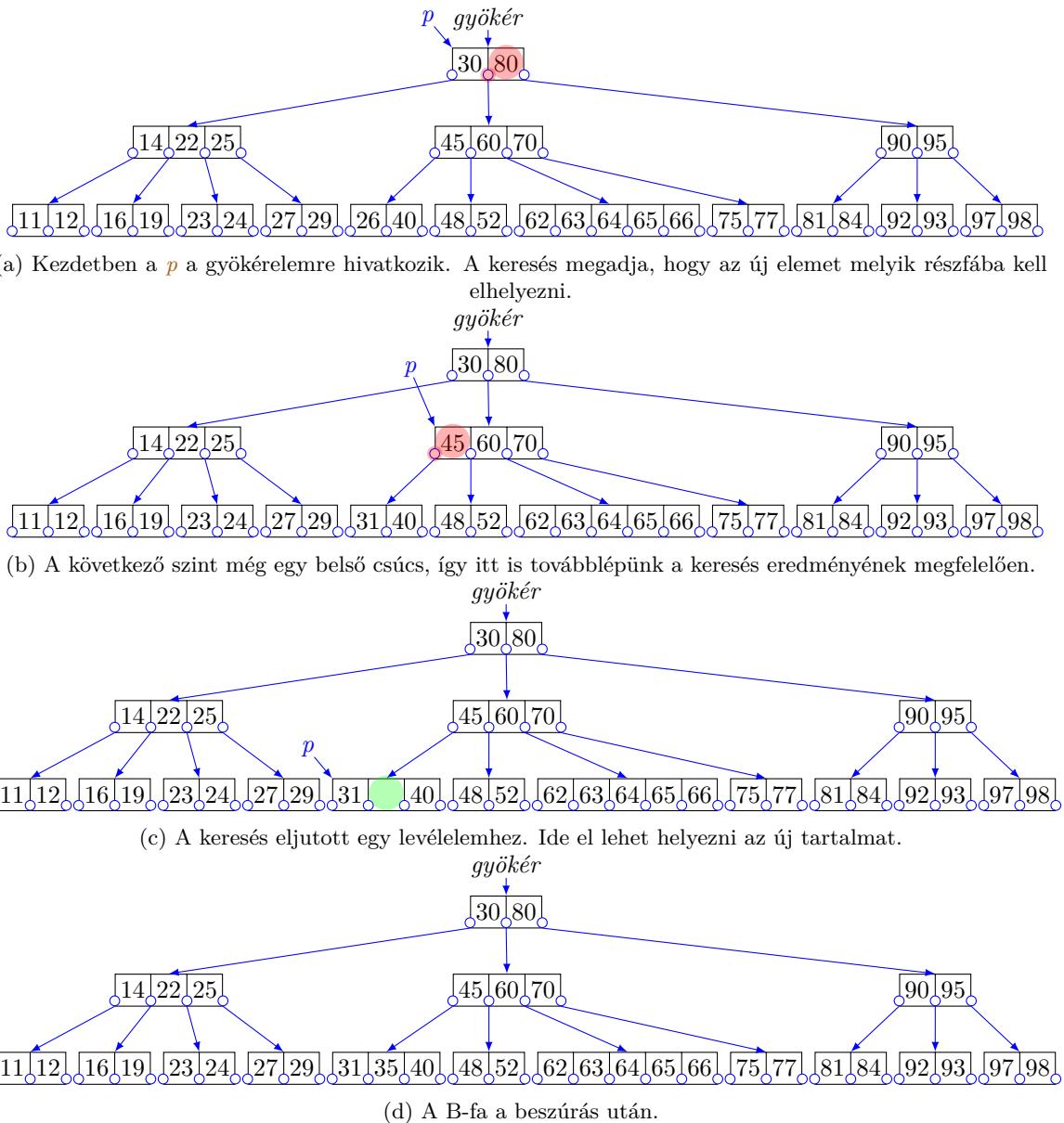
A B-fába beszúrás alapgondolata hasonló a bináris keresőfánál tanulthoz. B-fa esetében is élünk azzal a lehetőséggel, hogy minden csomópontban szűrunk be új elemeket, mivel ebben az esetben nem kell a fa már meglévő részeit átalakítani, elég lokálisan, a megfelelő levélnél végrehajtani a módosítást (új tartalom felvétele). Ennek megfelelően a beszúrás első lépése itt is egy keresés, amivel a B-fában megkeressük, hogy melyik levélnek kell majd tartalmaznia az új értéket. Ez a keresés nyilván azonos az általános B-fában való tartalom szerinti kereséssel, hiszen az új elemet oda kell elhelyeznünk, ahol az utóbbi majd keresni fogja.

A kiegyensúlyozottság fenntartása felvet azonban néhány további követelményt, amelyekhez igazodunk kell. A bináris keresőfa esetében a beszúrást megelőző keresés egy olyan üres helyet keresett, ahova új levélként fel lehetett venni az új tartalmat. A B-fa esetében ez azonban nem járható út, mivel egy már kiegyensúlyozott B-fába nem tudunk felvenni egy új levelet egy már meglévő alá anélkül, hogy ne szegjük meg azt a feltételt, hogy a gyökérből minden levélhez vezető út hosszának azonosnak kell lennie. Szerencsére erre nincs is szükség, mivel a B-fa egy csúcsa több kulcsot is tartalmazhat, így azt a megoldást választjuk, hogy a keresés során talált legalsó, még létező csúcsba felvesszük az új tartalmat. Ezt az esetet mutatja a 14.3. ábra.

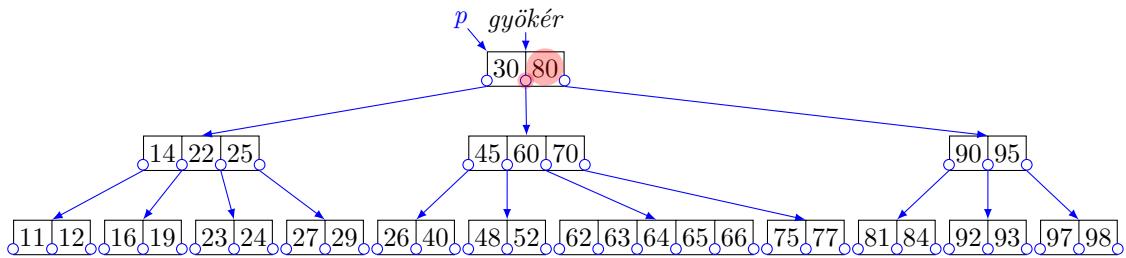
Ez persze csak akkor valósítható meg, ha legalább egy szabad hely még van ebben a levélben. A fa aktuális tartalmától függően ez vagy fennáll, vagy sem, biztosak nem lehetünk benne, tehát úgy kell elkészítenünk az algoritmust, hogy ha szükséges, tudja garantálni ezt az üres helyet. A gyakorlatban ezt a vágás nevű művelettel tudjuk megvalósítani, aminek a célja az, hogy egy telített csúcsot szétvágunk két kisebb (a minimumfeltételt éppen teljesítő) csúcsra, míg a középső tartalmat felvisszük a szülőbe.

A vágás csak akkor végezhető el, ha a szülőben van szabad hely még egy kulcs számára. Emiatt az előbbi követelményt, hogy a keresés során csak olyan belső csúcsra lépjünk, amelyiknek van szabad helye általánosíthatjuk, hogy általában csak olyan elemre lépjünk tovább, ahol van szabad hely. Az új érték helyének keresése során tehát mielőtt a rekurzió meghíváná önmagát valamelyik részfa gyökerére, előtte ellenőrzi, hogy az telített-e. Ha nem, akkor azonnal továbblép ebbe az irányba, ha pedig igen, akkor szétvágja azt. Erre a vágásra mutat példát a 14.4. ábra (illetve maga a vágási művelet a 14.4b–14.4c. ábrákon látható).

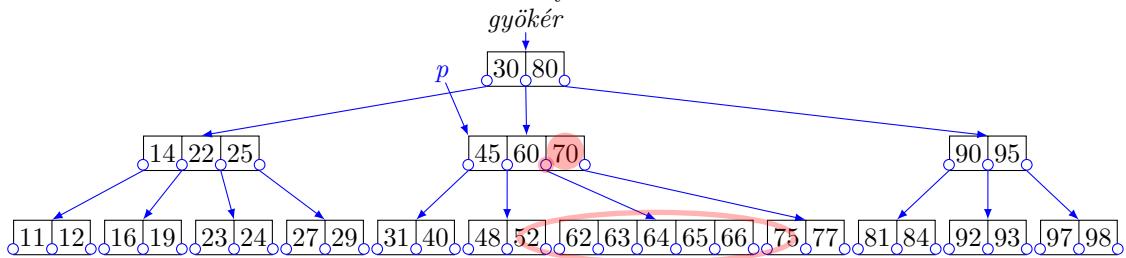
Bár a fentieknek megfelel ez is, de mégis speciális esetként kezelhető az, amikor a gyökérelem telített. A fenti szabály itt is alkalmazható, emiatt szétvágjuk ezt a csúcsot, annyiból viszont különbözik az általános esettől, hogy itt nincs szülő elem, ahol fel lehetne vinni a középső értéket. Emiatt ilyenkor létrehozunk egy új csomópontot, ez lesz az új gyökér. A gyökérelem szétvágására mutat példát a 14.5. ábra.



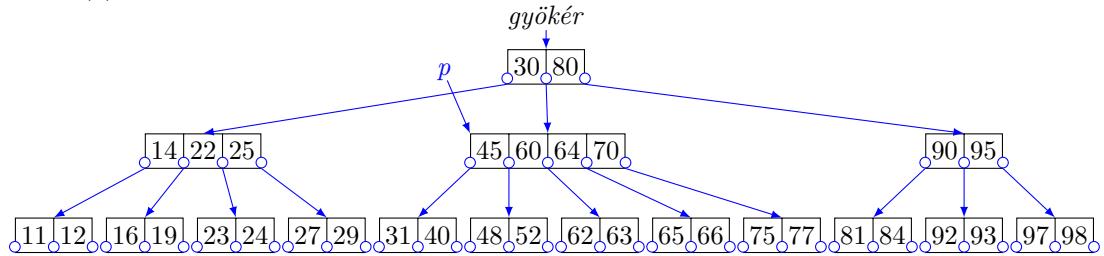
14.3. ábra. B-fába való beszúrás általános esete (35 beszúrása).



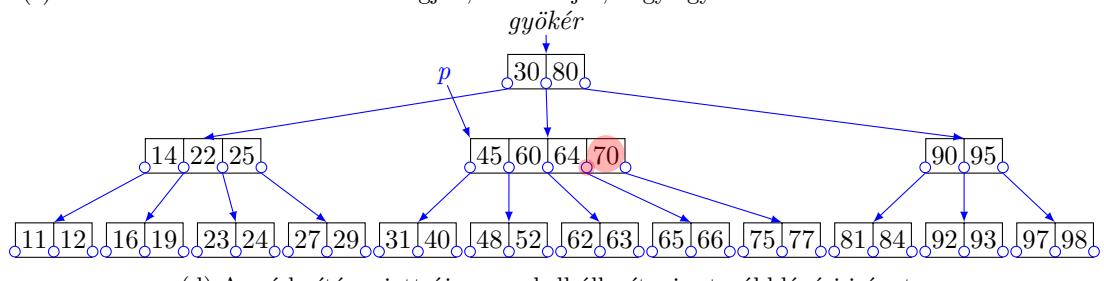
(a) Kezdetben a p a gyökérelemlre hivatkozik. A keresés megadja, hogy az új elemet melyik részfába kell elhelyezni.



(b) A következő szintről nem tudunk továbblépni, mivel a következő csúcs már telített.

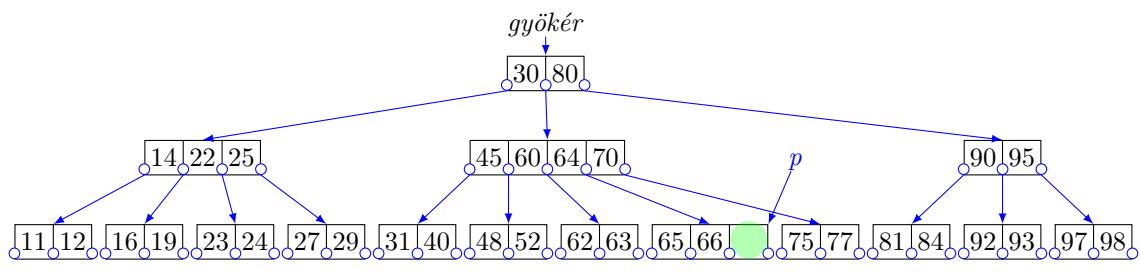


(c) Emiatt a következő elemet szétvágjuk, ez azzal jár, hogy egy tartalmat fel kell vinni a szülőbe.

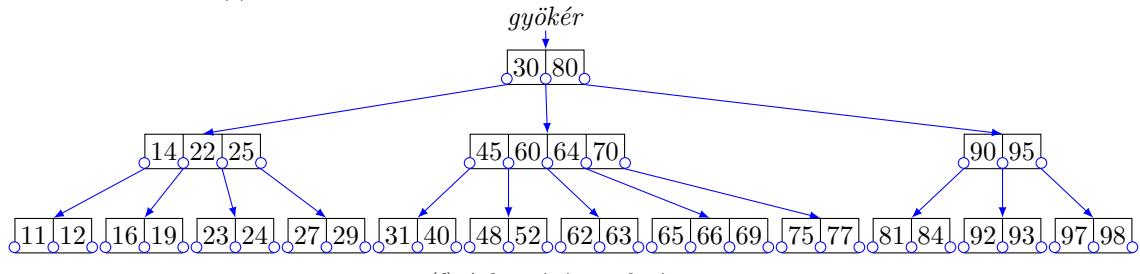


(d) A módosítás miatt újra meg kell állapítani a továbblépési irányt.

14.4. ábra. B-fába való beszúrás során elem szétvágása (69 beszúrása).

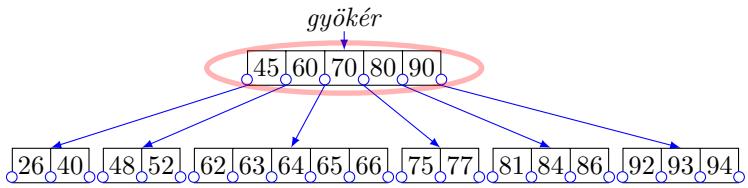


(e) Ismét eljutottunk egy levélig, ide már elvégezhető a beszúrás.



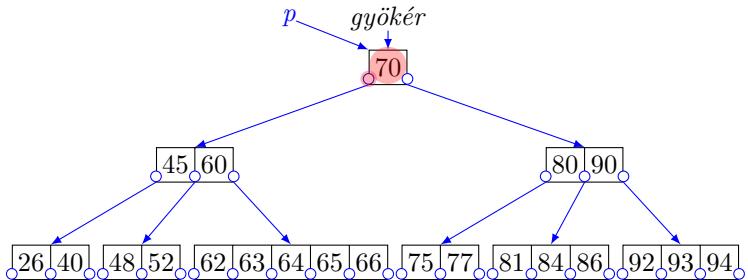
(f) A beszúrás eredménye.

14.4. ábra. B-fába való beszúrás során elem szétvágása (69 beszúrása). (folytatás)

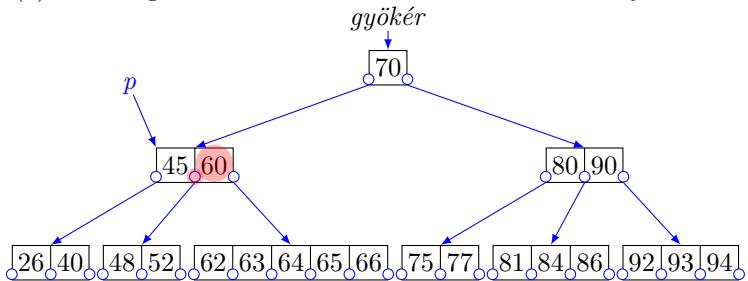


(a) A beszúrásnál már az első továbblépést se tudjuk megtenni, mivel a gyökérelem telített. Emiatt még a rekurzió indítása előtt szétvágjuk.

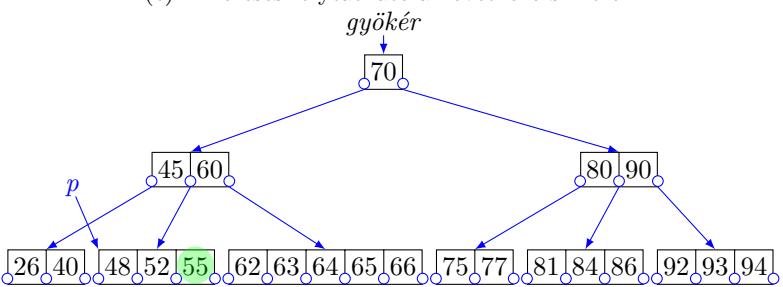
A vágás során létrehozott új elem az új gyökér.



(b) A szétvágást követően a keresés a szokásos módon folytatódik.



(c) A keresés folytatható a következő szintre.



(d) A következő iterációban a levélbe már elhelyezhető a beszúrandó elem.

14.5. ábra. B-fába való beszúrás során a gyökér szétvágása (55 beszúrása).

14.3.1. Beszúrás pszeudokódja

A beszúrás a bináris keresőfához hasonlóan rekurzív, itt azonban több speciális esettel is kell foglalkoznunk, emiatt nem közvetlenül a rekurzív eljárást hívjuk meg, hanem először egy segédfüggvényt. Ennek első paramétere a B-fa gyökere, a második pedig a beszúrandó érték. Ezt mutatja be a 14.1. algoritmus. A két említett speciális eset:

14.1. Algoritmus Beszúrás a B-fába

Bemenet: $\text{érték} - T$ (a beszúrandó érték)

```
1: eljárás BESZÚRÁS( $\text{érték}$ )
2:   ha  $\text{gyökér} = \emptyset$  akkor
3:      $\text{gyökér} \leftarrow \text{LÉTREHOZ}(BFaElem<T>)$ 
4:      $\text{gyökér.levél} \leftarrow \text{igaz}$ 
5:   különben
6:     ha  $\text{gyökér.n} = 2t - 1$  akkor
7:        $p \leftarrow \text{gyökér}$ 
8:        $\text{gyökér} \leftarrow \text{LÉTREHOZ}(BFaElem<T>)$ 
9:        $\text{gyökér.levél} \leftarrow \text{hamis}$ 
10:      SZÉTVÁGÁS( $p, \text{gyökér}$ )
11:      elágazás vége
12:    elágazás vége
13:    KERESÉSBESZÚR( $\text{gyökér}, \text{érték}$ )
14: eljárás vége
```

Felhasznált változók és függvények

- gyökér : A B-fa gyökere, ami esetleg meg is változhat az eljárás futása közben.
 - p : A szétvágáshoz használt $\text{BFaElem}<T>$ segédváltozó.
 - $\text{LÉTREHOZ}(BFaElem<T>)$: Létrehoz egy új csúcsot, és visszaad egy hivatkozást erre.
 - $\text{SZÉTVÁGÁS}(p, \text{szülő})$: Szétvágja a p mutató által hivatkozott csomópontot, amelyiknek a szülőjére a szülő paraméter hivatkozik (14.5. algoritmus).
 - $\text{KERESÉSBESZÚR}(p, \text{érték})$: Megkeresi az érték helyét a p gyökerű B-fában, majd elvégzi annak beszúrását (14.2. algoritmus).
-

A 2. sorban lévő ellenőrzés azt vizsgálja meg, hogy a fa gyökérelemének értéke \emptyset -e. Amennyiben igen, ez azt jelenti, hogy a fa jelenleg üres, így a beszúrás egy új gyökérelem felvételét jelenti. Az újonnan létrehozott elemnek egyetlen tartalma lesz csak (ennek megfelelően beállítjuk a n változó értékét), ez a kulcs pedig a paraméterként kapott érték, de ezt majd a rekurzió fogja felvenni a többi esethez hasonló módon.

Amennyiben a fa nem üres, úgy meg kell vizsgálnunk egy másik speciális esetet, a telített gyökér esetét. Ezt teszi a 6. sor, amely ellenőrzi, hogy a gyökérelemben hány érték található. Amennyiben a tartalmak száma egyenlő a maximális értékkal, akkor a már fent említett módon létrehoz egy új elemet (ez lesz majd az új gyökér), majd szétvágja úgy az aktuális gyökérelemet, hogy a középső érték ebbe az új csúcsba kerüljön. Ezt követi a gyökér változó aktualizálása.

A kezdeti karbantartó funkciókat követően, a fentiek től függetlenül elindítjuk magát a beszúrást végző rekurziót a 13. sorban. Az eljárásnak átadjuk paraméterként a fa aktuális gyökerét, illetve a beszúrandó értéket.

A 14.2 eljárás mutatja a rekurziót, ami szintről szintre lépkedve megkeresi azt a levelet, ahová majd az újonnan beszúrandó elemet el kell helyezni. A KERESÉSBESZÚR eljárás a rekurzió első szintjén a fa gyökerét kapja paraméterként, amennyiben ez nem levél, akkor megkeresi a tartalmak alapján, hogy melyik irányba kell tovább folytatni a keresést, és rekurzívan meghívja önmagát arra a gyerekre. Amennyiben egy levélhez érkezett, akkor pedig meghívja a BESZÚRÁSLEVÉLBE eljárását, ami elhelyezi az új értéket a levélben. Miután ez megtörtént, végeztünk is a beszúrással, így ezen a szinten további műveletek nem futnak le. A rekurzió szintenként visszatér a gyökérelelmig.

Ha azonban nem levélelemen állunk, akkor biztosan tovább kell majd lépnünk egy szinttel lejjebb. Az 5. sorban található CSÚCSBANKERESÉS eljárás megadja, hogy a csúcsban lévő tartalmak értékei alapján milyen irányban kell majd továbblépni a kereséskor.

14.2. Algoritmus Beszúrandó elem helyének megkeresése és az új tartalom beszúrása

Bemenet: p - $\tilde{\text{BFaElem}}<\mathbf{T}>$ (a rész)fa gyökere, amibe be kell szúrni)

Bemenet: érték - \mathbf{T} (a beszúrandó érték)

```
1: eljárás KERESÉSBESZÚR( $p$ , érték)
2:   ha  $p.\text{levél}$  akkor
3:     BESZÚRÁSLEVÉLBE( $p$ , érték)
4:   különben
5:      $gy \leftarrow \text{CsúcsBANKERESÉS}(p, \text{érték})$ 
6:     ha  $p.gyerek[gy].n = 2t - 1$  akkor
7:       SZÉTVÁGÁS( $p.gyerek[gy], p$ )
8:       ha  $p.tart[gy] < \text{érték}$  akkor
9:          $gy \leftarrow gy + 1$ 
10:        elágazás vége
11:      elágazás vége
12:      KERESÉSBESZÚR( $p.gyerek[gy]$ , érték)
13:    elágazás vége
14:  eljárás vége
```

Felhasznált változók és függvények

- gy : A továbblépéshez szükséges gyerekre hivatkozó segédváltozó.
 - $\text{BESZÚRÁSLEVÉLBE}(p, \text{érték})$: Beszúrja az érték tartalmat a p paraméter által hivatkozott levélbe (14.3. algoritmus).
 - $\text{CsúcsBANKERESÉS}(p, \text{érték})$: Megkeresi a p mutató által hivatkozott csomópontban az érték helyét. Ha már van ilyen tartalom a csomópontban, akkor annak az indexét, ellenkező esetben az első nála nagyobb tartalom indexét adja vissza (vagy $p.n$ -t ha ilyen sincs) (14.4. algoritmus).
 - $\text{SZÉTVÁGÁS}(p, szülő)$: Szétvágja a p mutató által hivatkozott csomópontot, amelyiknek a szülőjére a $szülő$ paraméter hivatkozik (14.5. algoritmus).
-

A tényleges továbblépés előtt azonban ellenőriznünk kell, hogy a folytatáshoz kiválasztott elem nem telített-e. Ezt a 6. sorban látható módon egyszerűen elvégezhetjük, hiszen elég megvizsgálni, hogy a kiszemelt gyereknek hány tartalma van. Amennyiben ez a feltétel igaz, akkor a már említett módon szét kell vágni ezt az elemet. A szétvágás paramétereiként meg kell adni a szétvágandó gyereket, illetve annak szülőjét (ami most az aktuális, p által hivatkozott csúcs), mivel a szétvágás során egy kulcsot fel fog majd hozni ez utóbbiba. Amennyiben történt szétvágás, akkor az eddig gyerekből kettő lett, így újra ellenőriznünk kell, hogy a kettő közül melyikre kell folytatni a rekurzív keresést. Ha a felhozott tartalomnál nagyobb a beszúrandó érték, akkor aktualizálnunk kell a megfelelő gyerek indexet, hiszen nem az eddigi jelölt felé, hanem annak jobb testvére felé (az újonnan létrehozott elem felé) kell folytatni a keresést.

Függetlenül attól, hogy történt-e szétvágás, vagy pedig sem, az eljárás a 12. sorban meghívja önmagát, az előzőleg kikeresett gyerek csúcs felé folytatva a keresést.

14.3.2. Beszúrás segédalgoritmusok

A teljes beszúrás algoritmust darabokra tördelve mutatja be ez a jegyzet, hogy ne vesszen el az alapvető mondanivaló a részletekben. Egy konkrét programnyelvi implementációhoz azonban szükség lehet a legapróbb részleteket megvalósító eljárásokra is, így ezeknek a pszeudokódjai itt, elkülönítve szerepelnek.

Az algoritmusban megjelent a levélbe való beszúrás lépése. Ezt ott külön nem részleteztük, mivel meglehetősen egyszerű műveletről van szó, ennek algoritmusát tartalmazza a 14.3 algoritmus. Alapvető lépései: az új tartalom helyének megkeresése, a mögötte lévő értékek hátrébb csúsztatása (mivel levéről beszélünk, így itt a gyerekmutatókkal nem kell foglalkozni), majd az új tartalom elhelyezése, végül a darabszám változó (n) beállítása.

A tartalom helyének megkereséséhez a logaritmikus keresést használja [10]. A csúcson belüli keresés megvalósítása is egyszerűen lett megvalósítva (mivel ez több helyen is meghívásra kerül, így külön függvénybe került: 14.4. algoritmus).

14.3. Algoritmus Beszúrás egy szabad helyet tartalmazó levélelembe

Bemenet: p - $\tilde{\text{BFaElem}}<\mathbf{T}>$ (a levélelem, amibe el kell helyezni az új elemet)

Bemenet: érték - \mathbf{T} (az új érték)

```
1: eljárás BESZÚRÁSLEVÉLBE( $p$ , érték)
2:    $i \leftarrow \text{CSÚCSBANKERESÉS}(p, \text{érték})$ 
3:   ciklus  $j \leftarrow p.n$ -től  $i$ -ig
4:      $p.tart[j+1] \leftarrow p.tart[j]$ 
5:   ciklus vége
6:    $p.tart[i] \leftarrow \text{érték}$ 
7:    $p.n \leftarrow p.n + 1$ 
8: eljárás vége
```

Felhasznált változók és függvények

- i, j : egész⁺ segédváltozók.
 - $\text{Csúcsbankerésés}(p, \text{érték})$: Megkeresi a p mutató által hivatkozott csomópontban az érték helyét. Ha már van ilyen tartalom a csúcsban, akkor annak az indexét, ellenkező esetben az első nála nagyobb tartalom indexét adja vissza (vagy $p.n$ -t ha ilyen sincs) (14.4. algoritmus).
-

Végül, az egyik legfontosabb segédeljárás a szétvágást végző algoritmus, ennek működése látható a 14.5. ábrán. A beszúrás során minden gyökér, minden pedig a belső csúcsok szétvágásakor is ez hívódik meg.

Bár az algoritmus meglehetősen hosszú, valójában csak a sokféle adatmozgatás miatt tűnik összetettnek. A szétvágáshoz létre kell hoznunk egy új elemet, ez lesz majd a szétvágandó elem új jobboldali testvére. Egy ciklus segítségével átmásoljuk ebbé a szétvágandó elem utolsó ($t - 1$) darab tartalmát (emiatt is lényeges, hogy a szétvágást csak telített elemeken tudjuk végrehajtani), illetve belső csúcs esetében az ezekhez a kulcsokhoz tartozó gyerekmutatókat. Mivel azonban az új elemet nem tudjuk hova láncolni a szülő tartalmai közé, emiatt a maradék értékek közül a legnagyobbat (tehát az eredeti

14.4. Algoritmus A B-fa egy csomópontjában egy tartalom keresése

Bemenet: p - $\tilde{\text{BFaElem}}<\mathbf{T}>$ (hivatkozás a rész(fa) gyökerére, amiben keresünk)

Bemenet: érték - \mathbf{T} (a keresett érték)

Kimenet: . - egész⁺ (az első nem kisebb tartalom indexe, vagy a tartalmak száma+1)

```
1: függvény Csúcsbankerésés( $p$ , érték)
2:    $e \leftarrow 1$ 
3:    $v \leftarrow p.n$ 
4:   ciklus
5:      $k \leftarrow \lfloor (e + v)/2 \rfloor$ 
6:     ha érték <  $p.tart[k]$  akkor
7:        $v \leftarrow k - 1$ 
8:     különben
9:       ha érték >  $p.tart[k]$  akkor
10:         $e \leftarrow k + 1$ 
11:      elágazás vége
12:    elágazás vége
13:    amíg  $e \leq v \wedge p.tart[k] \neq \text{érték}$ 
14:    ha  $p.tart[k] \geq \text{érték}$  akkor
15:      vissza  $k$ 
16:    különben
17:      vissza  $(k + 1)$ 
18:    elágazás vége
19: függvény vége
```

Felhasznált változók és függvények

- e, v, k : A bináris keresésnél használt egész⁺ típusú segédváltozók.
-

szétvágandó csúcs középső elemét) felvisszük a szülőbe (ami bár meglehetősen egyszerű művelet, de mégis a szülő jelentős átalakítását igényli, hogy helyet csinálunk az új tartalomnak a rendezettség szerinti megfelelő helyen). Ezt követi még néhány technikai lépés, hogy a szükséges változók ([n](#), gyerekmutatók) a megfelelő értékeket tartalmazzák. Ezzel a szétvágás megvalósult.

14.5. Algoritmus B-fa egy csomópontjának szétvágása

Bemenet: p - $\tilde{B}FaElem < T >$ (hivatkozás a szétvágandó elemre)

Bemenet: $szülő$ - $\tilde{B}FaElem < T >$ (hivatkozás a szétvágandó elem szülőjére)

```

1: eljárás SZÉTVÁGÁS( $p$ ,  $szülő$ )
2:    $új \leftarrow$  LÉTREHOZ( $BFaElem$ )
3:    $új.levél \leftarrow p.levél$ 
4:    $új.n \leftarrow t - 1$ 
5:   ciklus  $i \leftarrow 1$ -től  $t - 1$ -ig
6:      $új.tart[i] \leftarrow p.tart[t + i]$ 
7:   ciklus vége
8:   ha  $\neg új.levél$  akkor
9:     ciklus  $i \leftarrow 1$ -től  $t$ -ig
10:     $új.gyerek[i] \leftarrow p.gyerek[t + i]$ 
11:    ciklus vége
12:   elágazás vége
13:    $p.n \leftarrow t - 1$ 
14:    $i \leftarrow 1$ 
15:   ha  $szülő.n \neq 0$  akkor
16:     ciklus amíg  $szülő.gyerek[i] \neq p$ 
17:        $i \leftarrow i + 1$ 
18:     ciklus vége
19:     ciklus  $j \leftarrow szülő.n$ -től  $i$ -ig
20:        $szülő.tart[j + 1] \leftarrow szülő.tart[j]$ 
21:     ciklus vége
22:     ciklus  $j \leftarrow szülő.n + 1$ -től  $(i + 1)$ -ig
23:        $szülő.gyerek[j + 1] \leftarrow szülő.gyerek[j]$ 
24:     ciklus vége
25:   elágazás vége
26:    $szülő.tart[i] \leftarrow p.tart[t]$ 
27:    $szülő.gyerek[i] \leftarrow p$ 
28:    $szülő.gyerek[i + 1] \leftarrow új$ 
29:    $szülő.n \leftarrow szülő.n + 1$ 
30: eljárás vége

```

Felhasznált változók és függvények

- $új$: A szétvágáskor létrehozott új elemre hivatkozó $\tilde{B}FaElem < T >$ mutató.
 - i, j : A másolásnál használt **egész**⁺ típusú segédváltozók.
 - **LÉTREHOZ($BFaElem < T >$)** : Létrehoz egy új csúcsot, és visszaad egy hivatkozást erre.
-

14.4. Törlés B-fából

14.4.1. Tartalom eltávolítása

B-fából való törlés szintén egy kereséssel kezdődik, hiszen első körben meg kell találnunk a törlendő tartalmat. A keresés egészen hasonló a bináris keresőfánál tanultakhoz, csak itt igazodnunk kell ahhoz a tényhez, hogy egy csúcson belül több tartalom, és ennek megfelelően kettőnél több gyerek is elhelyezkedhet. A keresés tehát ugyanúgy rekurzív, minden szinten megvizsgáljuk az aktuális csúcsban lévő tartalmakat, amennyiben közük van a törlendő elem, akkor meg is kezdhetjük a törlést, ha nincs, akkor a tartalmak alapján kikereshető, hogy melyik részfa felé kell folytatni a keresést, és a törlést végző eljárást rekurzívan meghívjuk erre.

Miután megtaláltuk a törlendő tartalmat, az alábbi eseteket kell megkülönböztetnünk, attól függően, hogy az milyen csúcsban található (levél, vagy sem), illetve annak a környezete milyen értékeket tartalmaz.

- Levélből való törlés: amennyiben a levélnek van legalább t darab kulcsa, akkor a törlendő tartalom egyszerűen elhagyható. Mivel levélről van szó, nincs szükség a gyerekmutatók karbantartására, ugyanígy nem kell azzal foglalkozni, hogy a törlés milyen módon érinti az ezekből kiinduló részfákat. A törlés tehát lokálisan megoldható (14.6. ábra).
- Belső csúcsból való törlés: amennyiben belső csúcsból kell törölnünk egy x tartalmat, és ennek a csúcsnak van legalább t darab tartalma, akkor belátható, hogy az alábbiak közül legalább az egyik módszer használható:
 - Amennyiben az x kulcshoz tartozó baloldali gyerek tartalmaz legalább t darab tartalmat: keressük meg ennek a részfának a legnagyobb elemét (x'), ezt másoljuk az x helyére, majd rekurzívan folytassuk a törlést ebben a részfában az x' -re (14.7. ábra, illetve konkrétan az áthozatal látható a 14.7c. ábrán).
 - Amennyiben az x kulcshoz tartozó jobboldali gyerek tartalmaz legalább t darab tartalmat: keressük meg ennek a részfának a legkisebb elemét (x'), ezt másoljuk az x helyére, majd rekurzívan folytassuk a törlést ebben a részfában az x' -re (14.8. ábra, illetve konkrétan az áthozatal látható a 14.8c. ábrán).
 - Amennyiben az x kules bal- és jobboldalán is a fentiekkel kevesebb érték van, akkor belátható, hogy a két gyerek összevonható az x kulccsal. Végezzük el ezt az összevonást, majd az újonnan kapott elemben rekurzívan töröljük az x -et (14.9. ábra, illetve konkrétan az összevonás látható a 14.9d. ábrán).

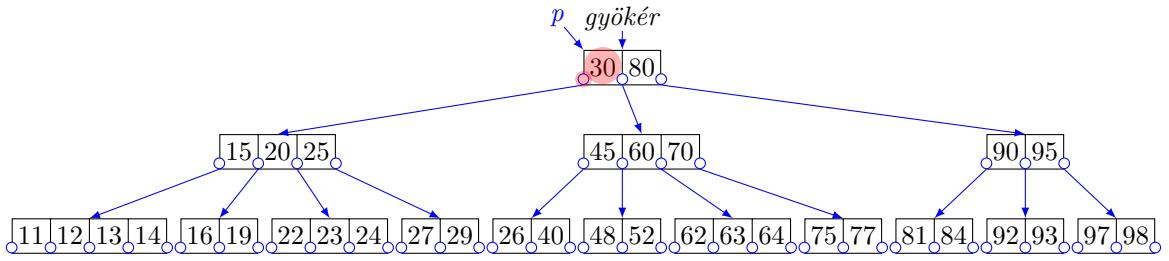
A legutolsó esethez megállapíthatunk egy speciális esetet is, amikor a törlendő érték a gyökérben található, és ez egyben az egyetlen tartalom is ebben a csúcsban (a gyökérben nem kell teljesülnie a tartalmakra vonatkozó minimumfeltételnek, így ez elköpzelhető; a belső csúcsokban - ésszerű t értéket választva - ezzel a lehetőséggel nem kell számolnunk). Ebben az esetben is az utolsó pontban megadott folyamatot kell követnünk, azonban egy kisebb módszertással: amikor az x tartalmat levisszük az új, összevonott elemben, akkor tulajdonképpen kivesszük az utolsó kulcsot is a gyökérből. Ilyenkor ezt a már üres elemet megszüntetjük, és az új összevonott csúcs lesz a B-fa új gyökere.

Fontos észrevennünk, hogy a törlés során eddig használt módszereink mindenkorral a feltételezéssel, hogy a törlendő x értéket tartalmazó belső csúcsnak/levélnek legalább t darab tartalma van. Ez természetesen nem minden esetben áll fenn, ilyen esetekben ezt nekünk kell biztosítanunk, erre visszatérünk a 14.4.2. fejezetben.

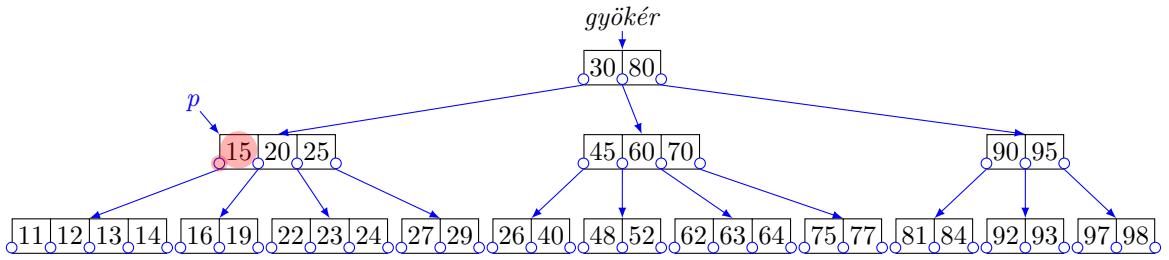
14.4.2. Legalább t darab tartalom biztosítása

Az előző fejezetben látottuk a kulcs törlésének módját, azonban végig éltünk azzal a feltételezéssel, hogy a törlendő értéket tartalmazó levélnek van legalább t darab tartalma, tehát a minimumfeltételhez képest legalább eggyel több (a gyökér esetében a minimumfeltétel nem ad korlátot, így itt elég ha csak 1 darab tartalom található).

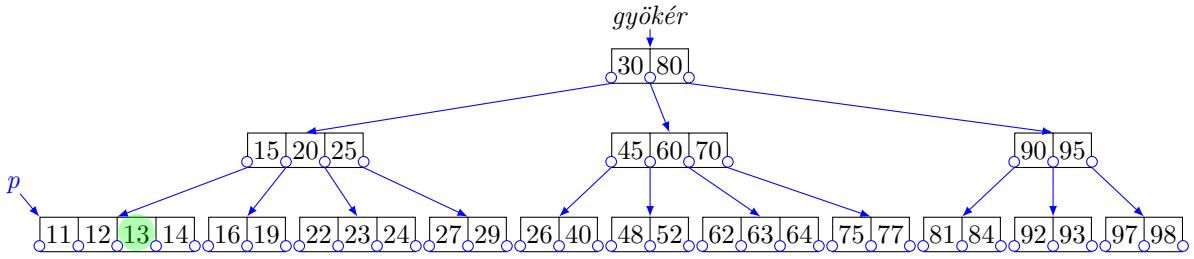
A beszúrásnál már megismertünk egy technikát, amivel már a beszúrás helyének keresése közben tudtuk biztosítani, hogy maga a beszúrás mindenkorral olyan levére jusson, ahol van legalább egy szabad



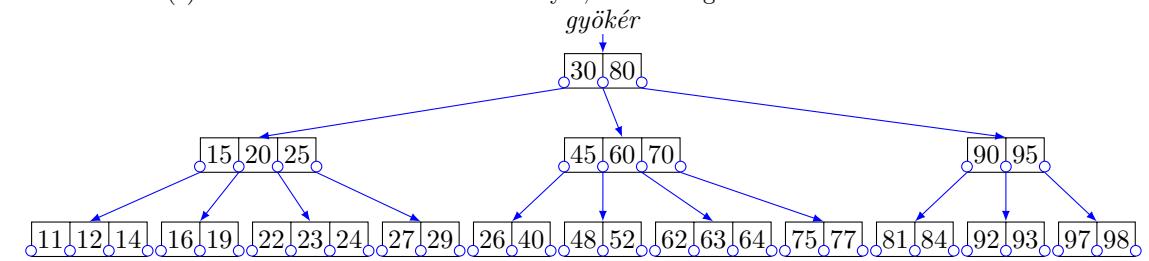
(a) Kezdetben a p a gyökérelemre hivatkozik. A csúcsban keresés megadja, hogy a keresett elem nincs itt, illetve azt is, hogy milyen irányba kell tovább folytatni a törlendő elem keresését.



(b) Újra tovább kell lépni a rendezettségnek megfelelő gyerekelemre.

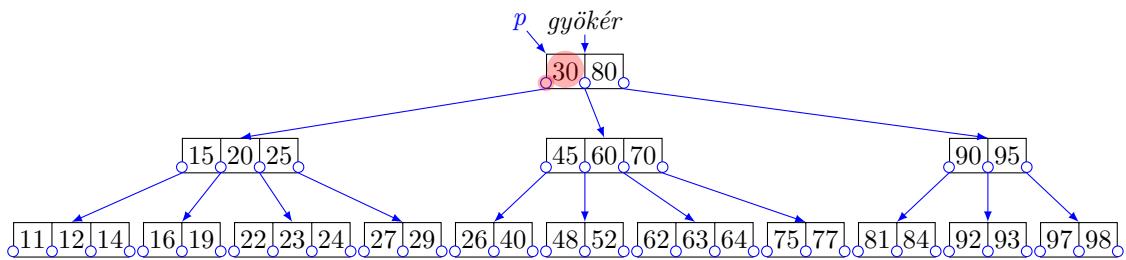


(c) A belső keresés itt már eredményes, hiszen megtalálta a törlendő értéket.

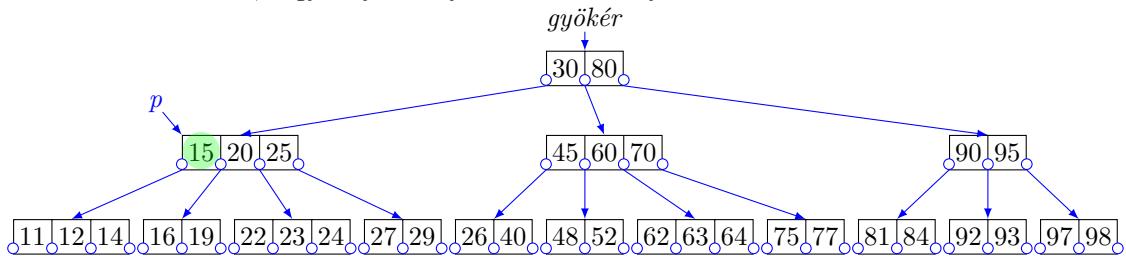


(d) Mivel a törlendő érték egy levélemben található, aminek több, mint $(t - 1)$ darab tartalma van, így az egyszerűen elhagyható.

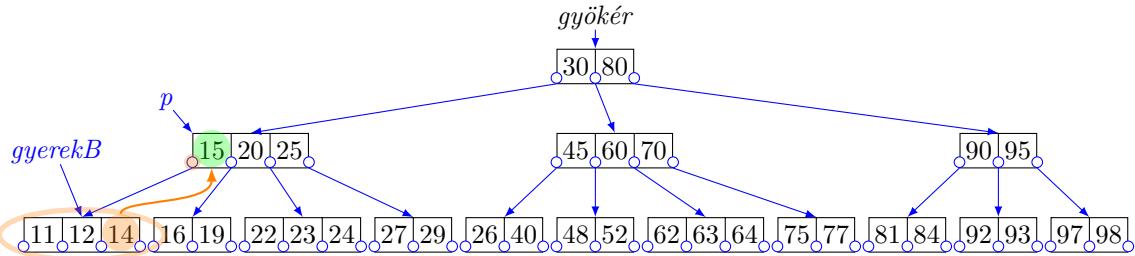
14.6. ábra. Törlés a B-fa leveléből (13-as elem).



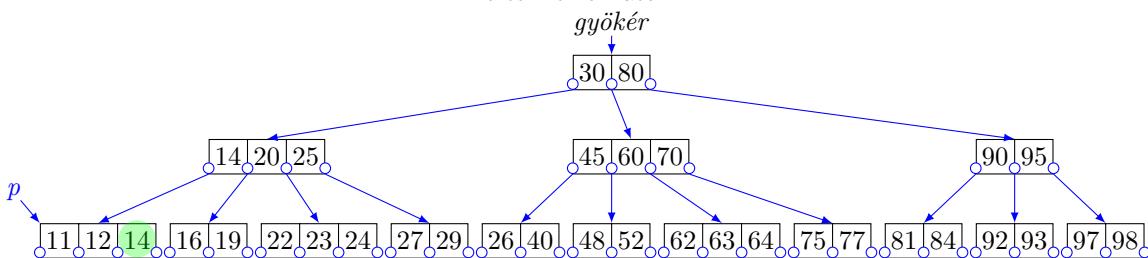
(a) Kezdetben a p a gyökérelemre hivatkozik. A csúcsban keresés megadja, hogy a keresett elem nincs itt, illetve azt is, hogy milyen irányba kell tovább folytatni a törlendő elem keresését.



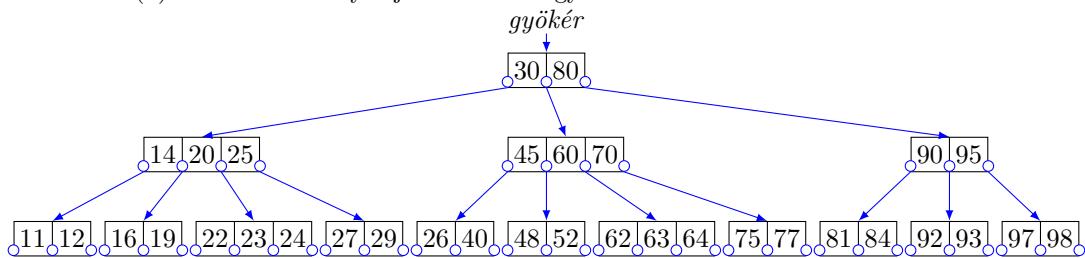
(b) A p továbblépése után a belső keresés megtalálja a törlendő értéket, látható, hogy ez nem levélelem, tehát nem lehet egyszerűen csak elhagyni azt.



(c) A törlendő értékez tartozó baloldali gyerekek több mint ($t - 1$) darab tartalma van, emiatt innen egy érték felhozható.

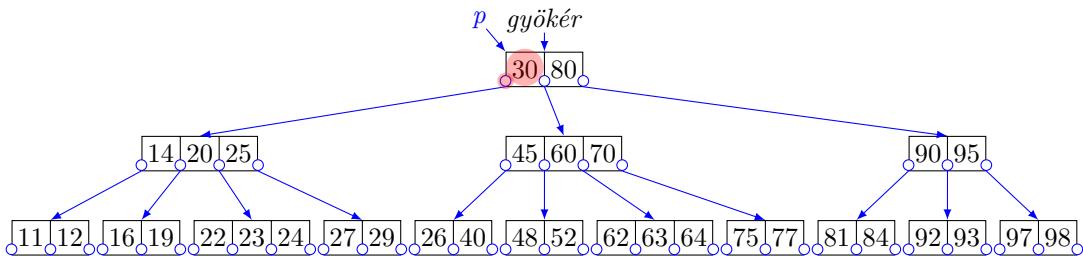


(d) Ezt követően folytatjuk a törlést a gyerekelemen a felhozott tartalomra.

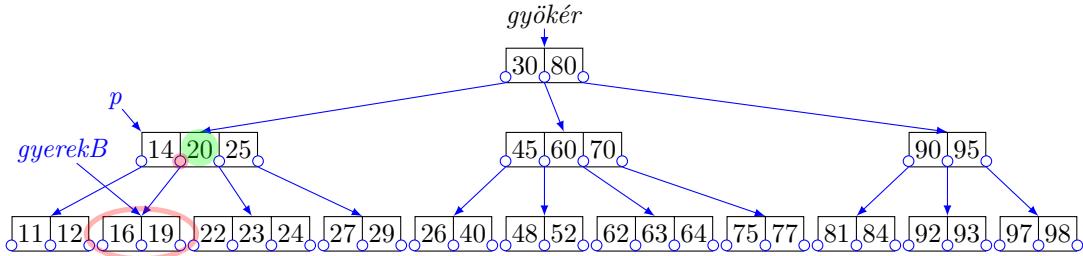


(e) Mivel ez a gyerek egy levélelem, így a törlés egyszerűen megoldható.

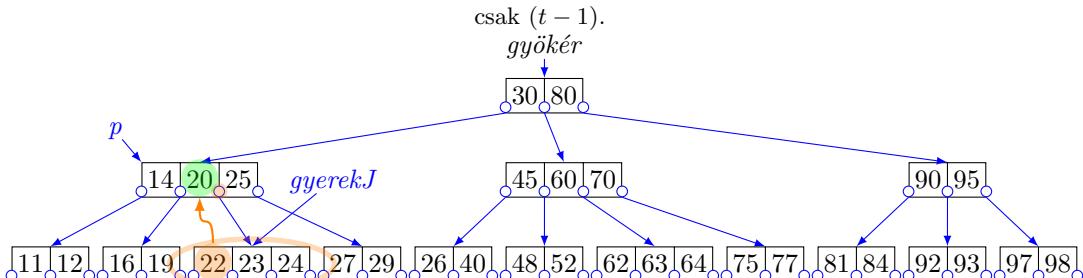
14.7. ábra. Törlés a B-fa belső csúcsából, bal gyerekből felhozás (15-ös elem).



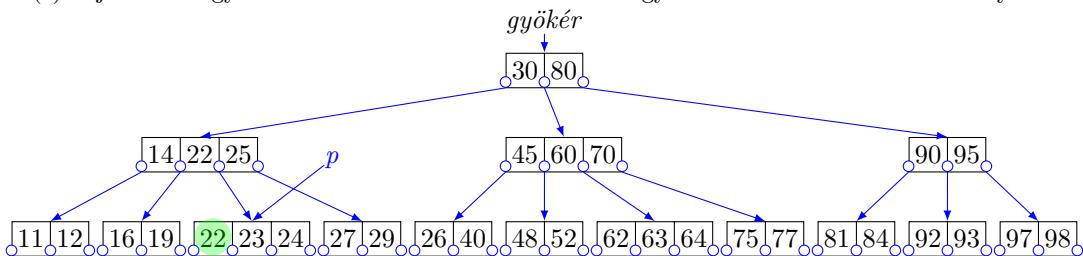
(a) Kezdetben a p a gyökérelelemre hivatkozik. A csúcsban keresés megadja, hogy a keresett elem nincs itt, illetve azt is, hogy milyen irányba kell tovább folytatni a törlendő elem keresését.



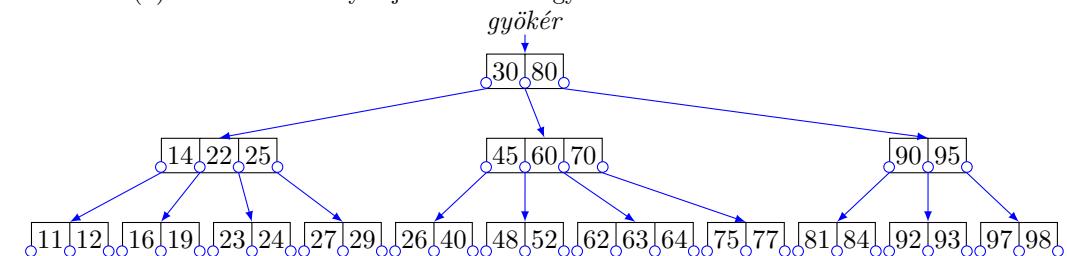
(b) Megtaláltuk a törlendő értéket, látható, hogy ez nem levélelem, tehát nem lehet egyszerűen csak elhagyni azt. Mint látható, a baloldali gyereknél nincs felhozható elem, hiszen a tartalmak száma itt csak $(t - 1)$.



(c) A jobboldali gyerekből viszont fel lehet hozni ismét egy tartalmat a törlendő érték helyére.

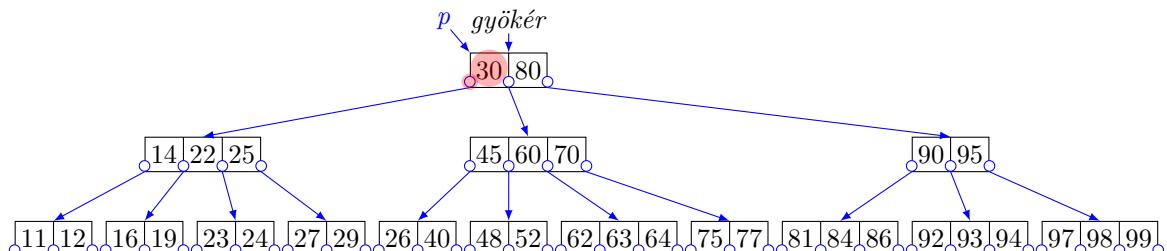


(d) Ezt követően folytatjuk a törlést a gyerekelemen a felhozott tartalomra.

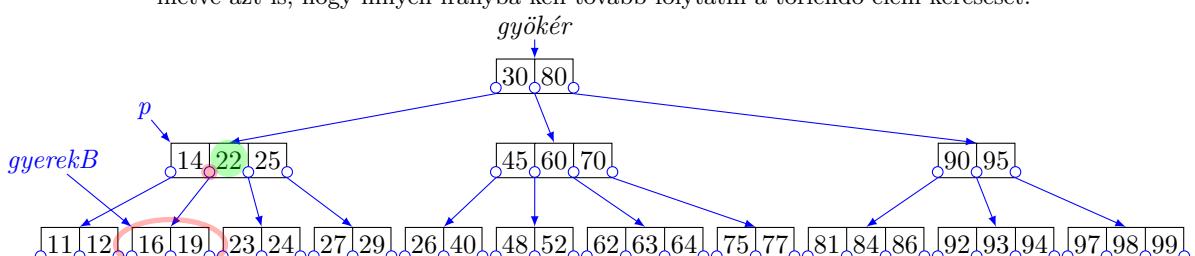


(e) Mivel ez a gyerek egy levélelem, így a törlés egyszerűen megoldható.

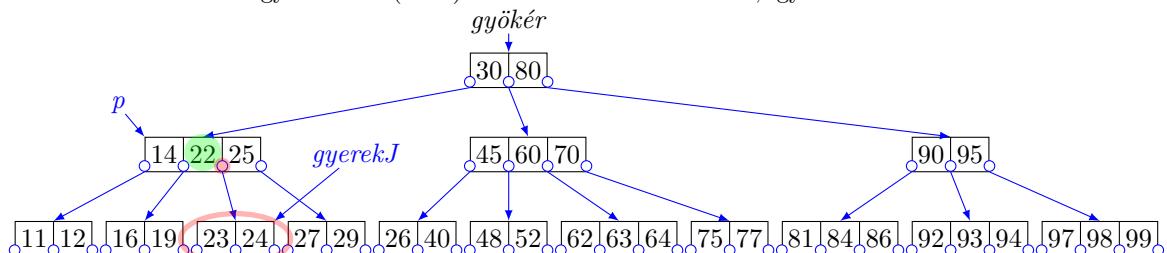
14.8. ábra. Törlés a B-fa belső csúcsából, jobb gyerekből felhozás (20-as elem).



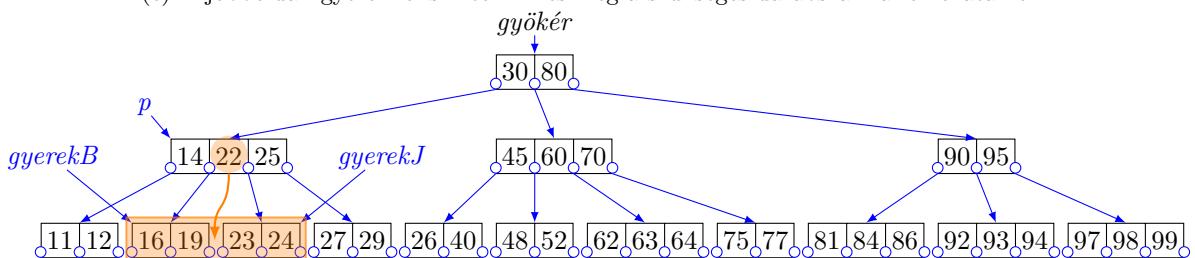
(a) Kezdetben a p a gyökérelemlére hivatkozik. A csúcsban keresés megadja, hogy a keresett elem nincs itt, illetve azt is, hogy milyen irányba kell tovább folytatni a törlendő elem keresését.



(b) Megtaláltuk a törlendő értéket, látható, hogy ez nem levélelem, tehát nem lehet egyszerűen csak elhagyni azt. A baloldali gyerek csak $(t - 1)$ darab tartalmat tartalmaz, így innen nem lehet felhozni.

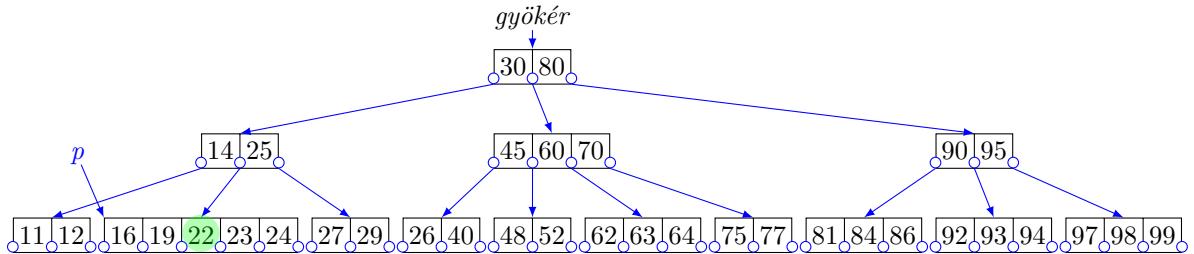


(c) A jobboldali gyereknél szintén nincs meg a szükséges darabszám a felhozatalhoz.

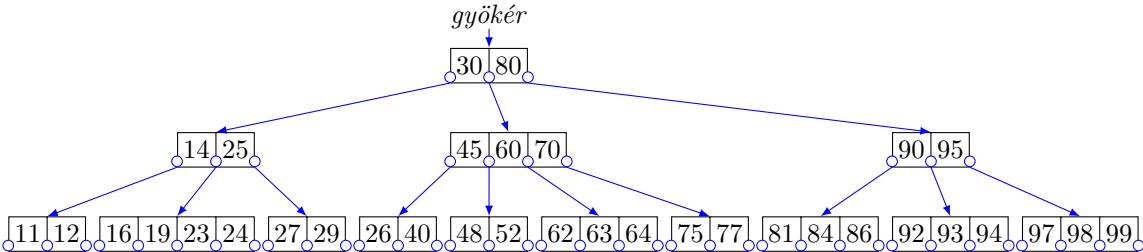


(d) Következő lépésként emiatt összevonjuk a két gyereket, és levisszük az új elemben a törlendő tartalmat is.

14.9. ábra. Törlés a B-fa belső csúcsából, két gyerek összevonása (22-es elem).



(e) Ezt követően rekurzívan folytatjuk a törlést az új elemen a lehozott tartalomra.



(f) A p által mutatott elem egy levél, emiatt a törlés csak az elem elhagyását jelenti.

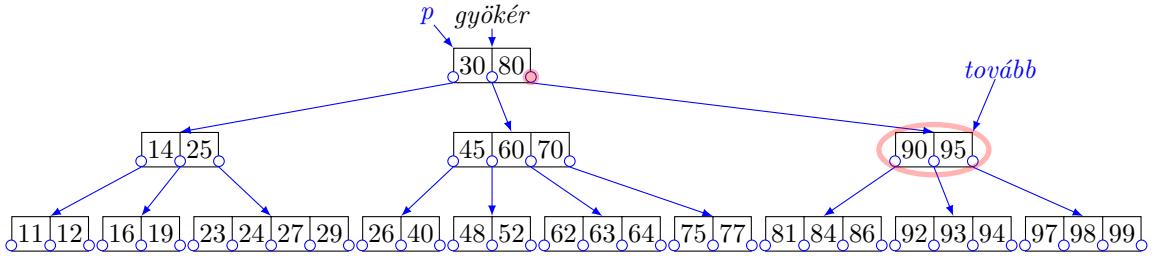
14.9. ábra. Törlés a B-fa belső csúcsából, két gyerek összevonása (22-es elem). (folytatás)

tartalomnak hely. Ezt úgy oldottuk meg, hogy a keresést végző rekurzió minden részfára (gyerekre) való újrahívás előtt ellenőrizte, hogy a továbblépésre megjelölt csúcs teljesíti-e ezt a feltételt, mert ha nem, akkor azt szétvágva tudtuk garantálni a szükséges elemszámot.

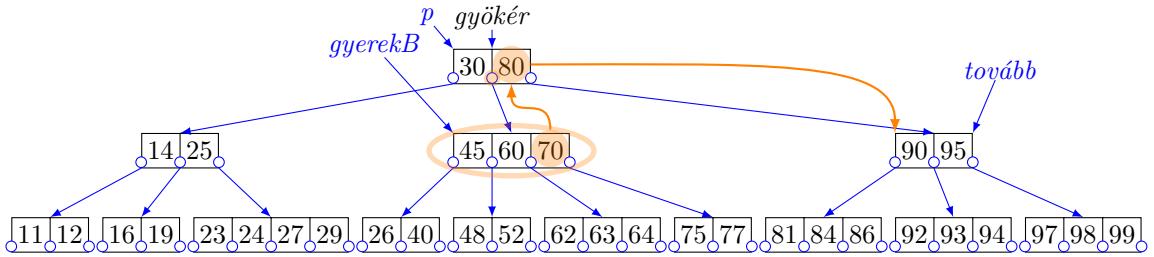
A törlésnél hasonlóan fogunk eljárni, csak itt éppen ellenkező követelményt kell garantálnunk, a gyökér kivételével minden egyes elemnek legalább t darab tartalmat kell tartalmaznia. A beszúráshoz hasonlóan itt is már a keresési fázisban átalakítjuk a fát, amennyiben az szükséges a feltétel biztosításához. Tehát mielőtt a keresést végző rekurzió meghívna önmagát, ellenőri, hogy a folytatáshoz kiszemelt elem (mivel a pszeudokódban erre a **tovább** mutatóval hivatkozunk, hívjuk így a későbbiekben itt is) rendelkezik-e megfelelő számú kulccsal. Amennyiben nem, akkor könnyen belátható, hogy az alábbi három módszer közül valamelyik minden esetben használható:

- A **tovább** elem baloldali testvérének van legalább t darab tartalma: ebben az esetben a két testvérhez tartozó szülőbeli kulcsot levisszük a **tovább** elem első kulcsaként, a baloldali testvér legnagyobb értékét pedig felvisszük a szülőbe ennek a helyére. Ezt követően a baloldali testvér legjobboldalibb gyerekét átláncoljuk a **tovább** csúcs legbaloldalibb gyerekeként. Ezt a folyamatot mutatja a 14.10. ábra, illetve magát a tartalom mozgatást a 14.10b. ábra.
- A **tovább** elem jobboldali testvérének van legalább t darab tartalma: ebben az esetben a két testvérhez tartozó szülőbeli kulcsot levisszük a **tovább** elem utolsó kulcsaként, a jobboldali testvér legkisebb értékét pedig felvisszük a szülőbe ennek a helyére. Ezt követően a jobboldali testvér legbaloldalibb gyerekét átláncoljuk a **tovább** csúcs legjobboldalibb gyerekeként. Ezt a folyamatot mutatja a 14.11. ábra, illetve magát a tartalom mozgatást a 14.11c. ábra.
- A **tovább** elem minden testvérének csak $(t - 1)$ darab tartalma van: ebben az esetben az egyik gyerek (a mi példánkban a baloldali) összevonható a **tovább** csúccsal, úgy, hogy az elembe levisszük a szülő rájuk hivatkozó kulcsát is. Ezt a folyamatot mutatja a 14.12. ábra. Amennyiben nincs baloldali gyerek, akkor ugyanezt megtehetjük a jobboldalival is.

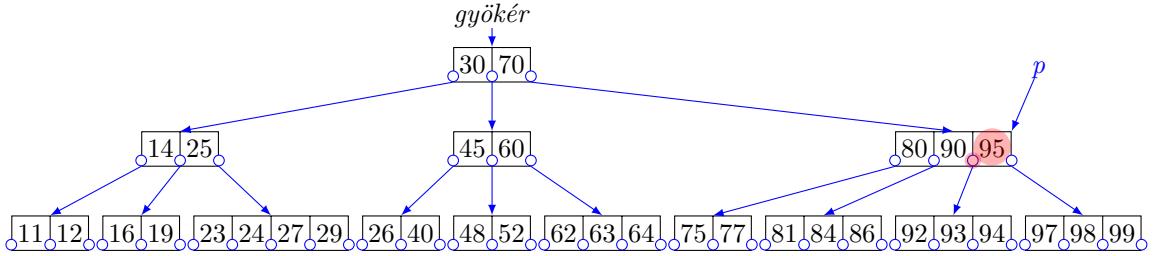
Az utolsó esetben a szülőből le kell vinnünk egy tartalmat az összevont gyerekekből álló csúcsba. Ez akkor eredményezhet problémát, ha ezt a gyökérből kellene levinnünk, ahol csak egy darab tartalmunk van. Emiatt az utolsó esetben itt is előfordulhat, hogy amennyiben a gyökérelemek csak egy tartalma van, és a túl kevés kulcs miatt az ehhez tartozó két gyerekelemet összevonjuk, akkor meg kell szüntetnünk a meglévő gyökért, és az új, összevont csúcsot kell a későbbiekben gyökérnek tekintenünk. Ezt a folyamatot mutatja a 14.13 ábra.



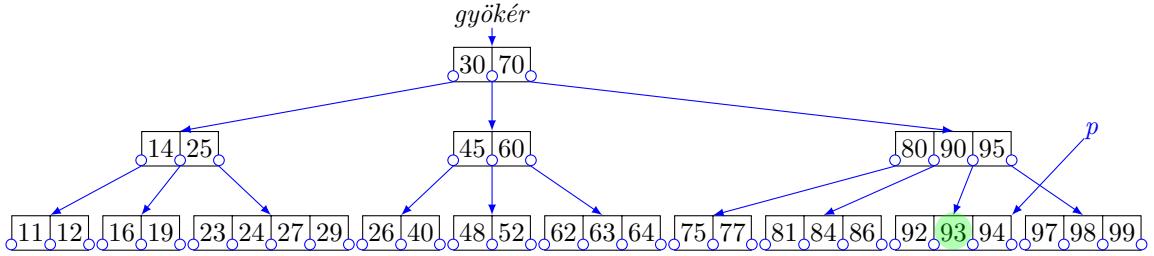
(a) A csúcsban keresés alapján látható, hogy a keresett elem nincs a gyökérelemben. A továbblépés irányára is adott, erre azonban nem tudunk továbbmenni, mivel nincs meg a szükséges t darab tartalom.



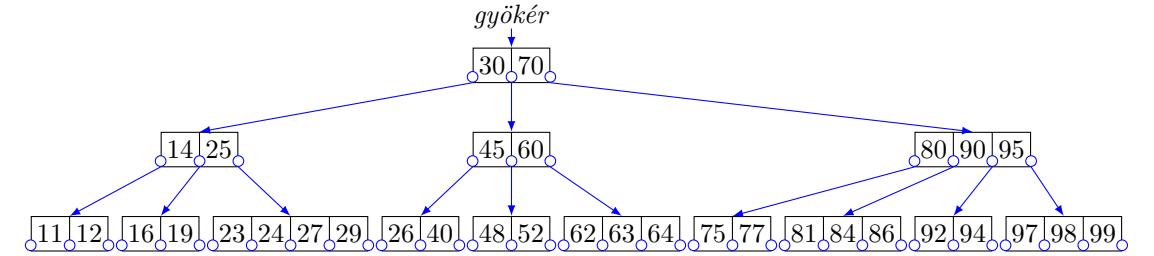
(b) A baloldali testvérben azonban van „felesleg”, ezért innen a legnagyobb értéket felvisszük a szülőbe, majd onnan a megfelelő tartalmat levisszük a tovább által mutatott elembe. A leírtaknak megfelelően átláncoljuk a legjobboldalibb gyereket is.



(c) Az átalakítás után már tovább tud lépni a kereső rekurzió erre az elemre.

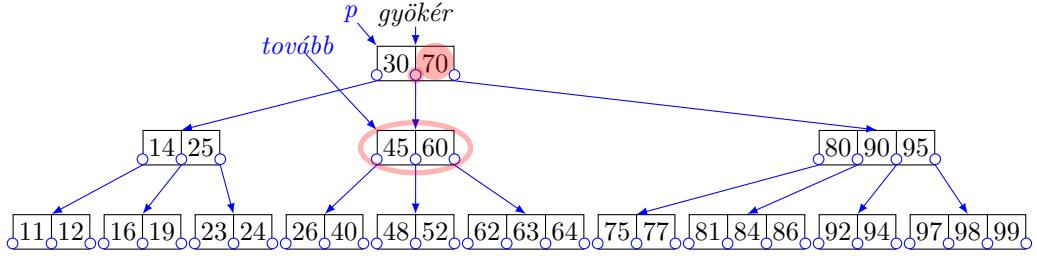


(d) A keresés folytatása ebben a részfában.

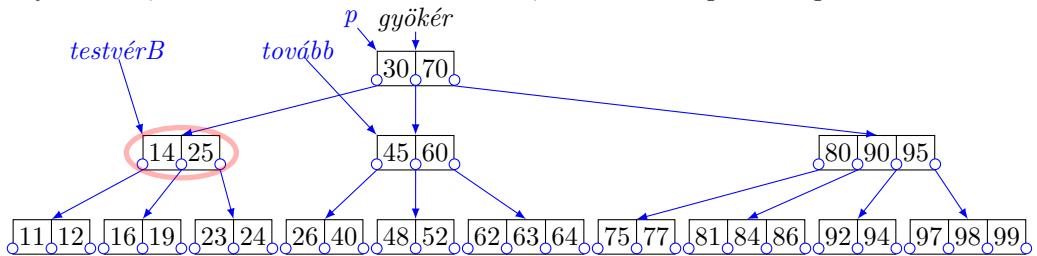


(e) A törlendő elem egy levélben volt, így onnan egyszerűen eltávolítható.

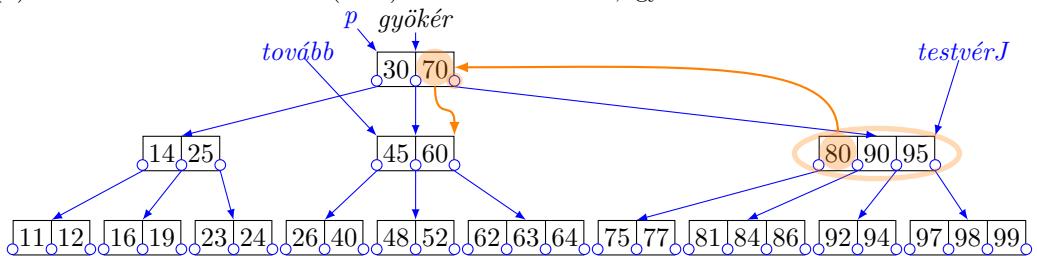
14.10. ábra. Legalább t darab tartalom biztosítása. Elem áthozása baloldali testvérből (93-as elem törlése).



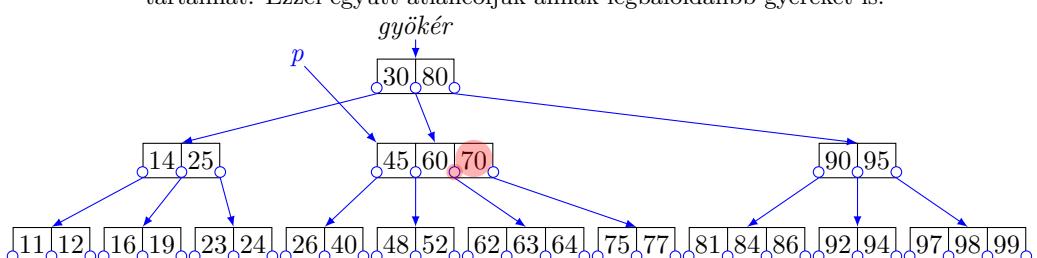
(a) A csúcsban keresés alapján látható, hogy a keresett elem nincs a gyökérelemben. A továblépés irányára is adott, azonban nem tudunk továbbmenni, mivel nincs meg a szükséges t darab tartalom.



(b) A baloldali testvérnek csak $(t - 1)$ darab tartalma van, így innen nem tudunk elemet áthozni.

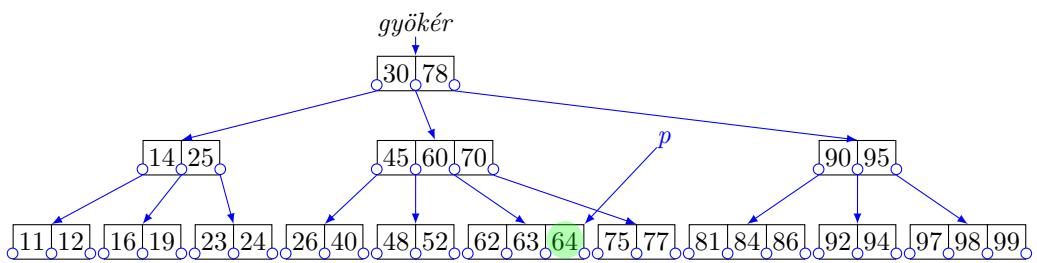


(c) A jobboldali testvérnek több eleme van, így innen a szülőn keresztül át tudunk hozni egy tartalmat. Ezzel együtt átláncoljuk annak legbaloldalibb gyerekét is.

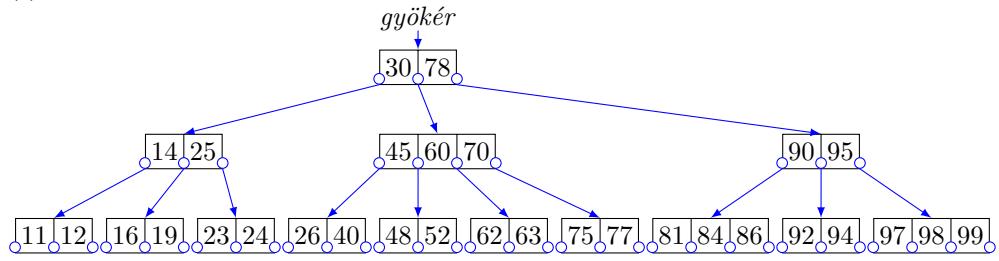


(d) Az átalakítás után már lehet folytatni a keresést a *tovább* által mutatott elemre.

14.11. ábra. Legalább t darab tartalom biztosítása. Elem áthozása jobboldali testvérből (64-es elem törlése).

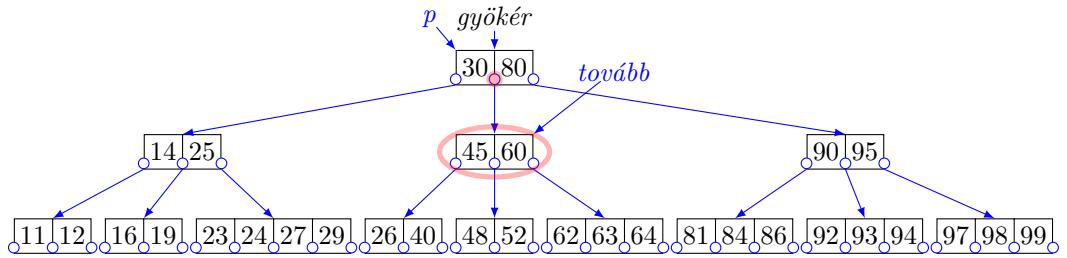


(e) A levélelemben megtaláltuk a törlendő értéket, itt már a törlés egyszerűen megoldható.

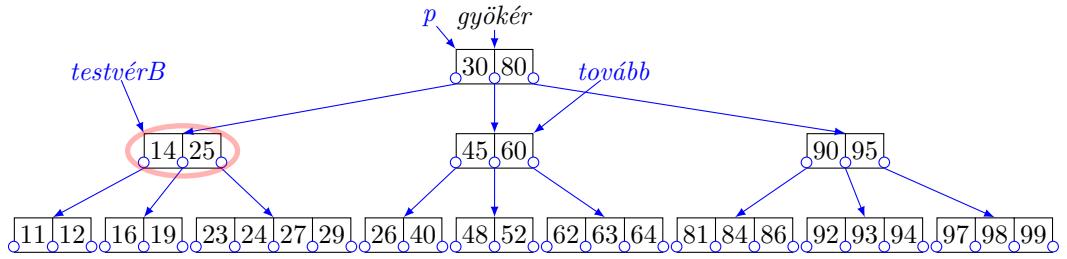


(f) A törlés végeredménye.

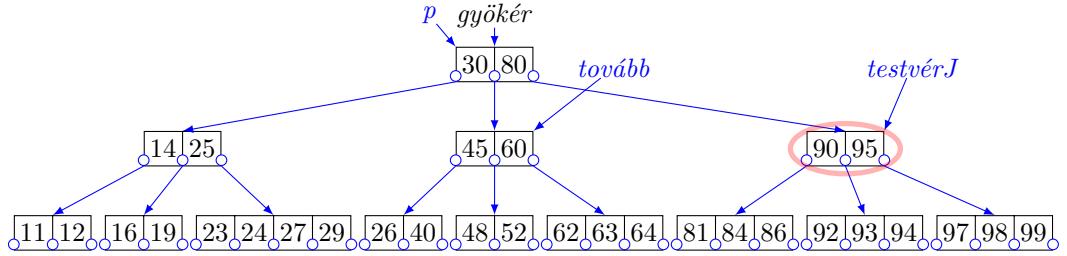
14.11. ábra. Legalább t darab tartalom biztosítása. Elem áthozása jobboldali testvérből (64-es elem törlése). (folytatás)



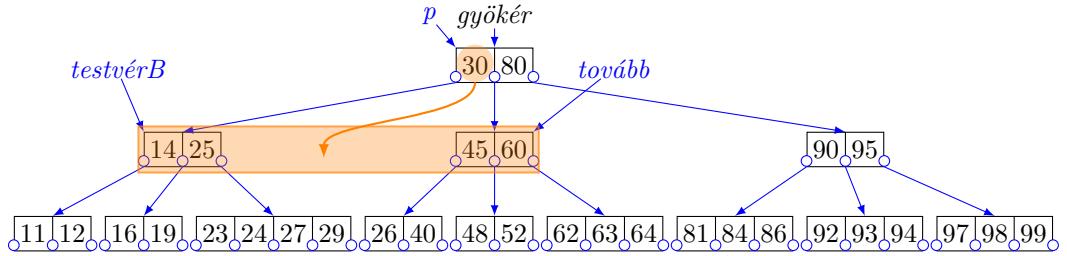
(a) A csúcsban keresés alapján látható, hogy a keresett elem nincs a gyökerelemben. A továblépés irányára is adott, erre azonban nem tudunk továbbmenni, mivel nincs meg a szükséges t darab tartalom.



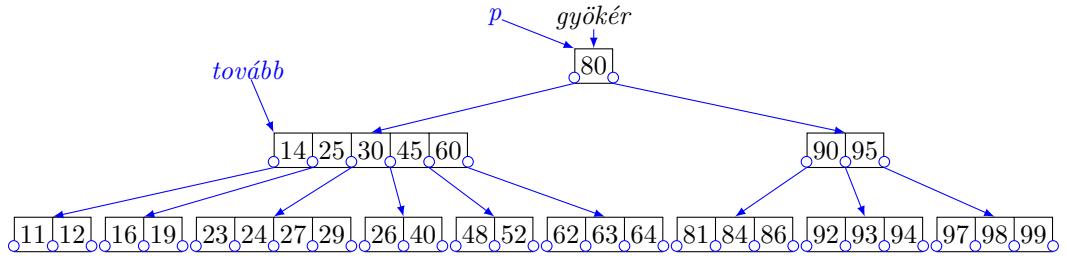
(b) A baloldali testvérnek csak $(t - 1)$ darab tartalma van, így innen tudunk elemet áthozni.



(c) A jobboldali testvérnek is csak $(t - 1)$ darab tartalma van, így innen sem tudunk elemet áthozni.

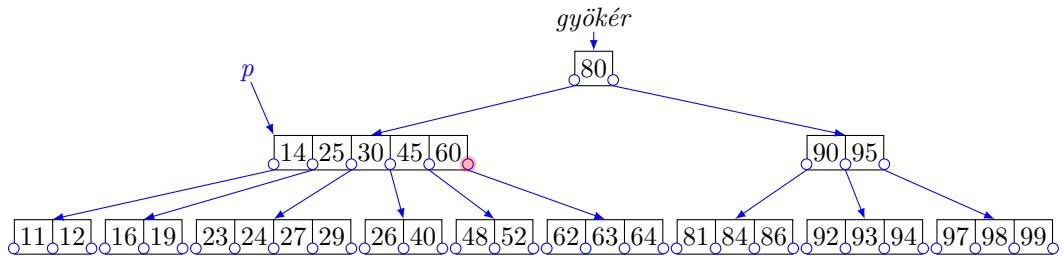


(d) Emiatt összevonjuk az elemet a baloldali testvérrel, ehhez a szülőből le kell hozni egy tartalmat.

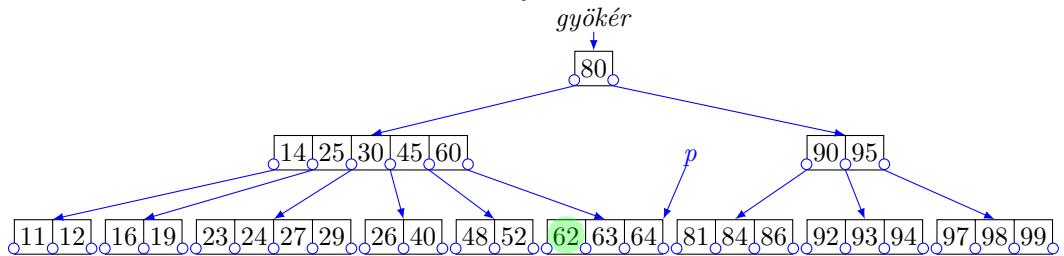


(e) Az összevonást követően az új, összevonott elem lesz a továblépés irányába.

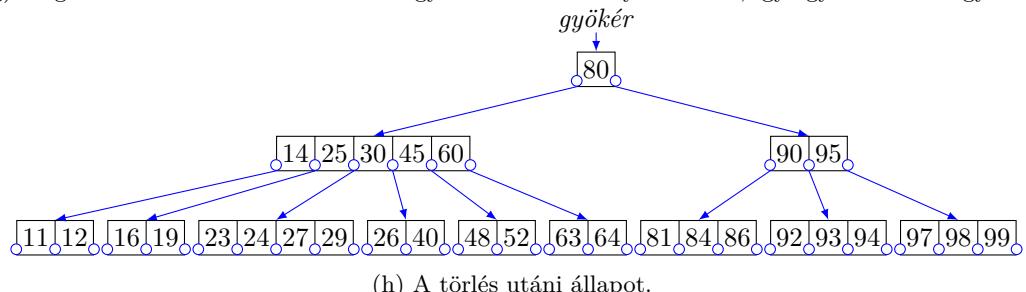
14.12. ábra. Legalább t darab tartalom biztosítása. Testvérek összevonása (62-es elem törlése).



(f) Ezt követően folytatjuk a keresést az összevonott elemen keresztül. Továbblépés a 62-es tartalom irányába.

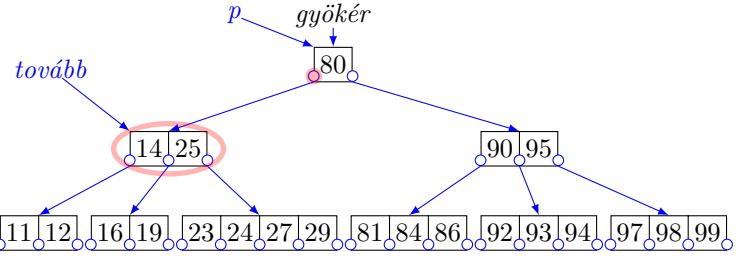


(g) Megtaláltuk a törlendő értéket. Ez egy levélelemben helyezkedik el, így egyszerűen elhagyható.

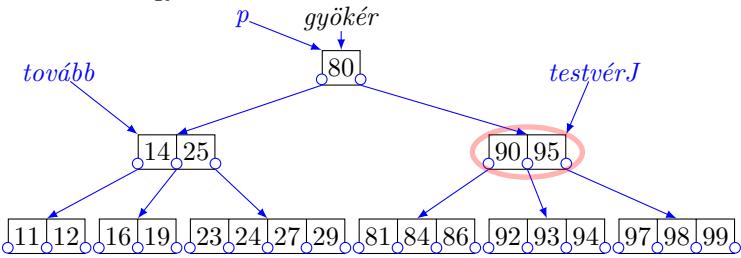


(h) A törlés utáni állapot.

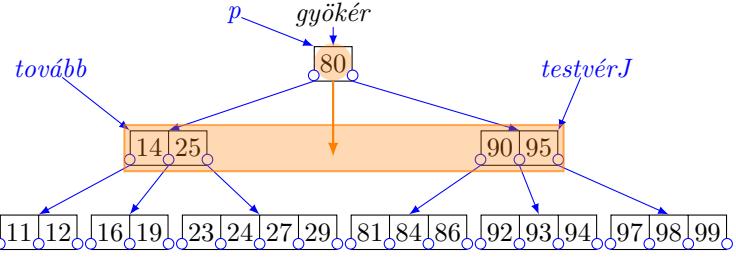
14.12. ábra. Legalább t darab tartalom biztosítása. Testvérek összevonása (62-es elem törlése). (folytatás)



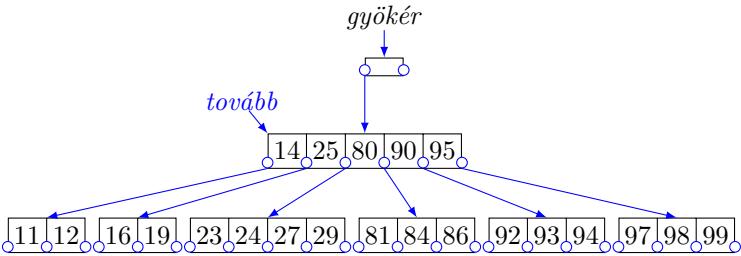
(a) Már a keresés első lépéseinél sem tudunk továbblépni, mivel a következő elemeknek csak $t - 1$ darab tartalma van. Bal szomszédja nincs, így innen nem tudunk tartalmat áthozni.



(b) A jobboldali testvéreknek csak $t - 1$ darab tartalma van, így innen sem tudunk elemet áthozni.

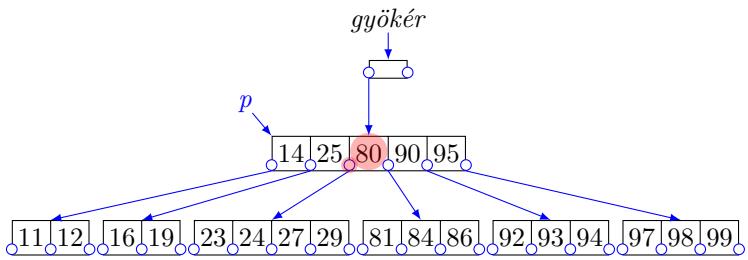


(c) Emiatt összevonjuk az elemet a jobboldali testvérével, ehhez a szülből le kell hozni egy tartalmat.

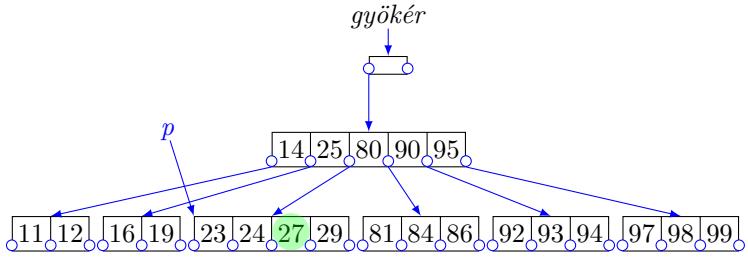


(d) A gyökérben már egy tartalom sincs, a gyerekek száma azonban még így is 1, ez mutat az összevont elemre.

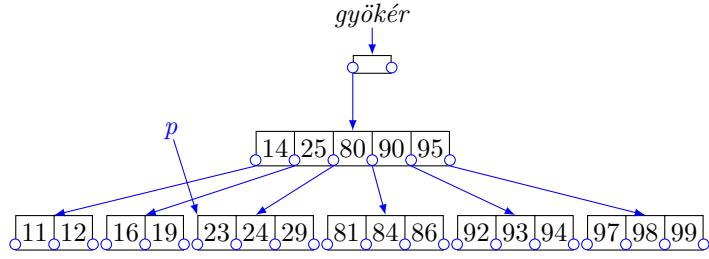
14.13. ábra. Legalább t darab tartalom biztosítása. Testvérek összevonása (27-es elem törlése).



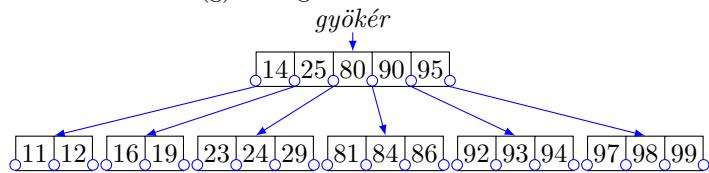
(e) A hiányos gyökért figyelmen kívül hagyva, folytatjuk a törlést. Az elemen belüli kereséssel továbblépünk.



(f) Megtaláltuk a törlendő értéket. Ez egy leveleben helyezkedik el, így egyszerűen elhagyható.



(g) A megtalált elem törlése.



(h) A törlés utáni ellenőrzük a gyökér állapotát. Ha a tartalmak száma 0, akkor az első gyerek mutató által hivatkozott elem lesz az új gyökér.

14.13. ábra. Legalább t darab tartalom biztosítása. Testvérek összevonása (27-es elem törlése). (folytatás)

14.4.3. Törlés pszeudokódja

Törlést indító eljárás

A fentiek alapján már megadható a törlés pszeudokódja. A kódban itt is szükség lehet a törlés után némi karbantartásra (pl. amikor a gyökér alatti elemek összevonása miatt meg kell szüntetnünk a már létező gyökérelemet), emiatt itt sem közvetlenül a törlést végző rekurziót hívjuk meg, hanem egy segédeljárást (14.6. algoritmus).

Látható, hogy ez az eljárás azonnal meghívja a rekurziót, paraméterként átadva a B-fa gyökerét. Miután a vezérlés visszaérkezik, meg kell vizsgálni (3. sor), hogy történt-e összevonás a gyökér utolsó tartalmával. Ha igen, akkor a gyökérben lévő értékek száma 0 lesz, míg a gyökérelem első gyereke az összevonás során kialakított (egyébként a régi baloldali) elemre hivatkozik. Ez jól látható a 14.13. ábrán is. Az, hogy az összevonás melyik lépésben történt, valójában lényegtelen, itt már csak az lényeges, hogy a visszamaradt egyetlen értéket se tartalmazó gyökérelement kicseréljük annak első gyerekére.

Speciális esetként jelenik meg az az állapot, amikor már az egész B-fa csak egyetlen értéket tartalmazott, és ezt törlöttük. Az algoritmus lépéseiit végigkövetve látható, hogy ilyenkor a gyökérben lévő utolsó tartalom is törlődik, az első gyerek mutató értéke pedig \emptyset lesz. Ezt betöltve a *gyökér* változóba, helyesen egy üres B-fát kapunk.

14.6. Algoritmus Tartalom törlése a B-fából

Bemenet: érték - T (a törlendő érték)

- 1: eljárás TÖRLÉS(érték)
 - 2: KERESÉSTÖRÖL(gyökér, érték)
 - 3: ha gyökér.n = 0 akkor
 - 4: gyökér \leftarrow gyökér.gyerek[1]
 - 5: elágazás vége
 - 6: eljárás vége
-

Felhasznált változók és függvények

- *gyökér* : A B-fa gyökere. A törlendő elem helyzetétől függően esetleg meg is változhat az eljárás végén.
 - *KERESÉSTÖRÖL(p, érték)* : Megkeresi az érték tartalom helyét a p gyökerű B-fában, majd elvégzi annak törlését (14.7. algoritmus).
-

Törlést végző rekurzió

Az előbbi indító eljárásnál természetesen jóval összetettebb magát a keresést, törlést, majd az ezt követő karbantartást végző rekurzív eljárás. Ezt mutatja be a 14.7. algoritmus.

Magát az eljárást három fő részre oszthatjuk:

- a törlendő tartalom megkeresése,
- a törlendő tartalom eltávolítása,
- keresés közben a legalább t darab tartalom biztosítása.

Az eljárás első sora pusztán azt ellenőrzi, hogy a törlendő elem benne lehet-e a fában. Amennyiben a p paraméter értéke \emptyset , az azt jelenti, hogy vagy eljutottunk a fa aljára, és nem találtuk meg az elemet, vagy pedig el se indult a rekurzió, mivel már eleve egy üres fán indítottuk el a törlést. Mindkét esetben hibát jelzünk.

Ezt követi a szokásos keresés. A rekurzió minden szintjén megvizsgáljuk, hogy az ott lévő tartalmak között szerepel-e a törlendő. A csomópontron belüli kereséshez a már megismert *CSÚCSBANKERESÉS* függvényt használja. Ennek eredménye alapján a 4. sorban ellenőrzi, hogy a visszakapott index a törlendő értéket mutatja-e.

Amennyiben igen, akkor meg kell kezdenünk a tartalom törlését. Ez egy vizsgálattal kezdődik (5. sor), hogy egy levélben van-e a megtalált elem. Ha igen, akkor egyszerűen töröljük ezt a tartalmat (a későbbiekben részletezett módszernek köszönhetően biztosak lehetünk benne, hogy legalább t darab érték van

ebben a levélben, gyerekei pedig értelemszerűen nem lehetnek). A tartalmakat előrébb kell tolni, majd csökkenteni az *n* értékét.

Amennyiben megtaláltuk az elemet, azonban az egy belső csúcsban van, akkor a fent leírtak szerint kell eljárnunk, a háromféle módszert egyesével megvizsgáljuk, hogy használhatók-e ebben az esetben. A 12. sor vizsgálja meg, hogy a baloldali gyereknek van-e legalább *t* darab tartalma. Ha igen, akkor megkeressük az ebből a gyerekből kiinduló részfa legnagyobb elemét. Ezzel az elemmel felülírjuk a törlendő tartalmat, majd rekurzívan folytatjuk a törlést a részfában, az előbb megtalált értékre.

A másik lehetőség, hogy a jobboldali gyereknek van legalább *t* darab tartalma, ezt ellenőrzi a 17. sor. Amennyiben a feltétel igaz, akkor az előzőhöz hasonló műveletek futnak le, csak éppen ellenkező irányba. Megkeressük a jobboldali gyerekből kiinduló részfa legkisebb elemét. Ezzel felülírjuk a törlendő tartalmat, majd rekurzívan folytatjuk a törlést a jobboldali részfában, az előbb megtalált értékre.

Amennyiben a fenti kettő közül egyik feltétel sem teljesült, akkor a harmadik módszert kell használnunk (hiszen ilyenkor mindenki gyerek pontosan (*t* – 1) elemmel rendelkezik), ami a 21. sorban indul. Ebben az esetben összevonjuk a két gyereket a *p*-beli *i*. tartalommal (ami nem más, mint a törlendő érték indexe). Ezt követően folytatjuk a törlést az új csomópont által meghatározott részfában, továbbra is az eredeti törlendő tartalmat keresve.

Érdemes észrevenni, hogy a három közül bármelyik módszert is alkalmaztuk, minden esetben folytatjuk a rekurziót. A belső csúcsokból ugyanis nem tudjuk a törlést megvalósítani egy lokális módosítással, csak egy szinttel lejjebb tudtuk tolni a problémát. Persze mindezt egymás után többször alkalmazva előbb-utóbb eljutunk a levél elemekig, ahol pedig már egyszerűen befejezhető a törlési művelet.

A törlést végző eljárás harmadik fontos szerepe a legalább *t* darab tartalom folyamatos biztosítása a rekurzió újabb szintjének átadott csúcson. Ennek módszereit tartalmazza a 14.4.2. fejezet, ugyanazokat a technikákat láthatjuk a kódban is. A 27–29. sorok beállítják néhány segédváltozó értékét a későbbi egyszerűbb használat érdekében. A *tovább* hivatkozás mutat arra az elemre, ahová a rekurzió következő szintjén lépnünk kellene, a *testvérB* és *testvérJ* változók pedig ennek a bal- illetve jobboldali testvéreire hivatkoznak (feltéve persze, ha ilyenek léteznek, ha nem, akkor értékük *Ø* lesz).

Ezt követi az első vizsgálat, hogy a baloldali testvérnek van-e legalább *t* darab tartalma (31. sor). Amennyiben van, akkor a 14.10b. ábrán látható módon áthozunk egy tartalmat a baloldali testvértől.

Ha ez a feltétel nem teljesül, akkor megvizsgáljuk a jobboldali testvérét, hogy annak van-e legalább *t* darab tartalma (34. sor). Amennyiben van, akkor az előzőhöz hasonló műveletet végezzük el, csak éppen a másik irányba: áthozunk egy tartalmat a jobboldali testvértől (miként ez a 14.11c. ábrán is látható).

Amennyiben egyik feltétel sem teljesült, akkor biztosak lehetünk benne, hogy mindenki testvében (amennyiben léteznek) pontosan *t* – 1 darab tartalom található. Ilyenkor az összevonást kell választanunk (14.12. ábra). Mindegy, hogy melyik testvérrel vonunk össze, ezért az eljárásban található elágazás csak azt ellenőrzi, hogy van-e egyáltalán baloldali testvér, mert ha van, akkor azzal von össze, ellenkező esetben pedig a jobboldalival. Az összevonás során minden a baloldali elembe másoljuk a tartalmakat a jobboldaliból, emiatt a *testvérB* és *tovább* elemek összevonását követően minden tartalom a *testvérB* csúcsba kerül. Ilyenkor a *tovább* változó értékét módosítanunk kell, hogy a keresés itt folytatódjon (39. sor).

A különböző módosítások után (ha egyáltalán szükség volt ezek közül bármelyikre is) bár biztosak lehetünk benne, hogy a *tovább* által mutatott csúcsnak legalább *t* darab tartalma van. Így folytathatjuk a rekurzív hívást erre az elemre (46. sor).

14.7. Algoritmus B-fából való törlés rekurzív része (keresés és törlés)

Bemenet: p - $\tilde{B}FaElem^<T>$ (a (rész)fa gyökere, amiből törölni kell)
Bemenet: érték - T (a törlendő érték)

1: eljárás KERESÉSTÖRÖL(p , érték)
2: ha $p = \emptyset$ akkor hiba "Nincs elem"
3: $i \leftarrow \text{CsúcsBANKERESÉS}(p, \text{érték})$
4: ha $i \leq p.n \wedge p.\text{tart}[i] = \text{érték}$ akkor
5: ha $p.\text{levél}$ akkor
6: ciklus $j \leftarrow i$ -től $p.n - 1$ -ig
7: $p.\text{tart}[j] \leftarrow p.\text{tart}[j + 1]$
8: ciklus vége
9: $p.n \leftarrow p.n - 1$
10: különben
11: $gyerekB \leftarrow p.\text{gyerek}[i - 1]$
12: ha $gyerekB.n \geq t$ akkor
13: $p.\text{tart}[i] \leftarrow \text{RÉSZFAMAX}(gyerekB)$
14: KERESÉSTÖRÖL($gyerekB$, $p.\text{tart}[i]$)
15: különben
16: $gyerekJ \leftarrow p.\text{gyerek}[i + 1]$
17: ha $gyerekJ.n \geq t$ akkor
18: $p.\text{tart}[i] \leftarrow \text{RÉSZFAMIN}(gyerekJ)$
19: KERESÉSTÖRÖL($gyerekJ$, $p.\text{tart}[i]$)
20: különben
21: ÖSSZEVONÉSSZÜLŐBÖLLEHOZ($gyerekB$, $gyerekJ$, p , i)
22: KERESÉSTÖRÖL($gyerekB$, érték)
23: elágazás vége
24: elágazás vége
25: elágazás vége
26: különben
27: $tovább \leftarrow p.\text{gyerek}[i]$
28: $testvérB \leftarrow (i > 1 ? p.\text{gyerek}[i - 1] : \emptyset)$
29: $testvérJ \leftarrow (i \leq p.n ? p.\text{gyerek}[i + 1] : \emptyset)$
30: ha $tovább \neq \emptyset \wedge tovább.n < t$ akkor
31: ha $testvérB \neq \emptyset \wedge testvérB.n \geq t$ akkor
32: BALTESTVÉRBÖLÁTHOZ($tovább$, $testvérB$, p , i)
33: különben
34: ha $testvérJ \neq \emptyset \wedge testvérJ.n \geq t$ akkor
35: JOBBTESTVÉRBÖLÁTHOZ($tovább$, $testvérJ$, p , i)
36: különben
37: ha $testvérB \neq \emptyset$ akkor
38: ÖSSZEVONÉSSZÜLŐBÖLLEHOZ($testvérB$, $tovább$, p , $i-1$)
39: $tovább \leftarrow testvérB$
40: különben
41: ÖSSZEVONÉSSZÜLŐBÖLLEHOZ($tovább$, $testvérJ$, p , i)
42: elágazás vége
43: elágazás vége
44: elágazás vége
45: elágazás vége
46: KERESÉSTÖRÖL($tovább$, érték)
47: elágazás vége
48: eljárás vége

Felhasznált változók és függvények

- $i, j, tovább, testvérB, testvérJ$: egész⁺ és $\tilde{B}FaElem^<T>$ típusú segédváltozók.
 - RÉSZFAMAX(p), RÉSZFAMIN(p) : A p gyökerű részfa legnagyobb/legkisebb tartalmának értéke.
-

14.4.4. Törlés segéd algoritmusok

Az áttekinthetőség kedvéért néhány művelet külső (esetenként csak egyszer hívott) függvényben lett megvalósítva.

Ilyenek például a megadott gyökerű részfa legkisebb, illetve legnagyobb elemének kiválasztása (14.9. és 14.8. algoritmusok). Ezek működése hasonló a bináris keresőfánál már tanulthoz, hiszen a fa legkisebb tartalma itt is a legbaloldalibb elem (mindig a levélben), illetve a fa legnagyobb tartalma itt is minden a legjobboldalibb elem lesz (ami nyilvánvalóan szintén levélben fog elhelyezkedni). Ennek megfelelőek ezek az eljárások csak egy egyszerű rekurziót tartalmaznak, ami folyamatosan halad a fában lefelé, amíg meg nem találja a megfelelő csúcsot, majd abban a szükséges tartalmat.

14.8. Algoritmus Adott gyökerű (rész)fa legkisebb tartalma

Bemenet: p - $\tilde{B}FaElem^<T>$ (*hivatkozás a részfa gyökerére, aminek a minimumát keressük*)

Kimenet: $. - T$ (*a legkisebb érték*)

- 1: **függvény** RÉSZFAMIN(p)
 - 2: **ha** $p.\text{levél}$ **akkor**
 - 3: **vissza** $p.\text{tart}[1]$
 - 4: **különben**
 - 5: **vissza** RÉSZFAMIN($p.\text{gyerek}[1]$)
 - 6: **elágazás vége**
 - 7: **függvény vége**
-

14.9. Algoritmus Adott gyökerű (rész)fa legnagyobb tartalma

Bemenet: p - $\tilde{B}FaElem^<T>$ (*hivatkozás a részfa gyökerére, aminek a maximimát keressük*)

Kimenet: $. - T$ (*a legnagyobb érték*)

- 1: **függvény** RÉSZFAMAX(p)
 - 2: **ha** $p.\text{levél}$ **akkor**
 - 3: **vissza** $p.\text{tart}[p.n]$
 - 4: **különben**
 - 5: **vissza** RÉSZFAMAX($p.\text{gyerek}[p.n+1]$)
 - 6: **elágazás vége**
 - 7: **függvény vége**
-

Megjegyzés

Az általunk használt algoritmusok nem optimálisak, például a minimum és maximum keresést követi egy törlés, ami tulajdonképpen ugyanazt az utat járja be. Ha tökéletesíteni szeretnénk a pszeudokódjainkat, akkor célszerű lenne ezeket az eljárásokat összevonni.

A legalább t darab tartalom biztosításához szükséges művelet a baloldali, illetve a jobboldali testvérből való értékek áthozatala. Önmagukban ezek az eljárások nem bonyolultak, azonban a meglehetősen nagy mennyiségű adatmozgatás miatt (három csúcsban is változnak a tart , gyerek , és n értékek) meglehetősen hosszúak. Ezért ezek két eljárásban lettek megvalósítva: 14.10. és 14.11. algoritmusok.

Még szintén a t darab tartalom biztosításához szükséges lehet az elemek összevonását végző eljárás, ezt mutatja be a 14.12. algoritmus. Mint látható, ennek három paramétere van, a bal- illetve jobboldali összevonandó elem, illetve egy hivatkozás ezek közös szüleire és a tartalomra, amelynek ők a bal és jobb oldalán állnak.

Az összevonás tulajdonképpen a baloldali elembe való átmozgatását jelenti, mind a szülőből lehozott új tartalom, mind pedig a jobboldali csúcsból áthozott tartalmak itt bővítik a már meglévő tartalmakat. Értelemszerűen ennek megfelelően növelni kell az n értékét, illetve át kell vinni a szükséges gyerek mutatókat is. A művelet során a szülőből törölni kell a lehozott tartalmat, továbbá a jobboldali elemet is töröljük, hiszen a későbbiekben nincs már rá szükség (a szülőből való tartalom törlést követően már gyerek hivatkozás sincs rá).

14.10. Algoritmus Baloldali testvérből elem áthozása

Bemenet: tovább - $\tilde{B}FaElem < T >$ (a B -fa egy csúcsa, amelyre tovább szeretnénk majd lépni (itt kell biztosítani az elegendő elemszámot) és pontosan t darab tartalma van)

Bemenet: testvérB - $\tilde{B}FaElem < T >$ (a tovább elem baloldali testvére, legalább t darab tartalma van)

Bemenet: szülő - $\tilde{B}FaElem < T >$ (a tovább elem szülője)

Bemenet: i - egész⁺ (a szülőből a tovább csomópontra vonatkozó tartalom indexe)

- 1: eljárás BALTESTVÉRBŐLÁTHOZ(tovább, testvérB, szülő, i)
 - 2: $tovább.n \leftarrow tovább.n + 1$
 - 3: $tovább.gyerek[tovább.n + 1] \leftarrow tovább.gyerek[tovább.n]$
 - 4: ciklus $j \leftarrow tovább.n - 1$ -től 1-ig
 - 5: $tovább.tart[j + 1] \leftarrow tovább.tart[j]$
 - 6: $tovább.gyerek[j + 1] \leftarrow tovább.gyerek[j]$
 - 7: ciklus vége
 - 8: $tovább.gyerek[1] \leftarrow testvérB.gyerek[testvérB.n + 1]$
 - 9: $tovább.tart[1] \leftarrow szülő.tart[i - 1]$
 - 10: $szülő.tart[i - 1] \leftarrow testvérB.tart[testvérB.n]$
 - 11: $testvérB.n \leftarrow testvérB.n - 1$
 - 12: eljárás vége
-

Felhasznált változók és függvények

- j : Egy egész⁺ típusú segédváltozó.
-

14.11. Algoritmus Jobboldali testvérből elem áthozása

Bemenet: tovább - $\tilde{B}FaElem < T >$ (a B -fa egy csúcsa, amelyre tovább szeretnénk majd lépni (itt kell biztosítani az elegendő elemszámot) és pontosan t darab tartalma van)

Bemenet: testvérJ - $\tilde{B}FaElem < T >$ (a tovább elem jobboldali testvére, legalább t darab tartalma van)

Bemenet: szülő - $\tilde{B}FaElem < T >$ (a tovább elem szülője)

Bemenet: i - egész⁺ (a szülőből a tovább csomópontra vonatkozó tartalom indexe)

Bemenet: tovább - $\tilde{B}FaElem < T >$, testvérJ - $\tilde{B}FaElem < T >$, szülő - $\tilde{B}FaElem < T >$, i - egész⁺

- 1: eljárás JOBBTESTVÉRBŐLÁTHOZ(tovább, testvérJ, szülő, i)
 - 2: $tovább.n \leftarrow tovább.n + 1$
 - 3: $tovább.tart[tovább.n] \leftarrow szülő.tart[i]$
 - 4: $szülő.tart[i] \leftarrow testvérJ.tart[1]$
 - 5: $tovább.gyerek[tovább.n + 1] \leftarrow testvérJ.gyerek[1]$
 - 6: ciklus $j \leftarrow 1$ -től $testvérJ.n - 1$ -ig
 - 7: $testvérJ.tart[j] \leftarrow testvérJ.tart[j + 1]$
 - 8: $testvérJ.gyerek[j] \leftarrow testvérJ.gyerek[j + 1]$
 - 9: ciklus vége
 - 10: $testvérJ.gyerek[testvérJ.n] \leftarrow testvérJ.gyerek[testvérJ.n + 1]$
 - 11: $testvérJ.n \leftarrow testvérJ.n - 1$
 - 12: eljárás vége
-

Felhasznált változók és függvények

- j : Egy egész⁺ típusú segédváltozó.
-

14.12. Algoritmus Két elem összevonása, és a szülőből egy kulccsal kiegészítése

Bemenet: bal - $\tilde{B}FaElem<T>$ (*hivatkozás a baloldali összevonandó elemre*)

Bemenet: $jobb$ - $\tilde{B}FaElem<T>$ (*hivatkozás a jobboldali összevonandó elemre*)

Bemenet: $szülő$ - $\tilde{B}FaElem<T>$ (*hivatkozás a szülő elemre*)

Bemenet: i - $egész^+$ (*az összevonáskor lehozandó elem indexe*)

- 1: eljárás ÖSSZEVONÉSSZÜLŐBÖLLEHOZ($bal, jobb, szülő, i$)
 - 2: $bal.n \leftarrow bal.n + 1$
 - 3: $bal.tart[bal.n] \leftarrow szülő.tart[i]$
 - 4: $bal.gyerek[bal.n + 1] \leftarrow jobb.gyerek[1]$
 - 5: **ciklus** $j \leftarrow 1$ -től $jobb.n$ -ig
 - 6: $bal.n \leftarrow bal.n + 1$
 - 7: $bal.tart[bal.n] \leftarrow jobb.tart[j]$
 - 8: $bal.gyerek[bal.n] \leftarrow jobb.gyerek[j + 1]$
 - 9: **ciklus vége**
 - 10: **ciklus** $j \leftarrow i$ -től $szülő.n - 1$ -ig
 - 11: $szülő.tart[j] \leftarrow szülő.tart[j + 1]$
 - 12: **ciklus vége**
 - 13: **ciklus** $j \leftarrow i + 1$ -től $szülő.n$ -ig
 - 14: $szülő.gyerek[j] \leftarrow szülő.gyerek[j + 1]$
 - 15: **ciklus vége**
 - 16: $szülő.n \leftarrow szülő.n - 1$
 - 17: eljárás vége
-

Felhasznált változók és függvények

- j : Egy $egész^+$ típusú segédváltozó.
-

Irodalomjegyzék

- [1] A. V. Aho – J. E. Hopcroft – J. D. Ulmann: *Számítógép-algoritmusok tervezése és analízise*. 1982, Műszaki Könyvkiadó. ISBN 963-10-4323-1.
- [2] T. H. Cormen – C.E. Leiserson – R. L. Rivest: *Algoritmusok*. 2003, Műszaki Könyvkiadó. ISBN 963-16-3029-3.
- [3] Kris Hermans: *Mastering Object-Oriented Programming: A Comprehensive Guide to Learn Object-Oriented Programming*. 2023, Cybellium Ltd. ISBN 979-8-86672-248-8.
- [4] Gy. Y. Katona – A. Recski – Cs. Szabó: *A számítástudomány alapjai*. 2002, Typotex. ISBN 978-963-9664-19-7.
- [5] D. E. Knuth: *A számítógép-programozás művészete - 1. kötet. Alapvető algoritmusok*. 1987, Műszaki Könyvkiadó. ISBN 963-10-7156-1.
- [6] D. E. Knuth: *A számítógép-programozás művészete - 3. kötet. Keresés és rendezés*. 1988, Műszaki Könyvkiadó. ISBN 963-10-7206-1.
- [7] G. Pap – P. Szlávi – L. Zsakó: *mikrológia 27 - Módszeres programozás: Rekurzív típusok*. 1998, ELTE TTK.
- [8] G. Pap – P. Szlávi – L. Zsakó: *mikrológia 34 - Módszeres programozás: Adattípusok*. 1998, ELTE TTK.
- [9] Rinus Plasmeijer – Marko Van Eekelen – MJ Plasmeijer: *Functional programming and parallel graph rewriting*. 857. köt. 1993, Addison-wesley Reading.
- [10] Sz. Sergyán: *Algoritmusok, adatszerkezetek I.* 2014, Óbudai Egyetem.
- [11] P. Szlávi – L. Zsakó: *mikrológia 38 - Módszeres programozás: Gráfok, gráfalgoritmusok*. 2001, ELTE TTK.
- [12] P. Szlávi – L. Zsakó: *mikrológia 19 - Módszeres programozás: Programozási tételek*. 2004, ELTE TTK.
- [13] John Von Neumann: First draft of a report on the edvac (1945). 2021.
- [14] N. Wirth: *Algorithms and Data Structures*. 1985, Prentice Hall. ISBN 0-13022-005-1.
- [15] Csörnyei Zoltán: *Lambda-kalkulus*. 2007, Typotex Kft.

```
namespace ALGA_ZH_A;

class Program
{
    static void Main(string[] args)
    {
        // 1. LancoltLista<string> példa
        LancoltLista<string> lista = new LancoltLista<string>();

        lista.Beszur(0, "alma");
        lista.Beszur(1, "korte");
        lista.Beszur(2, "szilva");

        Console.WriteLine("Beszúrás után a lista elemei:");

        Console.WriteLine($"Elemszám: {lista.Elemszam}");

        lista.Torles("korte");
        Console.WriteLine("Törlés után a lista elemei:");

        Console.WriteLine($"Elemszám: {lista.Elemszam}");

        // 2. SzamoloLancoltLista<string> példa
        SzamoloLancoltLista<string> szamoloLista = new SzamoloLancoltLista<string>();

        // 3.
        szamoloLista.Beszur(0, "alma");
        szamoloLista.Beszur(1, "alma");
        szamoloLista.Beszur(2, "korte");
        szamoloLista.Beszur(3, "szilva");

        Console.WriteLine($"Elemszám beszúrás után: {szamoloLista.Elemszam}");

        szamoloLista.Torles("alma");
        Console.WriteLine($"Elemszám törlés után: {szamoloLista.Elemszam}");

        szamoloLista.Torles("alma");
        Console.WriteLine($"Elemszám végleges törlés után: {szamoloLista.Elemszam}");

        SzamoloLancoltLista<string> szamoloLista2 = new SzamoloLancoltLista<string>();

        szamoloLista2.Beszur(0, "szilvafa");
        szamoloLista2.Beszur(1, "korte");
        szamoloLista2.Beszur(2, "barack");
        szamoloLista2.Beszur(3, "szilvafa");

        Console.WriteLine($"Elemszám módosítás előtt: {szamoloLista2.Elemszam}");

        szamoloLista2.DarabSzamModositas((tartalom, darabszam) =>
        {
            if (tartalom.Length > 5)
                return darabszam * 2;
            return darabszam;
        });
    }
}
```

```
        Console.WriteLine($"Elemszám módosítás után: {szamoloLista2.Elemszam}");  
    }  
}
```

```
namespace ALGA_ZH_A;
```

```
public class LancElem<T>  
{  
    public T tart;  
    public LancElem<T>? kov;  
    public int darabszam;  
  
    public LancElem(T tart, LancElem<T>? kov)  
    {  
        this.tart = tart;  
        this.kov = kov;  
    }  
  
    public LancElem(T tart, LancElem<T>? kov, int darabszam)  
    {  
        this.tart = tart;  
        this.kov = kov;  
        this.darabszam = darabszam;  
    }  
}
```

```
namespace ALGA_ZH_A;
```

```
public class LancoltLista<T>  
{  
    private LancElem<T>? fej;  
    public virtual int Elemszam { get; protected set; }  
  
    public LancoltLista()  
    {  
        this.fej = null;  
        Elemszam = 0;  
    }  
  
    public virtual void Beszur(int index, T ertek)  
    {  
        if (this.fej is null || index == 0)  
        {  
            LancElem<T>? uj = new LancElem<T>(ertek, this.fej);  
            this.fej = uj;  
        }  
        else  
        {  
            LancElem<T>? p = this.fej;  
            int i = 1;  
  
            while (p.kov is not null && i < index)  
            {  
                p = p.kov;  
                i++;  
            }  
            LancElem<T>? uj = new LancElem<T>(ertek, p);  
            p.kov = uj;  
        }  
    }  
}
```

```

    {
        p = p.kov;
        i++;
    }

    if (i <= index)
    {
        LancElem<T>? uj = new LancElem<T>(ertek, p.kov);
        p.kov = uj;
    }
    else throw new Exception("Hibás index!");
}

this.Elemszam++;
}

public virtual void Torles(T ertek)
{
    LancElem<T>? p = this.fej;
    LancElem<T>? e = null;

    do
    {
        while (p is not null && !p.tart.Equals(ertek))
        {
            e = p;
            p = p.kov;
        }

        if (p is not null)
        {
            LancElem<T>? q = p.kov;
            if (e is null)
            {
                this.fej = q;
            }
            else
            {
                e.kov = q;
            }

            p = q;
        }
    } while (p is not null);

    this.Elemszam--;
}
}

```

```

namespace ALGA_ZH_A;

public class SzamoloLancoltLista<T> : LancoltLista<T>
{

```

```

private LancElem<T>? fej;

public override int Elemszam
{
    get
    {
        int darabszam = 0;
        LancElem<T>? p = this.fej;
        while (p is not null)
        {
            darabszam += p.darabszam;
            p = p.kov;
        }
        return darabszam;
    }
}

public override void Beszur(int index, T ertek)
{
    bool vane = false;

    LancElem<T>? elem = this.fej;
    while (elem is not null)
    {
        if (elem.tart.Equals(ertek))
        {
            vane = true;
            elem.darabszam++;
        }

        elem = elem.kov;
    }

    if (vane == false)
    {
        if (this.fej is null || index == 0)
        {
            LancElem<T>? uj = new LancElem<T>(ertek, this.fej, 1);
            this.fej = uj;
        }
        else
        {
            LancElem<T>? p = this.fej;
            int i = 1;

            while (p.kov is not null && i < index)
            {
                p = p.kov;
                i++;
            }

            if (i <= index)
            {
                LancElem<T>? uj = new LancElem<T>(ertek, p.kov, 1);
                p.kov = uj;
            }
        }
    }
}

```

```

        }
        else throw new Exception("Hibás index!");
    }
}

public override void Torles(T ertek)
{
    LancElem<T>? p = this.fej;
    LancElem<T>? e = null;

    while (p is not null)
    {
        if (p.tart.Equals(ertek))
        {
            if (p.darabszam > 1)
            {
                p.darabszam--;
            }
            else
            {
                if (e is null)
                {
                    this.fej = p.kov;
                }
                else
                {
                    e.kov = p.kov;
                }
            }
            return;
        }
        e = p;
        p = p.kov;
    }
}

public void DarabSzamModositas(Func<T, int, int> modosito)
{
    LancElem<T>? p = this.fej;
    while (p is not null)
    {
        int modositott = modosito(p.tart, p.darabszam);
        p.darabszam = modositott;
        p = p.kov;
    }
}
}

```

```

namespace ALGA_ZH_B;

class Program
{
    static void Main(string[] args)
    {
        // FaHalmaz tesztelése
        FaHalmaz<int> fa = new FaHalmaz<int>();

        // Beszúrások
        fa.Beszur(10);
        fa.Beszur(5);
        fa.Beszur(15);

        // Elemszám kiírása
        Console.WriteLine("FaElemek száma: " + fa.Elemszam); // Kiírja: 3

        // Keresés
        Console.WriteLine("Van-e 5 a fában? " + fa.Eleme(5)); // Kiírja: True
        Console.WriteLine("Van-e 20 a fában? " + fa.Eleme(20)); // Kiírja: False

        // SzamoloFaHalmaz tesztelése
        SzamoloFaHalmaz<int> fa2 = new SzamoloFaHalmaz<int>();

        // Beszúrások
        fa2.Beszur(10);
        fa2.Beszur(5);
        fa2.Beszur(15);
        fa2.Beszur(10);

        // Elemszám kiírása (figyelembe véve a darabszámokat)
        Console.WriteLine("Számoló fa elemszáma (figyelembe véve a darabszámokat): " +
            fa2.Elemszam); // Kiírja: 4 (10 kétszer)

        // Szűrés minDarabSzam alapján (pl. darabszám >= 2)
        SzamoloFaHalmaz<int> szurtFa = fa2.Szures(2);
        Console.WriteLine("Szűrt elemszám (darabszám >= 2): " +
            szurtFa.Elemszam); // Kiírja: 2 (mert 10 van kétszer)

        // Szures minta
        Func<int, int, bool> feltetel = (ertek, darabszam) => darabszam < ertek;

        SzamoloFaHalmaz<int> ujHalmaz = fa2.Szures(feltetel);
        Console.WriteLine("Szűrt elemszám (darabszám < érték): " + ujHalmaz.Elemszam);
    }
}

```

```

namespace ALGA_ZH_B;

public class FaElem<T> where T : IComparable
{
    public T tart;
}
```

```

public FaElem<T>? bal;
public FaElem<T>? jobb;
public int darabszam;

public FaElem(T tart, FaElem<T>? bal, FaElem<T>? jobb)
{
    this.tart = tart;
    this.bal = bal;
    this.jobb = jobb;
}

public FaElem(T tart, FaElem<T>? bal, FaElem<T>? jobb, int darabszam)
{
    this.tart = tart;
    this.bal = bal;
    this.jobb = jobb;
    this.darabszam = darabszam;
}
}

```

namespace ALGA_ZH_B;

```

public class FaHalmaz<T> where T : IComparable
{
    private FaElem<T>? gyoker;

    public virtual int Elemszam { get; protected set; }

    public FaHalmaz()
    {
        this.gyoker = null;
        this.Elemszam = 0;
    }

    public virtual void Beszur(T ertek)
    {
        Elemszam++;
        gyoker = ReszFabaBeszur(gyoker, ertek);
    }

    private FaElem<T> ReszFabaBeszur(FaElem<T>? p, T ertek)
    {
        if (p == null)
        {
            FaElem<T> uj = new FaElem<T>(ertek, null, null);
            return uj;
        }
        else
        {
            if (p.tart.CompareTo(ertek) > 0)
            {
                p.bal = ReszFabaBeszur(p.bal, ertek);
            }
        }
    }
}

```

```

        else if (p.tart.CompareTo(ertek) < 0)
        {
            p.jobb = ReszFabaBeszur(p.jobb, ertek);
        }

        else if (p.tart.CompareTo(ertek) == 0)
        {
            throw new Exception("Van mar ilyen ertek!");
        }
    }
    return p;
}

public bool Eleme(T ertek)
{
    return ReszFaEleme(gyoker, ertek);
}
private bool ReszFaEleme(FaElem<T>? p, T ertek)
{
    if (p != null)
    {
        if (p.tart.CompareTo(ertek) > 0)
        {
            return ReszFaEleme(p.bal, ertek);
        }
        else
        {
            if (p.tart.CompareTo(ertek) < 0)
            {
                return ReszFaEleme(p.jobb, ertek);
            }
            else
            {
                return true;
            }
        }
    }
    else return false;
}
}

```

```

namespace ALGA_ZH_B;

public class SzamoloFaHalmaz<T> : FaHalmaz<T> where T : IComparable
{
    private FaElem<T>? gyoker;

    public override int Elemszam
    {
        get
        {
            int osszeg = 0;

```

```

Bejaras(gyoker, x =>
{
    if (x != null)
    {
        osszeg += x.darabszam;
    }
});

return osszeg;
}
}

public SzamoloFaHalmaz()
{
    this.gyoker = null;
    Elemszam = 0;
}

public override void Beszur(T ertek)
{
    gyoker = ReszFabaBeszur(gyoker, ertek);
}

private FaElem<T> ReszFabaBeszur(FaElem<T>? p, T ertek)
{
    if (p == null)
    {
        FaElem<T> uj = new FaElem<T>(ertek, null, null, 1);
        return uj;
    }
    else
    {
        if (p.tart.CompareTo(ertek) > 0)
        {
            p.bal = ReszFabaBeszur(p.bal, ertek);
        }

        else if (p.tart.CompareTo(ertek) < 0)
        {
            p.jobb = ReszFabaBeszur(p.jobb, ertek);
        }

        else if (p.tart.CompareTo(ertek) == 0)
        {
            p.darabszam++;
        }
    }
}

return p;
}

public void Bejaras(FaElem<T>? p, Action<FaElem<T>?> muvelet)
{
    if (p != null)

```

```
{  
    muvelet(p);  
    Bejaras(p.bal, muvelet);  
    Bejaras(p.jobb, muvelet);  
}  
}  
  
public SzamoloFaHalmaz<T> Szures(T minDarabSzam)  
{  
    SzamoloFaHalmaz<T> uj = new SzamoloFaHalmaz<T>();  
  
    Bejaras(gyoker, x =>  
    {  
        if (x.darabszam.CompareTo(minDarabSzam) >= 0)  
        {  
            uj.Beszur(x.tart);  
        }  
    });  
  
    return uj;  
}  
  
public SzamoloFaHalmaz<T> Szures(Func<T, int, bool> feltetel)  
{  
    SzamoloFaHalmaz<T> uj = new SzamoloFaHalmaz<T>();  
  
    Bejaras(gyoker, x =>  
    {  
        if (feltetel(x.tart, x.darabszam))  
        {  
            uj.Beszur(x.tart);  
        }  
    });  
  
    return uj;  
}  
}
```

```

namespace ALGA_ZH_C;

class Program
{
    static void Main(string[] args)
    {
        //Verem<int> peladahasznalat
        Verem<int> verem = new Verem<int>(5);

        // Értékek hozzáadása a veremhez
        verem.Verembe(10);
        verem.Verembe(20);
        verem.Verembe(30);

        Console.WriteLine("Veremben lévő elemek száma: " + verem.Elemszam); // Kiírja: 3

        // Kiolvassuk az elemeket a veremből
        int elem1 = verem.Verembol();
        int elem2 = verem.Verembol();

        Console.WriteLine("Első eltávolított elem: " + elem1); // Kiírja: 30
        Console.WriteLine("Második eltávolított elem: " + elem2); // Kiírja: 20

        Console.WriteLine("Veremben lévő elemek száma: " + verem.Elemszam); // Kiírja: 1

        //Íhelyettesithető megvalósító osztaly hasznalata
        // DinamikusVerem létrehozása Szam típusú objektumokkal
        DinamikusVerem<Szam> dinamikusVerem = new DinamikusVerem<Szam>(5);

        // Elemszámok hozzáadása a veremhez
        dinamikusVerem.Verembe(new Szam(4)); // páros, helyettesítendő
        dinamikusVerem.Verembe(new Szam(7)); // páratlan
        dinamikusVerem.Verembe(new Szam(10)); // páros, helyettesítendő
        dinamikusVerem.Verembe(new Szam(15)); // páratlan
        dinamikusVerem.Verembe(new Szam(8)); // páros, helyettesítendő

        //Paros szamok megszamlalása
        bool ParosE(Szam szam)
        {
            if (szam.Erték % 2 == 0) return true;
            return false;
        }

        int parosSzamok = dinamikusVerem.Megszamol(ParosE);
        Console.WriteLine($"A veremben lévő páros számok száma: {parosSzamok}");

        // Elemek kiszedése a veremből helyettesítéssel
        try
        {
            Console.WriteLine("Kivett elem: " + dinamikusVerem.Verembol()); // Nem helyettesítendő elem
            Console.WriteLine("Kivett elem: " + dinamikusVerem.Verembol()); // Helyettesíthető elem
            Console.WriteLine("Kivett elem: " + dinamikusVerem.VerembolCserevel()); // Csere a
helyettesítővel
        }
        catch (Exception ex)

```

```
        {
            Console.WriteLine("Hiba: " + ex.Message);
        }
    }
```

```
namespace ALGA_ZH_C;

public class Szam : IHelyettesitheto<Szam>
{
    public int Ertek { get; set; }

    public Szam(int ertek)
    {
        Ertek = ertek;
    }

    public bool Helyettesitendo => Ertek % 2 == 0;

    public bool Helyettesitheti(Szam masik)
    {
        return masik.Ertek > this.Ertek;
    }

    public override string ToString()
    {
        return Ertek.ToString();
    }
}
```

```
namespace ALGA_ZH_C;

public class Verem<T>
{
    private T[] E;
    private int n;

    public int Elemszam => n;

    public Verem(int meret)
    {
        this.E = new T[meret];
        n = 0;
    }

    public virtual void Verembe(T ertek)
    {
        if (n < E.Length - 1)
        {
```

```

        E[n++] = ertek;
    }
    else throw new Exception("Nincs hely");
}

public virtual T Verembol()
{
    if (n > 0)
    {
        T ertek = E[n-1];
        n--;
        return ertek;
    }
    else throw new Exception("Nincs elem");
}
}

```

namespace ALGA_ZH_C;

```

public class DinamikusVerem<T> : Verem<T> where T : IHelyettesitheto<T>
{
    private T[] E;
    private int n;

    public DinamikusVerem(int meret) : base(meret)
    {
        this.E = new T[meret];
        n = 0;
    }

    public override void Verembe(T ertek)
    {
        if (n == E.Length)
        {
            T[] uj = new T[E.Length * 2];
            for (int i = 0; i < E.Length; i++)
            {
                uj[i] = E[i];
            }

            E = uj;
        }
        E[n++] = ertek;
    }
}

```

```

public int Megszamol(Func<T, bool> feltetel)
{
    int szamlalo = 0;
    for (int i = 0; i < n; i++)
    {
        if (feltetel(E[i]))
    }
}

```

```

        {
            szamlalo++;
        }
    }
    return szamlalo;
}

public override T Verembol()
{
    if (n > 0)
    {
        T x = E[--n];

        if (!x.Helyettesitendo)
        {
            return x;
        }
        else
        {
            for (int i = n - 1; i >= 0; i--)
            {
                if (E[i].Helyettesitheti(x))
                {
                    T helyettesito = E[i];

                    for (int j = i; j < n - 1; j++)
                    {
                        E[j] = E[j + 1];
                    }
                    n--;
                }
            }
            return helyettesito;
        }
    }
    throw new Exception("Nincs helyettesíthető elem");
}

public T VerembolCserevel()
{
    if (n > 0)
    {
        T x = E[--n];

        if (!x.Helyettesitendo)
        {
            return x;
        }
        else
        {
            for (int i = n - 1; i >= 0; i--)
            {

```

```
if (E[i].Helyettesitheti(x))
{
    T helyettesito = E[i];
    E[n - 1] = helyettesito;

    for (int j = i; j < n - 1; j++)
    {
        E[j] = E[j + 1];
    }
    n--;
}

return helyettesito;
}
}
throw new Exception("Nincs helyettesíthető elem");
}
}
```

```
namespace ALGA_ZH_C;
```

```
public interface IHelyettesitheto<T>
{
    public bool Helyettesitendo { get; }
    public bool Helyettesitheti (T masik);
}
```

```

namespace ALGA_ZH_D;

class Program
{
    static void Main(string[] args)
    {
        // FaHalmaz<int> példányosítása
        FaHalmaz<int> fa = new FaHalmaz<int>();

        // Elembeszúrás
        fa.Beszur(5);
        fa.Beszur(3);
        fa.Beszur(7);
        fa.Beszur(2);
        fa.Beszur(4);
        fa.Beszur(6);
        fa.Beszur(8);

        // Elemek keresése
        Console.WriteLine("A fa tartalmazza a 4-es elemet? " + fa.Eleme(4)); // true
        Console.WriteLine("A fa tartalmazza a 10-es elemet? " + fa.Eleme(10)); // false

        // Új elem beszúrása és ellenőrzés
        fa.Beszur(10);
        Console.WriteLine("10-es elem beszúrása után: " + fa.Eleme(10)); // true

        // Függőség szűrési feltétel: Csak pozitív számok elfogadhatóak
        Func<SzamFuggo, bool> elfogadhato = x => x.Szam > 0;

        //.....
        
        // FuggoFaHalmaz példány létrehozása
        FuggoFaHalmaz<SzamFuggo> fa2 = new FuggoFaHalmaz<SzamFuggo>(elfogadhato);

        // Elembeszúrás (független elemek)
        fa2.Beszur(new SzamFuggo(-55));
        fa2.Beszur(new SzamFuggo(3));
        fa2.Beszur(new SzamFuggo(7));

        // Függő elemek beszúrása
        SzamFuggo fugg1 = new SzamFuggo(-2, new SzamFuggo(-1));
        SzamFuggo fugg2 = new SzamFuggo(4, fugg1);
        fa2.Beszur(fugg1);
        fa2.Beszur(fugg2);

        FaHalmaz<SzamFuggo> problemak = fa2.ProblemasElemek();

        // Problémás elemek kiíratása
        Console.WriteLine("Problémás elemek a fában:");
        problemak.PreOrderBejaras(x => Console.WriteLine(x));
    }
}

```

```
namespace ALGA_ZH_D;

public class FaElem<T> where T : IComparable<T>
{
    public T tart;
    public FaElem<T>? bal;
    public FaElem<T>? jobb;

    public FaElem(T tart, FaElem<T>? bal, FaElem<T>? jobb)
    {
        this.tart = tart;
        this.bal = bal;
        this.jobb = jobb;
    }
}
```

```
namespace ALGA_ZH_D;

public class FaHalmaz<T> where T : IComparable<T>
{
    protected FaElem<T>? gyoker;

    public FaHalmaz()
    {
        gyoker = null;
    }

    public void Beszur(T ertek)
    {
        gyoker = ReszfabaBeszur(gyoker, ertek);
    }

    private FaElem<T> ReszfabaBeszur(FaElem<T>? p, T ertek)
    {
        if (p is null)
        {
            FaElem<T> uj = new FaElem<T>(ertek, null, null);
            return uj;
        }
        else if (p.tart.CompareTo(ertek) > 0)
        {
            p.bal = ReszfabaBeszur(p.bal, ertek);
        }
        else if (p.tart.CompareTo(ertek) < 0)
        {
            p.jobb = ReszfabaBeszur(p.jobb, ertek);
        }
        return p;
    }

    public bool Eleme(T ertek)
    {
        return ReszfaEleme(gyoker, ertek);
    }
}
```

```

}

private bool ReszfaEleme(FaElem<T>? p, T ertek)
{
    if (p is not null)
    {
        if (p.tart.CompareTo(ertek) > 0)
        {
            return ReszfaEleme(p.bal, ertek);
        }
        else if (p.tart.CompareTo(ertek) < 0)
        {
            return ReszfaEleme(p.jobb, ertek);
        }
        else return true;
    }
    else return false;
}

public void PreOrderBejaras(Action<T> muvelet)
{
    ReszfaPreOrderBejaras(gyoker, muvelet);
}

private void ReszfaPreOrderBejaras(FaElem<T>? p, Action<T> muvelet)
{
    if (p is not null)
    {
        muvelet(p.tart);
        ReszfaPreOrderBejaras(p.bal, muvelet);
        ReszfaPreOrderBejaras(p.jobb, muvelet);
    }
}
}
}

```

```

namespace ALGA_ZH_D;

public class SzamFuggo : IComparable<SzamFuggo>, IFuggo<SzamFuggo>
{
    public int Szam { get; set; }
    public SzamFuggo? ToleFugg { get; set; }

    public bool FuggValakitol => ToleFugg != null;

    public SzamFuggo(int szam, SzamFuggo? toleFugg = null)
    {
        this.Szam = szam;
        this.ToleFugg = toleFugg;
    }

    public int CompareTo(SzamFuggo? other)
    {
        if (other == null) return 1;
        return this.Szam.CompareTo(other.Szam);
    }
}
```

```

}

public override string ToString()
{
    return $"Szám: {Szam}, Függ: {(ToleFugg != null ? ToleFugg.Szam.ToString() : "null")}";
}
}

namespace ALGA_ZH_D;

public class FuggFaHalmaz<T> : FaHalmaz<T> where T : IComparable<T>, IFuggo<T>
{
    private Func<T, bool> elfogadhato;

    public FuggFaHalmaz(Func<T, bool> elfogadhato)
    {
        this.elfogadhato = elfogadhato;
    }

    public FaHalmaz<T> ProblemasElemek()
    {
        FaHalmaz<T> uj = new FaHalmaz<T>();
        PreOrderBejaras(x =>
        {
            if (x.FuggValakitol && x.ToleFugg.Equals(default(T)) || !elfogadhato(x))
            {
                uj.Beszur(x);
            }
        });
        return uj;
    }

    public void ProblemasElemekTorlese()
    {
        ReszfaProblemasElemekTorlese(gyoker);
    }

    private FaElem<T>? ReszfaProblemasElemekTorlese(FaElem<T>? p)
    {
        if (p is null) return null;

        p.bal = ReszfaProblemasElemekTorlese(p.bal);
        p.jobb = ReszfaProblemasElemekTorlese(p.jobb);

        if (p.tart.FuggValakitol && p.tart.ToleFugg.Equals(default(T)) && !elfogadhato(p.tart))
        {
            if (p.bal is null && p.jobb is null)
            {
                return null;
            }
        }
    }
}

```

```

else if (p.bal is null)
{
    return p.jobb;
}
else if (p.jobb == null)
{
    return p.bal;
}
else
{
    p.bal = KetgyerekTorles(p, p.bal);
}
}
return p;
}

private FaElem<T>? KetgyerekTorles(FaElem<T> e, FaElem<T> r)
{
    if (r.jobb is not null)
    {
        r.jobb = KetgyerekTorles(e, r.jobb);
        return r;
    }
    else
    {
        e.tart = r.tart;
        return r.bal;
    }
}
}

```

namespace ALGA_ZH_D;

```

public interface IFuggo<T>
{
    public bool FuggValakitol { get; }
    public T ToleFugg { get; }
}

```