Aszalós László

Algoritmusok

mobiDIÁK könyvtár

Aszalós László

Algoritmusok

mobiDIÁK könyvtár

SOROZATSZERKESZTŐ

Fazekas István

Aszalós László

Algoritmusok

Szakmai segédanyag
Műszaki informatikusok részére
első kiadás

mobiDIÁK könyvtár
Debreceni Egyetem
Informatikai Kar

Lektor

Dr. Juhász István
Debreceni Egyetem
Informatikai Kar

mobiDIÁK könyvtár
Debreceni Egyetem
Informatikai Kar
4010 Debrecen, Pf. 12
http://mobidiak.inf.unideb.hu

A mű *A mobiDIÁK önszervező mobil portál* (IKTA, OMFB-00373/2003) és a *GNU Iterátor, a legújabb generációs portál szoftver* (ITEM, 50/2003) projektek keretében készült.

Tartalomjegyzék

I. fejezet
Előszó

Ez a jegyzet elsőéves műszaki informatikusok számára tartott Algoritmusok előadás anyagát tartalmazza. Ennek megfelelően a jegyzet nem feltételez fel korábbi informatikai ismereteket. A jegyzetnek nem célja a programozás oktatása, azt a következő féléves gyakorlat és más tantárgyak vállalják fel.
A jegyzetnek is címet adó algoritmusok általános leírásával, az algoritmusok mérésére szolgáló fogalmak megismerésével kezdünk. Ezután röviden áttekintjük az algoritmuselmélet főbb fogalmait és problémáit, hogy az algoritmusok jellemzésére használt idő- és tárbonyolultság fogalmához eljussunk. Ezt már a konkrét algoritmusok követik. Az algoritmusok leírására egy pszeudókódot használunk, amelyet a XII. fejezet ismertet. Próbáltunk minél több példával, ábrával kiegészíteni az algoritmusokat, hogy az algoritmusok lépései könnyen érthetőek legyenek. Ugyanezen célból készültek el azok a HTML oldalak, melyek az egyes algoritmusokat mutatják be véletlenszerűen generált adatokra, futás közben.
Az tematikában szereplő témakörök nagy számára, az idő rövidségére és a hallgatók felsőbb matematikai ismereteinek hiányára tekintettel a bonyolultságok matematikai bizonyításával nem foglalkozunk, ezeket az érdeklődő hallgatók megtalálhatják a lentebb említett könyvekben. Hasonlóan a bonyolultabb, csak hosszabban megfogalmazható algoritmusok kódjait sem szerepeltetjük a jegyzetben, csupán a leírással, példákon keresztül mutatjuk be azokat.
A jegyzet erősen épít T.H. Cormen, C.E. Leirson és R.L. Rivest Algoritmusok című könyvére, és Ivanyos G., Rónyai L. és Szabó R. Algoritmusok című jegyzetére, de bizonyos algortimusoknál tudatosan eltértünk az ott leírtaktól.

II. fejezet
Algoritmusokról általában

1. Euklidészi algoritmus

Az algoritmus szóról sokaknak elsőre az euklideszi algoritmus jut az eszébe, ezért kezdjünk ezzel!
Euklidészi algoritmus:

> Adott két pozitív egész szám: m és n. Keresendő legnagyobb közös osztójuk, vagyis az a legnagyobb pozitív egész, amelyik mindkettőnek az osztója.

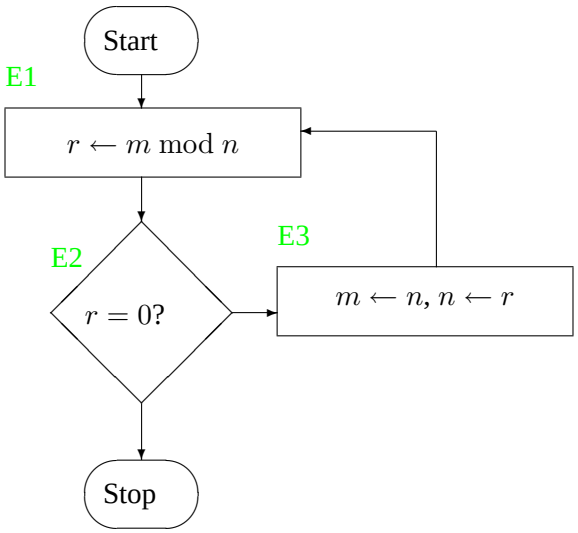
Az algoritmus a következő lépésekkel írható le:

E1. Osszuk el m-et n-nel és a maradék legyen r.
E2. Ha r=0, akkor az algoritmus véget ért, az eredmény n.
E3. Legyen $m \leftarrow n$ és $n \leftarrow r$, és folytassuk az algoritmust az E1 lépéssel.

Kövessük az algoritmus futását a 119 és 544 számokra, azaz $m = 119$ és $n = 544$!

Lépés	Értékadások	
	$m \leftarrow 119$	$n \leftarrow 544$
E1.	$r \leftarrow 119$	
E3.	$m \leftarrow 544$	$n \leftarrow 119$
E1.	$r \leftarrow 68$	
E3.	$m \leftarrow 119$	$n \leftarrow 68$
E1.	$r \leftarrow 51$	
E3.	$m \leftarrow 68$	$n \leftarrow 51$
E1.	$r \leftarrow 17$	
E3.	$m \leftarrow 51$	$n \leftarrow 17$
E1.	$r \leftarrow 0$	

Az n utolsó értéke 17 volt, ennek megfelelően a 119 és az 544 számok legnagyobb közös osztója 17. Az euklidészi algoritmust természetesen nem csak lépések listájaként, hanem folyamatábrával is megadhatjuk.



Az euklidesz.htm fájl tartalmazza azt a programot, amely a felhasználó által megadott két pozitív számra kiszámolja a legnagyobb közös osztójukat.

2. Az algoritmus tulajdonságai

Az előbb láthattunk egy algoritmust, de mi alapján dönthetjük el egy utasítássorozatról, hogy az algoritmust alkot, vagy sem? Az alábbiakban azokat a követelményeket, jellemzőket soroljuk fel, melyek az algoritmusok sajátosságai.

Végesség: Az algoritmus futása véges sok lépés után befejeződik. Esetünkben r kisebb mint n, azaz n értéke folyamatosan csökken az algoritmus végrehajtása során, és pozitív egészek egyre fogyó sorozata egyszer véget ér.

Meghatározottság: Az algoritmus minden egyes lépésének pontosan definiáltnak kell lennie. Miután az élőnyelvi megfogalmazás esetenként nem egyértelmű, használhatunk különféle programnyelveket, ahol mindennek pontos, egyértelműen definiált jelentése van. A következő fejezetben ismertetjük a Turing-gépet, melyet akár használhatnánk is az algoritmusok leírására, ám az ilyen nyelven írt programok hosszúak, és nehezen érthetőek lennének. Ezért a későbbiekben az algoritmusokat pszeudokódban adjuk meg. Ez a kód igen közel áll a Pascal programnyelvű kódokhoz, ám a változók deklarálásától, és minden olyan hasonló szerkezettől, melyek az algoritmus megértését nem befolyásolják, eltekintünk.

Bemenet/Input: Az algoritmus vagy igényel vagy sem olyan adatokat, amelyeket a végrehajtása előtt meg kell adni. Az input mindig bizonyos meghatározott halmazból kerülhet ki, esetünkben m és n pozitív egész szám.

Kimenet/Output: Az algoritmushoz egy vagy több output tartozhat, amelyek meghatározott kapcsolatban állnak az inputtal. Esetünkben az output az n utolsó értéke lesz, ami az input értékeknek legnagyobb közös osztója.

Elvégezhetőség: Elvárjuk, hogy az algoritmust végre lehessen hajtani, azaz a végrehajtható utasítások elég egyszerűek ahhoz, hogy egy ember papírral és ceruzával véges idő alatt pontosan végrehajthassa. Például a végtelen tizedestörtek osztása nem ilyen, mert ez végtelen lépéssorozat.

Univerzális: Az algorimusnak működnie kell tetszőleges (a feltételeknek megfelelő) bemeneti értékek esetén. Az euklidészi algoritmusunk természetesen tetszőleges pozitív számpárnak meghatározza a legnagyobb közös osztóját.

Ezek alapján tekinthetjük az algoritmust egy számolási probléma megoldásának. A probléma megfogalmazása általánosságban meghatározza a kívánt bemenet/kimenet kapcsolatot. Az algoritmus egy specifikus számolási eljárást ír le ennek a kapcsolatnak eléréséhez. A legnagyobb közös osztó problémájának egy esete a 119, 544 számpár. Egy *eset* az összes olyan bementő adatból áll, amelyek szükségesek a probléma megoldásának számításához. Egy algoritmust *helyesnek* nevezünk, ha minden konkrét bemenetre helyes kimenetet ad és megáll.

3. Jó és jobb algoritmusok

Ugyanaz a számítási probléma több különböző algoritmussal is megoldható. Például az elkövetkező órákon több rendezési algoritmust is megvizsgálunk. Hogyan lehet az algoritmusok közül választani?

Kisérletek: az egyes algoritmusok implementációt különböző adatokon tesztelünk, s a futási eredmények alapján döntünk.

Elméleti vizsgálat: matematikai módszerekkel meghatározzuk az adott algoritmus számára szükséges erőforrásokat (a végrehajtási időt, vagy az elfoglalt tárterületet), mint a bemenő adatok függvényét.

Elméleti vizsgálódáshoz mind az inputot, mind a végrehajtási időt számszerűsíteni kell.

Bemenet mérete: függ a probléma típusától: lehet az adatok száma (rendezés); lehet az adat mérete (például bit a prímtesztnél). Gyakran több módon is mérhetjük a bemenetet, például tekinthetjük a gráf méretének a gráf csúcsainak vagy az éleinek számát.

Futási idő: Lehetőség szerint gépfüggetlen jelölést szeretnénk használni. Ilyen lehet például a végrehajtott lépések (elemi utasítások) száma. A leggyakrabban vizsgált mennyiségek: *legjobb érték*, *legrosszabb érték*, *átlagos érték*, azaz minimálisan, maximálisan és átlagosan hány lépést kell végrehajtani az n méretú inputok esetén. A gyakorlatban talán az átlagos érték lenne a legjobban használható, de ezt gyakran nagyon nehéz pontosan meghatározni. A legrosszabb értéket rendszerint könnyebben

meghatározhatjuk, s ez az érték egy pontos felső korlátot ad minden egyes futásra. Bizonyos algoritmusoknál ez az eset igen gyakran előfordul, tehát ekkor közel áll az átlagos értékhez.

4. Kis ordó és nagy ordó

Milyen kapcsolatban áll a futási idő az input méretével, amikor ez a méret a végtelenbe tart? Tudjuk-e a futási idő függvényét valamilyen egyszerűbb függvénnyel becsülni, felülről korlátozni?
Az $O(f(n))$ jelölést (*nagy ordó*) rendszerint pozitív egész n-eken értelmezett f függvény esetén használjuk. Nem valami határozott mennyiséget jelöl, Az $O(f(n))$ jelölés az n-től függő mennyiségek becslésére szolgál. Egy X mennyiség helyére akkor írhatunk $O(f(n))$-t, ha létezik olyan konstans, mondjuk M, hogy minden elég nagy n-re,$|X| \leq M \cdot |f(n)|$, azaz aszimptotikusan felső becslést adunk egy konstansszorzótól eltekintve a lépésszámra. A definíció nem adja meg az M konstans értékét, és hogy honnan számít nagynak az n. Ezek különböző esetekben más és más értékek lehetnek.
Példa. Mutassuk meg, hogy $\frac{n^2}{2} - 3n = O(n^2)$!

A definíció szerint $|\frac{n^2}{2} - 3n| \leq M|n^2|$. Ha $n \geq 6$, elhagyhatjuk az abszolútértékeket. Kis átalakítások után a $3n \geq (\frac{1}{2} - M)n^2$ egyenlőtlenséget kapjuk. Ha $M \geq \frac{1}{2}$, akkor a feltétel teljesül, tehát $n > 6$ esetén létezik a feltételeket kielégítő M konstans.
Példa. Mutassuk meg, hogy nem teljesül a $6n^3 = O(n^2)$!

Pozitív n esetén a $6n^3 \leq Mn^2$ egyenlőtlenséget kellene belátnunk. Átalakítás után ebből $n \leq \frac{M}{6}$ származtatható, tehát adott M érték esetén n értéke nem lehet tetszőlegesen nagy, ellentétben az eredeti feltételekkel.

A gyakorlatban előforduló feladatok bonyolultsága rendszerint az alábbi osztályok valamelyikébe esik. A bonyolultsági osztályok hagyományos jelölését a szokásos elnevezés követi, majd a listát pár adott bonyolultságú feladat zárja.

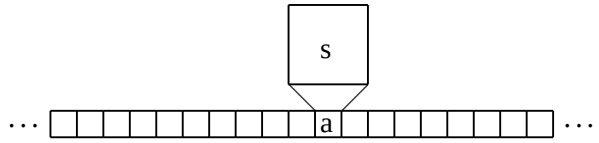
- $O(1)$**:** konstans idő: egyszerű értékadás, írás/olvasás veremből
- $O(\ln(n))$**:** logaritmikus: bináris keresés rendezett tömbben, elem beszúrása, törlése bináris fából, kupacból (heap)
- $O(n)$**:** lineáris: lista/tömb végigolvasása, lista/tömb minimális/maximális elemének, elemek átlagának meghatározása, $n!$, `fib(n)` kiszámítása
- $O(n \cdot \ln(n))$**:** quicksort, összefésüléses rendezés.
- $O(n^2)$**:** négyzetes: egyszerű rendezési algoritmusok (pl. buborékrendezés), nem rendezett listában az ismétlődések megtalálása
- $O(c^n)$**:** exponenciális: hanoi torony, rekurzív Fibonacci számok, n elem permutációinak előállítása

A nagy ordó definíciójában elegendő volt egy adott konstanst találni, melyre az összefüggés igaz. Ha ezt az összefüggést minden pozitív konstansra megkövetelnénk, akkor a *kis ordó* definícióját kapjuk. $2n^2 = O(n^2)$ és $2n = O(n^2)$ egyaránt teljesül, viszont $2n^2 = o(n^2)$ nem teljesül, míg $2n = o(n^2)$ igen. A kis

ordó jelölés éles aszimptotikus felső becslést jelöl, s ha $f(n) = o(g(n))$, akkor a $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ összefüggés is fennáll.

III. fejezet
Turing-gép

Az algoritmus fogalmának pontos megfogalmazásával a múlt század első felében többen is próbálkoztak. Ennek eredményeképpen több, egymástól különböző fogalom is megszületett, mint például a lambda-függvények, rekurzív függvények, Markov-gép és a Turing-gép. Ezek a fogalmak egyenrangúak egymással, egymásból kifejezhetők. A gyakorlatban leginkább a Turing-gép fogalma terjedt el. A Turing-gép több, egymással ekvivalens definíciójával is találkozhatunk különböző szakkönyvekben. Abban mindegyik definíció megegyezik, hogy a Turing-gépnek van egy központi vezérlőegysége, és egy vagy több szalag-egysége.



1. A Turing-gép felépítése

A szalagok négyzetekre, vagy más elnevezéssel mezőkre vannak osztva. Minden egyes mező maximum egy betűt vagy írásjelet tartalmazhat. Ezek a jelek egy előre rögzített, véges halmazból származnak. Ebben a halmazban van egy üres jelnek nevezett elem. A Turing-gép indulása előtt minden szalag minden egyes mezője —véges sok mezőtől eltekintve— ezt a jelet tartalmazza. Feltesszük, hogy minden egyes szalag (legalább) az egyik irányban végtelen. Minden egyes szalagegységhez tartozik egy-egy olvasó-író fej, amely minden egyes lépésben elolvassa a fej alatt álló mező tartalmát, azt törli, és egy jelet ír vissza (esetleg ugyanazt). Azt, hogy mit ír az adott mezőbe a fej, az olvasott jel és a vezérlő belső állapota határozza meg. Ugyanezek alapján a vezérlő új állapotba kerül és a fejet (más megfogalmazásban a szalagot) egy mezővel balra, jobbra mozgatja, vagy éppen helyben hagyja.
A vezérlő (legalább az) egyik állapota végállapot, s ha a gép ebbe kerül, akkor megáll. Az egyik kijelölt szalag üres jelektől különböző részét tekintjük a gép kimenetének (output). A vezérlő egy kijelölt állapotát kezdőállapotnak nevezzük, s a Turing-gép indulásakor a gép ebben az állapotban van.
Az egyszalagos Turing-gépet a (S, A, M, s_0, F) ötössel írhatjuk le, ahol az S a vezérlő állapotainak halmaza, az A a mezőkre írható jelek halmaza, az s_0 a kiinduló állapot, az F a végállapotok halmaza az M olyan $satbl$ ötösök halmaza, ahol s és t a vezérlő állapotai, a és b egy-egy karakter (betű vagy írásjel), l pedig a L,R,S betűk valamelyike, melyek rendre a balra, jobbra mozgást, illetve helybenmaradást

jelzik. (Az n-szalagos Turing-gép esetén az M $2+3n$-esek halmaza lesz, minden szalag esetén külön-külön meg kell adni, hogy mi kerül az adott szalagra, és a szalag merre mozdul.) Ha az (S, A, M, s_0, F) Turing-gépben az M ötöseiben a harmadik, negyedik és ötödik értéket az első kettő egyértelműen meghatározza, azaz a harmadik, negyedik és ötödik érték az első kettőnek függvénye, akkor determinisztikus Turing-gépről beszélünk, ellenkező esetben nemdeterminisztikus Turing-gépről.

Példa. A hárommal osztható hosszúságú egyesekből álló szavakat elfogadó egyszalagos, determinisztikus Turing-gép a következő:

$$\begin{aligned} S &= \{q_0, q_1, q_2, q_v\} \\ A &= \{0, 1\} \\ s_0 &= q_0 \\ F &= \{q_v\} \\ M &= \{q_0 1 q_1 1 R,\ q_1 1 q_2 1 R,\ q_2 1 q_0 1 R, \\ &\quad q_0 0 q_v 0 S,\ q_1 0 q_1 0 S,\ q_2 0 q_2 0 S\} \end{aligned}$$

Vannak akik jobban szeretik a Turing gép következő ábrázolását:

	0	1
q_0	$q_v 0 S$	$q_1 1 R$
q_1	$q_1 0 S$	$q_2 1 R$
q_2	$q_2 0 S$	$q_0 1 R$
$\underline{q_v}$		

Lássuk e Turing-gép futását pár bemenetre! Az egyszerűbb jelölés kedvéért csupán a szalag aktuális tartalmát írjuk le, s a fejet a soron következő karakter előtt található állapot fogja jelölni. Ennek megfelelően az `11` input esetén a következő a gép kezdeti konfigurációja: `...0`q_0`110....`

0. `...0`q_0`110...`
1. `...01`q_1`10...`
2. `...011`q_2`0...`

$\vdots$

i. `...011`q_2`0...`

$\vdots$

Mint a táblázatból lehet látni, az `11` input esetén a második lépestől kezdődően a Turing-gép nem vált állapotot, s így végtelen ciklusba került.

0. `...0`q_0`1110...`
1. `...01`q_1`110...`
2. `...011`q_2`10...`
3. `...0111`q_0`0...`
4. `...0111`q_v`0...`

Az `111` input esetén a negyedik lépésben a Turing-gép végállapotba kerül, s így megáll. Szokás ilyenkor azt mondani, hogy az adott Turing-gép elfogadta ezt az inputot. A Turing-gépeket felhasználhatjuk függvények kiszámolására, azaz az argumentumokat a bemeneti szalagra írva, a gépet elindítva a Turing-gép a kimeneti szalagra a függvény végeredményét írva megáll. Felhasználhatjuk a Turing-gépeket

nyelvek felismerésére is. *Szavaknak* nevezzük az A ábécé betűiből alkotott véges sorozatokat. *Nyelvnek* nevezzük szavak egy halmazát. A T Turing-gép által felismert L_T nyelv pontosan azokból a szavakból áll, melyekkel mint bemenettel indítva a Turing-gép megáll.
Egy L nyelvet *rekurzívan felsorolhatónak* nevezünk, ha van olyan Turing-gép amely által felismert nyelv éppen az L. Egy L nyelv *rekurzív*, ha létezik olyan Turing-gép, mely tetszőleges inputra megáll, és a szóhoz tartozó végállapot pontosan akkor egyezik meg az egyik előre kijelölt állapottal, ha a szó L-beli. Az f függvény *parciálisan rekurzív*, ha létezik olyan Turing-gép, amely kiszámolja. Az f függvény *rekurzív*, ha létezik olyan Turing-gép, amely kiszámolja; és az output minden bemenetre definiálva van.

2. Church-Turing tézis.

Ami algoritmussal kiszámítható, az Turing-értelemben kiszámítható:

- Az f parciális függvény akkor és csak akkor kiszámítható, ha f parciálisan rekurzív.
- Az f függvény akkor és csak akkor kiszámítható, ha f rekurzív.
- Az L nyelvhez tartozás problémája algoritmussal csak akkor és akkor eldönthető, ha L rekurzív.

Ezekben az állításokban két fajta kiszámíthatóságról esik szó. A Turing-értelemben kiszámíthatóság jól definiált fogalom, míg az algoritmussal kiszámíthatóság nem az. Ennek megfelelően ez a tétel nem bizonyítható, viszont a gyakorlati tapasztalatokkal egybevág.
Állítás. Van olyan nyelv, mely nem rekurzív felsorolható.

Bizonyítás: A Turing-gép végesen leírható, ennek megfelelően maximum megszámlálhatóan sok létezik, ezek pedig felsorolhatóak. Minden géphez egyértelműen tartozik egy rekurzív felsorolható nyelv, így ezek is felsorolhatóak. Konstruáljunk egy olyan nyelvet, amely mindegyiktől különbözik. A véges szavak is felsorolhatóak, így definiáljuk az új nyelvet úgy, hogy ha az első nyelvben szerepel — ebben a felsorolásban— az első szó, akkor az új nyelvben ne szerepeljen, s viszont. Hasonlóan a másodikra, és így tovább. Az új nyelv mindegyik korábbi nyelvtől különbözik, így nem rekurzív felsorolható.

	w_1	w_2	w_3	w_4	w_5	w_6	
L_1	**X**	X		X		X	
L_2	X			X		X	
L_3		X		X		X	$\cdots$
L_4	X	X	X			X	
L_5	X		X	X	**X**		
				$\vdots$			
L'		**X**	**X**	**X**		**X**	

3. Univerzális Turing-gép

Egy megfelelő nyelvet használva tetszőleges T Turing-gép leírható egy w_T karaktersorozattal, s létezik egy olyan U Turing gép, hogy az U pontosan akkor fogadja el a w_T, s inputot, amikor a T elfogadja az s inputot; feltéve, hogy a w_T egy Turing-gép kódja. Miután az U Turing-gép képes szimulálni minden Turing-gépet, ezért Univerzális Turing-gépnek nevezzük. (A konkrét konstrukció több szakkönyben is megtalálható, mi most nem részletezzük.)

Tétel. Létezik felsorolható, de nem rekurzív nyelv.

Bizonyítás: Tekintsük azokat a Turing-gépeket, melyek nem fogadják el a saját kódjukat inputként. Ezen Turing-gépek kódjai meghatároznak egy nyelvet. Erről a nyelvről belátható, hogy rekurzív felsorolható, ám ezzel mi nem foglalkozunk. Ha ez a nyelv még rekurzív is volna, akkor lenne egy Turing gép, amely pontosan ezt a nyelvet ismerné fel. Azaz azokat a kódokat ismerné fel, amelyhez tartozó Turing-gépek nem ismerik fel magukat. Felismeri-e ez a gép saját magát? Ha nem, akkor kódja benne van a nyelvben, de akkor a definíció miatt fel kellene ismernie saját magát. Ha pedig felismeri, akkor olyan a kódja, hogy nem ismerheti fel magát. Mindkét esetben ellentmondáshoz jutottunk, így ez a nyelv nem lehet rekurzív.

Eldöntési probléma. Tetszőleges Turing-gép kódjára és tetszőleges inputra el tudja-e dönteni az univerzális gép, hogy a szimulált gép megáll-e az adott inputra, vagy sem?

Miután felsorolhatóak azok a Turing-gép kódokból és megfelelő inputból álló párok, melyekre a szimulált gép megáll, ezen párok nyelvéhez létezik azt felismerő Turing-gép, s a párok nyelve rekurzív felsorolható. Ha ez a nyelv még rekurzív is lenne, a megfelelő Turing gép eldöntené az önmagukat fel nem ismerő gépek problémáját, felismerné az előbbi nyelvet is, de az meg kizárt.

4. Idő- és tárkorlát

A T Turing-gép $t(n)$ időkorlátos, ha n hosszú inputon legfeljebb $t(n)$ lépést tesz. TIME$(t(n))$ azon nyelvek halmaza, melyek felismerhetőek egy $O(t(n))$ időkorlátos Turing-géppel. A T Turing-gép $s(n)$ tárkorlátos, ha n hosszú inputon legfeljebb $s(n)$ mezőt használ a munkaszalagokon. SPACE$(s(n))$ azon nyelvek halmaza, melyek felismerhetőek egy $O(s(n))$ tárkorlátos Turing-géppel. Ezek alapján definiálhatjuk a következő halmazokat:

$$\text{P} = \bigcup_{k=0}^{\infty} \text{TIME}(n^k)$$

$$\text{PSPACE} = \bigcup_{k=0}^{\infty} \text{SPACE}(n^k)$$

$$\text{EXPTIMES} = \bigcup_{k=0}^{\infty} \text{TIME}(2^{n^k})$$

5. NP nyelvosztály

A determinisztikus Turing-gép akkor fogad el egy inputot, ha azzal indítva leáll. A nemdeterminisztikus Turing-gép ugyanannál az inputnál más és más lépéssorozatokat hajthat végre, egyes esetekben megáll, máskor pedig nem. Ha van olyan számítási eljárás, melyben a Turing-gép megáll, akkor azt a nemdeterminisztikus Turing-gép elfogadja az inputot.
A T nemdeterminisztikus Turing-gép $t(n)$ időkorlátos, ha n hosszú inputon minden számítási út mentén legfeljebb $t(n)$ lépést téve megáll. NTIME$(t(n))$ azon nyelvek halmaza, melyek felismerhetőek egy $O(t(n))$ időkorlátos nemdeterminisztikus Turing-géppel.

$$\text{NP} = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k)$$

Tétel. Egy L nyelv pontosan akkor tartozik az NP nyelvosztályba, ha létezik egy olyan párosokból álló L' P-beli nyelv, hogy x eleme L-nek akkor és csak akkor, ha (x, y) eleme L'-nek valamely y-ra. Az y-t gyakran x tanújának szokás nevezni.

6. Nevezetes NP bonyolultságú problémák

3 színnel színezhető gráfok: Adott egy gráf, s döntsük el, hogy kiszínezhetők-e a gráf csúcsai úgy, hogy két szomszédos (éllel összekötött) csúcs se legyen azonos színű. A megfelelő tanú maga a színezés.

Hamilton-kör: Adott egy gráf, s mondjuk meg, hogy létezik-e benne olyan kör, mely minden csúcsot pontosan egyszer tartalmaz. A tanú maga a Hamilton-kör.

SAT: Kielégíthető-e egy Boole-formula? A tanú maga az értékelés.

A kutatások során az derült ki, hogy az előbb felsorolt feladatok egyformán nehéz problémák.
Sokan sejtik, de bizonyítani még nem sikerült, hogy P$\neq$NP.

IV. fejezet
Rendezések

Adott n szám. Milyen módon lehet ezeket nagyság szerint növekvő sorrendbe rendezni? A lehetséges megoldásokat rendszerint az alábbi csoportok egyikébe lehet besorolni:

Beszúró rendezés: Egyesével tekinti a rendezendő számokat, és mindegyiket beszúrja a már rendezett számok közé, a megfelelő helyre. (Bridzsező módszer)

Cserélő rendezés: Ha két szám nem megfelelő sorrendben következik, akkor felcseréli őket. Ezt az eljárást ismétli addig, amíg további cserére már nincs szükség.

Kiválasztó rendezés: Először a legkisebb (legnagyobb) számot határozza meg, ezt a többitől elkülöníti, majd a következő legkisebb (legnagyobb) számot határozza meg, stb.

Leszámoló rendezés: Minden számot összehasonlítunk minden más számmal; egy adott szám helyét a nála kisebb számok száma határozza meg.

1. Beszúró rendezés

Az előbbi rövid leírásnak megfelelő pszeudókód a következő:

Procedure BESZÚRÓ(A)

Input: Az A tömb
Eredmény: Az A tömb elemei nagyság szerint rendezi

```
1  for j ← 2 to A.hossz do
2      kulcs ← A[j]
3      //A[j] beszúrása az A[1..j − 1] rendezett sorozatba
4      i ← j − 1
5      while i > 0 and A[i] > kulcs do
6          A[i + 1] ← A[i]
7          i ← i − 1
8      endw
9      A[i + 1] ← kulcs
10 endfor
```

A programban az `A[i]` jelöli az A tömb i-dik elemét, és `A.hossz` adja meg az A tömb méretét.

Példa. Tartalmazza az A tömb az 5, 2, 4, 6, 1 és 3 számokat! A tömb tartalma a következőképpen változik az algoritmus végrehajtása során:

j értéke	A tömb tartalma					
	5	2	4	6	1	3
2	2	5	4	6	1	3
3	2	4	5	6	1	3
4	2	4	5	6	1	3
5	1	2	4	5	6	3
6	1	2	3	4	5	6

A táblázatban az arany színű számok már rendezve vannak. A beszuro.htm fájl tartalmazza azt a programot, amely egy véletlen módon generált számsorozatra végrehajtja a beszúró rendezés és ciklusról ciklusra kiírja a tömb tartalmát.

Az algoritmus elemzése. Jelöljük a c_1 c_2, ..., c_9 konstansokkal, hogy mennyi idő (hány elemi lépés) kell az első, második, ..., kilencedik sor végrehajtásához, és jelölje t_j azt, hogy a `while` ciklus hányszor hajtódik végre a j érték esetén. (A nyolcadik és a tizedik sor csak a ciklus végét jelzi, ezek éppúgy nem végrehajtható utasítások, mint a harmadik sorban található megjegyzés.) Ha n jelöli az `A` tömb hosszát, akkor a program futása

$$c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_9(n-1)$$

ideig tart. Ha a sorozat növekvően rendezett, akkor a while ciklus nem hajtódik végre, azaz a t_j értéke minden esetben 1. Egyszerűsítés és átrendezés után az előbbi képlet

$$(c_1 + c_2 + c_4 + c_5 + c_9)n - (c_2 + c_4 + c_5 + c_9)$$

alakra egyszerűsödik. Ebből leolvasható, hogy a futásidő a hossz lineáris függvénye. Ha a sorozat csökkenően rendezett, akkor a while ciklust minden egyes megelőző elemre végre kell hajtani, azaz $t_j = j$. Ekkor a futási idő

$$c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6 \frac{n(n+1)}{2} + c_7 \frac{n(n+1)}{2} + c_9(n-1),$$

azaz a hossz négyzetes függvénye. Ez az eset a legrosszabb eset, így a beszúró rendezés bonyolultsága $O(n^2)$.

2. Oszd meg és uralkodj!

A politikában használatos elvnek az informatikában is hasznát vehetjük. A módszer alapelve a következő:

(1) Felosztjuk a problémát több alproblémára.
(2) Uralkodunk az alproblémákon.
   – Ha az alprobléma mérete kicsi, akkor azt közvetlenül megoldjuk.

– Egyébként rekurzív megoldást alkalmazunk, azaz az alproblémákat újra az oszd meg és uralkodj elvét alkalmazva oldjuk meg.

(3) Összevonjuk az alproblémák megoldásait az eredeti probléma megoldásává.

2.1. Összefésülő rendezés

Az összefésülő rendezés is ezt az elvet követi A lépések ebben az esetben a következők:

(1) Az n elemű sorozatot felosztja két $\frac{n}{2}$ elemű alsorozatra.
(2) A két alsorozatot összefésülő rendezéssel rekurzívan rendezi.
(3) Összefésüli a két sorozatot, létrehozva a rendezett választ.

Példa. Rendezzük az alábbi számsorozatot az összefésülő rendezéssel!

```
 5  2  4  6  1  3  2  6
[5  2  4  6][1  3  2  6]
[5  2][4  6][1  3][2  6]
[5][2][4][6][1][3][2][6]
[2  5][4  6][1  3][2  6]
[2  4  5  6][1  2  3  6]
 1  2  2  3  4  5  6  6
```

Először is fel kell bontani a sorozatot két négyes csoportra, s ezeket kell különkülön rendezni. Ehhez a négyeseket kettesekre kell bontani, s azokat rendezni. Miután továbbra is összefésülő rendezést használunk, a ketteseket is egyesekre bontjuk. Egy szám magában már rendezett, így elkezdhetjük a harmadik lépést, az összefésülést. Itt a rendezett sorozatok első elemeit kell figyelni, s a kettő közül a kisebbiket kell egy másik tárolóban elhelyezni, s törölni a sorozatból. Ha mindkét sorozat elfogy, a másik tárolóban a két sorozat összefésültje található. Így fésülhetőek össze az egyesek, majd a kettesek, s végül a négyesek.
A ofesulo.htm állomány tartalmazza azt a programot, amely egy véletlen módon generált számsorozatra végrehajtja az összefésülő rendezést. A program különböző szinekkel jelzi a két részsorozatot, s az összefésült sorozatokat.
A módszer elemzése. Ha $T(n)$ jelöli a probléma futási idejét, $D(n)$ a probléma alproblémákra bontásának idejét, $C(n)$ a alproblémák megoldásának összevonását, a darab alproblémára bontottuk az eredeti problémát, és egy alprobléma mérete az eredeti probléma méretének $1/b$ része, valamint c jelöli a triviális feladatok megoldásának idejét, akkor a következő összefüggést írhatjuk fel:

$$T(n) = \begin{cases} c, & \text{ha } n \text{ kicsi} \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), & \text{egyébként.} \end{cases}$$

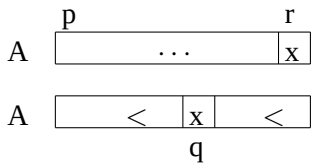
Összefésülő rendezés esetén

$$T(n) = \begin{cases} c, & \text{ha } n = 1 \\ 2T\left(\frac{n}{2}\right) + dn, & \text{egyébként.} \end{cases}$$

Ha a d és c konstansok közel egyformák, belátható, hogy $T(n) = c \cdot n \cdot \ln(n)$.

2.2. Gyorsrendezés

A gyorsrendezés is az „oszd meg és uralkodj” elvet követi. Ehhez az aktuális elemeket két csoportra bontja úgy, hogy az első csoport elemei mind kisebbek a második csoport elemeinél, majd ezeket külön-külön rendezzük.



A FELOSZT rutin a megadott tömb két indexe közti elemeket válogatja szét az alapján, hogy a második index által mutatott elemnél (x) kisebbek, vagy nagyobbak-e. A kisebb elemeket a résztömb elejére, a nagyobb elemeket a résztömb végére csoportosítja.

Function FELOSZT(A,p,r)

Input: Az A tömb, s két indexe, ahol $p < r$
Output: q, az az index, ahova az r indexnél található elem került.
Eredmény: Az A tömb elemeit átcsoportosítja úgy, hogy a p-től q-ig terjedő elemek kisebbek, mint az q-től r-ig terjedő elemek

```
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1 do
4  |  if A[j] ≤ x then
5  |  |  i ← i + 1
6  |  |  A[i] és A[j] cseréje
7  |  endif
8  endfor
9  A[i + 1] és A[r] cseréje
10 return i+1
```

A következő táblázatban az arany színű számok azok, melyekről kiderült, hogy a második index által mutatott számnál kisebbek, és világoskékek a nála nagyobbak. A példánkban $p = 1$ és $r = 6$.

A tömb elemei						i	j
4	2	5	6	1	3	0	1
2	4	5	6	1	3	1	2
2	4	5	6	1	3	1	3
2	4	5	6	1	3	1	4
2	1	5	6	4	3	2	5
2	1	3	6	4	5	0	6

A FELOSZT rutin az „oszd meg és uralkodj elvéből” a felosztást végzi. Mivel a q index előtt a q indexnél szereplő számnál kisebb, mögötte pedig nagyobb

számok szerepelnek, a harmadik lépésre, a megoldások összevonására nincs szükség. Egyedül a rekurzív függvényhívások hiányoznak. Ezt a GYORSRENDEZÉS rutin valósítja meg. Mivel ez a rutin a megadott két index közötti részt rendezi, az A tömb egészének rendezéséhez a rutint a `GYORSRENDEZÉS(A,1,A.hossz)` módon kell indítani.

Procedure GYORSRENDEZÉS(*A,p,r*)
Input: A tömb, s a rendezendő résztömböt meghatározó indexek
Eredmény: a tömb megadott indexei közti részt nagyság szerint rendezi

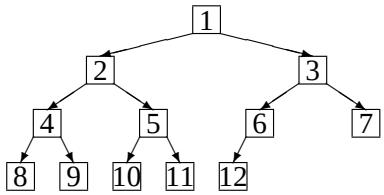
```
1 if p < r then
2 |   q ← FELOSZT(A,p,r)
3 |   GYORSRENDEZÉS(A,p,q-1)
4 |   GYORSRENDEZÉS(A,q+1,r)
5 endif
```

A gyorsrendezésben a partícionálásnak más módszerei is ismertek, például a két végéről haladunk a sorozatnak, elől/hátul átlépdelünk a kijelölt elemnél kisebb/nagyobb elemeken, majd ha nem ért össze a két mutató, kicseréljük a két mutató által jelölt elemeket, s folytatjuk az átlépdelést.
A módszer elemzése A gyorsrendezés hatékonysága azon múlik, hogy felosztás során mennyire egyforma méretű alsorozatokat kapunk. A legrosszabb esetben a sorozat rendezett, ekkor a módszer négyzetes függvénye a hossznak. Legjobb esetben $O(n\ln(n))$ lesz a futás bonyolultsága. Mivel rendezett, illetve közel rendezett esetekben a legrosszabb futási eredményeket kapjuk, ekkor szokásos a sorozat középső elemét választani $A[r]$ helyett. Általános esetben pedig a sorozat egy véletlen elemét.
A gyors.htm állomány tartalmazza azt a programot, amely egy véletlen módon generált számsorozatra végrehajtja a gyorsrendezést. A vgyors.htm állomány ennek a programnak azt a változatát tartalmazza, amely egy véletlenül választott elem alapján osztja szét a sorozatokat.

3. Gráfelméleti alapfogalmak

Egy G *gráf* két halmazból áll: a *csúcs*ok vagy pontok V halmazából, ami egy nemüres halmaz, és az E *él*ek halmazából, melynek elemei V-beli párok. Ha ezek a párok rendezettek, akkor *irányított*, ellenkező esetben *irányítatlan* gráfokról beszélünk. Gyakran csak arra vagyunk kíváncsiak, hogy két csúcs között van-e él, míg máskor ehhez az élhez valamilyen költségeket is rendelünk. *Út*on egy olyan $v_1, \ldots, v_k$ csúcssorozatot értünk, melyre (v_i, v_{i+1}) éle a gráfnak. Az utat körnek nevezzük, ha kezdő- és végpontja megegyezik (ám nincs más közös pontja). A csúcsok halmazán értelmezhetünk egy relációt aszerint, hogy a két kiválasztott

csúcs között (az irányítástól eltekintve) vezet-e út. Ezen ekvivalenciareláció ekvivalenciaosztályait *komponens*eknek hívjuk. Egy körmentes gráfot *erdő*nek nevezünk, s az egy komponensből álló erdőt *fá*nak. A *bináris fa* egy olyan fa, melynek a csúcsai szinteken helyezkednek el. A legfelső szinten pontosan egy csúcs van, ezt *gyökérnek* nevezzük. Egy tetszőleges x csúcsból legfeljebb két csúcs indul ki, ezek eggyel alacsonyabb szinten levő csúcsokhoz vezetnek. A balra menő él végpontja az x bal fia, a jobbra menő él végpontja az x jobb fia. Azokat a csúcsokat, melyeknek nincs fia, leveleknek nevezzük. Az alábbi ábrán látható egy bináris fa.



Az informatikában megszokott módon a fák ábrázolásakor a fa gyökere kerül felülre, és a levelek alulra. Az ábrán láthető fa speciális, *teljes fa*. A teljes fa levelei két szinten helyezkednek el, és legfeljebb egy kivétellel minden nem levél csúcsnak két fia van. Ha pedig sorfolytonosan tekintjük a fát, egyik csúcsnak sincs kevesebb fia, mint a sorban utána következőnek. Az ilyen fát tárolhatjuk tömbben, illetve egy tömböt tekinthetünk fának. Az előbbi fa csúcsaiban szereplő számok a tömb megfelelő elemeinek az indexét jelölik. Ha a fa csúcsai az A tömb $1, \ldots, n$ elemei, akkor az $A[i]$ bal fia $A[2i]$, míg a jobb fia $A[2i+1]$. Hasonlóan, ha $j > 1$, akkor az $A[j]$ apja $A[\lfloor \frac{i}{2} \rfloor]$, ahol az $\lfloor \cdot \rfloor$ az egészrész függvényt jelenti. Például az $A[5]$ fiai az $A[10]$ és az $A[11]$ csúcsok, míg az $A[9]$ és $A[8]$ csúcsok apja az $A[4]$ csúcs.

4. Kupac

Egy teljes bináris fát *kupac*nak tekintjük, ha erre a fára teljesül a *kupac tulajdonság*: egy tetszőleges csúcs eleme nem lehet nagyobb a fiaiban levő elemeknél.

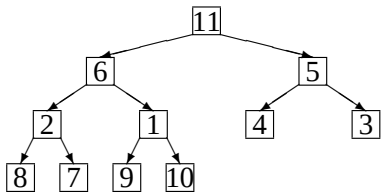
4.1. Kupacépítés

A bináris fa egy eleme és annak leszármazottjai egy részfát határoznak meg, melynek a gyökere az adott elem. Apák és fiaik elemeinek cseréjével elérjük, hogy egyre nagyobb és nagyobb részfákban teljesüljön a kupac tulajdonság. A csak levélből álló részfára természetesen egyből teljesül a kupac-tulajdonság. Ezért a tömb utolsó elemétől az első irányába haladva a következőket hajtjuk végre: ha $A[j] > \min(A[2j], A[2j+1])$, akkor apa és a kisebbik értékű fia elemei helyet cserélnek, majd rekurzívan hívjuk meg a KUPAC eljárást a szóban forgó fiúra. A haladási irány miatt eredetileg a fiúkra teljesült a kupac tulajdonság. Ez a cserével esetleg felborult, ezért ezt az adott fiúnál újra meg kell vizsgálni.

Procedure KUPAC-ÉPÍTŐ(A)
Input: Az A számtömb
Eredmény: Az A tömb kupac tulajdonságú
1 **for** $i \leftarrow \lfloor A.hossz/2 \rfloor$ ***downto*** 1 **do** KUPAC(A,i)

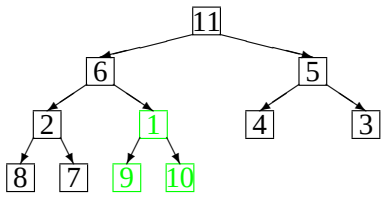
Procedure KUPAC(A,i)
Input: Az A számtömb, i index
Eredmény: Az A[i] gyökerű részfa kupac tulajdonságú
1 $b \leftarrow 2i$;
2 $j \leftarrow 2i + 1$;
3 **if** $b \leq A.hossz$ ***and*** $A[b] < A[i]$ **then** $k \leftarrow b$ **else** $k \leftarrow i$;
4 **if** $j \leq A.hossz$ ***and*** $A[j] < A[k]$ **then** $k \leftarrow j$;
5 **if** $k \neq i$ **then**
6 | $A[i]$ és $A[k]$ cseréje;
7 | KUPAC(A,k)
8 **endif**

Hosszas számolások után belátható az a meglepő tény, hogy a kupacépítés, azaz egy tetszőleges tömb kupaccá alakítása lineáris bonyolultságú, melynek bizonyítása megtalálható Knuth könyvében [3, 171.o].
Példa. Lássuk, hogyan építhető kupac a 11, 6, 5, 2, 1, 4, 3, 8, 7, 9 és 10 elemeket tartalmazó tömből! A könnyebb követhetőség érdekében ábrázoljuk a tömböt fával



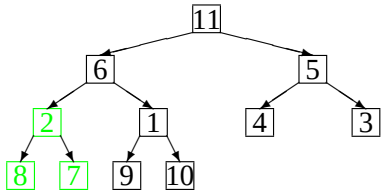
1. ábra.

(1. ábra)! A KUPAC-ÉPÍTŐ eljárás alapján, mivel 11 elem van a tömbben, elsőként



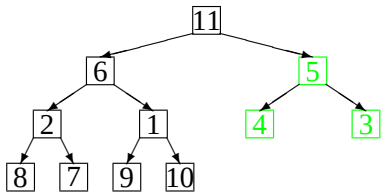
2. ábra.

az ötödik elemet kell összehasonlítani a tizedikkel és tizenegyedikkel. Az ötödik a legkisebb, ezért nem változik semmi (2. ábra). Ezután a negyediket kell összeha-



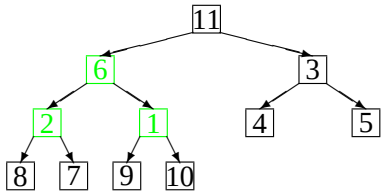
3. ábra.

sonlítani a nyolcadikkal és kilencedikkel, s a felső elem újra kisebb a többieknél (3. ábra). Majd a harmadikat kell összehasonlítani a hatodikkal és hetedikkel (4.



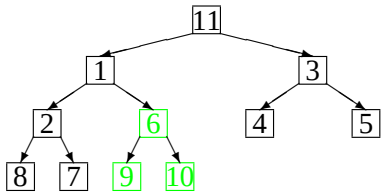
4. ábra.

ábra). A három szám közül a 3 a legkisebb, ezért ez kerül fel a harmadik helyre. Az itt található 5-öt a hetedik elem gyökerű kupacban kell elhelyezni, de mivel ez a fa csak a gyökérből áll, végeredményben a 3 és 5 helyet cserél. A soron következő



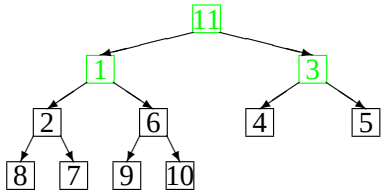
5. ábra.

lépésnél az 1 és a 6 cserél helyet (5. ábra). Majd ellenőrizni kell, hogy a 6 valóban

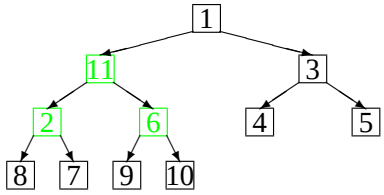


6. ábra.

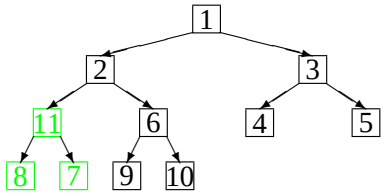
a helyére került-e. Mivel kisebb mint a 9, illetve a 10, ezért már nem kell tovább mozgatni (6. ábra). Ezután folytathatjuk a fa vizsgálatát az első elemnél, s az 1 és 11 helyet cserél (7. ábra). A 11 még nem került a helyére, mert van olyan fia



7. ábra.

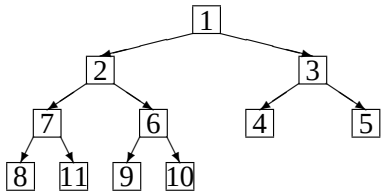


8. ábra.



9. ábra.

(mindkettő ilyen), mely nála kisebb. Ezért kicseréljük a kisebbikkel (8. ábra). A 11 még mindig nem került a helyére, így újabb csere következik (9. ábra). De most



10. ábra.

már minden a helyén van, tehát elkészült a kupac (10. ábra).

4.2. Kupacrendezés

A kupac-tulajdonság miatt a gyökérhez tartozó eleme a legkisebb. Ezt az elemet a kupacból törölve, helyére a tömb utolsó elemét írva, s a tömböt újra kupaccá rendezve megkaphatjuk a tömb következő elemét. Ezt módszert újra és újra végrehajtva sorra megkapjuk a tömb elemeit rendezve (esetünkben csökkenő sorrendben). Ezt nevezzük *kupacrendezésnek*. A kupacrendezés $O(n \ln(n))$ bonyolultságú. A

törölt elemek külön helyen tárolása helyett tárolhatjuk az elemeket a tömb törlés miatt felszabadult helyein.

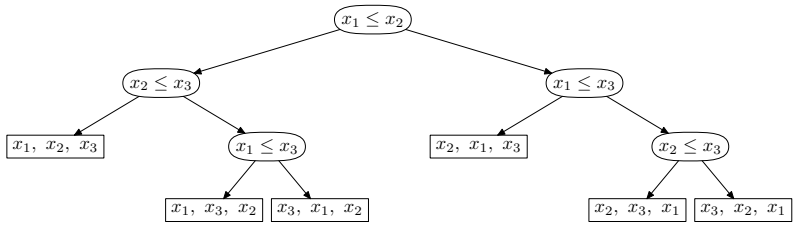
Procedure KUPAC-RENDEZÉS(A)
Input: Az A számtömb
Eredmény: A tömböt csökkenő sorrendbe rendezi

```
1 KUPAC-ÉPÍTŐ (A)
2 n ← A.hossz;
3 for i ← n downto 2 do
4 |   A[1] és A[i] cseréje;
5 |   A.hossz ← A.hossz − 1;
6 |   KUPAC(A,1);
7 endfor
8 A.hossz ← n;
```

A kupac.htm állomány tartalmazza azt a programot, amely egy véletlen módon generált számsorozatra végrehajtja a kupacrendezést. A program pirossal jelzi az apát és a két fiát, s kékkel a már rendezett számokat.

5. Lineáris idejű rendezések

A beszúró, az összefésülő, a gyors- és a kupacrendezésnél azt használtuk fel, hogy a sorozat két eleme között milyen reláció áll fenn. Könnyen belátható, hogy vizsgálatainkat leszűkíthetjük egyedül a kisebb-egyenlő vizsgálatára. Ezt az egy relációt használva döntési fákat készíthetünk, melyben ha a fa adott csúcsában szereplő kérdésre igaz a válasz, akkor a csúcs bal fiánál folytatjuk a vizsgálódást, különben a jobb fiánál. Ha elértünk egy levélhez, akkor az ott szereplő lista az elemek rendezését adja meg. Három elem esetén a következő ábra tartalmazza a döntési fát:



Miután az n elem összes permutációjának szerepelnie kell a döntési fa leveleiben, ebből az következik, hogy a döntési fa magassága $n\ln(n)$-nel lesz arányos, ha a döntési fa optimális felépítésű. Azaz legalább $n\ln(n)$ összehasonlítást kell elvégezni n elem rendezéséhez. Magyarul az optimális lépésszám $n\ln(n)$-nel arányos, ennél jobb bonyolultságú rendezés általános esetben nem létezik. Tehát a kupacrendezés és az összefésülés optimális rendezési módszer.

6. Leszámláló rendezés (ládarendezés)

Az előbb láttuk, hogy mind a gyorsrendezés, mind a kupacrendezés $O(n\ln(n))$ bonyolultságú, s hogy ennél jobb módszert általános esetben nem találunk. A leszámláló rendezés viszont bizonyos esetekben lineáris bonyolultságú. Ez a viszonylagos ellentmondás azzal oldódik fel, hogy a rendezendő n elem mindegyike 1 és k között helyezkedik el. A leszámláló rendezés alapötlete az, hogy meghatározza minden egyes x bemeneti elemre azoknak az elemeknek a számát, melyek kisebbek, mint az x. Ezzel az x elemet egyből a saját pozíciójába lehet helyezni a kimeneti tömbben. A módszer bonyolultsága $O(n+k)$, így ha $k = O(n)$, akkor a módszer lineáris. A leszamlalo.htm állomány tartalmazza azt a programot, amely egy véletlen módon generált számsorozatra végrehajtja a leszámláló rendezést.

Procedure LESZÁMLÁLÓ-RENDEZÉS(*A,B,k*)

Input: Az A és B számtömb, k maximális adat
Eredmény: Az A tömb elemeit a B tömbbe nagyság szerint rendezi

```
1  //A számláló tömb törlése
2  for i ← 1 to k do
3  |  C[i] ← 0;
4  endfor
5  //Mely kulcs pontosan hányszor fordul elő?
6  for j ← 1 to A.hossz do
7  |  C[A[j]] ← C[A[j]] + 1;
8  endfor
9  //Hány kisebb vagy egyenlő kulcsú elem van a sorozatban
10 for i ← 2 to k do
11 |  C[i] ← C[i] + C[i − 1];
12 endfor
13 for j ← A.hossz downto 1 do
14 |  B[C[A[j]]] ← A[j];
15 |  C[A[j]] ← C[A[j]] − 1;
16 endfor
```

7. Számjegyes (radix) rendezés

Ha a kulcsok összetettek, több komponensekből állnak, akkor rendezhetünk az egyes komponensek szerint. Például a dátum az év, hónap, nap rendezett hármasából áll. Ha a kulcsok utolsó komponense szerint rendezünk, majd az eredményt az utolsó előtti komponens szerint, és így tovább, akkor végül rendezett sorozathoz jutunk.

Az egész számok tekinthetőek bitsorozatoknak, s a rendezés minden egyes fordulójában két sorozattá bontjuk a kiinduló, illetve az eredményül kapott sorozatokat, aszerint, hogy a vizsgált bit 0 illetve 1. E két lista egymás után fűzéséből kapható meg a soron következő sorozat. (Természetesen nemcsak kettes, hanem bármilyen más számrendszerbeli számokként is tekinthetnénk a sorozat elemeit, s például négyes számrendszer esetén négy sorozattá bontatánk a sorozatot.) Kettes számrendszer használata esetén, ha a számok 0 és $2^k - 1$ közé esnek, akkor a bonyolultság $O(nk)$.

Példa.

Legyenek a számaink

4 9 13 15 5 2 10 7 1 8

Ezek kettes számrendszerben a következőek:

0100 1001 1101 1111 0101 0010 1010 0111 0001 1000

Az utolsó bit szerint szétválasztva az előbbi listát a következőt kapjuk:

0100 0010 1010 1000 1001 1101 1111 0101 0111 0001

A harmadik bit szerint

0100 1000 1001 1101 0101 0001 0010 1010 1111 0111

A második bit szerint

1000 1001 0001 0010 1010 0100 1101 0101 1111 0111

S végül az első bit szerint rendezve

0001 0010 0100 0101 0111 1000 1001 1010 1101 1111

Amelyek tízes számrendszerben

1 2 4 5 7 8 9 10 13 15

tehát valóban rendeztük a sorozatot.

8. Külső rendezés

A korábbi rendezéseknél feltettük, hogy az adatok a számítógépek belső memóriájában találhatóak, s az adatok összehasonlításának, mozgatásának ideje hasonló. Ha viszont a rendezendő adatok nem férnek el egyszerre a belső memóriában, az adatok elérése, mozgatása nagyságrendekkel tovább tart az egyszerű összehasonlításoknál, és a korábban ismertetett rendezési módszerek nagyon rossz eredményeket adnak. A külső tárakon tárolt adatok elérésének gyorsítására már évtizedek óta azt a módszert használjuk, hogy nem egyesével olvassuk be az adatokat, hanem egyszerre egy *lapnyi/blokknyi* információt olvasunk be. Ezért a következőkben azt vizsgáljuk, hogy az adatok rendezése hány újraolvassal oldható meg.

Az összefésülő rendezésre hasonlító *külső összefésülést* gyakran használták korábban. Itt az eredeti állományból a belső memóriát megtöltő részeket másoltak át, azt ott helyben rendezték, ezzel úgynevezett futamokat hoztak létre, s a futamokat felváltva két állományba írták ki. Ezután a két állomány összefésülték, s ezzel a dupla hosszú futamokat hoztak létre, amit szintén két állományba írtak ki. Ezt folytatták mindaddig, amig végül egy futam maradt, ami tartalmazott minden adatot. Könnyen belátható, hogy a fázisok száma logaritmikusan függ a kezdeti futamok

számától. Ezért érdemes minél hosszabb kezdő futamokkal dolgozni, (Természetesen nemcsak két input és output állománnyal lehet dolgozni, hanem többel is. Az állományok számát a hardver lehetőségei korlátozzák. A több állomány használata lecsökkenti a menetek számát.)
Példa. Az A állomány tartalmazzon 5000 rekordot, a memóriába pedig csak 1000 rekord férjen! A kezdeti 1000 rekord hosszúságú futamok elkészítése után a szalagok a következőeket tartalmazzák:

- A (input): üres
- B (output): R1-R1000,R2001-R3000,R4001-R5000
- C (output): R1001-R2000,R3001-R4000
- D: üres

A B és C állományok összefésülésével 2000 hosszú futamok készülnek.

- A (output): R1-R2000,R4001-R5000
- B (input): üres
- C (input): üres
- D (output): R2001-R4000

Újabb összefésüléssel elkészülnének a 4000 rekord hosszúságú futamok.

- A (input): üres
- B (output): R1-R4000
- C (output): R4001-R5000
- D (input): üres

Már csak egy összefésülés van hátra.

- A (output): R1-R5000
- B (input): üres
- C (input): üres
- D: üres

S az A állományban rendezve szerepel az összes elem.

9. Medián, minimális, maximális, i-dik legnagyobb elem

A sorozat minimális illetve a maximális elemének meghatározásához végig kell lépdelnünk az összes elemen, s megvizsgálni, hogy az adott elem kisebb-e/nagyobb-e mint az eddig talált minimum/maximum. Ezért a minimum/maximum meghatározása lineáris feladat. Az i-dik elem meghatározására a feltételektől függően más és más módszert érdemes használni.

- Ha viszonylag kis számú kulcs fordul elő, akkor a leszámláló rendezésben ismertett módszerrel a C tömbből könnyedén meghatározható az i. legnagyobb/legkisebb szám.
- Ha i kicsi n-hez képest, akkor a kupacrendezés elvét használva, a kupacból i számot törölve megkapjuk az i. legnagyobb/legkisebb számot.
- A gyorsrendezés az alsorozat elemeit két részre osztotta: a kijelölt elemnél kisebb elemek és annál nagyobbak sorozatára. Eme sorozatok hossza

alapján dönthetünk, hogy mely sorozatban található a keresett elem. Ezt a módszert követve átlagosan lineáris időben kereshetjük meg az i. elemet.

V. fejezet
Dinamikus halmazok

A számítástudományban gyakran használjuk a halmazokat. Az algoritmusok futása közben ezek a halmazok változhatnak, bővülhetnek, zsugorodhatnak. Rendszerint egy elem halmazba szúrására, adott elem törlésére, illetve arra van szükség, hogy eldöntsük, egy elem eleme-e a halmaznak. Az adataink/rekordjaink általában tartalmaznak egy kulcsmezőt, s esetleg kiegészítő adatokat. A most ismertetésre kerülő algoritmusokban a kiegészítő adatokat nem vesszük figyelembe, csak a kulcsmezőre, annak értékére összpontosítunk. A kulcsmezők lehetséges értékei egy teljesen rendezett halmazból származnak. Erre azért van szükségünk, hogy két tetszőleges értéket összehasonlíthassunk. Az itt használt jelölést a XII. fejezet írja le.

1. Műveletek típusai

A dinamikus halmazokon a következő *módosító* műveleteket végezhetjük:

- BESZÚR(S,x) az S halmazt bővítjük az x elemmel.
- TÖRÖL(S,x) az S halmazból töröljük az x elemet.

A dinamikus halmazokon a következő *lekérdező* műveleteket végezhetjük:

- KERES(S,x) az S halmazban megadja az x elem helyét, ha az a halmaznak eleme.
- MINIMUM(S) az S halmaz minimális elemének a helyét adja meg.
- MAXIMUM(S) az S halmaz maximális elemének a helyét adja meg.
- KÖVETKEZŐ(S,x) megadja az S halmaz azon elemének a helyét adja meg, amely a rendezés alapján követi az x elemet.
- ELŐZŐ(S,x) megadja az S halmaz azon elemének a helyét adja meg, amely a rendezés alapján megelőzi az x elemet.

2. Keresés

A bináris fák egy speciális osztálya a bináris keresőfák. A definiáló tulajdonság a következő: legyen `x` és `y` a fa egy-egy olyan csúcsa, hogy az `x` az `y` őse. Ha `y` az `x` baloldali fájában található, akkor `y.kulcs` $\leq$ `x.kulcs`, míg ha a `y` az `x` jobboldali fájában található, akkor `x.kulcs`$\leq$ `y.kulcs`.
Egy adott elem keresése a következőképpen történik. A fa gyökérében kezdünk, és a keresett kulcsot összehasonlítjuk a gyökér kulcsával. Ha a kulcsok megegyeznek, kész vagyunk. Ha a keresett kulcs kisebb a gyökér kulcsánál, akkor a keresett elem

a baloldali részfában található, ha egyáltalán szerepel a fában. Ellenkező esetben a jobboldali részfában kell keresni. A részfának is tekintjük a gyökerét, ennek kulcsát is összehasonlítjuk a keresett kulccsal, s ezt folytatjuk mindaddig, amíg rá nem találunk a kulcsra, vagy a keresett fa üres nem lesz. A Keres függvényt megírhatjuk rekurzív és iteratív formában is.

Function KERES(*x*,*k*) rekurzív változat

Input: x a fa gyökerének címe, k a keresett csúcs
Output: A keresett elem címe, illetve Nil

```
1 if x = Nil or k = x.kulcs then return x
2 if k < x.kulcs then Keres(x.bal,k) else Keres(x.jobb,k)
```

Function KERES(*x*,*k*) iteratív változat

Input: x a fa gyökerének címe, k a keresett csúcs
Output: A keresett elem címe, illetve Nil

```
1 while x ≠ Nil and k ≠ x.kulcs do
2 |   if k < x.kulcs then x ← x.bal else x ← x.jobb
3 endw
4 return x
```

Miután az adott csúcstól balra eső elemek kisebb kulccsal rendelkeznek, a leginkább balra található elem rendelkezik a minimális kulccsal.

Function MINIMUM(*x*)

Input: x a fa gyökerének a címe
Output: a minimális elemet tartalmazó csúcs címe

```
1 while x.bal ≠ Nil do x ← x.bal return x
```

A keresés, a minimális, maximimális kulcs megkeresésének bonyolultsága a keresett elem magasságával arányos, ami a legrosszabb esetben $O(n)$. Néha szükség van az előző és következő elem meghatározására. A fabejárásokkal lineáris bonyolultsággal megoldható a probléma, de létezik egyszerűbb megoldás is. Ha az adott elemnek van jobboldali részfája, akkor eme részfa minden elemének kulcsa nagyobb az x kulcsánál. Ezek közül kell a minimális, tehát ennek a részfának minimális elemére vagyunk kíváncsiak. Ellenkező esetben azt a legfiatalabb őst kell megkeresni, melynek baloldali részfájában szerepel ez az elem, azaz a bal mutatója mutat felé.

3. Naív beszúrás

A kereséshez hasonlóan a beszúrás is a fa gyökeréből indul. Az éppen vizsgált csúcs, és a beszúrandó elem kulcsai alapján a csúcs bal- vagy jobboldali fiánál folytatjuk a vizsgálatot. Ha már megtaláltuk az elem helyét, akkor levélként beszúrjuk.

Function KÖVETKEZŐ(x)

Input: x a fa gyökerének a címe
Output: a következő elemet tartalmazó csúcs címe, illetve Nil

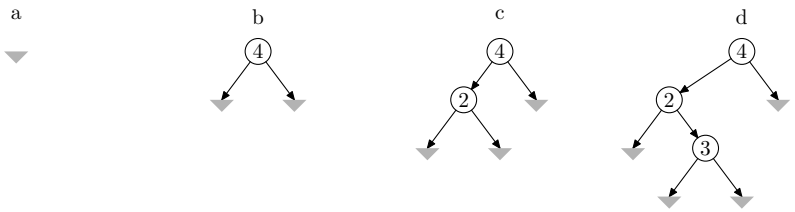
```
1 if x.jobb ≠ Nil then return MINIMUM(x.jobb)
2 y ← x.apa  // y legyen x szülője
3 while y ≠ Nil and x = y.jobb do
4 |   x ← y
5 |   y ← y.apa
6 endw
7 return y
```

Procedure BESZÚR(T,z)

Input: T a fa, z a beszúrandó csúcs
Eredmény: A T fába beszúrja a z csúcsot

```
1 y ← Nil
2 x ← T.gyöker
3 while x ≠ Nil do
4 |   y ← x
5 |   if z.kulcs < x.kulcs then x ← x.bal else x ← x.jobb
6 endw
7 z.apa ← y
8 if y = Nil then
9 |   T.gyöker ← z
10 else
11 |   if z.kulcs < y.kulcs then y.bal ← z else y.jobb ← z
12 endif
```

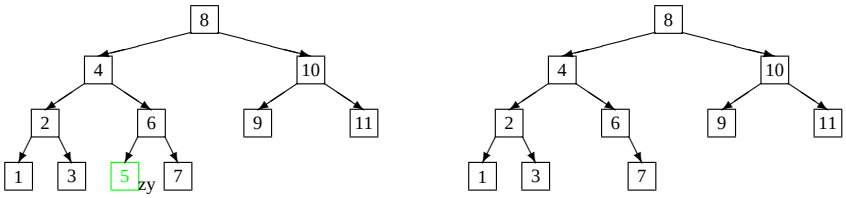
Példa.



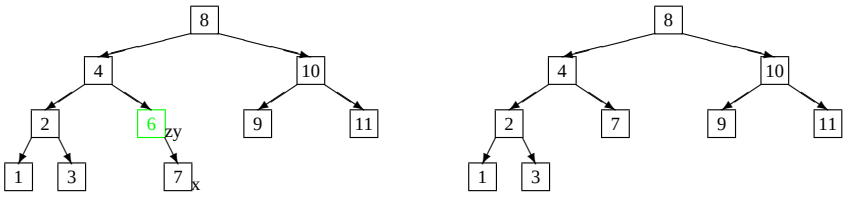
Az előbbi ábrán egy üres fával kezdtünk (a). Ezután a 4 beszúrásakor y és x is Nil értéket kap, így az algoritmus végrehajtásakor be sem lépünk a ciklusba. Az új elem lesz a fa gyökere (b). A soron következő elem kulcsa (2) kisebb mint a gyökéré (4), ezért a ciklus egyszer végrehajtódik. Az y a gyökérre mutat, ez alá fűzzük az új elemet. Mivel a kulcsa kisebb a gyökér kulcsánál, a gyökér bal oldalára kerül az új elem (c). Az utolsó elem kulcsa kisebb mint a gyökérelemé, így annak baloldali részfájába kerül. Viszont ennek a részfa gyökérkulcsától nagyobb kulcsa van az új elemnek, tehát a jobb oldalra kell felfűzni (d).

4. Naív törlés

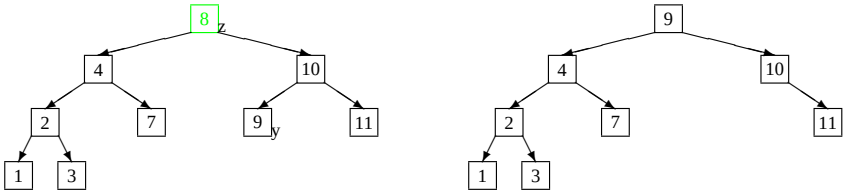
Új elemet a fa leveleként szúrtuk be. Ennek megfelelően levelet könnyű törölni.



Nem nehéz a törlés akkor sem, ha a törlendő csúcsnak csak egy utódja van. Ekkor az elemet a megfelelő részfával kell helyettesíteni.



Ha viszont egyik részfa sem üres, akkor kicsit bonyolultabb a helyzet. Lyuk nem maradhat a fában, s a beszúrandó elemnek nagyobbnak kell lennie, mint a megmaradó bal részfa összes elemének, s kisebbnek mint a megmaradó jobb részfa összes elemének. Két lehetséges megoldás lehet: a bal részfa maximális, vagy a jobb részfa minimális eleme. Ezen elemek egyikét törölni kell a régi helyéről, s a törlendő elem helyére rakni. Az előbbi ábrákon és a következő programban y



jelzi, hogy mely csúcs törlődik valóban. Az x az y utódát jelzi, s az y-ra mutató mutatónak ezután az x-re kell mutatnia.
A beszúrás és a törlés bonyolultsága a keresett elem magasságának lineáris függvénye $O(h)$, hasonlóan a kereséshez, vagy a minimum, maximum meghatározásához. A bin-fa.htm állomány tartalmazza azt a programot, amellyel egy bináris fába lehet elemeket beszúrni, illetve onnan törölni.

5. Piros-fekete fák

Ahogy azt láttuk, a lekérdező és módosító műveletek bonyolultsága arányos a vizsgált elem magasságával. Ezért a megközelítőleg teljes fákban optimálisak ezek a

Function TÖRÖL(*T,z*)

Input: T a fa, z a törlendő csúcs
Eredmény: A T fából törli a z csúcsot

```
1  if z.bal = Nil or z.jobb = Nil then y ← z else y ← KÖVETKEZŐ(z)
2  if y.bal ≠ Nil then x ← y.bal else x ← y.jobb
3  if x ≠ Nil then x.apa ← y.apa
4  if y.apa = Nil then
5  |  T.gyökér ← x
6  else
7  |  if y = y.apa.bal then y.apa.bal ← x else y.apa.jobb ← x
8  endif
9  if y ≠ z then
10 |  z.kulcs ← y.kulcs
11 |  // és át kell másolni a további mezőket is
12 endif
13 return y
```

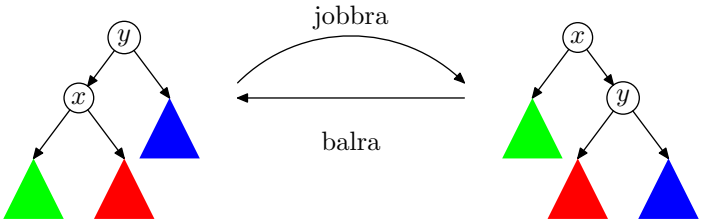
műveletek. Ennek megfelelően a naív beszúrás és törlés műveletét úgy módosítjuk, hogy közel teljes fát kapjunk. Ezt úgy tesszük meg, hogy a kiegyensúlyozott fákat használunk, azaz a fa bal- és jobboldali ágai közel egyforma hosszúak.

Az egyik ilyen módszer a piros-fekete fák használata. Itt a fa minden egyes csúcsát pirosra vagy feketére festjük. Tehát a fa minden egyes csúcsa a következő információkat tartalmazza: `szín`, `kulcs`, `bal`, `jobb`, `apa`. Ha nem létezik a mutatott fiú, vagy apa, a megszokott Nil jelölést használjuk. A Nil-lel jelölt fiúkat, (más elnevezéssel a külső csúcsokat) most levélnek tekintjük, míg a belső csúcsok a kulccsal rendelkező pontok.

5.1. Piros-fekete tulajdonság

- Minden csúcs piros vagy fekete.
- Minden levél fekete.
- Minden piros pont mindkét fia fekete.
- Bármely két, azonos pontból induló és levélig vezető úton ugyanannyi fekete pont van.

Egy x pont *fekete-magasságának* nevezzük a pontból induló, levélig vezető úton található, x-et nem tartalmazó fekete pontok számát. Egy piros-fekete fa fekete-magasságán a gyökér fekete-magasságát értjük. (A feltételek alapján tetszőleges út ugyanannyi fekete csúcsot tartalmaz, így a fekete magasság jól meghatározott érték.) Bármely n belső pontot tartalmazó piros-fekete fa magassága legfeljebb $2\log_2(n+1)$. Mivel az ágakon ugyanannyi fekete csúcs található, és minden piros csúcsot fekete csúcs követ, így nincs olyan ág, melynek hossza egy másikénak kétszeresét meghaladná.



5.2. Forgatások

A naív beszúrások és törlések gyakran elrontják egy eredetileg piros-fekete fa piros-fekete tulajdonságát. Ezt a csúcsok átszinezésével illetve "forgatásával" állítjuk helyre. Mint az ábrából leolvasható a zöld részfa az x-nél kisebb, a kék részfa az y-nál nagyobb, a piros részfa az x és y közti elemeket tartalmazza. Az alábbiakban látható az előbbi ábra balra forgatásának a programja. A JOBBRA-FORGAT ennek szimmetrikus változata, melyben a `jobb` és `bal` mezőnevek helyet cserélnek.

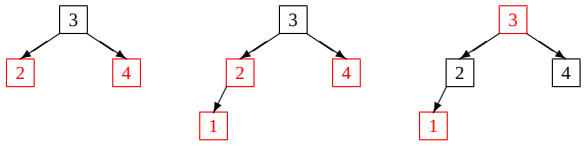
Procedure `BALRA-FORGAT(`*T,x*`)`

Input: A `T` fa, az `x` a piros-fekete tulajdonságot nem teljesítő csúcs
Eredmény: Az `y` gyökerű fában helyreáll a piros-fekete tulajdonság

```
1  y ← x.jobb
2  x.jobb ← y.bal
3  if y.bal ≠ Nil then y.bal.apa ← x
4  y.apa ← x.apa
5  if x.apa = Nil then
6  |  T.gyöker ← y
7  else
8  |  if x = x.apa.bal then x.apa.bal ← y else x.apa.jobb ← y
9  endif
10 y.bal ← x
11 x.apa ← y
```

5.3. Piros-fekete beszúrás

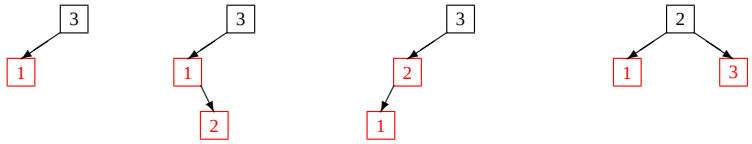
Az algoritmusban szereplő három különböző eset a következő: Az 1. ábra első



1. ábra.

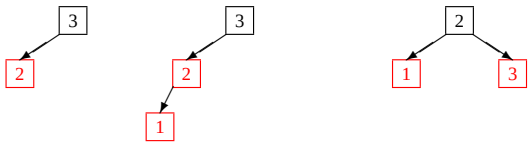
```
Function PF-BESZÚR(T,x)
    Input: A T piros-fekete fa, az x a beszúrandó csúcs
    Eredmény: Az T fába beszúrja az x csúcsot
 1  BESZÚR(T,x)
 2  x.szín ← PIROS
 3  while x ≠ T.gyökér and x.apa.szín = PIROS do
 4      if x.apa = x.apa.apa.bal then
 5          y ← x.apa.apa.jobb
 6          if y.szín = PIROS then
 7              x.apa.szín ← FEKETE
 8              y.szín ← FEKETE
 9              x.apa.apa.szín ← PIROS
10              x ← x.apa.apa
11          else
12              if x = x.apa.jobb then
13                  x ← x.apa
14                  BALRA-FORGAT(T,x)
15              endif
16              x.apa.szín ← FEKETE
17              x.apa.apa.szín ← PIROS
18              JOBBRA-FORGAT(T,x.apa.apa)
19          endif
20      else
21          ugyanaz, csak az irányok felcserélődnek
22      endif
23
24  endw
25  T.gyökér.szín ← FEKETE
```

fájába beszúrva az 1 elemet a piros-fekete tulajdonság sérül, ezért a most szinezünk (7-10. sor). A 2. ábra első fájába beszúrva a 2 elemet, a piros-fekete tulajdonság



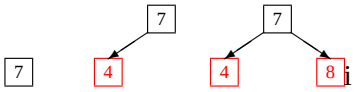
2. ábra.

megint sérül, de átszinezés most nem elegendő. Ezért egy balra- majd egy jobbraforgatással lehet helyreállítani a fát (13-18. sor). A 3. ábra első fájába beszúrva az 1 elemet, az átszinezés most sem elegendő, viszont egy jobbraforgatással helyre lehet állítani a fát (16-18. sor).

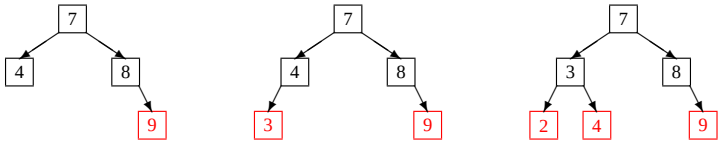


3. ábra.

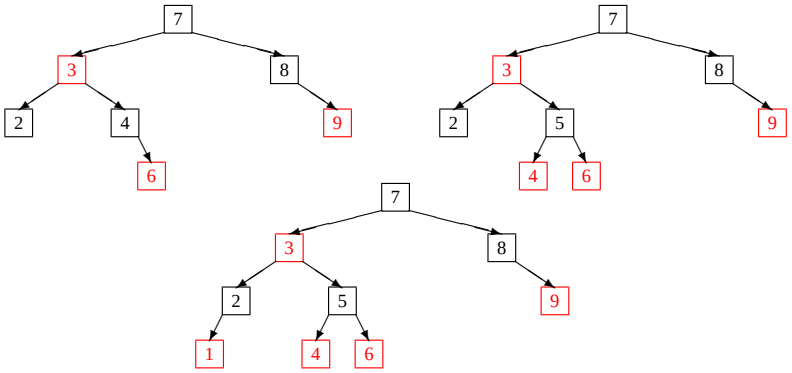
Példa. És most készítsünk egy fát a 7, 4, 8, 9, 3, 2, 6, 5 és 1 elemek sorozatos piros-fekete beszúrásával!



4. ábra. A 7, 4 és 8 beszúrása



5. ábra. A 9, 3 és 2 beszúrása



6. ábra. A 6, 5 és 1 beszúrása

5.4. Piros-fekete törlés

A piros-fekete törlés is a naív törlésre épül. Ha piros csúcsot töröltünk, akkor a fekete-magasságok nem változnak, tehát minden rendben van, nincs további műveletekre szükség. Ha fekete csúcsot töröltünk, akkor újra színezéssel és forgatással állítjuk helyre a piros-fekete tulajdonságot (PF-TÖRÖL-JAVÍT). Most a külső

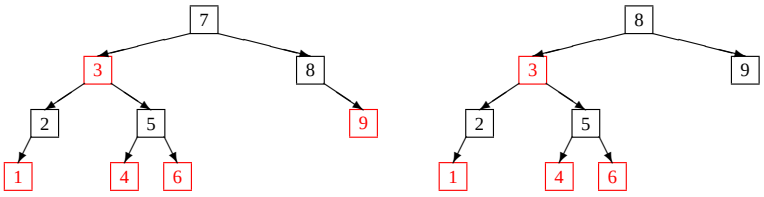
csúcsok is rendelkeznek a belső csúcsok mezőivel (szín, apa, stb.), hogy az algoritmus egyszerűbb legyen.

Procedure PF-TÖRÖL(*T,z*)

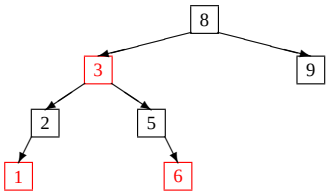
Input: A T piros-fekete fa, az x a törlendő csúcs
Eredmény: Az T fából törli az x csúcsot

```
1  if z.bal = NilT or z.jobb = NilT then y ← z else y ←KÖVETKEZŐ(z)
2  if y.bal ≠ NilT then x ← y.bal else x ← y.jobb
3  x.apa ← y.apa
4  if y.apa = NilT then
5  |  T.gyökér ← x
6  else
7  |  if y = y.apa.bal then y.apa.bal ← x else y.apa.jobb ← x
8  endif
9  if y ≠ z then
10 |  z.kulcs ← y.kulcs  // hasonlóan a többi adatra
11 endif
12 if y.szín = FEKETE then PF-TÖRÖL-JAVÍT(T,x)
13 return y
```

Példa. Az előzőleg felépített fából töröljük az elemeket a beszúrás sorrendjében. A jobb nyomonkövethetőség érdekében a program futása során kapott átmeneti fákat is megadjuk.



7. ábra. A 7 törlése



8. ábra. A 4 törlése

Az interneten több Java animáció is található a piros-fekete fával kapcsolatban.A www.ece.uc.edu címen található programban a törlés nem az előbb ismertetett algoritmussal lett megoldva, ezért ajánlom kevésbé látványos programot, amely a pf-fa.htm állományban található.

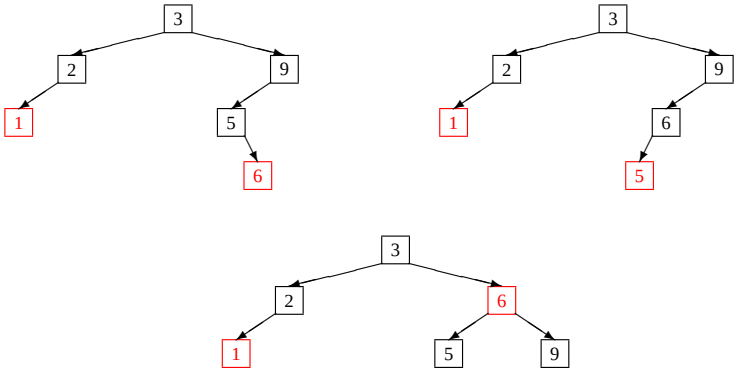
Procedure PF-TÖRÖL-JAVÍT(*T,x*)
Input: A T piros-fekete fa, az x az a csúcs, ahol sérül a piros-fekete tulajdonság
Eredmény: Az T fában helyreáll a piros-fekete tulajdonság

```
1  while x ≠ T.gyökér and x.szín = FEKETE do
2      if x = x.apa.bal then
3          w ← x.apa.jobb
4          if w.szín = PIROS then
5              w.szín ← FEKETE
6              x.apa.szín ← PIROS
7              BALRA-FORGAT(T,x.apa)
8              w ← x.apa.jobb
9          endif
10         if w.bal.szín = FEKETE and w.jobb.szín = FEKETE then
11             w.szín ← PIROS
12             x ← apa.x
13         else
14             if w.jobb.szín = FEKETE then
15                 w.bal.szín ← FEKETE
16                 w.szín ← PIROS
17                 JOBBRA-FORGAT(T,w)
18                 w ← x.apa.jobb
19             endif
20             w.szín ← x.apa.szín
21             x.apa.szín ← FEKETE
22             w.jobb.szín ← FEKETE
23             BALRA-FORGAT(T,x.apa)
24             x ← T.gyökér
25         endif
26     else
27         // ugyanaz, csak az irányok felcserélődnek
28     endif
29 endw
30 x.szín ← FEKETE
```

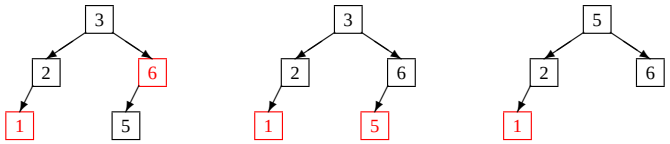
6. AVL-fa

Történetileg a piros-fekete fák kifejlesztését megelőzte az AVL-fák kifejlesztése. Az AVL elnevezés a szerzők neveinek rövidítéséből származik (Adelszon, Velszkij és Landisz). Egy fa magasságán a gyökértől levelekig tartó utak maximális hosszát értjük. Ez induktív definícióval a következőképpen fogalmazható meg:

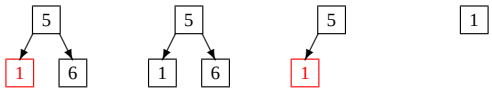
– Üres fa magassága 0.



9. ábra. A 8 törlése



10. ábra. A 9 és 3 törlése



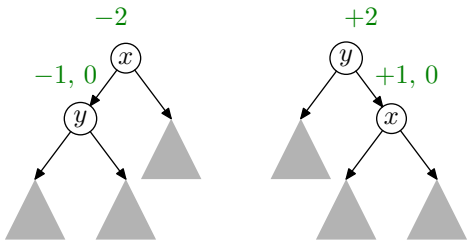
11. ábra. A 6 és 5 törlése

– Az egy gyökérből álló fa magassága 1.
– Az egynél több csúcsból álló fa magassága eggyel nagyobb, mint a magasabb részfájának magassága.

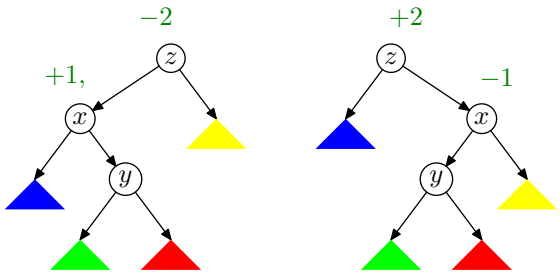
AVL-fának nevezünk minden olyan bináris keresőfát, melyben a bal- és jobboldali részfák magasságának különbsége abszolút értékben egyetlen csúcsnál sem haladja meg az 1-et.

6.1. Forgatások

Az AVL-beszúrás és AVL-törlés ugyancsak a naív beszúrásra és naív törlésre épít. A piros-fekete fákhoz hasonlóan itt is forgatással lehet az AVL-tulajdonságot helyreállítani. Attól függően, hogy hol és mennyire sérül az AVL tulajdonság, négy különböző forgatást kell használni. Az alábbi ábrákon található zöld számok az adott csúcs jobboldali részfája magasságának és baloldali részfája magasságának különbsége. A 12. ábrán az első esetben jobbraforgatást, a második esetben balraforgatást kell alkalmazni, hogy az AVL tulajdonság helyreálljon. A 13. ábrán az első esetben ez egy x körüli balraforgatással, majd egy z körüli jobbraforgatással,

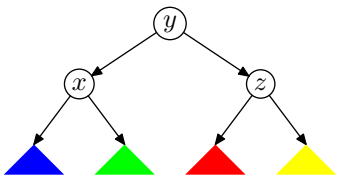


12. ábra. Egyszeres forgatások



13. ábra. Kétszeres forgatások

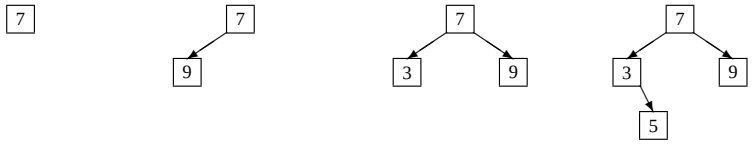
míg a második esetben ez egy z körüli jobbraforgatással, majd egy x körüli balraforgatással érhető el. Mindkét esetben a végeredmény a 14. ábrán látható. Mind a



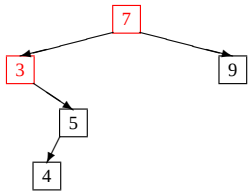
14. ábra. Kétszeres forgatások végeredménye

beszúrásnál, mind a törlésnél a beszúrt, illetve törölt csúcstól a gyökér fele vezető úton található csúcsokra ellenőrizni kell az AVL-tulajdonság meglétét. Az AVL fák algoritmusának kódja igen hosszú, ezért itt nem részletezzük. Az interneten majd minden programnyelvre megtalálható valamely megvalósítása.

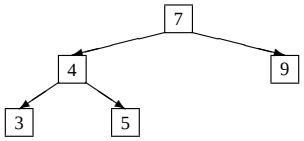
Példa. Alábbiakban az látható, hogy hogyan szúrhatjuk be egy üres AVL-fába sorban a 7, 9, 3, 5, 4, 2, 1, 6 és 8 számokat, majd sorra ugyanezeket töröljük is. A pirossal jelölt mezők jelzik azt, ahol az AVL-tulajdonság nem áll fenn. A 16. ábrán a 3-at és a 7-et tartalmazó csúcsok miatt nem AVL-fa a fa. Az 5-öt tartalmazó csúcsnál ez az érték az irányítást is figyelembe véve -1, így 3-at tartalmazó csúcs gyökerű fára kell alkalmazni a kétszeres forgatást. A 2 beszúrásával az AVL tulajdonság újra sérül (16. ábra), s mivel a négyet tartalmazó csúcshoz tartozó különbség -1, most egyszeres forgatásra lesz szükség (19. ábra).



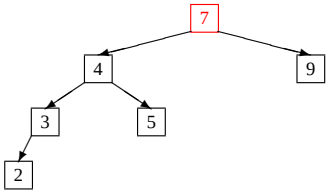
15. ábra. A 7, 9, 3 és 5 beszúrása



16. ábra. 4 naív beszúrásának eredménye



17. ábra. A forgatás után helyreáll az AVL-tulajdonság

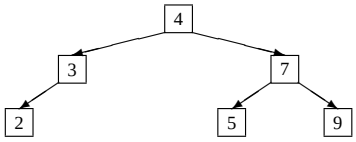


18. ábra. A 2 naív beszúrásával ismét sérül az AVL-tulajdonság

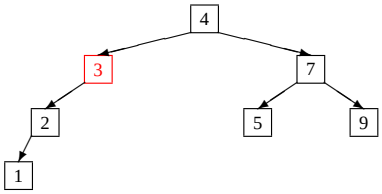
7. B-fa

A kiegyensúlyozott fák egy igen gyakran alkalmazott osztálya a Bayer-fa, röviden B-fa. A B-fa egyébként J. Hopcroft 2-3 fájának általánosítása. A B-fa nem bináris fa, egy n-edrendű B-fa minden belső csúcsa (az itt használatos terminológiában lap) maximum $2n + 1$ mutatót és maximum $2n$ kulcsot tartalmazhat. Az adatok a fa külső csúcsaiban, a levelekben helyezkednek el. Minden belső csúcs a gyökeret kivéve legalább n kulcsot és $n + 1$ mutatót tartalmaz. A kulcsok egy lapon nagyság szerint rendezettek, és két szomszédos kulcs közötti mutató által jelölt részfa minden kulcsának értéke e két kulcs közé esik. (A soron következő ábrákon az egyszerűség kedvéért csak a belső csúcsokat jelöljük.)

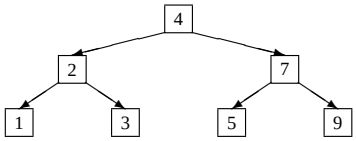
B-fa műveletek



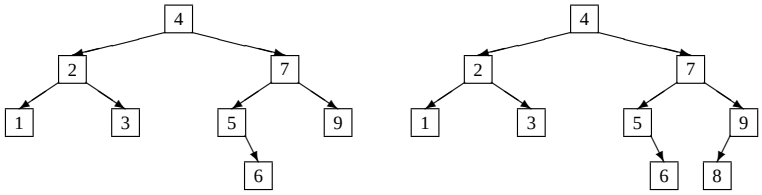
19. ábra. Forgatással megint helyreállítható



20. ábra. Az 1 naív beszúrásával ismét sérül az AVL-tulajdonság



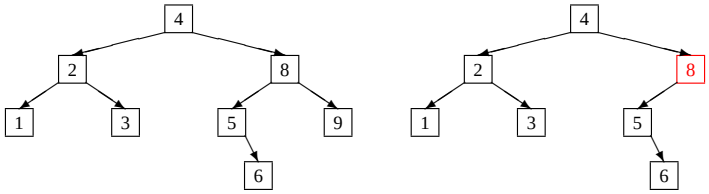
21. ábra. Megint egy egyszeres forgatásra van szükség.



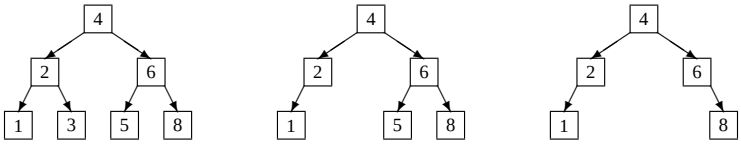
22. ábra. 6 és 8 beszúrása.

Keresésnél: ha a keresett kulcs
- a belső csúcsban található első kulcsnál kisebb, akkor az első mutató által jelölt részfában kell folytatni a kutatást.
- az utolsó, m-dik kulcsnál nagyobb, akkor az $m+1$-dik mutató által jelölt részfában kell folytatni a kutatást.
- az i-dik és $i+1$-dik kulcsok közé esik (vagy egyenlő a másodikkal), akkor az $i+1$-dik mutató által jelölt részfában kell folytatni a kutatást.
- Ha valamely mutató Nil, akkor a keresett elem nem szerepel a fában.

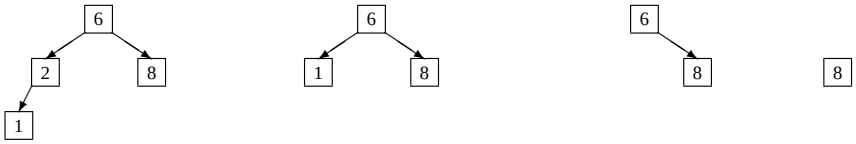
Beszúrásnál: először megkeressük a beszúrandó elem helyét, a megfelelő levelet, azaz a külső csúcsokat közvetlenül megelőző belső csúcsot.



23. ábra. A 9 törlése és a helyreállítás



24. ábra. 3 és 5 törlése



25. ábra. 2, 1 és 6 törlése

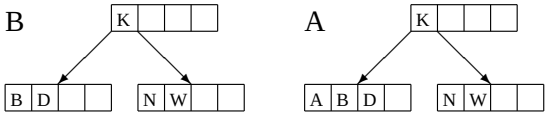
- Ha ebben belső csúcsban még nincs $2n$ kulcs, akkor nagyság szerint beszúrjuk a kulcsot, és a kulcsot követő mutatót a beszúrt elemre állítjuk.
- Ha az adott belső csúcs megtelt, de az egyik szomszédos levélen még van üres hely, akkor az első vagy utolsó kulcsot átmozgatjuk az apacsúcsra, míg az ott levő megfelelő kulcsot pedig a szomszédos levélre visszük.
- Ha mind az adott csúcs, és a két szomszédos csúcs is megtelt, akkor a létrehozunk egy új levelet, az ideeső $2n + 1$ értékből az utolsó n-et átvisszük ide, az $n + 1$-diket pedig az apacsúcsba szúrjuk be.

Törlésnél: ha nem levélből törlünk, akkor a kulcsot a szabványnak megfelelően, a nála kisebb kulcsok közül a maximálissal cseréljük ki, s azt a kulcsot töröljük valamely levélből. Levélből törlés esetén, ha n-nél kevesebb kulcs marad a levélben, a szomszédos levelekből kell kiegészíteni egy kulccsal. Ha azok mindegyikében csak n kulcs található, akkor a két szomszédos csúcsot össze kell vonni, és kiegészíteni az apacsúcsból egy elemmel. Ezek után meg kell vizsgálni az apaelemet is.

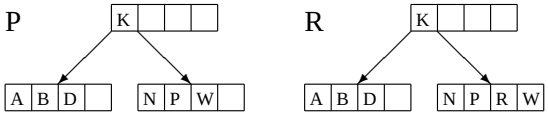
A [1] könyvben megtalálható a beszúrás programja pszeudónyelven, de a törlés programja már innen is kimaradt a mérete miatt. Épp ezért itt mi nem szerepeltetjük a programot, csak egy hosszabb példát beszúrásra és törlésre.

Példa a B-fába beszúrásra Sorra szúrjuk be egy B-fába sorra a N, D, W, K, B, A, P, R, X, G, Q, L, V, C, E, T, Y, U, F, Z, O kulcsokat, majd ugyanebben a sorrendben töröljük is a következő példában. A 26. ábrán a K beszúrása után betelt a 4 hely, ezért a következő elem beszúrásakor két új tárolót nyitunk. A középső érték marad az eredetiben, a többit szétdobáljuk. Az X nem fér be az jobboldali levélben, De a

26. ábra. N, D, W, K beszúrása

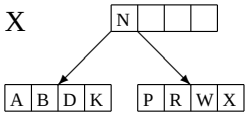


27. ábra. B és A beszúrása

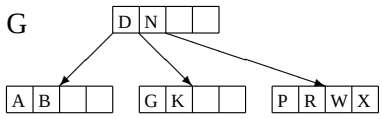


28. ábra. P és R beszúrása

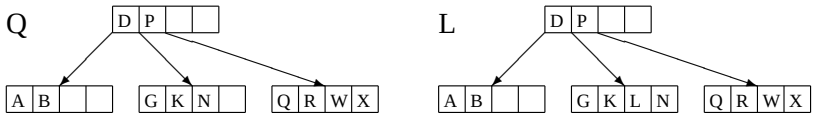
baloldali levélben még van üres hely, ezért forgatunk (29. ábra). (Más megvalósításokban a forgatás helyett újabb tárolókat alkalmaznak, tehát más könyvek, vagy más megvalósítások nem biztos, hogy ugyanezeket a fákat generálják.)



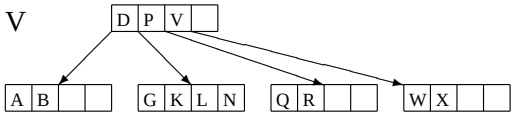
29. ábra. X beszúrása átforgatással



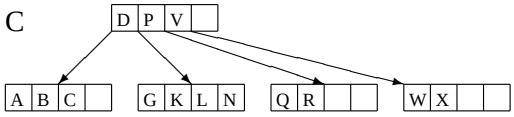
30. ábra. G beszúrása új levél nyitásával



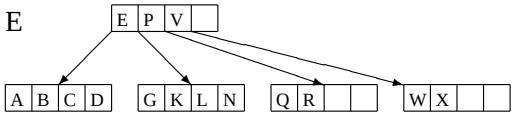
31. ábra. Q és L beszúrása



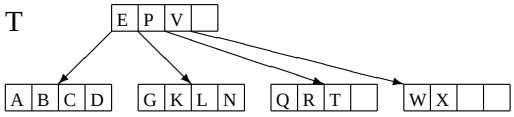
32. ábra. V beszúrása új levél nyitásával



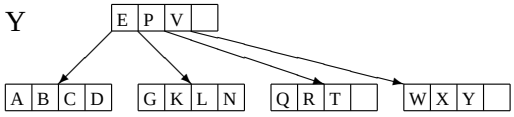
33. ábra. C beszúrása



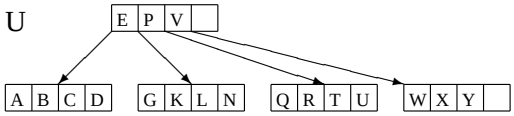
34. ábra. E beszúrása forgatással



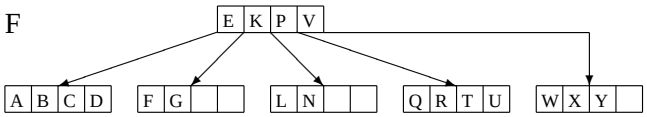
35. ábra. T beszúrása



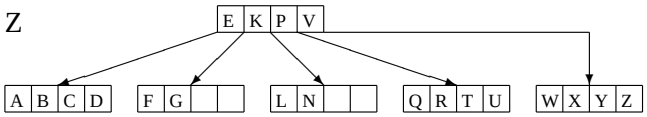
36. ábra. Y beszúrása



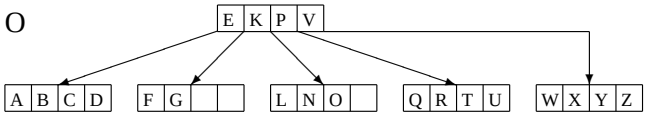
37. ábra. U beszúrása



38. ábra. F beszúrása új levél nyitásával

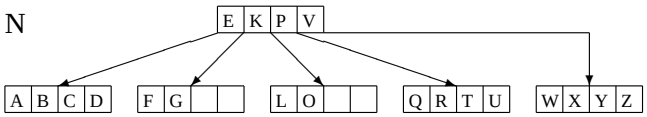


39. ábra. Z beszúrása

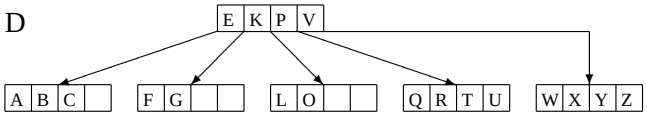


40. ábra. O beszúrása

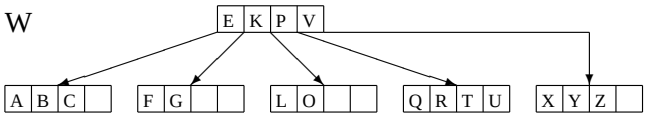
Példa törlésre Hogy a K kulcsot töröljük a felső tárolóból, helyére fel kell húzni a



41. ábra. N törlése

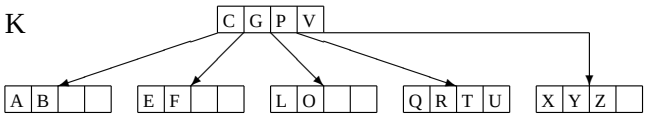


42. ábra. D törlése



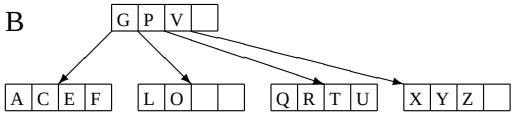
43. ábra. W törlése

nála kisebb kulcsokból a maximálist, a G-t. Viszont a megfelelő tárolóban legalább két elemnek lennie kell, ezért valamely szomszédból pótoljuk a hiányt, azaz átforgatunk (44. ábra). A B törlésekor az első tárolóban már nem marad két érték,

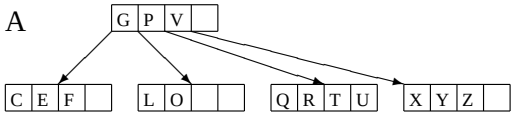


44. ábra. K törlése forgatással az első levélből

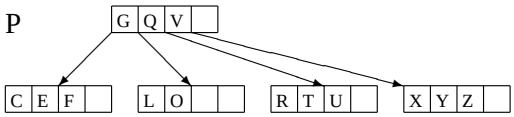
a szomszédból sem lehet kölcsönkérni, ezért a két első tárolót összevonjuk, és a hozzájuk csapjuk a felső tároló első elemét (45. ábra). A P törlésekor az O-t fel kell húzni, de a második tárolóban ezzel kevés kulcs marad, ezért a harmadikból forgatunk át egy elemet (47. ábra). A Q törlésekor a lenti tárolókban már kevés kulcs maradt, ezért a második és harmadik tárolót összevonjuk (51. ábra).



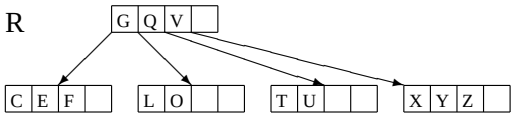
45. ábra. B törlése levelek összevonásával



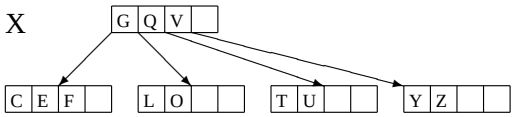
46. ábra. A törlése



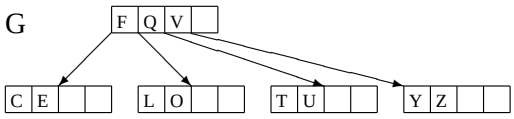
47. ábra. P törlése Q felhuzásával és Q átforgatásával



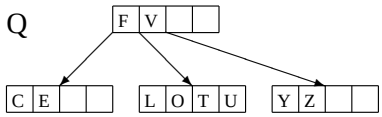
48. ábra. R törlése



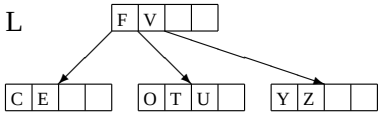
49. ábra. X törlése



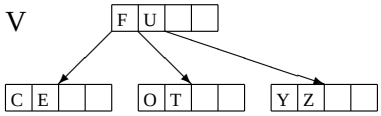
50. ábra. A G törlésekor az őt megelőző kulccsal helyettesítjük



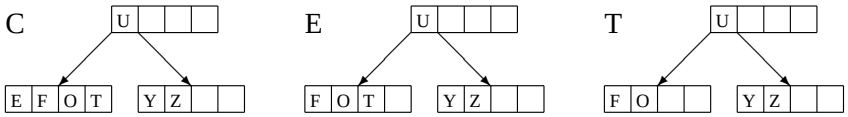
51. ábra. Q törlése levelek összevonásával



52. ábra. L törlése



53. ábra. V törlése

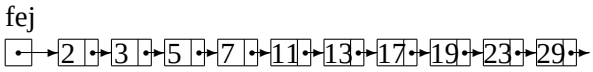


54. ábra. C, E és T törlése

55. ábra. Y törlése levelek összevonásával jár

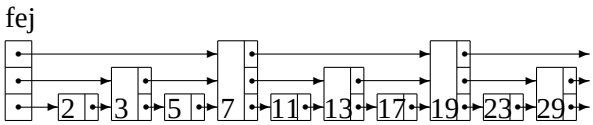
8. Ugrólisták

Az n-elemű láncolt listákban legfeljebb n elemet kell megvizsgálni ahhoz, hogy eldöntsük, hogy egy adott kulcs szerepel-e a listában, vagy sem (56. ábra). Ha



56. ábra. Láncolt lista

minden második listaelem tartalmaz egy plusz mutatót a nála kettővel későbbi listaelemre, akkor legfeljebb $\frac{n}{2}+1$ elemet kell megvizsgálni. Újabb és újabb mutatók beszúrásával (minden negyedik elemhez a nála néggyel későbbire mutató, stb.) a vizsgálatok száma $\ln(n)$-nel arányosra csökkenthető (57. ábra). Ebben az



57. ábra. Láncolt lista extra mutatókkal

adatstruktúrában gyorsan kereshetünk, de a beszúrás és törlés az adatstuktúra átszervezését igényelné. A listaelemet, ha k mutató tartozik hozzá, k-*adrendűnek* nevezzük. Az előbbi adatszerkezetben 1-szintű az elemek 50 százaléka, 2-szintű az elemek 25 százaléka, stb. Az ugrólistáknál megtartjuk ezeket a valószínűségeket, a beszúrandó elem szintjét véletlenszerűen adjuk meg, a többi elem szintjét nem változtatjuk meg a beszúrás és törlés során, azaz csak lokális változások történnek. Ennek megfelelően az i-dik mutató sem a 2^{i-1}-dik következő elemre mutat, hanem a sorban következő legalább i-szintű elemre. Miután a magasabb szintű mutatókat használva kihagyunk, átugrunk bizonyos elemeket, ezt az adatszerkezetet ugrólistának nevezzük. (W. Pugh 1990-es cikkében a skip-list elnevezést használta.)

8.1. Keresés ugrólistában

Keresésnél a magasabb szintű mutatóknál kell elkezdeni a keresést. Ha a soron következő mutató követésével túllépnénk a keresett elemen, akkor eggyel alacsonyabb szintű mutatókat vizsgálunk. Így végül a keresett elem előtt állunk meg, feltéve ha az az ugrólistában szerepel. Ha a 16-ot keressük az 57. vagy az 58. ábrán látható ugrólistában, akkor a fej harmadszintű mutatójánál kell kezdeni. Ez a 7-et tartalmazó elemre mutat, s mivel a 7 kisebb, mint a 16, így ennél folytatjuk. A 7 harmadrendű mutatója a 19-re mutat, ami már nagyobb a keresett elemnél, ezért egy szintet visszalépünk. A 7 másodszintű mutatója a 13-ra mutat, ez kisebb a

```
Function UGRÓLISTA-KERES(L,k)
  Input: L ugrólista, k keresett kulcs
  Output: A kulcsot tartalmazó listaelem címe, illetve Nil
1 x ← L.fej
2 for i ← L.szint downto 1 do
3 |   while x.köv[i].kulcs < k do x ← x.köv[i]
4 endfor
5 x ← x.köv[1]
6 if x.kulcs = k then return x else return Nil
```

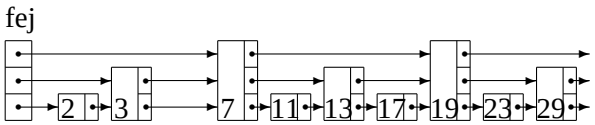
keresett elemnél, így ezzel folytatjuk. A 13 másodszintű mutatója a 19-re mutat, ez nagyobb a keresett elemnél, ezért egy szinttel vissza kell lépni. A 13 elsőszintű mutatója a 17-re mutat, de mivel nincs hova visszalépni, így kilépünk a for-ciklusból. A 13-at követő elem kulcsa nem 16, így a rutin Nil értéket ad vissza, jelezve azt, hogy a keresett kulcs nincs benne az ugrólistában.

8.2. Beszúrás ugrólistába

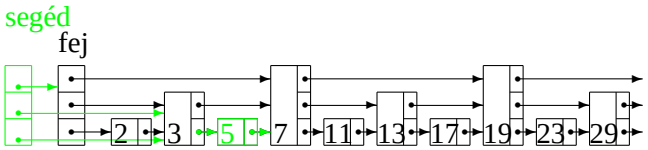
```
Procedure UGRÓLISTA-BESZÚR(L,z)
  Input: L ugrólista, z beszúrandó elem
  Eredmény: Az ugrólistába beszúrja az adott elemet
1 x ← L.fej
2 for i ← L.szint downto 1 do
3 |   while x.köv[i].kulcs < z.kulcs do x ← x.köv[i]
4 |   segéd[i] ← x
5 endfor
6 x ← x.köv[1]
7 if x.kulcs = z.kulcs then
8 |   x.érték = z.érték
9 else
10|   for i ← 1 to z.szint do
11|   |   z.köv[i] = segéd[i].köv[i]
12|   |   segéd[i].köv[i] ← z
13|   endfor
14 endif
```

Beszúrásnál először is megkeressük a beszúrandó elem helyét. Ez ugyanúgy megy mint a keresésnél, csupán a beszúrandó elemet a különböző szinteken megelőző elemeket feljegyezzük egy segéd mutatótömben. Miután megtaláltuk az adott elem helyét, a következő fejezetben szereplő láncolt listás beszúráshoz hasonlóan járunk el az adott elem minden szintje esetén, azaz ha a z elem egy adott szinten az x és y közé esik, ahol az y az x követője, akkor mostantól az x-et a z követi, míg a z-t

az y. Esetünkben az 59. ábrán látható ugrólistába szeretnénk beszúrni az 5 kulcsú egyszintű elemet. Ezért bennünket csak az érdekel, hogy az első szinten mit követ az 5 kulcsú elem. Ez a 3 kulcsú elem, melynek a követője a 7 kulcsú, így mostantól 3-at az 5, az 5-öt a 7 követi.



58. ábra. Az eredeti ugrólista



59. ábra. Az 5 beszúrása az ugrólistába

8.3. Törlés ugrólistából

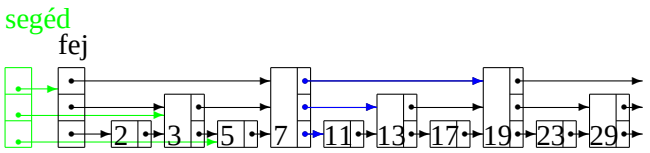
Procedure UGRÓLISTA-TÖRÖL(L,k)

Input: L ugrólista, k kulcs
Eredmény: A listából törli a megadott kulcsú elemet, ha az szerepel benne

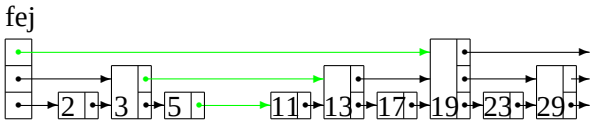
```
1  x ← L.fej
2  for i ← L.szint downto 1 do
3  |  while x.köv[i].kulcs < k do x ← x.köv[i]
4  |  segéd[i] ← x
5  endfor
6  x ← x.köv[1]
7  if x.kulcs = k then
8  |  for i ← 1 to z.szint do
9  |  |  segéd[i].köv[i] ← x.köv[i]
10 |  endfor
11 endif
```

Törlésnél nem a törlendő elem címét adjuk meg mint korábban, hanem a kulcsát. Ennek megfelelően a beszúráshoz hasonlóan itt is meg kell keresni az adott elemet, s itt is feljegyezzük a törlendő elemet a különböző szinteken megelőző elemeket. Ha a z elem kulcsa a k, és egy adott szinten az x előzi meg a z-t, és y követi,

akkor a törlésnél az x következője y lesz. A 60. és 61. ábrákon az látható, hogy a példánkban szereplő ugrólistából hogyan törölhető a 7 kulcsú elem. Az első ciklus végrehajtása során az derül ki, hogy ezt az elemet sorra a fej, a 3 és 5 kulcsú elem előzi meg. A 7 törléséhez a következőket kell beállítani: a fejet harmadszinten a 19, a 3 kulcsú elemet a másodszinten a 13 kulcsú, az 5 kulcsú elemet az elsőszinten a 11 kulcsú elem követi.



60. ábra. Az 7 törléséhez feljegyezzük az őt megelőző elemeket.



61. ábra. A 7 tényleges törlése

VI. fejezet
Elemi adatszerkezetek

Az informatikában nagyon gyakran használt adatszerkezet a verem és a sor. Eme dinamikus adatszerkezetek esetén a lekérdező műveleteket rendszerint nem használjuk, illetve csupán az teszteljük, hogy az adott adatszerkezet üres-e vagy sem. A módosító műveleteknél bizonyos korlátozások érvényesülnek, pontosabban a veremnél a legutoljára beszúrt elemet törölhetjük elsőként, míg a sornál a legelőször beszúrt elemet. Épp ezért a TÖRÖL utasításnál nem kell megadni a törlendő elemet, vagy annak kulcsát. Emiatt használják ezen adatszerkezetek megnevezésére a LIFO (last in-first out, azaz utolsóként be, elsőként ki) illetve a LILO (last in-last out, azaz utolsóként be, utolsóként ki) rövidítéseket. A sor adatszerkezetet jól jellemzi a keskeny alagút, ahol az alagútba először behajtó kocsi hagyja el azt elsőként. Míg a zsákutcába behajtó kocsik közül az utolsónak kell kitolatnia elsőként, ez pedig a verem jellemzője.

1. Verem

Mind a verem, mind a sor adatszerkezet ábrázolható tömbökkel. Tekintsük az S tömböt! Jelölje az *S.tető* a legutoljára berakott elem indexét, így az $S[1..\text{S.tető}]$ tömbrész tartalmazza a verem aktuális értékét. Ha *S.tető*$= 0$, akkor veremben nincs elem, azaz a verem *üres*. Ha üres veremből próbálunk törölni, az *alulcsordulás*nak nevezzük, míg ha egy $S.hossz + 1$-dik elemet próbálunk beszúrni, akkor *túlcsordulásról* beszélünk. Az egyetlen lekérdező művelettel azt állapíthatjuk meg, hogy üres-e a verem, vagy sem. Az angol szakirodalomban Push és Pop néven

Function ÜRES(S)
Input: S verem
Eredmény: IGAZ, ha üres a verem, és HAMIS, ha nem
1 **if** $S.tető = 0$ **then return** IGAZ **else return** HAMIS

nevezett beszúró és törlő műveletekre mi a VEREMBE és VEREMBŐL nevekkel hivatkozunk. Mind a lekérdezés, mind beszúrás és törlés is $O(1)$ bonyolultságú.

Procedure VEREMBE(S,x)
Input: S verem, x beszúrandó elem
Eredmény: Az adott elemet beszúrja verem tetejére
1 **if** $S.tető < S.hossz$ **then**
2 S.tető $= S$.tető $+ 1$
3 $S[S$.tető$] \leftarrow x$
4 **else**
5 **hiba** "túlcsordulás"
6 **endif**

Function VEREMBŐL(S)
Input: S verem
Output: A verem felső eleme, melyet egyben töröl is a veremből
1 **if** *ÜRES*(S) **then**
2 **hiba** "alulcsordulás"
3 **endif**
4 S.tető $\leftarrow S$.tető $- 1$
5 **return** $S[S$.tető $+ 1]$

2. Sor

Míg a veremnél egy adat meghatározza a verem méretét, a sornál kettőre van szükségünk. A sor *feje* a sor első elemét jelenti, míg a sor *vége* a sor utolsó elemét. Ha egy új elemet veszünk fel a sorba, akkor azt a sor végére írjuk, ahogy a pénztárnál is a sor végére állunk. Miután az első vevő fizetett a pénztárnál, elhagyja a sort. Hasonlóan a sor adattípusból is az első elemet, a sor fejét töröljük. Az alábbi programokban a Q listához tartozó *Q.fej* elem a sor első elemének indexe a $Q[1..Q.hossz]$ tömbben. A *Q.vége* a sor utolsó eleme mögötti index, erre a helyre kerül a soron következő beszúrandó elem. A sor esetén is van egy lekérdező művelet, ez is arra ad választ, hogy az adatszerkezet üres-e vagy sem. A lista mó-

Function ÜRES(Q)
Input: Q sor
Output: IGAZ, ha üres a sor, és HAMIS, ha nem
1 **if** $Q.fej = Q.vége$ **then return** IGAZ **else return** HAMIS

dosító műveletei hasonlóan a beszúrás (SORBA) és törlés (SORBÓL). Ha a lista üres, és így akarunk belőle törölni, akkor alulcsordul a sor. Ha pedig a megtelt listába akarunk még újabb elemeket szúrni, túlcsordul a sor. Ha hagyományos módon kezelnénk a tömböt, akkor a beszúrás és a törlés során a után mind a sor feje, mind a sor vége egyre hátrább és hátrább kerülne, s egy idő után bármilyen nagy tömb végét eléri mindkét index. Épp ezért a következő trükköt használjuk: kapcsoljuk össze a tömb végét és elejét, azaz készítsünk belőle egy gyűrűt.

Procedure SORBA(*Q,x*)

Input: Q sor, x beszúrandó elem
Eredmény: A sor végére beszúrja az adott elemet

```
1 if Q.vége = Q.hossz then y ← 1 else y ← Q.vége + 1
2 if Q.fej = y then
3 |   hiba "Megtelt a sor!"
4 else
5 |   Q[Q.vége] ← x
6 |   Q.vége ← y
7 endif
```

Function SORBÓL(*Q*)

Input: Q sor
Output: A sor első eleme, amelyet egyben töröl is a sorból

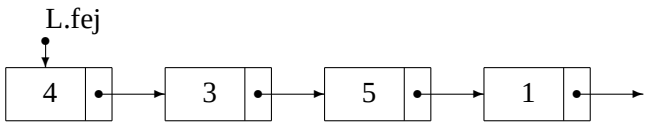
```
1 x ← Q[Q.fej]
2 if Q.fej = Q.hossz then Q.fej ← 1 else Q.fej ← Q.fej + 1
3 return x
```

Példa. Hajtsuk végre párhuzamosan ugyanazokat a beszúró és törlő műveleteket egy soron és egy vermen!

	Sor			Verem		
Szúrjuk be az 5 értéket:	5	_	_	5	_	_
Szúrjuk be a 3 értéket:	5	3	_	5	3	_
Töröljünk egy értéket:	_	3	_	5	_	_
Szúrjuk be a 4 értéket:	_	3	4	5	4	_
Töröljünk egy értéket:	_	_	4	5	_	_
Szúrjuk be az 1 értéket:	1	_	4	5	1	_

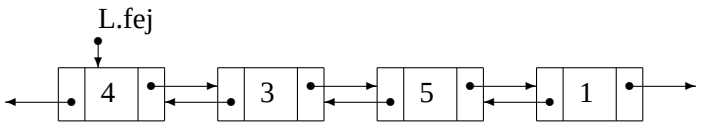
3. Láncolt lista

Míg a tömbök esetén a tömbindexek határozzák meg a lineáris sorrendet, a listákban ezt a mutatókkal érjük el. Listák esetén a lista fejének, azaz első elemének megadására van szükségünk. Az L lista fejét L.fej-jel jelöljük. Egyszerűbb esetben minden rekord (összefüggő adatok csoportja) tartalmaz egy mutatót a soron következő rekordra, ezt egyszeresen láncolt listának nevezzük (1. ábra). Ha min-



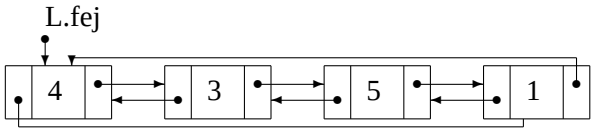
1. ábra. Egyszeresen láncolt lista

den rekord tartalmaz egy mutatót az őt megelőző rekordra is, akkor kétszeresen láncolt listáról beszélünk (2. ábra). Az x elemet követő illetve az x elemet megelőző



2. ábra. Kétszeresen láncolt lista

elemre *x.köv* és *x.előző* jelöléssel hivatkozunk. Ha a lista első és utolsó elemét összekapcsoljuk, ciklikus listát kapunk (3. ábra). Ha a soron következő elemek



3. ábra. Kétszeresen láncolt, ciklikus lista

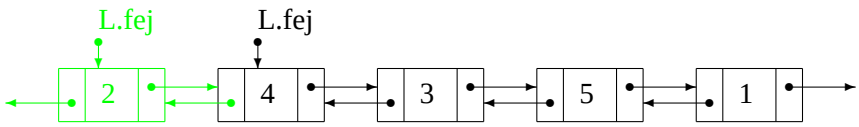
nagyság szerint növekszenek, akkor rendezett listáról beszélhetünk. A következőkben nem rendezett, kétszeresen láncolt rekordokkal foglalkozunk. A veremtől és sortól eltérően a láncolt listák esetén a többi lekérdező műveletet is használhatjuk. Itt most csupán a keresést mutatjuk be. Ha n hosszú listában keresünk, akkor

Function LISTÁBAN-KERES(L,k)

Input: L lista, k keresett kulcs
Output: A keresett elem címe, illetve Nil

```
1 x ← L.fej
2 while x ≠ Nil and x.kulcs ≠ k do x ← x.köv
3 return x
```

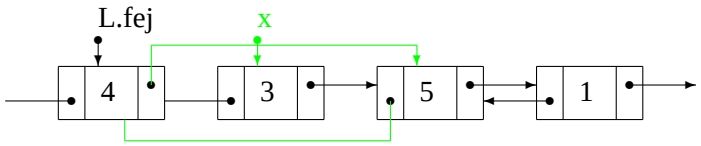
lehet, hogy minden elemet meg kell vizsgálnunk, tehát a keresés bonyolultsága lineáris, azaz $O(n)$. A listafejbe szúrás konstans, azaz $O(1)$ bonyolultságú, mivel csupán mutató-átállításra van szükség (4. ábra). Ha a törlésnél ismerjük a törlendő



4. ábra. Listafejbe szúrás

Procedure LISTÁBA-SZÚR(L,x)
Input: L lista, x beszúrandó elem
Eredmény: Az új lista feje az x elem, a farka a régi lista
1 x.köv $\leftarrow L.fej$
2 **if** $L.fej \neq$ *Nil* **then** $L.fej$.előző $\leftarrow x$
3 $L.fej \leftarrow x$
4 x.előző $\leftarrow$ Nil

Procedure LISTÁBÓL-TÖRÖL(L,x)
Input: L lista, x törlendő listaelem címe
Eredmény: A listából törli a megadott elemet
1 **if** x.*előző* $\neq$ *Nil* **then** x.előző.köv $\leftarrow x$.köv **else** $L.fej \leftarrow x$.köv
2 **if** x.*köv* $\neq$ *Nil* **then** x.köv.előző $\leftarrow x$.előző



5. ábra. Listából törlés

elem címét, akkor a törlés konstans, azaz $O(1)$ bonyolultságú. Ha csak a törlendő elem kulcsát ismerjük, akkor végre kell hajtani a LISTÁBAN-KERES eljárást törlés előtt, s így az együttes bonyolultság lineáris.

VII. fejezet
Hasító táblázatok

Sok gyakorlati feladatban csupán a beszúrás, törlés és a keresés műveleteit kell végrehajtani. Amint azt az előző feladatban láttuk, láncolt listák esetén ezeknek a műveleteknek a bonyolulsága lineáris. (Itt beszúrásnál nem a listafejbe beszúrásról, hanem a rendezett listába szúrásról van szó, ahol a konstans bonyolultságú beszúrást egy keresés előzi meg.) A hasító táblázatok alkalmazásával ez a bonyolultság közel konstanssá redukálható.

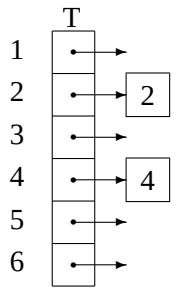
1. Közvetlen címzés

Tegyük fel, hogy az adataink kulcsai az 1 és m közötti számok közül kerülnek ki, és m viszonylag kicsi. Ekkor készíthetünk egy $T[1..m]$ táblázatot, ahol a táblázat elemei a megfelelő kulcsú adatra mutatnak, ha azok elemei a dinamikus halmaznak, egyébként Nil értékek (1. ábra). Ekkor a beszúr, töröl és keres műveletek a következőképpen néznek ki:

Function KÖZVETLEN-CÍMZÉS-KERESÉS(*T,k*)
Input: T táblázat, k kulcs
Output: A T táblázat k kulcsú adata
1 **return** T[k]

Procedure KÖZVETLEN-CÍMZÉS-BESZÚRÁS(*T,x*)
Input: T táblázat, x beszúrandó adat
Eredmény: Az adatot beszúrja a táblázatba
1 $T[x.kulcs] \leftarrow x$

Procedure KÖZVETLEN-CÍMZÉS-TÖRLÉS(*T,x*)
Input: T táblázat, x törlendő adat
Eredmény: Az adatot törli a táblázatból
1 $T[x.kulcs] \leftarrow$ Nil



1. ábra. Közvetlen címzés

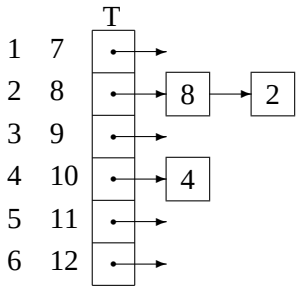
2. Hasító függvény

A kulcsok között gyakran viszonylag nagy számok is előfordulhatnak, de a dinamikus halmaz mérete kicsi (azaz kevés adatot tartalmaz). Ekkor a közvetlen címzés nem gazdaságos, vagy esetleg el sem fér a megfelelő tömb a memóriában. A megoldás ekkor az lehet, hogy a kulcsok értékéből egy hasító függvény segítségével $0,\ldots,m-1$ intervallumba eső számokat kapunk. A függvény több kulcshoz is ugyanazt a számot rendeli, ezt ütközésnek nevezzük. A hasító függvények „cseles" megválasztásával az ütközések esélye csökkenthető, de ki nem zárható. Az ütközések kezelésének egyik módszere, hogy az azonos függvényértékű adatokat egy láncolt listára fűzzük (2. ábra). A megfelelő függvények vázlata a következő:

LÁNCOLT-HASÍTÓ-BESZÚRÁS(T,x)
beszúrás a $T[h(x.kulcs)]$ lista elejére

LÁNCOLT-HASÍTÓ-KERESÉS(T,k)
keresés a $T[h(k)]$ listában

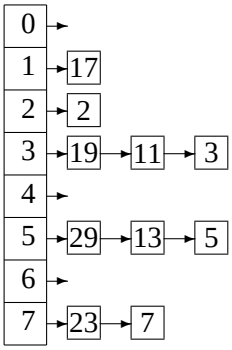
LÁNCOLT-HASÍTÓ-TÖRLÉS(T,x)
törlés a $T[h(x.kulcs)]$ listából



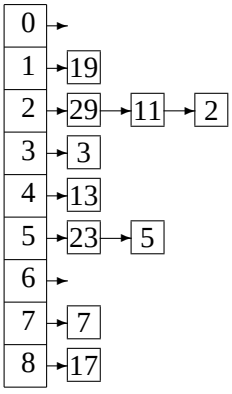
2. ábra. Ütközések kezelése láncolt listával

3. Hasító függvény kiválasztása

Az egyik gyakran használt módszer a $h(k)$ hasítókulcs értékének meghatározására a $k \bmod m$ függvény (osztásos módszer). A 3. és 4. ábrán látható, hogy mi lesz a végeredménye, ha az $m = 8$ illetve $m = 9$ értékekkel, az osztásos módszert használva beszúrjuk a harmincnál kisebb prímszámokat.



3. ábra. $k \mod 8$ hasítófüggvény használata

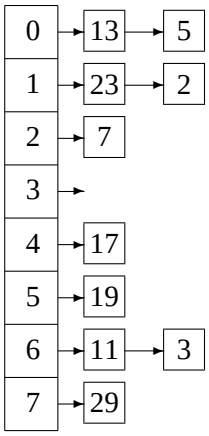


4. ábra. $k \mod 9$ hasítófüggvény használata

Másik gyakran alkalmazott módszer a szorzásos módszer. Ekkor egy 0 és 1 közötti konstanssal (A) megszorozzuk a kulcsot, a kapott szám törtrészét vesszük, majd a kapott értéket megszorozzuk m-mel, s a kapott szám egészrésze a keresett hasító-kulcs, azaz képlettel $h(k) = \lfloor m(kA \bmod 1) \rfloor$. Az 1. táblázatban az aranymetszés arányszámát használjuk az A konstansként, melynek közelítő értéke $0,618$. Az $m = 8$ esetet az 5. ábrán ábrázoltuk.

1. táblázat. Szorzásos módszer $A = 0.618$ esetén

Prímszám	Törtrész	m=8	m=9	m=15
2	0,24	1	2	3
3	0,85	6	7	12
5	0,09	0	0	1
7	0,33	2	2	4
11	0,80	6	7	11
13	0,03	0	0	0
17	0,51	4	4	7
19	0,74	5	6	11
23	0,21	1	1	3
29	0,92	7	8	13



5. ábra. Szorzásos módszer $m = 8$ és $A = 0.618$ esetén

4. Nyílt címzés

Az előbbi esetekben a tömbben csak mutatókat tároltunk, s az adatokat ehhez a tömbhöz láncoltuk. A nyílt címzés esetén a tömbben az adatokat tároljuk. Ebből következik, hogy maximum csak annyi adatot tárolhatunk, amekkora a tömb. (Az előbbi példákban egy nyolc- és kilencelemű tömbhöz kapcsolódóan tíz elemet tároltunk.) Miután az ütközéseket most sem kerülhetjük el, a hasítófüggvény kicsit megváltozik: $h(k)$ helyett $h(k, i)$-t használunk. A függvény megfelelő megválasztása esetén a $\{h(k, 0), \ldots, h(k, m)\} = \{0, \ldots, m - 1\}$. Gyakori választás a

- $h(k, i) = (h'(k) + i) \mod m$ (lineáris kipróbálás),
- $h(k, i) = (h'(k) + ci + di^2) \mod m$, ahol c és d megfelelően kiválasztott konstansok (négyzetes kipróbálás),
- $h(k, i) = (h'(k) + ih''(k)) \mod m$, ahol h' és h'' kisegítő hasítófüggvények (dupla hasítás).

Az alábbi programokban az egyszerűség kedvéért az adat helyett, csak annak kulcsát jelöljük. A beszúró és kereső művelet a következő:

Procedure HASÍTÓ-BESZÚRÁS(*T,k*)

Input: T hasító táblázat, k beszúrandó kulcs
Eredmény: Az adott kulcsot beszúrja a táblázatba

```
1  i ← 0
2  repeat
3  |  j ← h(k, i)
4  |  if T[j] = Nil then
5  |  |  T[j] ← k
6  |  |  return
7  |  else
8  |  |  i ← i + 1
9  |  endif
10 until i = m
11 hiba"A hasító táblázat túlcsordult!"
```

Function HASÍTÓ-KERESÉS(*T,k*)

Input: T hasító táblázat, k a keresett kulcs
Output: A keresett kulcs indexe, illetve Nil

```
1  i ← 0
2  repeat
3  |  j ← h(k, i)
4  |  if T[j] = k then return j
5  |  i ← i + 1
6  until T[j] = Nil or i = m
7  return Nil
```

Az alábbi ábrákon a lineáris kipróbálást használjuk a szorzásos módszerrel összekapcsolva, ahol $m = 15$. A megfelelő adatok az 1. táblázatból leolvashatók. A korábbi prímeket helyezzük el újra. Az első ütközés a 19 beszúrásakor történik (6. ábra). Az $i = 1$ esetén is ütközik (7. ábra), majd csak $i = 2$ esetén sikerül elhelyezni (8. ábra). A 23 beszúrásakor hasonlóan két ütközés történik (9. és 10. ábra), s itt is csak $i = 2$ esetén sikerül elhelyezni (11. ábra). A 29 beszúrásakor is lesz egy ütközés(12. ábra), és csak a $i = 1$ esetén kerül helyre (13. ábra).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	5		2	7			17				11 19	3		

6. ábra. Ütközés a 19 beszúrásakor, $i = 0$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	5		2	7			17				11	3		

19

7. ábra. Ütközés a 19 beszúrásakor, $i = 1$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	5		2	7			17				11	3	19	

8. ábra. A 19 elhelyezhető $i = 2$ esetén

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	5		2	7			17				11	3	19	

23

9. ábra. Ütközés a 23 beszúrásakor, $i = 0$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	5		2	7			17				11	3	19	

23

10. ábra. Ütközés a 23 beszúrásakor, $i = 1$

Hasító törlés nincs, mert ha a példában látható tömbben a 19 kulcsú elemet törölnénk, a 29 kulcsú elemet már nem találná meg a HASÍTÓ-KERESÉS eljárás, mert a ciklusból a $T[j] = \mathsf{Nil}$ feltétel miatt eredmény nélkül kilépne.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	5		2	7	23		17				11	3	19	

11. ábra. A 23 elhelyezhető $i = 2$ esetén

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	5		2	7	23		17				11	3	19 29	

12. ábra. Ütközés a 29 beszúrásakor, $i = 0$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	5		2	7	23		17				11	3	19	29

13. ábra. A 29 elhelyezhető $i = 1$ esetén

VIII. fejezet
Diszjunkt halmazok

Bizonyos feladattípusok esetén szükség van arra, hogy n különálló elemet diszjunkt halmazokba csoportosítsunk. Ekkor a következő két műveletetre lesz szükségünk:

- meg kell mondanunk, hogy két adott elem ugyanabba a diszjunkt halmazba tartozik-e vagy sem;
- két diszjunkt halmazot egyesítenünk kell.

Minden diszjunkt halmazban van egy kijelölt elem, ez a halmaz képviselője (reprezentánsa). Rendezett halmaz esetén ez lehet például a legkisebb elem. Az a fontos, hogy ha a halmaz képviselőjét keressük, minden eleméből kiindulva ugyanahhoz a képviselőhöz jussunk el. Ehhez a következő eljárásokat, függvényeket kell elkészítenünk:

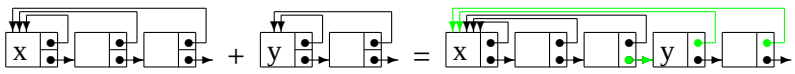
HALMAZT-KÉSZÍT(x)
: egy új halmazt hoz létre, melynek a képviselője x lesz

EGYESÍT(x,y)
: azt a két halmazt, melynek képviselője x ill. y, egyesíti egy új halmazba

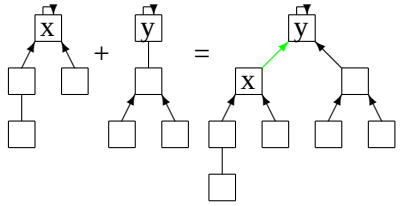
HALMAZT-KERES(x)
: visszadja az x-et tartalmazó halmaz képviselőjét

1. Láncolt listás ábrázolás

A diszjunkt halmazok egy egyszerű ábrázolása az, ahol a halmazokat láncolt listaként ábrázoljuk (1. ábra). Ekkor érdemes minden egyes rekordot kiegészíteni egy mutatóval, amely a halmaz reprezentánsára mutat. Ebben az esetben a mind a HALMAZT-KÉSZÍT, mind a HALMAZT-KERES bonyolultsága O(1). Az EGYESÍT műveletnél abban a listában, melyet a másik lista mögé fűzünk, minden egyes elemnél át kell állítani a képviselőt jelző mutatót. Ezért ésszerűbb a rövidebb listát a hosszabb mögé fűzni. Ezzel az összesen n elemet tartalmazó halmaz esetén m HALMAZT-KÉSZÍT, HALMAZT-KERES, EGYESÍT művelet bonyolultsága $O(m + \ln(n))$.



1. ábra. Láncolt listás ábrázolás



2. ábra. Diszjunkt-halmaz erdő ábrázolás

2. Diszjunkt-halmaz erdők

Egy másik ábrázolásnál egy-egy diszjunkt halmazt egy-egy fával ábrázolunk (2. ábra). A halmaz képviselője a fa gyökerében lévő elem. A HALMAZT-KÉSZÍT művelet ekkor egy gyökérből álló fát hoz létre. A HALMAZT-KERES művelet az apa-mutatókat járja be, s jut el a gyökérig, azaz a fa magasságával arányos bonyolultságú. Az EGYESÍT művelet az egyik fát a másik gyökere alá fűzi be. Ez bizonyos esetekben igen rossz fákat eredeményez, ezért különböző módszerekkel javíthatunk a hatékonyságon. Ennek keretében minden elemhez hozzárendelünk egy számértéket: a rangot, ami nagyjából az elemhez tartozó fa magasságát jelzi. Ez a rang kezdetben 0. Az egyesítésnél a rangok alapján dönt a program arról, hogy a magasabb fa alá szúrja be az alacsonyabbat, illetve ha két fa egyforma magas, akkor mindegy, hogy melyiket szúrjuk be, de ekkor a megváltozott rangot be kell állítani. A keresés eljárása pedig a reprezentáns megtalálása mellett mellékhatásként összekuszálja a fát olyan értelemben, hogy a gyökértől eltekintve a keresett elemtől a gyökérig vezető úton minden elemet a gyökér alá fűz be. Ezzel pedig lényegesen csökkentheti a fa magasságát. Ezen heurisztikákat követve n elemű diszjunkt-halmazon m művelet bonyolultsága $O(m \ln^* n)$, ahol $\ln^*$ a láncthatványozás $(2^{2^{2^{\cdot\cdot}}})$ inverze.

Procedure HALMAZT-KÉSZÍT(x)
Input: x elem
Eredmény: Létrehoz egy fát, melynek egyetlen csúcsa a megadott elem
1 $x.apa \leftarrow x$
2 $x.rang \leftarrow 0$

Procedure EGYESÍT(*x*,*y*)
Input: két elem
Eredmény: a két elemet tartalmazó diszjunkt halmazokat összekapcsolja
1 ÖSSZEKAPCSOL((*HALMAZT-KERES*(x),*HALMAZT-KERES*(y))

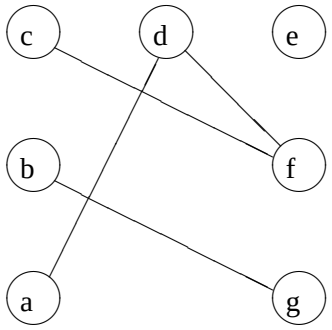
Procedure ÖSSZEKAPCSOL(*x,y*)
Input: két gyökér
Eredmény: a kisebb fát a nagyobb gyökere alá fűzi
1 **if** $x.rang > y.rang$ **then** $y.apa \leftarrow x$ **else** $x.apa \leftarrow y$
2 **if** $x.rang = y.rang$ **then** $y.rang \leftarrow y.rang + 1$

Function HALMAZT-KERES(*x*)
Input: x elem
Output: Az x reprezentánsa
1 **if** $x.apa \neq x$ **then** $x.apa \leftarrow$ HALMAZT-KERES(*x.apa*)
2 **return** x.apa

3. Összefüggő komponensek

Az ÖSSZEFÜGGŐ-KOMPONENSEK programmal egy G gráf összefüggő komponenseit határozhatjuk meg. A program elve a következő: kezdetben minden egyes csúcshoz készítünk egy diszjunkt halmazt, majd sorban az összes élre egyesítjük a végpontokhoz tartozó halmazokat. Mire az élek végére jutunk, elkészülnek a komponensek is. Tekintsük az algoritmus futását a 3. ábrán látható gráfra! Az 1. táblázatban láthatóak a program futása során generált diszjunkt halmazok. Az összefüggő komponenseket a táblázat utolsó sorában található halmazok adják meg.

Procedure ÖSSZEFÜGGŐ-KOMPONENSEK(G)
Input: G gráf
Eredmény: meghatározza a G gráf komponenseit
1 **forall the** $v \in V$ **do** HALMAZT-KÉSZÍT(*v*)
2 **forall the** $(u, v) \in E$ **do**
3 | **if** *HALMAZT-KERES*(u) $\neq$ *HALMAZT-KERES*(v) **then**
| EGYESÍT(*u,v*)
4 **endfall**



3. ábra. Példagráf összefüggő komponensek meghatározásához

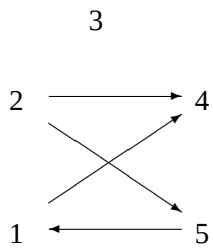
1. táblázat. A kiszámolt komponensek

Élek	Diszjunkt halmazok
	$\{a\},\ \{b\},\ \{c\},\ \{d\},\ \{e\},\ \{f\},\ \{g\}$
(a,d)	$\{a,d\},\ \{b\},\ \{c\},\ \{e\},\ \{f\},\ \{g\}$
(b,g)	$\{a,d\},\ \{b,g\},\ \{c\},\ \{e\},\ \{f\}$
(c,f)	$\{a,d\},\ \{b,g\},\ \{c,f\},\ \{e\}$
(d,f)	$\{a,c,d,f\},\ \{b,g\},\ \{e\}$

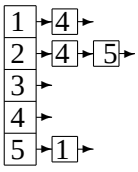
IX. fejezet
Gráfalgoritmusok

1. Gráfok ábrázolása

A gráfok ábrázolására két módszer terjedt el, s rendszerint a csúcsok és élek aránya dönti el, hogy adott feladatnál melyiket alkalmazzuk. A szomszédsági éllistás ábrázolásnál egy, a csúcsok számával megegyező elemszámú mutatótömböt használunk, és minden egyes v csúcshoz egy lista tartozik, azon u csúcsok listája, melyre (v, u) a gráf egy éle. Az 1. ábrán szereplő gráf éllistás ábrázolása a 2. ábrán látható. Ebben az ábrázolásban a szükséges tárterület a gráf csúcsai és élei számának összegével arányos. A fejezetben szereplő programokban a v csúcshoz tartozó éllistára $Adj(v)$ néven hivatkozunk. A szomszédsági mátrixhoz (másnéven csúcsmátrix) egy m kétdimenziós tömböt használunk, a csúcsokat megszámozzuk az $1, \ldots, n$ értékekkel és és $m_{ij} = 1$ akkor és csak akkor, ha (i, j) a gráf éle. Ebben az esetben a szükséges tárterület a csúcsok számával négyzetesen arányos. Az 1. ábrán szereplő gráf szomszédsági mátrixos ábrázolása az 1. táblázaton látható.



1. ábra. Példagráf



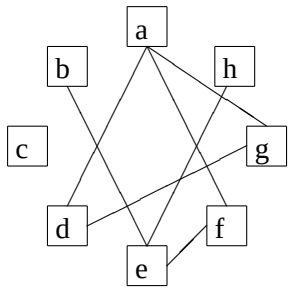
2. ábra. A példagráf éllistás ábrázolása

1. táblázat. A példagráf szomszédsági mátrixos ábrázolása

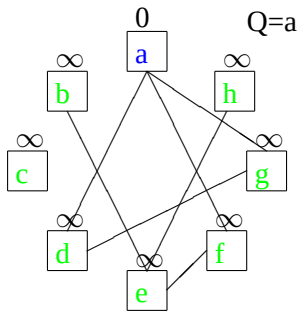
	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	0
4	0	0	0	0	0
5	1	0	0	0	0

2. Szélességi keresés

Több algoritmus alapja a szélességi keresés. Itt egy sor segítségével fedezi fel a program a gráfot, és épít fel ez alapján egy fát. Kezdetben a kezdőpontot, s-t szürkére színezi, majd a szürke csúcsok mindegyikének megkeresi a még fehér szomszédjait. Ezeket szürkére színezi, s felveszi egy sorba. Miután a szürke csúcs minden szomszédját meghatároztuk, feketére festjük. Lássuk a program futását a 3. ábrán látható gráfon. A kezdőcsúcs távolsága 0, minden más csúcs távolsága végtelen. Mivel a fehér oldalra fehér betűket nem érdemes írni, a következő színeket alkalmazzuk az soron következő ábrákon: fehér-zöld, szürke-kék, fekete-piros.



3. ábra. Eredeti gráf

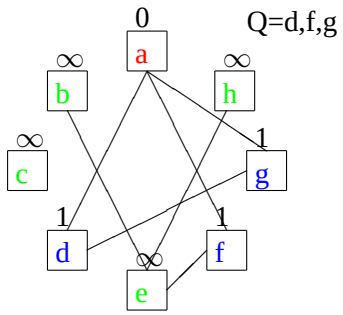


4. ábra. Kiinduló állapot, az a csúcsból kezdünk

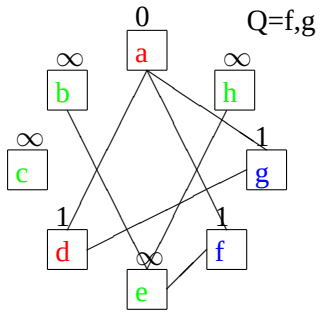
Procedure SZÉLESSÉGI-KERESÉS(*G,s*)

Input: G gráf
Eredmény: a gráf alapján egy fát generál

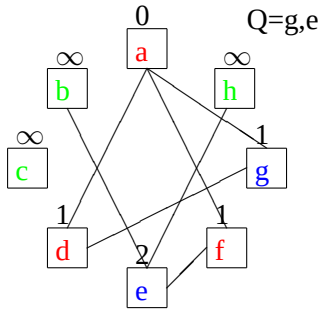
```
1  forall the u ∈ V − {s} do
2      u.szín ← FEHÉR
3      u.táv ← ∞
4      u.apa ← Nil
5  endfall
6  s.szín ← SZÜRKE
7  s.táv ← 0
8  s.apa ← Nil
9  SORBA(Q,s)
10 while not ÜRES(Q) do
11     u ← Q.fej
12     forall the v ∈ Adj(u) do
13         if v.szín = FEHÉR then
14             v.szín ← SZÜRKE
15             v.táv ← u.táv + 1
16             v.apa ← u
17             SORBA(Q,v)
18         endif
19     endfall
20     SORBÓL(Q)
21     u.szín ← FEKETE
22 endw
```



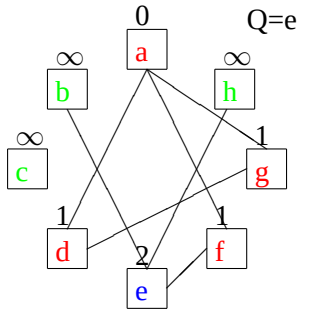
5. ábra. Az a három szomszédja d, f és g



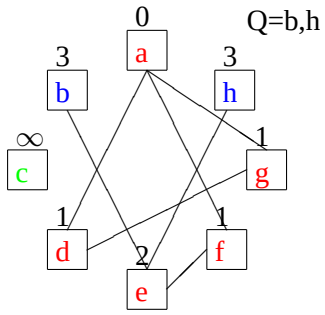
6. ábra. A d-nek nincs új szomszédja



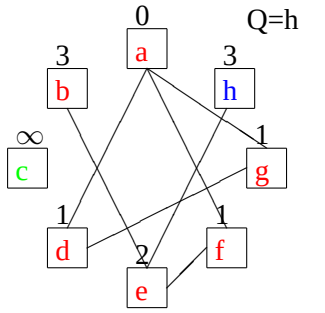
7. ábra. Az f-nek viszont van, az e



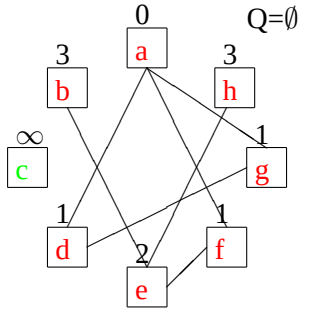
8. ábra. A g-nek sincs új szomszédja



9. ábra. Az e új szomszédjai a b és a h



10. ábra. A b-nek nincs új szomszédja



11. ábra. A h-nek sincs, és ezzel a sor is kiürült

3. Mélységi keresés

Mélységi keresés során az utoljára elért, új kivezető élekkel rendelkező csúcsok éleit derítjük fel. Ha az összes kivezető élt felderítettük, akkor a csúcs szülőjének kivezető éleit vizsgálja tovább az algoritmus. Ezt addig folytatjuk, míg a kiinduló csúcsból elérhető összes csúcsot fel nem derítettük. Ezek után a még fel nem derített csúcsok közül választunk egyet, s megkeressük az innen elérhető csúcsokat. Ezt ismételjük mindaddig, amíg az összes csúcsot fel nem fedeztük. Mikor a bejárás során szürkére festjük a csúcsot, az időpontot a pont be változójában tároljuk. A pont feketére festésének időpontját pedig ki változójában tároljuk. A három szín szerepe megegyezik a szélességi keresés színeinek szerepével.

Procedure MÉLYSÉGI-KERESÉS(G)

Input: G gráf
Eredmény: a gráf alapján egy erdőt generál

```
1  forall the u ∈ V do
2  |  u.szín ← FEHÉR
3  |  u.apa ← Nil
4  endfall
5  idő← 0
6  forall the u ∈ V do
7  |  if u.szín = FEHÉR then MÉLYSÉGI-BEJÁR(u)
8  endfall
```

Procedure MÉLYSÉGI-BEJÁR(u)

Input: u csúcs
Eredmény: a csúcsból elérhető, még fel nem derített csúcsok alapján felépít egy fát

```
1  u.szín ← SZÜRKE
2  u.be ← idő
3  idő ← idő + 1
4  forall the v ∈ Adj(u) do
5  |  if v.szín =FEHÉR then
6  |  |  v.apa = u
7  |  |  MÉLYSÉGI-BEJÁR(v)
8  |  endif
9  endfall
10 u.szín ← FEKETE
11 u.ki ← idő
12 idő←idő + 1
```

4. Topológikus elrendezés

Egy $G = (V, E)$ irányított gráf topológikus elrendezése a V elemeinek a sorbarendezése úgy, hogy ha $(u, v) \in E$, akkor u megelőzi a sorban v-t. Természetesen ha a gráf tartalmaz irányított kört, akkor nincs ilyen topológikus elrendezése. Az

Function TOPOLÓGIKUS-RENDEZÉS(G)
Input: G gráf
Output: L láncolt listája a topológikus rendezettségű csúcsoknak
1 MÉLYSÉGI-KERESÉS(G) meghívása, az $u.ki$ értékek meghatározása.
2 LISTÁBA-SZÚR(L,u) meghívása a csúcs elhagyásakor
3 **return** L

algoritmusnak megfelelően a bonyolultság $O(V + E)$.

5. Erősen összefüggő komponensek

Egy G irányított gráf erősen összefüggő komponense a csúcsok egy maximális U halmaza, melynek bármely két u és v csúcsára teljesül a következő: u-ból vezet út v-be és v-ből vezet út u-ba. A gráfot erősen összefüggő komponenseire bontó algoritmus felhasználja a G gráf G^T transzponáltját. A G^T-nek akkor és csak akkor éle az (u, v), ha G-nek éle (v, u). Az összefüggő komponenseket a csúcsok és élek számának összegével arányos bonyolultságú algoritmussal meghatározhatjuk. A megadott algoritmus tetszőleges végrehajtásával meghatározhatjuk az összefüggő komponenseket.

Function ERŐSEN-ÖSSZEFÜGGŐ-KOMPONENSEK(G)
Input: G gráf
Eredmény: a gráf alapján felépít egy erdőt, ahol egy fa felel meg egy komponensnek
1 MÉLYSÉGI-KERESÉS(G) meghívása, az $u.ki$ értékek meghatározása.
2 G^T, a G gráf transzponáltjának meghatározása
3 MÉLYSÉGI-KERESÉS(G^T) meghívása, ám a for ciklusban a pontokat $u.ki$ szerinti csökkenő sorrendben kell végigjárni.
4 Az így kapott mélységi fák csúcsai alkotnak egy-egy összefüggő komponenst.

6. Minimális költségű feszítőfa

Legyen $G = (V, E)$ egy gráf. A G gráf egy körmentes, összefüggő $F = (V, E')$ részgráfja a G gráf feszítőfája. A G gráf F feszítőfája minimális költségű, ha a

benne szereplő élek súlyának az összege minimális G összes feszítőfája közül. Kruskal algoritmusa sorra veszi az összes élt, s ha a két él különböző komponensekhez tartozik, összekapcsolja a komponenseket. Megfelelő adatszerkezet használatával az algoritmus bonyolultsága $O(E\ln(E))$.

Procedure KRUSKAL-FESZÍTŐ(*G,súly*)

Input: G gráf
Eredmény: Az A tartalmazza a feszítőfát

```
1  A ← ∅
2  forall the v ∈ V do HALMAZT-KÉSZÍT (v)
3  forall the (u,v) ∈ E (Súly szerint növekvő sorrendben!) do
4      if HALMAZT-KERES(u)≠HALMAZT-KERES(v) then
5          A ← A ∪ {(u,v)}
6          EGYESÍT (u,v)
7      endif
8  endfall
```

Prim algoritmusában felhasználunk egy `kulcs` segédtömböt, amely azt adja meg, hogy milyen messze vannak megadott gyökérből kinőtt fától ebben a fában még nem szereplő csúcsok. (Ezt a távolságot kezdetben végtelenre állítottuk.) Majd sorra átemeljük a fába a legközelebbi csúcsot. Természesen ekkor újra be kell állítani ezen csúcs szomszédjainak távolságát, s a távolsághoz tartozó szülőt. Az algoritmus bonyolultsága kupacok használatával $O(E\ln(V))$, ám más, speciális adatszerkezettel $O(E+V\ln(V))$-re csökkenthető.

Procedure PRIM-FESZÍTŐ(*G,súly,r*)

Input: G gráf, súly az élek súlyozása, r kezdőcsúcs
Eredmény: r gyökerű faként állítja elő a feszítőfát

```
9  Q ← V
10 forall the v ∈ Q do v.kulcs ← ∞
11 r.kulcs ← 0
12 r.apa ← Nil
13 while Q ≠ ∅ do
14     u ← KIVESZ-MIN(Q)
15     forall the v ∈ Adj(u) do
16         if v ∈ Q and súly(u,v) ≤ v.kulcs then
17             v.apa ← u
18             v.kulcs ← súly(u,v)
19         endif
20     endfall
21 endw
```

7. Legrövidebb utak problémája

Adott egy $G = (V, E)$ gráf és egy súlyfüggvény, amely az élekhez egy nemnegatív valós számot rendel. Egy út költsége az utat alkotó élekhez tartozó számok összege. A legrövidebb út súlya a minimális költségű út költsége. A feladatok természetétől függően más és más értékek iránt érdeklődünk. A jellemző problémák a következők:

- Adott csúcsból induló legrövidebb utak problémája.
- Adott csúcsba érkező legrövidebb utak problémája.
- Adott csúcspár közötti legrövidebb út problémája.
- Összes csúcspár közti legrövidebb utak problémája.

7.1. Fokozatos közelítés

A gráf minden egyes pontjához hozzárendelünk egy valós számot, amely majd az adott csúcstól (s-től) való távolságát fogja tartalmazni. Ezt kezdetben az s kivételével végtelenre állítjuk. Továbbá felépítünk egy fát, mely az apák fele mutató mutatókból áll össze, s melynek a gyökere s. A kezdőértékek beállítását a KEZDŐÉRTÉK rutin végzi.

Procedure KEZDŐÉRTÉK(*G,s*)
Input: G gráf, s kezdőcsúcs
Eredmény: a gráf csúcsaihoz végtelen távolságot rendel
1 **forall the** $v \in V$ **do**
2 v.táv $\leftarrow \infty$
3 $v.apa \leftarrow$ Nil
4 **endfall**
5 s.táv $\leftarrow 0$

A legrövidebb utakat meghatározó módszerek lényege a következő: sorra vesszük az éleket, s ha az adott élt használva rövidebb utat találunk, akkor frissítjük a megfelelő adatokat. Ehhez a frissítéshez a KÖZELÍT rutin tartozik.

Procedure KÖZELÍT(*u,v,súly*)
Input: u, v csúcsok, súly élek súlyozása
Eredmény: a v távolságát frissíti az u távolsága alapján
1 **if** $v.táv > u.táv + súly(u, v)$ **then**
2 v.táv $\leftarrow u$.táv + súly(u, v)
3 $v.apa \leftarrow u$
4 **endif**

7.2. Dijkstra algoritmusa

Dijkstra algoritmusa egy S halmazt alkalmaz, mely azokat a csúcsokat tartalmazza, melyeknek már ismerjük a pontos távolságát. A Q tartalmazza a maradék csúcsokat. Ezeket a csúcsokat olyan adatszerkezetben tároljuk, melyet a csúcsok *táv* értékei alapján rendeztünk. Az algoritmus sorra megkeresi az S halmazhoz legközelebbi csúcsot, ezzel a csúccsal bővíti az S halmazt, majd ezen csúcs szomszédjainak kiszámítja a távolságát. Ha Q tárolására tömböt használunk, a bonyolultság $O(V^2)$. Ritka gráfnál érdemes ugyanerre kupacot használni, s ekkor a bonyolultság $O((V+E)\ln(V))$

Procedure `DIJKSTRA(`*G,s*`)`

Input: `G` gráf, `s` kezdőcsúcs
Eredmény: a gráf csúcsainak az adott csúcstól mért távolságának meghatározása

```
1  KEZDŐÉRTÉK(G,s)
2  S ← ∅
3  Q ← V
4  while Q ≠ ∅ do
5  |  u ← KIVESZ-MIN(Q)
6  |  S ← S ∪ {u}
7  |  forall the v ∈ Adj(u) do
8  |  |  KÖZELÍT(u,v,súly)
9  |  endfall
10 endw
```

7.3. Bellmann-Ford algoritmusa

A Bellmann-Ford algoritmus azokban az esetekben is működik, ha egyes élekhez negatív súlyok is tartoznak. Természetesen ha negatív súlyú kör szerepel a gráfban, akkor a minimális távolság nem definiálható. A külső ciklus annyiszor fut le, amilyen hosszú lehet a leghosszabb út. Minden egyes ciklusmagban eggyel messzebb található csúcsok távolságát határozzuk meg. Végül teszteljük, hogy van-e negatív súlyú kör. A két ciklusnak megfelelően a bonyolultság $O(E \cdot V)$.

7.4. Irányított körmentes gráfok esete

Irányított körmentes gráfot O(V+E) bonyolultsággal topológikusan rendezhetjük a korábban már ismertetett módszerrel. Ezek után a legrövidebb utak meghatározásához csupán e rendezés alapján kell sorbalépkedni csúcsokon, s meghatározni a szomszédjaik távolságát.

Function BELLMANN-FORD(*G,súly,s*)

Input: G gráf, súly élek súlyozása, s kezdőcsúcs
Output: IGAZ, ha meghatározhatóak a minimális távolságok, s HAMIS, ha nem
Eredmény: a gráf csúcsainak az adott csúcstól mért távolságának meghatározása

```
1 KEZDŐÉRTÉK (G,s)
2 for i ← 1 to |V|-1 do
3 |   forall the (u, v) ∈ E do KÖZELÍT(u,v,súly)
4 endfor
5 forall the (u, v) ∈ E do
6 |   if v.táv > u.táv + súly(u, v) then return HAMIS
7 endfall
8 return IGAZ
```

Procedure IKG-LEGRÖVIDEBB-ÚT(*G,súly,s*)

Input: G gráf, súly élek súlyozása, s kezdőcsúcs
Eredmény: a gráf csúcsainak az adott csúcstól mért távolságának meghatározása

```
1 KEZDŐÉRTÉK(G,s)
2 forall the u ∈ V a topológikus rendezésnek megfelelően do
3 |   forall the v ∈ Adj(u) do KÖZELÍT(u,v,súly)
4 endfall
```

7.5. Legrövidebb utak meghatározása minden csúcspárra

A soron következő négy algoritmus bemenő adata egy W $n \times n$-es mátrix. Ezen mátrix főátlójában 0 értékek szerepelnek, (minden csúcs távolsága saját magától zéró) míg ha a jelzett két pont között él található, akkor a mátrix adott pozíciójában az él súlya szerepel, egyébként végtelen. Az algoritmusok kimenete egy ugyancsak $n \times n$-es mátrix, mely a megfelelő csúcsok közötti távolságot tartalmazza. Az ÚTBŐVÍTÉS rutin minden egyes végrehajtásakor az eddig már kiszámolt utakat egy éllel bővít. (Az ÚTBŐVÍTÉS rutinban a végtelenhez tetszőleges értékeket hozzáadva, nem végtelen értéket kivonva továbbra is végtelent kapunk eredményül.)

Ha egy, a főátlójában 0, egyébként végtelen értékeket tartalmazó mátrixból indulnánk ki, akkor ez a nulla hosszúságú utakat tartalmazná. Az ÚTBŐVÍTÉS rutint kell végrehajtani erre a mátrixra $n-1$-szer, miután a gráfban legfeljebb $n-1$ hosszú út található. Ezzel az algoritmus bonyolultsága $O(V^4)$. Ha a kiinduló mátrix a W, akkor a bővítést csak $n-2$-szer kell végrehajtani (LASSÚ-LEGRÖVIDEBB-ÚT). Az útbővítés során egyszer már kiszámolt útsorozatokat is fel lehet használni, s a D mátrixot nem a W alapján frissítjük, hanem saját magát használva (GYORSABB-LEGRÖVIDEBB-ÚT). Ezzel a bonyolultság $O(V^3 \ln V)$-re csökken (A útbővítés

Function ÚTBŐVÍTÉS(*D*,*W*)

Input: D számolt távolságmátrix, W távolságmátrix az élek alapján
Output: D′ továbbszámolt távolságmátrix

```
1  n ← sorok száma a W mátrixban
2  D′ = (d′_ij) ← n × n-es mátrix
3  for i ← 1 to n do
4  |  for j ← 1 to n do
5  |  |  d′_ij ← ∞
6  |  |  for k ← 1 to n do
7  |  |  |  d′_ij = min(d′_ij, d_ik + w_kj)
8  |  |  endfor
9  |  endfor
10 endfor
11 return D′
```

rutinja nagyon hasonlít a mátrixszorzásra. Míg az első esetben „sorozatos szorzás” szerepel, a második esetben „sorozatos négyzetreemelés”.)

Function LASSÚ-LEGRÖVIDEBB-ÚT(*W*)

Input: W távolságmátrix az élek alapján
Output: D a gráf csúcsainak egymástól mért távolságaiból álló mátrix

```
1  n ← sorok száma a W mátrixban
2  D ← W
3  for m ← 2 to n − 1 do D ← ÚTBŐVÍTÉS(D,W)
4  return D
```

Function GYORSABB-LEGRÖVIDEBB-ÚT(*W*)

Input: W távolságmátrix az élek alapján
Output: D a gráf csúcsainak egymástól mért távolságaiból álló mátrix

```
1  n ← sorok száma a W mátrixban
2  D ← W
3  m ← 1
4  while n − 1 > m do
5  |  D ← ÚTBŐVÍTÉS(D,D)
6  |  m ← 2m
7  endw
8  return D
```

7.6. Floyd-Warshall algortimus

Egy másik megközelítésben a gráf csúcsait csak korlátozottan vesszük igénybe, a k növekedésével egyre több és több csúcsot használhatunk az utak felépítésére. A három egymásba ágyazott ciklusnak megfelelően $O(V^3)$ az algoritmus bonyolultsága.

Function FLOYD-WARSHALL(W)

Input: W távolságmátrix az élek alapján
Output: D a gráf csúcsainak egymástól mért távolságaiból álló mátrix

```
1  n ← sorok száma a W mátrixban
2  D ← W
3  for k ← 1 to n do
4      for i ← 1 to n do
5          for j ← 1 to n do
6              d_ij = min(d_ij, d_ik + d_kj)
7          endfor
8      endfor
9  endfor
10 return D
```

X. fejezet
Mintaillesztés

Bizonyos típusú adatoknál gyakran előfordul, hogy egy hosszabb szövegben kell megkeresni egy minta összes előfordulását. Feltesszük, hogy a szöveget a $T[1..n]$ tömb tartalmazza, míg a mintát a $P[1..m]$ tömb. Mindkét tömb elemei egy adott véges ábécé elemei. Azt modjuk, hogy a P minta a T szövegben s eltolással előfordul, ha minden $1 \leq i \leq m$ értékre $P[i] = T[s + i]$

1. Brute force (nyers erő)

A legegyszerűbb módszer esetén a mintát - mint egy sablont - végigtoljuk a szövegen, s ellenőrizzük, hogy a minta jelei megegyeznek-e a szöveg megfelelő jeleivel. A 1. táblázat mutatja az algoritmust futását. A táblázatból leolvasható, hogy 1 eltolással a minta illeszkesdik a szövegre. A két egymásba ágyazott ciklusnak

1. táblázat. Brute force algoritmus

Szöveg	1	1	0	1	0	0	1	1
Minta 0 eltolással	1	0	1	0	0			
Minta 1 eltolással		1	0	1	0	0		
Minta 2 eltolással			1	0	1	0	0	
Minta 3 eltolással				1	0	1	0	0

megfelelően a bonyolultság $O(nm)$. A brute.htm állomány egy olyan programot tartalmaz, amely véletlen módon generált szövegre és mintára végrehajta a Brute force módszert. A pontok jelzik a teljes szöveget, a számok a minta illesztését a teljes szövegre. Ha a minta illeszkedik, akkor az a program zölddel jelzi.

2. Rabin-Karp algoritmus

Az előző módszernél újra és újra megvizsgáljuk T egyes elemeit. A soron következő módszernél egy tömbelemet rendszerint elegendő kétszer megvizsgálni. A módszer lényege a következő: mind a mintához, mind a szöveg mintával megegyező hosszúságú részeihez egy-egy számot rendelünk egy hasítófüggvény felhasználásával. A szöveg egymást átfedő ilyen részeihez tartozó számokat könnyen

Procedure EGYSZERŰ-MINTAILLESZTŐ(*T,P*)
Input: T szöveg, P minta
Eredmény: illeszkedés esetén figyelmeztetés
1 $n \leftarrow T.hossz$
2 $m \leftarrow P.hossz$
3 **for** $s \leftarrow 0$ ***to*** $n - m$ **do**
4 | **if** $P[1..m] = T[s + 1..s + m]$ **then**
5 | | **kiír** *A minta előfordul s eltolással*
6 | **endif**
7 **endfor**

meghatározhatjuk, csak a belépő és kilépő tömbelemeket kell felhasználni a számolás során. Az s eltolásnál a számhoz a következő értéket rendeljük

$$h = \sum_{i=1}^{m} (T[s + i] \cdot d^{m-i}).$$

Ugyanez $s + 1$ eltolásnál

$$h' = \sum_{i=1}^{m} (T[(s + 1) + i] \cdot d^{m-i}).$$

Innen $h' = h \cdot d - T[s + 1] \cdot d^m + T[s + m]$, azaz az eggyel eltolt mintához tartozó mennyiség a minta határainál található értékek alapján meghatározható. Mivel hosszabb minta esetén kényelmetlen a d^m méretű számokkal dolgozni, érdemes a hányados módszert használni egy q prímszámmal, s a korábbi mennyiségek maradékaival számolni. A 2. táblázatban a Rabin-Karp algoritmusa futását követhetjük nyomon, ahol $q = 17$ és $d = 2$. A táblázatból látható, hogy a minta hasítóértéke 14. Az első öt eltolás esetén a szöveg megfelelő részeinél a hasítóértékek három esetben egyeznek meg ezzel, de mint azt már láttuk, hogy az ütközéseknél nem feltétlenül egyeznek meg a kulcsok. Esetünkben is csak az 1 eltolásnál egyezik meg a minta a szöveg megfelelő részével. Mivel csak a hasítóértékek egybeesésénél kell a szövegrészek egyezését figyelni, a bonyolultság nagyjából $O(m + n)$. A rabin.htm állomány egy olyan programot tartalmaz, amely véletlen módon generált szövegre és mintára végrehajta a Rabin-Karp módszert. A pontok jelzik a teljes szöveget, a kék számok a hasító értéket, a piros számok az ütközést, a zöld számok minta illesztését jelzik. A számok szöveg megvizsgált betűit jelölik, az aláhúzások az eltolásra utalnak.

3. Knuth-Morris-Pratt algoritmus

A Knuth-Morris-Pratt algoritmus bonyolultsága hasonlóan az előző algoritmushoz $O(n + m)$. Ennél az algoritmusnál készítünk egy prefixtáblázatot, melyből leolvasható, hogy ha valamely karakter nem illeszkedik, mely eltolásokat hagyhatjuk ki mindenképp. A prefixtáblázat elkészítéséhez csak a mintára van szükség, és

2. táblázat. Rabin-Karp algoritmus

Szöveg	0	0	1	1	1	0	0	1	1	0
Minta (14)	0	1	1	1	0					
0 eltolás (7)	0	_	_	_	1					
1 eltolás (14)		0	1	1	1	0				
2 eltolás (14)			1	1	1	0	0			
3 eltolás (14)				1	1	0	0	1		
4 eltolás (13)					1	_	_	_	1	

Procedure RABIN-KARP-ILLESZTŐ(*T,P,d,q*)

Input: T szöveg, P minta, d egész, q prím
Eredmény: illeszkedés esetén figyelmeztetés

```
1  n ← T.hossz
2  m ← P.hossz
3  h ← d^(m−1) mod q
4  p ← 0
5  t ← 0
6  for i ← 1 to m do
7  |  p ← (dp + P[i]) mod q
8  |  t ← (dt + T[i]) mod q
9  endfor
10 for s ← 0 to n − m do
11 |  if p = t then
12 |  |  if P[1..m] = T[s+1..s+m] then
13 |  |  |  kiír A minta előfordul s eltolással
14 |  |  endif
15 |  endif
16 |  if s < n − m then
17 |  |  t ← (d(t − T[s + 1]h) + T[s + m + 1]) mod q
18 |  endif
19 endfor
```

azt kell vizsgálnunk, hogy a minta prefixeinek (kezdőszeletei) valamely suffixe (végszelete) prefixe-e vagy sem. magyarul, előáll-e a minta eleje egy $\alpha\beta\gamma$ alakban, ahol α, β és γ esetleg üres karaktersorozat és $\alpha = \gamma$, vagy $\alpha\beta = \beta\gamma$. Ha igen, a prefixtáblázatba a leghosszabb α, illetve $\alpha\beta$ hossza kerül be. A 3. ábrán sorra felsoroljuk a kezdőszeleteket, s alá- illetve föléhúzással jelöljük az egyező részeket.

A knuth.htm állomány egy olyan programot tartalmaz, amely véletlen módon generált szövegre és mintára végrehajta a Knuth-Morris-Pratt módszert. A pontok jelzik a teljes szöveget, az aláhúzások az eltolásra utalnak, a zöld számok pedig az illeszkedést jelzik.

3. táblázat. Prefixfüggvény számítás

Prefix	függvényérték
0	0
01	0
$\underline{0}1\overline{0}$	1
$\underline{01}\overline{01}$	2
$\underline{0}1\overline{0}1\overline{0}$	3
$\underline{0}10100\overline{0}$	1
$\underline{0}101000\overline{0}$	1
$\underline{01}0100\overline{01}$	2

Function PREFIX-FÜGGVÉNY-SZÁMÍTÁS(P)

Input: P minta
Output: prefixfüggvény táblázata

```
1  m ← P.hossz
2  p[1] ← 0
3  k ← 0
4  for q ← 2 to m do
5  |  while k > 0 and P[k+1] ≠ P[q] do k ← p[k]
6  |  if P[k+1] = P[q] then k ← k+1
7  |  p[q] ← k
8  endfor
9  return p
```

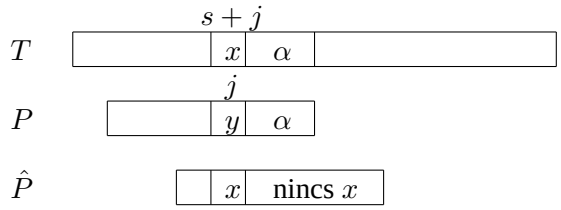
Procedure KMP-ILLESZTŐ(T,P)

Input: T szöveg, P minta
Eredmény: illeszkedés esetén figyelmeztetés

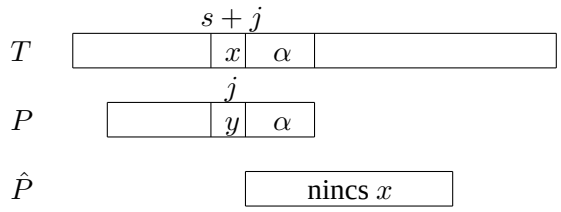
```
1  n ← T.hossz
2  m ← P.hossz
3  p ← PREFIX-FÜGGVÉNY-SZÁMÍTÁS(P)
4  q ← 0
5  for i ← 1 to n do
6  |  while q > 0 and P[q+1] ≠ T[i] do q ← p[q]
7  |  if P[q+1] = T[i] then q ← q+1
8  |  if q = m then
9  |  |  kiír A minta előfordul q eltolással
10 |  |  q ← p[q]
11 |  endif
12 endfor
```

4. Boyer-Moore algoritmus

Az eddigi módszerek a mintát balról jobbra ellenőrizték. A Boyer-Moore algoritmus a mintát ugyancsak ebbe az irányba mozgatja, viszont ott már jobbról balra ellenőrzi az illeszkedést. Ez a módszer két heurisztikát használ a soron következő eltolás méretének meghatározására. Az első, az „utolsó karakter” heurisztika minden egyes betűhöz (karakterhez) megadja, hogy az hol fordult elő utoljára a mintában (1. és 2. ábra). Ha a szöveg épp vizsgált karaktere nem fordul elő a mintában, akkor a mintát eltolhatjuk eme karakter után. Ha viszont szerepel benne a karakter, akkor annak az utolsó előfordulását illesztjük a szöveg megfelelő karakteréhez. Ez elvileg jelenthetne negatív (bal irányú) eltolást is, de ekkor a másik heurisztikát alkalmazzuk.

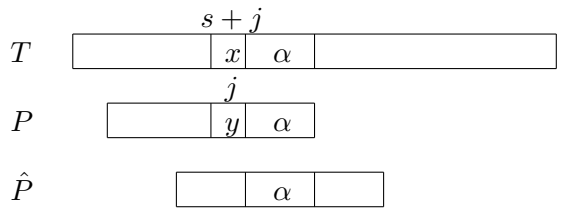


1. ábra. Utolsó pozíció heurisztika, amikor a nem egyező karakter szerepel a mintában

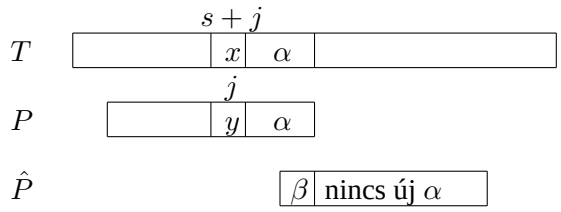


2. ábra. Utolsó pozíció heurisztika, amikor a nem egyező karakter nem szerepel a mintában

A „jó szuffix” heurisztikánál úgy mozgatjuk a mintát, hogy a szövegnek arra a részére, amelyre már illeszkedett minta vizsgált része, újra (legalább részben) illeszkedjék. Ha az illeszkedő részminta (az ábrán α) újra előfordul a szövegben, akkor ezt a részt kell illeszteni (3. ábra), egyébként eme minta egy szuffixét (az ábrán β) kell illeszteni a szöveghez (3. ábra).
A Boyer-Moore algoritmus nem illeszekedés esetén a két heurisztika közül azt választja, amellyel nagyobbat léphet. Noha a módszer bonyoltsága $O(nm)$, a legrosszabb eset csak kivételes esetekben fordul elő, s kellően nagy ábécé esetén az



3. ábra. A jó szuffix heurisztika, mikor az α újra előfordul



4. ábra. A jó szuffix heurisztika, mikor az α nem fordul elő újra

Function UTOLSÓ-POZÍCIÓ(*P,m,S*)

Input: P minta, m minta hossza,S ábécé
Output: u utolsó pozíció heurisztika értékei

```
1 forall the a ∈ S do u[a] ← 0
2 for j ← 1 to m do u[P[j]] ← j
3 return u
```

Function JÓ-SZUFFIX(*P,m*)

Input: P minta, m minta hossza
Output: g jó szuffix heurisztika értékei

```
1 p_1 ← PREFIX-FÜGGVÉNY-SZÁMÍTÁS(P)
2 P' ← FORDÍT(P)
3 p_2 ← PREFIX-FÜGGVÉNY-SZÁMÍTÁS(P')
4 for j ← 0 to m do g[j] ← m − p_1[m]
5 for l ← 1 to m do
6 |   j ← m − p_2[l]
7 |   if g[j] > l − p_2[l] then g[j] ← l − p_2[l]
8 endfor
9 return g
```

algoritmus rendszerint nagyon gyors. Ez az oka annak, hogy napjainkban szinte minden szövegszerkesztőben ezt a módszert implementálták. A boyer.htm állomány egy olyan programot tartalmaz, amely véletlen módon generált szövegre

```
Procedure BOYER-MOORE-ILLESZTŐ(T,P,S)
   Input: T szöveg, P minta, S ábécé
   Eredmény: illeszkedés esetén figyelmeztetés
 1 n ← T.hossz
 2 m ← P.hossz
 3 u ← UTOLSÓ-POZÍCIÓ(P,m,S)
 4 g ← JÓ-SZUFFIX(P,m)
 5 s ← 0
 6 while s ≤ n − m do
 7 |   j ← m
 8 |   while j > 0 and P[j] = T[s + j] do j ← j − 1
 9 |   if j = 0 then
10 |   |   kiír illeszkedés az s pozíción
11 |   |   s ← s + g[0]
12 |   else
13 |   |   s ← s + max(g[j], j − u(T[s + j]))
14 |   endif
15 endw
```

és mintára végrehatja a Boyer-Moore algoritmust. A honlapon az aláhúzások jelzik az illeszkedést, a számok pedig a szöveg megvizsgált karaktereit.

XI. fejezet
Fejlett programozási módszerek

1. Dinamikus programozás

A dinamikus programozást rendszerint valamilyen numerikus paraméterektől függő érték optimumának meghatározására használjuk. A lényeg a következő: az optimális megoldást optimális részmegoldásokból állítjuk elő. Az optimális részmegoldásokat egy táblázatban tároljuk, s az optimális megoldást ennek a táblázatnak szisztematikus feltöltésével kaphatjuk meg. A táblázat elemeit rendszerint egy rekurzív összefüggéssel határozhatjuk meg. Az egyik legegyszerűbb példa a dinamikus programozásra a binomiális együtthatók meghatározása. Itt a Pascal háromszög kiszámításával nyerhetők a kívánt együtthatók. A megfelelő rekurzív összefüggés a következő

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Természetesen a kezdéshez szükségünk van a $\binom{n}{0} = \binom{n}{n} = 1$ képletekre is.

Function SZORZÁSOK-SZÁMA(P)

Input: P a mátrixok méreteiből összeállított tömb
Eredmény: m tömbben a szorzások száma, s tömbben a zárójelezések helye

```
1  n ← P.hossz − 1
2  for i ← 1 to n do m[i, i] ← 0
3  for l ← 2 to n do
4      for i ← 1 to n − l + 1 do
5          j ← i + l − 1
6          m[i, j] ← ∞
7          for k ← i to j − 1 do
8              q ← m[i, k] + m[k + 1, j] × P[i − 1] × P[k] × P[j]
9              if q < m[i, j] then
10                 m[i, j] ← q
11                 s[i, j] ← k
12             endif
13         endfor
14     endfor
15 endfor
16 return m és s
```

A binomiális együtthatók meghatározásánál kicsit nehezebb az a feladat, amikor az $A_1A_2...A_n$ mátrixszorzatot kell kiszámolnunk. A mátrixszorzás asszociatív, így a szorzások sorrendje tetszőleges (ezért nem is használtunk zárójeleket az előbbi képletben). Az exponenciális számú lehetséges kiszámítási sorrend közül azt érdemes követni, amely a legkevesebb elemi szorzással jár. Egy $i \times j$ és $j \times k$ méretű mátrix összeszorzása $i \cdot j \cdot k$ elemi szorzást igényel. Tartalmazza az m mátrix i, j mezője azt, hogy minimálisan mennyi szorzás szükséges az $A_i...A_j$ mátrixszorzat előállításához! Ez a szorzat $A_i...A_k$ és $A_{k+1}...A_j$ mátrixok szorzataként állhat elő, ahol k i és j között mozoghat. Innen következik a rekurzív összefüggés: $m[i, j]$ a $m[i, k] + m[k + 1, j] + a[i, j, k]$ minimális értéke lesz, ahol az $a[i, j, k]$ a $A_i...A_k$ és $A_k + 1...A_j$ mátrixok méreteinek szorzatát jelenti. Ennek alapján a szorzások számát a SZORZÁSOK-SZÁMA algoritmus adja meg.

2. Mohó algoritmus

A dinamikus programozásnál több érték közül választottuk ki az optimálisat. Bizonyos esetekben nem kell több lehetőséget végigpróbálni, már elsőre kiválaszthatjuk a legjobbat. Egy ilyen módszer a Huffman kódolás. Tegyük fel, hogy egy 10000 hosszúságú üzenetben csak a, ..., f betűk szerepelnek, s a gyakoriságok rendre legyenek 30%, 10%, 5%, 10%, 25% és 20%. Ha minden betűt három bittel kódolunk, akkor 30000 bit segítségével tárolhatjuk vagy továbbíthatjuk az üzenetet. Ha viszont eltérő hosszúságú bitsorozatokkal kódoljuk a betűket, akkor spórolhatunk a bitekkel. Ha a kódolásunk a következő a-00, b-0100, c-0101, d-011, e-10 és f-11, akkor 24000 bit is elegendő.
Ebben az esetben is egyértelműen dekódolható az üzenet, ugyanis egyik betű kódja sem prefixe egy másiknak. Sőt mi több, ez a kódolás optimális, azaz nem kódolható rövidebben az üzenet. A kód elkészítéséhez a következőket tételezzük fel: a betűk halmaza az n elemű C halmaz, és minden egyes x betűhöz tartozik egy $x.f$ gyakorisági érték.
Az algoritmus dinamikus halmaz két legkisebb gyakoriságú elemét összeragasztja, s ezt addig folytatja, amíg csak egy pont marad. Az összeragasztás során feljegyeztük, hogy melyik betűnek hol a helye, s ennek alapján a keletkezett fából leolvasható a kód. (Ha balra kell haladni a keresésnél, akkor 0-t fűzünk a kódhoz, egyébként 1-et.) A tanult gráfalgoritmusok nagy része hasonlóan mohó algoritmusnak tekinthető.

3. Korlátozás és elágazás (Branch and bound)

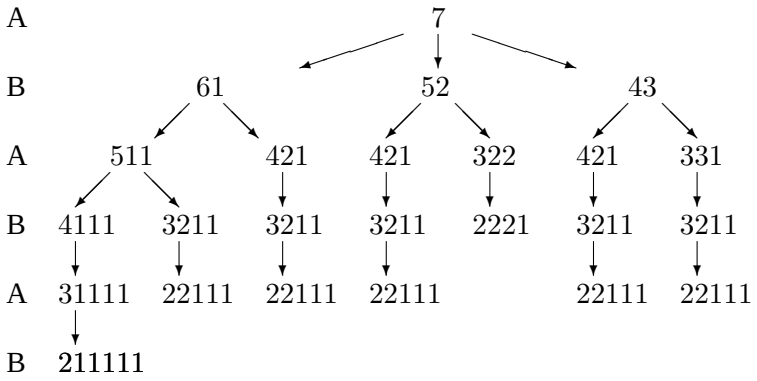
A kétszemélyes játékok esetén valamilyen kezdőállapotból kiindulva, a játékosok felváltva lépnek, s véges számú lépés után valamilyen végállapotba jutnak. A játékban nincs szerepe a szerencsének, s játék szabályai határozzák meg, hogy az adott végállapotban ki a győztes. Ilyen játékok például a sakk, a dáma, a go, de nem

Function HUFFMAN(C)

Input: C ábécé gyakoriságokkal
Output: egy fa, melyből leolvasható a kódolás

```
1  n ← C.hossz
2  Q ← C
3  for i ← 1 to n − 1 do
4  |   z ← PONTOT-LÉTESÍT()
5  |   x ← KIVESZ-MIN(Q)
6  |   y ← KIVESZ-MIN(Q)
7  |   z.bal ← x
8  |   z.jobb ← y
9  |   z.f ← x.f + y.f
10 |   BESZÚR(Q,z)
11 endfor
12 return KIVESZ-MIN(Q)
```

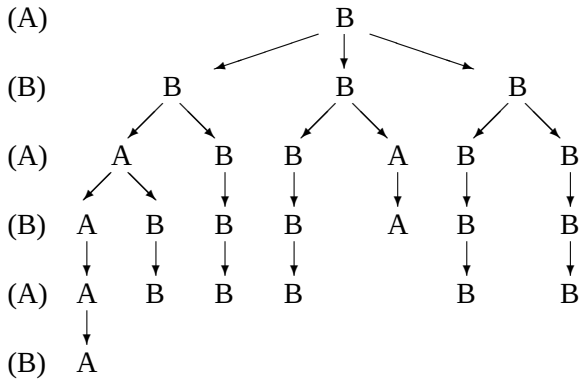
ilyen a backgammon (ostábla). Ilyen játék Grundy-féle játék is, ahol egy n érméből álló oszlopból indulunk. A játékosok felváltva lépnek, s egy lépésben egy oszlopot két eltérő magasságú oszlopra kell bontani. Ha ez nem lehetséges, a soron következő játékos vesztett. Az 1. ábrán látható a 7 érmével kezdődő játék lehetséges kimenetei. Minden egyes szám egy oszlopot jelent, így például a 322 azt jelenti, hogy van egy három, és két kettő magasságú oszlopunk. A két játékost A-val és B-vel jelöljük. Az ábráról látható, hogy az A kezd, s a legbaloldalibb ágon haladva az A nem tud tovább lépni, míg a legjobboldali ágon a B. Mely játékos érheti el, hogy mindenképpen győzzön? Az 1. ábrán látható fa leveleit aszerint



1. ábra. Grundy-féle játék fája 7 érme esetén

nevezzük el a 2. ábrán, hogy az adott ághoz tartozó játék esetén ki a nyerő. A nem levél csúcs esetén, ha csak egy fia van, akkor az apának is ugyanaz lesz az

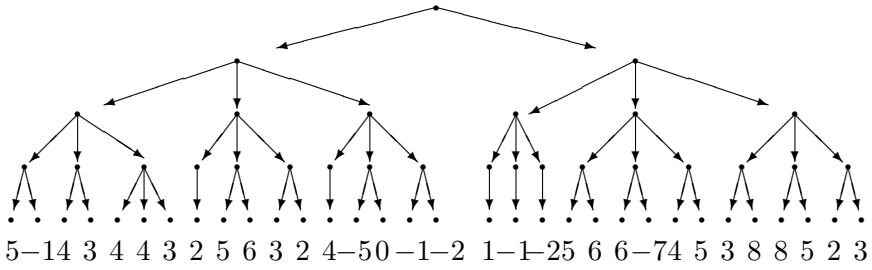
elnevezése. Több fiú esetén, mivel mindkét játékos nyerni akar, ha az A szinten levő apának van A jelű fia, akkor az apa is A jelű lesz, különben B. Hasonlóan, ha a B szinten levő apának minden fia A jelű, akkor az apa is A jelű, egyébként B jelű. Mivel a fa gyökere B jelű, így a kezdő játékos mindenképpen veszít, ha a második játékos okosan játszik. A játékfa felcimkézését elméletileg minden em-



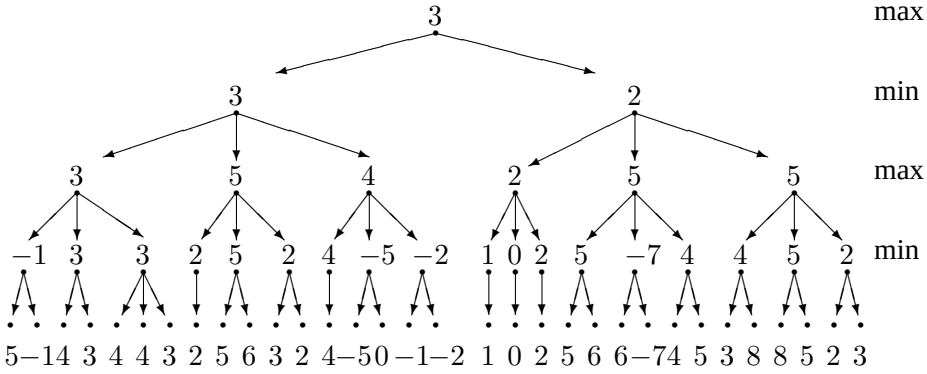
2. ábra. A felcimkézett Grundy-féle játékfa

lített játék esetén fel lehetne írni. Így kiderülhetne az is, hogy sakkban a kezdő vagy a másik játékosnak van nagyobb esélye a nyerésre. Viszont mivel a szabályok alapján egy 80 szintes fát kellene elkészíteni, amely rendszerint több mint 10 irányban ágazik el minden csúcsban, időtlen időkig eltartana a fa elkészítése és felcímkézése. Épp ezért a sakkprogramok nem készítik el az egész fát, csak annak egy kicsi részét (úgy 8-10 szintet). Ezután minden egyes pozícióhoz hozzárendelnek egy számot, amely arra utal, hogy az állás mennyire jó a program számára. (Nagy pozitív számok a nyerő, a nagy negatív számok s vesztes helyzetekre utalnak.) Egy ilyen részfát mutat a 3. ábra. A soron következő játékos természesen a legnagyobb értékű álláshoz szeretne eljutni, de az ellenfele ebben akadályozza. A játékos maximalizálni szeretné az adott állásban elérhető értéket, az ellenfél pedig minimalizálni. Ennek megfelelően számolhatjuk ki az apa értékét a fiai értékéből a minimum vagy a maximum függvényt alkalmazva, attól függően, hogy az apa páratlan vagy páros távolságra van-e a gyökértől (4. ábra). A gyökér értéke és a fiai értéke alapján könnyen megadható, hogy mely lépést kell választani, hogy a legjobbat hozzuk ki az adott állásból.

A minimax módszerrel (minimális-maximális értékek meghatározásánál) minden csúcs esetén ki kell számolni a csúcs értékét. Ez viszont gyakran felesleges. S mivel a játékprogram erőssége a megvizsgált szintek számával arányos, a programozók minden felesleges vizsgálatot szeretnének kiiktatni, hogy ugyanannyi idő alatt több szintet is megvizsgálhasson a programuk. Ezért minden program alkalmazza az α-β vágást (5. ábra). Ugyanazt a fát használjuk mint korábban, s ugyanazokat a minimax értékeket is kell megkapnunk, csak kevesebb számolással. Mivel a számozott szint feletti szint minimum szint, az a-val jelölt csúcs értéke $\min(5, -1)$,

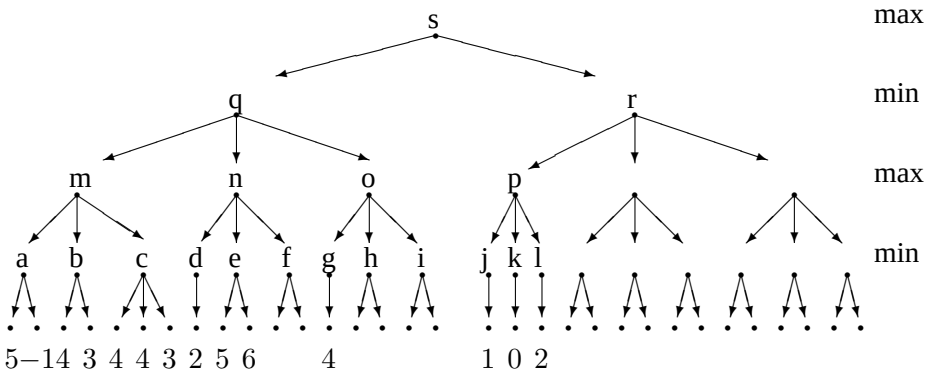


3. ábra. Játékfa-részlet



4. ábra. Minimax értékek

azaz $a = -1$. Miután $m = \max(a, b, c)$, így $a \leq m$, tehát $-1 \leq m$. A b értéke is minimummal határozható meg, így $b = 3$, s mivel $b \leq m$, tehát $3 \leq m$. Hasonlóan $c = 3$, s mivel a c az m utolsó fia, kiderült, hogy $m = 3$. Ezért miután $q = \min(m, n, o)$, $q \leq m$, azaz $q \leq 3$. Miután csak egy fia van, $d = 2$, s innen $2 \leq n$. Mivel a q meghatározásakor minimumot kell használni, s n értéke még lehet kisebb az m értékénél, tovább folytatjuk ebben az ágban. Az e fiait megvizsgálva kiderül, hogy $e = 5$, tehát $5 \leq n$. Azaz a $m < n$, tehát q meghatározásánál nincs szükség az n pontos értékére, az f meghatározását egy az egyben kihagyhatjuk (β vágás). Az o meghatározásához először a g értékét kell megadni, s ez 4 lesz. Innen $4 \leq o$, de így $m < o$, tehát az o pontos értéke sem érdekes, kihagyható mind a h, mind az i meghatározása. Miután a q összes fiát megvizsgáltuk, kapjuk a $q = 3$ eredményt. Minthogy $s = \max(q, r)$, $3 \leq s$. r értékéhez mindenképpen szükség van a p, s ehhez a j értékére. Könnyen adódik a $j = 1$. A k meghatározásánál minimumot számolunk, s mivel az első fiúnál szereplő 0 érték miatt $k \leq 0$, a p kiszámításánál már szóba se jöhetne a k, így a további fiaival nem kellene törődni, de nincs is több fia. Az l viszont érdekesebb, mert $l \leq 2$, ami még nagyobb lehet j-nél, s mivel itt is csak ez az egy fiú van, $l = 2$, s $p = l = 2$ Innen $r \leq 2$, s ebből biztos, hogy $r \leq q$, tehát az r pontos értéke már nem is fontos, a többi fiával felesleges foglalkozni (α vágás). A gyakorlat azt mutatja, hogy nagyjából a csúcsok felének a vizsgálatától eltekinthetünk ezt a módszert használva.



5. ábra. α-β vágások

4. Visszalépéses programozás (Back-track)

A megoldás keresésének gyakori módszere a próbálgatás. Ha egy pontban már az összes lehetőséget kipróbáltuk, s egyik sem vezetett eredményre, akkor azt a lépést, mellyel idejutottunk, meg nem történtté kell nyilvánítani, s másfele próbálkozni. Erre a módszerre az egyik jellemző feladat az n vezér feladata, ahol az $n \times n$-es sakktáblára úgy kell felrakni az n vezért, hogy azok ne üthessék egymást. A vezérek pozícióját a T tömbben tároljuk, s megpróbáljuk lerakni az i-dik vezért. Ha ez sikerült, akkor próbáljuk lerakni a következőt is. Ellenkező esetben a vezért továbbtoljuk a következő pozícióra, hátha ott nagyobb sikerrel jár. Ha viszont ezzel leléptünk a tábláról, akkor ezt az utolsó vezért leszedjük a tábláról, s az utolsó előttinek keresünk jobb helyet.

Procedure N-VEZÉR(n)

Input: n a tábla mérete
Eredmény: T tömbben az állás

```
1  i ← 1
2  j ← 1
3  while 0 < i and i ≤ n do
4      T[i] ← j
5      if j ≤ n then
6          üti← HAMIS
7          for k ← 1 to n − 1 do
8              if T[i] = T[k] or |T[i] − T[k]| = i − k then
9                  üti ← IGAZ
10             endif
11         endfor
12         if üti then
13             j ← j + 1
14         else
15             i ← i + 1
16             j ← 1
17         endif
18     endif
19     if j > n then i ← i − 1
20 endw
21 if i > 0 then j ← T[i] + 1
```

XII. fejezet
Pszeudókód

1. Adatok

Algoritmusaink különféle adatokkal dolgoznak. Ezeket az adatokat a számítógép a memóriájában tárolja, s a könnyebb felhasználás érdekében nevekkel hivatkozunk rá. A hivatalos elnevezésük változó. Egy változóhoz nem csupán az azt tároló memóriarész címe, valamint a e memóriarész tartalma, azaz a változó értéke tartozik hozzá, hanem arra is szükségünk van, hogy hogyan, miként értelmezzük ezeket az adatokat, magyarul milyen a típusuk.
Gyakran az egyszerűség kedvéért több, azonos szerkezetű adatot együtt kezelünk, egyben, sorfolyamatosan tároljuk őket. Ha szükség van rájuk, akkor a sorban elfoglalt helyükkel hivatkozunk rájuk. Ezt a számot szokás indexnek nevezni. Magát az adatszerkezetet tömbnek nevezzük. Az A tömb i-dik elemére $A[i]$ névvel hivatkozhatunk, például az első elem $A[1]$. Az A tömbben található elemek számára $A.hossz$ néven hivatkozunk algoritmusainkban, s ennek megfelelően az A tömb utolsó eleme $A[A.hossz]$.
Más esetekben több különböző fajtájú adatot kell együtt kezelni. Például egy személyt jellemez a neve, születési dátuma, lakcíme. (Ezen adatok között van szöveges, számszerű, stb.) Az ilyen adatcsoportot rekordnak nevezik, míg az egyes adatait mezőknek. Az x rekord bal elnevezésű mezőjére $x.bal$ névvel fogunk hivatkozni. A mezőknek természetesen lehetnek további mezői is, így az $x.bal.jobb$ kifejezés úgy értendő, hogy az x rekord bal mezőjének $jobb$ mező értékére kívánunk hivatkozni.

2. Utasítások

A legegyszerűbb utasítás az értékadás. Pszeudokódként a következőképpen írjuk: *változó* $\leftarrow$ *kifejezés*. Az értékadás bal oldalán szereplő változó ezen utasítás végrehajtásakor értékül kapja a jobb oldalon található kifejezés értékét. Ennek megfelelően a $i \leftarrow i + 1$ utasítás értelme az, hogy az i változó értékét eggyel növeljük.
Az algoritmusaink nem tartalmaznak input utasításokat, viszont output utasításokat igen. A **kiír** *"szöveg"* utasítás kiirja az idézőjelek közé írt szöveget. Hasonlóképpen működik az **hiba** *"szöveg"* utasítás is, viszont ezt a hibaüzenetek kiírására használjuk.
A progamszövegekben a `//` mögött írt részek megjegyzésnek számítanak, ezek nincsenek hatással algoritmus futására. A zölddel írt sorok olyan részeket fognak

össze, melyet hosszas utasítássorozatokkal tudnánk megfogalmazni, viszont ez az érthetőséget rontaná.

3. Feltételes szerkezetek

Algoritmusaink két feltételes szerkezetet tartalmaznak:

- **if** *feltétel* **then** *igaz ág*
- **if** *feltétel* **then** *igaz ág* **else** *hamis ág*

A feltétel teljesülése esetén az igaz ágban szereplő utasítás vagy utasítássorozat hajtódik végre. Ha a feltétel nem teljesül, akkor az első esetben nem hajtódik végre semmi utasítás, míg a második esetben a hamis ág utasítása vagy utasításai hajtódnak végre. Ha utasítássorozatot tartalmaz valamely ág, akkor minden egyes utasítást külön sorba szedünk, s beljebbkezdéssel és folytonos vonallal jelöljük az összetartozó utasításokat:

```
if feltétel then
   | utasítás 1
   | ⋮
   | utasítás n
endif
```

4. Ciklusok

Az algoritmusok tulnyomó része igényli adott utasítássorozat többszöri végrehajtását. A legegyszerűbb esetben előre ismert az ismétlések száma. Ezekben az esetekben az alábbi szerkezeteket használjuk:

- **for** *ciklusváltozó* ← *kezdőérték* **to** *végérték* **do** *ciklusmag*
- **for** *ciklusváltozó* ← *kezdőérték* **downto** *végérték* **do** *ciklusmag*

Az első esetben a kezdőérték kisebb vagy egyenlő mint a végérték, második esetben fordítva. Ha ez nem teljesül, a ciklusmag nem hajtódik végre. Ha viszont teljesül, akkor a ciklusváltozó először felveszi a kezdőértéket s ezzel végrehajtódik a ciklusmag, majd eggyel nagyobb értéket, s azzal is végrehajtódik, s így tovább, míg végül a végértékkel is lefut a ciklusmag. Például az alábbi részlet outputja a 2 3 4 5 6 7 lesz.

```
for i ← 2 to 7 do print "i"
```

Hasonlóan a **for** ciklust használtuk a gráfok esetén is. Ebben az esetben ha a **forall the** *ciklusváltozó* ∈ *halmaz* **do** *ciklusmag* szerkezet szerepel, a ciklusváltozó felveszi a halmaz minden elemének az értékét, s azokkal sorra lefuttatja a ciklusmagot. Ugyanez a helyzet a **forall the** (u,v) **in** E **do** szerkezet esetén is, itt

az u, v változók sorra megkapják az élek végpontjainak az értékét minden él esetén. A halmaz elemeinek sorrendje esetleges, ám pár ciklus esetén az algoritmus megköveteli, hogy speciális sorrendben vegyük figyelembe az elemeket. Erre a ciklusok mellett szereplő zölddel írt megjegyzések hívják fel a figyelmet.

Ha előre nem ismert az utasítás vagy utasítássorozat végrehajtásának száma, akkor használhatjuk a **while** *feltétel* **do** *ciklusmag* alakú ciklust, melyben a ciklusmag mindaddig végrehajtódik, amíg a feltétel igaz. Ennek megfelelően a **while** IGAZ **do** *utasítás* ciklus soha nem ér véget, végtelen ciklus lesz.

A **while** ciklus esetén a ciklusmag végrehajtása előtt a feltételnek teljesülnie kell. Ezért is hívják ezt a fajta ciklust előltesztelősnek. A **repeat** *utasítások* **until** *feltétel* szerkezetnél az utasítások végrehajtása után kell megvizsgálni, hogy a feltétel teljesül-e. Ha nem, akkor újra és újra végre kell hajtani az utasításokat, s csak akkor lehet kilépni a ciklusból, ha a feltétel teljesül. Ennek megfelelően a **repeat** *utasítások* **until** HAMIS szerkezet valósítja meg a végtelen ciklust.

Irodalomjegyzék

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest *Algoritmusok*, Műszaki Könyvkiadó, Budapest, 1997

[2] Ivanyos G., Rónyai L., Szabó R. *Algoritmusok*, Műszaki Könyvkiadó, Budapest, 1996

[3] D. E. Knuth *Számítógép programozás művészete 3, Keresés és rendezés*, Műszaki Könyvkiadó, Budapest, 1988

Tárgymutató