# Adatstruktúrák

Király Zoltán

2.72 verzió

2025. június 11.

Legfrissebb, on-line verzió:
`https://zkiraly.web.elte.hu/Adatstrukturak.pdf`

„Elekes György emlékére”

# Tartalomjegyzék

**1. Alapvető adatszerkezetek** **5**
1.1. Adatstruktúrák tervezése . . . 5
1.2. Láncolt lista . . . 5
1.3. Sor és verem . . . 6
1.4. Gráfok . . . 7

**2. Gráfok bejárása: szélességi keresés** **12**
2.1. Összefüggő-e a gráf? . . . 13
2.2. Gráfok komponenseinek meghatározása . . . 13
2.3. Legrövidebb utak . . . 13
2.4. Kétszínezhetőség . . . 16
2.5. Irányított gráfok . . . 16

**3. (Bináris) Kupac** **18**

**4. Minimális költségű feszítőfa, legrövidebb út keresése** **22**
4.1. Kruskal algoritmusa . . . 22
4.2. DISZJUNKT-UNIÓ–HOLVAN adatstruktúrák . . . 24
4.3. Prim algoritmusa . . . 31
4.4. Dijkstra algoritmusa . . . 32
4.5. $d$-edfokú kupacok . . . 34
4.5.1. A Dijkstra és Prim algoritmus lépésszáma $d$-edfokú kupaccal . . . 35

**5. Amortizációs elemzés** **36**
5.1. Potenciál és amortizációs idő . . . 36
5.1.1. Amortizációs idő és a strigulázás kapcsolata . . . 37
5.2. Konvex burok keresése . . . 38

**6. Rafináltabb kupacok** **40**
6.1. Fibonacci kupac . . . 40
6.2. Párosítós kupacok . . . 45
6.2.1. A párosítós kupacok hatékony változata . . . 48
6.2.2. A párosítós kupacok legújabb változatai . . . 48
6.3. Szigorú Fibonacci-kupac . . . 49
6.4. Jobb algoritmus minimális költségű feszítőfára . . . 49
6.5. r-kupacok . . . 50
6.6. Thorup kupaca . . . 54

**7. Szótárak** **55**
7.1. Bináris keresőfa . . . 56
7.1.1. Műveletek általános bináris keresőfában . . . 57
7.1.2. Optimális bináris keresőfa . . . 59
7.2. 2-3 fák . . . 61
7.3. B-fák . . . 64

7.4. Piros-fekete fák . . . 64
7.5. AVL-fák (Adelszon-Velszkij és Landisz) . . . 68
7.6. Önkiegyensúlyozó fák (S-fák; Sleator és Tarjan) . . . 70

**8. Hashelés** **74**
8.1. Klasszikus hash függvények . . . 74
8.2. Ütközések feloldása . . . 75
8.2.1. Láncolt (vödrös) hashelés . . . 75
8.3. Nyílt címzés . . . 77
8.4. Lineáris hashelés . . . 78
8.5. Dupla hashelés . . . 79
8.6. A dupla hashelés Brent-féle változata . . . 80
8.7. Egyenletes hashelés . . . 81
8.8. Univerzális hashelés . . . 82
8.9. Tökéletes hashelés . . . 82
8.9.1. Dinamikus tökéletes hashelés . . . 84
8.10. Kakukk hashelés . . . 84
8.11. Bloom-filter . . . 86
8.11.1. Alkalmazások . . . 87

**9. Az LCA és RMQ feladatok** **88**
9.1. A $\pm 1$-RMQ feladat . . . 89
9.2. Általános RMQ feladat visszavezetése az LCA feladatra . . . 90

**10. A van Emde Boas struktúra** **92**
10.1. Tárhely, megvalósítás, javítás . . . 94
10.2. Újabb megvalósítások . . . 95

**11. Geometriai adatstruktúrák** **98**
11.1. Egy dimenziós feladatok . . . 98
11.2. Két dimenziós feladatok . . . 100
11.2.1. Javítás Kaszkád tárolással . . . 102
11.3. Egy téglalapot metsző szakaszok lekérdezése . . . 102
11.3.1. Kupacos keresőfák . . . 104
11.4. Ferde szakaszok lekérdezése . . . 106
11.4.1. Szakasz-fa . . . 106
11.4.2. Egy függőleges szakaszt metsző ferde szakaszok . . . 108

**12. Dinamikus utak és fák** **110**
12.1. Dinamikus utak . . . 110
12.2. Irányítatlan dinamikus fák . . . 111
12.3. Irányított dinamikus fák . . . 111
12.4. Alkalmazás Dinic algoritmusára . . . 113

# 1. Alapvető adatszerkezetek

## 1.1. Adatstruktúrák tervezése

A programozási feladatok megoldása során először az algoritmus vázlatát készítjük el, majd a szükséges adatstruktúrák definícióját: milyen objektumokat akarunk tárolni, és ezekkel milyen műveleteket akarunk végezni.

**Specifikáció**:

- Milyen objektumokat akarunk tárolni
- Milyen műveleteket akarunk végezni ezeken az objektumokon, ezek a műveletek általában:
    - Üres adatstruktúra létrehozása
    - Elem berakása az adatstruktúrába
    - Speciális műveletek (pl. keresés, a legkisebb kulcsú elem kivétele)

Ezután az algoritmust fokozatosan pontosítjuk, egyre részletesebben kidolgozzuk. A finomítások során készítjük el az eljárásokat, amelyek előállítása során a fenti lépések ismétlődnek meg. Ehhez hasonló módon definiáljuk és finomítjuk az adatstruktúrákat is. Elsősorban az határozza meg, hogy milyen megvalósítást választunk, hogy milyen adatokat akarunk tárolni az adott adatstruktúrában és milyen műveleteket akarunk végezni rajtuk.

A tervezés fontos része az elemzés: minden művelethez kiszámoljuk a végrehajtási idejét, és ennek segítségével az egész algoritmus lépésszámát. Igyekszünk az algoritmus által gyakrabban használt műveletek idejét csökkenteni.

Általában nincs „univerzálisan legjobb" adatstruktúra, a felhasználás határozza meg, hogy melyik lesz a legmegfelelőbb (pl.: nem feltétlenül az az adatstruktúra lesz a legjobb, amelyben a leglassabb művelet elvégzésének ideje minimális). Azt a megvalósítást érdemes választani, ahol az algoritmus által leggyakrabban használt műveletek a leggyorsabban valósíthatók meg. Továbbá, természetesen, a gyakorlatban nem mindig az az adatstruktúra a leghatékonyabb, aminek az elméleti futási ideje a legjobb, az előadás során helyenként ki fogunk térni az ilyen meggondolásokra is.

Ha rendelkezésünkre áll már egy jó adatstruktúra, akkor akár az is előfordulhat, hogy érdemesebb az algoritmuson változtatni.

A gyakorlatban sokszor összetett (egymással kapcsolódó) adatstruktúrákat használunk, ilyenekre is fogunk példákat mutatni.

## 1.2. Láncolt lista

**Tulajdonságai**:

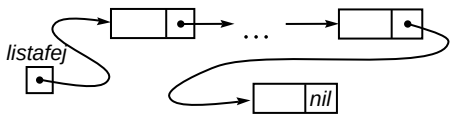
- A lista egy *listafejből* és tetszőleges számú *listaelemből* áll.
- Az elemek a memóriában szétszórtan helyezkedhetnek el.
- Minden elem két részből áll: egy, az adatot, vagy adatokat tartalmazó *adat-részből*, valamint egy mutatóból (*pointerből*), amely a következő elem memóriacímét tartalmazza.

- A lista első elemére a listafej mutat, a lista utolsó elemének mutatója pedig nem mutat sehová (*nil* mutató).

Listával általában a következő műveleteket akarjuk elvégezni: üres-e?, első elem, következő elem (alapműveletek). Ezen kívül: elem keresése, elem beszúrása, elem törlése. A lépésszám az elemi műveleteknél $O(1)$, a többinél általában $O(l)$, ahol $l$ a lista hossza, de pl. az éppen megtalált elem törlése $O(1)$ lépés, és a lista elejére való beszúrás (ha előtte nem kell keresni) is $O(1)$ lépés.

Később fogunk egyéb láncolt lista változatokkal is találkozni, pl.

- néha érdemes a lista utolsó elemére mutató pointert is fenntartani,
- néha érdemes a listában kulcs szerint rendezve tárolni az elemeket,
- sokszor hasznos kétszeresen láncolt lista, ahol egy listaelemhez két pointer tartozik, az egyik a következő, a másik a megelőző listaelemre mutat,
- vagy ciklikusan láncolt lista, ahol az utolsó elem pointere nem *nil*, hanem a legelső elemre mutat,
- illetve ezek kombinációja.




*Házi feladat*: Láncolt listában a keresés lépésszáma legrosszabb esetben a lista hossza. A beszúrás 1 lépés, ha tudjuk, hogy nincs benne; és a lista hossza, ha előtte ellenőrizni kell. A törlés lépésszáma a lista hossza; kivéve, ha a lista kétszeresen láncolt és híváskor a törlendő elemre mutató pointert kaptunk, ekkor $O(1)$ lépés.

## 1.3. Sor és verem

**Tulajdonságai**:

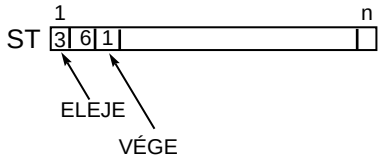
- Tetszőleges típusú objektumok tárolására alkalmas.
- Ha egy **sor**ból kiveszünk egy elemet, akkor a legrégebben berakott elemet kapjuk vissza. A **verem** abban különbözik a sortól, hogy elem-kivétel esetén az utolsónak berakottat kapjuk vissza.

**Műveletek**:

- *Új, üres sor létrehozása (S névvel n méretűt)*: Újsor $(S, n)$
- *Nem-üres-e a sor?*: $S \neq ?\emptyset$
- *Új elem berakása a sorba*: $S \leftarrow v$
- *Elem kivétele a sorból*: $u \leftarrow S$

**Műveletek megvalósítása**: Cél, hogy konstans időben el tudjunk végezni minden műveletet.
A sor méretét meghatározó $n$ érték lehet ismeretlen, ekkor nem tudunk jobbat, mint hogy a sort láncolt listában tároljuk listavég pointerrel (bár az STL könyvtárban pl. vannak változó hosszúságú tömbök, de inkább csak egy kényelmi szolgáltatás, nem konstans, hanem hosszban lineáris lépésszámokkal). Ha ismerünk valami felső becslést a sor „méretére", akkor jobbat is tudunk:

1. Ha $n$ felső becslés az $S \leftarrow v$ műveletek számára:
   ELEJE,VÉGE: pointer




   *Újsor*$(S, n)$:
   Újtömb(ST,n)
   ELEJE:=1; VÉGE:=0

   $S \neq ?\emptyset$: ELEJE $\leq$ VÉGE

   $S \leftarrow v$:
   VÉGE++
   ST(VÉGE):=v

   $u \leftarrow S$:
   u:=ST(ELEJE)
   ELEJE++

2. Ha csak egy $n \geq |S|$ felső becslést tudunk:
   $n + 1$ méretű tömbben ciklikusan oldjuk meg a műveleteket.

*Házi feladat*: Írjuk meg ezt a változatot.

A **verem** megvalósítása hasonló, itt a második eset is könnyen megvalósítható és nem kell az ELEJE mutató (mivel mindig 1).

## 1.4. Gráfok

**Cél**: adatstruktúra egy $V = \{1, 2, \ldots n\}$ csúcshalmazú $m$ élű gráf tárolására. (Ha a gráfunk nem ilyen módon adott, akkor feltesszük, hogy létezik egy olyan jól számolható bijektív függvényünk, ami ilyen csúcshalmazúvá számozza át a gráf csúcsait. Ezen kívül vagy jól számolható az inverze, vagy azt egy $n$ hosszú tömbben tároljuk.)

**1. Szomszédsági mátrix**: $n \times n$-es $A = \{a_{i,j}\}$ mátrixban tároljuk az éleket:

$$a_{i,j} = \begin{cases} 1 & \text{ha } ij \in E \\ 0 & \text{ha } ij \notin E \end{cases}$$

Ha a gráf irányítatlan, akkor a mátrix szimmetrikus lesz. Ekkor a főátló feletti részt elég (lenne) tárolni.

*Tárhely*: $n^2$ bit (kevés programnyelv támogatja a bit mátrixot, ráadásul használatuk sokszor lassabb, ezért a gyakorlatban $n^2$ rövid egészet (vagy `char` típust) használunk, legalábbis ha belefér a memóriába).

**Műveletek**:

- *Új (csupa-0) mátrix létrehozása*: $O(n^2)$,
- $ij \in ?E$: ez itt 1 lépésben megy,
- *Adott u csúcsból kimenő élek felsorolása*: **for** $u\underline{v} \in E$ ...
  Végigmegyünk a mátrix $u$-adik során, ez $O(n)$ lépés.
- *A gráf egy élének behúzása vagy törlése*: $a_{i,j} := 1$ vagy $a_{i,j} := 0$.

**Megjegyzés:** *Multigráf* (olyan gráf, amelyben lehetnek hurokélek is, és a csúcsok közt több párhuzamos él is mehet) esetén:

- $a_{i,j} = k$, ha $i$-ből $j$-be $k$ db párhuzamos él megy
- $a_{i,i} = 2k$, ha az $i$-edik csúcson $k$ darab hurokél van (így az $i$-edik sor összege továbbra is a csúcs fokszáma lesz).

*Súlyozott egyszerű gráfok*:

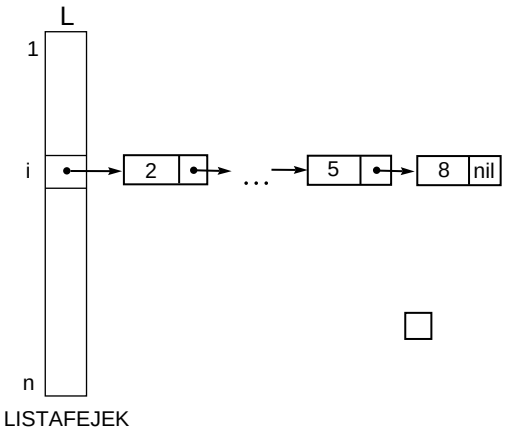
Két esetet különböztetünk meg:

- A 0 súly nem megengedett: Ebben az esetben nincs probléma: a 0 érték azt jelenti, hogy a két csúcs közt nem megy él, a nem nulla érték pedig a két csúcsot összekötő él súlyát jelenti.
- A 0 súly is megengedett: Ekkor több információra van szükségünk: ismernünk kell például a legkisebb súlyt. (Ha például a súlyok nemnegatívak, akkor $-1$ jelentheti azt, hogy az adott csúcsok közt nem megy él.)
- Többféle „súly” is tartozik egy élhez: több mátrixot használunk.

**Megjegyzés:** Ekkor persze a tár súly-típusonként $n^2 \log U$, ahol $U$ a max súly.

**Megjegyzés:** Súlyozott multigráfokat így nem tudunk tárolni.

**2. Éllista**: Egy $L[1:n]$ tömbben tároljuk a csúcsokból kiinduló éleket felsoroló láncolt listák listafejeit.




Az $i$. csúcsból legalább 3 él megy ki: pl. a 2-es,..., az 5-ös és a 8-as számú csúcsba.

*Tárhely*: $n+m$ mutató (irányítatlan esetben $n+2m$, mivel minden él kétszer szerepel) és $m$ (ill. $2m$) szám.

*Műveletek*:

- *Üres struktúra létrehozása*: A listafej-tömböt létrehozzuk és *nil*-ekkel töltjük fel: $O(n)$.
- $ij \in ?E$: Legrosszabb esetben végig kell menni a láncolt listán. Ez $d(i) \leq n-1$ (egyszerű gráf esetén, különben $d(i) > n$ is lehet) lépést jelent. (Irányított gráf esetén a felső korlát az adott csúcsra a $d_{\text{ki}}(i)$ kifok.)
- *Adott $i$ csúcsból kimenő élek felsorolása*: A lépésszám ebben az esetben is $d(i)$ vagy $d_{\text{ki}}(i)$.
- *Új él beszúrása* ($ij$ irányított él beszúrása):
  $p := L(i)$; $L(i) :=$ beszúrt új $(j, p)$ listaelem címe a memóriában.
  (Érdemes mindig a láncolt lista elejére beszúrni.) Lépésszám: $O(1)$.
- *$ij$ él törlése*: Az $i$-edik és a $j$-edik listában meg kell keresni, ez $d(i)+d(j)$ lépés, utána kitörölni már konstans.

**Megjegyzések.**

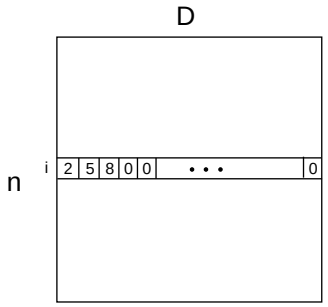
- Az élfelsorolás (általában a leggyakrabban használt) művelet ideje kis fokszám esetén sokkal jobb, mint a szomszédsági mátrixnál.
- Megfordított $\overleftarrow{G}$ gráf generálása: végigmegyünk az éleken és egy új éllistába átmásoljuk a fordítottjukat, ez $O(m)$ időben megoldható.
- Irányított gráfból a megfelelő irányítatlan $\overrightarrow{\overleftarrow{G}}$ gráf generálása: végigmegyünk az éleken és egy új éllistába átmásoljuk az éleket is és a fordítottjukat is, ez $O(m)$ időben megoldható.
- Multi- és súlyozott gráfok esetén: A láncolt listában minden gond nélkül tárolható. Mivel egy listaelem adatrésze akárhány adatból állhat, még az se gond, ha az élekhez több szám (kapacitás, súly és hossz) is tartozik.

- Irányítatlan gráf esetén, ill. irányítottnál, ha a fordított élekre is szüksége van az algoritmusnak, akkor egy $uv$ él szerepel az $u$ csúcs és a $v$ csúcs éllistáján is. Sokszor hasznos minden listaelembe egy új pointert felvenni, és az ilyen párokat egymásra linkelni. Pl. ha $u$ listáján megtaláltuk az élet, és mondjuk a hosszát változtatni szeretnénk, akkor persze a $v$ listáján a párjának hosszát is változtatni kell.

Input formátumként még használatos az ömlesztett éllista: az első sor az $n$ csúcsszámot tartalmazza, utána a gráf élei tetszőleges sorrendben vannak felsorolva, egy sorban egy él kezdőcsúcsa és végcsúcsa, utána esetleg az adott él súlya, hossza, költsége.

**3. Mátrixban tárolt éllista** (egyszerű gráfokra)

Egy $n \times D$ méretű mátrixot használunk. A mátrix $i$-edik sorában felsoroljuk az $i$-edik csúcsból kiinduló élek végpontjait. Ha $k$ darab él megy ki, akkor a mátrix $i$-edik sorában a $k+1$-edik oszloptól a $D$-edik oszlopig nullák szerepelnek.




Az $i$. csúcsból 3 él megy ki: a 2-es, az 5-ös és a 8-as számú csúcsba.

Ha az élek súlyozottak, akkor a súlyokat egy másik mátrixban tároljuk, pl. az $i5$ él súlyát $S(i,2)$ tartalmazza.

$D$ **megválasztása**:

- Ha a fokszámokról semmit nem tudunk: $D := n$.
- Ha $\Delta =$ maxfok $(G)$ ismert, akkor $D := \Delta + 1$ (azért érdemes $\Delta + 1$-et választani, hogy minden sor 0-val végződjön, így az $i$. csúcsból akkor nem indul ki több él, ha az $i$. soron lépkedve egy 0-hoz érünk).

Nyereség: Nem kell pointereket használni.
Veszteség: A tárhely nőhetett is. A tárhely mérete $n \times$ (maxfok+1), az $n\times$ átlagfok helyett – és ez még a jobbik eset, amikor ismerjük $\Delta$-t. Ha $\Delta$-ról nem tudunk semmit, akkor $n^2$ hely kell.

**Műveletek ideje**: Az alapműveleteknél ugyanaz, mint az éllista esetén, sőt, egy kicsit még gyorsabb is, mivel nem kell pointerekkel dolgoznunk. A beszúrás és a törlés lehet lassabb (ha a beszúrás fontos, egy külön tömbben tárolhatjuk az első 0 indexét).
Összefoglalva: ez a megoldás a gyakorlatban egy jó megoldásnak tekinthető, ha vagy ismerünk egy $n$-nél lényegesen kisebb felső korlátot $\Delta$-ra, vagy ha a tárfoglalás nem nagyon érdekel minket (hiszen az egy csúcsból kiinduló éleket itt is fel tudjuk sorolni fokszám időben).

Nem reguláris gráfokra, ha az algoritmus során nem változnak, jobban megéri az alábbi változat: a mátrix sorai végéről vágjuk le a 0-kat, majd a sorokat fűzzük egymás után egy $T$ tömbbe. Közben egy $K[1:n]$ tömbbe felírjuk, hogy melyik csúcs éllistája hol kezdődik, azaz $K(1) = 1,\ K(2) = d(1) + 1,\ K(3) = d(1) + d(2) + 1, \ldots$. A **for** $u\underline{v} \in E \ldots$ ciklus ekkor így néz ki: **for** $i = K(u)..K(u+1) - 1 \quad v := T(i); \ldots$. A tárhely pl. irányított gráfok esetén $m + n$-re csökkent.

A feladattól függően sokszor megéri, ha az akármilyen formátumban (pl. szomszédsági mátrixban) kapott gráfot az algoritmus elején ilyenné alakítjuk. Főleg, ha a gráfnak lényegesen kevesebb, mint $\binom{n}{2}$ éle van, és ráadásul egy-egy csúcs éllistáján az algoritmus során sokszor kell végigmenni.

**Speciális esetek**
Páros gráfok tárolása szomszédsági mátrix-szal: elég $\frac{n^2}{4}$ hely (az alsó halmaznak megfelelő sorok és a felsőnek megfelelő oszlopok).
Fák tárolása:

- Éllista
- Prüfer kód: Egy $n$ csúcsú gráf esetén $n-2$ darab szám (melyek 1 és $n$ közöttiek). Ez a tárolás nagyon alkalmas véletlen fák generálására (ami egyébként nem egyszerű feladat).
- Pointerekkel (gyökeres fáknál alkalmazzuk, ahol a felfelé menő élet meg akarjuk különböztetni a többitől; lásd később). Ha a gyerekek számára nincs jó felső korlát, akkor a csúcsban az első gyerekre mutat csak pointer, és a gyerekek egy láncolt listában vannak.
- Kupacoknál (teljesen kiegyensúlyozott fák): tömbben.

# 2. Gráfok bejárása: szélességi keresés

Szélességi keresés esetén egy adott pillanatban minden egyes csúcs 3 féle állapotban lehet:

*a*) Az algoritmus során eddig még nem látott

*b*) Látott, de még nem átvizsgált

*c*) Látott és átvizsgált

Egy $u$ csúcs átvizsgálása azt jelenti, hogy végigmegyünk a belőle kimenő éleken, és ha *a*) típusú csúcsot találunk, azt átsoroljuk *b*) típusúvá; végül $u$-t magát pedig *c*) típusúvá.

A szélességi keresésnél a *b*) állapotú csúcsok közül a legrégebbit vizsgáljuk át.

Egy $L$ bit-tömböt ($L(i) = 1 \Leftrightarrow$ ha $i$-t már láttuk az algoritmus során) és egy $S$ sort (ez tartalmazza a *b*) állapotú csúcsokat) használunk.

*Az algoritmus*:

INIT:

Újsor $(S, n)$; Újtömb $(L, n)$

**for** $i := 1 \ldots n \quad L(i) := 0$

$L(1) := 1; \quad S \leftarrow 1 \quad$ /* *látott, de nem átvizsgált*

**while** $S \neq \emptyset$

$u \leftarrow S$

**for** $u\underline{v} \in E$

**if** $L(v) = 0$ **then** $L(v) := 1; \quad S \leftarrow v$

Ez az algoritmus magában nem ad outputot, csak végiglátogatja a gráf 1-es csúcsából elérhető csúcsait.

*Lépésszám*: Egy csúcsot maximum egyszer rakunk be a sorba, így maximum $n$-szer tudunk kivenni elemet a sorból. Amennyiben szomszédsági mátrixban tároljuk a gráfot, akkor a lépésszám $O(n^2)$.

**1. Tétel.** *Nincs olyan algoritmus, amely tetszőleges, szomszédsági mátrixszal adott gráfról kevesebb, mint* $\binom{n}{2}$ *időben eldönti, hogy összefüggő-e, tehát a szélességi keresés lényegében optimális lépésszámú szomszédsági mátrix esetén.*

Ezt itt nem bizonyítjuk, ellenség-stratégia módszerrel nem túlságosan nehéz.

Éllistás tárolás esetén a **for** ciklus $d(u)$-szor fut le. A futási időt ekkor nem érdemes $n \times \max d(u)$-val becsülni, hanem $\sum_u (1 + d(u)) = n + 2m$-mel becsüljük. Így a futási idő $O(n + m)$ lesz, ami izolált csúcsot nem tartalmazó gráf esetén $O(m)$.

**2. Tétel.** *Ennél lényegesen gyorsabban éllistás tárolásnál sem valósítható meg a gráfbejárás feladata.*

**Bizonyítás:** azt állítjuk, hogy kell legalább $|E|/2$ input olvasás.

Minden élszámra tudunk olyan gráfot konstruálni, hogy ennél kevesebb olvasással ne legyen eldönthető a gráf összefüggősége. Legyen $G_1$ és $G_2$ két egyforma, tetszőleges összefüggő gráf, az inputnak a két komponense. Ha nem olvassuk végig legalább az egyiknek az összes élét, akkor $G_1$ és $G_2$ egy-egy nem végigvizsgált csúcsa közé berakhatunk egy új élet (az éllistáik végére), ezt az algoritmus nem veszi észre.

Az alábbiakban megnézünk néhány fontos feladatot, amelyeket a szélességi keresés segítségével könnyen megoldhatunk.

## 2.1. Összefüggő-e a gráf?

Ha minden $1 \leq i \leq n$-re $L(i) = 1$, akkor a gráf összefüggő, különben nem.

## 2.2. Gráfok komponenseinek meghatározása

A gráf egy adott komponense egy ekvivalencia osztály, ezt úgy tároljuk, hogy tetszőlegesen elnevezzük a komponenseket, és egy csúcsra az őt tartalmazó komponens sorszámát tároljuk (két csúcs akkor van ugyanabban a komponensben, ha ez a szám ugyanaz). Nyilvánvaló megoldás: először az $L$ tömbben 1-es értéket adunk minden látott csúcsnak, majd amikor a sor kiürül, akkor keresünk egy olyan w csúcsot, melyre $L(w) = 0$, és egy innen indított újabb kereséssel megkeressük a 2. komponens csúcsait, ezekre $L$ értékét 2-re állítjuk, majd így tovább.

Vegyük észre, hogy akár $c \cdot n^2$ időt is elhasználhatunk kizárólag a 0-k keresésére: Például, ha a $G$ gráf nem tartalmaz éleket, akkor az első komponens után 2 lépés az első 0 megtalálása, a második után 3, és így tovább, az utolsó 0 megtalálása $n$ lépés.

Hasznos észrevétel: A csúcsok „látottsága" csak egy irányba változik. Ha az első bejárásnál az $i$-edik csúcsnál találtunk 0-t, legközelebb elég az $i + 1$-edik elemtől kezdeni a következő nullás keresését.

*Az algoritmus*:

INIT:
    Újsor $(S, n)$; Újtömb $(L, n)$
    **for** $i = 1..n$  $L(i) := 0$
    $k := 0$
**for** $i = 1..n$
    **if** $L(i) = 0$ **then** $k{+}{+}$; $L(i) := k$; $S \leftarrow i$
    **while** $S \neq \emptyset$
        $u \leftarrow S$
        **for** $u\underline{v} \in E$
            **if** $L(v) = 0$ **then** $L(v) := k$; $S \leftarrow v$

*Lépésszám*(éllistás tárolásnál): $O(n + m)$.

## 2.3. Legrövidebb utak

Feljegyezzük, hogy az egyes csúcsokat mely csúcsból láttuk meg (a $v$ csúcsot a $p(v)$ nevű *szülőből*), és minden csúcshoz hozzárendelünk egy $SZ(v)$ szintet.

*Az algoritmus*:

INIT:
    Újsor $(S, n)$; Újtömb $(L, n)$; Újtömb $(p, n)$; Újtömb $(SZ, n)$
    **for** $i = 1..n$
        $L(i) := 0$; $p(i) := i$; $SZ(i) := 0$

$k := 0$
**for** $i = 1..n$
  **if** $L(i) = 0$ **then** $k{+}{+}$; $L(i) := k$; $S \leftarrow i$
  **while** $S \neq \emptyset$
    $u \leftarrow S$
    **for** $u\underline{v} \in E$
      **if** $L(v) = 0$ **then** $L(v) := k$; $S \leftarrow v$; $p(v) := u$
        $SZ(v) := SZ(u) + 1$

**1. Állítás.** *Szélességi keresés esetén*

1. *Bármelyik időpontban az $S$ sorban balról jobbra a szintszám monoton nő*
2. *Ha $x, y \in S$, akkor $|SZ(x) - SZ(y)| \leq 1$*

**Bizonyítás:** Időre vonatkozó indukcióval.

- Amikor egy elem van a sorban, triviálisan teljesülnek.
- Amikor új elemet rakunk be, elromlanak-e a tulajdonságok? Tegyük fel, hogy $v$ berakásakor valamelyik elromlik, legyen $v$ szülője $u$, tehát $SZ(u) = SZ(v) - 1$. Vegyük észre, hogy a sor minden $x$ elemére, ha $SZ(x) \neq SZ(v)$, akkor $x$ már a sorban volt $u$ kivétele előtt is.

1. tulajdonság:

| | u | ... | x | ... | v |
|---|---|---|---|---|---|
| SZ: | 3 | | 5 | | 4 |

Ekkor $x$ a $v$ betétele előtt is benne volt a sorban, ami ellentmond annak, hogy $u$ kivétele előtt teljesült a 2. tulajdonság.

2. tulajdonság:

| | u | ... | x | ... | v |
|---|---|---|---|---|---|
| SZ: | 4 | | 3 | | 5 |

Ekkor is $x$ a $v$ betétele előtt is benne volt a sorban, ami ellentmond annak, hogy $u$ kivétele előtt teljesült az 1. tulajdonság.

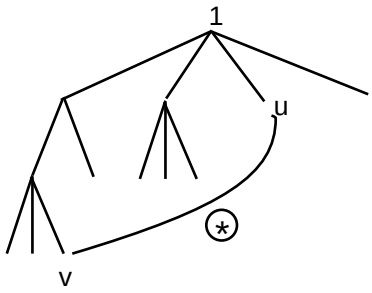
**3. Tétel.** *Szélességi keresésnél (összefüggő irányítatlan gráf esetén)*

1. *A gráf élei nem ugorhatnak át szintet, vagyis, ha $uv \in E$, akkor $|SZ(u) - SZ(v)| \leq 1$*
2. *Minden $x$ csúcsra $SZ(x)$ az 1 csúcsból az $x$-be vezető legrövidebb út hossza.*

3. *Az egyik legrövidebb út:* $(1, \ldots, p(p(x)), p(x), x)$

4. *Minden csúcsot bejárunk az algoritmus során, vagyis jó az algoritmus.*

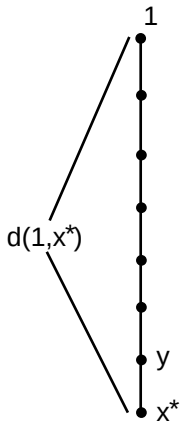
**Bizonyítás:** (Előző tétel alapján)




1. Nem lehet $uv$ típusú (lásd az ábrán) él a gráfban, mivel amikor $u$-t kivettük a sorból, $v$ még nem lehetett benne a sorban, így $u$ átvizsgálása során be kellett kerülnie eggyel nagyobb szintszámmal.

2. és 3. A 3. állításban felírt (jobbról balra generált) csúcssorozat valóban egy út lesz, és az 1. állítás miatt egy legrövidebb út is, mivel a gráf élei nem tudnak szintet ugrani. A hossza nyilván $SZ(x)$.

4. Jelölje $d(1, x)$ egy $x$ csúcs távolságát az 1-től. Indirekten tegyük fel, hogy létezik olyan csúcs, ami 1-ből elérhető, és ahol nem jártunk. Legyen $x^*$ az ilyenek közül egy olyan, amelyre $d(1, x^*)$ minimális.




Jelölje az 1-ből $x^*$-ba vezető $d(1, x^*)$ hosszú út utolsó előtti csúcsát $y$, ekkor $y$ közelebb van 1-hez, így $x^*$ választása szerint $y$-ban jártunk, ezért valamikor $y$ szerepelt a sorban. Amikor megvizsgáltuk a szomszédait, akkor $x^*$-ot is megtaláltuk volna.

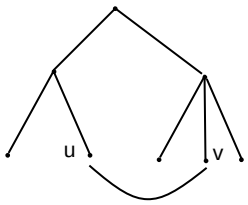
**2. Állítás.** *$\{(p(u), u)|p(u) \neq u\} \subseteq E(G)$ egy feszítőfa lesz. Ha a gráf irányított, akkor a gyökértől elfelé irányított fákat (fenyőket) kapunk.*

**Bizonyítás:** Ez $n-1$ darab él, ha nem feszítőfa lenne, akkor lenne benne kör, ez azonban nem lehet, mert a kör csúcsai közül a sorba leghamarabb berakottnak a szülője nem lehet egy később berakott.

## 2.4. Kétszínezhetőség

Tudjuk (de most újra be is bizonyítjuk), hogy egy gráf akkor és csak akkor nem kétszínezhető, ha van benne páratlan kör. Színezzük a gráf csúcsait az alábbi módon: Legyen piros, ha $SZ$ értéke páros és kék, ha $SZ$ páratlan.

Mikor nem jó ez a színezés? Felhasználjuk, hogy a gráf élei nem ugorhatnak át szintet. Probléma csak az egy szinten belül menő élekkel lehet. Legyen egy ilyen szinten belüli él $u$ és $v$ között.




Az algoritmust egészítsük ki az **else if** $SZ(u) = SZ(v)$ **then PTLANKÖR** $(u, v)$ sorral.

**PTLANKÖR** $(u, v)$: megkeressük $u$ és $v$ legközelebbi közös ősét. Az így meghatározott kör valóban páratlan, így a gráf nem kétszínezhető.

*A kör kiíratása*: Felváltva a szülőre lépkedve keressük meg az $x$ legközelebbi közös őst, és közben rakjuk $v, p(v), \ldots, p(\ldots(p(v))\ldots), \ldots x$-et egy verembe,
$u, p(u), \ldots, p(\ldots(p(u))\ldots), \ldots x$-et pedig egy sorba. Ha kiürítjük a vermet, majd a sort, akkor az adott kör csúcsait soroljuk fel, a sor utolsó elemét már nem íratjuk ki.

## 2.5. Irányított gráfok

Minden csúcsot elérünk, amibe van 1-ből irányított út. Ekkor is a legrövidebb ilyet találjuk meg. Ebben az esetben a gráf élei visszafelé már átugorhatnak szintet, lefelé ugyanúgy nem.

Irányított gráfok esetén kétféle összefüggőséget is definiálhatunk:

- *Gyenge összefüggőség*: Ha eltekintünk az irányítástól, akkor összefüggő-e a gráf (azaz, ha például a gráfok élei közlekedési utakat jelölnek, eljuthatunk-e mindenhonnan mindenhová egy biciklin)?

  Ekkor lényegében az a kérdés, hogy $\overrightarrow{\overleftarrow{G}}$ összefüggő-e.

- *Erős összefüggőség*: Bármely $x, y \in V$ esetén létezik-e $x$-ből $y$-ba menő irányított út?

  Triviális megoldás: $n$ darab szélességi keresés (minden csúcsból indítunk egyet).

  Jobb megoldás: Csinálunk egy szélességi keresést az eredeti gráfban, $\overrightarrow{G}$-n 1-ből, majd még egy szélességi keresést a fordított $\overleftarrow{G}$ gráfban szintén 1-ből. Így összességében 2 szélességi keresés elegendő az erős összefüggőség eldöntéséhez, mivel $G$ erősen összefüggő akkor és csak akkor, ha mindkét keresésben minden csúcs látottá válik.

# 3. (Bináris) Kupac

Egy fa *kupacrendezett*, ha igazak rá a következő tulajdonságok:

- Gyökeres fa, melynek csúcsaiban rekordok helyezkednek el.
- Minden rekordnak van egy kitüntetett mezője, a $K$ KULCS.
- Bármely $v$-re, ha létezik a $p(v)$ szülő, akkor $K(rek(p(v))) \leq K(rek(v))$ (ezt ezentúl $K(v)$-vel jelöljük), azaz $v$ leszármazottaira igaz, hogy az ott található kulcsok minimuma van $v$-ben. (Egy csúcsot mindig saját maga leszármazottjának tekintünk.)

*Kupac:* (egyszerű kupac, bináris kupac)

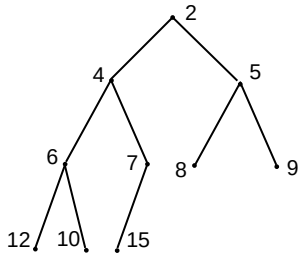
- Kupacrendezett bináris fa (egy csúcs gyermekeinek száma $0, 1$ vagy $2$)
- A levelek vagy egy, vagy két szomszédos szinten vannak, ez utóbbi esetben igaz, hogy a felső tele van, az alsón pedig balra vannak rendezve a levelek. (Tehát a bináris fában a szinteket mindig balról jobbra haladva töltjük fel.)

*Tárolás*: a kulcsokat egy $A$ tömbben tároljuk:

- $A(1)$ a gyökér kulcsa.
- Az $i$. csúcs bal fiának kulcsa $A(2i)$-ben lesz, jobb fiáé pedig $A(2i+1)$-ben.
- Az $i$. csúcs szülőjének kulcsa $A(\lfloor i/2 \rfloor)$.
- Ezenkívül még tárolunk egy VÉGE pointert (számot), mely megadja, hogy pillanatnyilag hány elem van a kupacban.

**Megjegyzés:** Igazából legtöbbször 3 tömbre van szükségünk. Az $A$-ban tároljuk a csúcsok kulcsait, a $B$ tömbben vagy az $i$-edik csúcsban tárolt objektum nevét, vagy a megfelelő rekordra mutató pointert, és $C(j)$ pedig a $j$ nevű csúcs (ill. rekord) indexe az $A$-ban és $B$-ben. Általában csak az $A$ tömbbel való operációt adjuk meg, a többi kitalálható. De példaként a FELBILLEGTET eljárásban megadjuk ezeket is. Persze ha a nevek nem $1$ és $n$ közötti számok, akkor a $C$ tömb helyett egy szótárra van szükségünk, lásd később.

Példa:




**A:**

| 2 | 4 | 5 | 6 | 7 | 8 | 9 | 12 | 10 | 15 | |
|---|---|---|---|---|---|---|---|---|---|---|

*Műveletek*:

*Újkupac($A, n$)*: egy új, üres kupacot hoz létre ($n$ az egy időben a kupacban szereplő elemek maximális száma).

*Beszúr(A, új)*: egy új elemet szúr be a kupacba.
*Mintörlés(A)*: a minimális kulcsú elemet kiveszi a kupacból és visszaadja.
*Kulcs-csökk(A, elem, $\Delta$)*: a kupacban lévő *elem* kulcsát csökkenti $\Delta$-val. (Csak $\Delta \geq 0$ esetén kell működnie). Az *elem*-et úgy határozzuk meg, hogy rámutatunk egy pointerrel (nem tudjuk megkeresni, mivel a kupacban nem tudunk hatékonyan keresni), azaz a $C(elem)$-edik kupacelem kulcsát csökkentjük.

*Műveletek megvalósítása:*

*Újkupac($A, n$)*:
Újtömb($A, n$); VÉGE:= 0

*Beszúr(A, új)*:
Ötlet: nyilván az $A$ tömb első üres helyére kell beraknunk az új elemet, ez meghatározza, hogy a fában hová kerül az új levél. A kupactulajdonság csak egy él mentén romolhatott el (az új levél és a szülője közöttin). Ezen él mentén javítjuk a kupactulajdonságot, ekkor felette egy másik él mentén romolhat el, így ismételjük, amíg mindenhol helyre nem áll (lehet, hogy egészen a gyökérig el kell mennünk).

*Beszúr(A, új)*:
VÉGE++
$A$(VÉGE) :=K (új)
$B$(VÉGE) :=új; $C$(új) :=VÉGE
FELBILLEGTET($A$,VÉGE)

*FELBILLEGTET($A, i$):*
$AA := A(i)$; $BB := B(i)$
**while** $i > 1$ && $A(\lfloor i/2 \rfloor) > AA$
$A(i) := A(\lfloor i/2 \rfloor)$
$B(i) := B(\lfloor i/2 \rfloor)$
$C(B(i)) := i$
$i := \lfloor i/2 \rfloor$
$A(i) := AA$
$B(i) := BB$
$C(BB) := i$

*Lépésszám*: A fa mélysége $\lfloor \log n \rfloor$, amiből következik, hogy a FELBILLEGTET lépésszáma $O(\log n)$.

**3. Állítás.** *Beszúrás után a kupactulajdonság helyreáll.*

**Bizonyítás:** Úgy tekintjük, hogy mindig $i$ és szülője tartalmát felcseréljük egy-egy lépésben. Állítás: mindig csak $i$ és a szülője közti él mentén lehet baj a kupactulajdonsággal. Ez kezdetben igaz. Ha $i$ gyökér, vagy szülőjének kulcsa kisebb-egyenlő, akkor itt sincs baj, tehát kész vagyunk. Különben csere után ezen él mentén a kupactulajdonság helyreáll, és mivel a szülőben lévő kulcs csökken, a másik gyereke felé menő él sem romlik el, tehát csak az új $i$ és szülője közötti élen lehet baj.

*Mintörlés(A)*: Kicseréljük a gyökeret és az utolsó elemet, majd töröljük az utolsó elemet. Itt két élen is elromolhatott a kupactulajdonság. Ha cserélni kell, akkor a két gyerek közül mindig a kisebb kulcsúval cserélünk.

*Mintörlés* $(A)$*:*
  **csere** $(A(1), A(\text{VÉGE}))$; VÉGE$--$
  LEBILLEGTET $(A, 1)$
  **return**$(A(\text{VÉGE}+1))$

*LEBILLEGTET*$(A, i)$*:*
  $AA := A(i)$; $j := 2i + 1$  /* *j a jobb gyerek*
  **while** $j \leq$ VÉGE
    **if** $A(j-1) < A(j)$ **then** $j--$  /* *átállítjuk j-t a bal gyerekre*
    **if** $A(j) < AA$ **then** $A(i) := A(j)$; $i := j$; $j := 2i + 1$
      **else** $j :=$VÉGE$+2$  /* *kilépünk, készen vagyunk*
  $j--$  /* *még kezelnünk kell azt az esetet, ha i-nek csak bal gyereke van*
  **if** $j \leq$VÉGE && $A(j) < AA$ **then** $A(i) := A(j)$; $i := j$
  $A(i) := AA$

**4. Állítás.** *Mintörlés után a kupactulajdonság helyreáll.*

**Bizonyítás:** Úgy tekintjük, hogy mindig $i$ és kisebbik kulcsú gyereke tartalmát felcseréljük egy-egy lépésben. Állítás: mindig csak az aktuális $i$ és a gyerekei közti élek mentén lehet baj a kupactulajdonsággal. Ez kezdetben igaz. Ha $i$-nek nincs gyereke, vagy mindkét gyerekének kulcsa nagyobb-egyenlő, akkor itt sincs baj, tehát kész vagyunk. Különben csere után ezen élek mentén a kupactulajdonság helyreáll, és csak az új $i$ és gyerekei között romolhat el.

*Lépésszám*: $O(\log n)$

*Kulcs-csökk(A, elem, $\Delta$)*:
  $i := C(elem)$
  $A(i) := A(i) - \Delta$
  FELBILLEGTET$(A, i)$

*Lépésszám*: $O(\log n)$

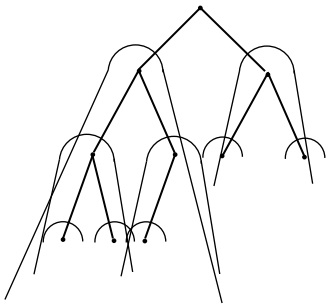
**Megjegyzés:** A LEBILLEGTET és FELBILLEGTET „belső" műveletek. Ez azt jelenti, hogy kívülről nem hívjuk meg ezeket, a kupacműveletek használják csak őket.

**Alkalmazások:**

Rendezés: üres kupacba sorban Beszúrjuk az elemeket, majd sorban Mintörlünk, ez $O(n \cdot \log n)$ lépés. Az alábbi, lineáris idejű Kupacépítés segítségével kicsit jobb lesz. Ezt főleg olyan esetekben érdemes használni, amikor a feladat például az első $k$ legkisebb elem kiíratása (nagyság szerinti sorrendben), ekkor a futási idő $O(n + k \cdot \log n)$ lesz.

*KUPACÉPÍTÉS* (nem alapművelet): $n$ darab elem kupacba szervezése.

Tetszőlegesen feltöltjük a tömböt, majd alulról felfelé rakjuk rendbe a kupacot. A levelek jó egyelemű kupacok. Az egy magasságú részfák egy lebillentéssel kupacrendezetté alakíthatóak. Általában, ha $v$ gyerekeinek részfái már kupacrendezettek, akkor $v$ LEBILLEGTETésével $v$ részfája is azzá válik.



*Az algoritmus:*
az adatok betöltése tetszőlegesen az $A$ tömbbe.
**for** $i = \lceil n/2 \rceil..1$ $(-1)$
LEBILLEGTET$(A, i)$

*Lépésszám*: (összes lebillentések száma)=
$\frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \cdots + 1 \cdot \lceil \log n \rceil \leq n \cdot (\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \dots)$
$a := (\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \dots)$, könnyen láthatóan ez a sor abszolút konvergens (majorálja a $\sum \left(\frac{3}{4}\right)^i$ mértani sor), ezért az $a$ szám létezik. Ekkor $2a - a = (\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \dots) - (\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \dots) = (\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots) = 1 \Longrightarrow \underline{a = 1}$.
Azaz legfeljebb $n$ darab lebillentés kell, így $O(n)$ elemi lépés lesz a lépésszám.

*Következmény*: Ha $n$ elemből a $k$ darab legkisebbet keressük rendezve, akkor ennek a lépésszáma $O(n + k \cdot \log n)$ lesz.
További alkalmazások találhatók a következő fejezetekben.

# 4. Minimális költségű feszítőfa, legrövidebb út keresése

*Adott*: $G = (V, E)$ összefüggő, irányítatlan gráf $c : E \to \mathbb{R}$ élköltségekkel.
*Feladat*: Olyan feszítőfát keresünk, melyre $c(T) = \sum_{e \in T} c(e)$ minimális.

## 4.1. Kruskal algoritmusa

**I. FÁZIS**: rendezzük az éleket költség szerint: $c(e_1) \leq c(e_2) \leq c(e_3) \leq \cdots \leq c(e_m)$. Ez $O(m \cdot \log n^2) = O(m \cdot \log n)$ időben megy, ahol $m = |E|$ és $n = |V|$.

**II. FÁZIS**: nyilvántartjuk a komponenseket: $K = \{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\}$

$T := \emptyset$
**for** $i = 1..m$
    **if not** $e_i$ két vége egy komponensben
        **then** $T := T + e_i$ ; két komponens uniója.

**Megjegyzés:** persze ha azt is figyeljük, hogy $T$-ben mikor lesz $n-1$ él, akkor már meg is állhatunk.

Naiv megoldás a komponensek nyilvántartására: minden él megépítésekor ($T$-be való bevételekor) lefuttatunk egy szélességi keresést (komponensekre bontást) a $T$ gráfon. Ez összesen $O(n^2)$ lépés. Ezen majd javítani szeretnénk.

**4. Tétel.** *Ez az algoritmus valóban minimális költségű feszítőfát ad.*

**Bizonyítás:**

| | $e_1$ | $e_2$ | $e_3$ | ... |
|---|---|---|---|---|
| T: | + | - | + | ... |

+/- jeleket teszünk aszerint, hogy az adott él benne van-e $T$-ben ($T$ az algoritmus által adott fa).

Legyen $T^*$ olyan optimális fa, melyre a $+/-$ sorozat a lehető leghosszabban megegyezik $T$-ével. Legyen az első különbség az $i$. oszlopban.

1. eset $(-/+)$:

| | ... | $e_i$ | ... |
|---|---|---|---|
| T: | ... | - | ... |
| T*: | ... | + | ... |

Ez nem lehetséges, mivel ha a $T$-be az algoritmus során nem vettük be az $e_i$ élet, akkor $e_i$ két végpontja össze van kötve kisebb sorszámú élekkel, azonban: $j < i$-re $e_j \in T$, tehát $e_j \in T^*$. Ekkor $e_i$ már a $T^*$ eddigi éleivel is kört alkot, ez pedig ellentmondás.

2. eset (+/−):

| | ... | $e_i$ | ... |
|---|---|---|---|
| T: | ... | + | ... |
| T*: | ... | - | ... |

Mi lesz $T^* + e_i$? Mivel $T^*$ feszítőfa, ebben lesz pontosan egy $C$ kör. Legyen $e_j$ a maximális indexű éle a $C$-nek.

**5. Állítás.** $j > i$.

**Bizonyítás:** különben $T$-ben is ott lenne a $C$ kör, hiszen $T$ az $i$-nél kisebb indexű éleken megegyezik $T^*$-gal.

*Következmény*: $c(e_j) \geq c(e_i)$

$T^* + e_i - e_j$ feszítőfa lesz (hiszen az egyetlen körből hagyunk el egy élt). $c(T^* + e_i - e_j) \leq c(T^*) = \text{OPT}$. Tehát $T^* + e_i - e_j$ is optimális megoldás, de ez ellentmondás, mert ez már az $i$. oszlopban is megegyezik $T$-vel.

*Következmények*:

1. Ha $G$-ben bármely két él költsége különbözik, akkor egyértelmű az optimális feszítőfa.

2. Ha az összes lehetséges jó (költség szerint monoton növő) él-rendezést vesszük, majd ezekre lefuttatjuk az algoritmust, akkor megkapjuk az összes optimális (minimális költségű) feszítőfát.

**Bizonyítás:** Adott $T^*$ optimális fához a sorrend megválasztása:

Az azonos költségűeket úgy rakjuk sorba, hogy a $T^*$-ban szereplő élek legyenek elöl. Az előző bizonyítást végiggondolva könnyű látni, hogy ekkor az algoritmus éppen $T^*$-ot adja.

A **II. FÁZIS** megvalósítása: DISZJUNKT-UNIÓ és HOLVAN műveletekkel.

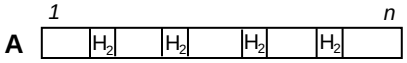
HOLVAN $(x)$:= az $x$-et tartalmazó halmaz neve (ez egyértelmű, mivel a halmazok diszjunktak).

D-UNIÓ$(H_1, H_2)$: $H := H_1 \cup H_2$, azaz lecseréljük $H_1$-et és $H_2$-t egy új, $H$ nevű halmazra (általában nem keresünk új nevet, az új halmazt $H_1$-nek vagy $H_2$-nek fogjuk hívni).

A **II. FÁZIS** során legfeljebb $2m$ darab HOLVAN és pontosan $n - 1$ darab D-UNIÓ műveletet hajtunk végre (mivel egy fát építünk).

## 4.2. DISZJUNKT-UNIÓ–HOLVAN adatstruktúrák

*3 megvalósítást fogunk vizsgálni*:
**A) megoldás**: Egy $A$ tömbbel. $A(i)$ tartalmazza az $i$. csúcsot tartalmazó halmaz nevét.




INIT: **for** $i := 1..n$
$A(i) := i$
*Lépésszám*: $O(n)$.

HOLVAN($i$): **return**($A(i)$)
*Lépésszám*: $O(1)$.

D-UNIÓ($H_1, H_2$):
**for** $i := 1..n$
**if** $A(i) = H_2$ **then** $A(i) := H_1$
*Lépésszám*: $O(n)$

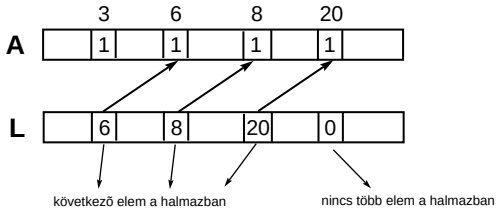
A **II. FÁZIS** összlépésszáma: $O(m + n^2) = O(n^2)$.

**Megjegyzés:** Sűrű gráfokra ez a jó megoldás.

*Kitérő*: Amikor algoritmusok lépésszámát akarjuk megbecsülni, a konstans szorzót nem tekintjük lényegesnek. 1 **Lépés**-nek (ejtsd „nagy lépés") konstans sok elemi lépést fogunk nevezni a továbbiakban. Sőt, ha szükség lesz rá, ezt a konstanst növelni is fogjuk az elemzés során (ez sosem okoz problémát, mert a nagyobb konstansba „belefférnek" a növelés előtti elemzés kisebb konstansai is).

**B) megoldás**: tömbökkel és láncolással okosabban, most 4 tömböt használunk:
$A(i)$ az $i$ csúcsot/elemet tartalmazó halmaz neve
$L(i)$ : a halmaz következő elemére mutató pointereket tároljuk, azaz az $i$ elemet tartalmazó halmaz következő elemének indexe $A$-ban, ill. 0, ha ez az utolsó elem
$K(j)$ a $j$. halmaz első eleme
$M(j)$ a $j$. halmaz mérete

Ha $H_2$ a kisebbik halmaz, akkor a $H_2$ elemeinél írjuk át az $A$ tömb megfelelő értékét, és a $H_2$ végére fűzzük a $H_1$-et, hogy annak elemein ne kelljen végigmenni.



INIT:
  **for** $i := 1..n$
    $A(i) := i;\ L(i) := 0;\ K(i) := i;\ M(i) := 1$

HOLVAN($i$): **return**($A(i)$)

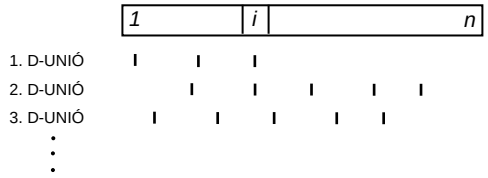
D-UNIÓ($H_1, H_2$):
  **if** $M(H_1) < M(H_2)$ **then csere** $(H_1, H_2)$
    /* *felcseréljük a halmazok neveit* $\Longrightarrow$ $H_1$ *lesz a nagyobb méretű*
  M($H_1$):= M($H_1$)+M($H_2$)
  $i := K(H_2)$
  **while** $L(i) \neq 0$
    $A(i) := H_1;\ i := L(i)$
  $A(i) := H_1$
  $L(i) := K(H_1)$
  $K(H_1) := K(H_2)$

*Elemzés*: most nem 1 darab D-UNIÓ művelet lépésszámát vizsgáljuk, hanem azt, hogy az algoritmus az $n-1$ darab D-UNIÓ művelet során összesen hány lépést tesz.

Az elemzés módszere a *strigulázás* lesz: egy táblázatot készítünk előre, és az algoritmus minden egyes Lépésénél húzunk egy strigulát a táblázat megfelelő helyére. A végén összeszámoljuk, hogy összesen hány strigulát húztunk és ez lesz az algoritmus Lépésszáma.

A Lépés konstansát úgy állítjuk be, hogy egy D-UNIÓ annyi Lépés legyen, mint a kisebbik halmaz mérete. Így egy művelet $\min(|H_1|, |H_2|)$ Lépés lesz. (Egy konkrét D-UNIÓ művelet esetén ennek a lépésszáma picit rosszabb – a konstans nagyobb – is lehet, mint az előző megvalósításban.)

Strigulázás:




A $j$. D-UNIÓ műveletnél a $j$. sorban $i$ alá húzunk egy strigulát, ha $A(i)$ tartalma megváltozott. Így pont $\min(|H_1|, |H_2|)$ strigulát húzunk be a D-UNIÓ$(H_1, H_2)$ műveletnél.

**6. Állítás.** *Minden $i$-re $i$ alá legfeljebb* $\log n$ *strigulát húzunk.*

**Bizonyítás:** Minden csúcs először egy egyelemű halmazban volt, ezután minden strigula behúzásakor egy legalább kétszer akkora méretűbe kerül (hiszen a kisebbik halmaz elemeinél húzunk strigulát). Így $\log n$ strigula után egy $n$ eleműbe kerül, tehát készen vagyunk.

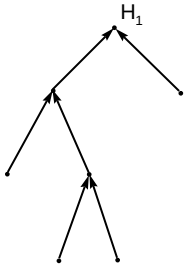
Így $n-1$ darab D-UNIÓ lépésszáma $O(n \cdot \log n)$ lesz.
Ezért a **II. FÁZIS** lépésszáma $O(m+n \cdot \log n)$, ez már egyértelmű javítás, és $m > n \cdot \log n$ esetén optimális is.

Okok, amiért a II. FÁZIS gyorsításán érdemes dolgozni (annak ellenére, hogy az I. FÁZIS teljes általánosságban nem gyorsítható):

- Sokszor az élek költségei kis egész számok, így $O(m)$ időben rendezhetőek (leszámláló rendezéssel).
- Sok más alkalmazásban is használják a HOLVAN és D-UNIÓ műveleteket.
- Előfordul, hogy eleve rendezve kapjuk az éleket, így nincs is szükség az I. FÁZIS-ra.
- Nem biztos, hogy a II. fázisban végig kell menni az összes elemen, ha „szerencsénk van", a legkisebb $n-1$ élet elég ismerni, tehát rendezés helyett a kupac (a lineáris idejű kupacépítéssel) segíthet.
- A rendezési algoritmus minden könyvtárban jól meg van írva, gyakorlatban gyorsan fut.

**C) megoldás**: a D-UNIÓ műveletet akarjuk 1 Lépésben megvalósítani, a HOLVAN művelet lesz lassabb. Általában ennek az ellenkezőjét fogjuk csinálni, a gyakoribb műveletet gyorsítjuk, a ritkábbat lassítjuk. Itt kivételesen azért érdemes fordítva, mert a gyakori HOLVAN műveleten csak picit lassítunk, viszont a D-UNIÓ sokat gyorsul, és most elsősorban $m < n \log n$ esetén akarunk gyorsítani.

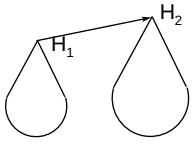
*Az ötlet:* A halmazokat gyökeres fákként ábrázoljuk (mindenkinek csak szülőpointerét tároljuk), a gyökérre ráírjuk a halmaz nevét.



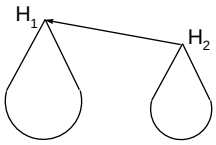

HOLVAN $(x)$ : $x$-ből indulva mindig a szülőre lépünk, míg a gyökérbe nem érünk, ott kiolvassuk a halmaz nevét. Az eddigi lustaságunk (uniók egy lépésben) miatt ez nagyon sok lépés is lehet. Ilyenkor viszont inkább még egyszer annyit dolgozva kicsit javítunk a fa alakján, hogy a jövőbeni HOLVAN kérdéseknél már jobb legyen a helyzet.

D-UNIÓ: $r$ rang segítségével. A rang lényegében a fa magassága lesz, pontosabban egy felső becslés a fa magasságára. Igazából a rangokat nem csak a fákhoz (gyökerekhez), hanem minden egyes csúcshoz hozzárendeljük, egy $v$ csúcsnál a $v$ magasságára ad felső becslést. Nem gyökér csúcsok esetén az algoritmushoz nem használjuk, csak az elemzéshez, tehát ilyeneknél tárolni sem kell. A rangokat kezdetben csupa nullára inicializáljuk. Mindig a kisebb rangú halmazt kötjük át a nagyobba:

Ha $r(H_1) < r(H_2)$

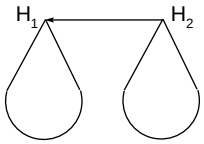


Ha $r(H_1) > r(H_2)$




Ebben a két esetben nem nő a rang.

Ha $r(H_1) = r(H_2)$

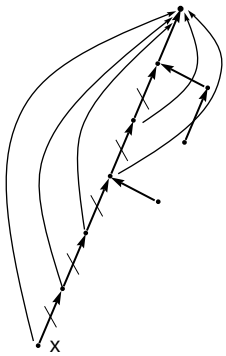



és $r(H_1)$++ (A $H_1$ rangját megnöveljük. Ez az egyetlen lehetőség, ahol $r$ változik.)

*Emlékeztető*: Egy $v$ csúcs *mélysége*: a gyökérből a $v$-be vezető út hossza. Egy $v$ csúcs *magassága*: ha $\ell$ a $v$ alatt a legtávolabbi levél, akkor a $v$-ből $\ell$-be vezető út hossza. A fa magassága:=gyökér magassága= $\max_v v$ mélysége=:a fa mélysége.

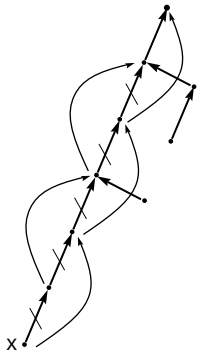
**HOLVAN** *javítása.* Ötlet: ha már sokat kell dolgoznunk, akkor dolgozhatunk egy kicsivel többet, ha ezzel a továbbiakban időt spórolunk. (A többletmunka során kicsit „helyrepofozzuk" az adatstruktúrát.) Ez a módszer a lustaság kompenzálására számtalan adatstruktúránál be fog még válni!

**Útrövidítés:**




Még egyszer végigmegyünk a fában az $x$-től a gyökérig vezető úton, és minden szülő pointert a gyökérre állítunk (kivéve a gyökérét).

**Útfelezés:**




Az $x$-től a gyökérig vezető úton minden csúcs szülő pointerét átállítjuk a nagyszülőjére (kivéve a gyökérét és a közvetlenül alatta lévőét). Ehhez nem kell még egyszer végigmenni az úton, egyszerűen egy lépéssel tovább megjegyezzük a szülő pointereket. Ez a módszer akár praktikusan picit gyorsabb is lehet mint az útrövidítés, mivel itt elég egyszer végigmenni a fában a keresőúton. Útrövidítés esetén pedig újabb ciklust kell kezdenünk, amely megkezdése (előkészítése) időt igényel. Természetesen az útrövidítés és az útfelezés során semelyik csúcs rangja nem változik.

*Az útrövidítés és az útfelezés elemzése*

A Kruskal algoritmus $n-1$ darab D-UNIÓ és $m' \leq 2m$ darab HOLVAN műveletet végez. Azt akarjuk elemezni, hogy összesen hány lépésben valósíthatjuk ezt meg. Most a D-UNIÓ $O(1)$ lépés, így a HOLVAN műveletek idejét kell elemeznünk.

**7. Állítás.** *Adott $v$-re $r(v)$ az elején 0 és monoton nő. Ha $v$ nem gyökér egy adott pillanatban, akkor sose lesz az, és az $r(v)$ már soha többé nem változik.*

**8. Állítás.** $r(v) < r(p(v))$ *(ha $v$ nem gyökér).*

**Bizonyítás:** D-UNIÓ művelet: Ha a kisebb rangút kötöttük be a nagyobb alá, akkor teljesül az állítás. Egyenlő rangúaknál szintén teljesül az állítás, mivel ekkor az új gyökér rangját növeljük.

HOLVAN: Ha átkötjük, akkor nagyobb rangúba kötjük, mivel átkötés előtt teljesült ez a tulajdonság; ezért nem romolhat el.

**9. Állítás.** *$x$ gyökerű fában van legalább $2^{r(x)}$ darab elem, ezért $\forall v:\ r(v) \leq \lfloor \log n \rfloor$.*

**Bizonyítás:** 1 pontú fára teljesül az állítás.
D-UNIÓ műveletnél: különböző rangúak esetén nyilván teljesül. Egyenlő rangúnál: mindkét fában legalább $2^{r(x)-1}$ darab elem van, ahol $r(x)$ az Unió utáni rang, a bekötés után legalább $2 \cdot 2^{r(x)-1} = 2^{r(x)}$ elem lesz a fában. A HOLVAN művelet nem változtatja meg ezt a tulajdonságot.

**10. Állítás.**

$$\big|\{v \mid r(v) = k\}\big| \leq \frac{n}{2^k}.$$

**Bizonyítás:** $A_v := \{w \mid w$ a $v$ leszármazottja volt, amikor $r(v) = k$ lett$\}$.

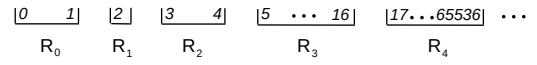
Belátjuk, hogy ha $r(x) = k$ és $r(y) = k$ és $x \neq y$, akkor $A_x \cap A_y = \emptyset$. (Az előző állítás miatt $|A_x| \geq 2^k$, tehát ebből tényleg következik az egyenlőtlenség.)

Tegyük fel, hogy $y$ később lett $k$ rangú, mint $x$. Ekkor $x$ nem volt benne az $y$ gyökerű fában a 8. állítás miatt. Ugyanakkor $A_x$ minden eleme $x$-szel azonos fában volt, tehát a két halmaz valóban diszjunkt. (Az állítás $k = 0$ esetén triviálisan igaz.)

**Definíció:** Torony függvény: $T(0) = 1$ és $T(n+1) = 2^{T(n)}$.
Az inverze: $\log^*(n) = \min\{k \mid T(k) \geq n\}$

**Megjegyzés:** Ha $n \leq 2^{65536} \Rightarrow \log^*(n) \leq 5$. (Az univerzumban a tudomány jelenlegi állása szerint kb. $2^{270}$, tehát sokkal kevesebb, mint $2^{65536}$ részecske van.)

*Rangcsoportok:*




$R_i := [T(i-1)+1, \ldots, T(i)]$, és $R_0 = [0, 1]$. A $v$ csúcs az $i$. rangcsoportban van, ha $r(v) \in R_i$.

**Elemzés:** A Lépés konstansát úgy állítjuk be, hogy egy D-UNIÓ 1 Lépés legyen, egy HOLVAN pedig annyi Lépés, ahány csúcsot érintünk a keresés (felfelé lépkedés) során.

A HOLVAN műveleteket ügyesen elemezzük, minden egyes művelet során annyi strigulát húzunk be, ahány csúcsot érintettünk. Azonban a strigulákat különböző helyekre rakjuk. A végén összeszámoljuk (felülről becsüljük) majd az egyes helyekre behúzott strigulákat és ezen darabszámok összegét.

2 féle helyre fogunk strigulát húzni: vagy az adott művelethez ($i$. HOLVAN), vagy a csúcsokhoz (az alaphalmaz elemeihez).

Az $i$. HOLVAN műveletnél a strigulázási szabály a következő. A művelet során bejárt út bármely $v$ csúcsára:

Behúzunk egy strigulát az $i$. HOLVAN művelethez, ha:

- $v$ vagy $p(v)$ gyökér, vagy
- ha $v$ szülője nem ugyanabban a rangcsoportban (tehát nagyobban) van, mint $v$.

Minden más esetben a $v$ csúcshoz húzunk egy strigulát.

Az összes ($m'$ darab) HOLVAN művelet után megszámoljuk, hogy hány strigula van a műveleteknél és mennyi a csúcsoknál.

Maximális rang $\leq \log n \Rightarrow$ a legnagyobb rangcsoport $R_\alpha$, ahol $\alpha \leq \log^*(\log n) = \log^*(n) - 1$. Tehát legfeljebb $\log^*(n)$ darab rangcsoportunk van. (A 0. rangcsoporttal együtt.)

Az $i$. HOLVAN sorában a strigulák száma legfeljebb $2 + (\log^*(n) - 1) = \log^*(n) + 1$, mert az út mentén a rangcsoport legfeljebb $(\log^*(n) - 1)$-szer nőhet.

Hány strigulát húzhattunk összesen egy csúcs mellé? Mikor húzunk strigulát egy $v$ csúcs mellé?

- Csak akkor kezdünk el strigulákat húzni egy $v$ csúcs mellé, ha már nem gyökér, és a szülője sem az. Ezután már $v$ rangja nem változik.
- Útfelezés/útrövidítés esetén, ha sem $v$ sem $p(v)$ nem gyökér, akkor a $v$ csúcs szülő pointere egy szigorúan távolabbi ősére fog mutatni. Így a rang szigorú monotonitása miatt $r(p'(v)) > r(p(v))$, azaz $v$ új – $p'(v)$-vel jelölt – szülőjének rangja szigorúan nagyobb, mint az átkötés előtti szülőjéé.

Legyen $v$ egy csúcs, melyre $r(v) = r$. Ekkor $v$ a $g := \log^*(r)$-edik rangcsoportban van.

**11. Állítás.** *Ilyen strigulázási szabályok mellett egy tetszőleges $g$-edik rangcsoportbeli $v$ csúcs mellé legfeljebb $T(g)-T(g-1)$ strigula kerülhet $g \geq 1$ esetén, $g = 0$ esetén pedig maximum* 1.

**Bizonyítás:** $g \geq 1$ esetén $T(g) - T(g-1)$ szám van a $g$. rangcsoportban. Ha már került strigula $v$ mellé, akkor $r(v)$ nem változik, míg $r(p(v))$ időben nő. Azonban ha a $g$. rangcsoporton túlnőtt, akkor az adott $v$ csúcs mellé már soha többé nem húzunk több strigulát. Tehát egy $g$ rangcsoportbeli $v$ csúcshoz maximum $T(g)-T(g-1)$ strigula kerülhet.

*Strigulák összeszámolása:*
$N(g)$ jelölje azt, hogy maximálisan hány csúcs lehet a $g$-edik rangcsoportban. A 10. állítás miatt:

$$N(g) \leq \sum_{r=T(g-1)+1}^{T(g)} \frac{n}{2^r} \leq \frac{n}{2^{T(g-1)+1}}(1 + \frac{1}{2} + \frac{1}{4} + \dots) = \frac{n}{2^{T(g-1)}} = \frac{n}{T(g)}.$$

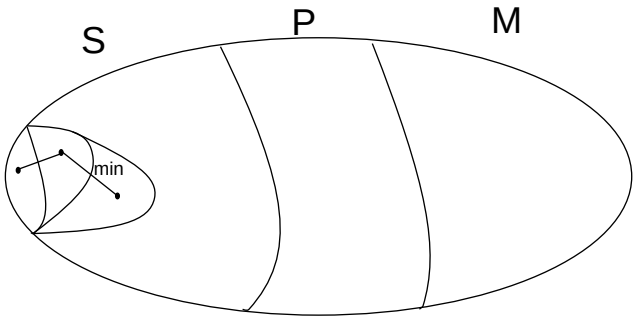
*Összefoglalva:* $g$ rangcsoportbeli csúcsok mellé összesen legfeljebb $\frac{n}{T(g)} \cdot (T(g)-T(g-1)) \leq n$ strigula kerülhet.
Ez igaz $g = 0$-ra is, mivel legfeljebb $n$ darab csúcs lehet a 0. rangcsoportban és a 0-s rangcsoportú csúcsok mellé csak 1 strigula kerülhet.
Így összesen a csúcsok mellé legfeljebb $\log^*(n) \cdot n$ strigula került, mivel $\log^*(n)$ rangcsoport van.

*Lépésszám:* $O(n - 1 + m' \cdot (\log^*(n) + 1) + n \cdot \log^*(n)) = O((n + m') \cdot \log^*(n))$, ahol $m'$ a HOLVAN műveletek száma.

Ha a Kruskal algoritmus II. FÁZISát így valósítjuk meg összefüggő gráfra, akkor a lépésszám $O(m \cdot \log^*(n))$ lesz, ami gyakorlati szempontból lényegében lineárisnak tekinthető. Valójában a legrosszabb esetben tényleg nem lineáris, nem csak mi voltunk nagyvonalúak az elemzésnél.

## 4.3. Prim algoritmusa

*Az algoritmus ötlete:* Kezdetben induljunk ki az 1-es számú csúcsból, majd minden egyes lépésben bővítsük ezt a fát (és az $S$ csúcshalmazát), úgy, hogy a belőle kiinduló ($S$ és $V-S$ közötti) legkisebb költségű élet vesszük hozzá.




$T$: aktuális fa
$S$: T-nek a csúcshalmaza
$P := \{v \in V - S \mid \exists uv \in E, u \in S\}$
$M := V - S - P$

A $P$-beli csúcsokra két dolgot tartunk nyilván:

- $K(v) := \min\{c(uv) \mid u \in S\}$ - ha a $v$ csúcsot kötjük be következőnek, akkor ez milyen költséggel fog járni.
- $p(v) := \text{argmin}\{c(uv) \mid u \in S\}$ - ha őt kötjük be, melyik csúcshoz kell kötni (ha több ilyen csúcs is van, akkor közülük az egyik).

*Az algoritmus:*

INIT: $P := \{s\};\ S := \emptyset;\ M := V - \{s\};\ T := \emptyset$
$K(s) := 0;\ p(s) := s$
**while** $P \neq \emptyset$
$v := \text{argmin}\{K(u) \mid u \in P\}$
$S \leftarrow v \leftarrow P$ /* *v-t átrakjuk S -be*
**if** $p(v) \neq v$ **then** $T := T + \{p(v), v\}$ /* *A fa megfelelő élét megépítjük*
**for** $v\underline{u} \in E$
**if** u $\in P$ && $c(vu) < K(u)$ **then** $K(u) := c(vu);\ p(u) := v$
**if** u $\in M$ **then** $K(u) := c(vu);\ p(u) := v;\ P \leftarrow u \leftarrow M$

*Lépésszám:*

*Szomszédsági mátrix esetén*: $O(n^2)$ (láttuk a szélességi keresésnél, hogy ennél gyorsabban nem is lehet).

*Éllistával, kupacokkal*:

| Művelet | Végrehajtások száma |
|---|---|
| Mintörlés | $n$ |
| Kulcs-csökk | $m$ |
| Beszúrás | $n$ |

Mindegyik művelet $O(\log n)$ idejű, ha bináris kupaccal csináljuk. Így az algoritmus $O(m \cdot \log n)$ idejű lesz.

**Megjegyzés:** ritka gráfok esetén ez sokkal jobb, mint az $O(n^2)$, de még javítani fogunk rajta. Az algoritmus jóságát nem bizonyítjuk, a Kruskal algoritmusnál tanult módszerrel könnyű ($e_1, e_2, \ldots, e_{n-1}$ legyenek az algoritmus által megépített élek az építések sorrendjében, utána a többi él sorrendje tetszőleges).

## 4.4. Dijkstra algoritmusa

*Adott:* $G = (V, E)$; és $c : E \to \mathbb{R}_0^+$ (hosszúság függvény, néha költségnek hívjuk).
*Feladat:* keressük meg az s-ből t-be vezető legrövidebb utat.

*Az algoritmus ötlete:* Irányítatlan gráfokra jól szemléltethető, ha elkészítjük a gráfot úgy, hogy a csúcsok gyöngyök, az élek megfelelő hosszúságú cérnák. Lefektetjük az asztalra, és $s$-nél fogva lassan felemeljük. Amikor egy csúcs felemelkedett, vonalzóval lemérhetjük az $s$ gyöngytől való távolságát, és nyilván ez lesz a legrövidebb út hossza abba a csúcsba. Ezt számítógépen úgy szimuláljuk, hogy minden egyes olyan gyöngyhöz, ami az asztalon fekszik, de megy belőle cérna a levegőbe, kiszámítjuk, hogy ha legközelebb ő emelkedne fel, akkor milyen messze lenne $s$-től. Könnyű meggondolni, hogy legközelebb az a gyöngy fog felemelkedni, amelyre a legkisebb számot írtuk.

A megvalósítás nagyon hasonlít a Prim algoritmuséhoz, holott az alapelv eléggé különböző, ez egyáltalán nem egy mohó algoritmus, inkább dinamikus programozási. Minden egyes lépésben azt tároljuk az egyes csúcsokra, hogy hol vannak, és hogy mi az eddig ide talált legrövidebb út $K(v)$ hossza (azaz ha $v$ emelkedne fel legközelebb, akkor milyen messze lenne $s$-től). Minden egyes lépésben a $P$-beli legkisebb kulcsú csúcsot „emeljük fel", majd szomszédain módosítjuk a kulcsot.

$S$: a levegőben lévő gyöngyszemek.

$P$: olyan gyöngyszemek, amelyek az asztalon vannak, de van olyan szomszédjuk, ami a levegőben van.

$M$: a többi csúcs.

**Megjegyzés:** Negatív élhosszakkal nem fog működni az algoritmus. Viszont irányított gráfokra formálisan ugyanez az algoritmus jó.

*Az algoritmus:*

INIT: $P := \{s\};\ S := \emptyset;\ M := V - \{s\}$
$K(s) := 0;\ p(s) := s$
**while** $P \neq \emptyset$
$v := \text{argmin}\{K(u) \mid u \in P\}$
$S \leftarrow v \leftarrow P$
**for** $v\underline{u} \in E$
**if** $u \in P$ && $K(v) + c(vu) < K(u)$ **then**
$K(u) := K(v) + c(vu);\ p(u) := v$
**if** $u \in M$ **then** $K(u) := K(v) + c(vu);\ p(u) := v;\ P \leftarrow u \leftarrow M$

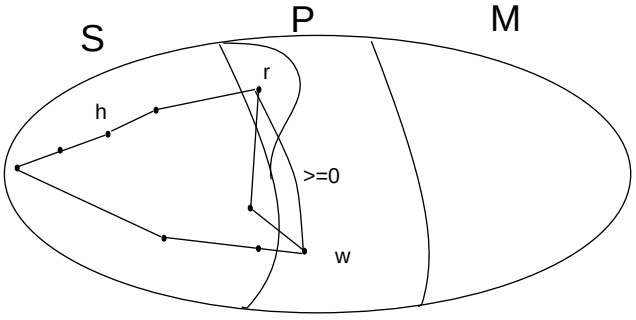
**1. Definíció.** *Egy $s \rightsquigarrow x$ út $S$-út, ha minden csúcsa $S$-ben van, kivéve esetleg $x$-et.*

**5. Tétel.** *Az algoritmus során minden* **while** *ciklus végén:*

*(a) Bármely $x \in S$-re $K(x)$ a legrövidebb $s \rightsquigarrow x$ út hossza, és az ilyen hosszúságú utak között van $S$-út is.*

*(b) Minden $x \in P$-re $K(x)$ a legrövidebb $S$-út hossza.*

**Bizonyítás** (vázlatosan): (a) Elég az éppen S-be átrakott csúcsra belátni, legyen ez $r$. Az $r$-be nem létezik $K(r)$-nél rövidebb út:




Indirekt tegyük fel, hogy létezik ennél rövidebb $Q$ út, ekkor az indukciós feltevés szerint ez nem $S$-út, legyen az első $P$-beli csúcsa $w \neq r$. Ekkor $r$ definíciója miatt $K(r) \leq K(w)$, és így az indukciós feltevés miatt a $Q$ út $s$-től $w$-ig terjedő része nem rövidebb, mint az $r$-be vezető legrövidebb $S$-út $K(r)$ hossza. Mivel az élhosszak nemnegatívak, a $Q$ út $w$-től $r$-ig terjedő része is nemnegatív, tehát összesen $Q$ hossza legalább $K(r)$, ami ellentmond $Q$ választásának.

Az állítások többi része hasonlóan bizonyítható.

**6. Tétel.** *(a) Ha az $S$-be kerülés sorrendjében vesszük a csúcsokat ($v_i$ : $i$.-nek került $S$-be), akkor $K(v_1) \leq K(v_2) \leq \cdots \leq K(v_n)$.*

*(b) Ha utoljára $v$ került $S$-be, akkor minden $u \in P$-re $K(v) \leq K(u) \leq K(v) + \max_{xy \in E} c(xy)$.*

**Bizonyítás:** szintén indukcióval, először (b)-t érdemes. Ha utoljára $v$ került $S$-be, akkor minden $u \in P$-re $K(v) \leq K(u)$, mert a $v$-t úgy választottuk. A kulcsmódosítások előtt az indukciós feltevés miatt (mind a két részt használva) a második egyenlőtlenség is fennállt, és kulcsmódosítás közben nyilván nem romolhat el egyik egyenlőtlenség sem. Ebből az (a) rész rögtön következik.

**12. Állítás.** *Az $x$ csúcsba vezető egyik legrövidebb út (hátulról előre állítjuk elő):* $s, \ldots, p(p(x)), p(x), x$.

**Megjegyzések.**

- Figyeljük meg, hogy itt „implicit" outputot ad meg az algoritmus. Ha fel akarnánk írni az összes csúcsba vezető legrövidebb utat, az akár $c \cdot n^2$ hosszú is lehetne. Ehelyett a $p$ tömböt adjuk ki outputként (ami a legrövidebb utak fájának egy leírása), melyből bármely legrövidebb út annyi időben kérdezhető le, mint az éleinek a száma.
- Az algoritmus módosítható úgy, hogy pl. legszélesebb, ill. legbiztonságosabb utat keressen.
- Ha a fordított gráfon futtatjuk, minden csúcsból egy adott $t$ csúcsba vezető legrövidebb utakat keressük meg.
- Ha két különböző súlyfüggvény ($c_1$ és $c_2$) adott az éleken, akkor meg tudjuk keresni a $c_1$ szerinti legrövidebb $s \rightsquigarrow t$ utak közül azt, amelyik a $c_2$ szerint a legrövidebb. Először futtassunk $c_1$ szerinti Dijkstrát a gráfon $s$-ből, és a fordítottján $t$-ből. Ezután minden $v$ csúcsra tudjuk a $d(s, v)$ és $d(v, t)$ „távolságokat". A $c_2$ szerinti Dijkstra futtatásánál az olyan éleket hagyjuk figyelmen kívül, amelyekre $c_1(uv) > d(s, t) - d(s, u) - d(v, t)$.
- Ha megengedjük a negatív élhosszakat, akkor a feladat $NP$-nehéz lesz (pl. csupa $-1$ élhossz esetén Hamilton-út létezését is el kellene dönteni).
- Irányított gráfok esetén, ha bármely irányított kör nemnegatív, akkor van másik algoritmus (Bellman-Ford), amely helyes és polinomiális, de lassabb: $O(m \cdot n)$.
- Viszont aciklikus irányított gráfok esetén negatív élsúlyra is van kicsit egyszerűbb, $O(m)$ idejű algoritmus, ezt használják pl. a PERT módszerben.
- Irányítatlan esetben, ha nincs negatív kör, akkor szintén létezik polinomiális algoritmus, de az bonyolultabb.

*Elemzés*

Ugyanúgy, mint a Prim algoritmusnál, tehát szomszédsági mátrix esetén $O(n^2)$ a lépésszám, éllista esetén, ha valamilyen kupacot használunk, akkor pedig legfeljebb $n$ darab Beszúrás, $n$ darab Mintörlés, és $m$ darab Kulcs-csökk kell.

## 4.5. $d$-edfokú kupacok

**2. Definíció.** *Egy fa $d$-*edfokú kupac*, ha*

- *minden csúcsnak legfeljebb $d$ gyereke van,*
- *kupacrendezett,*
- *kiegyensúlyozott.*

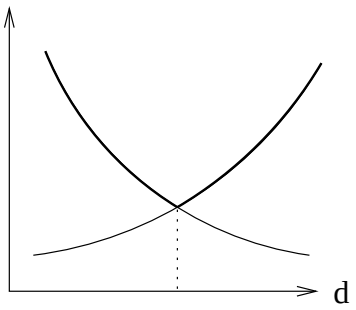
Tömbös megvalósítás esetén 0 az első index, és az $i$ indexű csúcs gyerekeinek indexei $d \cdot i + 1, d \cdot i + 2, \ldots, d \cdot i + d$. Tehát a $j$ szülője $\lfloor \frac{j-1}{d} \rfloor$. A kiegyensúlyozás itt (is) azt jelenti, hogy a tömbben nincs lyuk.

A műveleti idők változása:
Beszúrás, Kulcs-csökk (felbillegtetés): $O(\log_d n)$
Mintörlés (lebillegtetés): $d$ gyerekből minimum kiválasztása: $d-1$ összehasonlítás, meg az aktuális kulccsal való hasonlítás = $d$ db összehasonlítás lebillentésenként. Mivel a fa mélysége $\log_d n$, így $O(d \cdot \log_d n)$ adódik.

Érdekes eset: $d = 4$. Itt:
Felbillegtetés: $\log n$ helyett $\log_4 n = \frac{\log n}{2}$.
Lebillegtetés: $2 \cdot \log n$ helyett $4 \cdot \log_4 n = 2 \cdot \log n$, azaz ugyanannyi.
Tehát összességében egyértelműen jobb, mint a bináris kupac.

### 4.5.1. A Dijkstra és Prim algoritmus lépésszáma $d$-edfokú kupaccal

Mi vajon a legjobb $d$? Erre általában nincs egyértelmű válasz, az egyes műveletek gyakoriságától függ.
Tekintsük a Dijkstra (vagy a Prim) algoritmust. Itt az egyes műveletek száma:
$n$ db Beszúrás, Mintörlés
$m$ db Kulcs-csökk
Az elv: gyorsítsuk amiből sok van. A teljes lépésszám: $O(n \cdot d \cdot \log_d n + m \cdot \log_d n) = O(\max(n \cdot d \cdot \log_d n,\ m \cdot \log_d n))$. Az első tag $d \geq 3$-ra szigorúan növő, a második szigorúan csökkenő, tehát a maximum a lehető legkisebb értéket ott veszi fel, ahol a kettő egyenlő.




$$\begin{aligned} n \cdot d \cdot \log_d n &= m \cdot \log_d n \\ d &= \frac{m}{n} \\ d^* &:= \max\left(2, \left\lceil \frac{m}{n} \right\rceil\right) \end{aligned}$$

értéket érdemes használni, mert $d$-nek egésznek kell lennie és nem akarunk 1-et.

Így a lépésszám $O(m \cdot \log_{d^*} n)$. Ez ritka gráf esetén (ha $m \approx c \cdot n$) nem nagyon segít, de sűrűnél igen:

Ha $m \geq n^{1+\varepsilon} \Rightarrow d^* \geq n^{\varepsilon} \Rightarrow \log_{d^*} n \leq 1/\varepsilon \Rightarrow$ a lépésszám: $O\left(\frac{1}{\varepsilon} \cdot m\right) = O(m)$, ha $\varepsilon$ konstans.

**Megjegyzés:** Ne felejtsük el, hogy itt csak a legrosszabb eset lépésszámát minimalizáltuk. A gyakorlatban ez nem szokott előfordulni, hiszen két dolog is kellene hozzá: majdnem minden él miatt csinálunk egy Kulcs-csök műveletet, és ezeknél majdnem mindig a fa magasságával arányos felbillentést kell végeznünk. Ezért a gyakorlatban érdemes kisebb $d$ értéket választani, sokszor a $d = 4$ az optimális.

# 5. Amortizációs elemzés

Bemelegítés: A Számláló egy egyszerű adatstruktúra, két művelettel:

INIT $(n)$: Újtömb $(A, n)$; **for** $i = 1..n \quad A(i) := 0$

NÖV $(A)$: Az $A$ bit-tömb tartalmát azonosítjuk a bitek összeolvasásával kapott kettes számrendszerbeli $a$ számmal. Ezt kell eggyel megnövelni. Feltesszük, hogy ez a művelet maximum $2^n - 1$ alkalommal lesz meghívva. A NÖV műveletet is könnyű megvalósítani:
NÖV $(A)$:
**for** $i = n..1 \quad (-1)$
**if** $A(i) = 1$ **then** $A(i) := 0$
**else** $A(i) := 1; \quad i := 0$

Mennyi egy NÖV művelet lépésszáma? Nyilván a legrosszabb esetben $n$. De mennyi lesz átlagosan, azaz $m$ darab NÖV művelet összlépésszáma $m$-mel osztva (tetszőleges $m < 2^n$-re)?

**13. Állítás.** *Egy NÖV művelet átlagosan két lépésben végrehajtható.*

Képzeljük el a következőt. Minden NÖV híváskor kapunk két dollárt. Ki kell fizetnünk annyi dollárt, amennyit lépünk, tehát ahány elemet átírunk. Azonban ha marad pénzünk, azt betehetjük a bankba, és később, ha szükséges, a bankból kivett pénzünkkel is fizethetünk a lépéseinkért. Ha sose mehet le negatívba a bankszámlánk, akkor ez bizonyítja, hogy $m$ darab NÖV művelet összlépésszáma maximum $2m$.

Ezt egy kicsit finomabban is csinálhatjuk: nem egy bankot használunk, hanem az $A(i)$ cellákra rakunk dollárokat. Az a célunk, hogy minden olyan $i$-re legyen $A(i)$-n egy dollár, amelyre $A(i) = 1$. Az elején nincs ilyen cella, így ez teljesül. Ha egy NÖV műveletet végzünk, akkor az 1 tartalmú érintett cellákból felvesszük az egy dollárt és rögtön ki is fizetjük a lépésért. Amikor először érünk 0 tartalmú cellához, akkor a kapott két dollárunkból az egyiket rárakjuk a most 1 tartalmúvá átírt cellára, a másikkal fizetünk ezért az – utolsó – lépésért.

## 5.1. Potenciál és amortizációs idő

Bevezetjük az alábbi jelöléseket és fogalmakat:

$TI_i$: az $i$. művelet tényleges ideje (a Lépések száma).

Potenciál: egy függvény, amely az adatstruktúra tetszőleges állapotához hozzárendel egy nemnegatív valós számot. Egy művelet-sorozat esetén $P_i$ jelöli az $i$. művelet során létrejött struktúra potenciálját. $P_0 = 0$ (a kezdeti, üres struktúra potenciálja nulla), és minden $i$-re $P_i \geq 0$.

Az $i$. művelet *amortizációs* ideje, melyet $AI_i$-vel jelölünk, a következő:
$AI_i := TI_i + \Delta P = TI_i + (P_i - P_{i-1})$.

Ekkor $\sum_i AI_i = \sum_i TI_i + P_T - P_0 \geq \sum_i TI_i$.

Tehát az amortizációs idő egy felső becslést ad a műveletek összidejére.

**Megjegyzés:** Aki tanulta pl. Johnson algoritmusát, vagy a minimális költségű folyam/áram feladat duálisát, az már találkozott a potenciál fogalmával. Egy gráf csúcsaihoz rendelünk $\pi(v)$ potenciálokat, és az eredetileg $c(uv)$ költségű $uv$ él redukált költségének nevezzük a $c'(uv) := c(uv) + \pi(u) - \pi(v)$ mennyiséget. Egy $s$ csúcsból egy $t$ csúcsba vezető $P$ út $c'$ szerinti hossza megkapható a $c$ szerinti hosszából, ha hozzáadunk $\pi(t)$-t és kivonunk $\pi(s)$-et.

Ha most egy adatstruktúrához készítünk egy gráfot, melynek csúcsai a lehetséges állapotok, az élei pedig a műveletek $TI$ költséggel, akkor a redukált költségek pont az $AI$ mennyiségek lesznek.

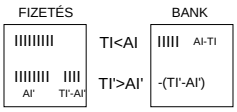
**Megjegyzés:** Tipikusan olyan állításokat bizonyítunk, hogy pl. egy-egy művelet amortizációs ideje legfeljebb $O(\log n)$, ebből következik hogy $m$ művelet elvégzésének tényleges összideje legfeljebb $O(m \cdot \log n)$. De persze 1 adott művelet elvégzésének tényleges ideje lehet akár $c \cdot n$ is, ezért az amortizációs elemzéssel garantált lépésszámú adatstruktúra nem hasznos valós idejű folyamatokban (pl. egy repülőgép navigációs rendszerétől nem azt várjuk, hogy átlagosan gyorsan döntsön, hanem azt, hogy minden egyes vészhelyzetre azonnal reagáljon). Azonban egy algoritmus teljes futása során végrehajtott adatstruktúra-műveletek összidejére a fentiek alapján korrekt becslést kapunk.

**Megjegyzés:** Persze korrekt becsléseket kapunk nemcsak az összidőre, hanem minden $m' < m$ esetén az első $m'$ művelet összidejére is (a fenti példában $O(m' \cdot \log n)$).

Az előző példára alkalmazva: legyen a $P$ potenciál az adott pillanatban az 1-esek száma. Ekkor kezdetben ez 0. Ha egy NÖV művelet során az utolsó $k$ jegy 1-es, akkor a művelet tényleges ideje $k + 1$ lesz. A potenciálváltozás pedig $\Delta P = -k + 1$. Tehát egy NÖV művelet amortizációs ideje $AI = (k + 1) + (-k + 1) = 2$.

### 5.1.1. Amortizációs idő és a strigulázás kapcsolata

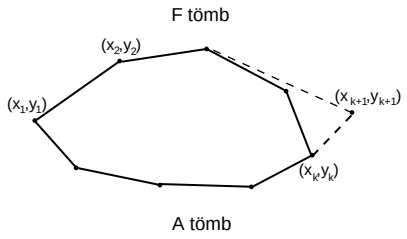
Amikor egy műveletet végzünk, akkor kapunk a művelet megengedett amortizációs idejének megfelelő $-t. Ha az algoritmust kevesebb Lépésből tudjuk végrehajtani, akkor a maradékot berakjuk a bankba. Ha viszont több Lépésre van szükségünk, akkor a különbözetet ki kell vennünk a bankból.




Kell: A bank pénze ne mehessen le negatívba. Ezt pont azzal biztosítjuk, hogy megköveteljük, hogy $P \geq 0$ legyen, ugyanis egy adott pillanatban a bankban levő pénz éppen az aktuális potenciál.

## 5.2. Konvex burok keresése

*Adott*: $p_i = (x_i, y_i) \in \mathbb{R}^2$ $(1 \leq i \leq n)$ pontok a síkban.
*Feladat*: A pontok konvex burkának megtalálása (a rajta lévő csúcsok felsorolása pozitív irányú bejárás sorrendjében).




**Megoldás**:
I. Rendezzük a pontokat $x_i$-k szerint $(x_1 \leq x_2 \leq \cdots \leq x_n)$.
*Idő*: $O(n \cdot \log n)$.
II. Egy adott ciklusban tudjuk a $p_1, \ldots, p_{i-1}$ pontok konvex burkát, külön tároljuk balról jobbra rendezetten az $F$ felső és az $A$ alsó ívet.
**for** $i = 2..n$
Felező kereséssel az alsó és a felső íven végigmenve megkeressük az ugrópontot (az utolsó olyat, ami még a $p_1, \ldots, p_i$ halmaz konvex burkán is rajta van)
$A$ és $F$ további részét elfelejtjük, majd végükre rakjuk $p_i$-t. (Ha $x_i = x_{i-1}$, akkor vagy $p_i$ rajta van az utolsó pontok közötti szakaszon, ekkor nincs tennivalónk, vagy pl. $y_i$ nagyobb az $F$ utolsó pontjának $y$ koordinátájánál, ekkor csak $F$-et kell változtatnunk).

**Megjegyzés:** $F(j)$ az ugrópont, ha az $F(j-1)$ és $F(j)$ pontokat összekötő egyenes $p_i$ felett megy, de az $F(j)$ és $F(j+1)$ pontokat összekötő egyenes már alatta.

Idő: 1 pont hozzáadása legfeljebb $O(\log n)$, így az összidő: $O(n \cdot \log n)$.

*Összefoglalva*: síkban a konvex burok keresés $O(n \cdot \log n)$ időben megoldható.
Itt is érdemes a második fázist gyorsítani (pl. lehet, hogy $x$ szerint rendezve kaptuk a pontokat).

**7. Tétel.** *Ha a felező keresés helyett jobbról indulunk, és sorban megvizsgáljuk, hogy az adott pont-e az ugrópont, akkor a második fázis ideje csak* $O(n)$.

**Bizonyítás:**
*A potenciál definiálása*: $P := |F| + |A|$.
*1 Lépés definiálása*: F-ben az ugrópont megkeresése $\ell + 1$ Lépés legyen, ha $F$ végéről $\ell$ csúcsot töröltünk ki.
(Ugyanígy definiáljuk a Lépéseket $A$ esetén is, ekkor $\ell'$ a törölt csúcsok száma). Az elemzés alapötlete az, hogy a kitörölt csúcs már soha nem fog szerepelni.

*Amortizációs elemzés*: $AI_i = TI_i + \Delta P = ((\ell+1)+(\ell'+1)+1+1)-((\ell-1)+(\ell'-1)) = 6$
Azaz egy csúcs beszúrásának amortizációs ideje 6. Ez azt jelenti, hogy $6n$ Lépésben, azaz $O(n)$ időben megoldható a feladat. Az elemzés persze attól működik, hogy ha egy

csúcsot $F$-ből vagy $A$-ból kidobunk, akkor azzal később már sose kell foglalkoznunk, hiszen már az aktuális ponthalmaz konvex burkának is belső pontjai. Összesen persze a két tömbből kevesebb mint $2n$ esetben dobhatunk ki pontot.

# 6. Rafináltabb kupacok

A kupacokat angolul egyaránt hívják „heap”-nek és „priority queue”-nak, azaz prioritásos sornak.

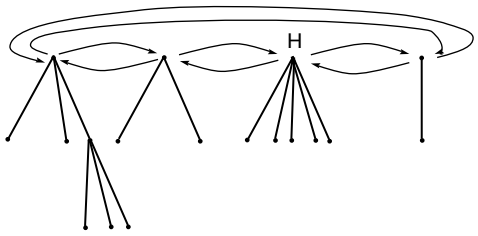
## 6.1. Fibonacci kupac

*Tulajdonságai:*

- Több kupacrendezett fából áll, amelyekben egy csúcs gyerekeinek száma nincs korlátozva, és nem is kiegyensúlyozottak,
- a gyökerek ciklikusan két irányban körbe vannak láncolva,
- a kupacot egy $H$ pointerrel azonosítjuk, amely a minimális kulcsú gyökérre mutat.

*Tárolás:*

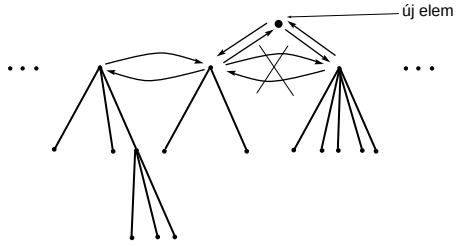
- Minden elemhez 4 pointert tárolunk: szülője, bal testvér, jobb testvér, egy gyereke (mindegy, hogy melyik gyerek: ők is ciklikusan két irányban körbe vannak láncolva, így gyorsan (lineáris időben) végigjárhatjuk őket).
- $r(v) \in \mathbb{N}$: a $v$ elem rangja (a gyerekeinek a száma).
- $m(v) \in \{0, 1\}$: jelölő-bit.




**Műveletek**:

*Beszúrás(H,új)*:

- A gyökerek közé új gyökérként (1 pontú fa) felvesszük új-at.
- Ha $K(\text{új}) < K(H)$, akkor a $H$-t az új-ra állítjuk.



Idő: $O(1)$.

A Mintörléshez szükségünk lesz egy belső műveletre.

LINK $(u, v)$ művelet:
Ha $K(u) > K(v)$, akkor először megcseréljük a csúcsok $u$ és $v$ neveit. Ezután $v$-t bekötjük $u$ alá és $u$ rangját növeljük eggyel. Az új $u$ gyökér a gyökér-listában a híváskori $u$ (azaz az első paraméter) helyére kerül (tehát, ha volt névcsere, akkor a keletkezett $u$ gyökeret átrakjuk a $v$ bekötés előtti helyére). Egy LINK művelet ideje: $O(1)$. Az 1 Lépést úgy definiáljuk, hogy beleférjen **két** LINK művelet és még néhány adminisztrációs lépés.

*Mintörlés* $(H)$:

- $MIN := H$, a végén **return**$(MIN)$.
- I. fázis: $H$-t kivesszük, helyére befűzzük a gyermekeit, és ezek szülő pointereit $nil$-re állítjuk.
- II. fázis: Amíg létezik 2 azonos rangú gyökér, addig LINK művelettel összekapcsoljuk ezeket.
- III. fázis: végigmegyünk a gyökereken és a minimális kulcsú gyökérre állítjuk $H$-t.

**Megjegyzés:** A rangra nem kötöttünk ki direkt felső korlátot. De magától igaz lesz, és ezt a végén bizonyítani fogjuk, hogy minden csúcs rangja legfeljebb $R = O(\log n)$. (*Emlékeztető: $n$ a kupacban tárolt elemek számára felső becslés.*)

*I. fázis ideje:*
a gyerekek befűzése a gyökerek listájába konstans időben megy, a szülő pointerek $nil$-re állítása $R = O(\log n)$ lépésben ($H$-nak legfeljebb $R$ gyereke lehet).

*II. fázis megvalósítása:*
$T$ segédtömb segítségével: $T[0 : R]$.
$T$ indexelése a rangokkal, $T(i)$ értéke: mutató egy $i$ rangú gyökérre, ha van az eddig vizsgáltak között, $nil$ különben.

*Az algoritmus:*

**for** $i = 0..R$
  $T(i) := nil$  /* *Ez itt fontos! Minden Mintörlésnél újra használjuk.*
**for** $v \in$ gyökerek  /* *ld az alábbi megjegyzést.*
  $w := v$
  **while** $T(r(w)) \neq nil$
    LINK $(w, T(r(w)))$  /* *Ezzel $r(w)$ eggyel megnő*
    $T(r(w) - 1) := nil$
  $T(r(w)) := w$

**Megjegyzés:** A gyökereken úgy lépdelünk végig, hogy a $H$ pointert ideiglenesen az első fázisban egy tetszőleges gyökérre állítjuk (pl. a kitörölt $H$ jobb testvérére), majd $H$-t egy $v$ gyökér sorravételekor annak jobb testvérére átállítjuk. Azonban valahogy észre kell vennünk, hogy körbeértünk, és ez azért nem egyszerű, mert ciklikusan láncolt listánk van, és a kezdő gyökeret pedig lehet, hogy egy LINK műveletnél kiszedtük a gyökerek közül. Talán a legegyszerűbb megoldás az, hogy ha a LINK műveletet két identikus pointerrel hívjuk, akkor kiszállunk a ciklusból.

Egy adott ilyen LINKelési sorozat elemzéséhez célszerű bevezetni egy ideiglenes $Q$ potenciált, legyen $Q$ a fele a tömbben levő $nil$ pointerek számának. Elég belátni, hogy erre nézve ezen algoritmus amortizációs ideje $O(\log n) + \#\text{LINK}$. Az inicializálásnál a $Q$ potenciál legfeljebb $(R+1)/2$-vel nő, a Lépések száma $\leq (R+1)/2$, tehát az amortizációs idő $\leq R + 1 \leq O(\log n)$. A főciklus egy lefutásának amortizációs ideje $\leq 0$, ha nem végzünk LINKelést, mivel $Q$ féllel csökken, és a tényleges idő bőven belefér fél Lépésbe. Ha a while ciklusban $k \geq 1$ db LINKelést végzünk, akkor a $Q$ potenciál $(k-1)/2$-vel nő, a tényleges idő $k/2$, így az amortizációs idő legfeljebb $k$.

*Idő*: $TI = O(\log n) + \#\text{LINK}$.

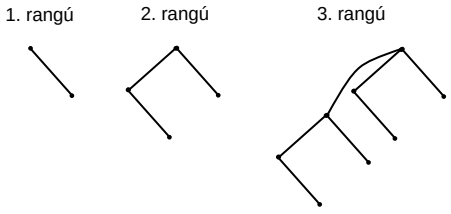
A II. Fázis végén legfeljebb $R + 1$ darab gyökér marad. (Minden lehetséges ranghoz legfeljebb 1.)

*III. fázis ideje:* Legfeljebb $R$ összehasonlítással, így $O(\log n)$ Lépésben megy.

*Ezeket összeadva:* $TI = O(\log n) + \#\text{LINK}$.

**1. potenciál** (később változtatnunk kell még): *gyökerek száma*
*Beszúrás*: $AI = O(1) + 1 = O(1)$.
*Mintörlés*: $AI \leq (O(\log n) + \#\text{LINK}) + (-1 + R - \#\text{LINK}) = O(\log n)$.

**Megjegyzés:** A LINKelések után az alábbi alakú ún. kanonikus fák (vagy másnéven binomiális kupacok) keletkeznek (mindaddig, míg csak Beszúrás és Mintörlés műveleteket végeztünk).



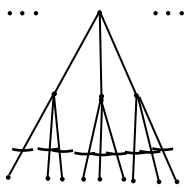

*Kulcs-csökk*:($H$, $v$, $\Delta$): (*1. Változat*)
$v$-t gyökeresítjük (elvágjuk a szülőjéhez vezető élet, két testvérét egymáshoz linkeljük, és $v$-t befűzzük a gyökerek listájába, a szülő rangját eggyel csökkentjük), majd:

- $K(v) := K(v) - \Delta$
- **if** $K(v) < K(H)$ **then** $H := v$

Ennek a lépésnek a tényleges ideje 1 Lépés, az amortizációs ideje 2 Lépés.

Látszólag készen vagyunk, de van egy nagy baj: Így nem marad igaz, hogy a rang legfeljebb $O(\log n)$.

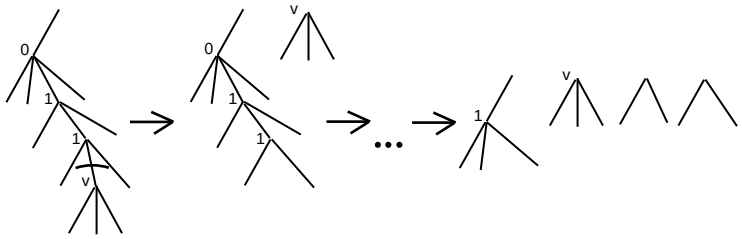
Szeretnénk: ha $r(v) = r$, akkor $v$-nek van legalább $c^r$ $(c > 1)$ leszármazottja (ami persze max. $n$ lehet). Ez Kulcs-csökk nélkül még teljesülne, de így nem teljesül: ha minden unokára hívnak egy-egy Kulcs-csökk műveletet, akkor $v$-nek saját magán kívül csak a gyerekei maradnak meg leszármazottnak, míg rangja nem változik:



**Megoldás**: $m(v)$ jelzőbit használata:

- Ha gyökér: 0 (gyökeresítéskor visszaállítjuk 0-ra, ha nem az volt).
- Ha $v$ nem gyökér és gyökeresítettük egy gyerekét, akkor $m(v) := 1$.

*Kaszkád-vágás*: addig vágunk, amíg a szülőjének jelzőbitje nem 0. (Utána ezt a 0-t 1-re állítjuk, ha nem gyökéren van.)




Tehát a Kulcs-csökk **végleges** megvalósítása:
*Kulcs-csökk*:$(H, v, \Delta)$:
$v$-t gyökeresítjük, majd:

- A Kaszkád-vágást hajtjuk végre, addig, amíg az utoljára gyökeresített csúcs szülője jelöletlen nem lesz, és a végén ezt a szülőt megjelöljük, ha nem gyökér.
- $K(v) := K(v) - \Delta$.
- **if** $K(v) < K(H)$ **then** $H := v$.

Az elemzéshez egy új potenciált kell definiálni:

**2.** (Végső) **Potenciál**: #gyökerek $+2 \cdot$ #jelölt csúcsok.

A Beszúrásnál és Mintörlésnél a jelöltek száma nem nő, tehát itt a potenciálváltozás nem lehet több, mint az első Potenciállal ($H$ gyerekeinek gyökeresítésekor néhány jelzőbit 0-ra változhat). Ezért az ilyen lépések amortizációs ideje nem nőtt.

*Kulcs-csökk*: $k$ db vágás van a kaszkádvágás során, ekkor legalább $k - 1$ helyen 0-ra változott meg a jelölő-bit, míg 1-re csak legfeljebb 1 helyen.

$TI = k$ (Lépés),
$\Delta P \leq -2(k-1) + k + 2 = 4 - k$.
(Mivel a jelölő-bitek száma legalább $(k-1)$-gyel csökken, $k$ új gyökér lesz és a végén lehet egy új jelölt csúcs.)
$AI = TI + \Delta P \leq 4$.

Most belátjuk, hogy a rang nem lehet túl nagy:

Egy tetszőleges időpontban indexeljük az $x$ csúcs $y_1, \ldots, y_{r(x)}$ gyerekeit a gyerekké válás sorrendjében (azaz, hogy mikor lett utoljára $x$ gyereke).

**14. Állítás.** $r(y_i) \geq i - 2$

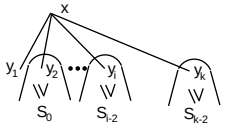
**Bizonyítás:**
Amikor $y_i$ az $x$ gyereke lett (utoljára):
LINK műveletet végeztünk, tehát rangjuk egyenlő volt. Tehát ekkor a rangjuk legalább $i-1$ volt ($x$-nek ekkor gyerekei voltak $y_1, \ldots, y_{i-1}$), és azóta $y_i$ rangja csak egyet csökkenhetett, különben levágtuk volna $x$-ről.

**3. Definíció.** *$S_k$ jelölje azt, hogy egy legalább $k$-adrangú csúcsnak minimum hány leszármazottja lehet (bármikor).*

**Megjegyzés:** minden elem leszármazottja saját magának. $S_0 = 1$, $S_1 \geq 2$.

**8. Tétel.** $S_k \geq 2 + \sum_{i=0}^{k-2} S_i$

**Bizonyítás:**




Fibonacci számok: $F_0 = 0$; $F_1 = 1$; $F_{n+2} = F_{n+1} + F_n$

**15. Állítás.** $S_k \geq F_{k+2}$.

**Bizonyítás:** Indukcióval triviális.

Legyen $\varphi = \frac{\sqrt{5}+1}{2}$. Tudjuk, hogy

$F_{k+2} \geq \varphi^k$, ebből $S_k \geq \varphi^k$.

Emlékeztető: $n$ jelöli a kupacban szereplő elemek maximális számát. Tehát bármely $v$ elemnek legfeljebb $n$ leszármazottja lehet, így $r(v) \leq \log_\varphi n \approx 1,44 \cdot \log n$. Ez lesz $R$ értéke.

Azaz beláttuk, hogy a Fibonacci-kupaccal:

$n$ db Beszúrás

$n$ db Mintörlés

$m$ db Kulcs-csökk

összes ideje: $O(n \log n + m)$.

Ebből következik:

**9. Tétel.** *A Dijkstra és Prim algoritmus lépésszáma Fibonacci-kupaccal* $O(n \log n + m)$.

**Megjegyzések.**

- A Dijkstra-ban semmilyen adatstruktúrával nem lehet ez alá lemenni (mivel tud rendezni, pl. egy csillag éleire írjunk $a_i$ súlyokat; másrészt összefüggőséget is el tud dönteni).
- A Prim-nek is ez a legjobb futási ideje (ugyanezért), de minimális költségű feszítőfa keresésére $m < n \log n$ esetén ismerünk gyorsabb algoritmust (érdekes módon az is a Fibonacci-kupacon alapul).

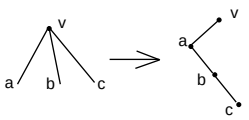
## 6.2. Párosítós kupacok

Ami miatt a Fibonacci-kupac a gyakorlatban ritkán hatékony:

- nagy helyigény (sok pointer),
- 1 Lépés elég sok kis elemi műveletből áll.

Ezért most definiáljuk a *Párosítós kupac*ot, amely a gyakorlatban általában gyorsabb, mint a Fibonacci, főleg a később leírt lustább változat. Azonban nem találtak hozzá olyan potenciált, amivel be tudnák látni ugyanazokat az amortizációs időket, mint a Fibonacci-kupacnál, sőt, később kiderült, hogy ilyen nem is létezhet.

*Ötlet*: A Fibonacci-kupacot akarjuk utánozni, de bináris fával és kevesebb pointerrel. Először emeljük fel a $H$ gyökeret és helyére fűzzük be a gyerekeit. Majd forgassuk el 45 fokkal az egészet, ez lesz a párosítós kupac. Tehát itt a bal gyerek felel meg az „első" gyereknek, a jobb gyerek a jobb testvérnek.

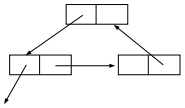



A párosítós kupac definíciója:

- Bináris fa.
- A gyökérnek (ami $H$-nak felel meg) nem lehet jobb gyereke.

- Félig kupacrendezett: $K(v) \leq K(x)$ a $v$ bal fiának minden $x$ leszármazottjára.

Így csúcsonként akár 2 pointert is nyerhetünk, ha használjuk még a bináris fák memóriatakarékos megvalósítását: csúcsonként 2 pointer és 1 bit, mely azt jelöli, hogy bal vagy jobb gyerek (ez a bit egyébként is hasznos, látni fogjuk, hogy bináris fákban sokszor fontos ez az információ).



1. pointer:

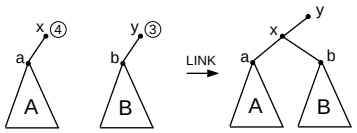
- bal gyerek, ha létezik
- jobb gyerek, ha csak az létezik.
- *nil*: ha egyáltalán nincs gyereke.

2. pointer:

- ha ő maga bal gyerek, akkor a testvérre mutat, ha létezik, különben a szülőre.
- Ha jobb gyerek: a szülőre mutat.
- Gyökérben *nil*.

Könnyű Hf: egy eredeti pointer szerinti mozgáshoz így két lépés és legfeljebb három kiolvasás és vizsgálat kell.

Vissza a párosítós kupachoz: definiálnunk kell még a műveleteket.
*LINK művelet*: (ha pl. $K(x) \geq K(y)$)

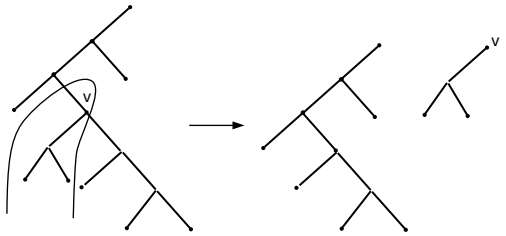



Idő: $O(1)$.

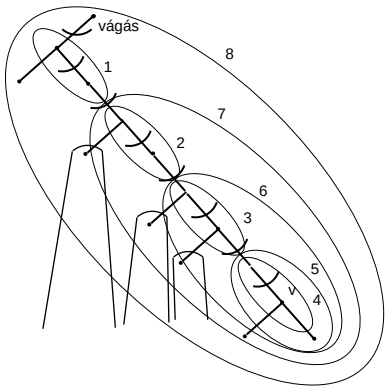
*Kupacműveletek*:

*Beszúrás*: 1 elemű új kupac + LINK. Ekkor $TI = O(1)$.
*Kulcs-csökk*:

- kivágjuk az elemet a bal fiának leszármazottaival együtt,
- csökkentjük a kulcsot,
- majd LINK. $TI = O(1)$.




*Mintörlés*:




A gyökeret levágjuk, majd az új gyökértől jobbra lefelé haladva minden élet elvágunk. Így sok kisebb párosítós kupacunk keletkezik. Össze kell LINKelni a szétvágott kupacokat.

*A leghatékonyabb megoldás*: Elindulunk lefelé és párosával összeLINKeljük a kupacokat, majd ezután alulról felfelé haladva végzünk LINKeléseket (mindig a következőt az előző eredményéhez).

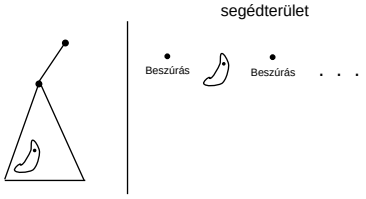
*Miért ez a legjobb?* Az a rossz, ha a két linkelendő kupacnak nagyon eltér a mérete. Az a jó, ha kb. egyenlő nagyságúakat linkelünk, ezt próbáljuk elérni. Másrészt vegyük észre, hogy az első fázisban egy útfelezésnek megfelelő dolgot csinálunk, melynek hatékonyságát már láttuk.

Ami bizonyítható: a $P = \sum_v \log(\#v \text{ leszármazottai})$ potenciállal $O(\log n)$ az amortizációs ideje minden műveletnek.

Azonban a gyakorlati futtatások sokáig azt mutatták, hogy igazából a Kulcs-csökkentések átlagos ideje konstans. Fredman '99-ben megmutatta. hogy ez mégsem igaz, konstruált olyan művelet-sorozatokat, melyekben átlagosan $\log \log n$ idő kell a Kulcs-csökkentésekhez.

### 6.2.1. A párosítós kupacok hatékony változata

*Ötlet*: Még lustábban. Nem jó kicsiket sokkal nagyobbakkal LINKelni. Ezt tipikusan a Beszúrás és Kulcs-csökk műveleteknél tesszük. Ezért ennél a két műveletnél LINKelés helyett egy külön segédterületre rakjuk őket.




A segédterületet tárolhatjuk a gyökér jobb leszármazottaiként. Most a Mintörlésnél kell egy kicsit többet dolgoznunk.

*Mintörlés*:

- Először a segédterületen végzünk LINK műveleteket. Párosával keletkezési sorrendben LINKelünk, majd *újra elölről* párosával, míg az egész segédterület 1 kupac nem lesz.
- Összelinkeljük az eredeti és a segédkupacot.
- Majd hagyományos Mintörlés.

### 6.2.2. A párosítós kupacok legújabb változatai

Elmasry 2009-ben mutatott be egy olyan változatot, ahol a műveletek bizonyítható amortizációs ideje $O(\log n)$, ezen belül a Kulcs-csökkentésé $O(\log \log n)$. A fő ötlet a következő: külön segédterületen tároljuk a Beszúrások 1 csúcsú fáit, és egy másikon a Kulcs-csökkentéseknél kivágott fákat, valamint egy pointert a minimális kulcsú gyökérre. Mintörlés után, illetve, ha a második segédterületen levő fák száma elérte a $\log n$-et, akkor mindent összefűzünk egy kupaccá. Az 1 csúcsúakat úgy, mint előbb, a nagyobbakat viszont úgy, hogy először rendezzük a gyökereket, majd csökkenő sorrendben összelinkeljük őket. Így végül a művelet előtti gyökerek a végső gyökértől szigorúan balra menő úton lesznek.

Haeupler, Sen és Tarjan szintén 2009-ben vezették be a Rang-párosítós kupacokat. Ebben minden művelet ugyanannyi időben (amortizált) megy, mint a Fibonacci kupacban, de lényegében megtartja a párosítós kupacok előnyeit, így a gyakorlatban is gyors. Minden csúcshoz nyilvántartunk egy rangot. A Beszúrásoknál, Kulcs-csökkentéseknél minden a

segédterületre kerül, a Mintörlésnél az elején (jobbra menő élek szétvágása során) keletkező kisebb kupacokat is ide rakjuk. Viszont linkelni csak azonos rangú gyökereket szabad, ilyenkor az új gyökér rangját eggyel növeljük. A rangra azt a tulajdonságot kell fenntartani, hogy egy $u$ csúcs gyerekeinél mindig igaz legyen, hogy vagy mindkettő rangja eggyel kisebb, mint $u$ rangja, vagy pedig az egyiké ugyanannyi, a másiké kisebb (akármennyivel). Emiatt a Kulcs-csökkentéseknél a kivágás után esetleg felfelé ezt a tulajdonságot ki kell javítani. Ezzel együtt is kijön a konstans amortizációs idő a Kulcs-csökkentésre és a Beszúrásra, és a Mintörlés amortizációs ideje is marad $O(\log n)$.

## 6.3. Szigorú Fibonacci-kupac

2012-ben Brodal, Lagogiannis és Tarjan megcsinálták a Fibonacci-kupac olyan változatát is, ahol ugyanazok a műveleti idők érhetők el, de nemcsak amortizált, hanem legrosszabb idő változatban is, tehát a Mintörlés legrosszabb esetben megy $O(\log n)$ időben, a többi művelet pedig legrosszabb esetben megy konstans időben.

Ez a struktúra is a Fibonacci-kupacon alapul, a változtatások lényege a következő:

A kupacrendezett fákat egyetlen kupacrendezett fában tároljuk, és itt segít, hogy feltesszük, hogy a kulcsok különbözőek, tehát igazából a (KULCS, név) párokat lexikografikusan rendezzük.

A fa csúcsai lehetnek aktívak és passzívak. Egy csúcs aktív gyökér, ha ő aktív, de szülője passzív (lényegében ezek felelnek meg a Fibonacci-kupac gyökereinek). A rang az aktív gyerekek száma. A jelölő-bit helyett minden aktív csúcsra nyilvántartunk egy nemnegatív VESZTESÉG értéket.

Legyen $R = 2\log n + 7$, ez lesz a felső korlát az aktív csúcsok rangjára. A következő (helyenként ravasz) invariánsokat tartja fent az algoritmus:

- A gyökér passzív, legfeljebb $R + 2$ gyereke van.
- Aktív gyökér vesztesége 0.
- Egy aktív csúcs $i$-edik aktív gyerekénél a rang és a veszteség összege *legalább* $i - 1$ (a gyerekek rendezettek, amikor $y$ az $x$ gyereke lesz, akkor első gyerek lesz, ha passzív, különben utolsó).
- Az aktív gyökerek száma legfeljebb $R$.
- A veszteségek összege maximum $R$.
- Egy $Q$ segéd-sorban a $p$-edik csúcs gyerekeinek száma maximum $2\log(2n - p) + 10$.

A részletek az eredeti cikkben olvashatók, amely elérhető a segédanyagokban.

## 6.4. Jobb algoritmus minimális költségű feszítőfára

Egy kis kitérőt teszünk, hogy bemutassuk a Fibonacci-kupac egy szép alkalmazását. A Prim algoritmus Fibonacci-kupaccal $O(m + n\log n)$ időben fut, ennél szeretnénk jobbat az $m < n\log n$ esetben.

Először megismerkedünk a legrégebbi minimális költségű feszítőfa algoritmussal, amely 1926-ból származik. Ehhez az éleken egy szigorú rendezés kell, ezt úgy érjük el, hogy egy $e = uv$ élhez (ahol feltesszük, hogy $u < v$) hozzárendeljük a $(c(e), u, v)$ hármast, és ezek szerint rendezünk lexikografikusan.

Borůvka algoritmusa: Minden csúcshoz rendeljük hozzá a legközelebbi szomszédját, ezeket építsük meg (házi feladat: nem keletkezik kör). Az így kapott erdőnek már legfeljebb $n/2$ fája van, húzzuk ezeket össze és ismételjünk.

Fredman és Tarjan 1984-es algoritmusa ezen, és a Prim algoritmuson, és az ugyanazon 84-es cikkükben bevezetett Fibonacci kupacon alapul. Vázlatosan: elindítjuk Prim algoritmusát, de csak addig futtatjuk, amíg a kupacméret $k = 2^{2m/n}$ alatt marad. Ha ez bekövetkezik, egy másik is helyről is elindítjuk Prim algoritmusát ugyanígy. Ha már nem tudjuk, akkor ez a fázis véget ér, az erdő fáit összehúzzuk és új fázist kezdünk.

Kicsit részletesebben: Az előző fázis végén kapott erdővel (elején az él nélkülivel) kezdünk, ennek fáit régi fáknak hívjuk, darabszámukat $t$-vel jelöljük. Ezeket össze kell húzni $O(m)$ időben, ez számjegyes rendezéssel megoldható. Először szélességi kereséssel minden $u$ csúcsra megkeressük az őt tartalmazó fa $1 \leq K(u) \leq t$ sorszámát, majd most egy $uv$ élhez a $(K(u), K(v))$ párt rendeljük, ahol $K(u) \leq K(v)$. Eszerint rendezzük lexikografikusan az éleket, két rögzített régi fa közötti élek egymás után lesznek, végigmenve ezek listáján könnyű kiválasztani a legkisebb költségűt. Ebből egyszerű felépíteni a $t$ csúcsú összehúzott gráf éllistáját, a csúcsait jelöletlenre állítjuk.

Ezután azt ismételjük, hogy amíg van jelöletlen régi fa, addig abból indítunk egy Primet (minden bekötött régi fát jelölünk), amit viszont leállítunk két esetben: ha belenőtt egy másik új fába (a most bekötött csúcs már jelölt volt), vagy ha a kupacméret elérte a $k = 2^{2m/t}$ korlátot (ekkor a most növelt új fából már legalább $k$ él lép ki). Így egy fázison belül minden Mintörlés legfeljebb $2m/t$ ideig tart, mivel legfeljebb $t$ ilyen lehet, ezért egy fázis ideje $O(m)$.

Most már csak a fázisok számát kell korlátozni. Állítás: ha a jelenlegi fázis elején $t$ régi fa volt és $m_0 \leq m$ éle volt az összehúzott gráfnak, valamint ha a következő fázis $k$ értékét $k'$-vel jelöljük, és a fázis során $t' > 1$ új fa keletkezett, akkor $t' \leq 2m_0/k$, hiszen könnyen láthatóan minden új fára legalább $k$ él illeszkedik. Innen $k' = 2^{2m/t'} \geq 2^k$. Mivel az első fázisban $k = 2^{2m/n}$, és ha az aktuális $k \geq n$, akkor ez az utolsó fázis, ezért a fázisok száma legfeljebb $\beta(m, n) + O(1)$, ahol $\beta(m, n) = \min\{i \mid log^{(i)} n \leq m/n\}$. Tehát az algoritmus futási ideje $O(\beta(m, n) \cdot m)$.

**Megjegyzés:** Azaz amennyiben $m \geq n \log\log n$, akkor a fázisok száma legfeljebb 2.

## 6.5. r-kupacok

Ha a kulcsok kis egészek, akkor tudunk jobb kupacot is csinálni. A naiv megközelítés a vödrös kupac: ha a kulcsok egészek a $[0, C]$ intervallumból, felveszünk $C + 1$ vödröt a kulcsokkal megcímkézve. Egy $i$ címkéjű vödörbe rakjuk az összes olyan rekordot, melynek kulcsa $i$. Ezeket egy kétszeresen láncolt listában tároljuk (minden vödörhöz). Beszúrásnál berakjuk a megfelelő vödörbe (a lista elejére, $O(1)$ lépés), Kulcs-csökkentésnél kifűzzük (ezért kell kétszeresen láncolt lista), csökkentjük, az új vödörbe berakjuk (ez is $O(1)$ lépés). A Mintörlésnél a nehézség az első nem-üres vödör megtalálása, a többi könnyű,

így a Mintörlés ideje legrosszabb esetben $O(C)$ lépés. Ez így egész jól működik a Prim-algoritmusra, ha a költségek egészek a $[0, C]$ intervallumból $O(m + nC)$ összes lépésben, de később sokkal jobbat fogunk csinálni (lásd a 10. fejezetet). Azonban a Dijkstra algoritmushoz így nem igazán jó, mivel ilyen költségek esetén a kulcsok akár $(n-1)C$ nagyságúak is lehetnek. A Dijkstra algoritmus tulajdonságait kihasználva ez is levihető $O(m + nC)$ lépésre (Dial, 1969). Vegyünk $nC$ vödröt, és vegyük észre, hogy a 6. tétel miatt két egymás utáni Mintörlés esetén a másodiknál nagyobb a minimális kulcs, de legfeljebb $C$-vel. Így egy Mintörlésnél legfeljebb $C$ jobbra lépés után (az előző Mintörlés kulcsa szerinti vödörtől indulva) találunk nem-üres vödröt. Most ezen fogunk lényegesen javítani.

**Megjegyzés:** Igazából megoldható Dial verziója $C + 1$ vödörrel is. Például tároljuk a $K$ kulcsú elemeket a $K \bmod (C + 1)$ sorszámú vödörben.

Tehát legrövidebb utakat keresünk, és feltesszük, hogy az élek hossza a $[0, C]$ tartománybeli egészek, azaz a kulcsok a $[0, (n - 1)C]$ tartományban lesznek.

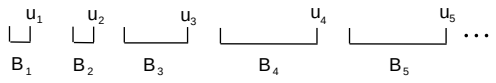
**Monoton kupac** tulajdonságai:

- Mintörlések kulcsai az idő előrehaladtával monoton nőnek
- Ha az utolsó Mintörléskor a kulcs $d_{min}$, akkor az összes aktuális kulcs eleme a $[d_{min}, d_{min} + C]$ tartománynak.

**Megjegyzés:** a 6. tétel miatt a Dijkstránál használt kupac ezeket teljesíti, tehát monoton.

Monoton kupacokat, ha $C$ nem túl nagy, az úgynevezett r-kupacokkal (r = radix, ill. redistributable) érdemes megvalósítani.

Legyen $c = \lceil \log(C + 1) \rceil + 2$.




Vödrök:

$B_1, B_2, \ldots, B_c$ vödröket készítünk. A $B_i$ vödörhöz az $u_i$ felső korlátot és a range $(B_i)$ számintervallumot rendeljük. Egy $B_i$ vödörbe azokat az elemeket rakjuk, amelyek kulcsa eleme a range $(B_i)$-nek.

range $(B_i) = [u_{i-1} + 1, u_i]$.

- mindig fenntartjuk: $|\text{range}\,(B_i)| \leq 2^{i-2}$ , ha $i = 2, \ldots, c - 1$ és $|\text{range}\,(B_1)| = 1$. Az $u_i$ sorozat mindig monoton növő lesz, de nem szigorúan, így menet közben lehet olyan $i$, hogy range $(B_i) = \emptyset$.

*Kezdetben így állítjuk be az értékeket*:

$u_0$ = -1

$u_i = 2^{i-1} - 1$

$u_c = nC$ (ebbe a vödörbe biztosan belefér minden elem)

$\text{range}\,(B_1) = [0]$
$\text{range}\,(B_2) = [1]$
$\text{range}\,(B_3) = [2, 3]$
$\text{range}\,(B_4) = [4, 5, 6, 7]$
$\vdots$

A kupac elemeit úgy tároljuk, hogy $v$ a $B_i$-ben legyen, ha $K(v) \in \text{range}\,(B_i)$. Egy vödör tartalmát egy kétszeresen láncolt listában tároljuk.

*Műveletek*:

*Beszúrás*$(v)$:

Itt megkereshetnénk felező kereséssel is a megfelelő vödröt, azonban, mint a konvex burok keresésénél már tapasztaltuk, az rosszabb megoldást adna.

**for** $j = c..1 \ \ (-1)$
  **if** $u_{j-1} < K(v)$ **then** $v \to B_j$ ; $j := 0$

**Megjegyzés:** 1 elem beszúrásának tényleges ideje $O(c) = O(\log C)$.

*Kulcs-csökk*$(v, j, \Delta)$:
(híváskor nemcsak $v$ címét, hanem az őt tartalmazó vödör sorszámát is tudnunk kell)

- $K(v) := K(v) - \Delta$
- **if** $K(v) \leq u_{j-1}$ **then** kivesszük, elindulunk balra és megkeressük a jó vödröt.

*Mintörlés*:

- Ha $B_1 \neq \emptyset \to$ tetszőleges elemét töröljük és visszaadjuk.
- Különben legyen $B_j$ az első nem-üres vödör (ezt persze meg kell keresnünk). Végignézzük a $B_j$ vödör minden elemét. Legyen $v$ az egyik minimális kulcsú és $d_{min} := K(v)$.
- Átcímkézzük $u_i$-ket:
  $u_0 := d_{min} - 1$
  $u_1 := d_{min}$
  $u_i := \min(u_{i-1} + 2^{i-2}, u_j)$, ha $i = 2, \ldots, j-1$.
- Majd $B_j$ minden elemét (egyesével balra mozgatva) berakjuk a neki megfelelő vödörbe, kivéve $v$-t, amelyet törlünk.

**16. Állítás.** *Ezután $B_j$ minden elemének a helye $B_i$ lesz valamilyen $i < j$-re, azaz a $B_j$ vödör tényleg kiürül.*

**Bizonyítás:** Mivel a $B_j$ előtti vödrök üresek voltak, az átcímkézés legális. Mivel a művelet végrehajtása előtt $d_{min} \in \text{range}\,(B_j)$, ezért $d_{min} \geq u_j - 2^{j-2} + 1$. Emiatt az átcímkézések után $u_{j-1} = \min(u_j, d_{min} + 1 + 2 + \cdots + 2^{j-3}) = \min(u_j, d_{min} + 2^{j-2} - 1) = u_j$. Így minden elemet tényleg balra kell mozgatni, mivel $\text{range}\,(B_j)$ üres lett. Mivel $B_j$-ben semelyik elem kulcsa nem kisebb, mint $d_{min}$, mindenkit jól el tudunk helyezni. (A $|\text{range}\,(B_i)|$ értékek nyilván teljesítik a megkívánt felső korlátokat minden $i \leq j$-re, máshol pedig nem változtatjuk őket.)

**Amortizációs elemzés**

*A potenciál*: $P = \sum_{v \in \text{kupac}} b(v)$, ahol $b(v)$ a $v$-t tartalmazó vödör sorszáma.
Egy Lépés legyen (először) 1 elem eggyel balra átrakásának ideje (kivesszük, ellenőrizzük, hogy odavaló-e, és ha igen, akkor berakjuk).

*Az egyes műveletek amortizációs ideje.*

*Beszúrás*$(v)$: $TI = (c + 1 - b(v))$
$\Delta P = b(v)$
Így $AI = c + 1 = \log C + O(1)$.

*Kulcs-csökk*$(v, \Delta)$: $TI = 1 + (b(v) - b_{\text{uj}}(v))$
$\Delta P = b_{\text{uj}}(v) - b(v)$.
Tehát $AI = 1$.

*Mintörlés*: Ha $B_1$ nem volt üres, akkor nyilván $AI = 0$. Egyébként:
$TI = j + |B_j| + j + \Big(1 + \sum_{u \in B_j, u \neq v} \big(j - b_{\text{uj}}(u)\big)\Big)$, ahol

$j$: $B_j$, az első nem-üres vödör megkeresése
$|B_j|$ = a $j$. vödörben levő elemek száma: végignézzük $B_j$ tartalmát és kiválasztjuk $v$-t,
$j$: $u_i$-k átcímkézése,
utolsó tag: balra pakolások ideje (a szumma előtti 1+ a $v$ kihagyására fordított idő).

**Megjegyzés:** $\sum_{u \in B_j, u \neq v} (j - b_{\text{uj}}(u)) \geq |B_j| - 1$, mivel $(j - b_{\text{uj}}(u)) \geq 1$ minden $u$-ra a 16. állítás miatt.

Egy Új Lépés := 2 · Lépés (tehát kétszer annyi elemi lépés), így most már

$$TI \leq j + |B_j|/2 + 1/2 + (1/2) \cdot \sum_{u \in B_j, u \neq v} \big(j - b_{\text{uj}}(u)\big) \leq j + 1 + \sum_{u \in B_j, u \neq v} \big(j - b_{\text{uj}}(u)\big)$$

Mivel $\Delta P = -j - \sum_{u \in B_j, u \neq v} \big(j - b_{\text{uj}}(u)\big)$, ezért $AI \leq 1$.

Tehát ha például $n$ Beszúrást, $m$ Kulcs-csökkentést, és $n' \leq n$ Mintörlést végzünk, akkor: $\sum TI \leq \sum AI \leq n \cdot (c + 1) + m + n' \leq n \cdot (\log C + 4) + m$ darab Lépés, így $O(m + n \cdot \log C)$ összes futásidőt kapunk (ha $C \geq 2$). Tehát a Dijkstra futási ideje $[0, C]$-beli egész hosszak esetén r-kupaccal: $O(m + n \cdot \log C)$.

## 6.6. Thorup kupaca

Ennél a kupacnál is jobbat készített *Thorup*, legalábbis elméletileg (gyakorlati megvalósításról és tesztekről nem tudunk):

Az ún. word RAM modellen dolgozunk: legyen $w$ a szóhossz az adott gépen (azaz egy memória cellába legfeljebb $w$ bites számok férnek). Feltesszük, hogy $w \geq \log n$, és $w \geq \log C$, ahol $C$ a hosszak maximuma (a hosszak itt is nemnegatív egészek). Tehát azt tesszük fel, hogy minden élhossz és egy csúcs neve belefér 1 szóba. Valamint feltesszük, hogy 1 Lépésben 2 szó összege és szorzata kiszámolható (ezeket a jegyzet utolsó fejezeteiben is fel fogjuk tenni).

Ezen feltevések mellett Thorup kupaca a Beszúrást és Kulcs-csökkentést $O(1)$ időben, a Mintörlést pedig $O(\log\log\min(n, C))$ időben elvégzi, ráadásul nem csak amortizációs időben, hanem garantáltan minden egyes műveletnél.

Ezzel a kupaccal a Dijkstra- és a Prim-algoritmus $O(m + n \cdot \log\log\min(n, C))$ idejű a word RAM modellben.

# 7. Szótárak

Legyen adott egy $U$ univerzum és egy ezen a halmazon értelmezett $<$ rendezés.

*Feladat*: adatstruktúra készítése $S \subset U$ elemek tárolására. $S$ (a pillanatnyi szótár) minden elemét csak egyszer tároljuk.

**Megjegyzés:** Igazából persze legtöbbször nemcsak magát az $s$ szótár-elemet kell tárolni, hanem $s$ mellé vagy egy pointert is az $s$ kulcsú rekordra; vagy, amennyiben ez a rekord az $s$-en kívül csak a $j(s)$ jelentést tartalmazza, akkor ezt a $j(s)$ szót is. Ezt a továbbiakban nem részletezzük, de az $s$ helyének megkeresése pont azt jelenti, hogy ezt is meg kell találnunk.

*Műveletek*:

*Keresés* $(x, S)$:

- Könnyített: $x \in ?S$
- Normál: keressük meg $x$ helyét, ha $x \in S$, különben pedig „NINCS”.

Ha egy szótár csak ezt az egy műveletet tudja megvalósítani, akkor *statikus szótárnak* nevezzük.

Ha a szótár nem statikus, akkor a második legfontosabb művelet a

*Beszúrás* $(x, S)$:

- Könnyített: garantált, hogy $x \notin S$, az $x$-et beszúrjuk, azaz $S := S + x$.
- Normál: ha $x \notin S$, akkor $S := S + x$, különben semmit sem csinálunk.

Ha egy adatstruktúrában mindkét művelet megoldott, akkor *szótárról* beszélünk.

*Törlés* $(x, S)$:
Ha $x \in S$, akkor $S := S - x$.

Ha mindhárom művelet megoldott, akkor az adatstruktúra *szótár törléssel*.

Ezen műveleteken kívül az alábbi műveletek is hasznos kiegészítők lehetnek:

- $min(S)$
- $max(S)$
- *Köv* $(S, a)$: visszaadja $a$-t, ha $a \in S$, különben pedig a legkisebb $b \in S$ szótárelemet, melyre $b \geq a$.
- *Tólig* $(S, a, b)$: $\{x \in S \mid a \leq x \leq b\}$
- *Fésül* $(S_1, S_2)$: Tudjuk, hogy $S_1 \cap S_2 = \emptyset$. Legyen $S := S_1 \cup S_2$.
- *Olvaszt* $(S_1, a, S_2)$: Tudjuk, hogy $S_1 < a < S_2$. Legyen $S := S_1 \cup \{a\} \cup S_2$.
- *Szétvág* $(S, a)$: Szétvágja $S$-et olyan $S_1$ és $S_2$-re, hogy $S_1 < a < S_2$.

Szótárak megvalósítása: Ha $|U|$ kicsi: tárolhatjuk karakterisztikus vektorban pointerekkel az elemeket. Az indexek $U$ elemei lesznek. Persze ha $U$ nem a $[0, |U|-1]$ természetes számokból áll, akkor kell egy $h$ függvény, mely $U$ elemeit ezekbe képezi, legtöbbször ilyet könnyen konstruálunk; de előfordulhat az is, hogy nem, ekkor sokszor érdemesebb az általános esetre vonatkozó algoritmusok egyikét használni.

*Mostantól feltesszük, hogy $|U|$ nagy.*

Statikus szótárak esetén $S$ elemeit rendezett tömbben tárolhatjuk és felező keresést használhatunk. Látni fogjuk, hogy ennél az általános megoldásnál sokszor jobbat is találunk.

## 7.1. Bináris keresőfa

Ez a felező keresés nyilvánvaló általánosítása. Két változatát ismerjük, az irodalomban többnyire a második változat szerepel.

1. *Külső tárolású*: a fa leveleiben vannak a rekordok, a többi csúcsban pedig $U$ elemei szerepelnek „Útjelzőkként", a $v$ csúcsban tárolt útjelzőt $u(v)$ jelöli, és az a jelentése, hogy az ennél kisebb-egyenlő elemeket a bal gyerek részfájában, a nagyobbakat a jobb gyerek részfájában kell tovább keresni. A *Keresés* eljárástól különböző visszatérési értékeket várunk. Először is egy logikai értéket, hogy a keresett szó szerepel-e a szótárban. Ha igen, akkor a helyére is szükségünk van. Ha nem szerepel, akkor is várunk valami helyre vonatkozó visszatérési értéket, amire a *Beszúrásnál* lesz szükségünk (statikus szótárnál ez felesleges). Meghívásnál a keresendő szón ($x$) kívül megadjuk a szótár nevét is, ami a gyökérre mutató $r$ pointer. Külső tárolású szótárnál *feltesszük, hogy nincsen egy-gyerekes csúcs* (mivel felesleges lenne; ezt a tulajdonságot persze fent is kell tartanunk). Keresőfáknál a szótár $S$ nevét azonosítjuk az őt tároló fa $r$ gyökerével.

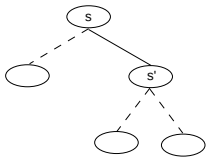
   **Megjegyzés:** A levelekben balról jobbra haladva $S$ elemei rendezve vannak.

   *Keresés*$(x, r)$:

   $v := r$
   **while** $bal(v) \neq nil$
   **if** $x \leq u(v)$ **then** $v := bal(v)$
   **else** $v := jobb(v)$
   **if** $x = u(v)$ **then return** $(VAN, v, bal)$ /* *mindegy, hogy jobb vagy bal, $VAN$ esetén nem használjuk* */
   **else if** $x < u(v)$ **then return** $(NINCS, v, bal)$
   **else return** $(NINCS, v, jobb)$.

2. *Belső tárolású*: a fa minden csúcsában tárolunk egy rekordot (megfeleltethetők $S$ elemeinek), illetve praktikusan általában csak a kulcsot és egy pointert a rekordra. A keresés során 3-felé elágazó **if** van, amit direktben elég kevés nyelv támogat

(pl. FORTRAN). Előnye viszont, hogy kevesebb tárhelyet foglal (kb. felét). Az elemzések céljából a fa minden egy-gyerekes csúcsa alá *képzeletben* odarakunk egy, minden levele alá pedig kettő *fiktív levelet*. Ezeket *nem tároljuk*, de a definícióknál és elemzéseknél hasznosak lesznek.




*Keresés* $(x, r)$:

$v := r$
**repeat**
  **if** $x = u(v)$ **then return** $(VAN, v, bal)$  /* *bal-jobb mindegy*
  **if** $x < u(v)$ **then**
    **if** $bal(v) \neq nil$ **then** $v := bal(v)$
      **else return** $(NINCS, v, bal)$
  **else**  /* $x > u(v)$
    **if** $jobb(v) \neq nil$ **then** $v := jobb(v)$
      **else return** $(NINCS, v, jobb)$

*A belső és külső tárolású fák összehasonlítása (abban az esetben, ha teljesen kiegyensúlyozottak)*:

| | Külső | Belső |
|---|---|---|
| Memória | $2n-1$ kulcs $+n$ rekord, és $6n-3$ pointer | $n$ kulcs $+n$ rekord, és $3n$ pointer |
| Mélység | $\lceil \log n \rceil$ | $\lceil \log(n+1) \rceil - 1$ |
| Keresés | $\lceil \log n \rceil + 1$ összehasonlítás | $\lceil \log(n+1) \rceil$ összehasonlítás, de 3 irányú! |

Bizonyos alkalmazásoknál néhány elemet sokkal többször keresünk, mint másokat. Ha ezeket kis mélységben tároljuk, akkor sokat nyerhetünk.

**Megjegyzések.**

- A három irányú összehasonlítás a legtöbb környezetben 2 lépés lesz!
- Az esetek túlnyomó részében belső tárolású keresőfát fogunk használni, mert az irodalomban így szokásos (a belső tárban elhelyezett szótáraknál a memória-igény nagyon fontos).

### 7.1.1. Műveletek általános bináris keresőfában

*Keresés* $(x, r)$: lásd fentebb.

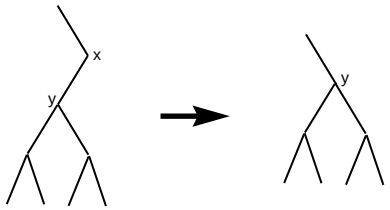
*Beszúrás* $(x, r)$: Először Keresés $(x, r)$.

- Külső tárolásúnál: ha $x \notin S$, akkor megkapunk egy $v$ levelet és egy irányt. Létrehozunk $v$ helyére egy új $w$ csúcsot, az iránytól függően $x$ vagy $u(v)$ útjelzővel, az irány szerinti gyereke egy $x$-et tartalmazó levél lesz, a másik gyereke pedig $v$.
- Belső tárolásúnál: ha $x \notin S$, akkor megkapunk egy $v$ csúcsot és egy irányt (ez így együtt egy fiktív levél „neve"). Létrehozzuk a $v$ csúcs irány szerinti gyerekét, és ebbe a levélbe tároljuk le $x$-et.

*Törlés*$(x, r)$:

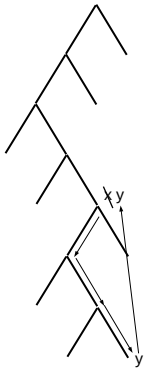
- Külső: triviális (megkeressük és kitöröljük), de a testvérét felrakjuk a szülőbe (ne legyen egy-gyerekes csúcs és felesleges útjelző tábla).
- Belső: Két esetet külön-külön vizsgálunk.

  1. eset Ha $x$-nek legfeljebb egy gyereke van, akkor a gyereke jön a helyére. Ekkor az új fa is jó keresőfa lesz.




  2. eset Ha $x$-nek 2 gyereke van:

     – Megkeressük az $x$-et közvetlenül megelőző $y$ elemet. Ezt úgy tesszük, hogy 1-et balra lépünk és utána amíg lehet mindig jobbra.




     – $x$-et és $y$-t felcseréljük, majd az új $x$-et töröljük az első módszer szerint (ennek már nincs jobb gyereke).

  Helyes az algoritmus, mert ha töröljük $x$-et, akkor $y$ már jó helyen lesz, azaz nem romlik el a keresőfa tulajdonság.

*Lépésszám*:

- A lépésszámok legrosszabb esetben a fa mélységével egyenlőek.
- Nem tudunk semmit a fa mélységéről!

- Ha például $n$ elemet szúrunk be növekvő sorrendben, akkor a fa mélysége $n$ lesz.

*Célunk* ezek alapján olyan fa kialakítása, amelynek mélysége $O(\log n)$.

**4. Definíció.** *Egy $n$ rekordot tartalmazó bináris fát (gyengén) kiegyensúlyozottnak hívunk, ha mélysége legfeljebb*

$$2 \cdot \lceil \log(n+1) \rceil + 1 \approx 2 \log n.$$

### 7.1.2. Optimális bináris keresőfa

Statikus szótárat akarunk tárolni úgy, hogy feltételezzük, hogy rengetegszer kell majd benne keresni. Ezen keresések összidejét akarjuk minimalizálni.
*Adottak*:

- $S = \{a_1 < a_2 < \cdots < a_n\}$ a szótár elemei.
- $a_i$-t $p_i$ valószínűséggel keressük.
- ha $a_i < b < a_{i+1}$, az ilyen $b$-ket $q_i$ valószínűséggel keressük.
    - ha $b < a_1$, akkor $b$-t $q_0$, ha pedig $a_n < b$, akkor $b$-t $q_n$ valószínűséggel keressük.

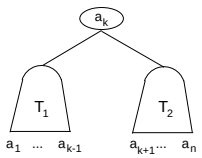
Egy adott $T$ keresőfa esetén egy keresés várható lépésszáma, melyet a fa *költségének* hívunk:

$$E(\text{keresés}) = c(T) := \sum_{i=1}^{n} p_i(d(a_i)+1) + \sum_{i=0}^{n} q_i d(i), \text{ ahol}$$

- $d(a_i)$: az $a_i$-t tartalmazó csúcs mélysége; a $+1$ azért kell, mert a gyökér keresése sem 0 lépés.
- $d(i)$: az $i$. (balról jobbra számozva) fiktív levél mélysége.

*Feladat*: Minimális költségű bináris keresőfa konstrukciója, azaz olyan $T$, amelyre $c(T)$ minimális.

**Megoldás**: Ha $a_k$ lenne az optimális $T$ fa gyökerében:




Ha ez az optimum, akkor szükségképpen $T_1$ és $T_2$ is optimális fa (a bennük szereplő $a_i$-k által meghatározott részfeladatra). Ez a szuboptimalitás elve.

**Megjegyzés:** Azok a feladatok, amelyekre a szuboptimalitás elve teljesül, többnyire megoldhatóak hatékonyan, erre való a dinamikus programozás, ahol először is jól definiálunk részfeladatokat, utána ezeket a megfelelő sorrendben kiszámoljuk, a régebbi részfeladatok megoldását jól felhasználva.

$T_{i,j}$ := optimális fa az $a_{i+1} \dots a_j$ szavakon. Ennek gyökere $r_{i,j}$, költsége $c_{i,j}$, súlya pedig $w_{i,j} := q_i + p_{i+1} + q_{i+1} + \dots + p_j + q_j$ ($w_{i,j}$ az a valószínűség, hogy belépünk egy $T_{i,j}$ fába).

*Inicializálás*:
$T_{i,i} = \emptyset$; $c_{i,i} = 0$; $w_{i,i} = q_i$.

Nekünk a $T_{0,n}$ fát kell megkeresni. Vegyük észre, hogy ha tudnánk, hogy $r_{i,j} = k$, akkor a szuboptimalitás elve miatt a $T_{i,j}$ fa gyökerének bal részfája $T_{i,k-1}$, jobb részfája pedig $T_{k,j}$ lesz. Ezért ekkor a költsége $c_{i,j} = (c_{i,k-1} + w_{i,k-1}) + p_k + (c_{k,j} + w_{k,j}) = c_{i,k-1} + c_{k,j} + w_{i,j}$ lesz, mivel a $T_{i,j}$ fában minden csúcs mélysége eggyel nagyobb, mint a $T_{i,k-1}$, ill. a $T_{k,j}$ fákban. Vegyük észre, hogy a költségben $w_{i,j}$ állandó, nem függ a $k$-tól.

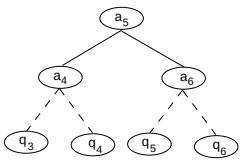
Ezek alapján könnyen készíthetünk egy rekurzív algoritmust:

$R(i,j)$ :
$c'_{i,j} := \infty$
**for** $k := i+1..j$
  $C_1 := R(i, k-1)$
  $C_2 := R(k, j)$
  **if** $C_1 + C_2 < c'_{i,j}$ **then** $c'_{i,j} := C_1 + C_2$; $r_{i,j} := k$
**return** $(c_{i,j} := c'_{i,j} + w_{i,j})$

Ez az algoritmus azért **exponenciális**, mert ugyanazt a dolgot sokszor számoljuk ki. A hívások száma az ún. Catalan-szám lesz, ami kb. $\frac{1}{c \cdot n \cdot \sqrt{n}} \cdot 2^{2n}$. Hogy ezt elkerüljük, dinamikus programozással oldjuk meg a feladatot.

A *dinamikus programozás* jellemzői:

- Jól kell definiálni, hogy milyen részfeladatokat akarunk megoldani.
- Amit már egyszer kiszámoltunk, azt feljegyezzük, hogy ne kelljen még egyszer kiszámolni.
- Jó sorrendben oldjuk meg az egyes részfeladatokat.




*Az algoritmus* (a fenti Inicializálás után), az argmin itt azt a $k$ értéket adja vissza, amelyre a min felvétetik:

**for** $l = 1..n$
  **for** $i = 0..n-l$
    $j := i + l$
    $c_{i,j} := w_{i,j} + \min_{i<k\le j}(c_{i,k-1} + c_{k,j})$
    $r_{i,j} := \mathrm{argmin}_{i<k\le j}(c_{i,k-1} + c_{k,j})$

*Idő*: $O(n^3)$.

**Megjegyzés:** Házi feladat végiggondolni a megfelelőjét külső tárolású bináris fákra.

## 7.2. 2-3 fák

*Definíció* (2-3 fa):

- Minden nem-levél csúcsnak 2 vagy 3 gyereke van (ha $n > 1$).
- Külső tárolású a fa (mi ezt tárgyaljuk, a belső tárolású változatot lásd pl. az [1] könyvben).
- Minden levél ugyanazon a szinten van. Így legfeljebb $\lfloor \log n \rfloor$ mélységű lesz a fa.
- Minden nem-levél csúcsban két útjelző táblát, egy szülő- és 3 gyerek-pointert tárolunk.

Tehát egy $v$ csúcsban tároljuk a $p(v)$ szülő pointeren kívül a $bal(v)$ bal gyerekre mutató pointert, az $u_1(v)$ első útjelzőt, a $koz(v)$ második gyerekre mutató pointert, az $u_2(v)$ második útjelzőt és a $jobb(v)$ harmadik gyerekre mutató pointert ($nil$, ha csak két gyereke van). Itt $u_1(v)$ egy tetszőleges olyan elem, hogy a bal gyerek alatti minden levélben szereplő KULCS $\leq u_1(v)$, de a második gyerek alatti minden levélben a KULCS $> u_1(v)$. Az $u_2(v)$ szerepe hasonló, ha csak két gyerek van, akkor értéke lehet pl. a szótár (vagy az univerzum) legnagyobb eleme.

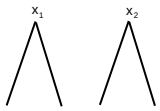
Egy $v$ levélnél jelölje most is $u(v)$ az ott tárolt rekord (szótárbeli) kulcsát.

*Keresés* $(x, r)$:
$v := r$
**while** $bal(v) \neq nil$
  **if** $x \leq u_1(v)$ **then** $v := bal(v)$
    **else if** $x \leq u_2(v)$ **then** $v := koz(v)$
      **else** $v := jobb(v)$
**if** $x = u(v)$ **then return** $(VAN, v)$
  **else return** $(NINCS, v)$

Ez mindig legfeljebb $O(\log n)$ lépés lesz, mivel a fa mélysége nyilván legfeljebb $\log n$.

*Beszúrás* $(s, r)$:

- Megkeressük, hogy hová kell beszúrni $s$-et, jelölje $x$ a keresési úton az utolsó nem-levél csúcsot, azaz $p(v)$-t.
- Ha $x$-nek 2 gyereke volt, akkor beszúrjuk $s$-et, és így $x$-nek 3 gyereke lesz, de $x$-ben átállítjuk az útjelzőket (máshol nem kell – Hf).
- Ha $x$-nek 3 gyereke volt, akkor szétszedjük $x$-et 2 darab 2 gyerekes csúccsá.




Ha $x$ szülőjének 2 gyereke volt, akkor nincs gond (csak ott kell útjelzőket állítani), ha 3, akkor ezt a csúcsot is szétvágjuk és így folytatjuk a gyökér felé. Ha eljutunk a gyökérig és a gyökérnek 3 gyereke volt, akkor a gyökeret is kettészedjük, beszúrunk egy új gyökeret, és így nő a fa magassága.

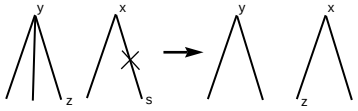
*Idő*: $O(\log n)$

Ehhez csak azt kell észrevennünk, hogy egy aktuális $x$ kettészedésénél tudjuk, hogy ki a szülő, $x$ neki hányadik gyereke, és magában $x$-ben tudunk (ideiglenesen) 3 elhatároló útjelzőt és 4 gyerek-pointert. Ezekből könnyen konstans időben kiszámolható a szétszedés utáni állapot a szülő új útjelzőivel együtt.

*Törlés* $(s, r)$:

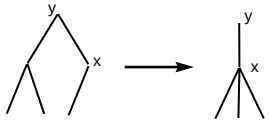
- Megkeressük az $s$-et tartalmazó levelet és kitöröljük.
- Ha $p(s) = x$-nek 3 gyereke volt, akkor készen vagyunk (csak $x$-ben kell útjelzőket és gyerek-pointereket állítanunk).
- Ha nem, akkor 2 esetet különböztetünk meg:

  1. Létezik $x$-nek szomszédos testvére 3 gyerekkel: elkérjük az egyiket (a felénk esőt).




     Itt pl. $u_2(y)$ jó lesz új útjelzőnek $p(x)$-ben $y$ és $x$ közé, $x$ pedig örökölheti $p(x)$ megfelelő útjelzőjét.

  2. Ha nem létezik, akkor $x$-et összevonjuk valamelyik szomszédos (és így kétgyerekes) testvérével.




     Lehet, hogy ekkor az új $x$ szülőjének lesz csak 1 gyereke, tehát felfelé folytatni kell. Hogyan ér véget az algoritmus?

     – A szülő 3 gyerekes volt, és most kettő maradt neki, ekkor megállhatunk.

– Sikerült elkérnünk egy 3 gyerekes szomszédos testvér egy gyerekét, ekkor is megállhatunk.
– Feljutunk egészen a gyökérig, töröljük az 1 gyerekes gyökeret és így csökken a fa magassága.

Házi feladat meggondolni, hogy ez is megy $O(\log n)$ időben, minden lehetséges esetben könnyű állítgatni az útjelző táblákat és a pointereket.

## 7.3. B-fák

Tulajdonságai:

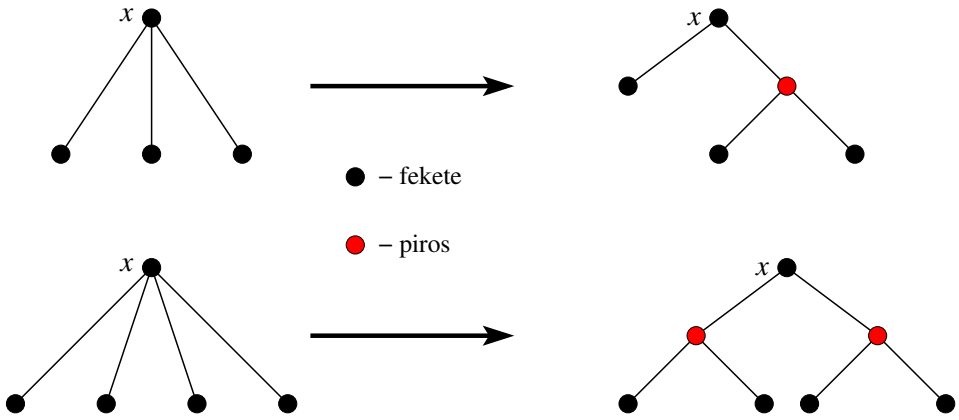
- Minden nem-levél csúcsra igaz, hogy $\left\lceil \frac{B}{2} \right\rceil \leq \#\text{ gyerekek} \leq B$ $\left(\text{kivéve ha } n < \left\lceil \frac{B}{2} \right\rceil\right)$.
- Egy csúcsban $B-1$ útjelző van.
- Minden levél ugyanazon a szinten van.
- Minden művelet a 2-3 fák mintájára megy, a lépésszám $O(B \cdot \log_B n)$.

Elsősorban külső táraknál (pl. hard disc) használják. Ezek a tárolók 1 olvasás során egy egész lapot olvasnak be. Az tart sokáig, míg a fejet a megfelelő lapra pozicionálja a vezérlés. Az a cél, hogy minél kevesebb lapot kelljen olvasni ahhoz, hogy megtaláljuk a tárolón keresett adatot. Úgy érdemes $B$-t megválasztani, hogy a $B-1$ útjelző és $B+1$ pointer elférjen egy lapon. És persze a levelekben sem egy, hanem annyi egymás utáni rekordot tárolunk, amennyi ráfér egy lapra.

*Példa*: Tegyük fel, hogy egy lapra ráfér 31 kulcs és 33 pointer, illetve 10 rekord. Ekkor ha $n = 10 \cdot 2^{32} \approx 43$ milliárd és $B = 32$, akkor a fa mélysége legfeljebb 8 lesz. Ha a felső 4 szintet a memóriában tároljuk (ez maximum $32^3 + 32^2 + 33 = 33825$ lap), akkor minden művelet legfeljebb 5 lapolvasás lesz (azaz legfeljebb 5-ször kell olvasni a winchestert).

## 7.4. Piros-fekete fák

*Eredetük*: Bináris faként tárolt 2-4 fák:




*Tulajdonságai*:

0. Belső tárolású bináris keresőfa,
1. minden csúcsa piros vagy fekete; a fiktív levelek és a gyökér feketék,
2. piros csúcs szülője fekete,
3. hogy kiegyensúlyozott legyen a fa: minden $x$ csúcsára igaz, hogy bármely alatta levő fiktív levélig vezető úton mindig ugyanannyi fekete csúcs van. Ezt $x$ fekete magasságának hívjuk.

Érdemes megjegyezni, hogy a fenti feltételek implicite tartalmazzák, hogy
(∗) ha egy $v$ csúcsnak csak egy valódi gyereke van, akkor az egy piros levél
($v$ másik gyereke egy fekete fiktív levél, ezért $v$-nek csak piros valódi leszármazottjai lehetnek, de a 2. és 3. tulajdonság miatt csak egy darab).

**10. Tétel.** *Egy piros-fekete fa mélysége legfeljebb* $2 \cdot \log(n+1)$.

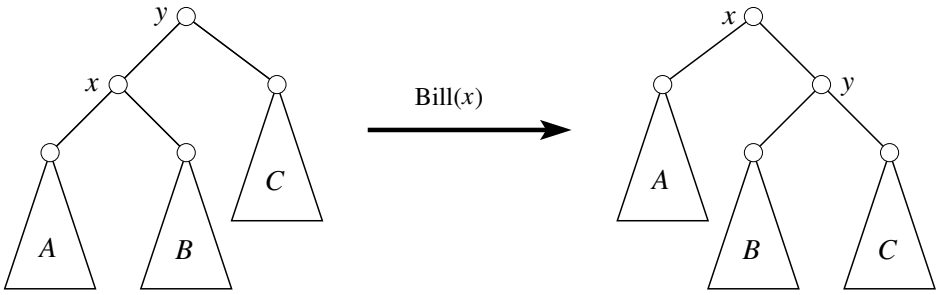
**Bizonyítás:** a piros csúcsok fekete szülőbe való behúzásával. Ekkor minden fiktív levél mélysége ugyanaz lesz, és minden nem fiktív csúcsnak legalább két gyereke lesz, tehát a mélység legfeljebb $\log(n+1)$, mert egy $n$ csúcsú bináris fának $n+1$ fiktív levele van. Mivel bármely úton maximum minden második csúcs lehet piros, ezért az eredeti mélység ennek legfeljebb a kétszerese.

*Műveletek*:

*Keresés*$(x, r)$: Ugyanúgy kell megvalósítani, mint egy tetszőleges bináris fában.

*Egy kiegészítő* **belső művelet** *a* **billentés**, *mely minden bináris keresőfában értelmezhető, és többször fogjuk használni*:

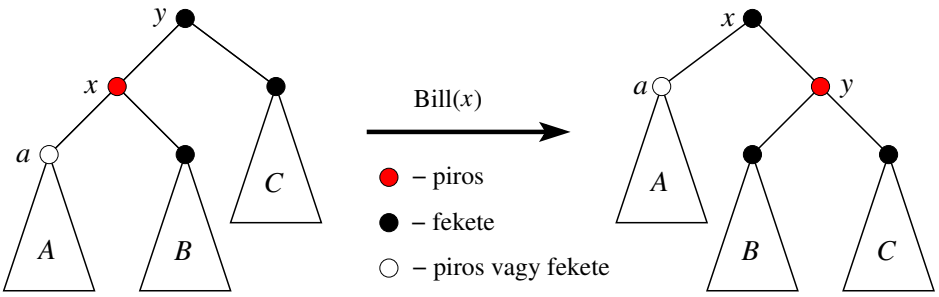
*Bill*$(x)$:



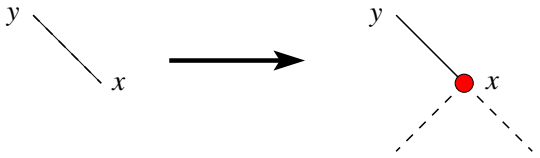

Könnyű végiggondolni, hogy ha egy általános bináris keresőfára alkalmazunk egy Billentés műveletet, akkor újra bináris keresőfát kapunk. Azonban piros-fekete fáknál ez nem mindig lesz jó, elronthatja a 2. és 3. tulajdonságokat. Ráadásul egy *kicsit elrontott* piros-fekete fára akarjuk alkalmazni, ami azt jelenti, hogy egyetlenegy csúcsra megengedjük, hogy a 2. tulajdonság ne teljesüljön. Gondoljuk meg, hogy a következő két esetben a Billentés ilyenkor is alkalmazható.

i) Ha $y = p(x)$ és $x$ is piros. Ekkor a 3. tulajdonság nyilván egy csúcsra sem romlik el, és továbbra is csak egy helyen nem fog a 2. tulajdonság teljesülni.

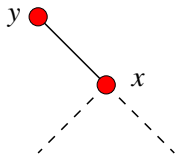
ii) Ha $y$ fekete, $x$ piros, de $x$ testvére és $y$ felőli gyereke (az ábrán a $B$ részfa gyökere) fekete. Ekkor azonban még át is kell színezni $x$-et és $y$-t, hogy teljesüljön a 3. tulajdonság. Lásd az ábrán, könnyű leellenőrizni, hogy ezután a 3. tulajdonság valóban megmarad, a 2. se romlik el sehol, sőt, ha $x$-nek a baloldali $a$ nevű gyerekénél volt elrontva, akkor az meg is javul.



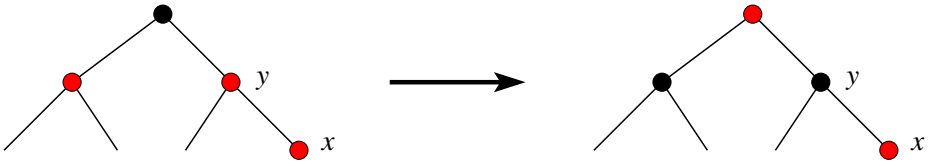

*Beszúrás* $(s, r)$: A beszúrás egy fekete színű $x$ fiktív levél (amit a Keresés $(s, r)$ visszaad) helyére történik, alatta létrejön két új fekete fiktív levél. Ezt a csúcsot ($x$-et) pirosra kell színeznünk a 3. tulajdonság fenntartása miatt.



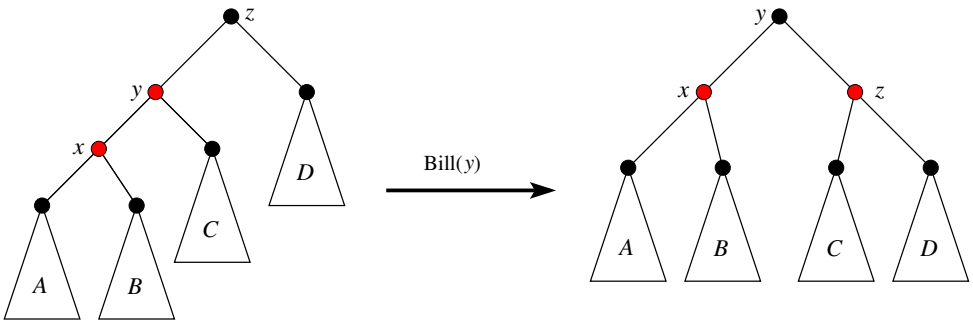

Akkor van probléma, ha a beszúrt $x$ elem $y$ szülője piros színű – ha fekete, vagy nem létezik ($x$ az elsőnek beszúrt csúcs – ekkor feketére színezzük), akkor készen vagyunk. Ezen belül három esetet különböztetünk meg:



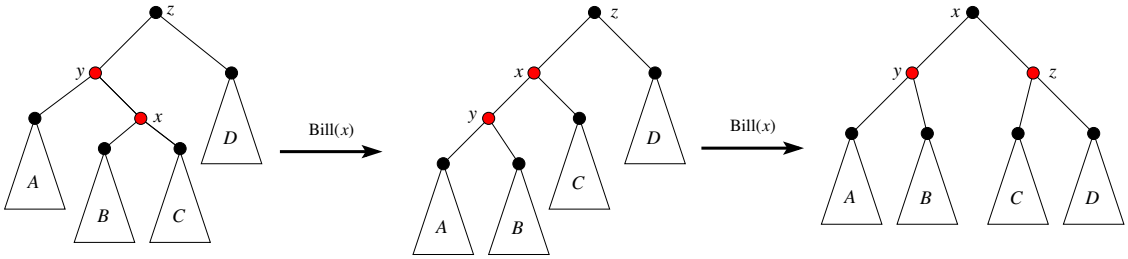

*1. eset*: Ha $y$ testvére piros. Ekkor átszínezünk: $y$-t és testvérét feketére, $y$ szülőjét (ha nem a gyökér) pirosra. Ezután vagy készen vagyunk, vagy $y$ nagyszülője piros volt, ekkor $x := p(y)$ és folytatjuk felfelé az eljárást (ez legfeljebb $O(\log n)$ lépést jelent).




*2. eset*: Az $y$ testvére fekete (lehet fiktív levél is).
*2a. eset*: Ha $p(p(x)) = z$-nek az $x$ bal-bal, vagy jobb-jobb unokája. Ekkor 1 Billentés után leállhatunk.



*2b. eset*: Ha nem a 2a. eset áll fenn. Ekkor is elég 2 Billentés:

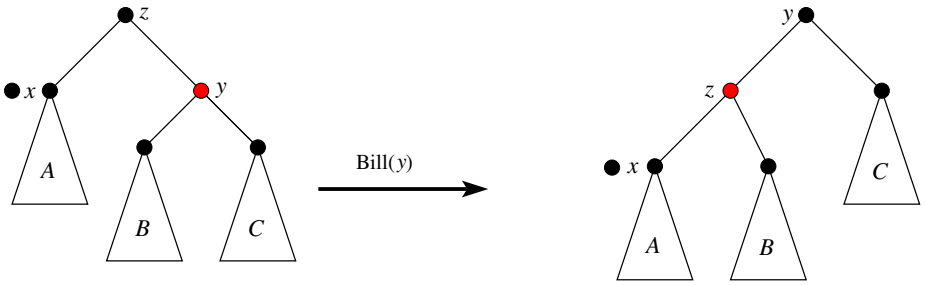



**17. Állítás.** *A 2a. esetben tett billentés, ill. a 2b. esetben tett dupla billentés után helyreállnak a tulajdonságok.*
Összefoglalva*: Egy* Beszúrás $(s, r)$ *utáni helyreállítás legfeljebb* $O(\log n)$ *átszínezés és legfeljebb 2* Billentés *művelettel megvalósítható.*

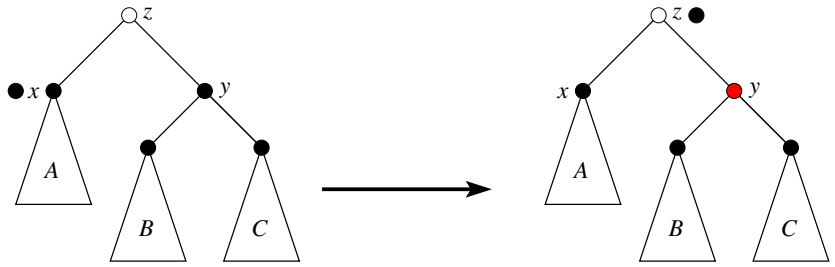
*Törlés* $(s, r)$: Először végrehajtjuk a hagyományos törlést, mely egy esetleges csere után törli ki az $s$ csúcsot, amelynek legfeljebb 1 gyereke van (jelölje ezt $x$; ha nincs gyereke, jelölje $x$ a helyére kerülő fiktív levelet). Ha $s$ piros volt, akkor készen is vagyunk. Ha $s$ fekete, és a helyére kerülő $x$ piros, akkor $x$-et átszínezve feketére szintén készen vagyunk. Az alfejezet elején tett $(*)$ megjegyzés miatt, mindig ez az eset áll fent, ha $s$ nem valódi levél volt.

Tehát az az eset maradt, amikor $s$ egy fekete valódi levél. Ekkor egy $x$ fiktív levél kerül a helyére. Jelöljük $z$-vel a törlés után $x$ szülőjét, és $y$-nal a törlés után $x$ testvérét (ez nyilván létezik, és nem fiktív levél). Sajnos $z$-ből az $x$ felé lefele vezető utakon eggyel kevesebb fekete csúcs van, mint az $y$ felé lefele vezető utakon. Ekkor $x$-et tekintjük problémás csúcsnak, neki kétszeresen feketének kellene lennie, hogy a tulajdonságok teljesüljenek. Ezért adunk neki egy fekete pontot. Ezt a fekete pontot próbáljuk meg felfele vinni, ha felvittük egy piros csúcsba, azt feketére színezzük és leállhatunk. Ha pedig felér a gyökérbe, és az fekete, akkor elfelejthetjük (ekkor a fa fekete magassága eggyel csökken).

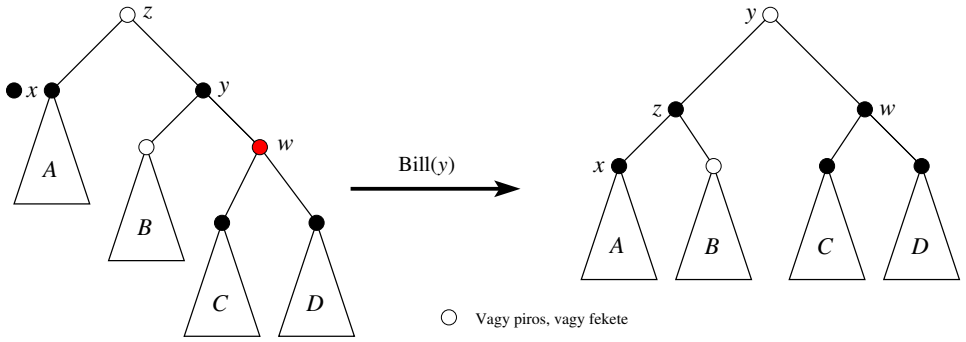
*1. eset*: $y$ piros. Ekkor $z$ is és $y$ gyerekei is feketék. Bill $(y)$ után már $x$ új testvére fekete, így folytathatjuk a következő esettel, de most már $x$ szülője biztosan piros.



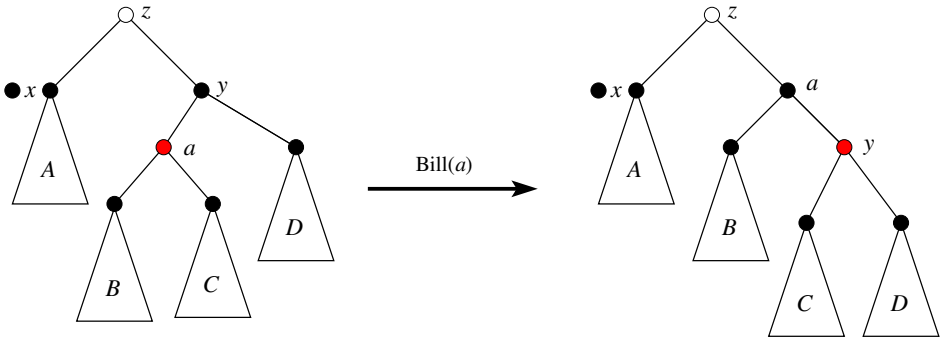

*2. eset*: $y$ fekete. A fentiek miatt $y$ nem fiktív levél.
*2a. eset*: $y$ gyerekei feketék. Ekkor színezzük $y$-t pirosra, és a fekete pontot vigyük át $x$-ről $z$-re, és ha $z$ fekete volt, akkor folytassuk felfelé az eljárást (különben $z$-t feketére színezzük és leállunk).




*2b. eset*: $y$-nak az $x$-szel ellentétes oldalon levő $w$ gyereke piros. Ekkor Bill $(y)$ után $w$-t feketére színezve (valamint $y$ és $z$ színét felcserélve) készen vagyunk.




*2c. eset*: $y$-nak az $x$-szel ellentétes oldalon levő gyereke fekete, a másik, $a$ gyereke piros. Ekkor egy Bill $(a)$ segítségével visszavezetjük az előző esetre.




**18. Állítás.** *Legfeljebb $O(\log n)$ vizsgálat és átszínezés és felfele lépés után legfeljebb 3 Billentéssel megoldható a művelet.*

Ehhez csak azt kell észrevenni, hogy egyedül a 2a. eset után kell felfelé folytatnunk, és csak akkor, ha $z$ fekete volt. Az első eset után (és persze a 2b., 2c. esetek után) nem kerülünk ebbe a részesetbe.

## 7.5. AVL-fák (Adelszon-Velszkij és Landisz)

*Tulajdonságai*:

- Bináris keresőfa,

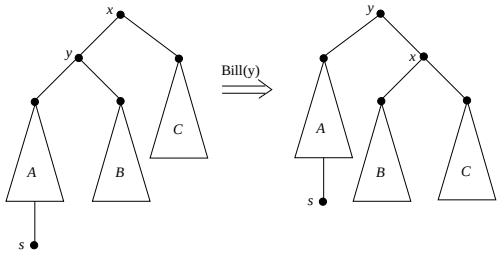
- $m(v)$ a $v$ csúcs magassága, $m(f) = -1$, ha $f$ fiktív levél,
- $\forall x : \ |m(bal(x)) - m(jobb(x))| \leq 1$.

Ezen tulajdonság ellenőrzése és fenntartása csúcsonként 2 bittel biztosítható: 1. a bal gyerek magasabb-e mint a jobb gyerek, illetve 2. a jobb gyerek magasabb-e mint a bal gyerek.

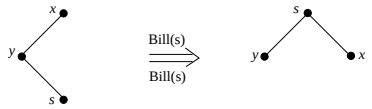
Indukcióval könnyen bizonyítható, hogy ha ezek a tulajdonságok teljesülnek egy $d$ mélységű fára, akkor ennek legalább $F_{d+3} - 1$ csúcsa van. Ebből következik, hogy a fa mélysége legfeljebb $1,44 \cdot \lceil \log n \rceil$ lehet.

*Műveletek*:

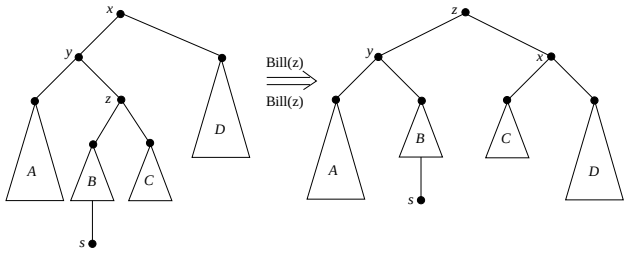
*Beszúrás*$(s, r)$: Beszúrjuk $s$-et, majd megkeressük a legközelebbi $x$ ősét, ahol a 3. tulajdonság elromlott; közben persze a fenti jelzőbiteket megfelelően átállítjuk. Majd itt egy szimpla vagy egy dupla billentéssel a tulajdonság helyreállítható az alábbi módon:
*1. eset*: $s$ az $x$ bal-bal, vagy jobb-jobb unokájának a leszármazottja




*2a. eset*: $s$ az $x$ bal-jobb, vagy jobb-bal unokája




*2b. eset*: $s$ az $x$ bal-jobb, vagy jobb-bal $z$ unokájának valódi leszármazottja




Nyilvánvaló, hogy egy beszúrásnál az $x$ elem helyének megkeresése legfeljebb $\lceil 1,44 \cdot \log n \rceil$ fellépést kíván, így az egész művelet végrehajtása legfeljebb $O(\log n)$ bit átállítással, valamint legfeljebb 2 Billentéssel jár. Érdemes megjegyezni, hogy $x$ megkeresését már a Beszúrás előtt, miközben $s$ helyét keressük, könnyen elvégezhetjük. Azonban ekkor is az $s$ és $x$ közötti csúcsoknál kell állítani a biteket. Azonban $x$ felett már nem, mert a billentések után azok magassága ugyanaz lesz, mint beszúrás előtt.

*Törlés*$(s, r)$: Legfeljebb $\lceil 1,44 \cdot \log n \rceil$ billentéssel megoldható, lehet, hogy az egész fát végig kell billegtetni (itt nem részletezzük).

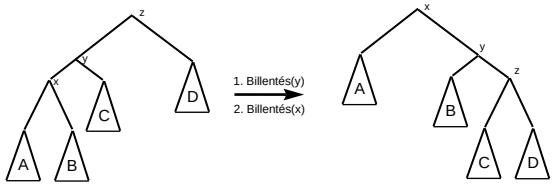
## 7.6. Önkiegyensúlyozó fák (S-fák; Sleator és Tarjan)

Itt is belső tárolású bináris fákat használunk, azonban a kiegyensúlyozottságra nem teszünk explicit feltételt, amit fent kellene tartani. Helyette időnként billegtetünk, és majd belátjuk, hogy átlagosan, azaz amortizációs időben minden művelet elég gyors lesz – azaz $O(\log n)$ idejű.

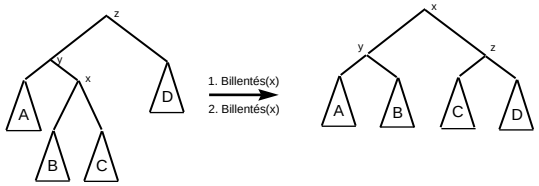
*Új belső műveletek*:

*Dbill* $(x)$: Kétfajta Duplabillentést különböztetünk meg.

(a) Ha $x$ a nagyszülőnek bal-bal, vagy jobb-jobb unokája (és $y = p(x)$):
Bill $(y)$, majd Bill $(x)$.




(b) Ha $x$ a nagyszülőnek bal-jobb, vagy jobb-bal unokája:
Bill $(x)$, majd újra Bill $(x)$.




*Felbillegtet* $(x)$:
**while** $p(p(x)) \neq nil$
Dbill $(x)$
**if** $p(x) \neq nil$ **then** Bill $(x)$

A felbillegtetés feltételezett hasznossága hasonlít az útfelezésére. Ha $x$ mélysége $h$ volt, akkor Felbillegtet $(x)$ után $x$ minden leszármazottjának mélysége $\lfloor h/2 \rfloor$-lel csökken. Más csúcsok mélysége egy alkalommal akár 2-vel is nőhet, de mivel ekkor $x$ leszármazottjává válnak, ezért ezután már minden további dupla billentésnél csökkenni fog.

Az alábbiakban a vesszős műveletek az új műveletek, a vesszőtlenek a hagyományos bináris fában végrehajtott műveletek.

Keresés' $(x, r)$:
Keresés $(x, r)$
IF $x$ nem fiktív levél THEN Felbillegtet $(x)$ ELSE Felbillegtet $(p(x))$

Beszúrás' $(x, r)$:
Beszúrás $(x, r)$
Felbillegtet $(x)$

Törlés' $(x, r)$:
$z := p(y)$, ahol $y$ az $x$ megelőzője (lásd az általános törlési algoritmust)
($y = x$, ha $x$-nek legfeljebb 1 gyereke van)
Törlés $(x, r)$
Felbillegtet $(z)$

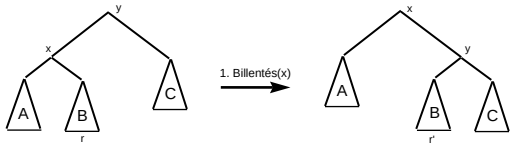
*Idő*: Mindhárom művelet tényleges ideje nagyjából kétszerese a Felbillegtet tényleges idejének. A Lépést először úgy választjuk meg, hogy annyi elemi lépést tartalmazzon, mint egy dupla billentés, és ezzel kielemezzük a Felbillegtet művelet amortizációs idejét. Majd végül megduplázzuk a Lépés konstansát, ekkor egy művelet tényleges ideje legfeljebb 2+ a Felbillegtetésnél végrehajtott (dupla vagy szimpla) billegtetések száma.

*Potenciál*:
$x$ súlya: $w(x) :=$ $x$ leszármazottainak száma (saját magát is beleértve)
$x$ rangja: $r(x) := \lfloor \log w(x) \rfloor$
A potenciál: $P := \sum r(x)$

**11. Tétel.** $AI(\textit{Felbillegtet}\,(x)) \leq 1 + 3 \cdot (r(\textit{gyökér}) - r(x)) \leq 1 + 3 \cdot \lfloor \log n \rfloor = O(\log n)$.

**Bizonyítás:** Belátjuk minden egyes billentő lépésre, hogy teljesül a fenti állítás megfelelője. Összeadva ezeket egy teleszkopikus összeget kapunk, amelyből minden tag kiesik, kivéve $1+3\cdot(r(\text{gyökér})-r(x))$. A továbbiakban $r$ jelöli a billentés előtti, míg $r'$ a billentés utáni rangot.

1. Szimpla billentés: ($y = p(x)$ gyökér):
   $AI \leq 1 + 3 \cdot (r(y) - r(x))$.




   $\Delta P = r'(x)+r'(y)-r(x)-r(y) = r'(y)-r(x) \leq r(y)-r(x) \leq 3\cdot(r(y)-r(x))$, mivel $r'(x) = r(y)$ és $r'(y) \leq r(y)$ és $r(y) \geq r(x)$. Tehát $AI = 1+\Delta P \leq 1+3\cdot(r(y)-r(x))$.

2. (a) típusú Duplabillentés:

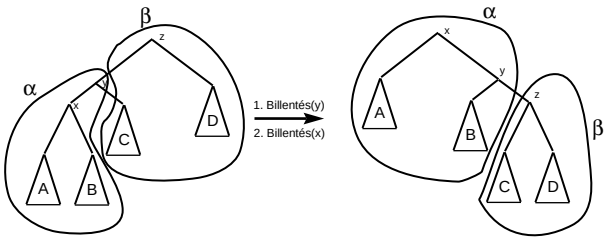
   Be kell látnunk, hogy $AI \leq 3 \cdot (r(z) - r(x))$.

   $\Delta P = r'(z) + r'(y) + r'(x) - r(z) - r(y) - r(x) \leq 2 \cdot (r(z) - r(x))$, mivel $r'(z)$ és $r'(y) \leq r'(x) = r(z)$, és $r(y) \geq r(x)$.

   Ha $r(z) > r(x) \Rightarrow$ ekkor $3 \cdot (r(z) - r(x)) \geq 2 \cdot (r(z) - r(x)) + 1 \geq \Delta P + TI$.

   Ha $r(z) = r(x) = r$ (ez az alsó-egészrész miatt lehetséges) $\Rightarrow r(y) = r$ és $r'(x) = r$.

   Elég belátni, hogy $r'(z) < r$, mert akkor $\Delta P \leq 2r + r'(z) - 3r \leq -1$, és így mivel $TI = 1$, ezért $AI \leq 3 \cdot 0$.



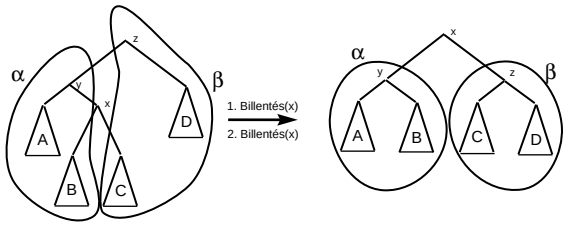

$\alpha$ és $\beta$ jelentse a bal oldalon bejelölt adott részfák csúcsainak a számát. Nyilván a jobb oldalon bejelölt részfák csúcsszáma is $\alpha$, ill. $\beta$.

Ekkor: $w(z) = \alpha + \beta$.

Indirekt tegyük fel, hogy $r'(z) = r \Rightarrow \beta \geq 2^r$. Mivel $r(x) = r \Rightarrow \alpha \geq 2^r$, innen: $\alpha + \beta \geq 2^{r+1} \Rightarrow r'(x) \geq r + 1$, ami ellentmondás.

3. (b) típusú Duplabillentés:

   Minden a 2. ponthoz hasonlóan megy.




Most $r'(y) < r$ vagy $r'(z) < r$ -et kell megmutatnunk.

Indirekt tegyük fel, hogy $r'(y) = r'(z) = r \Rightarrow \alpha \geq 2^r$, $\beta \geq 2^r \Rightarrow r'(x) \geq r + 1$, ami ellentmondás.

*Végső elemzés.* Most duplázzuk meg a Lépés konstansát.

$AI(\text{Keresés'}) \leq \Delta P(\text{Keresés}) + \Delta P(\text{Felbillegtet}) + 2 + TI(\text{Felbillegtet})$
(mivel $TI(\text{Keresés})$ belefér a Felbillegtet tényleges idejébe). Így, mivel $\Delta P(\text{Keresés}) = 0$, ezért $AI(\text{Keresés'}) = O(\log n)$.

$AI(\text{Törlés'}) \leq \Delta P(\text{Törlés}) + \Delta P(\text{Felbillegtet}) + 2 + TI(\text{Felbillegtet})$
(mivel $TI(\text{Törlés})$ belefér a Felbillegtet tényleges idejébe). Így, mivel $\Delta P(\text{Törlés}) < 0$, ezért $AI(\text{Törlés'}) = O(\log n)$.

$AI(\text{Beszúrás'}) \leq \Delta P(\text{Beszúrás}) + \Delta P(\text{Felbillegtet}) + 2 + TI(\text{Felbillegtet})$
(mivel $TI(\text{Beszúrás})$ belefér a Felbillegtet tényleges idejébe). Így, ha belátjuk, hogy $\Delta P(\text{Beszúrás}) = O(\log n)$, akkor $AI(\text{Beszúrás'}) = O(\log n)$.

$x$ beszúrásakor $w(y)$ csak akkor nő, ha $x$ az $y$ leszármazottja. $r(y)$ csak akkor nő, ha $y$ a gyökérből $x$-be vezető úton van és $w(y) = 2^k - 1$ alakú volt. Mivel az úton felfelé $w(y)$ szigorúan nő $\Rightarrow$ csak $\leq \lfloor \log n \rfloor$ ilyen lehet $(k = 1 \ldots \lfloor \log n \rfloor) \Rightarrow \Delta P(\text{Beszúrás}) \leq \log n$.

*Összegezve*: Minden művelet amortizációs ideje $O(\log n)$.

Ez az adatstruktúra nagyon jól alkalmazható például a *Szétvág*$(S, a)$ műveletre is: $a$-t felbillegtetjük és a két részfa lesz a megoldás.

**1. Megjegyzés.** *Az elemzés szó szerint megy akkor is, ha a szótárelemekhez saját pozitív egész súlyok tartoznak, ezek összegére felső becslés az $n$, és $w(x)$ definíciójában a leszármazottak saját súlyait adjuk össze.*

# 8. Hashelés

*Cél*: A szótárműveleteket akarjuk megvalósítani úgy, hogy átlagosan jó megoldást kapjunk. Ehhez először is egy olyan $h : U \to [0, 1 \dots M-1]$ függvényt keresünk, amely az $x \in U$ elemekre ad egy címet. A célunk az, hogy ezen a címen tároljuk $x$-et. $M$-et táblaméretnek hívjuk, $N$ pedig az aktuálisan tárolt adatok számát jelöli; és $\alpha := \frac{N}{M}$.

Az az egyik – intuitíve nyilvánvaló – cél, hogy a $h$ függvény egyenletesen terítse szét az inputokat, vagyis $\Pr(h(x) = i) \approx \frac{1}{M}$ legyen minden $i < M$ számra. Ha itt a valószínűség az univerzumon egyenletes eloszlás szerint értendő, akkor erről persze általában könnyű gondoskodni. Azonban itt a valószínűséget az inputnak érkező kulcsokon kell érteni, amelyeken egy (általában előttünk csak legjobb esetben is körülbelül ismert) eloszlás van, amely szerint érkezni fognak. Az elemzésekben fel fogjuk tenni, hogy sikerült olyan hash függvényt találni, amelyre a $\Pr(h(x) = i) = \frac{1}{M}$ feltétel $(*)$ teljesül. Persze az igazi véletlen nem jó, minden $K$ kulcsra és minden végrehajtáskor $h(K)$-nak ugyanazt az értéket kell visszaadnia. Azonban ez a megkívánt „véletlenszerűség" se garantál önmagában túl nagy sikert, mint azt a jól ismert születésnap paradoxon mutatja.

*Születésnap paradoxon*: Ha $N \geq \sqrt{\ln 4 \cdot M}$ és $h$ egy véletlen függvény, akkor legalább $\frac{1}{2}$ valószínűséggel lesz olyan $x \neq y$, hogy $h(x) = h(y)$. Ezt *ütközésnek* nevezzük.

Két dolgot vizsgálunk: hogyan keressünk jó hash függvényt és hogyan kezeljük az ütközéseket.

Persze mindenekelőtt az $U$ univerzum elemeit valahogyan természetes számokra célszerű leképeznünk. Ez általában könnyen megoldható úgy, hogy különböző kulcsoknak különböző számok feleljenek meg. Ezt a továbbiakban feltételezzük, tehát innentől úgy tekintjük, mintha $U$ elemei legfeljebb $O(|U|)$ nagyságú természetes számok volnának. Persze ez nem mindig egyszerűen megoldható, pl. lehetnek az univerzum elemei bizonyos, legfeljebb 32 hosszú stringek. Megjegyezzük, hogy ilyenkor a gyakorlatban jól használhatóak (az univerzum egész számokra leképezéséhez, de akár végleges hash függvényként is) a különböző ellenőrző-összeg számoló eljárások (pl. a szabvány CRC 32 bites egészekre képez, az Md5sum 128 bitesekre). Persze a célunk igazából injektív leképezés megtalálása, de az alkalmazásokhoz elég, ha minden természetes számnak kevés ősképe van.

## 8.1. Klasszikus hash függvények

A cél gyorsan számolható $h$ függvény keresése, melyre a $(*)$ feltétel „várhatóan" nagyjából teljesül.

1. Osztó-módszer:

   $h(K) = K \bmod M$ , ahol K a kulcs (egy szám) és M egy prím.

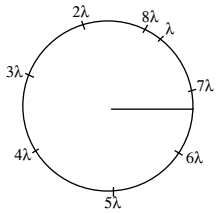
2. Szorzó-módszer:

   $h(K) = \left\lfloor \left\{ \frac{A}{W} \cdot K \right\} \cdot M \right\rfloor$, ahol $\{\}$ a törtrész, $M = 2^m$ és $W = 2^w$ alakú (általában $w$ a word RAM gépünk szóhossza). Ilyenkor a $W$-vel való osztás, a törtrész, az $M$-el

való szorzás és az egészrész bitléptetésekkel/levágásokkal gyorsan megvalósítható; igazából csak az $A \cdot K$ szorzás kiszámítása igényel lényeges időt.

Az $A$ érték megválasztása: az lenne a legjobb, ha $\frac{A}{W} = \lambda < 1$ irracionális lenne, de persze ez nem lehet.

*Szemléltetés*: egy egységnyi kerületű körön $K$-szor mérjük fel egymás után $\lambda$-t.




**12. Tétel.** *(T. Sós Vera): Ha $\lambda$ irracionális, akkor a szomszédos pontok közötti távolságok összesen 3 félék lesznek és a következő $\{(K+1) \cdot \lambda\}$ az egyik legnagyobb ívet osztja ketté.*

Ezen belül a legegyenletesebb akkor lesz a felosztás, ha $\lambda = \frac{\sqrt{5}-1}{2}$, ekkor a legnagyobb intervallumot a következő elem az aranymetszés arányában osztja fel.

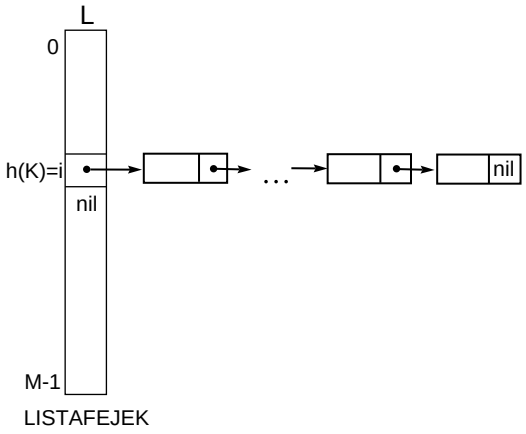
Ezek alapján $A$ megválasztása: úgy válasszuk $A$-t, hogy $\frac{A}{W} \sim \frac{\sqrt{5}-1}{2} \approx 0,618$ legyen.

3. Általános:

   Tulajdonképpen bármilyen, gyorsan számolható álvéletlenszám-generátor megfelel.

## 8.2. Ütközések feloldása

### 8.2.1. Láncolt (vödrös) hashelés




Képzeljünk el $M$ db vödröt $0..M-1$ címkékkel, a $K$ kulcsot a $h(K)$ címkéjű vödörbe rakjuk. Egy-egy vödröt láncolt listában tárolunk. A láncolt lista adatrészei:

- egy-egy teljes rekord, vagy
- mutató a rekordra és a rekord kulcsa

*Keresés*$(K)$: a $h(K)$-adik vödröt végigjárjuk.

*Beszúrás*$(K)$:

- Ha tudjuk, hogy nincs a táblában, akkor a $h(K)$-adik lista elejére szúrhatjuk (1 lépés).
- Ha nem tudjuk, akkor végigjárjuk a listát, és ha nem találtuk, akkor a végére beszúrjuk.

*Törlés*$(K)$: egyszerű (keresés közben az előző elemet megjegyezzük).

*Idő*:

A Lépést úgy definiáljuk, hogy beleférjen egy kulcs és egy pointer kiolvasása és a $h(K)$ függvény kiszámítása.

$C'_N$: a sikertelen keresés várható Lépésszáma, ahol $h(K) = i$ pontosan $\frac{1}{M}$ valószínűséggel minden $i$-re. (Megjegyzés: az $(N+1)$-edik elem beszúrása Lépésszámának is ezt tekinthetjük.)

$C_N$: a sikeres keresés várható Lépésszáma, ha minden tárolt elemet $\frac{1}{N}$ valószínűséggel keresünk. Ennek becsléséhez már fel kell tennünk, hogy az eddigi Beszúrásoknál igaz volt, hogy a beszúrandó kulcsokra $\Pr(h(K) = i) = \frac{1}{M}$ teljesül minden $i$-re. (A későbbi elemzéseknél a $C'_N$ becsléséhez is fel kell ezt tenni a régebben Beszúrt kulcsokra is, itt még nem.)

Emlékeztető: $\alpha = \frac{N}{M}$.

Legyenek $k_0, k_1, \ldots k_{M-1}$ a kialakult láncok hosszai, nyilván ezek átlaga $\alpha$.

Ekkor: $C'_N = \sum_{i=0}^{M-1} \frac{1}{M}(k_i + 1) = \frac{\sum k_i}{M} + \sum \frac{1}{M} = \frac{N}{M} + 1 = 1 + \alpha.$

A $\xi_i$ valószínűségi változó jelentse azt, hogy az $i$. elem beszúrásához hány lépést tettünk meg (a keresése is ennyi lépés lesz).

Ekkor: $E(\xi_i) = C'_{i-1}$ és $C_N = E(\sum \frac{1}{N}\xi_i) = \frac{1}{N}\sum E(\xi_i) = \frac{1}{N}\sum C'_{i-1}$ a várható érték linearitása miatt. Innen:

$$C_N = \sum_{i=1}^{N} \frac{1}{N}\left(1 + \frac{i-1}{M}\right) = 1 + \frac{1}{N}\sum_{i=1}^{N}\left(\frac{i-1}{M}\right) = 1 + \frac{\frac{N\cdot(N-1)}{2}}{NM} = 1 + \frac{N-1}{2M} \approx 1 + \frac{\alpha}{2}.$$

A későbbiekben szükségünk lesz még az alábbi következményre is:

$$E\left(\sum k_i^2\right) = 2 \cdot E\left(\sum \frac{k_i \cdot (k_i+1)}{2}\right) - E\left(\sum k_i\right) = 2 \cdot E\left(\sum_{j=1}^{N} \xi_j\right) - N =$$

$$= N + \frac{N^2 - N}{M} < (1+\alpha) \cdot N.$$

*A láncolt hashelés előnyei*:

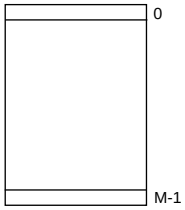
- egyszerű,
- mindig tudunk új elemet berakni, azaz nem telik be (amíg tudunk helyet foglalni a memóriában),
- a nagyon sok elemet tartalmazó vödörbe nem megy nagyobb valószínűséggel elem, mint egy üres vödörbe,
- könnyű törölni,
- ha például $\alpha \approx 1$, akkor $C_N \approx 1,5$ és $C'_N \approx 2$,
- külső táras megoldásként kifejezetten jó. Ekkor a listafej tömb a memóriában van, a pointerek a külső tár lapjaira mutatnak, és egy lapon annyi rekordot tárolunk, amennyi elfér (meg egy pointernek is kell hely!). *Például ha egy lapra* 10 *rekord fér és* $\alpha$*-t* 8*-nak választjuk, akkor a lapelérések várható száma sikeres keresésnél* $1,4$*, beszúrásnál* $1,8$ *alatt marad.*
- Ha sokkal több keresés van, mint egyéb művelet, akkor célszerű lehet a láncolt listákat valamilyen értelemben rendezve tárolni.
  - Ha kulcs szerint növekvően rendezve tároljuk, az a sikertelen kereséseket gyorsítja meg, ha a láncban nagyobb kulcsot találtunk, mint a keresett, akkor megállhatunk.
  - Ha tudjuk a keresési valószínűségeket, és ezek szerint csökkenő sorrendben tároljuk a rekordokat, akkor a sikeres keresések várható ideje lesz kisebb.
  - Ha ezeket nem tudjuk, de feltesszük, hogy egy időben állandó eloszlás által determináltak, akkor érdemes mindig a megtalált rekordot a lista elejére rakni.
  - Ha az ismeretlen eloszlás időben is változik, akkor a legjobb recept az, hogy a megtalált rekordot cseréljük fel a megelőzővel (lásd a [4] könyvben).

*Hátránya*:

- nagy memóriaigény: $M + N$ darab pointer kell (külső tárolásnál kevesebb),
- minden beszúrásnál új memóriarekeszt kell allokálni, és főleg egy lista elemei összevissza lesznek a memóriában, ezért a listán végigmenés a gyakorlatban elég lassú (nem tudjuk jól kihasználni a gyorsítótárat). Ez persze javítható több tárat használva, ha egyszerre több elemnek foglalunk egy vödörben helyet.

## 8.3. Nyílt címzés

A nyílt címzéses módszerekben közös, hogy mindig egy fix táblát használunk (tömböt) és feltesszük, hogy $N \leq M - 1$ (mindig hagyunk egy üres helyet, hogy a sikertelen keresésnél ne kelljen két feltételt vizsgálni).




A hash függvény: $h : U \to [0, M-1]$ mellett több másik hash függvényt is használunk: $h_i : U \to [1, M-1]$, ahol $i = 1, \ldots M-1$ és $h_0 \equiv 0$.

A következő úgynevezett „keresési sorozatot" vizsgáljuk:
$h(K)-h_0(K), h(K)-h_1(K), \ldots h(K)-h_{M-1}(K) \bmod M$, ahol a $h_1(K), h_2(K), \ldots h_{M-1}(K)$ sorozat kötelezően az $1, 2, \ldots M-1$ egy permutációja minden $K$-ra.

*Keresés*: kiszámítjuk a sorozatban adott címeket addig, amíg vagy megtaláljuk a keresett elemet, vagy pedig találunk egy üres helyet az adott táblában (ekkor nincs benne).

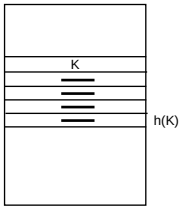
*Beszúrás*: ahol a sikertelen keresés leállt: a sorozat első üres helyére.

**Megjegyzések.**

- A nyílt címzés név onnan származik, hogy nincs előre eldöntve, hogy hová rakjuk az egyes elemeket.
- Annál jobb lesz az eredmény, minél "véletlenebb" a permutáció minden egyes $K$ kulcsra.

## 8.4. Lineáris hashelés

A második hash függvény: $h_i \equiv i$. Ennek előnye, hogy nem függ a kulcstól, így nem kell kiszámolni. Ha a tömbben felfelé lépkedve elérünk a legfelső elemig, akkor alulról indulunk tovább. (Ezért szerepel a hash függvényben kivonás összeadás helyett, a 0-val gyorsabban tudunk összehasonlítani.)




*Előny*:

- Gyors és egyszerű.

*Lépésszámok*: $C_N = \frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$ és $C'_N = \frac{1}{2} \cdot \left(1 + (\frac{1}{1-\alpha})^2\right)$.
$C_N$ és $C'_N$ néhány értéke:

| $\alpha$ | $\frac{1}{2}$ | $\frac{2}{3}$ | 0,8 | 0,9 |
|---|---|---|---|---|
| $C_N$ | 1,5 | 2 | 3 | 5,5 |
| $C'_N$ | 2,5 | 5 | 13 | 50,5 |

*Hátrányok*:

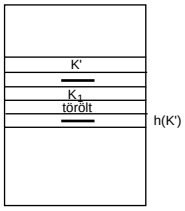
- Ha kialakul egy hosszú telített rész (*lánc*), akkor nagy annak a valószínűsége, hogy a következő elem is ebbe a láncba érkezik.
- A láncok össze tudnak nőni (tehát nagyon érzékeny a jó $h$ választására, az ismeretlen input eloszlás korrelációi is fontosak).
- Ha nincs előre jó becslésünk $N$ maximális értékére és emiatt betelik a tábla, akkor nincs jobb megoldás, mint felvenni egy új, praktikusan kétszer akkora táblát (és persze egy új $h$ függvényt), és oda újra egyesével behashelni az eddigi elemeket. Ez minden nyílt címzéses módszerre igaz.

Ez azonban nem olyan nagy baj, ha amortizált időben számolunk. Sőt, érdemes már pl. $N \geq M/2$ esetén új, kétszer akkora táblát nyitni.

Gondoljuk meg az alábbiakat (erre a gondolatmenetre még szükségünk lesz később is). Ha a végső tábla mérete $M$, akkor $N \geq M/4$. Az összes táblánk együttesen $\leq 2 + 4 + \ldots + M < 2M \leq 8N$ helyet foglal el. A beszúrások összes száma (beleértve az új táblákba való beszúrásokat is) legfeljebb $N$ plusz a kisebb táblaméretek összegének $(2 + 4 + \ldots M/2 < M)$ fele, ez pedig legfeljebb $M/2 + N \leq 3N$. Tehát egy elemet átlagosan maximum háromszor szúrunk be (egy maximum 50%-os telítettségű táblába). Egy új táblába való átpakoláskor persze a régi táblán végig kell menni, hogy „összeszedjük" az addig beszúrt elemeket, de ez is csak $2 + 4 + \ldots + M/2 < M \leq 4N$ lépés.

Ennél a módszernél még elég egyszerűen lehet törölni.

*Törlés*: Az okoz gondot, ha a későbbiekben egy olyan $K'$-t keresünk, hogy $K'$ a törölt elem felett (ciklikusan értendő!), $h(K')$ pedig alatta van.




**Megoldás**: Elindulunk a törölt elemtől felfelé. Ha gondot okozó $K'$ elemet találunk, azt nyugodtan átrakhatjuk a törölt elem helyére, és $K'$ eredeti helyét tekintve törölt helynek, rekurzívan folytatjuk az eljárást. Ha viszont elérünk egy üres helyet, akkor nyugodtan megállhatunk, mindent rendbe raktunk.

## 8.5. Dupla hashelés

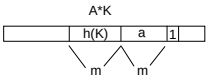
Itt a $h$ mellé csak egy darab $h'$ hash függvényt keresünk, melyre $h'(K) \in [1, M-1]$ és minden $K$-ra relatív prím $M$-hez. Legyen $h_i(K) = i \cdot h'(K)$.

Ez a módszer akkor lesz igazán jó (amit a további elemzésekben fel is teszünk), ha $\Pr\Big(h(K_1) = h(K_2)$ és $h'(K_1) = h'(K_2)\Big) \approx \frac{1}{M(M-1)}$ teljesül minden $K_1 \neq K_2$-re.

*Osztó-módszer*: $M' < M$ legyenek ikerprímek és $h'(K) := 1 + (\lfloor \frac{K}{M} \rfloor \bmod M')$.

**Megjegyzés:** a $K \bmod M$ kiszámolásakor a $\lfloor \frac{K}{M} \rfloor$ értéket is kiszámoltuk. A $+1$ pedig azért kell, hogy $i \cdot h'$ biztosan ne legyen 0.

*Szorzó-módszer*: Jelölje $a$ az $AK$ szorzatnak a $h(K)$-t követő $m$ bitjét. $h'(K) := a$ **or** 1 (az utolsó bitjét 1-re állítjuk, így nem lesz 0 és biztosan relatív prím lesz a kettő-hatvány $M$-hez).




*Lépésszámok*: $C_N = \frac{1}{\alpha} \cdot \left(\ln \frac{1}{1-\alpha}\right)$ és $C'_N = \frac{1}{1-\alpha}$. A 8.7. alfejezetben látható, hogy ezek a függvények miért „szeretnek" kijönni.

$C_N$ és $C'_N$ néhány értéke:

| $\alpha$ | $\frac{1}{2}$ | $\frac{2}{3}$ | 0,8 | 0,9 |
|---|---|---|---|---|
| $C_N$ | 1,39 | 1,65 | 2,01 | 2,56 |
| $C'_N$ | 2 | 3 | 5 | 10 |

**Megjegyzés:** Ez egy elég jó módszer (ha $h$ és $h'$ teljesítik a feltételeket), ennél sokkal jobbat nem is igen várhatunk a nyílt címzéseken belül.

*Törlés dupla hashelésnél*: Az előző trükköt itt nem alkalmazhatjuk. Amikor törlünk egy elemet, akkor egy speciális szimbólumot helyezünk el a helyére, ami azt jelöli, hogy az adott elem törölt. A beszúrásnál ide beszúrhatunk, de a keresésnél tovább kell lépnünk. Az a probléma, hogy sok törlésnél gyorsan megtelik a tábla ilyen szimbólumokkal.

## 8.6. A dupla hashelés Brent-féle változata

Ennél a módszernél $C_N \to 2,49$, ha $\alpha \to 1$. A beszúrásnál némi plusz munkát végzünk, hogy a majdani kereséseket gyorsítsuk. Jó pl. program fordítása során szimbólumtábla építésére.

**Megoldás**: Ha a beszúrandó $K$ kulcsot be tudom rakni az első vagy a második helyre, akkor berakom. Különben sima berakásnál az ő keresése már legalább három lépés lenne. Legyen a $h(K)$ helyet elfoglaló kulcs $K_1$. Ha $K_1$-et eggyel hátrébb tudom rakni a keresési sorában, akkor $K_1$ keresésén 1-et rontunk, de legalább 2-t javítunk $K$ keresésén.

Az algoritmus ez után is mohó módon folytatódik. Ha $K_1$-et nem sikerült a $h(K) - h'(K_1)$ helyre rakni, akkor utána $K$-t próbáljuk a $h(K) - 2h'(K)$ helyre rakni, ha ez se megy, akkor $K_1$-et a $h(K) - 2h'(K_1)$ helyre és $K$-t a $h(K)$ helyre, ha ez se megy, akkor $K_2$-t (aki a $h(K) - h'(K)$ helyet foglalja) próbáljuk eggyel hátrébb rakni a $h(K) - h'(K) - h'(K_2)$ helyre és $K$-t berakjuk a $h(K) - h'(K)$ helyre, és így tovább.

## 8.7. Egyenletes hashelés

Az egyenletes hashelés egy idealizált elméleti modell, nem tartoznak hozzá algoritmusok. Azt tesszük fel, hogy bármely pillanatban bármely rögzített $N$ helye a táblának egyforma, azaz $1/\binom{M}{N}$ valószínűséggel foglalt. Cserébe viszont itt a lépésszám-elemzések egyszerűek, és láthatjuk, hogy hogyan és miért jönnek ki a dupla hashelésnél megismert lépésszámok. Ugyanígy elemezhető a kissé jobban kézzel fogható uniform nyílt címzés is, amikoris azt tesszük fel, hogy minden kulcsra a $h(K) - h_0(K), h(K) - h_1(K), \ldots h(K) - h_{M-1}(K)$ mod $M$ keresési sorozat $1/M!$ valószínűséggel veszi fel bármely permutáció értékét. Ekkor ugyanis egy elem beszúrásakor mindegy, hogy éppen melyik $N$ sora foglalt a táblázatnak, mivel egy véletlen permutáció által adott keresési sorozat mentén próbálgatjuk beszúrni, ez ugyanaz, mintha egy fix sorrend mentén akarnánk beszúrni, viszont bármely $N$ sor egyforma valószínűséggel foglalt.

$P_k :=$ Pr( Az $(N+1)$-edik elem beszúrásához pontosan $k$ keresőlépés szükséges) $= \binom{M-k}{N-k+1}/\binom{M}{N} = \binom{M-k}{M-N-1}/\binom{M}{N}$. Ugyanis legyen az $N+1$. elem keresési címsorozata $h_1, \ldots, h_k, \ldots$. Ha a beszúráshoz $k$ keresőlépés kell, az pontosan azt jelenti, hogy a táblában $h_1, \ldots, h_{k-1}$ címek foglaltak, míg a $h_k$ cím szabad. Az elején tett feltevés szerint ennek valószínűsége pedig pont $\binom{M-k}{N-k+1}/\binom{M}{N}$. Felhasználva, hogy a valószínűségek összege 1, azaz hogy $\sum_{k=1}^{M} P_k = 1$, kapjuk:

$$C'_N = \sum_{k=1}^{M} k \cdot P_k = M + 1 - \sum_{k=1}^{M} (M+1-k) \cdot P_k =$$

$$M + 1 - \sum_{k=1}^{M} (M+1-k) \cdot \binom{M-k}{M-N-1} \Big/ \binom{M}{N} =$$

$$M + 1 - \sum_{k=1}^{M} (M-N) \cdot \binom{M+1-k}{M-N} \Big/ \binom{M}{N} =$$

$$M + 1 - (M-N) \cdot \binom{M+1}{M-N+1} \Big/ \binom{M}{N} =$$

$$M + 1 - (M-N) \cdot \binom{M+1}{N} \Big/ \binom{M}{N} =$$

$$M + 1 - (M-N) \cdot \frac{M+1}{M-N+1} = \frac{M+1}{M-N+1} =$$

$$1 + \frac{N}{M-N+1} \approx \frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$$

Ez alapján a sikeres keresés várható lépésszáma is meghatározható a láncolt hashelésnél látott alapelv alapján:

$$C_N = \frac{1}{N} \cdot \sum_{k=0}^{N-1} C'_k = \frac{M+1}{N} \cdot \left( \frac{1}{M+1} + \frac{1}{M} + \frac{1}{M-1} + \cdots + \frac{1}{M-N+2} \right) =$$

$$\frac{M+1}{N} \cdot (H_{M+1} - H_{M-N+1}) \approx \frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha},$$

ahol $H_n$ a természetes számok reciprokösszege $n$-ig, amiről ismert, hogy $H_n \approx \ln n + \gamma$, ahol $\gamma$ az Euler-állandó.

## 8.8. Univerzális hashelés

*Ötlet*: A hash függvényt válasszuk véletlenül egy elegendően nagy halmazból.

$\mathcal{H} : U \to [0, M-1]$ : legyen hash függvények egy halmaza.

$\mathcal{H}$ univerzális, ha $\forall K_1 \neq K_2 \in U$ -ra : $\Pr_{h\in\mathcal{H}}\big(h(K_1) = h(K_2)\big) \leq \frac{1}{M}$.

*Egy egyszerű konstrukció*: Feltesszük, hogy $M$ prím, és $K$-t felírjuk $M$ alapú számrendszerben:

$K = \sum_{i=0}^{r} x_i \cdot M^i$, ahol $0 \leq x_i < M$ és $r$-et úgy választjuk, hogy $K < M^{r+1}$ minden kulcsra.

Legyenek $a_0, a_1, \ldots a_r \in [0, M-1]$ véletlenek és függetlenek.

$h_{a_0,\ldots a_r}(K) := \sum_{i=0}^{r} a_i \cdot x_i \bmod M$. $\mathcal{H} := \{h_{a_0,\ldots a_r}\}_{0\leq a_0,\ldots a_r<M}$.

**13. Tétel.** *Ez univerzális, vagyis* $\Pr\big(\sum a_i \cdot x_i \bmod M = \sum a_i \cdot y_i \bmod M\big) = \frac{1}{M}$, *ha legalább egy* $j$*-re* $x_j \neq y_j$.

**Bizonyítás:** ha $K_1 \neq K_2$ (és $x_i$ reprezentálja $K_1$-et, $y_i$ pedig $K_2$-t), akkor valóban $\exists j$, hogy $x_j \neq y_j$.

Elég belátnunk, hogy a fenti egyezés feltételes valószínűsége $1/M$ minden rögzített $a_0, a_1, \ldots a_{j-1}, a_{j+1}, \ldots a_r$ értékre.

Ha egyezés van, akkor: $a_j(x_j - y_j) \equiv -\sum_{i=0,i\neq j}^{r} a_i(x_i - y_i) \bmod M$. Mivel $x_j - y_j \not\equiv 0$ és a jobb oldal a feltételes valószínűségben (rögzített $a_i$-k esetén, $i \neq j$) egy rögzített szám, így egyetlen $a_j$ létezik, ami kielégíti a kongruenciát.

**Megjegyzések:**

- Igazából ezt csak gyengén univerzálisnak szokás hívni, az erősen univerzális $\mathcal{H}$ család definíciója: $\forall K_1 \neq K_2 \in U$ -ra és $r_1, r_2 < M$-re: $\Pr_{h\in\mathcal{H}}\big(h(K_1) = r_1 \wedge h(K_2) = r_2\big) = \frac{1}{M^2}$.
- A fenti konstrukció apró módosítása ezt is tudja, ha még választunk egy véletlen $0 \leq b < M$ számot is, és ezt hozzáadjuk modulo $M$.
- Néha szükség van arra, hogy ne csak páronként legyenek függetlenek. Ha $\Pr_{h\in\mathcal{H}}\big(h(K_1) = r_1 \wedge \ldots \wedge h(K_k) = r_k\big) = \frac{1}{M^k}$ teljesül, akkor $k$-univerzális függvénycsaládról beszélünk. Ilyet látszólag könnyű csinálni, ha a fenti lineáris függvények helyett $k$-adfokú polinomokat veszünk. Azonban a függvények gyors kiszámíthatósága is követelmény.
- Ezért nagyobb $k$ értékekre (pl. $k = \log N$) már túlzott az ilyen precíz elvárás, így helyette a $(c, k)$-univerzális függvényeket szokás használni, itt az egyenlőség helyett csak azt követeljük meg, hogy a valószínűség legfeljebb $\frac{c}{M^k}$ legyen.

## 8.9. Tökéletes hashelés

A Fredman, Komlós és Szemerédi által kitalált technikának részletesen csak a statikus változatát vizsgáljuk, a dinamikusról csak röviden teszünk említést. Cél: adott egy $N$ elemű ismert, fix szótár. Építsünk fel $O(N)$ várható időben egy $O(N)$ tárhelyet foglaló struktúrát, hogy utána minden (sikeres ill. sikertelen) keresés ideje garantáltan $O(1)$ legyen. Ezt a látszólag lehetetlen feladatot részint a hashelésnél eddig is használt feltevések

($h$ kiszámítását konstansnak tekintettük, és feltettük, hogy egy kulcs elfér egy memóriarekeszben, azaz a word RAM modellben dolgozunk) miatt lehet megvalósítani, másrészt a felépítő algoritmusunk randomizált lesz, tehát csak a várható futási ideje lesz lineáris.

Először is rögzítünk egy $M \geq N$ prímet, választunk egy $h$ univerzális hash függvényt, és egy segédterületre láncolt hasheléssel behasheljük a szótár elemeit. Ezután kiszámítjuk a láncok $k_i$ hosszait, és ellenőrizzük, hogy $\sum k_i^2 \leq 4N$ igaz-e. Ha nem, akkor új $h$ függvényt sorsolunk, újra hashelünk, ellenőrzünk, s.í.t. Ha teljesül, akkor folytatjuk a második résszel.

Mielőtt a második részt részleteznénk, előbb nézzük meg az első rész várható futási idejét. Mint már bizonyítottuk, $E\left(\sum k_i^2\right) < 2N$, mivel $\alpha \leq 1$. Ezért a Markov-egyenlőtlenség miatt $\Pr\Big(\left(\sum k_i^2\right) > 4N\Big) < 1/2$. Tehát maximum $1/2$ valószínűséggel kell második $h$ függvényt sorsolni, maximum $1/4$ valószínűséggel kell harmadikat, s.í.t., tehát az egyébként lineáris időben végrehajtható hashelést (egy szótár elemeit szúrjuk sorba be, tehát lehet az új elemet ellenőrzés nélkül a láncok elejére rakni) várhatóan kevesebb, mint kettőször hajtjuk végre.

**Megjegyzés:** Igazából az sem baj, ha az inputban egyes szótárelemek többször is szerepelhetnek. A $\left(\sum k_i^2\right) > 4N$ feltétel ekvivalens azzal, hogy $\frac{1}{2}\left(\sum k_i(k_i+1)\right) > \frac{5}{2}N$, tehát azt is megtehetjük, hogy mindig ellenőrzéssel (és így a végére) szúrunk be (persze csak ha nincs már benne), és számoljuk, hogy az egyes elemek hányadikak lettek a láncukban. Ha ez az összesített szám meghaladja az $\frac{5}{2}N$-et, akkor az egészet eldobjuk, új $h$-t sorsolunk, és elölről kezdjük.

A végső struktúra úgy fog kinézni, hogy egy 0-tól $(M-1)$-ig indexelt $T$ tömbben nem láncfejek lesznek, hanem az $i$. sorában néhány szám, és egy link egy $T_i$ hashtáblára, melynek mérete $M_i \geq k_i^2$. A $T_i$ tábla tartalmazza a láncolt hashelésnél az $i$. láncba került elemeket. Ezekbe a táblákba újabb univerzális $h_i = \sum_{j=0}^{r_i} a_j^{(i)} \cdot x_j^{(i)} \bmod M_i$ hash függvényekkel hasheljük be ezeket az elemeket, ahol $x_j^{(i)}$ a $K$ kulcs számjegyei $M_i$ alapú számrendszerben. Arról fogunk gondoskodni, hogy egyáltalán ne legyen ütközés. Tehát minden $i$-re a kisorsolt $h_i$ függvénnyel behashelünk a táblába, és ha egyszer is ütközés van, akkor a táblát kiürítjük és új $h_i$ függvényt sorsolunk. Annak a valószínűsége, hogy van olyan két különböző kulcs, melyek ütköznek, legfeljebb $1/2$, mivel $\binom{k_i}{2}$ pár van, és egy pár $1/M_i \leq 1/k_i^2$ valószínűséggel ütközik. Tehát az ismétlések várható száma itt is kettő alatt van.

A $T$ tömb $i$. sorában tároljuk $k_i$ és $M_i$ értékét, az $a_j^{(i)}$ értékeket (ezek összhossza ugyanannyi, mint egy kulcsé), és a pointert a $T_i$ táblára. Persze $k_i \leq 1$ esetén nem érdemes külön táblát tárolni. Könnyű látni, hogy ez összesen $O(N)$ tár. (Megjegyzés: az univerzális hashelés fentebb tárgyalt konstrukciójánál feltettük, hogy az $M_i$ számok prímek. De ismert, hogy minden számhoz van nem sokkal nagyobb prím. Másrészt léteznek másfajta, nem prím $M_i$-kre is működő univerzális hashelések is, ott $M_i = k_i^2$ is választható (lásd az [1] könyvben)).

Világos, hogy ezután a Keresés$(K)$ így megy: kiszámítjuk $i = h(K)$ értékét. Kiolvassuk $T(i)$-t, kiszámítjuk $h_i(K)$-t, és megnézzük, hogy a $T_i$ tábla ennyiedik sorában $K$ van-e. Ha igen, megtaláltuk, ha nem, akkor $K$ nincs a szótárban. Ez persze $O(1)$ lépés.

### 8.9.1. Dinamikus tökéletes hashelés

Meglepő módon ez a módszer kiterjeszthető dinamikus szótárakra, még törlésekkel együtt is. Itt a célzott és elért lépésszámok a következők: a Keresés továbbra is legrosszabb esetben $O(1)$ lépés. A Beszúrás és Törlés *amortizációs idejének a várható értéke* pedig szintén konstans. És folyamatosan csak $O(N)$ tárat akarunk használni ($N$ most is az aktuálisan a szótárban levő szavak száma).

A dinamikus változat alapelvei: Fent kívánjuk tartani, hogy $N \leq M \leq 2N$, ahol $M$ a $T$ tömb mérete. Beszúrás$(K)$ esetén, ha $N > M$ lenne, akkor először is új struktúrát készítünk: $M$-et „megkétszerezzük", és az eddigi szótár elemeit a statikus esetnél látott módon átrakjuk az új struktúrába.

Ha $K$ helye üres a megfelelő $T_i$ táblában, akkor berakjuk. A $T$ táblában $k_i$ mellett az aktuális $T_i$ tábla $M_i$ méretét most is tároljuk, de ilyenkor csak $k_i$-t frissítjük. Ha $K$-t nem tudjuk berakni a $T_i$ táblába, mert ütközne, akkor két eset van. Ha az aktuális $M_i$ legalább négyzete az aktuális $k_i$-nek, akkor új $h_i$ függvényt sorsolunk. Ha nem, akkor $M_i$ értékét megduplázzuk (illetve választunk a kétszeresénél picit nagyobb prímet), és ellenőrizzük, hogy a $\sum M_i$ érték meghaladta-e $4M$-et. Ha igen, akkor a fentiek szerint $M$-et megkétszerezzük, és az eddigi szótár elemeit a statikus esetnél látott módon átrakjuk egy új struktúrába. Ha nem, akkor a régi $T_i$ elemeit új $h_i$ választásával átrakjuk az új, kétszer akkora $T_i$ táblába. A Törlés$(K)$ simán megy, de ha $N < M/2$ lett, akkor $M$-et megfelezzük, és az eddigi szótár elemeit a statikus esetnél látott módon átrakjuk egy új struktúrába.

## 8.10. Kakukk hashelés

Ezt a praktikus módszert Pagh és Rodler fejlesztette ki 2001-ben. Mi csak vázlatosan tárgyaljuk.

Választunk egy megfelelően univerzális (ld később) függvénycsaládot, és ebből függetlenül két függvényt, $h_1$-et és $h_2$-t. Azt kívánjuk fenntartani, hogy ha $K \in S$, akkor a táblában vagy a $h_1(K)$, vagy a $h_2(K)$ indexű helyen legyen, így a Keresés $O(1)$, mivel csak két helyet kell ellenőriznünk.

A Beszúrás$(K)$ lényegében így történik: ha $K \notin S$, akkor berakjuk a $h_1(K)$ helyre, az onnan kitúrt elemet tekintjük ezután $K$-nak. Ennek van pontosan egy másik lehetséges helye, ezért odarakjuk, és most már az onnan kitúrt elemet tekintjük $K$-nak, és folytatjuk. Ha egyszer is üres helyre rakjuk be, akkor sikeresek voltunk és megállunk.

Ez persze nem csak hogy sokáig tarthat, de végtelen ciklusba is kerülhet. Ezért jól beállítunk egy MAXLOOP nevű paramétert, és ennyi kitúrás után mást kell csinálnunk. Ez pedig új tábla felvétele, új $h_1$ és $h_2$ sorsolása, és minden elem áthashelése a fenti módszerrel az új táblába. Ha táblánk elég üres, akkor az új tábla ugyanakkora, különben pl. kétszer akkora.

Talán érezhető, hogy a módszer egyszerűsége ellenére az elemzés nem lehet triviális. Az eredeti verzióban MAXLOOP$= 3\log_{1+\varepsilon} N$ és $(2, 2{\cdot}\text{MAXLOOP})$-univerzális családot használtak.

Az alapelvet azonban szeretnénk bemutatni. Egy Beszúrás$(K)$ műveletnél definiáljuk a kakukk-gráfot, ennek csúcshalmaza $\{0, \ldots, M{-}1\}$, és $S \cup \{K\}$ minden eleméhez tartozik

egy él, ami a két lehetséges helyét köti össze. Azt szeretnénk, hogy két tetszőleges fix index csak kis (azaz $O(1/N)$) valószínűséggel legyen egy komponensben. Ezért indukcióval bizonyítjuk a következő lemmát.

**1. Lemma.** *Minden $i \neq j$ indexre, $\ell \geq 1$ egészre, és $c > 1$ valósra, ha $M \geq 2cN$, akkor annak valószínűsége, hogy a kakukk-gráfban $i$ és $j$ között vezet pontosan $\ell$ hosszú út, legfeljebb $1/(c^\ell M)$.*

**Bizonyítás**(vázlat): Az $\ell = 1$ esetben egy adott szótárelemre annak valószínűsége, hogy a két helye pont $i$ és $j$ legfeljebb $2/M^2$, így annak valószínűsége, hogy van ilyen elem legfeljebb $2N/M^2 \leq 1/(cM)$.

Rögzített $i$ és $j$ mellett legyen $B'_k$ az az esemény, hogy van $i$-ből $k$-ba pontosan $\ell-1$ hosszú út, $B_k$ pedig az az esemény, hogy van $i$-ből $k$-ba pontosan $\ell-1$ hosszú út amely nem tartalmazza $j$-t; valamint $A_k$ az, hogy $kj$ él.

$\Pr(B_k \wedge A_k) = \Pr(A_k|B_k) \cdot \Pr(B_k) \leq (2N/M^2) \cdot \Pr(B'_k) \leq (1/(cM)) \cdot (1/(c^{\ell-1}M)) = 1/(c^\ell M^2)$ az indukciós feltevés miatt. Így $\Pr(\cup_k(B_k \wedge A_k)) \leq 1/(c^\ell M)$.

Legyen most $C$ a kakukk-gráfban a $h_1(K)$-t tartalmazó komponens, és $K' \in S$. Ekkor $\Pr(h_1(K') \in C) \leq \sum_\ell \frac{1}{c^\ell M} = \frac{1}{(c-1)M}$. Így $C$ méretének várható értéke legfeljebb $2 + 2\frac{N}{(c-1)M} \leq 2 + \frac{1}{c(c-1)} \leq 11,1$ (ha pl. $c > 1,1$).

A fenti lemma körökre is működik, annak valószínűsége, hogy egy adott $i$ indexű csúcsa a kakukk-gráfnak benne van egy körben, legfeljebb $\sum_\ell \frac{1}{c^\ell M} = \frac{1}{(c-1)M}$. Innen az analízis úgy folytatódik, hogy az erősebb $M \geq 2(1+\varepsilon)cN$ feltétel mellett $\varepsilon n$ egymás utáni beszúrásnál az újrahashelések várható darabszáma konstans, így várható ideje $O(N)$, azaz elemenként amortizációs időben $O(1)$.

## 8.11. Bloom-filter

Ezt a napjainkban igen gyakran használt adatstruktúrát Bloom 1970-ben fejlesztette ki. Azóta több mint 60 változata létezik, és számtalan területen használják (néhány példa a fejezet végén található).

Mi az alapváltozatot tárgyaljuk. Itt is szótárat kell tárolni, de a KERESÉS helyett csak ELEME? művelet van, ami csak azt mondja meg, hogy az univerzum egy eleme benne van-e a szótárban. Ráadásul tévedni is szabad az egyik irányba: ha a helyes válasz IGEN, akkor mindenképpen azt kell visszaadnunk, de ha a helyes válasz NEM, akkor kis hiba-valószínűséggel adhatunk IGEN választ is. Az ELEME? műveleten kívül csak BESZÚRÁS van, TÖRLÉS nincs.

A cél kettős: egyrészt minden művelet menjen konstans időben, másrészt szeretnénk minél kevesebb memóriát használni.

Egy $B[0 : m-1]$ bit-tömböt használunk, valamint $h_1, \ldots, h_k : U \to [0, m-1]$ hash függvényeket (a $k$ konstans szerepéről persze még lesz szó, nagyjából $1/\varepsilon$, ha $\varepsilon$ hibát engedünk meg).

A $h_i$ függvények függetlenek, és mindegyik az univerzum bármely elemére nagyjából $1/m$ valószínűséggel adja bármely $j < m$ természetes számot. Tehát az elemzésnél feltesszük, hogy minden $K \in U$-ra a $h_1(K), \ldots, h_k(K)$ sorozat $1/m^k$ valószínűséggel adja bármely lehetséges sorozatot. Kezdetben $B$ minden eleme nulla.

BESZÚRÁS($K$): $B(h_1(K)) := B(h_2(K)) := \ldots := B(h_k(K)) := 1$.

ELEME?($K$): Ha $B(h_1(K)) = B(h_2(K)) = \ldots = B(h_k(K)) = 1$, akkor a válaszunk IGEN, különben NEM.

Nyilván ha a $K$ kulcsot már beszúrtuk, akkor mindig a helyes IGEN választ adjuk. Még ki kell számolnunk a hiba valószínűségét, azaz hogy $K \notin S$ esetén milyen valószínűséggel adunk helytelenül IGEN választ. Legyen $\gamma = \ln 2 \approx 0,693147$ és $k = \gamma \cdot m/n$, ahol $n = |S|$ (mint mindig).

A feltevésünk miatt egy elem beszúrásánál annak a valószínűsége, hogy bármely adott bit nulla marad, az $\left(1 - \frac{1}{m}\right)^k$, ezért $n$ elem beszúrása után egy bit nulla marad

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}} = e^{-\gamma}$$

valószínűséggel. Tehát ha a $K$ kulcs nem szerepel a szótárban, akkor annak a valószínűsége, hogy $B(h_1(K)) = B(h_2(K)) = \ldots = B(h_k(K)) = 1$ körülbelül

$$\left(1 - e^{-\gamma}\right)^k = \frac{1}{2^k} \approx 0,6185^{m/n}.$$

Tehát ha pl. a hiba valószínűségét le akarjuk vinni 1/1000-re, akkor $k$ értékét 10-nek és $m$ értékét $14{,}4 \cdot n$-nek választhatjuk, azaz $n$ elemű szótárhoz $14{,}4 \cdot n$ **bitet** használunk.

**Megjegyzés:** Persze nem is olyan egyszerű $k$ darab független véletlenszerű (de gyorsan számolható) függvényt találni. Ezért szokás azt is csinálni, hogy legyen $p$ egy prím, és csak két $h, h' : U \to [0, p-1]$ függvényt választunk (ahol $h'(K) \neq 0$.) Ezután így definiáljuk a $h_i$ függvényeket: $h_i(K) := h(K) + i \cdot h'(K) \mod p$. Most viszont a $B$ tömb $m = kp$ hosszú lesz, és a műveleteknél a $B(h_1(K)), B(h_2(K)) + p, B(h_3(K)) + 2p, \ldots, B(h_k(K)) + (k-1)p$ helyeket használjuk. Ekkor a hibás válasz valószínűsége (nem könnyű belátni!):

$$\left(1 - e^{-n/p}\right)^k.$$

Ez az $m = 14{,}4 \cdot n,\ p \approx 1{,}44 \cdot n,\ k = 10$ választással szintén kb. $1/1000$.

### 8.11.1. Alkalmazások

A Bloom-filter néhány érdekes alkalmazása:

- Webcache. Nagyon sok URL-t csak egyszer használunk, ennek tartalmát felesleges berakni a cache-be. Ezért első használatkor berakjuk az URL-t a Bloom-filter szótárába, és csak akkor rakjuk a tartalmát a cache-be, ha már másodszor használjuk.
- Bármilyen szótár fölé rakhatunk egy $14{,}4 \cdot n$ bitet tartalmazó Bloom-filtert, ezt használják pl. a Google Bigtable és a PostgreSQL alkalmazásokban. És csak akkor indítjuk el a KERESÉS műveletet, ha a konstans időben lefutó filter IGEN választ adott.
- Helyesírás-ellenőrzőknél is szokták használni, itt a helyes szavak szótárát tároljuk Bloom-filterben. Ekkor egy picit kellemetlen, hogy egy hibás szót kis valószínűséggel elfogad helyesnek.
- Elosztott adatbázisoknál és peer-to-peer hálózatoknál gyorsan tudni akarjuk, hogy mit hol érdemes keresni.
- Sok más helyen is használják, pl. a Bing keresőben, a Bitcoin és Ethereum kriptovalutáknál.

További példák találhatóak a Wikipédia oldalon.

# 9. Az LCA és RMQ feladatok

Nagyon sokszor tárolunk adatokat gyökeres fákban. Ezeken az egyik leggyakrabban használt (sokszor belső) alapművelet az LCA (Least Common Ancestor = Legközelebbi Közös Ős) feladat: ha $u$ és $v$ a fa csúcsai, akkor LCA$(u, v)$ visszaadja $u$ és $v$ legközelebbi (a gyökértől legtávolabbi) közös ősének a nevét.

Természetesen szinte bármilyen tárolási mód esetén meg tudunk válaszolni egy ilyen kérdést $O(d(u, v))$ lépésben, ahol $d(u, v)$ a csúcsok távolsága a fában. A célunk ezzel szemben az, hogy $O(n)$ előfeldolgozás után (ahol $n$ a fa csúcsainak a száma), $O(n)$ tárban úgy tároljuk el a fát, hogy utána minden LCA$(u, v)$ kérdést meg tudjunk válaszolni $O(1)$, azaz konstans lépésben!

Egy látszólag teljesen független feladat az RMQ (Range Minimum Query = Intervallum Minimum Lekérdezés). Tárolandó egy $n$ méretű $A$ tömb, a kérdés két számból áll: $1 \leq i \leq j \leq n$. Az RMQ$(i, j)$ kérdésre a válasz egy $k$ index, melyre $i \leq k \leq j$, és az ilyenek közül az, amelyre $A(k)$ minimális (ha több ilyen van, akkor bármelyik).

**14. Tétel.** *Ha az* RMQ *feladatra van* $(O(n), O(1))$ *megoldásunk (azaz olyan, mely* $O(n)$ *időben és tárban előfeldolgozva a tömböt, utána minden kérdésre konstans időben választ ad), akkor az* LCA *feladatra is kaphatunk* $(O(n), O(1))$ *megoldást.*

**15. Tétel.** *Ha az* LCA *feladatra van* $(O(n), O(1))$ *megoldásunk, akkor az* RMQ *feladatra is kaphatunk* $(O(n), O(1))$ *megoldást.*

Ez így még „fából vaskarika”, azaz teljesen haszontalannak látszik. Mégis van egy kitörési pont. Definiáljuk a $\pm 1$-RMQ feladatot, mint az RMQ feladat azon megszorítását, amikoris az $A$ tömb minden eleme pontosan eggyel tér el az előzőtől: $\forall\ 2 \leq i \leq n : |A(i) - A(i-1)| = 1$.

A fenti 14. tétel bizonyítása automatikusan adja majd a következő, erősebb tételt.

**16. Tétel.** *Ha a* $\pm 1$-RMQ *feladatra van* $(O(n), O(1))$ *megoldásunk, akkor az* LCA *feladatra is kaphatunk* $(O(n), O(1))$ *megoldást.*

Először ezt bizonyítjuk, majd megadjuk a kívánt megoldást a $\pm 1$-RMQ feladatra. Végül bizonyítjuk a 15. tételt, azaz az általános RMQ feladatot visszavezetjük a (bináris) LCA feladatra.

**Bizonyítás:** (16. tétel): Egy fa Euler-sétájának nevezzük azt a körsétát, amely a fa minden élén mindkét irányban pontosan egyszer halad át. A séta által érintett csúcsok neveit egy $E$ nevű $2n-1$ méretű tömbbe rendezzük, az első és utolsó elem legyen a gyökér. Ebben minden csúcs a fokszáma-szor szerepel (kivéve a gyökér, az eggyel többször). Ez a tömb mélységi kereséssel könnyen előállítható, ugyanúgy, mint az alábbi két tömb is: $L(i)$ jelölje a séta során $i$-ediknek érintett csúcs mélységét, $R(u)$ pedig az $u$ nevű csúcs első előfordulásának indexét az $E$ tömbben. Nézzünk egy LCA$(u, v)$ kérdést, feltehetjük, hogy $R(u) \leq R(v)$ (különben $u$-t és $v$-t kicseréljük). Könnyű meggondolni, hogy LCA$(u, v) = E(\text{RMQ}_L(R(u), R(v)))$, azaz a legközelebbi közös ős a séta mentén $u$ és $v$ között érintett csúcsok közül a legkisebb mélységű. Könnyű látni hogy az $L$ tömb teljesíti $\pm 1$-RMQ feladat inputjára tett megszorítást.

## 9.1. A $\pm 1$-RMQ feladat

Először nézzük meg, milyen naiv megoldás adható az általános RMQ feladatra. Könnyen kaphatunk egy $(O(n^2), O(1))$ megoldást, ha minden $i \leq j$ párra kiszámoljuk és eltároljuk a választ. Ezt $O(n^2)$ időben tényleg ki tudjuk számítani dinamikus programozással.

Ennél sokkal rafináltabb megoldás (még mindig az általános RMQ feladatra) az alábbi, $(O(n \log n), O(1))$ idejű:

Nem kell minden $i \leq j$ párra kiszámítani a megoldást, elég az olyan $i \leq j$ párokra, ahol $j-i+1$ kettőnek egy hatványa. Egy adott $i$-hez maximum $\log n$ ilyen $j$ van, tehát ez csak $n \cdot \log n$ tárat igényel. Meg kell gondolni, hogy ezeket az értékeket tényleg ki is tudjuk számítani $O(n \cdot \log n)$ időben, ez megint csak dinamikus programozással megy: ha már adott $k$-ra minden $i$-re kiszámítottuk az $(i, i+2^k-1)$ kérdésekre a választ, akkor ezekből $k+1$-re minden $i$-re megkaphatjuk a választ egy összehasonlítással: $A(i : i+2^{k+1}-1)$ minimuma a $\min\big(\min A(i : i+2^k-1),\ \min A(i+2^k : i+2^{k+1}-1)\big)$ lesz.

Hátra van még, hogy ezekből az eltárolt értékekből hogyan válaszolunk meg konstans időben egy kérdést.

Legyen $(i, j)$ egy kérdés, és $k = \lfloor \log(j+1-i) \rfloor$. Ezt azért könnyű kiszámolni, mert pont a $j+1-i$ legszignifikánsabb bitjének a helye. Nyilván: $\min A(i : j) = \min\big(\min A(i : i+2^k-1), \min A(j-2^k+1 : j)\big)$.

Most rátérünk a $\pm 1$-RMQ feladatra. Legyen $k = \lfloor (\log n)/2 \rfloor$. Osszuk fel az $A$ tömböt $B_1, B_2, \ldots, B_{\lceil n/k \rceil}$ blokkokra, minden blokk $k$ hosszú (az utolsót nagy számokkal kiegészítjük). Csinálunk egy $\lceil n/k \rceil$ hosszú $A'$ tömböt, ahol $A'(i)$ tartalmazza a $B_i$ blokk minimális értékét. Szükségünk lesz még egy $\lceil n/k \rceil$ hosszú $B'$ tömbre is: $B'(i)$ tartalmazza azt a *relatív* $j$ indexet a $B_i$ blokkon belül, ahol $A((i-1)k+j)$ felveszi az $A'(i)$ minimum-értéket.

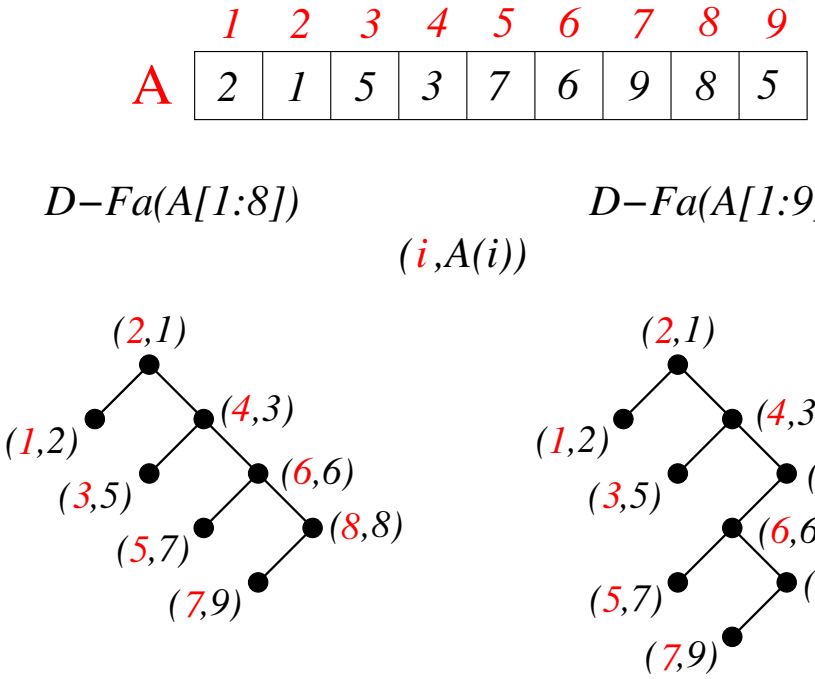
Egy adott $(i, j)$ kérdésre kiszámoljuk a következő értékeket: $i'$ az $i$ indexet tartalmazó blokk neve, $j'$ a $j$ indexet tartalmazó blokk neve, $i_{rel}$ az $i$ relatív indexe a $B_{i'}$ blokkon belül, végül $j_{rel}$ a $j$ relatív indexe a $B_{j'}$ blokkon belül. Megjegyzés: ezeket nem feltétlenül kell és érdemes a kérdéseknél kiszámítani, az előfeldolgozás során is számíthatjuk és tárolhatjuk. Három eset van: ha $i' = j'$, akkor a kérdést a $B_{i'}$ blokkon belül kell megválaszolni, lásd később. Ha $j' = i'+1$, akkor a $B_{i'}$ blokkon belüli $(i_{rel}, k)$ és a $B_{j'}$ blokkon belüli $(1, j_{rel})$ kérdésekre adott válaszokból az eredmény könnyen számítható (a két minimum-érték minimuma lesz az érték, a helye is könnyen számítható). A harmadik esetben kiszámítjuk $\mathrm{RMQ}_{A'}(i'+1, j'-1)$ értékét és helyét (az $A'$ ill. $B'$ tömb segítségével); ebből, valamint az $\mathrm{RMQ}_{B_{i'}}(i_{rel}, k)$, valamint az $\mathrm{RMQ}_{B_{j'}}(1, j_{rel})$ kérdésekre adott válaszokból a végső válasz ismét könnyen számolható.

Mivel az $A'$ tömb mérete csak $O(n/\log n)$, a fenti második megoldással az $A'$-re vonatkozó RMQ kérdések $O(1)$ időben megválaszolhatók $O(n)$ előfeldolgozás után. Hátra van még, hogy az egyes blokkokon belüli kérdésekre tudjunk válaszolni. Két blokkot nevezzünk hasonlónak, ha van olyan $c$ egész, hogy minden $i$-re a második blokk $i$-edik eleme pont $c$-vel nagyobb, mint az első blokk $i$-edik eleme. A kulcs észrevétel az, hogy két hasonló blokk esetén minden $i \leq j$-re az $\mathrm{RMQ}(i, j)$ kérdésre ugyanaz a válasz. Egy blokk normalizált blokkjának nevezzük a hozzá hasonló blokkok közül azt, amelyik első eleme 0. Egy $B_i$ blokk normalizált indexe kettes számrendszerben az a $k-1$ hosszú 0-1 sorozat, melynek $j$. betűje 0, ha $B_i(j+1) = B_i(j) - 1$, és 1 egyébként, ha tehát $B_i(j+1) = B_i(j)+1$. Az előfeldolgozás során minden blokkhoz kiszámítjuk a normalizált

indexét, és mind a $2^{k-1}$ lehetséges indexhez kiszámítjuk a hozzá tartozó normalizált blokk minden lehetséges $1 \leq i \leq j \leq k$ kérdéséhez a választ a naiv megoldás szerint. Mivel csak $2^{k-1} < \sqrt{n}$ normalizált blokk van, ez $O(\sqrt{n} \cdot k^2) = O(n)$ időben és tárban kiszámolható.

## 9.2. Általános RMQ feladat visszavezetése az LCA feladatra

Adott $A$ tömbhöz definiáljuk rekurzívan a Descartes-fáját (Cartesian tree), ami a következő bináris fa: a gyökér neve legyen $k$, ha az egész $A$ tömb (első) legkisebb eleme $A(k)$. A gyökér bal részfája legyen az $A(1 : k-1)$ résztömb Descartes-fája, jobb részfája pedig az $A(k+1 : n)$ résztömb Descartes-fája. Vegyük észre, hogy ha az $i$ nevű fa-csúcshoz kulcsként hozzárendeljük az $A(i)$ értékeket, akkor ez a fa egyszerre lesz a csúcsnevek (indexek) szerinti bináris keresőfa és a kulcsok szerint pedig kupacrendezett fa.




Először azt állítjuk, hogy ez a fa $O(n)$ időben megkonstruálható. Szépen sorban minden $i = 1 \ldots n$-re megkonstruáljuk az $A(1 : i)$ résztömb Descartes-fáját. Ha $i$-re már megvan, akkor $i+1$-re így járunk el: Legyenek a gyökértől szigorúan jobbra menő út csúcsai: $v_0, v_1, \ldots, v_t$. Nyilván $A(v_0) \leq A(v_1) \leq \ldots \leq A(v_t)$. Tegyük fel, hogy $k$ a legnagyobb olyan index, melyre $A(v_k) \leq A(i+1)$. Ekkor az új fa úgy néz ki, hogy $v_k$ jobb gyereke helyére berakjuk $i+1$-et, akinek nem lesz jobb gyereke, a bal gyereke pedig $v_{k+1}$ lesz (ha $k = t$, akkor bal gyereke sem lesz). A $k$ indexet bináris kereséssel is megkereshetnénk, de a már megszokott módon jobb lesz, ha alulról felfelé egyesével lépkedve keressük. Legyen a potenciál a gyökértől szigorúan jobbra menő út csúcsainak a száma. Ekkor $\ell$ kereső lépés esetén, a potenciál $-(\ell - 1) + 1 = -\ell + 2$ -vel változik, a berakás 1 lépés, így az amortizációs idő 3. Innen a teljes fa felépítése $O(n)$.

Azt is könnyű meggondolni, hogy az RMQ$(i, j)$ kérdésre adandó válasz a Descartes-fában az LCA$(i, j)$ kérdésre adott válasz. Gondoljunk a rekurzív definícióra. Kezdetben

$i$ és $j$ egy résztömbben van. Állítsuk meg a rekurzív konstrukciót akkor, amikor már nem lesznek egy résztömbben. Ekkor egy $A(i' : j')$ résztömb legkisebb elemének $k$ indexe lesz az aktuális csúcs, és az ettől balra/jobbra álló rész megy a bal/jobb gyerekbe; aholis $i' \leq i \leq k \leq j \leq j'$. Tehát $k$ lesz a helyes válasz az $\mathrm{LCA}(i, j)$ kérdésre, másrészt pedig $k = \mathrm{LCA}(i, j)$.

# 10. A van Emde Boas struktúra

A Prim algoritmushoz szeretnénk jobb kupacot készíteni abban az esetben, ha a kulcsok (az élek költségei) nem túl nagy természetes számok. Először is vegyük észre, hogy ha van egy szótárunk törléssel, mely még tudja a Min(S) műveletet is (ez visszaadja a szótár legkisebb elemét), akkor tudunk belőle kupacot is csinálni úgy, hogy a szótárelemek mellett „értékként" egy láncolt listában tároljuk azon csúcsok neveit, melyek kulcsa pont az adott szótárelem. Itt pl. a Kulcs-csökkentés így néz ki: az adott elemet kitöröljük a láncolt listából, ha az kiürült, hívunk egy szótárra vonatkozó Törlést. Ezután lecsökkentjük a kulcsát. Ha az új kulcs benne van a szótárban, akkor befűzzük annak listájába, egyébként előbb kell egy szótárbeli Beszúrás művelet. Ez általában nem túl hatékony, de most itt kivételesen az lesz, mert a szótár az $\{0, 1, \ldots, C-1\}$ univerzum része.

Egy általánosabb szótárat csinálunk, megtartjuk a Beszúrás$(x, S)$ és Törlés$(x, S)$ műveleteket, viszont a Keresés$(S)$ és a Min$(S)$ műveletek helyett egy közös általánosítást valósítunk meg, a Köv $(x, S)$ műveletet, ahol Köv $(x, S)$ visszaadja a szótár legkisebb olyan elemét, mely $\geq x$. (Ez nyilván általánosítja a Keresés és a Min műveleteket, hiszen ha $z$ az univerzum legkisebb eleme, akkor Köv $(z, S)$ =Min$(S)$.) Az ilyen szótárak önmagukban is fontosak, nemcsak kupacot lehet a segítségükkel csinálni, hanem sok más feladathoz is használják őket. Az univerzumunk tehát a $[0, C-1]$ intervallum egész számaiból áll. A továbbiakban csak a szótárral foglalkozunk, mivel már megértettük, hogy a szótárból hogyan kell kupacot készíteni. Ezért $n = |S|$ a szótár méretét fogja jelölni.

Az a célunk, hogy minden művelet $O(\log \log C)$ időben fusson. Legyen $k = \lceil \log \log C \rceil$, ekkor feltehetjük, hogy $C = 2^{2^k}$. Először egy rekurzív definíciót adunk meg, amely sajnos nem teljesíti az időkorlátot. Utána ezt kijavítjuk, majd rátérünk az implementálás néhány fontos kérdésére.

Egy $\ell$ típusú **vEB struktúra** $2^{2^\ell}$ szótárelem tárolására alkalmas. Három számból és $2^{2^{\ell-1}} + 1$ pointerből áll. A számok: meret, az aktuálisan tárolt szavak száma, min, a legkisebb tárolt szótárelem, és max, a legnagyobb tárolt szótárelem. A felso nevű pointer egy darab $\ell - 1$ típusú vEB struktúrára mutat, míg az also$(a)$ pointerek ($0 \leq a < 2^{2^{\ell-1}}$) egy-egy $\ell - 1$ típusú vEB struktúrára mutatnak. Ugyan nem fontos külön Keresés műveletet csinálnunk, mégis az mutatja legjobban, hogy hogyan fog működni az egész. Tegyük fel, hogy a $k$ típusú vEB struktúrában akarjuk keresni az $i < 2^{2^k}$ (tehát $2^k$ bites) számot. Először is felírjuk $i$-t $a\sqrt{C} + b$ alakban, ahol $a, b < 2^{2^{k-1}}$. Ez valójában egy egyszerű művelet: $a$ lesz $i$ első $2^{k-1}$ bitje, míg $b$ a második $2^{k-1}$ bitje. Ezután egyszerűen az also$(a)$ pointer által mutatott $k - 1$ típusú vEB struktúrában keressük rekurzívan a $b$ számot. A felso struktúra elsősorban a Köv művelethez kell: azon $a$ szavakat tartalmazza, melyekre also$(a)$ nem üres.

A fentiek alapján az első rekurzív megvalósításunk a következő. Legyen $T$ egy tárolandó vEB struktúrára mutató pointer. A struktúra típusát a második paraméterben adjuk meg. Egy $p$ pointer által mutatott struktúra max változóját $p$.max-szal jelöljük, és persze hasonlóan a min és meret változókat is. A furcsa $G := 2^{2^{\ell-1}}$; $i = a \cdot G + b$ sor csak azt jelöli, hogy a $2^\ell$ bites $i$ szám első felét $a$-ban, második felét $b$-ben tároljuk.

Köv $(T, \ell, i)$:
$G := 2^{2^{\ell-1}}$; $i = a \cdot G + b$

**if** also$(a)$.max $\geq b$ **then return** $(a \cdot G$+Köv (also$(a), \ell - 1, b))$
**else** $c :=$Köv (felso, $\ell - 1, a + 1)$
**return** $(c \cdot G$+also$(c)$.min)

Ha a Köv eljárást meghívjuk egy $k$ típusú struktúrára, $O(1)$ lépést teszünk, plusz egy darab rekurzív hívás egy $k-1$ típusú struktúrában. Ez nyilván a kívánt $O(k)$ lépést adja összesen.

Beszúrás $(T, \ell, i)$:
$G := 2^{2^{\ell-1}}$; $i = a \cdot G + b$
**if** also$(a)$.meret$= 0$ **then** Beszúrás (felso, $\ell - 1, a)$
Beszúrás (also$(a), \ell - 1, b)$
meret, max, min frissítése

Törlés $(T, \ell, i)$:
$G := 2^{2^{\ell-1}}$; $i = a \cdot G + b$
Törlés (also$(a), \ell - 1, b)$
**if** also$(a)$.meret $= 0$ **then** Törlés (felso, $\ell - 1, a)$
meret, max, min frissítése

A Beszúrásnál az utolsó sor végrehajtása nyilvánvaló, a Törlésnél a rész-struktúrák törlés utáni értékeiből minden könnyen számolható, például
$c :=$ (felso.min) ; min:$= c \cdot G$+also$(c)$.min.

Sajnos ezen két utóbbi műveletnél a konstans sok lépés mellett KÉT darab rekurzív hívás is lehet legrosszabb esetben, ez pedig csak $O(2^k)$ lépésszámot garantál.

A kijavításhoz először is észrevesszük, hogy amikor két rekurzív hívás van, akkor az egyik vagy egy üres struktúrába beszúrás, vagy egy 1-eleműből való törlés. Ezeket kellene rekurzív hívás helyett konstans időben megvalósítani. A következő, meglepően egyszerű trükk segít: a min változóban tárolt szót nem tároljuk a rész-struktúrákban! Ez persze egy picit elbonyolítja a fenti kódokat, de mivel garantáltan csak egy rekurzív hívás lesz mindig, ezért kiadja az $O(k)$ lépésszámot.

Köv $(T, \ell, i)$:
$G := 2^{2^{\ell-1}}$; $i = a \cdot G + b$
**if** $i \leq$min **then return** (min)
**if** also$(a)$.max $\geq b$ **then return** $(a \cdot G +$Köv $(\text{also}(a), \ell - 1, b))$
**else** $c :=$Köv (felso, $\ell - 1, a + 1$)
**return** $(c \cdot G +$also$(c)$.min$)$

Beszúrás $(T, \ell, i)$:
**if** meret$= 0$ **then** meret$:= 1$; max$:=$min$:= i$; **return**
**if** min$> i$ **then csere** (min, $i$)
$G := 2^{2^{\ell-1}}$; $i = a \cdot G + b$
**if** also$(a)$.meret$= 0$ **then** Beszúrás (felso, $\ell - 1, a$)
Beszúrás $(\text{also}(a), \ell - 1, b)$
meret, max frissítése

Törlés $(T, \ell, i)$:
**if** meret$= 1$ **then** meret$:= 0$; **return**
$G := 2^{2^{\ell-1}}$
**if** min$= i$ **then** $i :=$min$:=$felso.min$\cdot G +$also(felso.min).min
$i = a \cdot G + b$
Törlés $(\text{also}(a), \ell - 1, b)$
**if** also$(a)$.meret $= 0$ **then** Törlés (felso, $\ell - 1, a$)
meret, max frissítése

Látszólag itt is megmaradt a két rekurzív hívás, azonban az egyik nem eredményez további rekurzív hívásokat és konstans időben végrehajtódik.

**Megjegyzés:** a 0 típusú struktúra maximum két elemet tárol. Erre elég a max és min, így ezekhez nem tartoznak pointerek (rész-struktúrák). Persze a kódba is bele kellene írni ezt a kivételt...

## 10.1. Tárhely, megvalósítás, javítás

Mennyi tárat használunk? Egyáltalán összesen hány darab vEB struktúrát kell tárolnunk? És hány pointert? És ezeket hogyan? Ami elég világos: egy struktúra részstruktúráinak méretét érdemes magában a struktúrában a pointer mellett tárolni, (illetve a 0 méretűekre pointer nem is kell). Az inicializálás sem egyszerű, ha minden lehetséges rész-struktúrát az elején inicializálni akarnánk, az már magában nagyon sok időt venne el. Egy rész-struktúrát igazából elég lenne akkor inicializálni, ha mérete 0 fölé nőtt. Ekkor viszont az a baj, hogy az üres struktúrába való beszúrás nem konstans idejű. Tehát jobb a rész-struktúrák minimumát is a szülőben tárolni, és csak akkor inicializálni, ha a méret 2-re nő, de még ezzel is több baj van. Egyrészt ilyenkor mennyi a futási idő? Mit csinálunk törléskor, ha egy részstruktúra mérete 1-re csökken? Ha töröljük, akkor lehet, hogy ugyanazt sokszor kell újra inicializálni, ha meg nem, akkor a tárhelyünk lehet túl nagy az aktuálisan tárolt elemek $n$ számához képest.

Mostantól $k$-t (és így $C$-t) fixnek tekintjük. Egy $c < 2^t$ természetes számhoz rendeljük hozzá a $\underline{c}_t \in \{0,1\}^t$ szót, mely a $c$ bináris alakja az elején elegendő számú 0-val. Egy-egy tárolandó vEB struktúrához hozzárendelünk egy $(\underline{c}_t, \ell)$ nevet. $(\underline{c}_t, \ell)$ egy $\ell$-típusú vEB struktúra neve, ha

i) $0 \leq \ell \leq k$, és
ii) $0 \leq t$ és $t + 2^\ell \leq 2^k$.
iii) A $(\underline{c}_t, 0)$ nevű struktúráknak nincsenek részstruktúrái.
iv) Ha $\ell \geq 1$, akkor a $(\underline{c}_t, \ell)$ nevű struktúra felso pointere a $(\underline{c}_t, \ell - 1)$ nevű struktúrára mutat,
v) és ha $a < 2^{2^{\ell-1}}$, akkor az also$(a)$ pointere a $(\underline{c}_t \underline{a}_{2^{\ell-1}}, \ell - 1)$ nevű struktúrára mutat. (Itt nem szorzás van, hanem konkatenáció!)
vi) A főstruktúra neve $(\emptyset, k)$, ahol most $\emptyset$ az üres szó.

Egy $(\underline{c}_t, \ell)$ nevű vEB struktúra **indexe** legyen a következő kettes számrendszerbeli szám: $1\underline{c}_t 01 \ldots 1$, ahol a végén $\ell$ darab egyes van. Mivel $t + \ell \leq 2^k$, ezért maximum $2^{2^k+2} = 4C$ darab struktúrára van szükségünk. Ha ezeket egy tömbben tároljuk (az $i$ indexű struktúra meret, min, és max értékeit az $i$-edik sorban), akkor a struktúra pointereit már nem kell tárolni, nevük és címük a fentiek alapján számolható. Így a tömb sorainak száma $4C$, az összes tárigény $12C$.

Nem beszéltünk még az inicializálásról: sajnos legalább a fenti tömbben levő meret értékeket 0-ra kell állítani. Ez a gyakorlatban egy utasításnak látszik (calloc), és tényleg gyors is, de elméleti szempontból ront a vEB struktúrák szépségén.

Azonban a $12C$ tárhely nem mindig elfogadható. A $\log\log C$ futási idő még nagyon jó pl. $C = n^5$ esetén is, de $n$ adatot nem szeretünk $n^5$ helyen tárolni. Erre az ad megoldást (és az inicializálási problémát is kiküszöböli), ha a fenti tömb helyett dinamikus tökéletes hashelést használunk. Ennek univerzuma a címtartomány (a számok 0-tól $4C$-ig), és minden nem-üres struktúra címe szerepel, mint kulcs, mellette értékként a meret,max,min hármas. Kezdhetünk egy, csak a fő struktúrát tartalmazó egyelemű hashtáblával, és csak ha a keresett részstruktúra indexe nem szerepel a hash táblában, akkor szúrjuk oda be a dinamikus hashelés Beszúrás műveletével. Mivel a dinamikus hashelés műveletei átlagosan konstans várható idejűek, ez nem rontja el a vEB műveletek lépésszámát, a tárat viszont leviszi $O(n \cdot \log\log C)$-re (ennyi nem-üres struktúránk lesz $n$ Beszúrás után). Még praktikusabb, ha kakukk-hashelést használunk.

## 10.2. Újabb megvalósítások

Itt először definiáljuk a szófát, majd vázolunk három kapcsolódó, érdekes megoldást.

**Szófa (trie)** (Eredeti kiejtése trí, mint a reTRIEval szóban, de sokan a try szó mintájára ejtik, hogy megkülönböztessék az általános fáktól).

Ez a keresőfák egy speciális alfaja, amivel tulajdonképpen már mindenki találkozott. Legyen $\Sigma$ egy véges abc, szokásos módon $\Sigma^*$ jelöli a $\Sigma$ feletti véges szavak halmazát, most az univerzumunk ez a $\Sigma^*$ halmaz lesz, vagy ennek egy részhalmaza.

Gyökeres fában tároljuk az $S \subseteq \Sigma^*$ szótárat, minden csúcsnak legfeljebb $d = |\Sigma|$ gyereke van. Az élekre a $\Sigma$ elemeit képzeljük, tárolás szempontjából ez azt jelenti, hogy pl. ha $x \in \Sigma$ és a fa egy $u$ csúcsából megy lefelé $x$ címkéjű él egy $v$ csúcsba, akkor az $u$ csúcs $x$ nevű gyerek-pointere a $v$ csúcsra mutat; ha $u$-ból nem megy lefelé $x$ címkéjű él, akkor pedig az $u$ csúcs $x$ nevű gyerek-pointere $nil$.

A fa minden csúcsához hozzárendelünk egy szót. A gyökérhez ez az üres szó, a többi csúcsnál az odavezető út éleire írott betűkből alkotott szó, tehát egy $h$ mélységben levő csúcshoz rendelt szó mindig $h$ betűből áll. Minden csúcshoz tartozik még egy speciális pointer is, ami $nil$, ha a csúcshoz rendelt szó nincs a szótárban, egyébként a csúcshoz rendelt szóhoz rendelt rekordra mutat. Általában feltesszük, hogy a fa minimális, azaz ha $u$ egy csúcsa a fának, akkor $u$-nak van olyan leszármazottja, amelynek a speciális pointere nem $nil$.

Példa: ha $\Sigma$ az angol abc, R a gyökér, akkor a retrieval szó magyar jelentése az R.r.e.t.r.i.e.v.a.l.spec által mutatott helyen van.

Az ilyenek használata akkor igazán hatékony, ha az abc-hez van egy sztenderd kódolás a $\{0, 1, \ldots, d-1\}$ elemeivel. Az alábbiakban csak a $\Sigma = \{0, 1\}$ abc-re fogjuk alkalmazni.

**x-fast trie** (Willard, 1982). Itt $O(n \log C)$ tárat használunk, a Köv művelet $O(\log \log C)$ időben megy, viszont a Beszúrás és Törlés csak $O(\log C)$ amortizált időben. Veszünk egy $\log C$ magasságú bináris fát, a sztenderd kódolással (bal gyerek = 0, jobb gyerek = 1) ennek levelei megfelelnek az univerzumnak. Csak azokat a fa-csúcsokat tároljuk, amelyek alatt van szótárelem, tehát $|S| \leq n$ levelünk lesz az alsó szinten, ezeket duplán összeláncoljuk. Ha egy fa-csúcsnak nincsen bal (ill. jobb) gyereke, akkor ehelyett a pointer a részfájában levő legkisebb (ill. legnagyobb) levélre mutat. Azonban nem faként tároljuk az egészet, hanem minden szintet (az esetleges max/min pointerekkel együtt) egy-egy dinamikus tökéletes hashtáblában (vagy kakukk-táblában).

A Köv művelet: ha $x$ benne van a legalsó táblában, akkor kész, különben felező kereséssel (a szinteken) megkeressük a leghosszabb $y$ kezdőszeletét, ami a fában van. Ha ennek a fa-csúcsnak nincs jobb gyereke, akkor van egy pointere a bal gyerek max elemére, ennek jobb testvére (a duplán láncolt levél-listában) lesz a válasz. Ha pedig nincs bal gyereke, akkor az alatta levő min elem a válasz.

Beszúrásnál megkeressük a rákövetkezőt, és hasonlóan a megelőzőt is, a két levél közé beláncoljuk új levélként $x$-et, majd $y$-tól sorban lefelé a szükséges kezdőszeleteket beszúrjuk a megfelelő táblákba.

Törlésnél megkeressük $x$-et, és kifűzzük a megelőzője és rákövetkezője közül, de ezeket megjegyezzük. Utána $x$-ből felfelé töröljük azokat, akiknek $x$ volt az egyetlen leszármazottjuk, majd továbbmenve állítjuk a speciális pointereket, ha $x$-re mutattak.

**y-fast trie** (Willard, 1982). Itt már $O(n)$ tárat használunk, és a Beszúrás/Törlés idejét levisszük amortizáltan $O(\log \log C)$-re. A szótárat blokkokra osztjuk, minden blokk mérete $(\log C)/4$ és $2 \log C$ között lesz végig. A blokkokhoz egy-egy reprezentáns elemet választunk (ez nem feltétlenül szótárelem), ezeket tároljuk egy $x$-fast trie struktúrában. Mivel legfeljebb $4n/\log C$ reprezentáns van, ezért erre elég $O(n)$ tár.

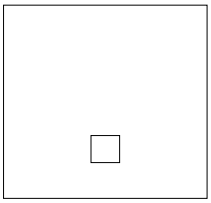
Az $x$-fast trie minden leveléhez tároljuk az általa reprezentált blokkban levő szótárelemeket egy $O(\log \log C)$ mélységű bináris keresőfában, ezeken belül mindent meg tudunk csinálni $O(\log \log C)$ lépésben. A Köv műveletnél megkeressük a nagy fában az $x$-et megelőző és az $x$-et követő reprezentánst, azután a két kis fában megkeressük a valódi választ.

A Beszúrásnál és Törlésnél csak akkor kell a kis fákon kívül bármit csinálnunk, ha a blokkméret kilépett az előírt korlátok közül. Ekkor kellhet két blokk összevonása (majd esetleg újraosztása), vagy felezése, és a reprezentánsok lecserélése az $x$-fast trie struktúrában, ami $O(\log C)$ tényleges idejű, de átlagosan csak legfeljebb minden $(\log C)/4$ ilyen művelet után kell végrehajtani, így ennek a résznek az amortizációs ideje műveletenként konstans.

**z-fast trie** (Belazzougui et al., 2009-2010). Ez az előző általánosítása változó bithosszúságú szavakra.

# 11. Geometriai adatstruktúrák

Egy GPS-hez szükséges program alapvető feladatait (a Dijkstra algoritmus kivételével, amit már jól körüljártunk) szeretnénk megvalósítani ebben a részben.



*A feladat*: Adott egy fix térkép és arra vagyunk kíváncsiak, hogy hogyan néz ki a térkép egy adott környezetünkben. A kérdés, hogy hogyan érdemes tárolni a térképet, hogy ezt a feladatot gyorsan meg tudjuk valósítani.

*Feltételezések (egyszerűsítések)*:

1. a térkép objektumai pontokból és szakaszokból állnak (körökkel nem érdemes foglalkozni, mert vagy csak akkor akarjuk kirajzolni, ha a középpontjuk is a téglalapba esik, vagy pedig egy térképen csak 2-3 féle méretű kör van, amelyek középpontjait a megfelelően megnövelt téglalapban kereshetjük),

2. a nevek az objektumokhoz tartoznak, amikor megjelenítünk egy objektumot, kiírjuk a hozzá tartozó nevet is,

3. a szakaszok nem metszik egymást (ez csak azt jelenti, hogy ha mégis metszenék, akkor a metszéspontot is tárolandó pontnak kell tekintenünk),

4. a kérdés-ablakról feltesszük, hogy tengelyirányú téglalap. Sajnos a valóságban ez nem tehető fel (ahogy fordulunk, a térképrészlet is fordul), de egyrészt csak ezt van esély megvalósítani, másrészt pedig a fordulásnál gyorsan kell tudnunk forgatni a kirajzolandó részletet, tehát mindenképpen érdemes az adott ferde téglalapot tartalmazó kicsit nagyobb, tengelyirányú téglalapot lekérdezni a térképből, és a gyors memóriában tárolni.

**Megjegyzés:** Rengeteg további alkalmazás van, pl. VLSI áramkörök tervezése, CAD rendszerek, ill. több dimenzióban VR alkalmazások (repülőgép-szimulátor, játékok), vagy végül, de egyáltalán nem utolsósorban, adatbázisok bizonyos tulajdonságú elemeinek kilistázása.

## 11.1. Egy dimenziós feladatok

Ezeket két okból tárgyaljuk: egyrészt szükségünk lesz rájuk a két dimenziós feladatok megoldásában, másrészt egyszerűek, ámde jól mutatják a vizsgált feladattípusoknál használható/használandó elveket.

**I.** *TÁROLNI*: $x_1, x_2, \ldots x_n \in \mathbb{R}$ pontokat.
*KÉRDÉS*: $PONTOK(x_\mathbf{b}, x_\mathbf{j})$ – felsorolni azokat a pontokat, amelyek ebbe az intervallumba esnek, vagyis a válasz: $\{i : x_\mathbf{b} \leq x_i \leq x_\mathbf{j}\}$.

Az output hosszát nem tudjuk előre. Jelölje ezentúl ezt a mennyiséget $k$. A futási időket ezentúl $n$ és $k$ függvényében vizsgáljuk. Az ilyen algoritmusokat *output érzékeny* algoritmusoknak nevezzük. (Ilyennel már találkoztunk, pl. a legrövidebb út kiíratása is ilyen volt, az $O(k)$ időben futott a szülő pointerek felhasználásával.)

**Megoldás**: rendezett tömbben tároljuk a pontokat; $x_\mathbf{b}$-t megkeressük felező kereséssel, majd innentől listázunk, míg $x_\mathbf{j}$ felé nem érünk.

*IDŐ*: $O(\log n + k)$
*TÁR*: $O(n)$
*FELÉPÍTÉS*: $O(n \cdot \log n)$

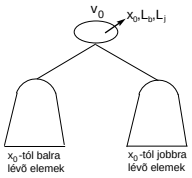
**Megjegyzés:** A cél mindig ez a három függvény, de néha nem tudjuk pontosan ezt elérni.

**II.** *TÁROLNI*: $I_1, I_2, \ldots I_n \subseteq \mathbb{R}$ intervallumokat, ahol $I_i = [x^i_\mathbf{b}, x^i_\mathbf{j}]$.
*KÉRDÉS*: INTERV($x$) – felsorolni $i$-ket, amelyekre $x \in I_i$.

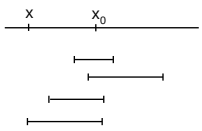
**Megoldás**: *Intervallum fa.* Az alábbi lépésszámokat/tárhelyet célozzuk itt is meg:

*IDŐ*: $O(\log n + k)$
*TÁR*: $O(n)$
*FELÉPÍTÉS*: $O(n \cdot \log n)$

A $2n$ darab $x^i_\mathbf{j}, x^i_\mathbf{b}$ közül legyen $x_0$ a középső (az $n$-edik). Rekurzívan fát építünk:




- a $v_0$ gyökérben raktározzuk el az összes $x_0$-t tartalmazó intervallumot (és persze $x_0$ értékét is). Ezen intervallumok halmaza $\mathcal{I}(v_0) = \{i \mid x^i_\mathbf{b} \leq x_0 \leq x^i_\mathbf{j}\}$.
- a bal részfában tároljuk azokat az intervallumokat, amelyek szigorúan balra vannak az $x_0$-tól, vagyis: $\{i \mid x^i_\mathbf{j} < x_0\}$. Itt legfeljebb $\frac{n}{2}$ intervallum lesz.
- a jobb részfában a szigorúan jobbra lévő intervallumokat tároljuk.
- a két részfát rekurzívan ugyanígy építjük fel.



Tegyük fel, hogy az INTERV($x$) kérdésnél pl. $x \leq x_0$. Az $x$-et tartalmazó intervallumok kétfajták lehetnek:

- azok, amelyek nem érik el $x_0$-t – ezek a bal részfában vannak,
- azok, amelyek elmennek $x_0$-ig – ezeket a gyökérben tároltuk.

**Megoldás**: A $v_0$ csúcsban $x_0$ értéke mellett $\mathcal{I}(v_0)$-t is tároljuk, mégpedig kétszer, egy $L_{\mathbf{b}}(v_0)$ és egy $L_{\mathbf{j}}(v_0)$ tömbben.

$L_{\mathbf{b}}(v_0)$: $x_0$-t tartalmazó intervallumok, a bal végpont szerinti növekvő sorrendben.

$L_{\mathbf{j}}(v_0)$: $x_0$-t tartalmazó intervallumok, a jobb végpont szerinti csökkenő sorrendben.

Ha $x < x_0$, akkor $L_{\mathbf{b}}$ elemeit felsoroljuk az első elemétől, amíg a bal végpont $\leq x$. Ha pedig $x \geq x_0$, akkor $L_{\mathbf{j}}$ elemeit felsoroljuk az első elemétől, amíg a jobb végpont $\geq x$.

Ha $x < x_0$, akkor rekurzívan folytatjuk a bal fában, ha $x > x_0$, akkor a jobb részfában, ha pedig $x = x_0$, akkor megállhatunk. Lépésszám $v_0$-ban: $1 + k_{v_0}$, ha $k_{v_0}$ darab kiírandó intervallum található itt (azaz $\mathcal{I}(v_0)$-ban). A fa mélysége – a felezés miatt – legfeljebb $\log n$ lesz. Így a válaszolás összlépésszáma $O(\log n + k)$.

A felépítési eljárás lényege: az elején rendezzük az intervallumokat két tömbbe: a bal végpont szerint növekvően rendezve, és jobb végpont szerint csökkenően rendezve. Egy adott $v_i$ csúcsnál először akár a két tömb virtuális összefésülésével, akár a tanult lineáris mediáns számítási módszerrel megkapjuk az $x_i$ középső számot. Utána a két tömböt lineáris időben szétszedjük három részre (az első részbe kerülnek a bal gyerekbe továbbmenő intervallumok, a középsőbe az adott $x_i$-t tartalmazók, a harmadikba a jobb gyerekbe menők). A két kapott középső tömb pontosan $L_{\mathbf{b}}$ és $L_{\mathbf{j}}$ lesz. Egy szinten belül az $\mathcal{I}(v)$ halmazok diszjunktak, így a listák összhossza legfeljebb $n$. Tehát minden újabb szintet $O(n)$ időben el tudunk készíteni. Mivel minden intervallumot csak egy fa-csúcsnál tárolunk, így a tárhely valóban $O(n)$.

## 11.2. Két dimenziós feladatok

**III.** *TÁROLNI*: $p_1, p_2, \ldots p_n \in \mathbb{R}^2$ pontokat, $p_i = (x_i, y_i)$.

*KÉRDÉS*: Egy adott $[x_{\mathbf{b}}, x_{\mathbf{j}}] \times [y_{\mathbf{a}}, y_{\mathbf{f}}]$ téglalapba eső pontok felsorolása, azaz $\{i \mid x_{\mathbf{b}} \leq x_i \leq x_{\mathbf{j}}$ és $y_{\mathbf{a}} \leq y_i \leq y_{\mathbf{f}}\}$.

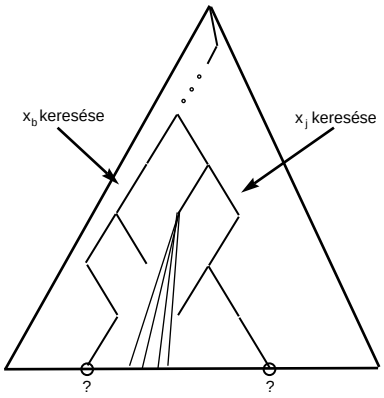
**Megoldás**: *Tartomány-fa* (range tree).

Először is az $x$ koordináták szerint kiegyensúlyozott külső tárolású bináris keresőfát építünk (a belső csúcsokban útjelzők, a levelekben balról jobbra növekvően a különböző $x_i$ értékek vannak).

Ha $v$ a fa egy csúcsa, akkor ASSOC($v$) := $\{i \mid x_i$ egy $v$ alatti levélben van$\}$.

Minden $v$ csúcshoz az ASSOC($v$) halmazt egy $L(v)$ tömbben tároljuk, az $y$ koordináták szerint rendezve. (Egy $u$ levélben $L(u)$ az azonos $x$ koordinátájú pontokat tartalmazza $y$ szerint rendezve.)

*Válaszoló algoritmus*: Először $x_{\mathbf{b}}$ és $x_{\mathbf{j}}$ keresése.




Legyen $v_{split}$ az a csúcs, ahol a két keresés elágazik. Egy csúcs fontos, ha a $v_{split}$-től az $x_{\mathbf{b}}$ keresésekor elért levélbe vezető úton lévő olyan csúcs jobb gyereke, ahol balra léptünk, illetve, ha a $v_{split}$-től az $x_{\mathbf{j}}$ keresésekor elért levélbe vezető úton lévő olyan csúcs bal gyereke, ahol jobbra léptünk. Ezeken kívül a két megtalált levél is fontos, ha a rájuk írt $c$ címkére: $x_{\mathbf{b}} \leq c \leq x_{\mathbf{j}}$.

Könnyű látni, hogy a kilistázandó pontok mind ASSOC($v$)-ben vannak valamely fontos $v$ csúcsra. (Valamint a definíció alapján az a tény is nyilvánvaló, hogy ha $u$ és $v$ fontos csúcsok, akkor ASSOC($u$) és ASSOC($v$) diszjunktak.) Tehát minden fontos $v$ csúcsban ki kell válogatni ASSOC($v$) elemeiből a kérdésnek megfelelőeket.

A fa mélysége $\log n$, így legfeljebb $2 \cdot \log n$ fontos csúcs van.

Minden fontos $v$-re $y_{\mathbf{a}}$-t megkeressük felező kereséssel $L(v)$-ben és kilistázzuk a pontokat $y_{\mathbf{f}}$-ig.

Egy darab $v$-nél legfeljebb $\log n + 1 + k_v$ lépés kell, ahol $k_v$ a $v$-nél kilistázott elemek száma.

*IDŐ*: $O(\log^2 n + k)$

*TÁR*: $O(n \cdot \log n)$ (egy pont minden szinten pontosan egyszer szerepel).

*FELÉPÍTÉS*: $O(n \cdot \log n)$

*Vázlat: rendezzük a pontokat $x$ és $y$ koordináta szerint is. Ha $r$ a gyökér neve, nevezzük ezeket $AX_r$ ill. $AY_r$-nek, és legyen $a_r = n$. Általában, ha a fa egy $v$ csúcsánál járunk, „felülről" megkapjuk $AX_v$-t, ami* ASSOC($v$) *az $x$ koordináta szerint rendezve, $AY_v$-t, ami* ASSOC($v$) *az $y$ koordináta szerint rendezve, és az $a_v = |$*ASSOC($v$)*$|$ méretet. Ekkor $v$ útjelzője az $AX_v(\lfloor a_v/2 \rfloor)$ pont $x$ koordinátája lesz. Végigmenve az $AX_v$ és $AY_v$ tömbökön, minden pont $x$ koordinátáját ezzel hasonlítva a tömböket könnyen szétszedhetjük a rendezett $AX_{bal(v)}$ és $AX_{jobb(v)}$, ill. $AY_{bal(v)}$ és $AY_{jobb(v)}$ tömbökre.*

**Megjegyzés:** ez az algoritmus működik több dimenzióban is, ha úgy módosítjuk, hogy $d$ dimenzióban az első koordináta szerinti keresőfában az ASSOC($v$) halmazt rekurzívan egy $d-1$ dimenziós tartomány-fában tároljuk (a pontok első koordinátáját elfelejtve, mint ahogy eddig is tettük, hiszen a fenti módszerben az I. feladatnál tárgyalt 1-dimenziós tartományfát használtuk). Ekkor a lekérdezési idő $O(\log^d n + k)$ lesz, a szükséges tárhely pedig $O(n \cdot \log^{d-1} n)$.

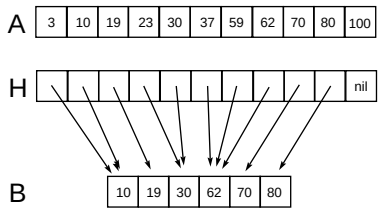
*Kérdés*: az $O(\log^2 n + k)$ időt lehet-e csökkenteni?

*A cél*: $O(\log n + k)$ keresési idő, ehhez az kell, hogy a $v$-nél csak $O(1 + k_v)$ időt töltsünk.

### 11.2.1. Javítás Kaszkád tárolással

Vegyük észre, hogy felülről lefele haladva mindig ugyanazt az $y_{\mathbf{a}}$ értéket akarjuk keresni egyre rövidebb rendezett tömbökben, amelyek résztömbjei a szülő megfelelő tömbjének.

Egy ehhez kapcsolódó feladat: $A$ és $B$ rendezett tömböket szeretnénk tárolni, ahol $A \supseteq B$.




$H(i) := min\{l \mid B(l) \geq A(i)\}$, illetve $nil$, ha nem létezik ilyen $l$.

Használata: ha például $y_{\mathbf{a}} = 20$, akkor megkeressük az $A$-ban az első ennél nagyobb-egyenlő elemet (ez $A(4) = 23$), és $H(4)$ megmutatja, hogy a $B(3) = 30$ lesz a $B$-ben az első megfelelő elem.

$\text{ASSOC}(v) = \text{ASSOC}(b(v)) \cup \text{ASSOC}(j(v))$.

Kaszkád tárolásnál minden $v$ csúcsban az $L(v)$ mellett egy $H_{\mathbf{b}}(v)$ és $H_{\mathbf{j}}(v)$ pointer-tömböt is tárolunk.

$y_{\mathbf{a}}$-t csak a gyökérnél keressük meg felező kereséssel, utána $H_{\mathbf{b}}$ vagy $H_{\mathbf{j}}$ mutatja meg a "helyét" (az első $y_{\mathbf{a}}$-nál nem kisebb elemet).

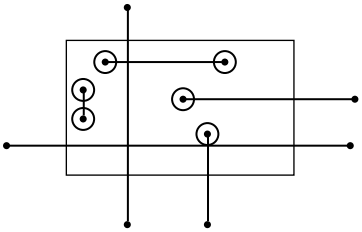
*IDŐ*: $O(\log n + k)$
*TÁR*: $O(n \cdot \log n)$
*FELÉPÍTÉS*: $O(n \cdot \log n)$
*Vázlat: Ugyanúgy, mint az előbb, de amikor kettészedünk egy listát, a $H_{\mathbf{b}}$ és $H_{\mathbf{j}}$ tömböket is feltöltjük. Minden megy a tárhelyben lineáris időben.*

**Megjegyzés:** A $d$ dimenziós feladatra alkalmazva az idő $O(\log^{d-1} n + k)$-ra csökken.

## 11.3. Egy téglalapot metsző szakaszok lekérdezése

Visszatérve az alapfeladatunkra: a kérdezett téglalapba (ablak) eső pontokat le tudjuk kérdezni, ha tartomány-fában tároltuk őket. Hasonló a helyzet az összes olyan szakasszal, amelynek legalább egyik végpontja az ablakba esik, itt csak két dologra kell vigyázni:

- a szakaszok végpontjaival együtt tároljuk el a szakasz sorszámát is (persze egy pont sok szakasz végpontja is lehet, így igazából egy-egy ponthoz tartozik egy vödör (láncolt lista), ami tartalmazza az összes olyan szakasz indexét, amelyeknek ez a végpontja),
- ha egy szakasz mindkét végpontja az ablakban van, akkor csak egyszer írjuk ki.



De persze lehetnek olyan, a téglalapot metsző szakaszok is, melyeknek mindkét végpontjuk kívül van. Először azt a speciális esetet oldjuk meg, amikor minden tárolt szakaszunk **vízszintes vagy függőleges**, és csak a következő fejezetben kezeljük a ferde szakaszok esetét (pl. VLSI tervezésnél ilyenek nincsenek is). Tehát most azokat a vízszintes és függőleges szakaszokat szeretnénk kiíratni, melyek teljesen átmetszik az ablakot. Az ilyenek vagy vízszintesek és metszik a jobb oldalt, vagy függőlegesek és metszik a felső oldalt. Ez a két feladat nyilván „szimmetrikus", tehát elég az első esetet kezelni. Azonban újra figyelni kell arra, hogy most kiválogatjuk az összes vízszintes szakaszt, mely metszi a jobb oldalt, tehát azokat is, amelyek bal végpontja benne van az ablakban – ezeket azonban már egyszer kiírtuk, nem szabad újra.

*A kétszeri kiírás megakadályozása*: pl. egyszerű, de nem optimalizált megoldás, ha egyrészt felveszünk egy $n$ hosszú tömböt ($n$ a szakaszok száma) és egy sort. Ha bármely fázisban megtaláljuk az $i$. szakaszt, akkor megnézzük, hogy a tömb $i$. eleme 1-e, ha igen megyünk tovább, ha nem, akkor egyre állítjuk és $i$-t berakjuk a sorba. A végén a sorból ki tudjuk íratni az összes megtalált szakaszt, és közben a tömböt újra le tudjuk nullázni.

**IV.** *TÁROLNI*: Vízszintes szakaszok a síkon.
*KÉRDÉS*: *METSZŐ* $([x] \times [y_{\mathbf{a}}, y_{\mathbf{f}}])$.
*VÁLASZ*: felsoroljuk az összes olyan vízszintes szakaszt, mely metszi a kérdezett függőleges szakaszt.

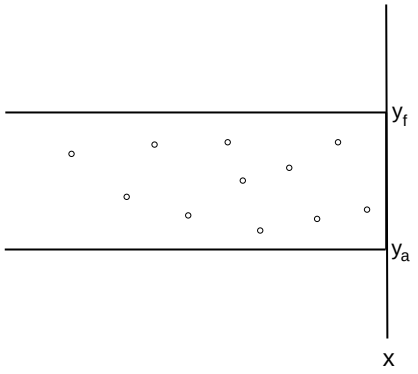
**IV. A)** Először egy intervallum-fát készítünk az $x$ tengelyre eső vetületek szerint. DE az eredeti megoldással ellentétben az $\mathcal{I}(v)$ halmazokat nem két listában tároljuk, hanem helyette két kupacos keresőfában (lásd alább). Az elsőben az intervallumokat a bal, a másodikban a jobb végpontjuk szerint. A keresés majdnem ugyanúgy megy, mint a hagyományos intervallum-fában, csak amikor az adott $v$ csúcs $\mathcal{I}(v)$ halmazából ki akarjuk válogatni a minket érdeklőket, akkor pl. $x < x_0$ esetén az első kupacos keresőfában keressük meg azokat, akiknek a bal végpontja a $(-\infty, x] \times [y_{\mathbf{a}}, y_{\mathbf{f}}]$ végtelen téglalapba esik. (Hasonlóan, $x > x_0$ esetén, a második kupacos keresőfában keressük meg azokat, akiknek a jobb végpontja a $[x, \infty) \times [y_{\mathbf{a}}, y_{\mathbf{f}}]$ végtelen téglalapba esik, ehhez a második kupacos keresőfát MAX-kupaccal kell definiálni.)

Tehát a következő feladatot kell még megoldanunk, melyre a tartomány-fa is megoldás lenne, de most egy még hatékonyabb megoldást adunk (kisebb tárhellyel). Amit meg tudunk valósítani a IV. feladatra (mivel $\mathcal{I}(v)$-ből kiíratás így nem lehet $O(k_v)$ idejű):

*IDŐ*: $O(\log^2 n + k)$
*TÁR*: $O(n)$
*FELÉPÍTÉS*: $O(n \cdot \log n)$

### 11.3.1. Kupacos keresőfák

**IV. B)** *TÁROLNI*: $p_1, p_2, \ldots p_n \in \mathbb{R}^2$
*KÉRDÉS*: PONTOK($(-\infty, x] \times [y_{\mathbf{a}}, y_{\mathbf{f}}]$).




**Megoldás**: *Kupacos keresőfa* – a legszebb öszvér adatstruktúra (mely a már tárgyalt Descartes-fa általánosításának tekinthető).

Először egy kis előkészületre van szükségünk, egy új műveletet vizsgálunk meg kupacokban.

Adott egy bináris kupacrendezett fa, és egy $K$ kulcs, a feladat pedig az, hogy soroljuk fel (tetszőleges sorrendben) az összes olyan $v$ csúcsát a kupacnak, amelyre $K(v) \leq K$.

**Megoldás**: Kiír $(v_0, K)$ hívása, ahol $v_0$ a kupac gyökere.
Kiír $(v, K)$
**if** $K(v) \leq K$ **then** $PRINT\ v$; Kiír $(b(v), K)$; Kiír $(j(v), K)$

**19. Állítás.** *Ez $O(1+k)$ időben fut.*

**Bizonyítás:** Ha meghívtuk Kiír $(u, K)$-t, akkor vagy $u$ gyökér volt, vagy $u$ eleme az outputnak, vagy $p(u)$ eleme az outputnak, tehát $k$ hosszú output esetén legfeljebb $2k+1$ hívás lesz (indukcióval könnyen láthatóan, egy $k$ csúcsú bináris fának $k+1$ fiktív levele van).

**Megjegyzés:** Ez persze csak akkor igaz, ha nem rendezve kellenek az elemek, különben mindenhol Mintörlés kellene.

A *Kupacos keresőfa* egy kiegyensúlyozott bináris fa, amelynek minden $v$ csúcsában egy $q(v)$ pont és egy $y(v)$ „útjelzőtábla" van.

Ez az öszvér adatstruktúra egyszerre lesz az $x$ koordináta szerint kupac, és az $y$ koordináta szerint bináris keresőfa; pontosabban csak annak hasonmása, azt a fontos keresőfa tulajdonságot teljesíti, hogy egy adott $v$ csúcsnál a bal részfában csupa olyan pontot tárolunk, melyek $y$ koordinátája $\leq y(v)$, a jobb részfában pedig csupa olyan pontot tárolunk, melyek $y$ koordinátája $> y(v)$, (de magában a $v$ csúcsban tárolt $q(v)$ pontnak semmi köze nem lesz $y(v)$-hez). Ezektől a $q(v)$ pontoktól eltekintve ez nagyon hasonlít a már tárgyalt

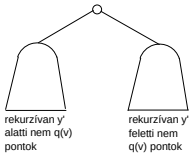
Descartes-fára, ahol az $A(i)$ elem $y$ koordinátájának $i$-t tekintettük, $x$ koordinátájának pedig az $A(i)$ értéket.

Felveszünk egy $v_0$ gyökeret, $q(v_0) :=$ a legbaloldalibb pont (ha több ilyen van, akkor közülük bármelyik).

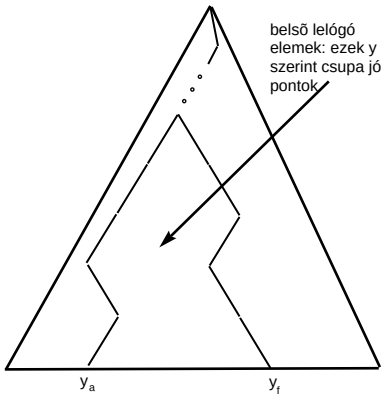
$y(v_0) := \min y'$, amelyre az $S = \{p_1, p_2, \ldots p_n\} \setminus \{q(v_0)\}$ halmaznak legfeljebb a fele van az $y = y'$ egyenes felett. Legyen $S_{\mathbf{a}} = \{i \in S \mid y_i \leq y(v_0)\}$ és $S_{\mathbf{f}} = \{i \in S \mid y_i > y(v_0)\}$. Ezeket rekurzívan ugyanígy tároljuk a gyökér bal, ill. jobb részfájában.

**Megjegyzés:** ha egy adott pillanatban minden pont $y$ koordinátája azonos, akkor az adott csúcsot megjelöljük (a keresőfának egy levele lesz), és a pontokat már egyszerűen $x$ szerinti kupacban tároljuk a részfájában. Ezzel elérjük, hogy a keresőfa mélysége akkor is maximum $\log n$, ha vannak azonos $y$ értékek, és a keresőfa levelein is maximum $\log n$ mélységű kupacok „lógnak".

A fő fa kiegyensúlyozott bináris fa, így mélysége $\leq \lceil \log n \rceil$. Az ilyen fákat (ez az egész fejezetre érvényes!) tárolhatjuk tömbben, úgy, mint a kupacokat (legfeljebb $2n$ hosszú tömb kell). A keresőfa levelei alatti „kupacokat" külön-külön tömbben érdemes ekkor tárolni.




A Keresés így megy: először $y_{\mathbf{a}}$-t és $y_{\mathbf{f}}$-et keressük a fában (amíg megjelölt csúcshoz nem érünk), a fontos csúcsokat ugyanúgy definiáljuk, mint a tartomány-fánál.




A keresési utak során bejárt csúcsokban tárolt $q(v)$ pontokat egyesével leellenőrizzük, hogy benne vannak-e a kérdezett téglalapban. Ezeken kívül minden más kiírandó pont valamely fontos csúcs alatt lesz. Ráadásul egy $v$ fontos csúcs alatti minden pont $y$ koordinátája az $[y_{\mathbf{a}}, y_{\mathbf{f}}]$ intervallumba esik, így a fontos csúcsok részfáit már kupacként

kezelhetjük, és az előbb látott módon a Kiír $(v, x)$ használatával ki tudjuk írni a jó pontokat $O(1+k_v)$ időben, ahol most $k_v$ az output összes, a $v$ részfájába eső pontjainak számát jelöli.

*IDŐ*: $O(\log n + k)$

Ez ugyanolyan gyors, mint a tartomány-fa kaszkádolással, de egyszerűbb a megvalósítás és kisebb a tárigény:

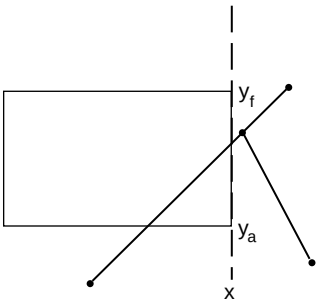
*TÁR*: $O(n)$

*FELÉPÍTÉS*: $O(n \cdot \log n)$

**Megjegyzés:** Ezzel azt az egydimenziós feladatot is meg lehet oldani, ahol intervallumokat kell tárolni, a kérdés $INTVTART(x_{\mathbf{b}}, x_{\mathbf{j}})$, és itt azon $I_i$ intervallumokat kell kiíratni, amelyekre $I_i \subseteq [x_{\mathbf{b}}, x_{\mathbf{j}}]$.

## 11.4. Ferde szakaszok lekérdezése

Ami hátravan, az az ablakot teljesen átmetsző ferde szakaszok lekérdezése. Ezt négy részben valósítjuk meg, külön-külön lekérdezzük a téglalap négy oldalát metsző ferde szakaszokat. Szimmetria okokból nyilván itt is elég lekérdezni egy adott függőleges szakaszt metsző ferde szakaszokat.




**Megjegyzés:** Most először használjuk ki, hogy a szakaszok nem metszhetik egymást.

**V.)** *TÁROLNI*: $S_1, S_2, \ldots S_n \subseteq \mathbb{R}^2$ egymást belső pontban nem metsző szakaszok.

*KÉRDÉS*: *METSZŐ* $([x] \times [y_{\mathbf{a}}, y_{\mathbf{f}}])$.

*VÁLASZ*: felsoroljuk az összes olyan szakaszt, mely metszi a kérdezett függőleges szakaszt.

**Megoldás**: A vetületeket most is mint $\{I_1, \ldots, I_n\}$ intervallumokat tároljuk. Azonban erre a célra az intervallum-fa itt nem felel meg nekünk, ezért egy újabb, első ránézésre sokkal rosszabb megoldást kell adnunk erre az egy dimenziós feladatra.

### 11.4.1. Szakasz-fa

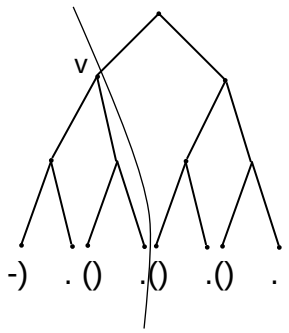
Az ún. Locus ($\approx$ mértani hely) megközelítést alkalmazzuk. Vegyük a lehetséges kérdések terét (itt $\mathbb{R}$, de egy ablaknál pl. $\mathbb{R}^4$ lenne), és particionáljuk ekvivalencia osztályokra (cellákra) úgy, hogy két kérdés ekvivalens, ha ugyanaz a halmaz a válasz rájuk. Ha ezt a partíciót hatékonyan tudjuk tárolni úgy, hogy egy adott kérdésre meg is tudjuk

találni gyorsan az ő ekvivalencia-osztályát, akkor az osztályokhoz felírva a válaszokat megoldottuk a feladatot. Ehhez persze eleve kell, hogy az osztályok száma ne legyen túl nagy, pl. ablak-kérdésekre ez nincs így.

Felbontjuk a számegyenest a végpontok által meghatározott egypontú és nyílt szakaszokra. Egy-egy ilyen szakaszba azok az $x$-ek esnek, amelyekre biztosan ugyanaz a válasz (az őt tartalmazó intervallumok halmaza).

Összesen legfeljebb $4n + 1$ darab ilyen szakasz lesz: legfeljebb $2n$ darab 1 pontú és legfeljebb $2n+1$ darab nyílt szakasz (két tárolt intervallum végpontjai egybe is eshetnek).

Egy kiegyensúlyozott bináris keresőfát építünk, melynek leveleire ráírjuk az egypontú és nyílt szakaszokat úgy, hogy balról jobbra rendezve legyenek. A fa egy $v$ csúcsára definiáljuk egyrészt az $\mathrm{Int}(v)$ intervallumot, mely a $v$ alatti levelekre írott halmazok uniója; másrészt az $\mathrm{ASSOC}(v) := \{i \mid I_i \supseteq \mathrm{Int}(v)$, de $I_i \not\supseteq \mathrm{Int}(p(v))\}$ halmazt. A $v$ csúcsnál tároljuk $\mathrm{ASSOC}(v)$-t mégpedig egy tömbben (itt még van némi szabadsági fokunk, hogy milyen sorrendben, ezt a következő alfejezetben ki is fogjuk használni).




**20. Állítás.** *Egy adott $I_i$ intervallumot a fa egy szintjén maximum kétszer tároljuk, így az összes tárigény $O(n \cdot \log n)$.*

**Bizonyítás:** Ha $I_i$ egy szinten háromszor lenne tárolva, akkor legyenek balról jobbra $a, b, c$ a csúcsok, amiben tároltuk. Mivel $\mathrm{Int}(a) \subseteq I_i$ és $\mathrm{Int}(c) \subseteq I_i$, ezért a keresőfa tulajdonságai miatt nyilván $b$ szülőjére $\mathrm{Int}(p(b)) \subseteq I_i$, tehát $I_i$ nem lehet benne $\mathrm{ASSOC}(b)$-ben.

*Keresés*: a fában itt is $x$-et keressük, és a keresési út során érintett minden $v$ csúcsra a teljes $\mathrm{ASSOC}(v)$-t kilistázzuk. Tehát a keresés ideje $O(\log n + k)$.

A szakasz-fa előnye az intervallum-fához képest, hogy a keresési út csúcsaiban tárolt $\mathrm{ASSOC}(v)$ halmazok uniója pontosan az, ami a kívánt output, ezekből itt nem kell leválogatni. Pont ezért, ha az $\mathrm{ASSOC}(v)$ halmazokat alkalmasan rendezett tömbökben tároljuk, akkor még egy másik szempont szerint tudunk belőlük válogatni.

Egy másik lehetséges alkalmazás az, amikor csak az input pontot tartalmazó intervallumok számára vagyunk kíváncsiak; ekkor minden csúcsban csak az $|\mathrm{ASSOC}(v)|$ számosságokat tároljuk (ekkor a tárigény csak $O(n)$), és az egyetlen kimenet $O(\log n)$ időben előállítható.

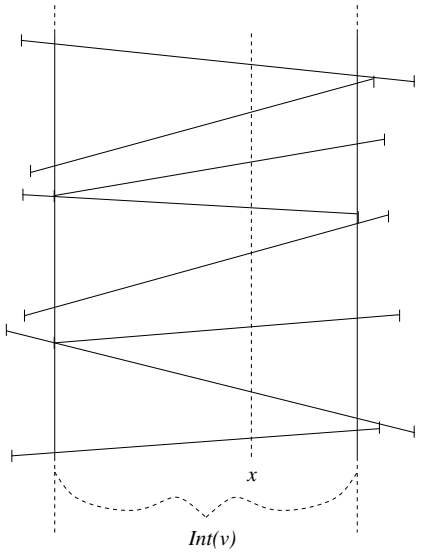
Meggondolható, hogy ez is felépíthető $O(n \cdot \log n)$ időben, a végpontok rendezése után a fát most *alulról felfelé építjük*, és közben minden $v$ csúcsához meghatározzuk $\mathrm{Int}(v)$-t (bal végpontja, bal zártság, jobb végpontja, jobb zártság alakban).

Ezután végigmegyünk az input intervallumokon, és mindegyiket $O(\log n)$ időben berakjuk a megfelelő ASSOC($v$) halmazokba. Ha az aktuális $I_i = [x, x']$ és $r$ a felépített fa gyökere, akkor meghívjuk az alábbi rekurzív eljárást Rek-Besz$(i, x, x', r)$ alakban.

Rek-Besz $(i, x, x', v)$:
**if** Int($v$) $\subseteq [x, x']$ **then** ASSOC($v$) := ASSOC($v$) $\cup \{i\}$
**else**
**if** Int(bal($v$)) $\cap [x, x'] \neq \emptyset$ **then** Rek-Besz($i, x, x'$,bal($v$))
**if** Int(jobb($v$)) $\cap [x, x'] \neq \emptyset$ **then** Rek-Besz($i, x, x'$,jobb($v$))

### 11.4.2. Egy függőleges szakaszt metsző ferde szakaszok

A szakaszokat az $x$ tengelyre eső vetületük szerint szakasz-fában tároljuk, és ebben keressük $x$-et.




A keresés során a fa egy adott $v$ csúcsánál ASSOC($v$)-ben azok a ferde szakaszok vannak, amelyek vetülete tartalmazza Int($v$)-t, de nem tartalmazza Int($p(v)$)-t. Ezek tehát mind átmennek az Int($v$) $\times (-\infty, \infty)$ függőleges végtelen csíkon. Mivel nem metszik egymást, így alulról felfelé egyértelműen rendezhetők, tehát eszerint tároljuk őket az ASSOC($v$)-t tartalmazó tömbben. Kereséskor felező kereséssel könnyen megtalálhatjuk a legalsó olyat, mely metszi a kérdezett $[x] \times [y_{\mathbf{a}}, y_{\mathbf{f}}]$ szakaszt, majd ettől kezdve addig íratjuk ki ASSOC($v$) elemeit, míg metszik ezt a kérdezett szakaszt. A $v$-nél eltöltött idő így $O(\log n + k_v)$, tehát az összes keresési idő $O(\log^2 n + k)$.

Az egész struktúra továbbra is $O(n \cdot \log n)$ tárat használ, és felépíthető $O(n \cdot \log^2 n)$ időben (a szakasz-fa felépítése után az ASSOC($v$) halmazokat még rendeznünk kell). Ez egy ügyes (nem triviális) trükkel levihető $O(n \cdot \log n)$ időre.

*Vázlat: rendezzük az intervallum-végpontok $x$ koordinátáit, majd létrehozunk egy segédgráfot, melyben $ij$ él, ha van olyan $x$ koordináta, ahol az $S_j$ közvetlenül az $S_i$ felett van. Állítás: Ennek legfeljebb $3n$ éle lesz.*

*Legyen $x_1 < x_2 < \ldots < x_m$ olyan, hogy a páratlan indexűek az intervallum-végpontok $x$ koordinátái ($m \leq 4n - 1$). Elindulunk egy üres kiegyensúlyozott bináris keresőfával*

*(piros-fekete, AVL- vagy S-fa), és sorban minden $i$-re karbantartjuk, hogy az $x = x_i$ egyenest metsző szakaszok indexei legyenek benne, $y$ koordináta szerint rendezve. Ha $i$ páratlan, akkor be kell szúrni azokat a szakaszokat, melyek kezdőpontjának $x$ koordinátája $x_i$; ha pedig $i$ páros, akkor ki kell törölni azokat, amelyek végpontjának $x$ koordinátája $x_{i-1}$. Ha beszúrunk egy $S_j$ szakaszt ($S_a$ és $S_b$ közé), akkor a segédgráfba felveszünk egy $aj$ és egy $jb$ élet, ha pedig kitörlünk egy $S_j$ szakaszt ($S_c$ és $S_d$ közül), akkor felveszünk egy $cd$ élet. (Tehát $S_j$ miatt legfeljebb 3 új élet veszünk be.)*

*Végül $O(n)$ időben meghatározzuk a segédgráf topológiai sorrendjét, és a szakasz-fa felépítésénél ebben a sorrendben megyünk végig a szakaszokon az* ASSOC*-ok meghatározásánál.*

# 12. Dinamikus utak és fák

Ez a fejezet négy részből áll, először a dinamikus utakat tárgyaljuk, majd az irányítatlan dinamikus fákat, ezután az irányított fákat. Végül ez utóbbi felhasználására mutatunk egy fontos példát (Dinic algoritmusát), mely motiválta ezek kifejlesztését.

## 12.1. Dinamikus utak

Diszjunkt irányított utak egy kollekcióját kell tárolnunk, ahol még minden $v$ csúcsra egy **érték**$(v)$ is rá van írva.

Műveletek:

Újút $(v)$: létrehoz egy csak $v$-ből álló utat, érték $(v) = 0$.
Útnév $(v)$: visszaadja a $v$-t tartalmazó út nevét.
Vége $(p)$: a $p$ út utolsó csúcsa.
Érték $(v)$: visszaadja a $v$ csúcs értékét.
Út-minérték $(p)$: visszaad egy $x, u$ párt, ahol $x$ a $p$ út csúcsai értékének minimuma, $u$ pedig az utolsó csúcs ezzel az értékkel.
Út-értéknöv $(p, \Delta)$: a $p$ út minden csúcsának értékét megnöveli $\Delta$-val (ami negatív is lehet).
Út-egyesít $(p, v, q)$: a $p$ végéhez illeszti a $v$ csúcsot, majd ehhez a $q$ út elejét, visszaadja az egyesített utat ($p$ ill. $q$ lehet üres is).
Út-vág $(v)$: a $v$-t tartalmazó $p'$ utat szétvágja. Visszaad egy $p, q$ útpárt, ahol $p$ a $v$ előtti csúcsok részútja, $q$ pedig a $v$ utániaké.

Megvalósítás: minden egyes utat egy-egy S-fában tárolunk, ahol $u < v$, ha $u$ előrébb van, mint $v$. Az út neve az aktuális gyökér-csúcs neve (ez egy kicsit trükkös, de meggondolható, hogy nem okoz félreértést és nagyon praktikus).

Az igazi ötlet az értékek tárolásánál van. Az S-fa egy $v$ csúcsára definiáljuk a következő mennyiségeket (jelölje $p_S(v)$ a $v$ szülőjét ebben a fában):

- minérték $(v)$ = a $v$ csúcs S-fabeli leszármazottai értékének a minimuma
- $\Delta$érték $(v)$ = érték $(v)-$minérték $(v)$
- $\Delta$min $(v)$ = minérték $(v)$, ha $v$ a gyökér, egyébként minérték $(v)-$minérték $(p_S(v))$

Tárolni az egyes csúcsoknál a $\Delta$érték $(v)$ és $\Delta$min $(v)$ számokat kell. A legtöbb művelet megvalósítása elég nyilvánvaló, csak egyrészt azt kell meggondolni, hogy egy Bill műveletnél ezeket a számokat is karban tudjuk tartani konstans lépésben. Másrészt pl. Út-értéknöv $(p, \Delta)$ műveletnél elég a $\Delta$min $(p)$ számhoz $\Delta$ értéket hozzáadni (emlékeztető: $p$ az S-fa gyökere). Csak a legérdekesebb műveletet írjuk le részletesen.

Út-minérték $(p)$:
$w := p$
**while** $\Delta$érték $(w) > 0$ || $(jobb(w) \neq nil$ && $\Delta$min $(jobb(w)) = 0)$
    **if** $jobb(w) \neq nil$ && $\Delta$min $(jobb(w)) = 0$ **then** $w := jobb(w)$

**else** $w := bal(w)$
**return** $(\Delta\min(p), w)$

**17. Tétel.** *Az útműveletek amortizációs ideje* $O(\log n)$ *az S-fánál tanult elemzés miatt.*

## 12.2. Irányítatlan dinamikus fák

Diszjunkt fák egy kollekcióját kell tárolnunk, ahol minden $v$ csúcsra egy **érték**$(v)$ van ráírva.

Műveletek:

Újfa $(v)$: létrehoz egy csak $v$-ből álló fát, érték $(v) = 0$.

Érték $(v)$: visszaadja a $v$ csúcs értékét.

Minérték $(v)$: visszaadja a $v$-t tartalmazó fa csúcsai értékének a minimumát és egy $u$ csúcsot, ahol ez felvétetik.

Értéknöv $(v, \Delta)$: a $v$-t tartalmazó fa minden csúcsának értékét megnöveli $\Delta$-val (ami negatív is lehet).

Egyesít $(v, w)$: ha $v$ és $w$ két különböző fában vannak, akkor a két fát összeköti a $vw$ él hozzáadásával.

Vág $(v, w)$: ha $vw$ egy fa-él, akkor az ezt tartalmazó fát kettévágja a $vw$ él törlésével.

Megvalósítás: egy $T = (V, E)$ fához hozzárendeljük a $T' = (V, A)$ irányított Euler-gráfot: a fa minden élét bevesszük mindkét irányban, plusz minden csúcsához egy hurokélet. Ennek egy Euler-sétáját valahol szétvágjuk, majd vesszük a kapott nyílt séta élgráfját. Ennek tehát pontjai $A$ elemei, és $aa'$ éle, ha a szétvágott sétában $a$ után $a'$ következik. Tehát ez egy út, a hurokéleknek megfelelő pontokra képzeljük a megfelelő eredeti csúcs értékét, az eredeti fa-éleknek megfelelő pontokra pedig egy elegendően nagy értéket. Ezt az utat az előző fejezetben leírtak szerint tároljuk. Nem nehéz meggondolni, hogy a famű-veleteket meg tudjuk valósítani a megfelelő útműveletek segítségével (az Egyesít műveletnél ez nem triviális, kell néhány Út-vág és Út-egyesít, de csak konstans sok).

**18. Tétel.** *Az irányítatlan famű-veletek amortizációs ideje* $O(\log n)$.

**Megjegyzés:** Ilyen fákat használnak pl. a minimális költségű folyam feladatra kifejlesztett egyik leghatékonyabb algoritmus, a „network simplex" implementálásánál.

## 12.3. Irányított dinamikus fák

Diszjunkt gyökeres fák egy kollekcióját kell tárolnunk, ahol minden $v$ csúcsra egy **érték**$(v)$ van ráírva. A fákat úgy képzeljük, hogy élei a gyökér felé vannak irányítva.

Műveletek:

Újfa $(v)$: létrehoz egy csak $v$-ből álló fát, érték $(v) = 0$.

Gyökér $(v)$: visszaadja a $v$ csúcsot tartalmazó fa gyökerét.

Út $(v)$: a $v$-t tartalmazó fában a $v$-től a gyökérig menő irányított út.

Minérték $(v)$: visszaad egy $x, u$ párt, ahol $x$ az Út $(v)$ csúcsai értékének minimuma, $u$ pedig az utolsó csúcs ezzel az értékkel.

Értéknöv $(v, \Delta)$: az Út $(v)$ minden csúcsának értékét megnöveli $\Delta$-val (ami negatív is lehet).

Egyesít $(v, w)$: ha $v$ és $w$ két különböző fában vannak és $v$ gyökér, akkor a két fát összeköti a $vw$ irányított él hozzáadásával.

Vág $(v)$: ha $v$ nem gyökér, akkor kitörli a $v$-ből a $p(v)$ szülőbe menő irányított élet, ezzel kettévágja a fát egy $v$-t tartalmazó és egy $p(v)$-t tartalmazó fára.

Megvalósítás: Minden egyes fát utak összelinkelt halmazaként fogunk tekinteni. A fa néhány élét *folytonosnak*, másokat *szaggatottnak* hívunk, azzal a kikötéssel, hogy egy csúcsba csak maximum egy folytonos él mehet be. Ezáltal a folytonos élek utakat alkotnak, ezeket folytonos utaknak nevezzük. Egy ilyen $p$ út utolsó csúcsából kilépő szaggatott él végét $sz(p)$-vel jelöljük ($nil$, ha az út a gyökérben végződik). Időben változni fog, hogy mely élek folytonosak ill. szaggatottak. A fő ötlet az, hogy egy belső művelet segítségével elérjük, hogy egy adott $v$ csúcsra Út $(v)$ (tehát a $v$-ből a gyökérbe vezető út) pontosan egy folytonos út legyen. A folytonos utakat az első részfejezetben leírt módon tároljuk, így a faműveletek pontosan megfelelnek az ott leírt útműveleteknek.

A belső művelet, melyet Exponál $(v)$-nek hívunk tehát arra hivatott, hogy Út $(v)$ élei folytonosak legyenek, az ezen belüli csúcsokba vezető többi él pedig szaggatott.

Exponál $(v)$:
$p := null$
**while** $v \neq nil$
    $w := sz(\text{Útnév}\,(v))$
    $q, r :=$Út-vág $(v)$
    **if** $q \neq null$ **then** $sz(q) := v$
    $p :=$Út-egyesít $(p, v, r)$
    $v := w$
$sz(p) := nil$
**return** $(p)$

Először ezt a műveletet elemezzük. Egy $uv$ fa-élet *nehéznek* nevezünk, ha $2 \cdot \text{size}(u) > \text{size}(v)$, különben *könnyűnek* (ahol size$(u)$ az $u$ leszármazottainak a száma, tehát azon csúcsok száma, melyekből irányított fa-éleken $u$ elérhető).

**21. Állítás.** *Egy csúcsba maximum egy nehéz él lép be. Bármely irányított út mentén legfeljebb* $\log n$ *könnyű él lehet.*

Egyelőre kivételesen egy LÉPÉS legyen konstans sok **útművelet** ideje, oly módon, hogy egy Exponálás tényleges ideje $1 + k$ LÉPÉS legyen, ha a while ciklust $k$-szor hajtjuk végre. Azt állítjuk, hogy az Exponálás amortizációs ideje $O(\log n)$ LÉPÉS. Legyen a $P$ potenciál a fa-élek száma mínusz a **nehéz folytonos élek** száma. Ez kezdetben nulla (nincsenek fa-élek), és mindig nemnegatív (csak fa-éleket hívhatunk nehéznek). Ha egy ciklus elején $v$-be megy be folytonos él, legyen ez $u'v$, a Vége $(p)$-ből belépő szaggatott él pedig $uv$. A ciklusban $u'v$ (ha létezik) szaggatottá válik, $uv$ pedig folytonossá. Ha $uv$ nehéz, akkor $u'v$ vagy nem létezik, vagy könnyű, ezért $\Delta P = -1$, így az amortizációs

idő nulla. Különben $uv$ rajta van a kiindulási $v$-ből a gyökérig vezető úton, tehát ilyen legfeljebb $(\log n)$-szer fordulhat elő. Ezekben az esetekben $\Delta P \leq 1$, így az amortizációs idő legfeljebb 2, összesen tehát az egész Exponálás amortizációs ideje $1 + 2\log n$ LÉPÉS.

Mivel az útműveleteket meg tudtuk valósítani $O(\log n)$ elemi lépéssel (amortizációs időben), ezért egy LÉPÉS átlagosan $O(\log n)$ elemi lépés. Mivel a faműveleteket könnyen meg tudjuk valósítani egy (vagy két) Exponálás után konstans sok útművelettel, így a következő tétel adódik, amennyiben belátjuk, hogy az Exponálás(ok) után végzett konstans sok útművelet során a potenciál legfeljebb $O(\log n)$-nel nő.

**19. Tétel.** *Az irányított faműveletek amortizációs ideje* $O(\log^2 n)$.

Elég az Egyesít $(v, w)$ és a Vág $(v)$ műveleteknél megvizsgálni a potenciál változását.

Az Egyesít $(v, w)$ művelet így néz ki:
$sz(\text{Út-egyesít}\,(null, \text{Exponál}\,(v), \text{Exponál}\,(w))) := nil$.
Az Exponálások után $v$ egypontú folytonos út lesz, míg a $w$-ből a gyökeréig vezető út egy másik folytonos út lesz. Az út-egyesítés után eggyel több fa-élünk lesz, valamint néhány folytonos él nehézzé válhat, tehát $\Delta(P) \leq 1$ az Út-egyesít művelet során.

A Vág $(v)$ művelet így néz ki:
Exponál $(v)$; $q, r :=$Út-vág $(v)$; $sz(v) := sz(r) := nil$
Itt bizony az Exponál $(v)$ művelettel keletkező folytonos úton néhány nehéz él könnyűvé válhat, de az Állítás miatt legfeljebb $\log n$ darab, ezért $\Delta(P) \leq \log n$ az Út-vág művelet során.

Azonban a kétféle billegtetést lehet együtt kezelni, és közös amortizációs idővel számolni, felhasználva az 1. Megjegyzést. A folytonos utakat tároló S-fákat folytonos fáknak nevezzük, ezekben egy $u$ csúcs szülője $p_S(u)$. Virtuális fákat definiálunk: $p'(v)$ legyen egyenlő $p_S(v)$-vel, ha $v$ nem gyökere egy folytonos fának, különben pedig legyen egyenlő $sz(v)$-vel (ekkor $v$ a neve a $v$-t tartalmazó folytonos útnak). Ez az implementációt is egyszerűsíti [3], másrészt lehetővé teszi az elemzés pontosabbá tételét. Az 1. Megjegyzésben említett saját súlyokat így kell definiálni: egy $v$ csúcs saját súlya legyen $\text{size}(v)$, ha $v$-be nem lép be folytonos él, ha pedig $uv$ egy folytonos él, akkor $\text{size}(v) - \text{size}(u)$. Egy $v$ csúcs $tw(v)$ teljes súlya most az őt tartalmazó folytonos fában vett leszármazottai saját súlyának összege, $r(v)$ rangja pedig $\lfloor \log tw(v) \rfloor$. Az S-fák elemzésénél vett 11. tétel alkalmazásával egy kis munkával [3] kapható:

**20. Tétel.** *Az irányított faműveletek amortizációs ideje* $O(\log n)$.

## 12.4. Alkalmazás Dinic algoritmusára

Emlékeztető: Dinic algoritmusa $n$-szer elkészíti a szintezett maradékhálózatot, abban keres blokkoló folyamot, majd ezzel javít. Az egyszerűség kedvéért a maradékhálózatot $(G = (V, A), r, s, t)$ jelöli, és csak egy blokkoló $f$ folyam keresésével foglalkozunk.

Kezdetben minden $v \in V$ csúcsra hívunk egy Újfa $(v)$ és egy Értéknöv $(v, N)$ faműveletet, ahol $N > \sum_{a \in A} r(a)$ egy elég nagy szám, valamint minden $a$ élre $f(a) := 0$. Egy $a$ él a következő három típus valamelyikébe esik:

- sima, ezekre $f(a) = 0$,

- $a = (u, p(u))$ fa-él, ezekre $f(a) = r(a) -$érték $(u)$, de ezekből csak az input $r(a)$ van tárolva, valamint a dinamikus fában implicite az érték $(u)$,
- kitörölt, ezekre már a végleges $f(a)$ érték rá van írva.

Tehát az $a = (u, p(u))$ fa-élen menő folyamnak az $r(a)$ kapacitástól vett távolságát tároljuk az érték $(u)$-ban. Ez az implicit tárolás teszi lehetővé, hogy ha egy $c \cdot n$ hosszú javító úton javítunk, és ezáltal $k$ él válik telítetté, akkor $k$ faművelettel el tudjuk végezni a folyam növelését az út minden élén.

Négy segéd-eljárást használunk.

Éltörlés $(u, v)$:
**if** $v \neq p(u)$ **then** $A := A - uv$; **return**
**else**
Vág $(u)$
$\Delta, u :=$Minérték $(u)$
Értéknöv $(u, N - \Delta)$
$f(uv) := r(uv) - \Delta$
$A := A - uv$
**return**

Csúcstörlés $(v)$:
**if** $v = s$ **then** Vége
**else**
**for** $\underline{u}v \in A$
Éltörlés $(u, v)$
**return**

Vége:
**for** $\underline{u}p(u) \in A$
Éltörlés $(u, p(u))$
**STOP**

Javítás:
$\Delta, v :=$Minérték $(s)$
Értéknöv $(s, -\Delta)$
$\Delta := 0$
**while** $\Delta = 0$
Éltörlés $(v, p(v))$
$\Delta, v :=$Minérték $(s)$
**return**

A fő eljárás pedig a következő (a fentebb leírt inicializálás nélkül):

```
Blokkoló (G, r, s, t):
repeat
    v :=Gyökér (s)
    if v = t then Javítás
        else
            if ∃vw ∈ A then
                Értéknöv (v, r(vw) − N)
                Egyesít (v, w)
                p(v) := w
                else
                    Csúcstörlés (v)
```

Könnyen látható, hogy minden konstans sok faművelet során legalább egy él státusza megváltozik, márpedig egy él státusza csak kétszer változhat, ezért összesen $O(|A|)$ faművelet kell (és szintén ennyi egyéb elemi művelet).

**21. Tétel.** *Dinic algoritmusa dinamikus fák segítségével összesen* $O(n \cdot m \cdot \log n)$ *lépést igényel.*

# Hivatkozások


[1] Cormen-Leiserson-Rivest-Stein **Új Algoritmusok,** Műszaki Kiadó, 2000.

[2] Aho-Hopcroft-Ullman **Számítógépalgoritmusok tervezése és analízise,** Műszaki Könyvkiadó, 1982.

[3] Robert Endre Tarjan **Data Structures and Network Algorithms,** Philadelphia: Society for Industrial and Applied Mathematics, 1983.

[4] Rónyai-Ivanyos-Szabó **Algoritmusok,** Typotex Kiadó, 1998.

[5] D. E. Knuth **A számítógép-programozás művészete, III. kötet,** Műszaki Könyvkiadó, 1994.

[6] L. Lovász **Algoritmusok bonyolultsága,** ELTE jegyzet, 1992. ill. `http://www.cs.elte.hu/~kiraly/alg.pdf`

[7] Berg-Kreveld-Overmars-Schwarzkopf **Computational Geometry: Algorithms and Applications,** Springer-Verlag, 1997.