I. tétel - Adatszerkezetek és típusok

Az adattípus absztrakciós szintjei – a **típusspecifikáció és típus fogalma**, absztrakt adattípus és megadási módjai, verem ADT **axiomatikus** és funkcionális leírása, absztrakt adatszerkezet megadása, reprezentáció, aritmetikai és láncolt ábrázolás, az adatszerkezetek osztályozása (több szempont szerint), statikus és dinamikus szerkezetek

Az adattípus absztrakciós szintjei – a **típusspecifikáció és típus fogalma**

A **típus** egy adat által felvehető lehetséges értékek halmazán kívül megadja az adaton értelmezett műveleteket is. A **típus-reprezentáció**, tehát a típusértékek ábrázolásához szükséges

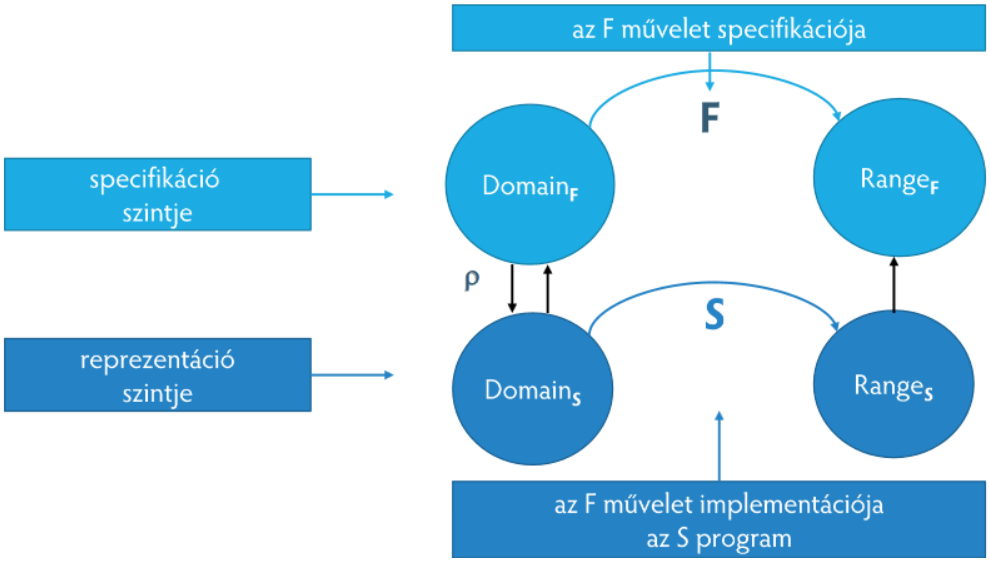
- Ábrázolandó elemek egy H, illetve az ábrázoláshoz szükséges T halmaz továbbá
- egy az ábrázolandó elemek és típusértékek kapcsolatát leíró leképezés

$$\rho: H \to T, \rho \subseteq H \times T$$

- és az ún. **típus-invariáns**, mely kiválasztja a hasznos ábrázoló elemeket.

A **típus-specifikáció** egy adat külső jellemzésére szolgál, megadja a

- **típus érték halmazt** - azaz, hogy az adat milyen értékeket vehet fel, továbbá a
- **típus műveleteket**, vagyis a T-n értelmezett feladatokat.



Absztrakt adattípus és megadási módjai

Az **absztrakt adattípus** *(abstract datatype - ADT)* a típus-specifikáció közvetett megadására szolgál. Ebben az esetben semmilyen feltételezéssel nem élünk a típus szerkezetéről, megvalósításáról, így a specifikációban kizárólag matematikai fogalmakat használunk. Jellemzi még az enkapszuláció (egységbe zárás).

Verem ADT **axiomatikus** és funkcionális leírása

Az ADT megadható axiomatikusan (algebrai) és funkcionálisan is, más kritériumokkal.

- **Axiomatikus:**
  - Részei: típusértékhalmaz, műveletek, megszorítások, axiómák
  - Kérdések: helyesség, teljesség, redundancia
- **Funkcionális:**

- Részei: típusérték halmaz, műveletek, állapottér, paramétertér, előfeltétel, utófeltétel

Ezekre példaként tekintsük a *Verem* adatszerkezetet, melynek az ADT axiomatikus leírása:

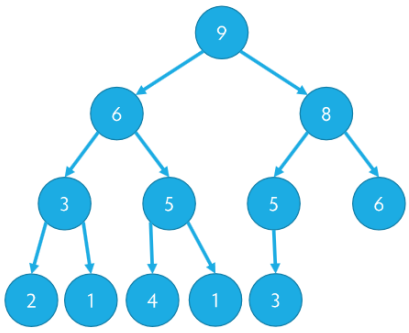
Műveletek	• empty • isempty • push • pop • top	*Üres verem létrehozása* *Üres a verem?* *Elem betétele a verembe* *Elem kivétele a veremből* *Felső elem lekérdezése*
Megszorítások	$D_{top,pop} = V \setminus \emptyset$	*üres vermen nincs értelmezve*
Axiómák	• $\text{isempty}(\text{empty})$ • $\text{isempty}(v) \rightarrow v = \text{empty}$ • $\neg\text{isempty}(\text{push}(v, e))$ • $\text{top}\big(\text{push}(v, e)\big) = e$	

Illetve az ADT funkcionális leírása

Matematikai reprezentáció	• A verem rendezett párok halmaza $v = \{(e1, t1), \dots, (ei, ti) \mid tj - \text{k különbözőek}\}$ • A két komponens a veremben elhelyezett érték, illetve a behelyezés időpontja
Megszorítások	• Az idő értékek különbözőek
Megjegyzés	• Valóságban nem használjuk mert egy-egy függvény leírása is borzalmasan hosszú lenne

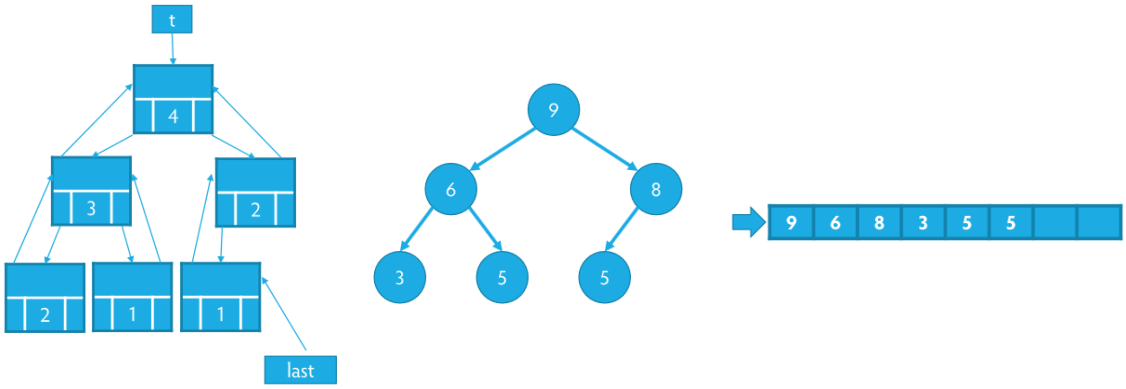
Absztrakt adatszerkezet megadása, reprezentáció, aritmetikai és láncolt ábrázolás

Az **absztrakt adatszerkezeteknél** (*abstract data structure - ADS*) a típus alapvető - absztrakt - szerkezetét egy irányított gráffal ábrázoljuk, melyben a csúcsok az adatelemeket, az élek pedig egy rákövetkeztetési relációt jelölnek.



Az absztrakt adatszerkezetet ábrázolhatjuk pointerekkel (**láncolt ábrázolás**), vagy cím-/indexfüggvénnyel (**aritmetikai reprezentáció**), illetve természetesen vegyes ábrázolás is lehetséges.

A láncolt ábrázolás során az éleket pointerekkel ábrázolhatjuk. A műveletek algoritmusait meg kell adni. Index- és címfüggvényeknél az adatokat folytonosan helyezzük el az absztrakt memóriában / egy vektorban. Az elemek közötti rákövetkeztést egy ilyen függvénnyel adhatjuk meg. A műveletek algoritmusait ebben az esetben is meg kell adni.

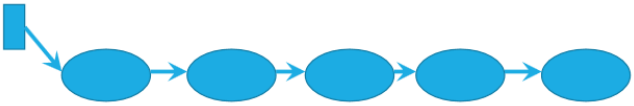


Az adatszerkezetek osztályozása (több szempont szerint), statikus és dinamikus szerkezetek

Az **adatszerkezet** egy $\langle A, R \rangle$ rendezett pár, ahol A az adatelemek véges halmaza, R az A halmazon értelmezett valamilyen reláció. $R \subseteq A \times A$

Az adatszerkezeteket **osztályozhatjuk** az alábbiak szerint

- **Adatelemek típusa**
  - homogén: Az adatszerkezet valamennyi eleme azonos típusú
  - heterogén Az adatszerkezet elemei különböző típusúak lehetnek
- **Az R reláció szerint**
  - Struktúra nélküli: Az adatelemek közt nincs kapcsolat (pl. halmaz)
  - Asszociatív címzés: Lényegi kapcsolat nincs, de tartalom alapján címezhetők
  - Szekvenciális: A szekvenciális adatszerkezetnél az R reláció tranzitív lezártja (tranzitív, tartalmazza R-t és minimális elemszámú) teljes rendezési reláció. Az egyes adatelemek egymás után helyezkednek el, minden elem csak egy helyről elérhető, a két kitüntetett elem az első és az utolsó.

    
  - Hierarchikus: Van egy kitüntetett r elem, a gyökérelem, úgy, hogy ez nem lehet végpont, minden más csúcs pedig max egyszer lehet végpont és elérhető a gyökérből
  - Hálós: Ebben az esetben nincs megkötés a relációra. Az adatelemekhez több helyről is eljuthatunk, több irány is van (például gráf).
- **Adatelemek száma szerint**
  - Statikus: rögzített számú adatelem van, a memóriában elfoglalt hely állandó
  - Dinamikus: adatelemek száma véges, de tetszőlegesen változhat
- **Reprezentáció szerint**
  - Folytonos: A központi tárban az elemek egymás után helyezkednek el
  - Szétszórt: Az elemek véletlenszerűen helyezkednek el, de minden elem tartalmaz a többi elem elhelyezkedésére vonatkozó információt.

II. tétel - Objektumorientált programozás

Osztályok és objektumok, objektum létrehozása, inicializálása, példányváltozó, példánymetódus, osztályváltozó, osztálymetódus, **öröklődés, polimorfizmus, dinamikus összekapcsolás (példákkal)**, absztrakt osztály, a többszörös öröklődés problémái, lehetséges megoldásai

Osztályok és objektum

Egy **objektumnak** belső állapota van, melyben információkat tárol (adattagokkal valósítjuk meg), kérésre feladatokat hajt végre, melyekkel saját állapotát is képes változtatni, más objektumokkal kommunikálhat. Minden objektum egyértelműen azonosítható.

Az **osztály** egy olyan objektumminta vagy típus, mely alapján példányokat (objektumokat) hozhatunk létre. Egy példány egy adott osztály alapján létrejött konkrét példány. Minden objektum egy osztályhoz tartozik.

Objektum létrehozása, inicializálása

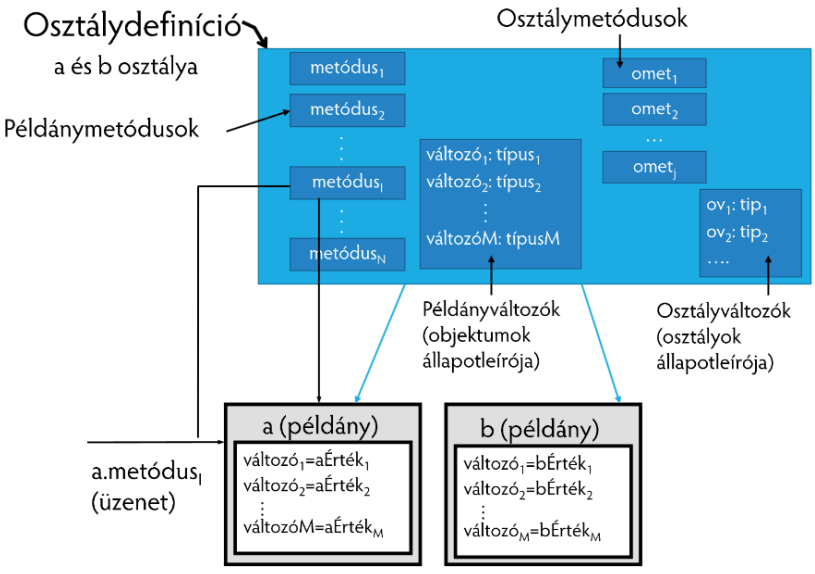
Az objektumokat minden esetben létre kell hozni és inicializálni kell. Az objektum inicializálását az ún. konstruktor végzi. Megadhatjuk az adatok kezdőértékét, illetve végrehajtjuk vele az objektum működéséhez szükséges tevékenységeket.

Nem létezik objektum konstruktor nélkül, ha nem írunk „automatikusan” jön létre egy. A konstruktoroknak nincsen visszatérési értéke. A konstruktor párja a destruktor, mely az elem törlésekor hívódik meg.

Példányváltozó, példánymetódus, osztályváltozó, osztálymetódus

Osztáldefiníció:

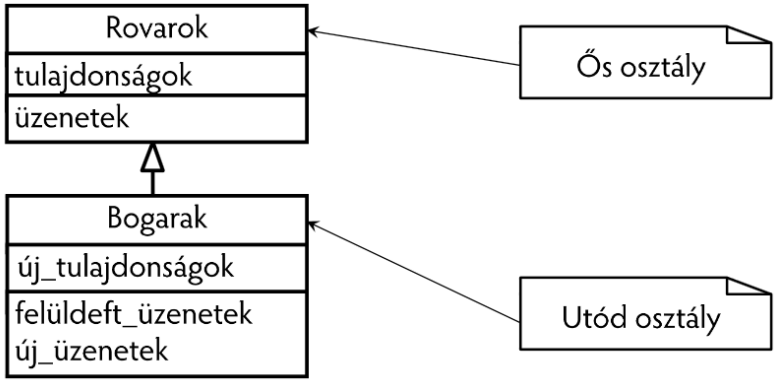
- **Példányváltozó**: példányonként helyet foglaló változó.
- **Példánymetódus**: példányokon dolgozó metódus.
- **Osztályváltozó**: osztályonként helyet foglaló változó.
- **Osztálymetódus**: osztályokon dolgozó metódus.



Öröklődés, polimorfizmus, dinamikus összekapcsolás (példákkal)

Az **öröklődés** alapgondolata, hogy a gyerekek örökölik őseik metódusait és változóit. Az ősosztály minden metódusa és adattagja a gyerekosztálynak is metódusa és adattagja lesz.

A **leszármazott** bevezethet új műveleteket és adattagokat, ezek egyszerűen hozzáadódnak az öröklött adattagokhoz, vagy akár felül is írhatja ezeket.



A **polimorfizmus** (többalakúság) az a jelenség, hogy egy változó nem csak egyfajta típusú objektumra hivatkozhat. Statikus típusnak nevezzük azt a típust, melyet a deklaráció során kap, dinamikusnak azt, amilyen típusra éppen hivatkozik az objektum (vagy a statikus típus, vagy annak valamiféle leszármazottja).

```
Employee emp ("Kati","Fekete", 3);
Manager m ("Jozsef", "Kovacs", 3, 2);
emp = m;
```

Altípusos polimorfizmusnak nevezzük azt a jelenséget, ha B altípusa az A típusnak, és B objektumainak referenciái értékül adhatók az A típus referenciáinak.

Az a jelenség, hogy a változó az éppen aktuális dinamikus típusának megfelelő metódus implementáció hajtódik végre, az ún. **dinamikus összekapcsolás.**

```
Employee* empp=new Employee ("Istvan", "Nagy",5);
…
Manager* mp=new Manager("Laszlo", "Hajto",2,3);
…
empp = mp;
…
empp->print();
```

Absztrakt osztály

Egy felső szinten fogja össze a közös tulajdonságokat. A metódusok között akár olyan is lehet, melynek csak specifikációja van, törzse nincs. Közvetlenül nincs értelme példányt létrehozni, egy leszármazott teszi konkréttá. (pl. Widget osztály)

A többszörös öröklődés problémái, lehetséges megoldásai

Egy adott osztálynak akár több közvetlen őse is lehet. Ilyenkor felmerülhet problémaként, hogy egyes adattagok hányszor szereplenek a leszármazottban, és hogy melyik metódust hívja meg ütközés esetén?

A legtöbb ilyen esetben a kódot nem lehet lefordítani, a fordító vagy a futtató hibát jelez. Amennyiben ezen továbbjutunk vagy az ősosztály döntsön, hogy mi történjen ilyen esetben, vagy a származtatott osztály mondja meg, hogy melyiket akarja használni.

III. tétel - Tömbök

Definíció ADT és ADS szinten, reprezentáció, sorfolytonos és oszlopfolytonos ábrázolás**, cím és index függvény**, speciális mátrixok -tridiagonális, alsó háromszög mátrix stb., cím és index függvényei, hézagos mátrixok reprezentációja

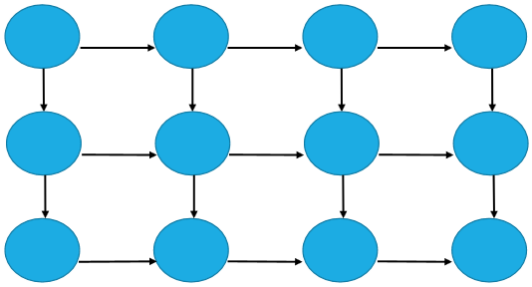
Definíció ADT és ADS szinten

(ADT) Az E alaptípusú $k \geq 1$ dimenziós T tömbtípus

- Legyen $I = I_1 \times I_2 \times \ldots \times I_k$ egy indexhalmaz, ahol $\forall j \in [1,k]: I_j = [1, n_j]$
- Az $A \in T$ tömbnek $N = n_1 * \ldots * n_k$ eleme van $\{a_1, \ldots, a_N\}$
- Mindig van egy $f: I \to \{a_1, a_2, \ldots, a_N\}$ bijektív leképezés.
- Műveletek
  - Indexelés: az $i_1, \ldots, i_k$ indexhez tartozó $A[i_1, i_2, \ldots, i_k]$ elem kiválasztása
  - Elemmódosítás, értékadás: $A[i_1, \ldots, i_k] = a$
  - Értékadás: $A \coloneqq B$
- Elnevezés
  - $k = 1$ (vektor), $k = 2$ (mátrix)

(ADS)

- Itt nem szükséges szerkezetet definiálni az elemek között.
- Elfogadott a köv_j reláció bevezetése $\forall j \in [1,k]$
  - ha $i_j < n_j$, akkor $\text{köv}_j(A[i_1, \ldots, i_j, \ldots, i_k]) = A[i_1, \ldots, i_{j+1}, \ldots, i_k]$, máskülönben nem értelmezzük
  - egy belső elemnek minden dimenzióban van rákövetkezője.
- (megjegyzés) A legalább 2 dimenziós tömböt ortogonális adatszerkezetnek nevezik.



Reprezentáció, sorfolytonos és oszlopfolytonos ábrázolás, **cím és index függvény**

Az aritmetikai ábrázolása során egy k dimenziós tömböt szeretnénk elhelyezni egy alkalmas méretű egydimenziós tömbben, és megadjuk a leképezés címfüggvényét. Az elhelyezés vagy sorfolytonos, vagy oszlopfolytonos lehet.

(n_1: sor, n_2: oszlop)

Indexfüggvény: $\forall i \in [1, n_1], \forall j \in [1, n_2]$

- SF: $\text{ind}(A[i,j]) = (i-1) * n_2 + j$
- OF: $\text{ind}(A[i,j]) = (j-1) * n_1 + i$

Címfüggvény: $\forall i \in [1, n_1], \forall j \in [1, n_2]$

- SF: $\text{cím}(A[i,j]) = \text{cím}(A) + (i-1) * n_2 * h + (j-1) * h$

- OF: $\text{cím}(A[i,j]) = \text{cím}(A) + (j-1) * n_1 * h + (i-1) * h$

Speciális mátrixok -tridiagonális, alsó háromszög mátrix stb.

$a_{1,1}$	$a_{1,2}$	0	0	0	0			
$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	0	0	0			
0	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	0	0			
0	0	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	0			
0	0	0	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	…		
					…			

$a_{1,1}$	$a_{1,2}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	…		

$a_{1,1}$	0	0	0	0			
$a_{2,1}$	$a_{2,2}$	0	0	0			
$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	0	0			
$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	0			
$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	…		
				…			

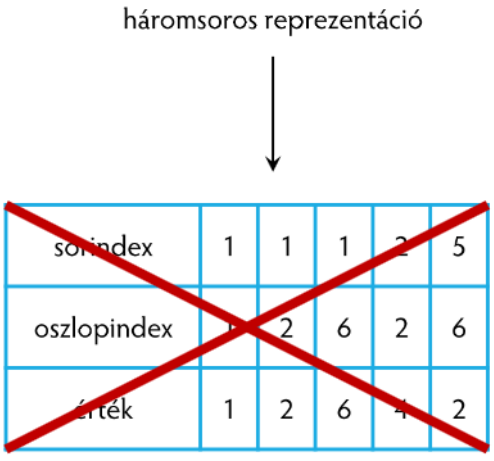
$a_{1,1}$	$a_{2,1}$	$a_{2,2}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	…	0

Hézagos mátrixok reprezentációja

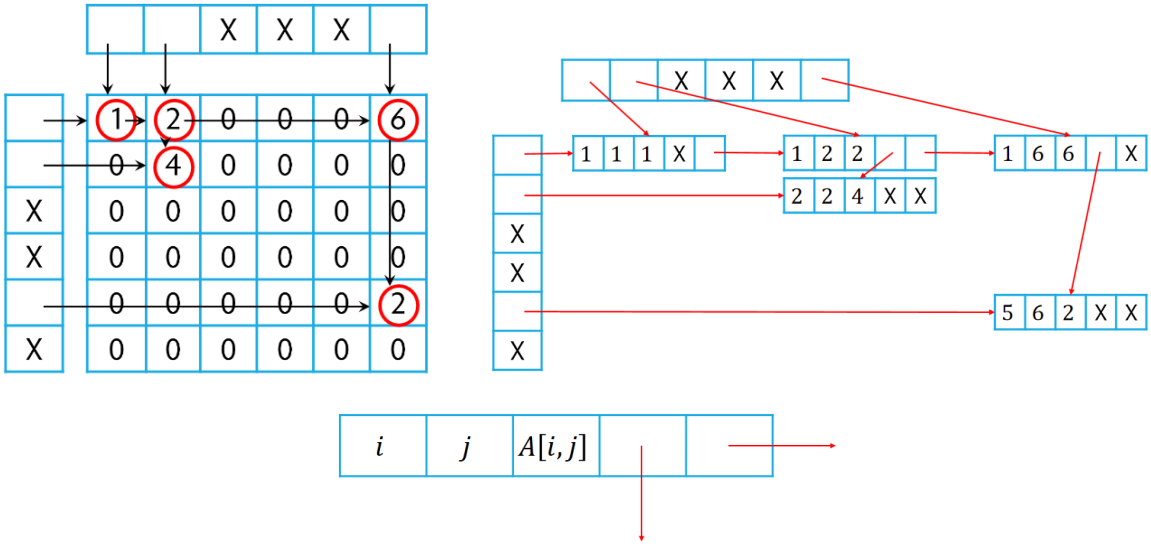
Hézagosan kitöltött mátrixok esetén a reprezentáció nehezebb, hiszen a 0-kat tárolni pazarlás lenne. Észszerűnek tűnik az ún. Háromsoros reprezentáció, de ehelyett praktikusabb ha láncolt ábrázolást használunk.

1	2	0	0	0	6
0	4	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	2
0	0	0	0	0	0

háromsoros reprezentáció



sorindex	1	1	1	2	5
oszlopindex	1	2	6	2	6
érték	1	2	6	4	2


X X X
1 2 0 0 0 6
0 4 0 0 0 0
X 0 0 0 0 0 0
X 0 0 0 0 0 0
0 0 0 0 0 2
X 0 0 0 0 0 0
X X X
1 1 1 X
1 2 2
1 6 6 X
2 2 4 X X
X
X
X
5 6 2 X X

i
j
A[i, j]

IV. tétel - Verem, sor és használatuk

ADT **axiomatikus** és funkcionális specifikáció, ADS, **statikus** és dinamikus reprezentáció, **műveletek megvalósítása (algoritmusok)**; Kifejezések lengyelformára alakításának és a lengyelforma kiértékelésének algoritmusai

ADT **axiomatikus** és funkcionális specifikáció

A *Verem* (LIFO) adatszerkezetet ADT axiomatikus leírása:

Műveletek	• empty • isempty • push • pop • top	*Üres verem létrehozása* *Üres a verem?* *Elem betétele a verembe* *Elem kivétele a veremből* *Felső elem lekérdezése*
Megszorítások	$D_{top,pop} = V \setminus \emptyset$	*üres vermen nincs értelmezve*
Axiómák	• $\text{isempty}(\text{empty})$ • $\text{isempty}(v) \rightarrow v = \text{empty}$ • $\neg\text{isempty}(\text{push}(v, e))$ • $\text{top}(\text{push}(v, e)) = e$	

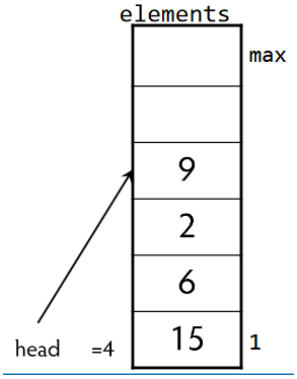
Illetve az ADT funkcionális leírása

Matematikai reprezentáció	• A verem rendezett párok halmaza $v = \{(e1, t1), \dots, (ei, ti) \mid tj - \text{k különbözőek}\}$ • A két komponens a veremben elhelyezett érték, illetve a behelyezés időpontja
Megszorítások	• Az idő értékek különbözőek
Megjegyzés	• Valóságban nem használjuk mert egy-egy függvény leírása is borzalmasan hosszú lenne

A *Sor* (FIFO) adatszerkezetet ADT axiomatikus leírása:

Műveletek	• empty • isempty • in • out • first	*Üres sor létrehozása* *Üres a sor?* *Elem betétele a sorba* *Elem kivétele a sorból* *Első elem lekérdezése*
Megszorítások	$D_{out,first} = S \setminus \emptyset$	*üres soron nincs értelmezve*
Axiómák	• $\text{isempty}(\text{empty})$ • $\text{isempty}(s) \rightarrow s = \text{empty}$ • $\neg\text{isempty}(\text{in}(s, e))$	

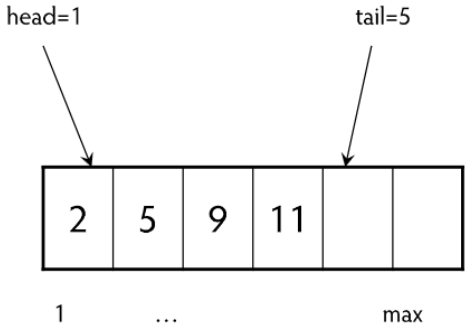
ADS, **statikus** és dinamikus reprezentáció



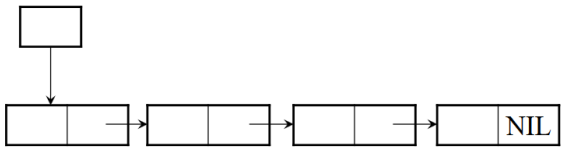
A statikus verem reprezentációnál szükségünk van egy

- max hosszú vektorra (tömbre)
- illetve a verem tetejének a mutatójára (ha ennek az értéke 0, akkor a verem üres.

Statikus sor esetén két mutatót, használunk egy sor végét, illetve sor elejét jelző változót. Ezek a head és a tail. Kezdetben mindketten a sor elejére mutatnak. Ekkor a sor üres. Ugyanez sajnos fent áll teli sornál is, ezért egy változót tartunk arra is, hogy üres e a sorunk.

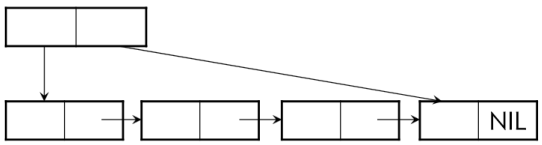


Dinamikus verem ábrázolásnál láncolt módszert használunk. Ebben az esetben az adatszerkezet „egységei” a saját értékükön túl egy mutatót is tartalmaznak a következő elemre. Hasonlóan van egy head mutató, ami a verem utolsó elemére mutat. Amennyiben ez nullpointer a verem üres.



Dinamikus sornál teljesen hasonlóan működnek a dolgok, de a head mutatón kívül még egy tail mutatót is fenntartunk.

Nem létezik objektum konstruktor nélkül, ha nem írunk „automatikusan” jön létre egy. A konstruktoroknak nincsen visszatérési értéke. A konstruktor párja a destruktor, mely az elem törlésekor hívódik meg.



Kifejezések lengyelformára alakításának és a lengyelforma kiértékelésének algoritmusai

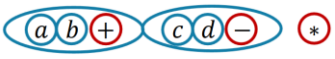
Egy matematikai kifejezés az operandusok elhelyezkedésétől függően lehet

- **Infix**: Amikor az operátor az operandusok között van: $A + B$
- **Postfix**: Amikor az operátor az operandusok mögött van: $AB +$
- **Prefix**: Amikor az operátor az operandusok előtt van: $+AB$

A Lengyel formát J. Lukasewitz lengyel matematikus találta fel. A prefix alakot lengyelformának, a postfixet fordított lengyel formának nevezzük. Hétköznapi életben az infix alakot használjuk. Ennek egy átalakítása

$$(A + B) * (C - D) \rightarrow AB + CD - *$$

A fordított lengyel forma előnye, hogy a műveleti jelek olyan sorrendben követik egymást, amilyen sorrendben végre kell hajtani azokat, illetve az operátor közvetlenül az operadnusai után áll.



Műveletek megvalósítása (algoritmusok)

A statikus verem műveletek megvalósítása pszeudonyelven:

v. empty	v. head ← 0
v. isempty	return v. head = 0
v. isfull	return v. head = max

v. push(*e*)	if v. isfull
	  then error "túlcsordulás"
	else
	  v. head ← v. head + 1
	  v. elements[v. head] ← e
	end if

v. pop	if v. isempty
	  then error "alulcsordulás"
	else
	  v. head ← v. head − 1
	  return v. elements[v. head + 1]
	end if

v. top	if v. isempty
	  then error "alulcsordulás"
	  else return v. elements[v. head]
	end if

A statikus sor műveletek megvalósítása pszeudonyelven:

s. empty	s. head ← 1 s. tail ← 1 s. empt ← true
s. isempty	return s. empt
s. isfull	return (! s. empt && s. head == s. tail)

s. in(*e*)	if s. isfull
	  then error "túlcsordulás"
	else
	  s. empt ← false
	  s. elements[s. tail] ← e
	  if s. s. tail = max
	  then s. tail ← 1
	  then s. tail ← s. tail + 1
	end if

s. out	if s. empt
	  then error "alulcsordulás"
	else
	  e ← s. elements[s. head]
	  if s. head = max
	  then s. head ← 1
	  else s. head ← s. head + 1
	  end if
	  if s. head = s. tail
	  then s. empt ← true
	  end if
	end if

s. First	if s. isempty then error "alulcsordulás"

	else return s. elements[s. head] end if

A láncolt ábrázolás pszeudokódjai itt nem szerepelnek, de logikusan kitalálhatóak.

V. tétel - Listák

Szekvenciális adatszerkezetek definíciója, statikus és dinamikus reprezentáció, műveletek és megvalósítások algoritmusai (**beszúrás**, törlés, ...); rendezés listával

Szekvenciális adatszerkezetek definíciója

A **szekvenciális adatszerkezet** olyan $\langle A, R \rangle$ rendezett pár, amelynél az $R \subseteq A \times A$ reláció tranzitív lezártja teljes rendezési reláció.

Egy reláció **tranzitív lezártja** az a reláció, amely tranzitív, tartalmazza az $R \subseteq A \times A$-t, és a lehető legkevesebb elemet tartalmazza.

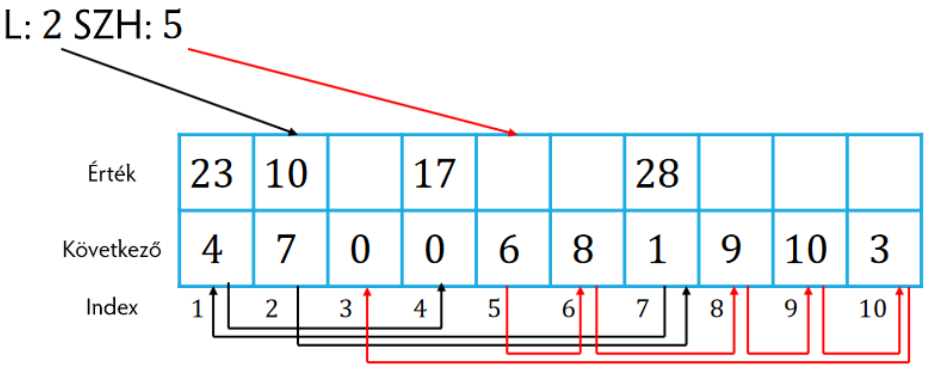
A szekvenciális adatszerkezetben az egyes elemek egymás után helyezkednek el - van egy logikai sorrendjük. Minden adatelem csak egy helyről érhető el, és az adott elemtől csak egy másik látható. Van két kitüntetett elem: az első és az utolsó.

Kitűnő példa a szekvenciális adatszerkezetre a **láncolt lista**, melynek minden eleme egy vagy több mutatót tartalmaz egy másik ugyanolyan típusú elemre (következő elem). Az első elem címét egy „listafejben” tároljuk. Ennek nincs információs része.
A lánc utolsó eleme az az elem, amelynek a rákövetkezője nullpointer.

Statikus és dinamikus reprezentáció

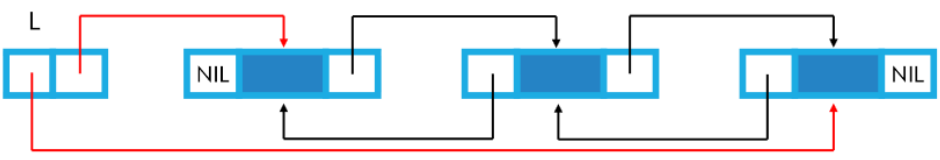
A statikus reprezentációnál szükségünk van

- egy tömbre, melyben az értékeket tároljuk
- egy másik tömbre, melyben a rákövetkezők indexét.



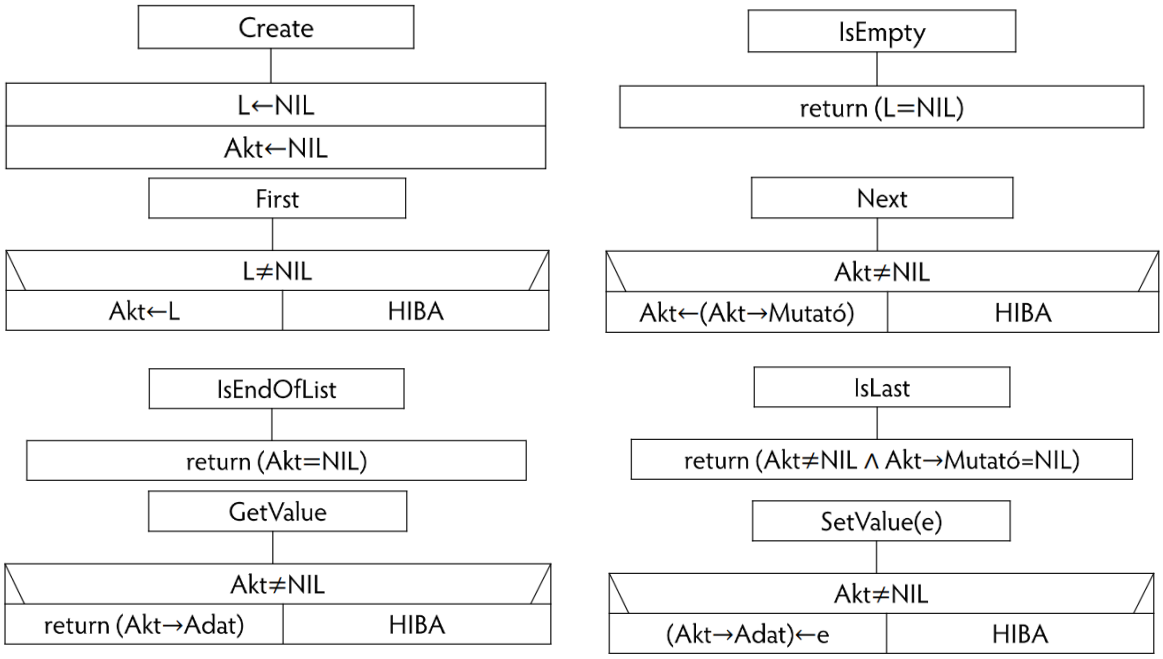
Dinamikus reprezentáció esetén pointereket használunk. Az első elemet mutató pointeren kívül szükségünk van egy plusz mutatóra, mely az aktuális elemre mutat.

Lehetséges két irányú láncoltlista létrehozása is, ekkor a megelőző elemre is tartalmaznak mutatót.

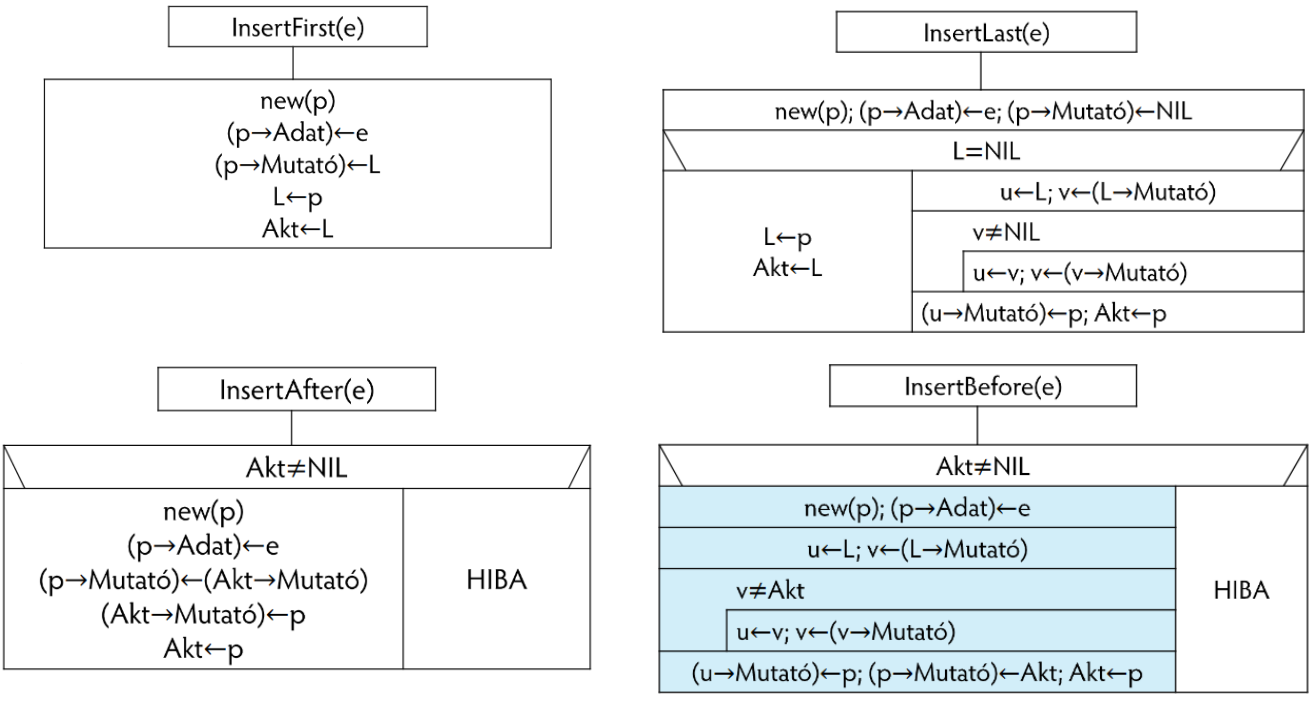


Műveletek és megvalósítások algoritmusai (**beszúrás**, törlés, ...)

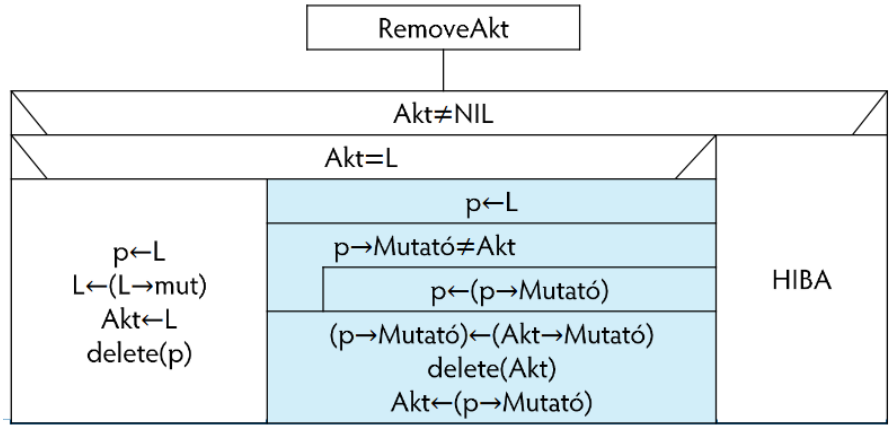
Az alapvető műveletek implementációja struktogrammal:



Az **beszúró** műveletek implementációja struktogrammal:



Az **törlés** művelet implementációja struktogrammal:



VI. tétel - Hierarchikus adatszerkezetek és bináris fák

Definíciók, általános és bináris fák reprezentációi, **bejárásai**, műveletei

Definíciók

A **hierarchikus adatszerkezet** olyan $\langle A, R \rangle$ rendezett pár, amelynél van egy kitüntetett elem: r , amit gyökérelemnek nevezünk. Erre az alábbi tulajdonságok teljesülnek:

- r nem lehet végpont
- minden más elem egyszer és csak egyszer végpont
- minden más elem elérhető a gyökérből.

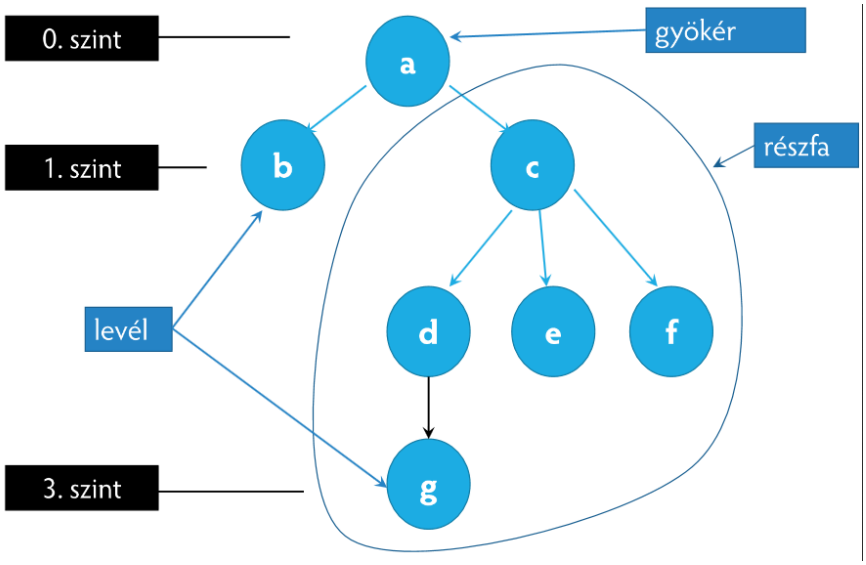
Egy elemnek akárhány rákövetkezője lehet, de minden elemnek csak egyetlen megelőző eleme van. (**egy-sok** kapcsolat)

A **fa** egy olyan hierarchikus adatszerkezet, mely véges számú csúcspontból áll, és igazak a következők:

- Két csomópont között egyirányú a kapcsolat (egyik kezdőpont, másik végpont)
- Van a fának egy kitüntetett csomópontja, amely nem lehet végpont (gyökér)
- Összes többi csomópont pontosan egyszer végpont.

A **fa rekurzív** definíciója:

- A fa vagy üres, vagy
- Van egy kitüntetett csomópontja (gyökér)
- A gyökérhez 0 vagy több diszjunkt fa kapcsolódik.



Az adatszerkezetekben:

- A fa **csúcsai** az adatelemeknek felelnek meg
- Az **élek** az adatelemek közötti kapcsolatot (sorrendet)
- A **gyökérelemnek** nincs megelőzője
- **Levél** az az elem, melynek nincs rákövetkezője
- **Közbenső elem**, ami nem gyökér és nem levél.
- Minden közbenső elem tekinthető egy részbenső fa gyökerének.

- **Elágazásszámnak** nevezzük a közvetlen részfák számát.
- A **fa szintje** a gyökértől való távolságot mutatja (élek száma)
- A fa szintjeinek száma a **fa magassága**.

További definíciók:

- **Csomópont foka**: a csomóponthoz kapcsolt részfák száma.
- **Fa foka**: a fában található legnagyobb fokszám
- **Levél**: 0 fokú csomópont
- **Elágazás**: > 0 fokú csomópont
- **Szülő**: kapcsolat kezdőpontja (csak a levelek nem szülők)
- **Gyerek**: kapcsolat végpontja (csak a gyökér nem gyerek)
- **Testvér**: ugyanazon csomópont leszármazottai.
- **Szintszám**: gyökértől mért távolság.
- **Útvonal**: egymást követő élek sorozata
- **Ág**: az az útvonal, mely levélben végződik
- **Üresfa**: az a fa, melynek nincs eleme
- **Fa magasság**: leghosszabb gyökérből induló ág hossza.
- **Minimális magasságú fa**: az adott elemszám esetén a lehető legkisebb.
- **Kiegyensúlyozott**: ha csomópontjai azonos fokúak, és minden szintjén az egyes részfák magassága nem ingadozik többet egy szintnél.
- **Rendezett fa**: ha az egy szülőhöz tartozó részfák sorrendje lényeges, akkor rendezett.
- **Bináris fa**: olyan fa, melyben minden csúcsnak legfeljebb két gyermeke van.
- **Tökéletesen kiegyensúlyozott**: Minden elem bal és jobb részfájában az elemek száma legfeljebb eggyel tér el.
- **Teljes**: Minden közbenső elemnek pontosan két leágazása van.
- **Majdnem teljes**: Ha csak a leveleknél van hiány.

Általános és bináris fák reprezentációi

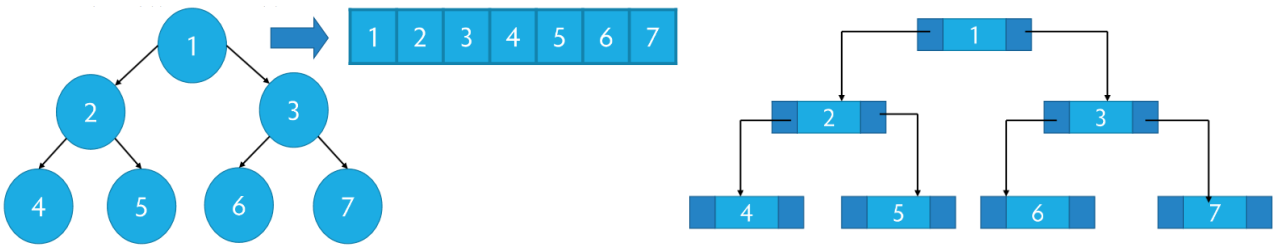
Általános fa esetén úgy reprezentálhatjuk az adatszerkezetet, hogy minden csomópont tartalmaz három mutatót: bal-gyermek, jobb-gyermek, szülő.

Használható továbbá multilista, melyben minden csomóponthoz egy lista tartozik, melynek első eleme az adat, a többi a kapcsolatok listája. Annyi kapcsolati elem, ahány fokú a csomópont.

Korlátos általános fa esetén lehetőség nyílik aritmetikai, illetve láncolt ábrázolásra is.

Aritmetikai ábrázolás esetén szintfolytonosan tároljuk a bináris fát egy tömbben. Adott elem bal és jobb gyermekének az indexe az alábbi összefüggéssel kapható:

- $ind\big(bal(c)\big) = 2 * ind(c)$
- $ind\big(jobb(c)\big) = 2 * ind(c)$



Bejárásai, műveletei

A fák műveletei:

- Lekérdező műveletek
  - ürese *logikai értéket ad vissza*
    - vane_bal / vane_jobb ***(bináris keresőben)***
    -
  - Gyökérelem *visszaadja a gyökér adatelemet*
  - Keres(e) *adott e adatelemet keres, egy ilyen elem mutatóját adja vissza*
- Módosító műveletek
  - üres *létrehoz egy üres fát*
  - beszúr *adott e adatelemet beszúr*
    - beszúr_bal / beszúr_jobb ***(bináris keresőben)***
  - MódGyök(e) *adott e adatelem lesz a gyökér*
  - Töröl(e) *törli az adatelemet (egy vagy összes előfordulást)*
    - részfa törlése
  - Törölfa *összes elemet törli a fában*
  - Részfa helyettesítés

A fák bejárása lehet:

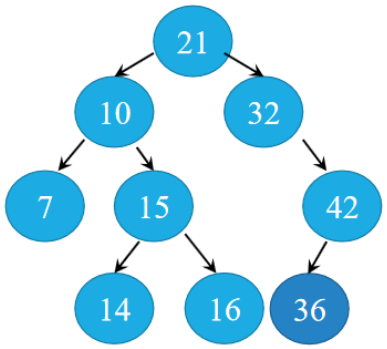
- **Preorder** (gyökérkezdő)
  - gyökérelem kiírása
  - bal részfa preorder bejárása
  - jobb részfa preorder bejárása
- **Indorder** (gyökérközepű)
  - bal részfa inorder bejárása
  - gyökérelem kiírása
  - jobb részfa inorder bejárása
- **Postorder** (gyökérvégző)
  - bal részfa postorder bejárása
  - jobb részfa postorder bejárása
  - gyökérelem kiírása

VII. tétel - Bináris keresőfák

Definíció, műveletek (algoritmusok) (**beszúrás**, törlés, **rákövetkező**, megelőző stb.)

Definíciók

A **bináris keresőfa** olyan bináris fa, melynek kialakítása a különböző adatelemek között meglévő rendezési relációt követi.



A fa felépítése olyan, hogy minden csúcsra igaz az, hogy

- a csúcs értéke nagyobb, mint tetszőleges csúcsé a tőle balra lévő leszálló ágon és
- a csúcs értéke kisebb minden tőle jobbra lévő leszálló ágon található csúcs értékénél.

A rendezési fa az őt tartalmazó elemek beviteli sorrendjét is visszatükrözi. Ugyanazon elemekből különböző rendezési fák is felépíthetők.

A tulajdonságokból következik, hogy **inorder fabejárás** esetén rendezetten kapjuk az elemeket. A fa bejárás pontosan $\sigma(n)$ ideig tart.

Műveletek (algoritmusok) (**beszúrás**, törlés, **rákövetkező**, megelőző stb.)

A **keresés** algoritmust megadhatjuk rekurzív és iteratív megoldásban is, de ez utóbbi a legtöbb számítógépen hatékonyabb.

```
Fában-keres(x, k)
   if x = NIL or k = kulcs[x]
     then return x
   if k < kulcs[x]
     then return Fában-keres(bal[x], k)
     else return Fában-keres(jobb[x], k)
```

```
Fában-iteratívan-keres(x, k)
   while x ≠ NIL and k  ≠ kulcs[x] do
     if k < kulcs[x]
       then x ← bal[x]
       else x ← jobb[x]
   return x
```

Minimum keresés esetén a fa legbaloldalibb, **maximum** esetén a legjobboldalibb elemét keressük.

```
Fában-minimum (T)
  x ← gyökér[T]
  while bal[x] ≠ NIL
    do  x ← bal[x]
  return x
```

```
Fában-maximum (T)
  x ← gyökér[T]
  while jobb[x] ≠ NIL
    do  x ← jobb[x]
  return x
```

Az x csúcs rákövetkező elemét az alábbi algoritmussal kaphatjuk meg:

```
Fában-következő(T, x)
if jobb[x] ≠ NIL
   then return Fában-minimum (jobb[x])
y ← szülő[x]
while y ≠ NIL és x = jobb[y] do
  x ← y
  y ← szülő[x]
return y
```

A h magasságú bináris keresőfában a keresés, a minimum, a maximum és a következő műveletek $\sigma(h)$ idő alatt végezhetők el.

Beszúró művelet esetén a keresés algoritmushoz hasonlóan megkeressük a helyét a fában, majd, ha megtaláltuk oda beillesztjük az elemet. Ez minden esetben a fának egy levele lesz.

```
Fába-beszúr (T,p)
y ←NIL; x ← gyökér[T]
while x  ≠ NIL
  do y ← x
    if kulcs[p] < kulcs[x]
      then x ← bal[x]
      else x ← jobb[x]
szülő[p] ← y
if y = NIL
  then gyökér[T] ← p
  else if kulcs[p] < kulcs[y]
    then bal[y] ←p
    else jobb[y] ←p
```

Törlés esetén sajnos külön kell tekintenünk az alábbi három lehetőséget.

1. p-nek még nincs gyereke: szülőjének megfelelő mutatóját nullpointerre állítjuk, majd p-t töröljük. Ez az egyszerű törlés.
2. p-nek egy gyereke van: ebben az esetben egyszerűen átkötjük a szülő és gyerek kapcsolat, p gyereke lesz p szülőjének a gyereke, p-t töröljük.
3. p-nek két gyereke van: megkeressük p jobbrészfájában a legkisebb elemet (tehát a legbalrább lévőt, majd megcseréljük a kettőt. A csere után törölhetjük az elemet az első két szabály alapján.

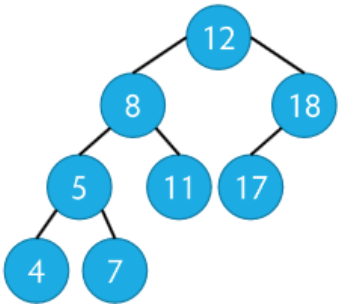
```
if bal[p] = NIL vagy jobb[p] = NIL
  then y ←p                                  -- 0, vagy 1 gyerek
  else y ← Fában-következő(T, p)             -- 2 gyerek
if bal[y]  ≠ NIL
  then x ← bal[y]                            -- x az y 0, vagy 1
  else x ← jobb[y]                              gyerekére mutat
if x  ≠ NIL                                  -- ha volt gyereke, akkor
  then szülő[x] ← szülő[y]                      befűzi az új szülőhöz
if szülő[y] =NIL                             -- ha a gyökeret töröltük
  then gyökér[T ] ←x                            akkor be kell állítani az újat
  else  if  y = bal[szülő[y]]                -- különben a szülőnek megfelelő
    then bal[szülő[y]] ← x                      oldalhoz tartozó mutatót kell
    else  jobb[szülő[y]] ← x                    az x-re állítani
if  y  ≠ p                                   -- amennyiben a ténylegesen
  then kulcs[p] ← kulcs[y]                      törlendő csúcs nem azonos
return y                                        azzal, amit kiláncolunk át kell
                                                írni az adatot is
```

VIII. tétel - AVL fák

Definíció, műveletek (**beszúrás utáni kiegyensúlyozás**, törlés utáni kiegyensúlyozás) megfelelő esetszétválasztással

Definíció

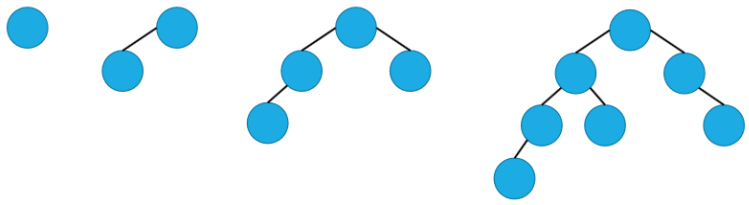
Egy bináris keresőfa AVL-fa, ha minden x csúcsára teljesül, hogy $|\mathrm{m}(\mathrm{bal}[x]) - \mathrm{m}(\mathrm{jobb}[+]) \leq 1|$, ahol $\mathrm{m}(x)$ jelöli egy x-gyökerű részfa magasságát.



Az AVL fa tulajdonságai

- bináris keresőfa
- a bal és jobb részfák magassága legfeljebb 1-gyel különbözik egymástól
- a részfák is AVL fák

$k = 1$ $S_1 = 1$ $k = 2$ $S_2 = 2$ $k = 3$ $S_3 = 4$ $k = 4$ $S_4 = 7$



Mekkora a k-szintű AVL-fa minimális csúcsszáma? Vegyük észre a rekurziót!

Műveletek (**beszúrás utáni kiegyensúlyozás**, törlés utáni kiegyensúlyozás) megfelelő esetszétválasztással

A **beszúrás** hasonlóan történik, mint egy bináris keresőfában, azonban a beszúrás után sérülhet az AVL tulajdonság, tehát ki kell javítanunk azt. Ebben az esetben több eset is előállhat, melyet mind különböző módon, megfelelő sorrendben végre hajtott jobbra, illetve balra forgatásokkal hajthatunk végre.

Az egyes csúcsokra felírogatjuk, hogy mennyiben tér el a jobb és bal részfa magassága.

- -1: bal részfa magasabb
- 0: egyforma magasak
- +1: jobb részfa magasabb

A (++,+) szabály szerint egy balra forgatást hajtunk végre, melynek tükörképe a (--,-) szabály, melyben jobbra forgatást végzünk.

A (++,-) szabálynál először lent forgatunk jobbra, majd fent balra. Ennek megfelelően a (--,+)-nál alul forgatunk balra, majd felül jobbra.

(++,+)	
(--,-)	
(++,-)	
(--,-)	

Törlés esetén hasonlóan járunk el mint bináris keresőnél, majd a kiegyensúlyozás a fenti szabályok szerint történik, azonban ebben az esetben felléphet (++,0) és (--,0) szabályok is, melyek egy-egy balra, illetve jobbra forgatással megoldhatók.

(++,0)	
(--,0)	

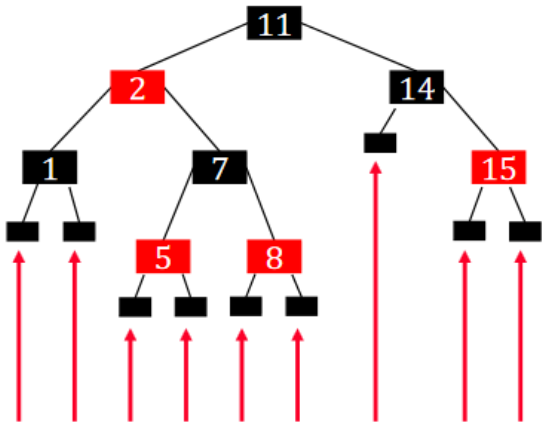
IX. tétel - Piros-fekete fák

Definíció, műveletek (**beszúrás utáni kiegyensúlyozás**, törlés utáni kiegyensúlyozás) algoritmusai, példán keresztül is

Definíció

A piros-fekete fa olyan bináris keresőfa, melynek minden pontja egy extra információt tartalmaz: a pont színe, mely lehet piros vagy fekete.

A piros fekete fában bármely, a gyökértől levélig vezető út hossza nem lehet nagyobb, mint a legrövidebb ilyen út hosszának a kétszerese. Ebből következik is, hogy az ilyen fák megközelítőleg kiegyensúlyozottak.



A szokásos bináris keresőfa minden csúcspontját, aminek kevesebb mint egy gyereke van kiegészítjük egy levéllel. Így a fa azon pontjai, melyek valódi értékeket tartalmaznak nem lesznek gyerekek.

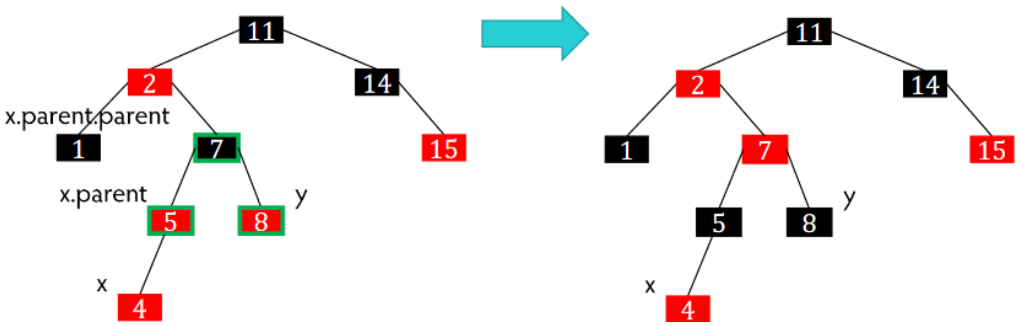
Minden csúcs piros vagy fekete, de a gyökér és minden levél fekete, és ha egy csúcs piros, akkor mindkét gyereke fekete. (nincs egymást követő két piros csúcs) Minden csúcsból minden út egy levélig ugyanannyi fekete tartalmaz.

Keresési idő a fában $\sigma(\log_2 n)$, ez mutatja a PF fák hatékonyságát.

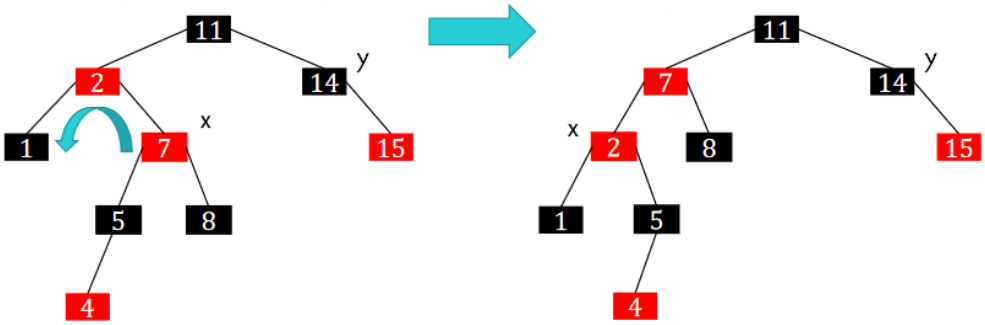
Műveletek (**beszúrás utáni kiegyensúlyozás**, törlés utáni kiegyensúlyozás) algoritmusai, példán keresztül is

A **beszúrás** nagyon hasonlóan történik, mint az AVL fáknál (alapból: piros). Ebben az esetben a forgatásokon kívül, átszínezést is kell alkalmazni a PF tulajdonságok helyreállításához.

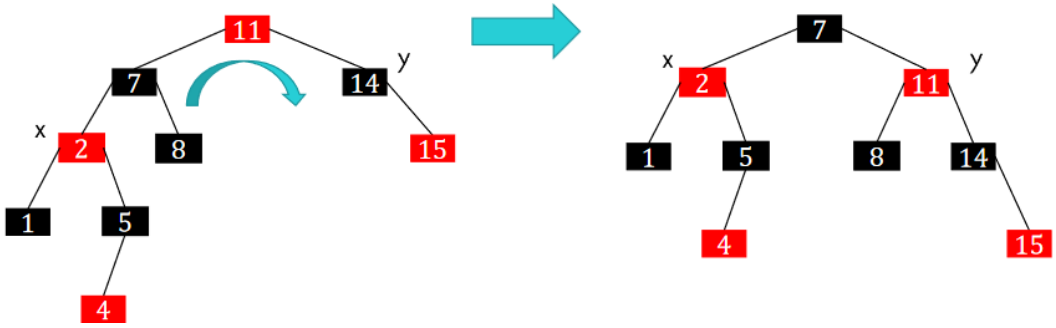
1. Ha a nagybácsi színe piros, cseréljük meg y, a nagyszülő és a szülő színét.



2. Ha az x nagybácsija fekete, az x szülője piros és a szülő ellenkező oldali gyereke a nagyszülőnek, mint x a szülőnek, akkor forgassunk x és szülője körül balra, legyen x az eddigi x szülője, majd forgassunk x és gyereke körül balra.



3. Ha az x nagybácsija fekete, az x és szülője is piros, és a szülő azonos oldali gyereke a nagyszülőnek, mint x a szülőnek, akkor kicserélem a színeket a szülő és a nagyszülő között, és forgatunk a szülő és a nagyszülő mentén jobbra.



A forgatásokat nagyon hasonlóan végezzük, mint AVL-fákon, azonban figyelnünk kell arra, hogy a fa szélén lévő levelek milyen értékeket tárolnak.

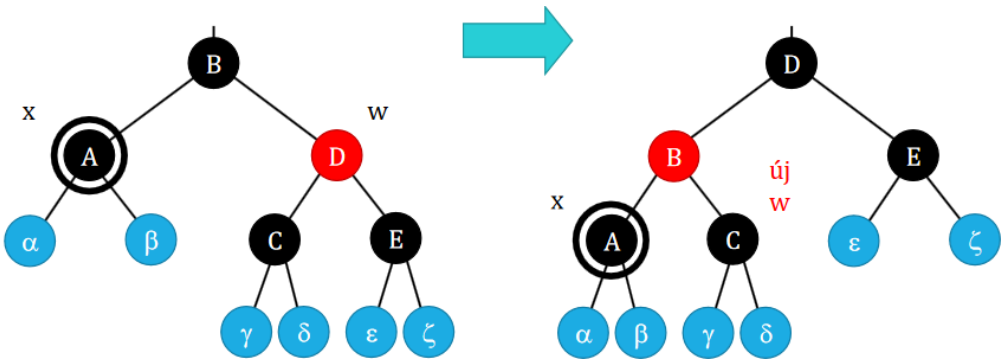
```
BALRA-FORGAT(T, x)
y ←jobb[x]; jobb[x] ← bal[y]
if bal[y]  ≠ NIL
  then szülő[bal[y]] ← x
szülő[y] ← szülő[x]
if szülő[x]= NIL
  then gyökér[T] ← y
  else if x=bal[szülő[x]]
         then bal[szülő[x]] ← y
         else jobb[szülő[x]] ← y
bal[y] ← x
szülő[x] ← y
```

Törlésnél könnyű dolgunk van, ebben az esetben ugyanis semmi dolgunk nincs, ha a csúcs piros volt, a PF tulajdonságok megmaradnak.

Tulajdonság és leírása	Igaz?
1. Minden csúcs piros vagy fekete	igen
2. A gyökér fekete	igen
3. Minden levél (nil[T]) fekete	igen
4. Ha egy csúcs piros, mindkét gyerek fekete	igen
5. Minden út egy csúcstól a leszármazott levelekig ugyanannyi fekete csúcsot tartalmaz	igen

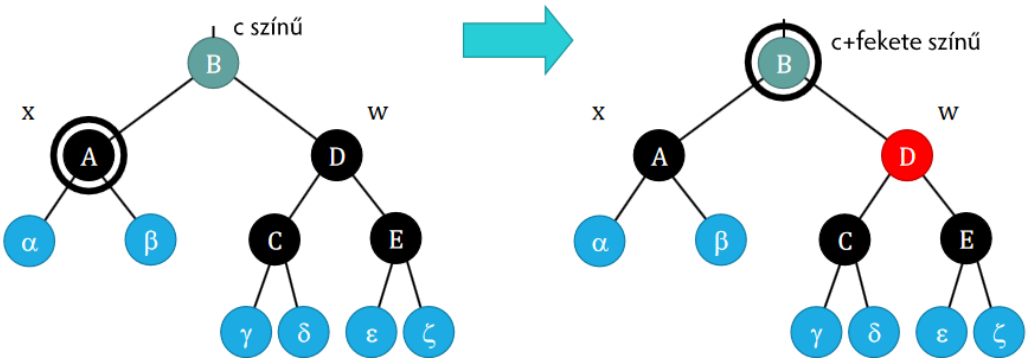
Ha a csúcs fekete volt, nehezebb dolgunk van, mert a 2.,4. és 5. tulajdonság sérülhet. Törlésnél nem a nagybácsit, hanem a testvért vizsgáljuk.

1. Ha a testvér piros, akkor változtassuk feketére, majd a szülőjük legyen piros, és hajtsunk végre egy balra forgatást a szülő körül. Majd a w-t állítsuk a szülő jobb gyerekére.



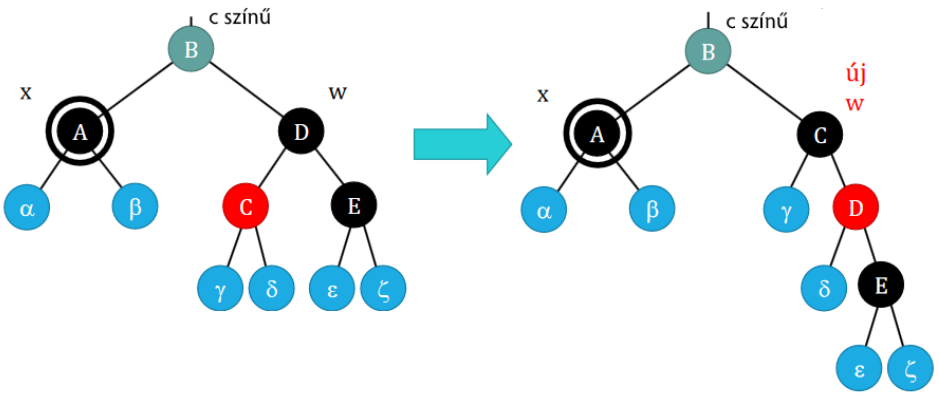

```
if szín[w]=PIROS
then szín[w]←FEKETE
  szín[szülő[x]]←PIROS
  BALRA-FORGAT (T, szülő[x])
  w←jobb[szülő[x]]
```

2. Ha a testvér (w) fekete, akkor nézzük a w gyerekeit: ha mindkettő fekete, akkor a w színét pirosra állítjuk, és az x-et átállítjuk x szülőre.



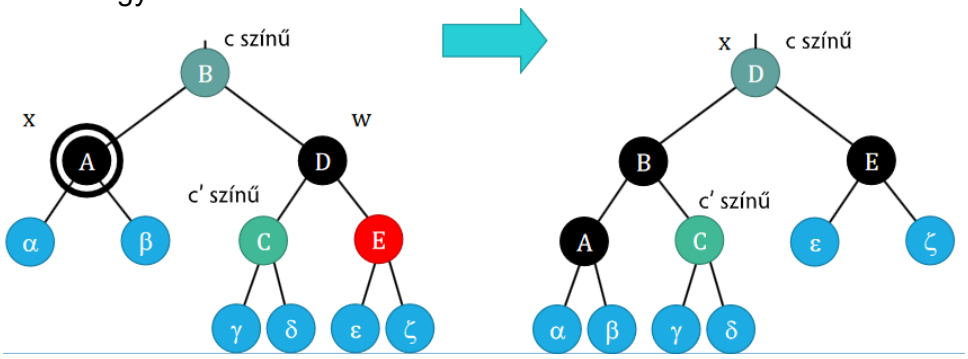

```
if szín[bal[w]]=FEKETE and
  szín[jobb[w]]=FEKETE
  then szín[w]←PIROS
     x←szülő[x]
```

3. Ha a w fekete, és bal fia piros, jobb fia fekete, akkor legyen a bal gyerek színe fekete, a w piros, és forgassunk jobbra w körül. Végül legyen a w a jobb szülője x-nek.




```
else if szín[jobb[w]]=FEKETE
     then szín[bal[w]]←FEKETE;
          szín[w]←PIROS,
          JOBBRA-FORGAT(T,w);
          w←jobb[szülő[x]]
```

4. Ha a w fekete, és jobb fia piros, akkor w színe, legyen a szülő színe, majd a szülő színe legyen fekete. A w jobb gyereke legyen fekete, majd forgassunk balra a szülő körül. Ekkor x lesz a gyökér.



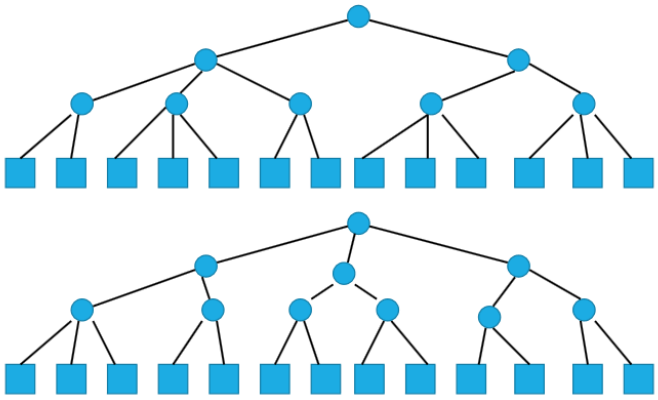

```
szín[w]←szín[szülő[x]]
szín[szülő[x]]←FEKETE;
szín[jobb[w]]←FEKETE
BALRA-FORGAT(T, szülő[x])
x ← gyökér[T]
```

X. tétel - 2-3 fák

Fogalma, műveletek (keresés 2-3 fában, **beszúrás 2-3 fába**, törlés 2-3 fából), műveletek költsége; B fák: definíciója, műveletek (keresés B fában, **beszúrás B fába**, törlés B fából)

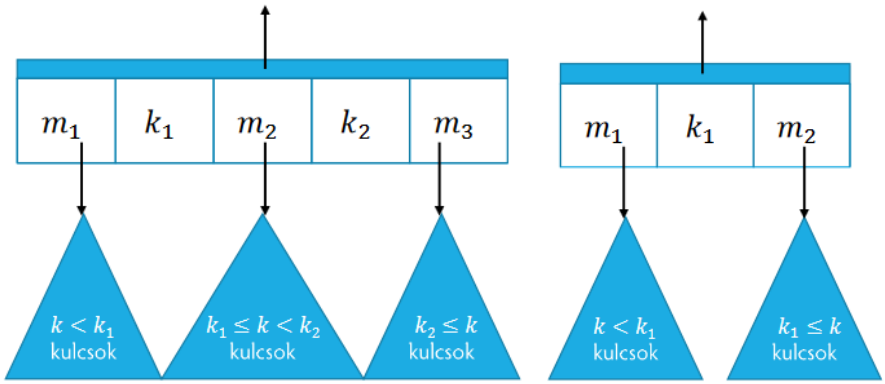
Fogalma

A 2-3 fa egy olyan speciális fa, melyben minden csúcsnak, ami nem levél pontosan kettő, vagy 3 gyereke van.



Adatszerkezetek szempontjából a 2-3 fa egy lefelé irányított gyökeres fa, melyre:

- A rekordok a fa leveleiben helyezkednek el a kulcs érték szerint balról jobbra növekvő sorrendben. Egy levél egy rekordot tartalmaz.
- Minden belső csúcsból 2 vagy 3 él megy lefelé, ennek megfelelően a belső csúcsok egy, illetve két kulcsot tartalmaznak.
- A belső pontokban a fizikai szerkezet ilyen

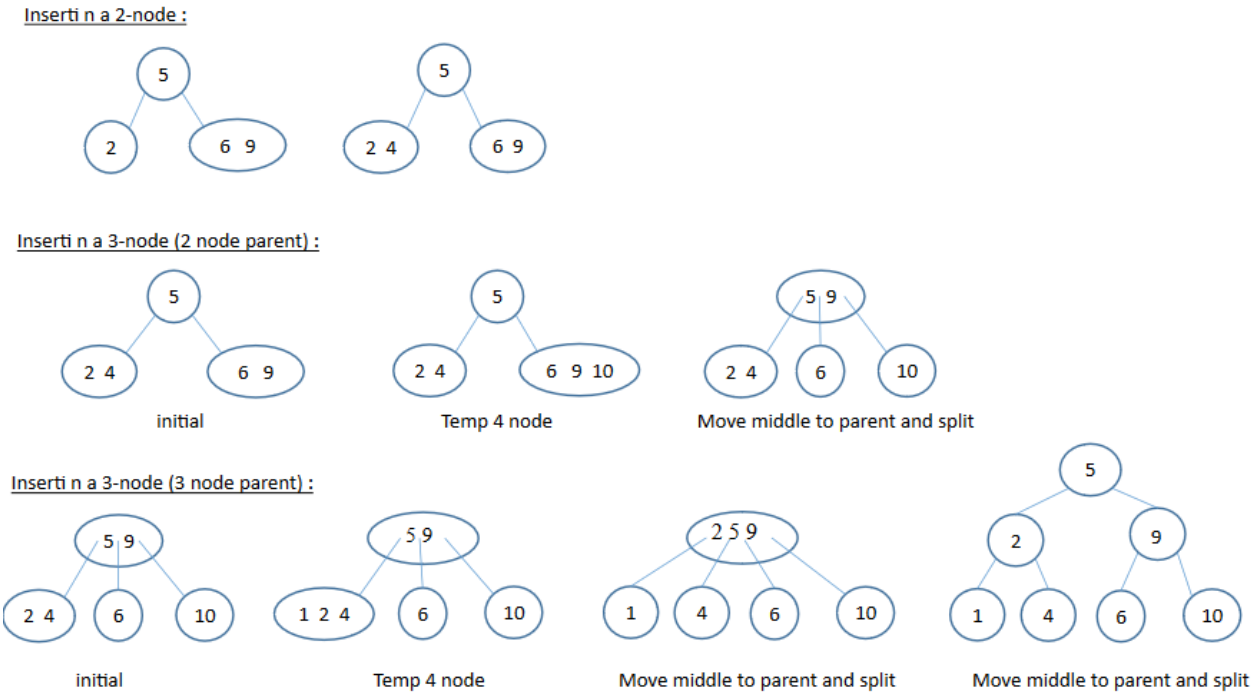


Egy belső csúcs logikailag lehet:

- 3 gyerekes:
  - m_1, m_2, m_3 mutatók a csúcs részfái, és $k_1 < k_2$ kulcsok.
  - m_1 által mutatott részfában minden kulcs kisebb mint k_1
  - m_2 által mutatott részfában minden k kulcs a k_1 és k_2 közé esik.
  - m_3 által mutatott részfában minden kulcs nagyobb vagy egyenlő mint k_2
- 2 gyerekes:
  - m_1, m_2 mutatók a csúcs részfái, és k_1 a kulcs.
  - m_1 által mutatott részfában minden kulcs kisebb mint k_1
  - m_3 által mutatott részfában k_1 a legkisebb kulcs.

Műveletek (keresés 2-3 fában, beszúrás 2-3 fába, törlés 2-3 fából), műveletek költsége

A **beszúrás** esetében a PF és az AVL fához hasonlóan több esetet különböztetünk meg. Megkeressük először a logikus helyet a fában, és odatesszük. Ha a csúcs ettől 4-csúcs lesz, akkor a középső elemet szülővé tesszük, melynek két gyereke lesz a másik kettő elem. A különböző eseteket az alábbi ábra kitűnően prezentálja.



Törlés esetén két esetet különböztetünk meg. Az egyik, ha a kulcs szülőjének 3 gyereke van, a másik, ha 2. Amennyiben 3 gyereke van töröljük a gyereket **EZT NEM TOM HIÁNY**

A műveletek költsége minden esetben $\sigma(\log_2 n)$.

B fák: definíciója

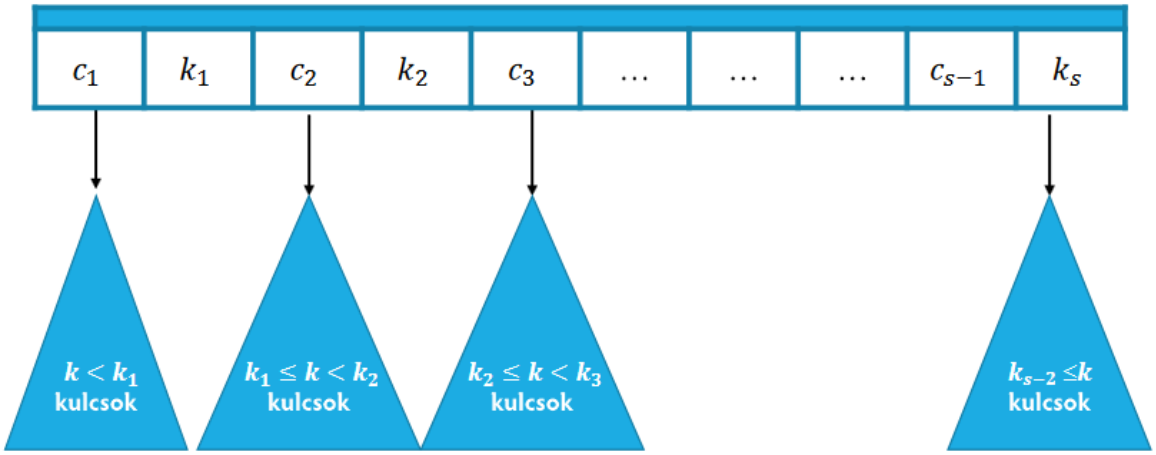
A B fa tulajdonképpen a 2-3 fa általánosítása. Nagy méretű adatbázisok, külső táron lévő adatok feldolgozására használják. Nem az összehasonlítás időigényes, hanem az adatok kiolvasása. Tegyük fel, hogy egy adott kulcshoz tartozó minden „kísérő információ" ugyanabban a csúcsban van, mint a kulcs. Minden kisírő információ a gyerekekben van.

A B-fa definíciója:

1. minden x csúcsnyak a következő mezői vannak:
   - $n[x]$ - az x csúcsban tárolt kulcsok darabszáma
   - A kulcsokat monoton növekvő sorrendben tároljuk ($x.kulcs_1 < \cdots < x.kulcs_{n[x]}$)
   - $x.level$, mely igaz, ha x levél, és hamis, ha belső csúcs
2. Ha x egy belső csúcs, akkor tartalmazza az $x.c_1, \ldots, x.c_{n[x]+1}$ mutatókat x gyerekeire.
3. A kulcsértékek meghatározzák azon kulcsérték tartományokat, melybe a részfák kulcsai esnek.
   $$k_1 \leq x.kulcs_1 < k_2 \leq x.kulcs_2 < \cdots \leq x.kulcs_{n[x]} < k_{n[x]+1}$$
4. Minden levélnek azonos a mélysége (a fa magassága)
5. A csúcsokban található kulcsok darabszámára adott egy alsó és felső korlát. Ezeket a korlátokat egy t 2-nél nagyobb egész számmal lehet kifejezni, ez a B-fa minimális fokszáma.

6. Minden nem gyökér csúcsnak, legalább $t-1$ a kulcsa, minden belső csúcsnak legalább t gyereke van, és ha a fa nem üres, akkor a gyökérnek legalább egy kulcsa van.
7. Minden csúcsnak $2t-1$ kulcsa van tehát maximum $2t$ gyerek lehet, és akkor telített, ha van $2t-1$ kulcsa.

A belső csúcsok logikailag így ábrázolhatók:



Műveletek (keresés B fában, beszúrás B fába, törlés B fából)

A háttértár műveletek modellezéséhez az alábbi definíciókat vezetjük be:

- Legyen x egy objektumra mutató pointer.
- Ha az objektum pillanatnyilag a központi memóriában van, akkor a mezőire a szokásos módon hivatkozunk ($x.kulcs$)
- Ha a mágneslemezen van, akkor először a $\text{LEMEZROL\_OLVAS}(x)$ műveletet használjuk, amely beolvassa az x -et a memóriába.
- Hasonlóan a $\text{LEMEZROL\_IR}(x)$ menti el a megváltozott mezőjű x objektumot a mágneslemezre.

A fában keresés algoritmusa:

```
B-FABAN-KERES(x,k)
  i←1
  while i≤n[x] és k>x.kulcs_i do
    i←i+1
    if i≤n[x] és k=x.kulcs_i
      then return (x,i)
    if x.levél
      then return NIL
    else LEMEZROL-OLVAS(x.c_i)
         return B-FABAN-KERES(x.c_i,k)
```

A $\text{PONTOT} - \text{ELHELYEZ}(x)$ eljárást használjuk fel:

```
B-FAT-LETREHOZ(T)
  x← PONTOT-ELHELYEZ()
  x.levél←IGAZ
  n[x]←0
  LEMEZRE-IR(x)
  gyökér[T]←x
```

Hasonlóan a 2-3 fához tudunk használni csúcs vágás műveletet. Ekkor a csúcs középső elem lesz a szülő melynek gyerekei a tőle balra eső, és a tőle jobbra eső elemek lesznek. Megadunk továbbá egy i intervallumot, hogy mekkora legyen a középső elem.

```
B-FA-VAGAS-GYEREK(x, i, y)
  z←PONTOT-ELHELYEZ()
    • O(1) idő alatt lefoglalja az új csúcsnak a lemez egy lapját
  z.levél← y.levél
  n[z]←t-1
  for j←1 to t-1 do
    • Átmásoljuk bele az y „végét"
    z.kulcs_j←y.kulcs_j+t
  if not y.levél
     then for j←1 to t do
             z.c_j←y.c_j+t
  n[y]←t-1
  for j←n[x]+1 downto i+1 do          a csúcsokra mutatókat arrébb
    x.c_j+1←x.c_j                     tesszük x-ben
  x.c_i+1←z                           z középre
  for j←n[x] downto i do              a kulcsokat arrébb tesszük
    x.kulcs_j+1←x.kulcs_j
  x.kulcs_i+1←y.kulcs_t               az y középső kulcsa a helyére
  n[x]←n[x]+1                         x elemszáma nőtt

  LEMEZRE-IR(y)
  LEMEZRE-IR(z)
  LEMEZRE-IR(x)
```

A fába beszúrás műveletét két részletben valósítjuk meg:

```
B-FABA-BESZUR(T,k)
  r←gyökér[T]
  if n[r]=2t-1
    • ha telített a gyökércsúcs, vág
    then s←PONTOT-ELHELYEZ()
      gyökér[T] ← s
      s.levél← HAMIS
      n[s] ← 0
      s.c1 ← r
      B-FA-VÁGÁS-GYEREK(s,1,r)
      NEM-TELITETT-B-FABA-BESZUR(s,k)
    else NEM-TELITETT-B-FABA-BESZUR(r,k)
```

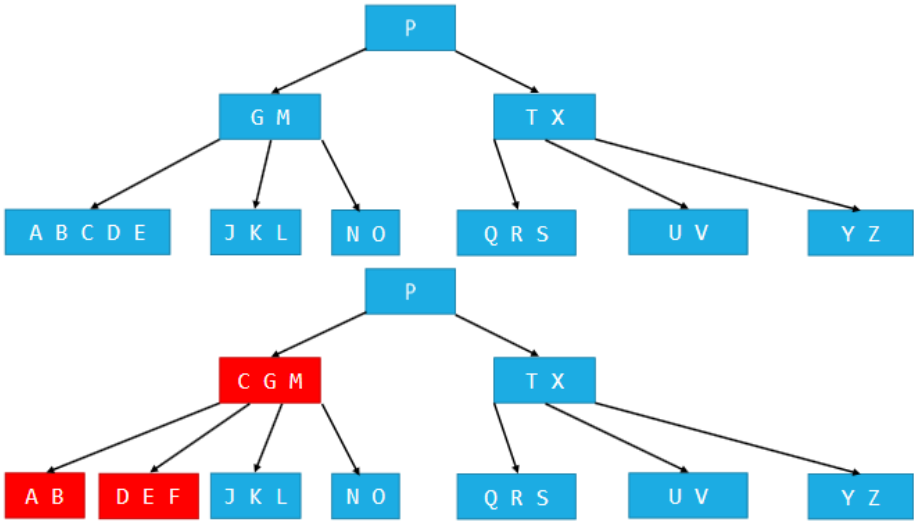
```
NEM-TELITETT-B-FABA-BESZUR(x,k)
  i←n[x]
  if x.levél
    then
       while i≥1 és k<x.kulcs_i do
         x.kulcs_{i+1}←x.kulcs_i
         i←i-1
       x.kulcs_{i+1}←k
       n[x]←n[x]+1
       LEMEZRE-IR(x)
    else
       while i≥1 és k<x.kulcs_i do
         i←i-1
       i←i+1
       LEMEZROL-OLVAS(x.c_i)
    if n[x.c_i]=2t-1
      then B-FA-VAGAS-GYEREK(x,i,x.c_i)
       if k>x.kulcs_i
         then i←i+1
    NEM-TELITETT-B-FABA-BESZUR(x.c_i,k)
```

hátulról kezdjük

itt a k helye
x mérete nő

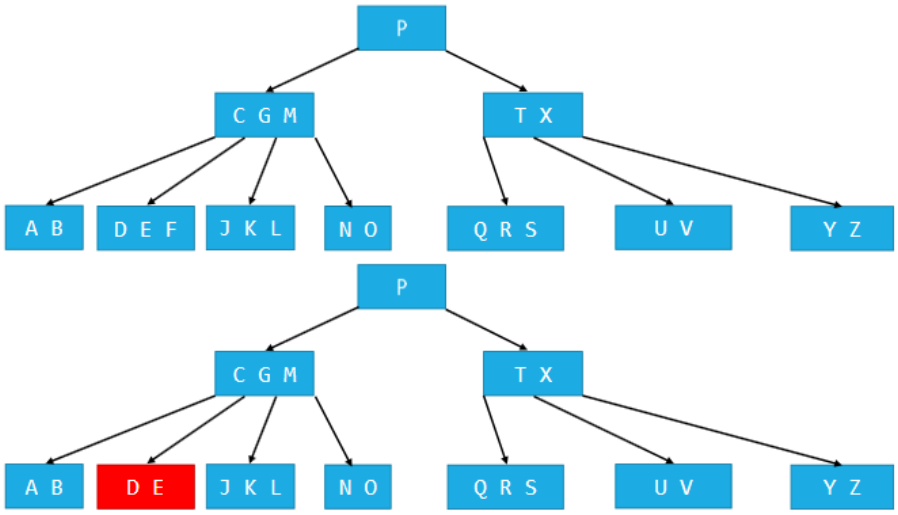
ha nem levél,
megkeressük a helyét

Telített?

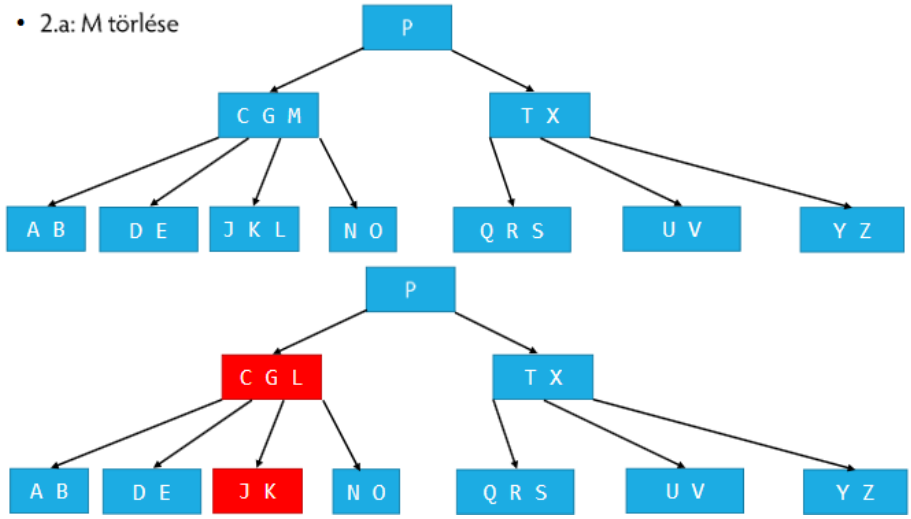


Törlés esetén a kulcsot nem csak a levélből, hanem tetszőleges csúcsból lehet törölni. Ügyelni kell arra, hogy a csúcs ne váljon túl kicsivé. A különböző lehetőségek:

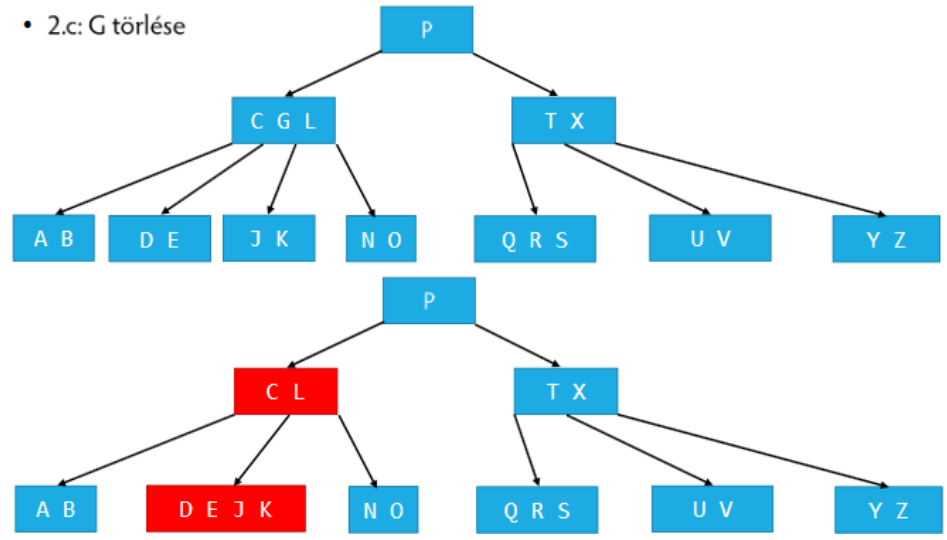
1. A k kulcs az x csúcsban van, x egy levél, akkor k kulcsot törlöm x-ből.



2. A k kulcs az x csúcsban van, x a fa egy belső csúcsa
    a. Ha x-ben a k-t megelőző gyereknek (y) legalább t kulcsa van, akkor megkeressük az y részfában a k-t közvetlenül megelőző k' kulcsot. Rekurzívan töröljük és helyettesítsük k-t k'-vel a az x-ben.



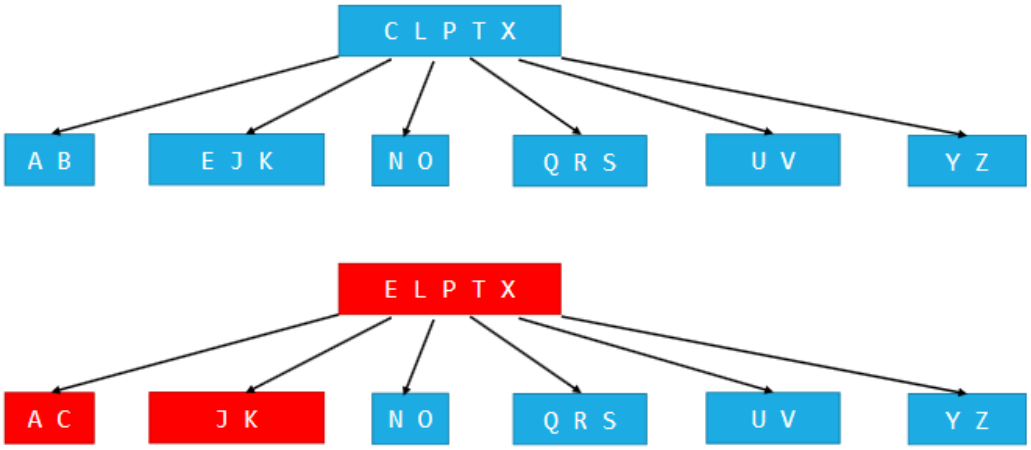
    b. Szimmetrikusan, ha a z gyerek következik az x-beli k után, és z-nek legalább t kulcsa van, akkor keressük meg a z gyökércsúcsú részfában a k-t közvetlenül követő k' csúcsot. Rekurzívan töröljük és helyettesítsük k-t k'-vel a az x-ben.
    c. Ha mind y-nak és z-nek csak $t-1$ kulcsa van, akkor egyesítsük k-t és z kulcsait y-ba úgy, hogy x-ből töröljük a k-t és a z-re mutató pointert. Ekkor az y-nak $2t-1$ kulcsa lesz. Ezurán szabadítsuk fel z-t és rekurzívan töröljük k-t az y-ból.



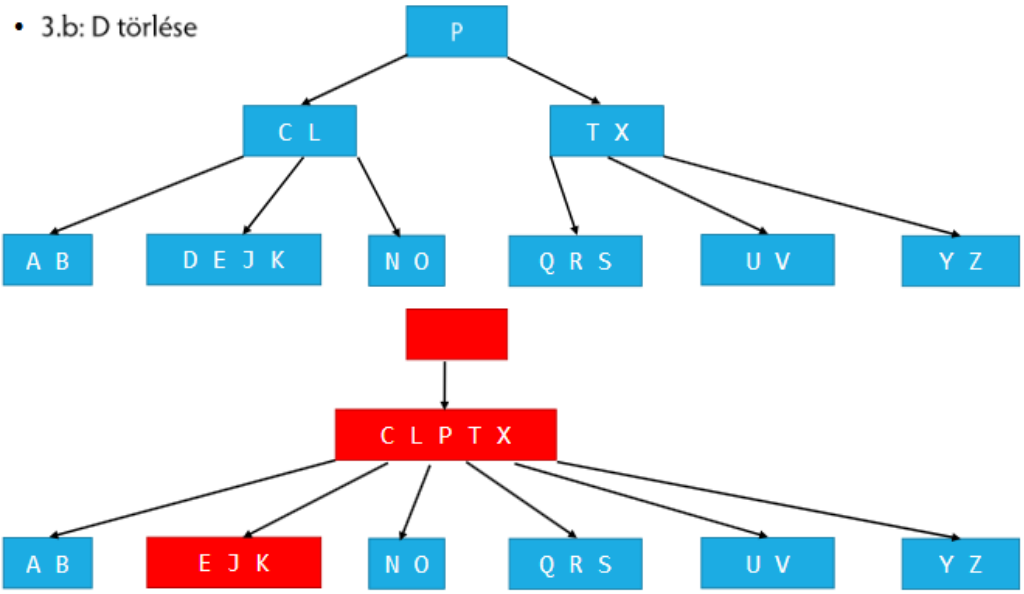
    d.
3. Ha a k kulcs nincs benne az x belső csúcsban, akkor határozzuk meg annak a részfának az $x.c_i$ gyökércsúcsát, amelyikben benne lehet a k, ha egyáltalán szerepel. Ha $x.c_i$-nek csak $t-1$ csúcsa van, akkor a 3a, vagy 3b szerint járunk el. Ezután rekurzívan megyünk tovább.
    a. Ha $x.c_i$ -nek $t-1$ csúcsa van, de van egy közvetlen testvére, amelynek legalább t csúcsa van, akkor vigyünk le $x.c_i$-be egy kulcsot x-ből, és az $x.c_i$ közvetlen bal vagy jobb testvérétől vigyünk fel egy kulcsot x-be, és vigyük át a megfelelő gyerek mutatóját a testvérétől $x.c_i$-be.

b. Ha $x.c_i$-nek és mindkét közvetlen testvérének $t-1$ kulcsa van, akkor egyesítsük $x.c_i$-t az egyik testvérével, majd vigyünk le egy kulcsot az x-ből ebbe az egyesített csúcsba, középre.

- 3.a: B törlése



- 3.b: D törlése



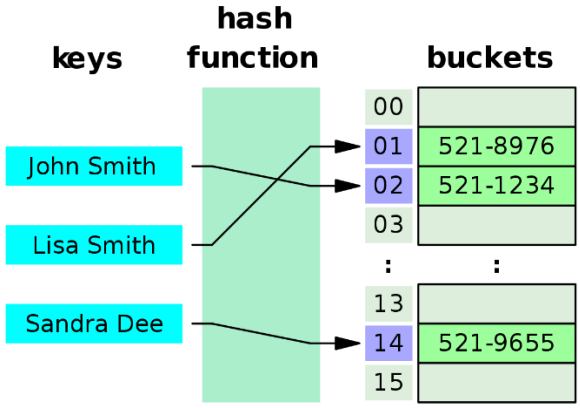
XI. tétel - Hasító táblák

Fogalma, közvetlen hozzáférésű táblák, hash függvények (egyszerű egyenletestől univerzálisig), kulcsütközések feloldása, **láncolt hashelés**, túlcsordulási terület, **nyílt címzés**

Fogalma közvetlen hozzáférésű táblák, hash függvények (egyszerű egyenletestől univerzálisig)

A **hash tábla** egy olyan adatszerkezet, amely egy **hash függvény** segítségével állapítja meg, hogy melyik kulcshoz milyen érték tartozik. A hash függvény segítségével a kulcsot leképezzük az adatokat tároló tömb egy adott indexére, ahol a keresett érték fellelhető.

- Adott K kulcsok halmaza. A K elemeivel azonosított rekordok száma, azonban várhatóan kisebb, mint a K számossága.
- Ekkor K-t egy alkalmas h függvénnyel leképezzük az ábrázolás alapját képző tartományra. Ezt hash függvénynek nevezzük. A függvény egyenletes, ha minden elem egyforma valószínűséggel képződik bármelyik értékre.
- Mivel h nem injektív ezért szükségszerűen fellép kulcsütközés, tehát van két olyan különböző kulcs, amire ugyanaz lesz a hash értéke.
- A feladat ennek a megoldása.

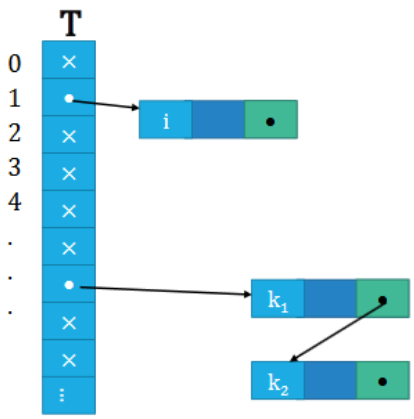


Azonban a kulcsok viselkedését vizsgálva néhány megszorítást kell tennünk:

- Kulcsok **egészek** kell legyenek
- Kulcsok **egyediek** kell legyenek
- Kulcsok **kis intervallumból** kerülhetnek ki
- Kulcsok **sűrűn** legyenek az intervallumban

Kulcsütközések feloldása, láncolt hashelés, túlcsordulási terület, nyílt címzés

A kulcsütközések kézenfekvő megoldása a **láncolt hashelés**. Ekkor minden táblaelemhez egy láncolt listát rendelünk.



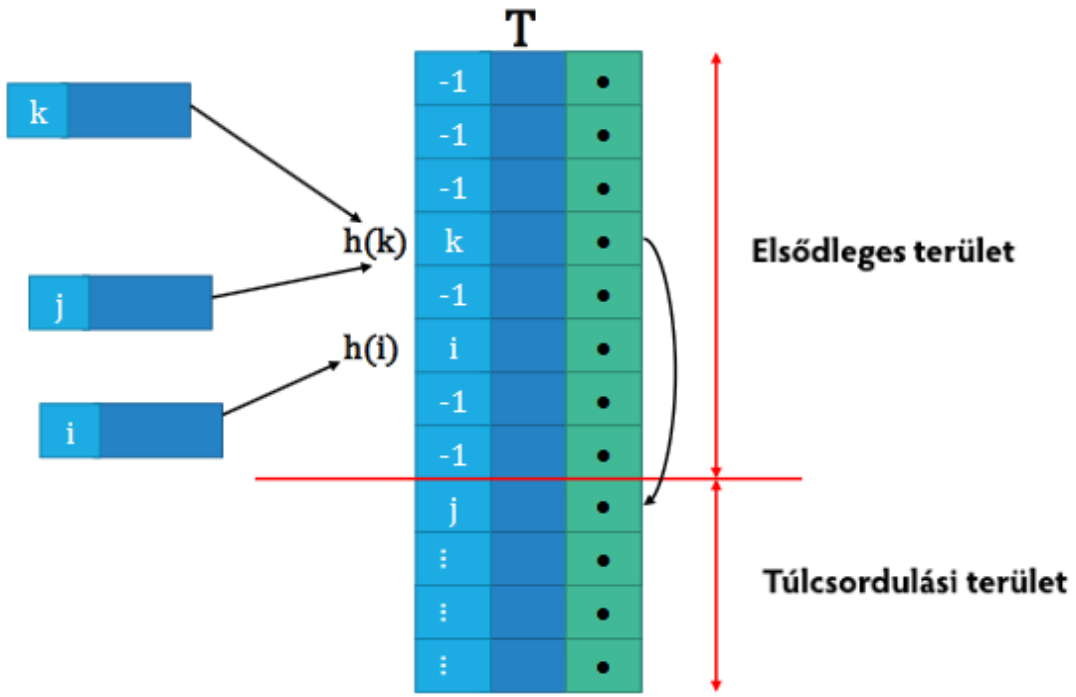
Ebben az esetben a keresés és a beszúrás értelemszerűen történik, úgy ahogy a láncolt listák esetén.

A táblánk hatékonyságának mérése érdekében bevezetjük a betöltési tényezőt, mely

$$\alpha = \frac{n}{m} = \frac{elemek\ a\ táblában}{hely\ a\ táblában}$$

A **láncolt hasítás** művelet igénye legrosszabb esetben $\sigma(n)$ lesz, hiszen ekkor minden egy listába kerül meg és abból lineárisan ki keresem az adott elemet.

A másik megoldás **a túlcsordulási terület** bevezetése. Ekkor a listát a tábla egy speciális területén hozzuk létre, amely az ún. túlcsordulási terület. Ekkor ha két ugyanolyan hashű elem kerül a táblába, akkor megkeressük az első szabad helyet a túlcsordulási területen, és beállítjuk az előző elem pointerét, hogy erre mutasson (elég index is).



A **nyílt címzésű** hash táblákban a láncolással ellentétben egy indexen kizárólag egy értéket tárolunk. Beszúráskor a megadott módszerrel kapott hashtől elindulva megkeressük az első szabad indexet, ahova az elem bekerül. Kereséskor ugyanígy járjuk be az egyes indexeket, ekkor O(n) lépésben megállapíthatjuk, hogy egy adott kulcs szerepel e a táblában.

Kereséskor végig járjuk a táblázatot, amíg meg nem találjuk az elemet, vagy egyértelműen meg nem tudjuk, hogy nincsen benne a táblában. Elem beszúrásánál kipróbáljuk az összes helyet, amíg üreset nem találunk (valamilyen stratégia szerint).

Törlésnél ilyenkor nem elég csak NIL-t írni, ekkor nem találná meg keresésnél az utána jövő elemeket a kereső. Ezért bevezetünk egy új szimbólumot a TÖRÖLT-et.

```
Hasító_töröl(T, k)
i←0
repeat j←h(k,i)
  if T[j]=k
    then T[j]←TÖRÖLT
      return
  i←i+1
until T[j]=NIL vagy i=m
```

```
Hasító_beszúr(T,k)
i←0
repeat j←h(k,i)
  if T[j]=NIL vagy T[j]=TÖRÖLT
    then T[j]←k
      return
    else i←i+1
until i=m
error „hasító táblázat túlcsordulás"
```

Az **univerzális hashelés** azt jelenti, hogy a hash-függvényt véletlenül, az aktuálisan tárolandó kulcstoktól függetlenül választjuk meg, ez jobb átlagos teljesítményhez vezet (mert gonoszan pakolnak elemeket). ekkor a hash függvényt egy gondosan megtervezett függvényosztályból futás közben választjuk ki.

Legyen $H: K \to [0, m)$ hasító függvények egy véges halmaza. A H-t univerzálisnak hívjuk, ha $\forall x \neq y \in K$-ra azon hash függvényeknek a száma, melyekre $h(x) = h(y)$ pontosan $\frac{|H|}{m}$.

XII. tétel - Kupacok és prioritásos sor

Definíció, reprezentáció, műveletek (algoritmusok!) (**beszúrás**, törlés, …); Prioritásos sor: ADT **axiomatikus specifikáció**, ADS, reprezentáció

Definíció

A majdnem teljes bináris fa **heap tulajdonságú**, ha üres, vagy a gyökérben lévő kulcs nagyobb, mint mindkét gyerekében, és mindkét részfája is heap tulajdonságú.

Egy bináris fa teljes, ha magassága h és $2^h - 1$ csomópontja van. Egy h magassgú bináris fa majdnem teljes, ha üres, vagy

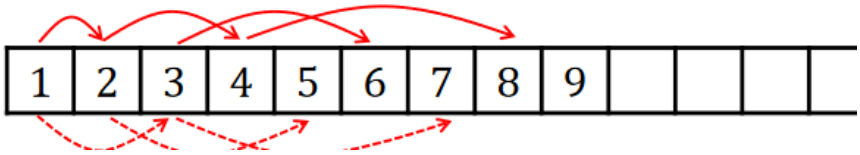
- A magassága h és bal részfája $h-1$ magas és majdnem teljes, a jobb részfája pedig $h-2$ magas és teljes, vagy
- A magassága h és bal részfája $h-1$ magas és teljes, a jobb részfája pedig szintén $h-1$ magas és majdnem teljes.

A majdnem teljes fákat „balról töltjük fel”.

A kupacokat használjuk prioritásos sorok megvalósítására, mivel a gyökérben lévő elem mindig maximális lesz. A törlések után csak helyre állítjuk a heap tulajdonságot. Ehhez a törölt gyökeret megcseréljük a kupac utolsó elemével, majd levélként töröljük. Ezután addig nyomjuk lefelé a gyökérből, amíg a helyére nem talál.

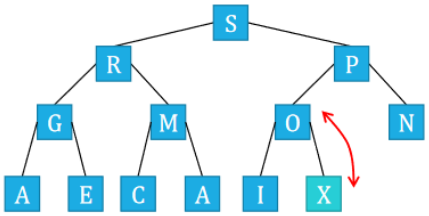
A kupacokat reprezentálhatjuk csomópontokkal és mutatókkal, ahogy eddig az összes fát, de inkább a tömbös reprezentációt preferáljuk. Ekkor kihasználjuk a majdnem teljes tulajdonságát:

- a k csomópont gyerekei a $2k, 2k+1$-nél vannak
- a k szülője a $k/2$-nél van
- ha $k > n$, akkor nincs ilyen csomópont



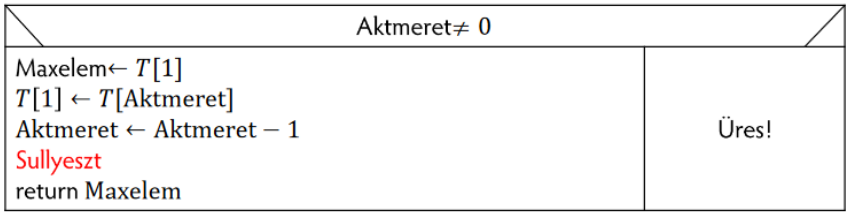
Műveletek (algoritmusok!) (**beszúrás**, törlés, …)

A kupachoz úgy adunk elemet, hogy először betesszük a következő üres helyre a jobb szélen, majd addig visszük fölfelé, amíg nagyobb nem lesz, mint a szülei.

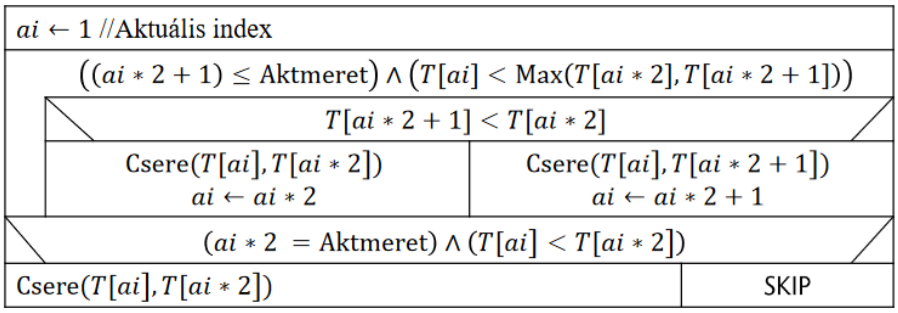


Aktmeret≠Maxmeret	
Aktmeret ← Aktmeret + 1 T[Aktmeret] ← ujelem Szulo ← Aktmeret / 2 Gyerek ← Aktmeret	Tele!
$(Szulo \geq 1) \wedge (T[\text{Szulo}] < T[\text{Gyerek}])$	
Csere(T[Szulo], T[Gyerek]) Gyerek ← Szulo Szulo ← Szulo/2	

A legnagyobb elem törlését egyszerűen implementálhatjuk:



Az ehhez használatos süllyesztő függvénynél feltételezzük, hogy a kupacban két jó rész fa van, de $T[1]$-t megfelelően le kell süllyeszteni:



Prioritásos sor: ADT axiomatikus specifikáció, ADS, reprezentáció

A prioritásos sor egy olyan sor, amelyben meghatározott sorrend szerint helyezkednek el az elemek (rendezett). Az alábbi függvények implementálása szükséges.

- `Empty` (az üres prioritásos sor konstruktor – létrehozás)
  - `Empty` (üres kupac)
- `isempty` (üres a prioritásos sor?)
  - `IsEmpty` (üres a kupac?)
- `insert` (elem betétele a prioritásos sorba)
  - `Insert` (elem betétele a kupacba)
- `delmax` (maximális elem kivétele a prioritásos sorból)
  - `DelMax` (maximális elem kivétele a kupacból)
- `max` (maximális elem lekérdezése a prioritásos sorból)
  - `Max` (maximális elem lekérdezése a kupacból)

Látható, hogy a kupacokat könnyen használhatjuk rendezésre. Beleszúrjuk az összes elemet a kupacba, majd addig pakolunk ki belőle, míg ki nem ürül, és a ki vett elemeket egy listába tesszük.

Az így kapott rendező elég hatékony, összesen $\sigma(n\log(n))$ sebességű.

Az ADT axiomatikus leírása a prioritásos sor adatszerkezetének (E alaptípus feletti P elsőbbségi sor)

Műveletek	• empty • isempty • insert • delmax • max	*Üres priorsor létrehozása* *Üres a priorsor?* *Elem betétele a priorsorba* *Maximális elem kivétele a priorsorból* *Maximlis elem lekérdezése*
Megszorítások	$D_{delmax,max} = P \setminus \emptyset$	*üres priorsoron nincs értelmezve*

Axiómák	• $\text{isempty}(\text{empty})$ • $\text{isempty}(p) \rightarrow p = \text{empty}$ • $\neg\text{isempty}(\text{insert}(p, n))$ • $\text{insert}(\text{delmax}(\text{p})) = p$

Illetve az ADS leírása:

- Rndezetlen tömb, beérkezési idő szerint
  - műveletigény egy maxker, vagyis $\sigma(n)$
- Rendezett tömbbel
  - insert műveletigénye (keresés: $\sigma(\log_2 n)$ jobbra lépés: $\sigma(n)$, összesen $\sigma(n)$.
- Kupaccal
  - fentiek szerint.

XIII.tétel - A rendezés feladata

Definíciója; rendezők osztályozása; Négyzetes rendezés algoritmusa és időigénye (**buborék rendezés**, beszúró rendezés, **maximum kiválasztásos** rendezés)

Definíciója, rendezők osztályzása

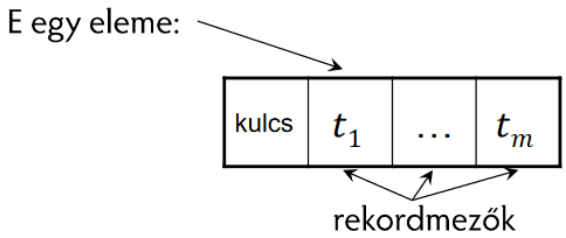
A rendezés során bemenetként egy n számot tartalmazó $(a_1, \dots, a_n)$ sorozatot kapunk, és kimenetkén egy olyan sorozatot adunk vissza, mely ezen elemek permutációja és $a_1' \leq \cdots \leq a_n'$.

Adott U halmazon egy $<$ kétváltozós reláció. Ha $a < b \in U$ akkor azt mondjuk, hogy a kisebb, mint b. Ez a reláció egy rendezés, ha

- irreflexív: $\neg a < a$
- tranzitív: $a < b \ \wedge b < c \rightarrow a < c$
- teljes: $a \neq b$ esetén $a < b$ vagy $b < a$

Egy ilyen $(U; <)$ párt rendezett halmaznak nevezünk. (Pl. egész számokon a <)

Általánosan egy „elem” több rekordból áll. Legyen K egy teljesen rendezett halmaz, a kulcsok halmaza, és T_i a tetszőleges típusok halmaza.



Ezt a kulcs szerint rendezzük: $S_i.kulcs \leq S_{i+1}.kulcs \ \forall i$

Tekintsünk egy általános példát! Ekkor bármelyik mezőt tekinthetjük kulcsnak, mindegyiken tudjuk értelmezni a rendezést. Például, ha az év a kulcs:

Abigél	Janka	Zsuzsi	Dávid	Dorka
167	164	158	160	162
1996	1998	2001	2000	2002

Abigél	Janka	Dávid	Zsuzsi	Dorka
167	164	160	158	162
1996	1998	2000	2001	2002

A rendezőket osztályozhatjuk számos aspektus szerint. A legfontosabbak:

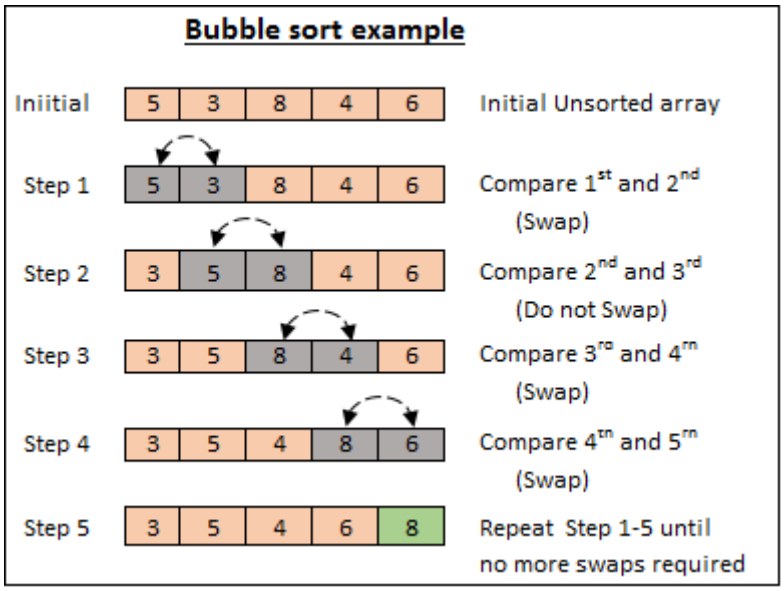
- Skalárrendező: $m = 0$
  Rekordrendező: $m \geq 1$
- Belső rendezők: központi memória + indexelés
  Külső rendezők: háttértárolón
- Összehasonlításos rendezők: kulcsok értékét hasonlítjuk
  Edényrendezők: kulcsok értéke szerint szétrakjuk
- Helyben rendezők: segéd memória konstans
  Nem helyben rendezők
- Stabil rendezők: azonos kulcsú rekordok sorrendje nemváltozik)
  Nem stabil rendezők

- (Használt módszer szerint)
  Lineáris adatszerkezet
  Fa
- (Módszer szerint)
  Max kiválasztó
  Csere rendezők
  Egy elemet helyre vivők
  Összefuttatásos rendezők

Hatékonyságuk vizsgálatakor leginkább a lépésszám számít.

Négyzetes rendezés algoritmusa és időigénye (buborék rendezés, beszúró rendezés, maximum kiválasztásos rendezés)

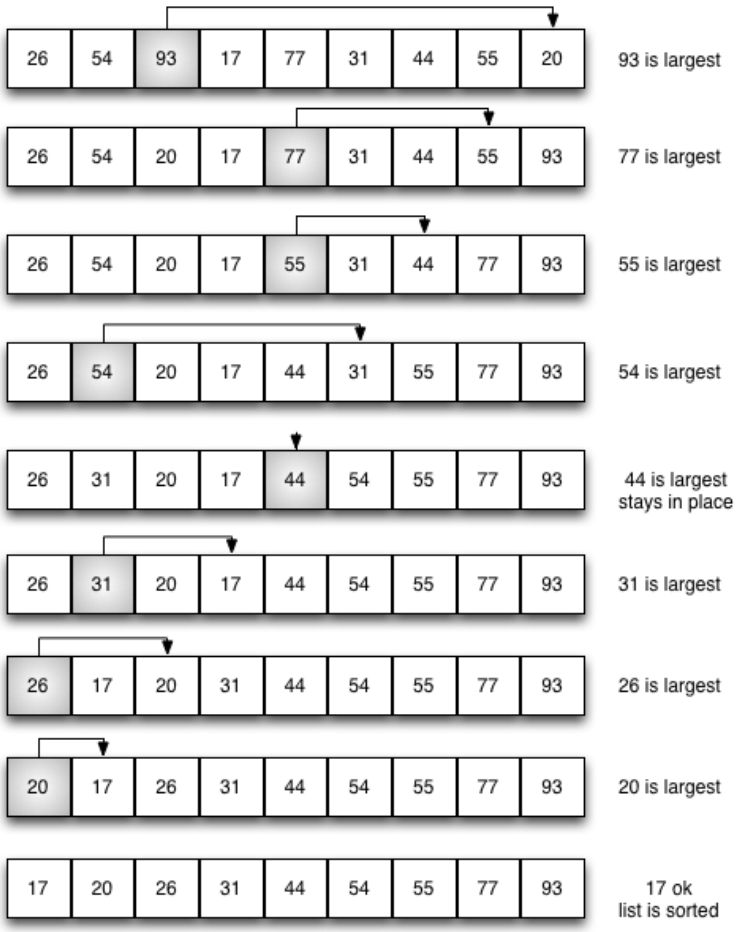
A **buborék rendezés** esetén az egymás mellet lévő elemeket hasonlítom összes sorban, majd, ha a baloldali a nagyobb, megcserélem őket. Így egy ciklus végén a legnagyobb elem a helyére kerül. Az algoritmus a nevét a buborékokról kapta, ami az elemekhez hasonlóan „felszáll” a pohár tetejére. Mivel egyáltalán nem hatékony szinte sehol se használják.




```
CIKLUS i = n TŐL 2 IG {
    CIKLUS j = 1 TŐL i-1 IG {
        HA TOMB[j] > TOMB[j+1] AKKOR {
            CSERÉLD FEL ŐKET: TOMB[j], TOMB[j+1]
        }
    }
}
```

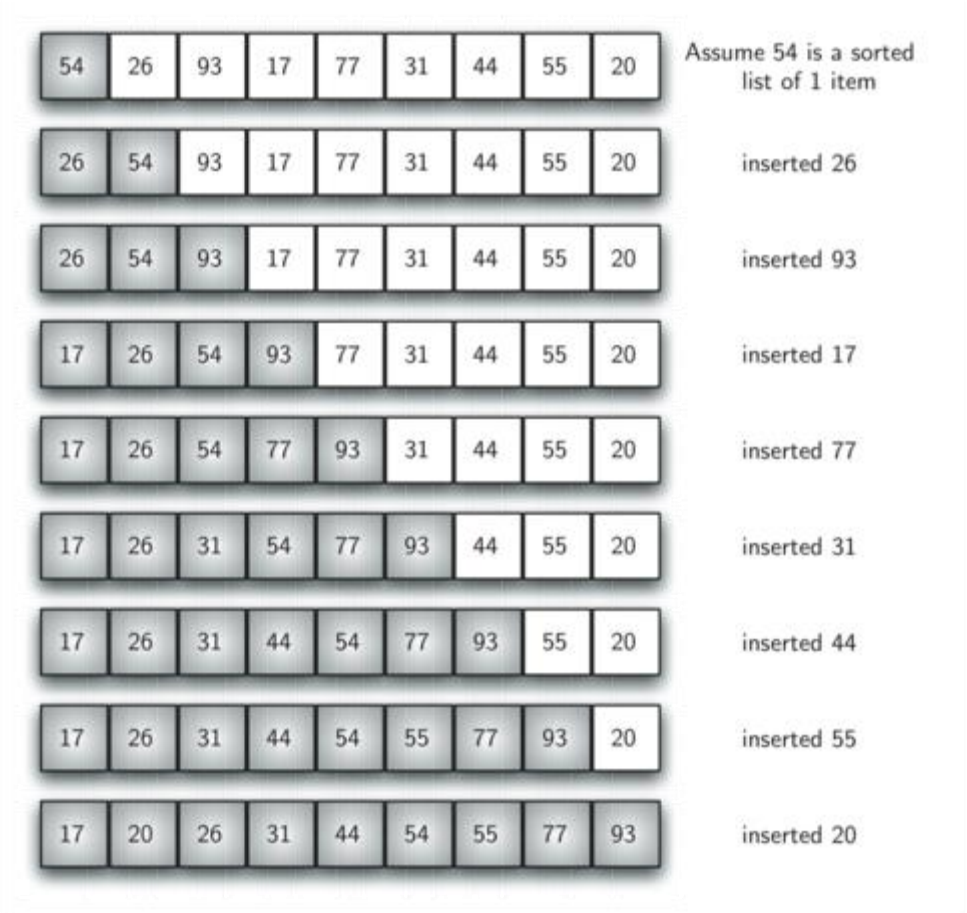
```
11  /// ========== BUBURÉKRENDEZÉS ==========
12  void bubbleSort(int* A, int n) {
13
14      for (int j = n - 1; j >= 1; --j) {
15          for (int i = 0; i <= j - 1; ++i) {  //Végigjárja előről hátra a tömböt
16              if (A[i] <= A[i + 1]) {         //Ha az aktuális elem kisebb, mint a következő
17                  //skip                      // akkor nem történik semmi
18              } else {                        //Ha az aktuális elem nagyobb, mint a következő
19                  swap(A[i], A[i + 1]);       // akkor megcseréljük a két elemet
20              }
21          }
22      }
```

A **maximum kiválasztásos** rendezésnél hátulról indulunk és mindig megkeressük a hátralévő tömbben lévő legnagyobb elemet, majd beillesztjük a helyére egy cserével.




```
/// ========== MAXIMUM KIVÁLASZTÁSOS RENDEZÉS ==========
// A rendező Segédfüggvénye: megkeresi a maximális értékű elemet a megadott tartományban (j-ig)
int MaxSel(const int* A, int j) {

    int ind = 0;                //A maximális értékű elem indexe
    for (int i = 0; i < j; ++i) {
        if (A[i + 1] > A[ind]) {
            ind = i + 1;
        }
    }
    return ind;                 //Visszatér a maximális értékű elem indexével

}

void maxSort(int* A, int n) {

    for (int j = n - 1; j >= 1; --j) {  //Végigjárja hátulról előrefelé a tömböt

        int ind = MaxSel(A, j);         //Megkeresi a maximumot az elejétől j-ig
        swap(A[ind], A[j]);             //A megtalált maximális elemet kicseréljük az aktuális hátsó elemmel
    }

}
```

A **beszúró rendezés** az, amit például kártyáknál csinálunk mikor húzunk egy új lapot és berakjuk a kezünkbe. Megkeressük a számára a megfelelő helyet és oda beillesztjük. Ehhez a tömb elejétől indulunk majd mindig visszafele. Megkeressük az adott elem után jövő első elemet, aminél ő nagyobb, majd azt betesszük mögé.




```
50  /// ========== BESZÚRÓ RENDEZÉS ==========
51  void insertionSort(int* A, int n) {
52
53      for (int j = 0; j <= n - 2; ++j) {  //Az 0. indexű elem már rendezett,
54          int w = A[j + 1];               //az 1. indexű (j+1) elemtől indulunk, amit eltárolur
55          int i = j;
56          while (i >= 0 && A[i] > w)      //Megkeressük a 'w' helyét a rendezett szakaszban
57          {
58              A[i + 1] = A[i];            //Jobbra toljuk a 'w'-nél nagyobb elemeket
59              i = i - 1;
60          }
61          A[i + 1] = w;                   //Beszúrjuk a helyére az elmentett értéket
62      }
63
64  }
```

XIV. tétel - Quicksort

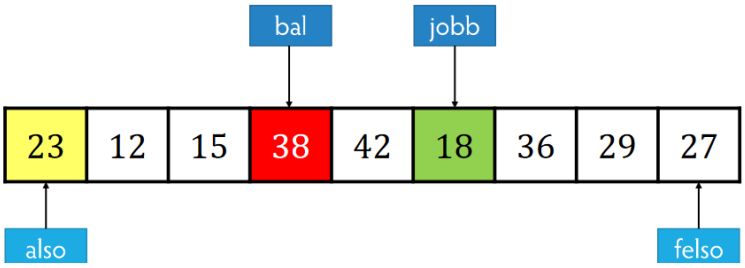
Algoritmusa, műveletigénye, pivot választási stratégia jelentősége

Algoritmusa

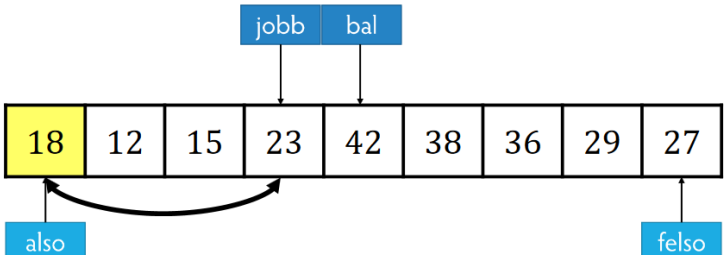
A gyorsrendezés egy rendkívül hatékony rendezési algoritmus. Alapja, hogy a rendezendő elemek sorozatát két részre osztom, majd külön-külön gyorsrendezem őket. Az algoritmus során választunk egy pivotot (strázsa), majd olyan pozícióra hozzuk, hogy a tőle jobbra lévő elemek nála nagyobbak, a tőle balra lévők kisebbek legyenek. Maga a keresés implementációja rendkívül egyszerű.

```
quicksort( void *a, int also, int felso )
{
  int pivot;
  /* Terminálási feltétel! */
  if ( felso > also )
  {
    pivot = feloszt( a, also, felso );   // Oszd meg!
    quicksort( a, also, pivot-1 );       // Uralkodj!
    quicksort( a, pivot+1, felso );      // Uralkodj!
  }
}
```

Ezt követően a felosztó függvényt kell implementálni. Ennek lényege az lesz, hogy a tömb két széléről megindítunk egy indexelést ez lesz a jobb és a bal. A két indexet elindítjuk egymással szemben. A bal jelzőt addig visszük jobbra, míg nem lesz igaz, hogy az általa hivatkozott elem nagyobb vagy egyenlő, mint a pivot, míg a jobb jelzőt addig visszük balra, míg az általa mutatott elem kisebb vagy egyenlő a pivottal.

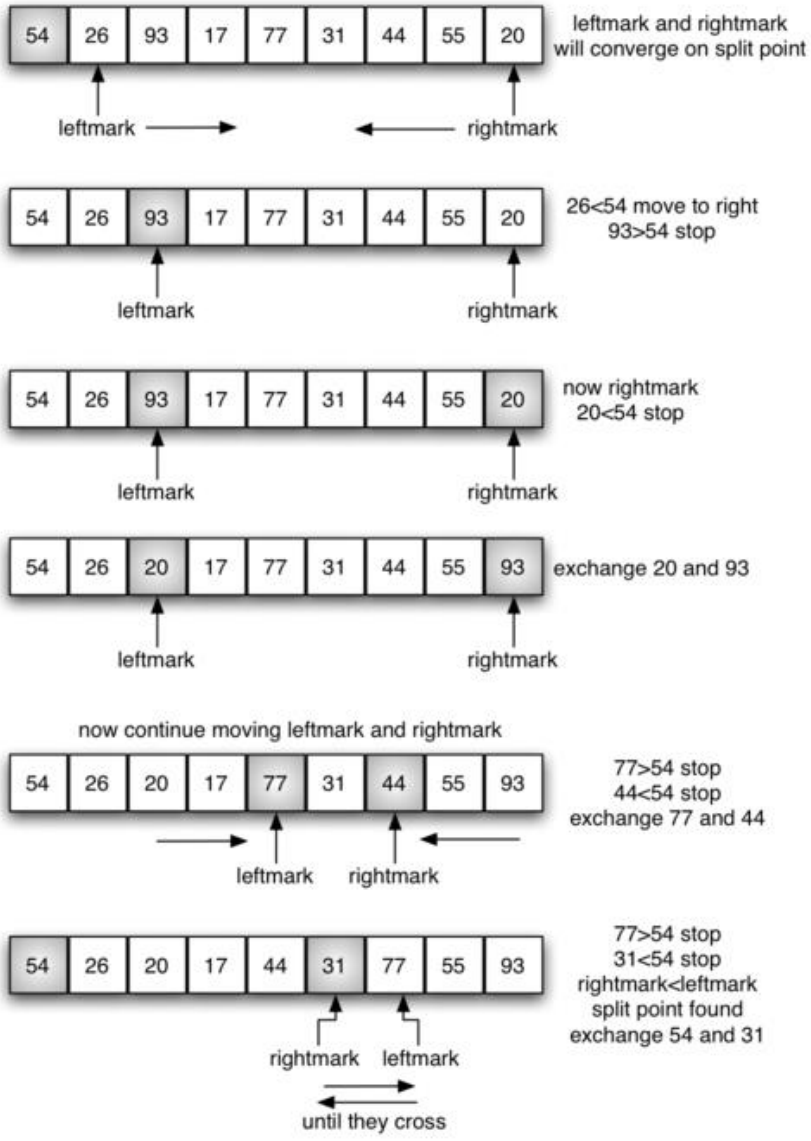


Amikor a két index nem találkozik, akkor meg kell cserélni a két elemet, mivel a strázsa rossz oldalain állnak. Ezután folytatjuk a balra jobbra lépkedést, majd mikor szembe találkoznak leállítjuk, majd ebben a lépésben megcseréljük a pivotot az aktuális „jobb” elemmel.



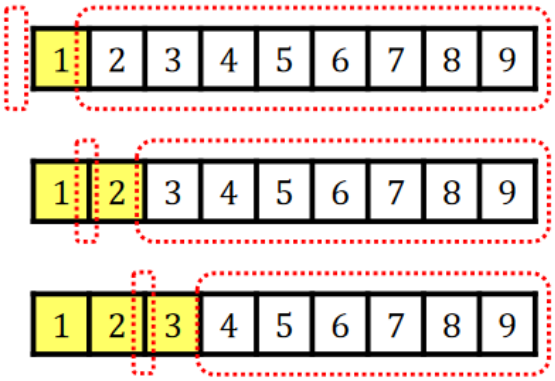
Ezután a felosztó eljárás visszatér a jobb indexével, és meghívjuk a quicksortot a strázsától jobbra, illetve a balra lévő elemekre is.

```
13  //A gyorsrendező felosztó függvénye
14  int Divide(int* A, int down, int up) {
15
16      int pivot = A[down];        //pivot választás
17      int left = down;
18      int right = up;
19
20      while (left < right) {
21          while (A[left] <= pivot && left < up) {     //Addig lépegetek előre a baloldalon,
22              left = left + 1;                        //amíg a pivotnál kisebb elemeket találok
23          }
24          while (A[right] >= pivot && right > down) { //Addig lépegetek hátra a jobboldalon,
25              right = right - 1;                      //amíg a pivotnál nagyobb elemeket találok
26          }
27          if (left < right) {
28              swapint(A[left], A[right]);
29          }
30      }
31      A[down] = A[right];
32      A[right] = pivot;
33      return right;
34
35  }
```



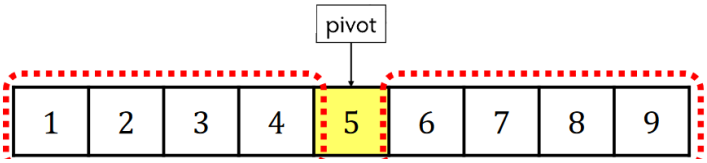
Művelet igénye, pivot választási stratégia jelentősége

A felosztás minden elemet egyszer vizsgál, tehát $\sigma(n)$ komplexitású, az „uralkodás” pedig $\sigma(\log_2 n)$ idő alatt történik, tehát összesen $\sigma(n \log n)$ idővel számolhatunk. A baj az, hogy ha az elemek rendezettek, akkor minden partícionálás során létrejön egy 0 és egy n-1 méretű feladat.



Ekkor n időigénye $\sigma(n)$, tehát az egészé $\sigma(n) * n = \sigma(n^2)$, magyarul nem gyorsabb a négyzetes rendezőknél, például a buborék algoritmusnál.

Erre megoldás lehet, hogy okosabban választunk strázsát, mert ha a partíciók egyformák, akkor $\sigma(\mathrm{n} * \log n)$ a rendezés ideje. Ha a középsőt választjuk pivotnak, ez rendezett elemeken jó választás lehet.



Másik remek megoldás, ha három strázsát választunk majd azok közül az érték szerinti középsőt választjuk. Például vehetjük a két szélső és a középső indexet.

Mivel a rendezett és majdnem rendezett adatok igen gyakoriak, ezért ez egy gyakran használt stratégia.

Lehet a pivotot véletlenszerűen is választani, azonban azt mindig figyelembe kell vennünk, hogy a pivot kiválasztása $\sigma(1)$ idő kell legyen.

Sajnos a logaritmusos sebesség sohasem garantálható, azonban a pivot választási stratégia javíthat ezen. A legtöbb esetben a quicksort lényegesen gyorsabb, mint a négyzetes társai, azonban azok is hatékonyak lehetnek, ha az egyszerűbb kód a cél, és kevés adatunk van. (rendszer függő).

A felosztás művelet másik implementációjában a felső elemet vesszük indexnek, és egy i és egy j index-szel indulunk alulról. Az i 0-tól a j 1-től. Egészen addig megyünk a j-vel,míg nem találunk a strázsánál egy kisebb elemet, ekkor növeljük az i-t, és megcseréljük az i-t és a j-t mutató elemet, majd folytatjuk a j növelését. Ha elérkezett a j a sor végére, akkor a pivotot megcserélem i+1-gyel.

XV. tétel - Kupac

Kupac **definíció**, reprezentáció, műveletek algoritmusai (**beszúrás**, törlés, ...), és a **rendezési algoritmus a kupaccal**, műveletigénye

Már volt

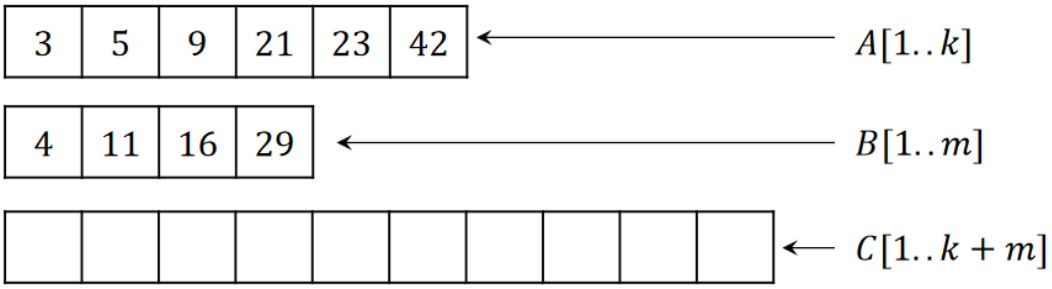
A XII. tételben szerepel.

XVI. tétel - Összefuttatásos rendezés

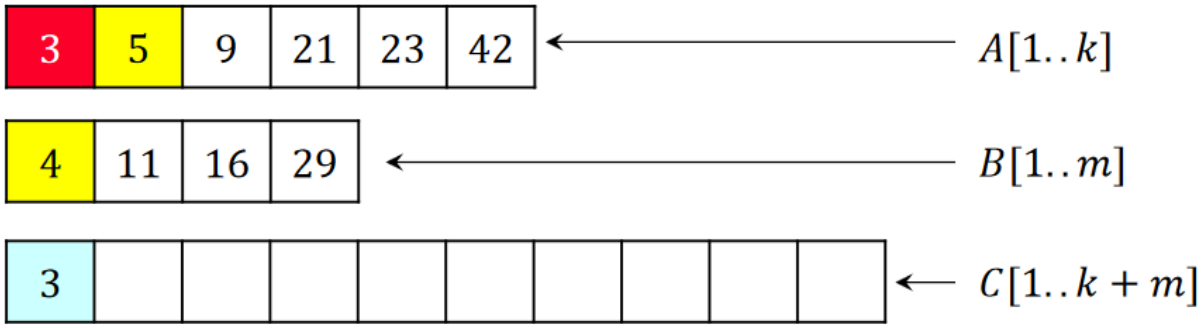
Algoritmusa, műveletigénye; **az összehasonlításos rendezés minimális összehasonlítás száma a legrosszabb esetben** (bizonyítás döntési fával); Batcher-féle páros-páratlan összefésüléses rendezés - **algoritmus szövegesen**, helyességének belátása

Algoritmusa

A feladatunk két rendezett sorozat összefuttatása. Legyen ez $A[1 \dots k]$ és $B[1 \dots m]$, ekkor tekinthetjük példának, hogy



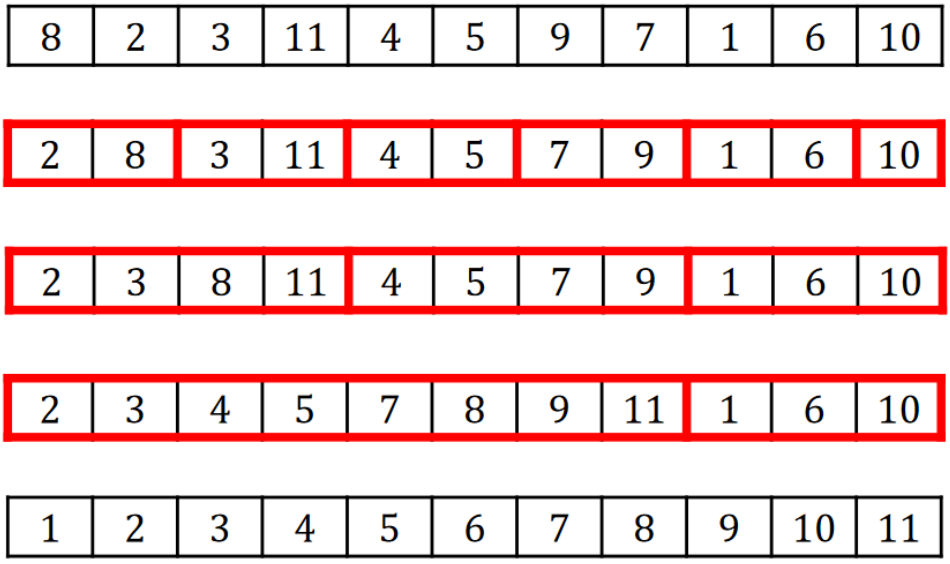
Ezt úgy csinálom, hogy egyesével elindulok mindkét tömbön elemenként, mondjuk A-n i-vel, B-n j-vel. Összehasonlítom A[i]-t B[j]-vel, majd amelyik kisebb azt beteszem C-be, és azon a tömbön növelem az indexet:




```
7   /// Osszefesul ket rendezett sorozatot.
8   void mergeArrays(const int vec1[], size_t n1, const int vec2[], size_t n2,
9           int result[]) {
10      unsigned int it1 = 0, it2 = 0, it3 = 0;
11      while (it1 < n1 && it2 < n2) {
12          if (vec1[it1] < vec2[it2])
13              result[it3++] = vec1[it1++];
14          else if (vec1[it1] > vec2[it2])
15              result[it3++] = vec2[it2++];
16          else {
17              result[it3++] = vec1[it1++];
18              result[it3++] = vec2[it2++];
19          }
20      }
21      /// Itt meg valamelyik tombben lehetnek tagok.
22      while (it1 < n1)
23          result[it3++] = vec1[it1++];
24      while (it2 < n2)
25          result[it3++] = vec2[it2++];
26  }
```

A *MergeSort* lényege, hogy egy adott sorozatot rendezünk az összefésülés eljárása segítségével rekurzívan. Szétvágjuk fele felére, mindaddig amíg 1 elemű részhalmazokat nem kapunk. Ezeket a kis sorozatokat pedig párosával összefésüljük.

Length$(S) > 1$	
Szétvág(S, S_1, S_2)	SKIP
MergeSort(S_1) MergeSort(S_2)	
Összefuttat(S_1, S_2, S)	



Tömböknél másik hasznos stratégia, hogy végig megyünk rajta, egy egyre növekvő $1,2,2^2,\dots,2^{\log_2 n-1}$ lépésközzel, és az ilyen hosszú szomszédos tömbrészleteket fésüljük össze, egy segédtömböt használva.

Az összehasonlításos rendezés minimális összehasonlítás száma a legrosszabb esetben (bizonyítás döntési fával)

Az összehasonlítások számát akarjuk megbecsülni a legrosszabb esetben. Azt tudjuk, hogy

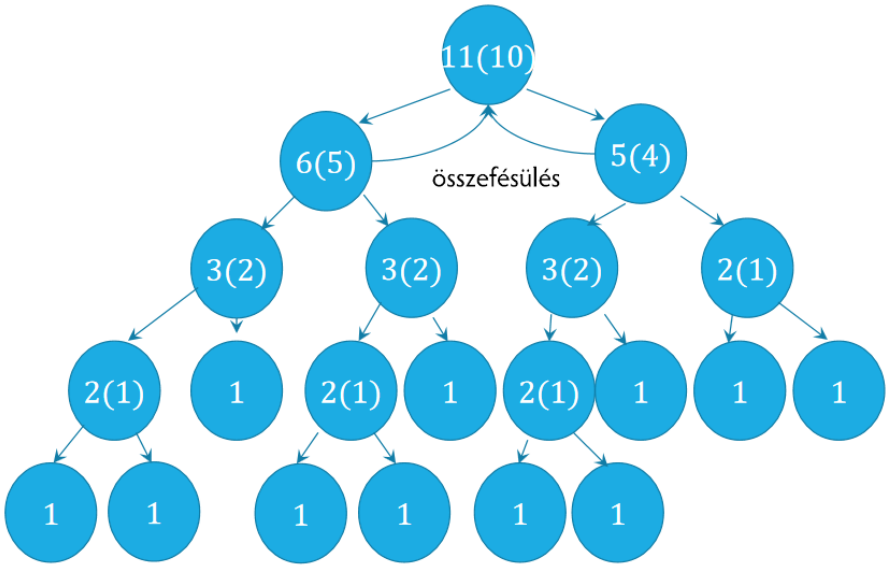
- Az összefuttatás a k és m hosszú sorozatok összefésülésekor $k+m-1$ összehasonlítást végez.

$$\text{MÖ}_{összefuttat}(k,m) = k+m-1$$

- ha a két sorozat együttes hossza n

$$\text{MÖ}_{összefuttat}(n) = n-1$$

- A MergeSort eljárásnál a közel egyenlő széfvágást alapul véve: milyen hosszú részekre vágja az eljárás a (rész)sorozatokat a rekurzív hívások szintjein.



$n = 11$-re
( )-ben:öh-k száma $\quad \text{MÖ} = 10 + (5 + 4) + (2 + 2 + 2 + 1) + (1 + 1 + 1) = 29$

Az egyenlő szétvágások bináris száma majdnem teljes. A fa magassága $h = [\log_2 n] = [\log_2 11] = 4$. A fa leveleihez nem tartozik összehasonlítás, így éppen h szinten kell összegezni az összehasonlítások számát. Ezek összege szintenként csökken és mindenütt $\leq n - 1$. Így:

$$\text{MÖ}_{\text{MS}}(n) = (n-1) * [\log_2 n] = \sigma(n * \log_2 n)$$

Műveletigénynél hasonlóan járhatunk el. Azt akarjuk kitalálni, hogy az elemek melyik sorrendje az igazi sorrend. Kezdetben tehát $n!$ lehetőségünk, van azonban ez midnen összehasonlítás után két részre osztható, mert ha például $x < y$, akkor azok a sorrendek már nem jöhetnek szóba, ahol az x hátrébb van, mint y. Ha a válasz mindig olyan, hogy minél több sorrend maradjon meg, akkor k kérdés után még szóba jövő sorrendek száma $\frac{n!}{2^k}$.

Az összehasonlítások tekinthetők döntési fáknak, amik a rendezés során történt összehasonlításokat ábrázolják. A csúcsokat egy $a_i : a_j$ párral jelölhetjük, ahol $i, j \in [1, n]$. A levelek egy-egy permutációnak feleltethetők meg.

Tétel: Bármely n elemet rendező döntési fa magassága $\sigma(n * \log_2 n)$.

Bizonyítás: Vegyük az n elemet rendező döntési fát, jelöljük ennek magasságát h-val. A fának $n!$ levele van - ezek a permutációk.

A h mélységű bináris fa leveleinek száma $\leq 2^h$, ezért

$$n! \leq 2^h$$

A logaritmikus monotonitás alapján

$$\log_2 n! \leq h$$

A Stirling-formula

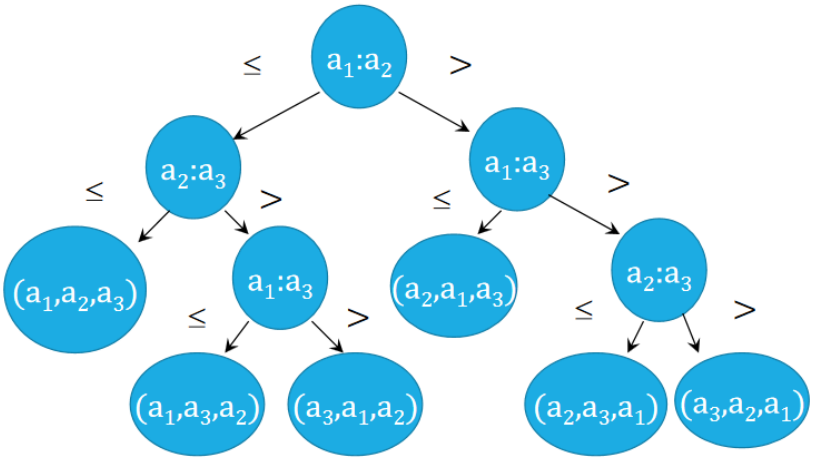
$$n! \approx \sqrt{2\pi n} * \frac{n^n}{e^n}$$

Ezzel alulról becsülhetjük $n!$-t, úgy hogy

$$n! > \frac{n^n}{e^n}$$

Ezt behelyettesítve az eredeti kifejezésbe meg is kapjuk a kívánt eredményt.

$$h \geq \log_2 \left(\frac{n}{e}\right)^n = n * \log_2 n - n * \log_2 e = \Omega(n * \log_2 n)$$



Batcher-féle páros-páratlan összefésüléses rendezés - algoritmus szövegesen, helyességének belátása

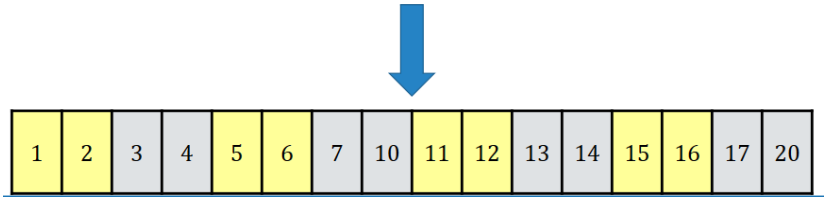
A MergeSort olyan változata, amelyben a lépések jelentős része párhuzamosan végezhető el. Először az A páratlan és a B páros indexű elemeit fésüli össze U-ba, majd az A páros és B páratlan indexű elemeit fésüli össze V-be.

$$A = a_1 < a_3 < a_5 < \cdots$$
$$B = b_2 < b_4 < b_6 < \cdots \quad \Rightarrow \quad U = u_1 < u_2 < u_3 < u_4 < \cdots$$

$$A = a_2 < a_4 < a_6 < \cdots$$
$$B = b_1 < b_3 < b_5 < \cdots \quad \Rightarrow \quad V = v_1 < v_2 < v_3 < v_4 < \cdots$$

Ekkor az U és V sorozatokat már nem kell összefésülni, elég, ha csak az egymás alatti párokat tesszük sorrendbe: $\{u_1, v_1\}, \{u_2, v_2\}, \{u_3, v_3\}$ .... Ebből kialakul a helyes sorrend.

U	1	3	5	10	11	14	15	17
V	2	4	6	7	12	13	16	20



A 2-2 rövidebb sorozat összefésülését ugyanezzel az eljárással végezzük. Ez rekurzív hívással valósul meg. Ehhez meg kell adni a legkisebb k és m értéket, amelyre az eljárás már nem hívja meg önmagát.

- Egy 2 hosszú tömböt még szétvágunk 1 hosszú részre, és meghívjuk az összefésülőt, ekkor $k = 1, m = 1$
- Egy 1 hosszú tömböt már nem fésüljük össze, itt felismerjük, hogy már rendezett, ekkor $k = 1, m = 0$.

Az eljárás helyességét, azonban ez még bizonyításra szorul. Be kell látnunk hogy az eljárás a helyes sorozatot eredményt adja meg.

Esetszétválasztással gondolkodunk, legyen

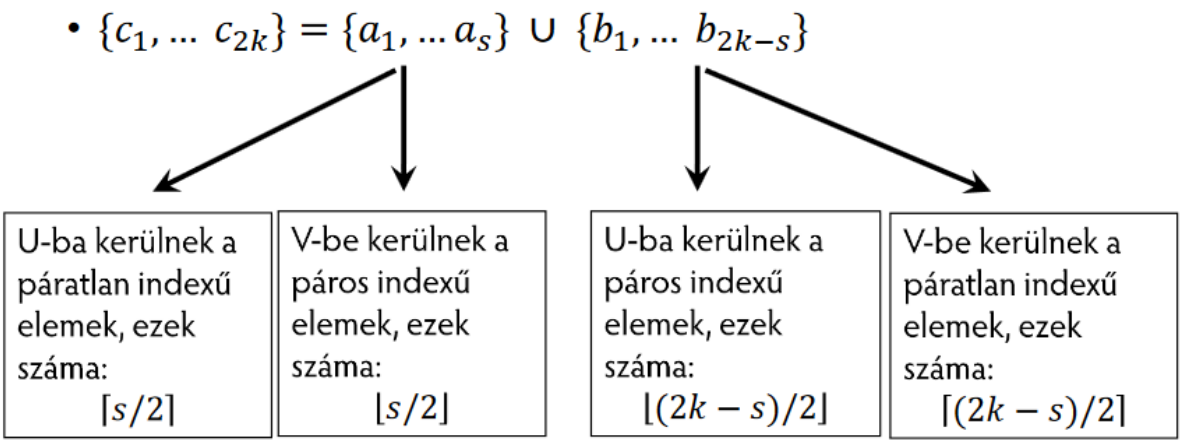
$$c_1 = \min\{u_1, v_1\}, \qquad c_2 = \max\{u_1, v_1\}$$

Általában felírhatjuk, hogy

$$c_{2i-1} = \min\{u_i, v_i\}, \qquad c_{2i} = \max\{u_i, v_i\}$$

Abban az esetben, ha a kimeneti tömb mérete $k + m$ páratlan, még azt is be kell látni, hogy az utolsó elem helyesen képződik, mert az nincs benne a képletben.

1. Először lássuk be, hogy a C sorozatban bármely báros indexű tagig elmenve ugyanannyi u_i szerepel, mint v_i, vagyis az U és V sorozat „cipzár" -szerűen építi fel C-t.
   - A C a rendezett A és B sorozatok összefésülésével adódik. Tegyük fel ,hogy A −ból $a_1, \dots, a_s$, B-ből pedig $b_1, \dots, b_{2k-s}$ kerülnek a C első $2k$ elemébe, azaz
   $$\{c_1, \dots, c_{2k}\} = \{a_1, \dots, a_s\} \cup \{b_1, \dots, b_{2k-s}\}$$



   Ekkor az állítás egyenértékű azzal, hogy

   $$\left\lceil \frac{s}{2} \right\rceil + \left\lfloor \frac{2k-s}{2} \right\rfloor = \left\lfloor \frac{s}{2} \right\rfloor + \left\lceil \frac{2k-s}{2} \right\rceil$$

   Ez nyilván igaz, ha s páros, mert akkor minden tag egész. Ha s páratlan, akkor a kérdés az, hogy

   $$\frac{s+1}{2} + \frac{2k-(s+1)}{2} = \frac{s-1}{2} + \frac{2k-(s-1)}{2}$$

   Ami szintén igaz, tehát az állítás helyes.

2. Eszerint

   $$\{c_1, \dots, c_{2i}\} = \{u_1, \dots, u_i\} \cup \{v_1, \dots, v_i\}$$
   $$\{c_1, \dots, c_{2(i-1)}\} = \{u_1, \dots, u_{i-1}\} \cup \{v_1, \dots, v_{i-1}\}$$

   A két halmaz kivonásával

   $$\{c_{2(i-1)}, c_{2i}\} = \{u_i, v_i\}$$

   Így mivel $c_{2i-1} < c_{2i}$, ezért az eredeti állítás is igaz.

XVII. tétel - Edény- és radix rendezés

Algoritmusa, lépésszáma, szokásos implementációja; leszámláló rendezés **algoritmusa**, lépésszáma

Algoritmusa

Tegyük fel, hogy a bemenő elemek egy m elemű U halmazból kerülnek ki. Foglaljunk le egy U elemeivel indexelt B tömböt, először mind üres. Először végig olvassuk a tömb elemeit és beletesszük a megfelelő edénybe. Ezután végig megyünk a B-n, és a listák tartalmát visszaírjuk A-ba.

Például tegyük fel, hogy a rendezendő $A[1..7]$ tömb elemei 0 és 9 közötti egészek.

A:	5	3	1	5	6	9	6

B:		1		3		5 5	6 6			9	

Általánosabban leírva legyen a K az S sorozat elmeinek típusérték halmaza, és legyen $\varphi: K \to [0 \ldots M-1]$ szigorúan monoton növő függvény. Legyenek $E_0, \ldots, E_{M-1}$, melyek éppen olyan sorozatok, mint S. Az egyes edényekben megmarad az S-beli elemek ottani relatív sorrendje.

$E_0, E_1, \ldots, E_{M-1} \leftarrow \varepsilon, \varepsilon, \ldots, \varepsilon$
$S \neq \varepsilon$
$\text{Out}(S, x)$ $In(E_{\varphi(x)}, x)$
$S \leftarrow$ Összefűzés$(E_0, E_1, \ldots E_{M-1})$

A radix rendezőt akkor használjuk, ha az elemek kulcsai összetettek, vagyis $(t_1, \ldots, t_k)$ alakú szavak, ahol a t_i komponens az L_i rendezett típusból való, és a rendezés lexikografikus. A sorozatot mindig úgy rendezzük, hogy hátulról az utolsó komponenssel kezdjük a rendezést és onnan haladunk előre. Az edényekkel való rendezés azért jó, mert az algoritmusa közben az elemeket a lista végére tettük, azaz két azonos kulcsú elem sorrendje nem cserélődik fel az eredetihez képest, tehát stabil rendezésről van.

Az edények szokásos implementációja egy fejelemes és egy vége elemes láncolt listával történik, mert így nem kell kiolvasni az elemeket, csupán össze kell őket láncolni.

Példa:

1969. jan. 18. 1969. jan. 1. 1955. dec. 18. 1955. jan. 18. 1918. dec. 18.

1. menet után:

1969. jan. 1. 1969. jan. 18. 1955. dec. 18. 1955. jan. 18. 1918. dec. 18.

2. menet után:

1969. jan. 1. 1969. jan. 18. 1955. jan. 18. 1955. dec. 18. 1918. dec. 18.

3. menet után:

1918. dec. 18. 1955. jan. 18. 1955. dec. 18. 1969. jan. 1. 1969. jan. 18.

Általánosan

Az $e = e_d e_{d-1} \dots e_2 e_1$ számot jobbról balra, az alacsony helyiértékek felől indulva pozíciónként szétrakja edényekbe, majd összefűzi az edények tartalmát

Az i. pozíción a φ_i függvényt alkalmazzuk: $\varphi_i(e) = e_i$

Az i. pozíción végrehajtott szétrakás és összefűzés után S „i-rendezett" lesz.

Definíció

- S „i-rendezett" (jelölés: $x \leq_i y$), ha minden
  - $x = x_d x_{d-1} \dots x_2 x_1$ és $y = y_d y_{d-1} \dots y_2 y_1$–ra: $x <_0 y$
  - $x \leq_i y \Leftrightarrow x_i < y_i$ vagy $x_i = y_i$ és $x <_{i-1} y$ $(i > 0)$
- Ekkor a „d-rendezés" a közönséges rendezés

Hatékonyság:

- $2 * d$-szer megyünk végig az S sorozaton, így

$$T(n) = \Theta(d * |S|)$$

```
radix(S)
i <- 1
while i<d
  bucketsort(S, phi(i))
  i <- i+1
end while
```

Lépésszáma

- B létrehozása: $\sigma(m)$
- első fázis: $\sigma(n)$
- második fázis: $\sigma(n+m)$, összesen $\sigma(m+n)$

Ez gyorsabb, mint az általános alsó korlát, ha $m \leq c * n$

Leszámoló rendezés algoritmusa, lépésszáma

A leszámoló rendezés alapötlete, hogy megszámoljuk egy adott elemre a nála kisebb elemek számát, így tudni fogjuk a pozícióit. Feltéve, hogy van bemeneti tömbünk n db. elem, melyek mindegyike 1 és k közötti egész. Legyen a bemenet A tömb, és a kimenet a B. Mindekettő hossza n. Szükség van még egy $C[1..k]$ hosszú tömbre is munkaterülethez.

1. Végig megyünk A-n és ha egy elem értéke i, akkor megnöveljük C[i] értékét eggyel.

• A

3	6	4	1	3	4	1	4

• C

2	0	2	3	0	1

2. Mindegyik i-re meghatározzuk, hogy hány olyan bemeneti elem van, amelyiknek az értéke kisebb nála.

• A

3	6	4	1	3	4	1	4

• C

2	2	4	7	7	8

3. Minden i-re n és 1 között az A[i]-t betesszük B megfelelő pozíciójába (ezt állapítjuk meg C-ből), majd csökkentjük a C-beli értéket.

• A

3	6	4	1	3	4	1	4

• C

2	2	4	7	7	8

• B

						4	

• A

3	6	4	1	3	4	1	4

• C

0	2	2	4	7	7

• B

1	1	3	3	4	4	4	6

Az algoritmus pszeudo kódja:

```
for i←1 to k do
   C[i]←0                          ← C-ben az i-vel egyenlő elemek száma
for i←1 to hossz(A) do
   C[A[i]]←C[A[i]]+1               ← C-ben az i-nél kisebb, vagy egyenlő elemek száma
for i←2 to k do
   C[i]←C[i]+C[i-1]
for i←hossz(A) downto 1 do
   B[C[A[i]]]←A[i]
   C[A[i]]←C[A[i]]-1
```

Ennek futási ideje sorban:

Futási idő:

- 1. for ciklus: $\Theta(k)$
- 2. for ciklus: $\Theta(n)$
- 3. for ciklus: $\Theta(k)$
- 4. for ciklus: $\Theta(n)$

Így a teljes időigény: $\Theta(k+n)$

- Ha $k = \Theta(n)$, akkor a rendezés futási ideje $\Theta(n)$!

Ez nem összehasonlító rendezés

- A helyigénye viszont nagyobb ☹

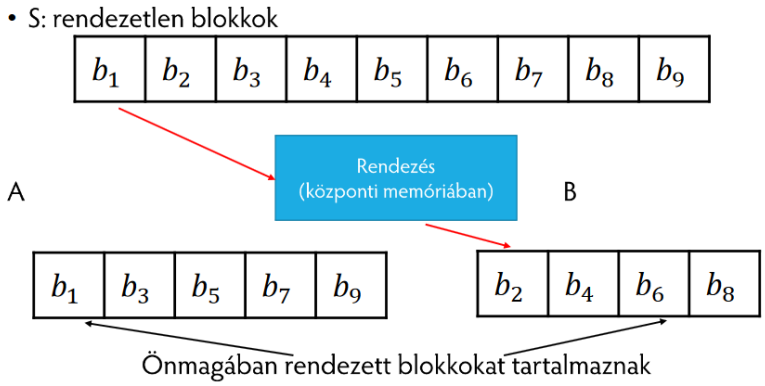
XVIII. tétel - Külső rendezések

2 segédfájlos (4 fájlos), 3 fájlos stb.

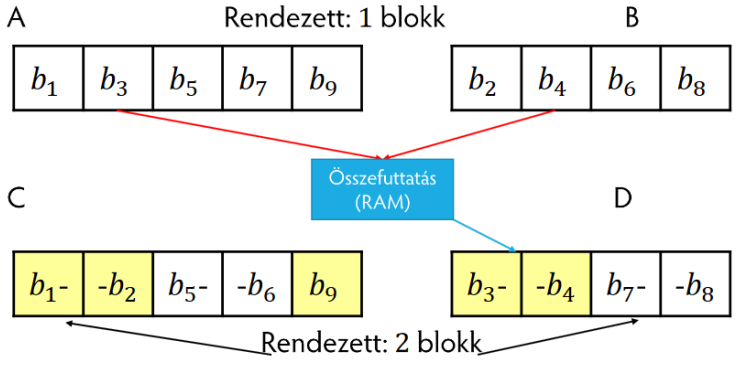
Adattárolás sémája (bevezetés)

Az adattárolás egy háttértárolón nem bit/byte egységekbe van szervezve. Ilyen nagyobb egység a lap (HDD: szektor, SSD: blokk). Ez lehet például 2048 byte, vagy 4096 byte. Ez az átvitel egysége! Lényegében az átvitelek száma határozza meg a sebességet. A háttértárról egy lap olvasása lassabb, mint a főmemória olvasása. Az eddigi rendezéseknél feltettük, hogy az adatok a központi memóriában vannak. Ennek megfelelt, hogy a hatékonyságot az összehasonlítások számában mértük, ebben az esetben viszont ún. I/O utasítások teszik ki a futási idő döntő részét.

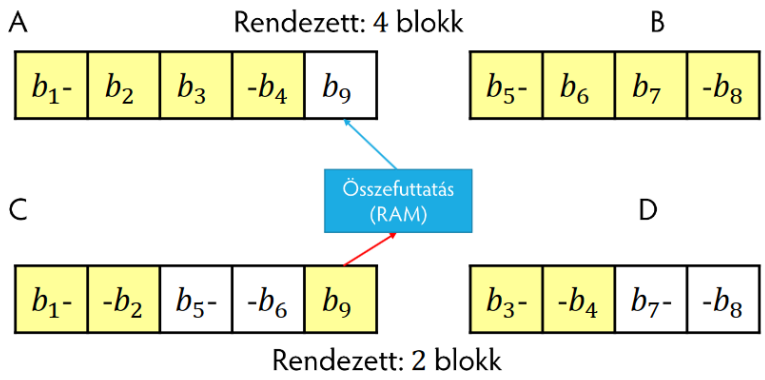
1. Beolvassuk az S rendezetlen blokkjait valamilyen belső rendezővel rendezzük, majd felváltva A-ba és B-be íratjuk.



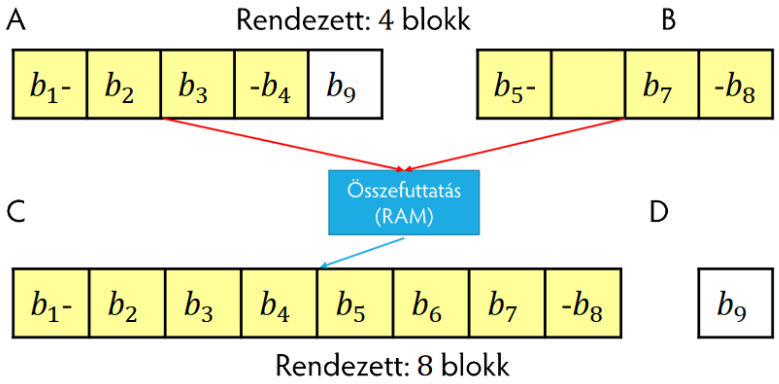
2. Sorban beolvassuk az A és B 1-1 blokkját. Ezek rendezettek. Összefésüljük őket és a rendezett két blokk hosszú adatot felváltva C-be, illetve D-be írjuk. Az utolsó blokknak nincs párja, azt beírjuk C-be.



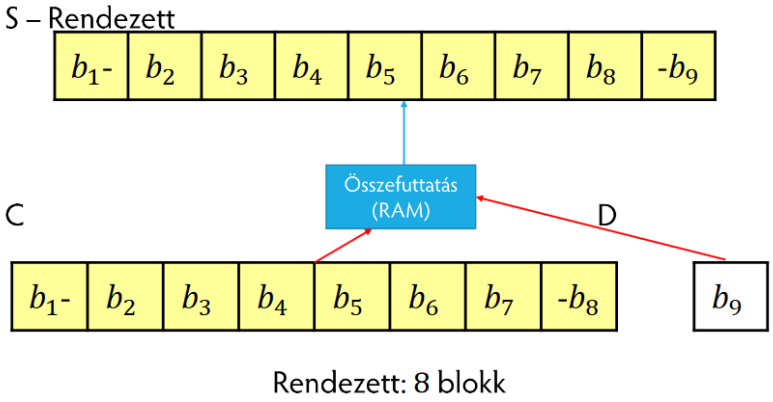
3. C-ből és D-ből olvasunk 2-2 rendezett blokkot, összefésüljük őket és felváltva A-ba és B-be írjuk a rendezett 4 blokkot. Az utolsó magányos blokk A végére kerül.



4. C-be kerül A és B 4-4 rendezett blokkja összefésülésének egy 8 hosszú rendezett az eredménye, D-be pedig kerül a maradék 1.



5. C 8 blokkját és a D 1 blokkját összefésüljük S-be. Egy k blokkból álló összefüggő rendezett részt k hosszú futamnak nevezünk.



Nem törvényszerűen történik az, hogy nem kerül kapcsolatban a magányos darab más részekkel, pl 15-re már nem igaz. Ha a központi memória mérete korlátozott és nem képes befogadni az egyre növekvő méretű rendezett részeket (futamokat), akkor ezek összefésülését lehet pufferelve, akár blokkonként végezni.

Általában

- 1. menet eredménye: 1 hosszú futamok
- 2. menet eredménye: 2 hosszú futamok
- 3. menet eredménye: 4 hosszú futamok
- ….
- $(k-1)$. menet eredménye: 2^{k-2} hosszú futamok
- k. (utolsó) menet eredménye: $\leq 2^{k-1}$ hosszú egyetlen futam = S
  - előtte maradék mindig lehet

Gyorsítási lehetőség, hogy nagyobb kezdőfutamokat hozunk létre. Ha a központi memória lehetővé teszi, akkor az 1. menetben megtehetjük, hogy S-ből m>1 blokkot olvasunk be és ezt rendezzük, és az így keletkezett m hosszú kezdőfutamokat írjuk ki A-ba és B-be. Ezután először két m hosszút fésülünk össze, majd két 2m hosszút stb.

1. menet eredménye: m hosszú futamok
2. menet eredménye: $2 * m$ hosszú futamok
3. menet eredménye: $4 * m$ hosszú futamok
….
$(k-1)$. menet eredménye: $2^{k-2} * m$ hosszú futamok
k. (utolsó) menet eredménye: $\leq 2^{k-1} * m$ hosszú egyetlen futam = S
Innen $2^{k-2} * m < n \leq 2^{k-1} * m$

Lehet azt is, hogy több mint kétfelé fésülünk, például ha az S fájl mellet nem 2*2, hanem általában 2*m fájllal dolgozunk, akkor hatékonyabb eljárás végezhető.

Három fájlos külső rendező is létezikm ekkor annyi a különbség, hogy 2 fájlba olvasunk először ugyan, a rendezett blokkokat 2 fájl tárolja, de az összefésülést csak egy fájlba tudjuk megtenni, mégpedig úgy, hogy amíg ki nem ürül az egyik vagy másik feldolgozó fájl, addig írjuk a harmadikba, amint kiürült, úgy kezdjük előröl az összefésülést a két nem üres fájllal. Ennek futási sebességét azonban jelentősen befolyásolja, hogy milyen kezdeti futamszámot állítunk be, illetve az is, hogy az egyes menetekben, hogy alakítjuk a kimenti fájlok elrendezését. Belátható, hogy a 3 fájlos rendező éppen akkor fut le a leggyorsabban, ha a szomszédos Fibonacci számokat feleltetünk meg a menetekben az egyes fájlok futamszámának, úgy hogy az első lépésben elindulunk két szomszédostól és így egyesével lépkedünk vissza a sorozaton.