Adatszerkezetek és Algoritmusok

Erdélyi Áron

2020.09.14.

Table of Contents

1 Adatszerkezetek és típusok

1.1 Az adattípus absztrakciós szintjei

- Absztrakt adattípus (ADT)
  - Semmit nem feltételez a belső szerkezetről
  - Enkapszuláció
- Absztrakt adatszerkezet (ADS)
  - Absztrakt szerkezet
  - Irányított gráf mutatja a rákövetkezéseket:
    - Csúcsok - Adatelemek
    - Élek - Rákövetkezések
- Reprezentáció
  - ADS gráf az absztrakt memóriában
  - Fontos, hogy a rákövetkezések megmaradjanak!
  - Láncolt, vagy aritmetikai ábrázolás
- Implementáció
  - Programnyelveken
- Fizikai ábrázolás
  - Biteken

1.2 A típus-specifikáció és típus fogalma

1.3 Típus-specifikáció

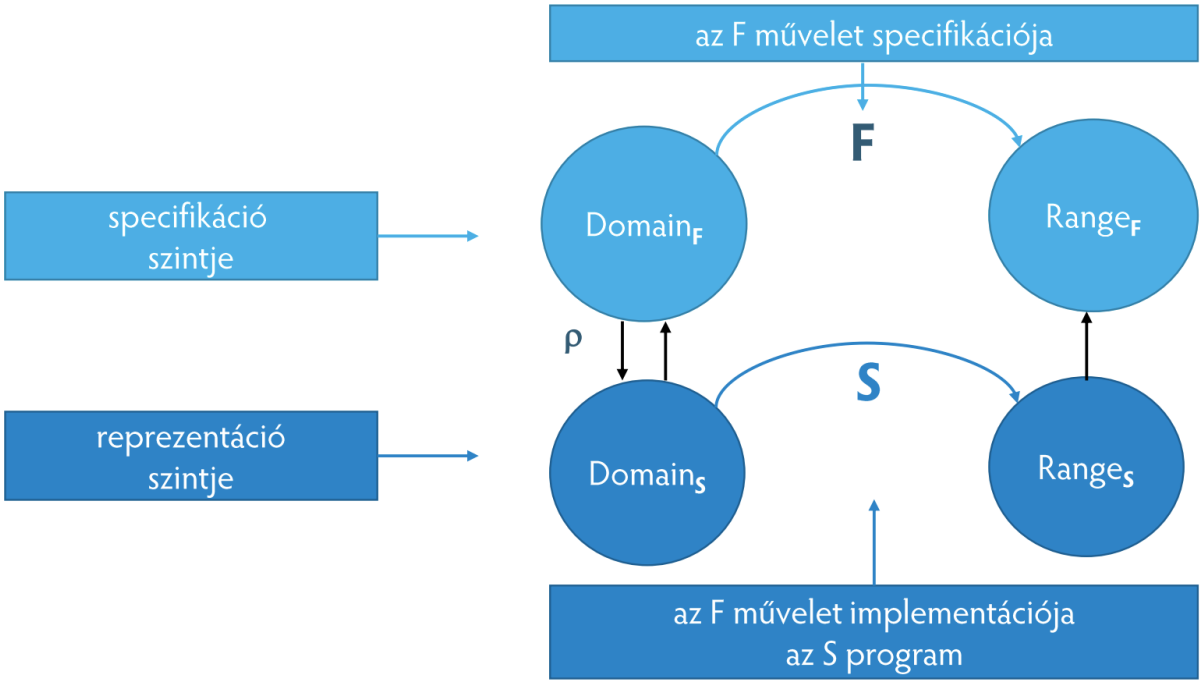
- Egy adat külső jellemzésére szolgál (interfész)
  - típusérték-halmaz (T)
  - típusműveletek (T-n értelmezett feladatok)
- Megadására több lehetőség van:
  - Algebrai specifikáció (axiómák megadásával)
  - Funkcionális specifikáció (elő- és utófeltételekkel)

1.4 Típus

- Típus-reprezentáció:
  - Ábrázoló elemek H halmaza. (típus-szerkezet)
  - Az ábrázoló elemek és a typusértékek kapcsolatát leíró lehépezés:

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

- Típus-implementáció
  - Nem a típusértékekkel, hanem az azokat ábrázoló elemekkel működő programok.



1.5 Absztrakt adattípus és megadási módjai

- A típus szemléletének legmagasabb szintje
- Semmilyen feltételezéssel nem élünk a szerkezetről, vagy a megvalósításról
- A típus-specifikáció megadására szolgál
  - Nem szükséges egy konkrét programozási környezetben ábrázolni a típusértékeket.
  - Elég a műveletek programjainak csak a hatását ismerni.
- Szükség van egy őt kiváltó konkrét típusra.
- A megadásnál csak tisztán matematikai fogalmakat használunk.

1.5.1 Az ADT algebrai specifikációja

- Részei
  - Típusérték halmaz
  - Műveletek (leképezések)
  - Megszorítások (értelmezési tartományok)
  - Axiómák
- Kérdések
  - Helyesség
  - Teljesség
  - Redundancia

1.5.2 Az ADT funkcionális specifikációja

- A típus matematikai reprezentációját használjuk.
- Részei
  - Típusérték halmaz
  - Máveletek

- Állapottér
- Paramétertér
- Előfeltétel
- Utófeltétel

1.6 Verem ADT axiomatikus és funkcionális leírása

1.6.1 A verem ADT axiomatikus leírása

E alaptípus feletti V verem típus jellemzése

- Műveletek
  - empty: $\rightarrow V$
  - isEmpty: $V \rightarrow L$
  - push: $V \times E \rightarrow V$
  - pop: $V \rightarrow V \times E$
  - top: $V \rightarrow E$
- Megszorítások
  - pop és top értelmezési tartománya: $V \backslash \{\emptyset\}$
- Axiómák
  - $isempty(\emptyset)$
  - $isempty(v) \Rightarrow v = \emptyset$
  - $\neg isempty(push(v, e))$
  - $pop(push(v, e)) = (v, e)$
  - $push(pop(v)) = v$
  - $top(push(v, e)) = e$

1.6.2 A verem ADT funkcionális leírása

- Matematikai reprezentáció
  - a verem rendezett párok halmaza

$$v = \big\{(e_1, t_1), ..., (e_i, t_i) | t_i \neq t_j\big\}$$

  - e: a veremben elhelyezett ($push$) érték
  - t: a veremben elhelyezés időpontja
- Megszorítás: az idő értékek külömbözőek
- A pop specifikációja:
  - Állapottér:

$$(v, e) \in V \times E$$

  - Paramétertér:

$$v \text{ és } v'$$

  - Előfeltétel és utófeltétel:
    - $Ef = (v = v' \wedge v' \neq \emptyset)$
    - $Uf = \Big(\big(v = v' \backslash \{(e_j, t_j)\}\big) \wedge (e = e_j) \wedge \big((e_j, t_j) \in v'\big) \wedge \big(\forall i \big((e_i, t_i) \in v' \wedge i \neq j\big) : t_j > t_i\big)\Big)$

1.7 Absztrakt adatszerkezet megadása

- A típus elepvető szerkezetét egy irányított gráffal ábrázoljuk. A gráf jelentése:
  - Csúcsok: adatelemek
  - Élek: rákövetkezési reláció
- A műveletek ezen a szinten is jelen vannak (élek)
- A művelet hatása szemléltethető az ADS-gráf változásaival.

1.8 Reprezentáció

- Absztrakt reprezentáció - absztrakt memóriában
- Az absztrakt szerkezet ábrázolható
  - Pointerekkel (láncolt ábrázolás)
  - Cím-/indexfüggvényekkel (aritmetikai reprezentáció)
  - (vegyes ábrázolás lehetséges)
- Mindkét reprezentáció megadja az adatelemek közötti rákövetkezési relációt.

1.9 Aritmetikai és láncolt ábrázolás

1.9.1 Aritmetikai ábrázolás

- Index- és címfüggvények megadása
  - Az adatelemeket folyamatosan elhelyezzük az absztrakt memóriában / egy ugyanilyen alaptípusú vektorban
  - Az elemek közötti rákövetkezési relációt egy cím/indexfüggvénnyel adjuk meg
  - A címfüggvényből további rákövetkezések is kiolvashatók, nem csak az ADS-beli
  - A műveletek és a feladat-specifikus függvények algoritmusait meg kell adni
- Példa: Kupac

1.9.2 Láncolt ábrázolás

- Az ADS-gráf éleit pointerekkel ábrázoljuk
- A műveletek algoritmusait már itt meg kell adni
- A feladatban bevezetett függvények kiszámító algoritmusait is meg kell adni
- Következmény: Egy feladat megoldása gazdagabb reprezentációt igényelhet, mint maga az ADS
- Példa: Fák, kupac

1.10 Az adatszerkezetek osztályozása

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

1.10.1 Adatelemek típusa szerint

- **Homogén**: Az adatszerkezet mindegyik eleme azonos típusú.
- **Heterogén**: Az adatszerkezet elemei különböző típusúak lehetnek.

1.10.2 R reláció szerint

- **Struktúra nélküli**: Az egyes adatelemek között nincsen kapcsolat. Nem beszélhetünk az elemek sorrendjéről (pl.: halmazok).
- **Asszociatív címzésű**: Az adatelemek között lényegi kapcsolat nincs. Az adatszerkezet elemei tartalmuk alapján címezhetők.
- **Szekvenciális**: Az egyes adatelemek egymás után helyezkednek el. Az adatok között egy-egy jellegű kapcsolat van: minden adatelem csak egy helyről érhető el és az adott elemről csak egy másik látható. Két kitüntetett elem az első és az utolsó.
- **Hierarchikus**:
  - Olyan $\langle A, R\rangle$ rendezett pár, amelynél van egy kitüntetett r elem, ez a gyökérelem, úgy, hogy
    - r nem lehet végpont
    - $\forall a \in A\backslash\{r\}$ egyszer és csak egyszer végpont
    - $\forall a \in A\backslash\{r\}$ elem r-ből elérhető
  - Az adatelemek között egy-sok jellegű kapcsolat áll fenn.
  - Minden adatelem csak egy helyről érhető el, de egy adott elemből akárhány adatelem látható.
- **Hálós**: Olyan $\langle A, R\rangle$ rendezett pár, amelynél az R relációra semmilyen kikötés nincs. Az adatelemek között a kapcsolat sok-sok jellegű: bármelyik adatelemhez több helyről is eljuthetunk, és bármelyik adatelemből elvileg több irányban is mehetünk tovább.

1.10.3 Reprezentáció szerint

- **Folytonos ábrázolású**: A központi tárban a tárelemek egymás után helyezkednek el. Az adattételek tárolási jellemzői azonosak. Ismert az első elem címe, ehhez képest bármely elem címe számítható.
- **Szétszórt ábrázolású**: A tárelemek véletlenszerűen helyezkednek el. Közöttük a kapcsot az teremti meg, hogy minden elem tartalmaz más elemek elhelyezkedésére vonatkozó információt.

1.10.4 Adatelemek száma szerint

- **Statikus**: Egy statikus adatszerkezet rögzített számú adatelemet alkot. A feldolgozás során az adatelemek csak értéküket változtatják, de maga a szerkezet, az abban szereplő elemek száma változatlan. Emiatt az adatszerkezetnek a memóriában foglalt helye konstans.
- **Dinamikus**: Egy dinamikus adatszerkezetben az adatelemek száma egy adott pillanatban véges ugyan, de a feldolgozás alatt tetszőlegesen változhat.Lehetnek rekúrzív, vagy nem-rekurzív, lineáris, vagy nem-lineáris struktúrák.

  Rekurzív, ha definíciójában saját magára hivatkozik. Ha egy ilyen hívás van lineáris, ha több, akkor nem-lineáris.

  Mivel az adatelemek száma változik, így egy-egy területet kell allokálunk, illetve a lefoglalt területeket fel kell szabadítanunk.

2 Objektumorientált programozás 1.

2.1 Osztályok és objektumok

2.1.1 Osztályok

Objektumminta, vagy típus, ami alapján objektumokat hozhatunk létre. Az objektumokat kategóriákba, osztályokba soroljuk. A hasonló tulajdonságokkal rendelkező objektumok egy osztályba kerülnek, míg a különböző, vagy eltérő tulajdonságokkal rendelkező objektumok pedig külön osztályba kerülnek.

Az **objektum-osztályok** hordozzák a hozzájuk hasonló objektumok jellemzőit, **objektumok mintái**nak tekinthetők.

2.1.2 Objektumok

Egy osztály alapján létrejött **konkrét példány**. Az objektumnak **belső állapot**a van, információt tárol. Kérésre feladatot hajt végre, metódusok segítségével. Ezek hatására az állapota megváltozhat.

Kommunikál más objektumokkal. Minden objektum **egyértelműen azonosítható**.

2.2 Objektum létrehozása, inicializálása

Az objektumot létre kell hozni és inicializálni kell.

- Inicializálás:
  - Konstruktor végzi
  - Adatok kezsőértékének megadása
  - Objektum működéséhez szükséges tevékenységek végrehajtása
  - Típusinvariáns beállítás

2.3 Osztálydefiníció

2.3.1 Példányváltozó

Példányonként helyet foglaló változó.

2.3.2 Példánymetódus

Példányokon dolgozó metódus.

2.3.3 Osztályváltozó

Osztályonként helyet foglaló változó (*static* kulcsszó, osztályon kívül kell definiálni).

2.3.4 Osztálymetódus

Osztályokon dolgozó metódus (*static* kulcsszó, osztályon kívül kell definiálni, csak publik metódus lehet).

2.4 Öröklődés, Felüldefiniálás

Az alapgondolat: a gyerek örökli az őseik metódusait és változóit. Itt az öröklés azt jelenti, hogy az ősosztály minden metódusa és adattagja a gyermejosztálynak is metódusa és adattagja lesz.

A gyermekosztály bevezethet új műveleteket és adattagokat, amelyek egyszerűen hozzáadódnak az örökölt metódusokhoz és adattagokhoz.

Átdefiniálhat/Felüldefiniálhat örökölt metódusokat, ezek a hierarchiában felülbírálják az örökölt metódusokat.

2.5 Elfedés

Az egyik OOP elvárás az információ elfedése (information hiding). Az objektum belügyeit csak az interfészen keresztül lehet megközelíteni. C++-ban:

- **public**: A külső felhasználók is elérik.
- **protected**: Csak az adott osztály, és leszármazottai érhetik el.
- **private**: Csak az adott osztály és "barátai" számára elérhető - alapértelmezett osztályoknál.

3 Objektumorientált programozás 2.

3.1 Öröklődés

Az alapgondolat: a gyerek örökli az őseik metódusait és változóit. Itt az öröklés azt jelenti, hogy az ősosztály minden metódusa és adattagja a gyermejosztálynak is metódusa és adattagja lesz.

A gyermekosztály bevezethet új műveleteket és adattagokat, amelyek egyszerűen hozzáadódnak az örökölt metódusokhoz és adattagokhoz.

Átdefiniálhat/Felüldefiniálhat örökölt metódusokat, ezek a hierarchiában felülbírálják az örökölt metódusokat.

3.2 Polimorfizmus

A polimorfizmus (többalakúság) az a jelenség, hogy egy változó nem csak egyfajta típusú objektumra hivatkozhat (pl.: A Manager Employee is, A Háromszög Alakzat, stb...).

- **Statikus típus**: Deklaráció során kapja
- **Dinamikus típus**: Futásidőben változhat, hogy éppen milyen típusú objektumra hivatkozik. Statikus, vagy annak leszármazottja.

Altípusos polimorfizmus: Ha B altípusa az A típusnak, akkor B objektumainak referenciái értékül adhatók az A típus referenciáinak.

3.3 Dinamikus összekapcsolás (példákkal)

Run-time fogalom: Az a jelenség, hogy a változó éppen aktuális dinamikus típusának megfelelő metódus implementáció hajtódik végre.

3.3.1 Példák:

Employee-Manager:

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

Az első printnél az Employee class print függénye, a másodiknál a Manager class printje fog lefutni.

3.4 Absztrakt osztály

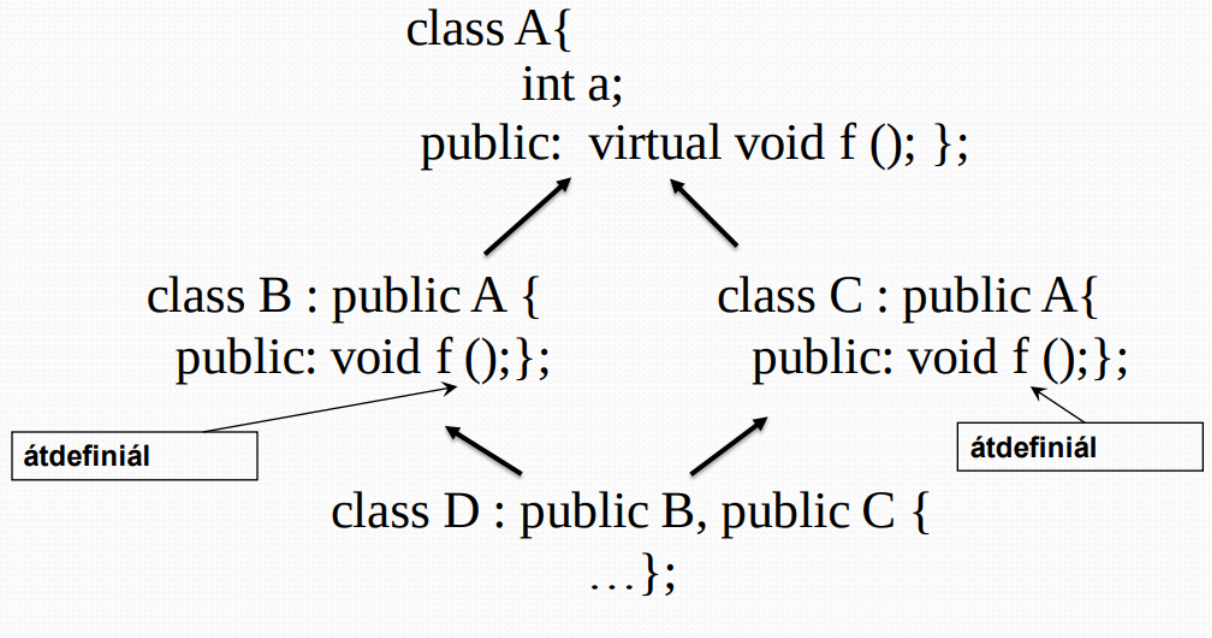
- Tervezés eszköze.
- Egy felső szinten összefogja a közös tulajdonságokat.
- A metódusok között van olyan, aminek csak specifikációja van, törzse nincs.
- Nem hozható létre példánya.
- A leszármazott teszi konkréttá.
- C++-ben fontos kulcsszó: virtual

3.5 A többszörös öröklődés problémái, lehetséges megoldásai

A többszörös öröklődés jelentése, hogy egy osztálynak egynél több közvetlen őse lehet.

3.5.1 Problémák

- Adattagok hányszor öröklődnek?
- Melyik őstől melyik metódus?



- Ha egy D-beli f-re hivatkozunk, (ami át lett definiálva B-ben és C-ben) akkor az melyiket jelentse?
- Az a attribútum hány példányban jelenjen meg D-ben?
- A két kérdés lényegében ugyan az. Ha kétértelműség van, hogyan válasszunk?

3.5.2 Megoldási lehetőségek

- A legtöbb esetben az ilyen kódot nem lehet lefordítani, a fordító, vagy a futtató környezet kétértelműségre hivatkozva hibajelzéssel leáll.
- Az ősosztály mondja meg, hogy mit szeretne tenni ilyen esetben.
- A származtatott osztály mondja meg, hogy melyikeket szeretné használni.

4 Tömbök

4.1 Definíció ADT és ADS szinten

4.1.1 Definíció ADT szinten

- **Definíció**:
  - Az E alaptípusú $k(k \geq 1)$ dimenziós T tömbtípus
  - Legyen $I = I_1 \times I_2 \times ... \times I_k$ egy indexhalmaz, ahol $\forall j \in [1,k] : I_j = [1, n_j]$.
  - Az $A \in T$ tömbnek $N = n_1 * n_2 * ... * n_k$ eleme van, $\{a_1, a_2, ..., a_N\}$.
- Mindig van egy $f : I \to \{a_1, a_2, ..., a_N\}$ egy-egy értelmű leképezés.
- **Jelölés**: $A[i_1, i_2, ..., i_k]$ a tombnek az $i_1, i_2, ..., i_k$ indexek által azonosított eleme.
- **Műveletek**:
  - Indexelés: az $i_1, i_2, ..., i_k$ indexhez tartozó $A[i_1, i_2, ..., i_k]$ elem kiválasztása.
  - Elemmódosítás - értékadás: $A[i_1, i_2, ..., i_k] := a$
  - Értékadás: $A = B$
- **Elnevezés**:
  - $k = 1$: vektor
  - $k = 2$: mátrix
- Invariáns is adható - speciális megszorítás

4.1.2 Definíció ADS szinten

- Nem kötelező szerkezetet definiálni az elemek között
- Elfogadott a kov_j reláció bevezetése $\forall j \in [1,k]$-ra:
  - $kov_j(A[i_1, ..., i_j, ..., i_k]) = A[i_1, ..., i_j + 1, ..., i_k]$, ha $i_j < n_j$. Egyébként nem definiált.
  - Egy belső elemnek minden dimenzióban van rákövetkezője.
- A legalább 2 dimenziós tömböket ortogonális adatszerkezetnek nevezik.

4.2 Reprezentáció

Egy k dimenziós tömböt szeretnénk elhelyezni egy alkalmas méretű egydimenziós tömbben, és megadjuk a leképezés címfüggvényét. Az elhelyezés általában sorfolytonos (SF), vagy oszlopfolytonos (OF).

4.3 Sorfolytonos és oszlopfolytonos ábrázolás

SF esetben a sorokat rakjuk egymás mögé, az oszlopfolytonos esetben, pedig az oszlopokat rakjuk kvázi egymás alá.

4.4 Cím és index függvény

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

Indexfüggvény:

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

Címfüggvény:

$$SF: \quad cim(A[i,j]) = cim(A) + (i-1) * n_1 * h + (j-1) * h$$
$$OF: \quad cim(A[i,j]) = cim(A) + (j-1) * n_2 * h + (i-1) * h$$

4.5 Speciális mátrixok cím és indexfüggvényei

Az R invariáns leggyakrabban a tömb alakját módosítja, például megadja a 0 elemek helyét, és a nemnulla elemek határozzák meg a tömb speciális alakját.

Konvenció: A gyakran szerepló elemeket (általában 0) is tároljuk egy példányban az egydimenziós tömbben, és az erre vonatkozó hivatkozást beépítjük a címfüggvénybe.

4.5.1 Tridiagonális

Ha $|i-j| \leq 1$:

$$ind(A[i,j]) = (i-1)*3-1+\begin{cases}1 & i>j\\ 2 & i=j\\ 3 & i<j\end{cases}.$$

Ha a 0 elemet is tároljuk a vektor elelején:

$$ind(A[i,j]) = \begin{cases}(i-1)*3+1 & i=j+1\\ (i-1)*3+2 & i=j\\ i*3 & i+1=j\\ 1 & \text{különben}\end{cases}.$$

4.5.2 Alsó háromszög mátrix

Elemeinek száma: $m = n*\frac{n+1}{2}+1$

$$ind(A[i,j]) = \begin{cases}i*\frac{i-1}{2}+j & i\geq j\\ n*\frac{n+1}{3}+1 & i<j\end{cases}.$$

4.6 Hézagos mátrixok reprezentációja

Láncolt, illetve vegyes ábrázolásban egy elemet a koordinátákkal, az értékkel, illetve minden irányban a rákövetkező elemeire mutató pointerekkel reprezentáljuk. Előnyös, ha

$$(h+(2*i)+(2*p))*k+(m+n)*p << m*n*h,$$

ahol a mátrix $m \times n$-es, k a nem-nulla elemek száma, h egy érték helyfoglalása, p egy mutató helyfoglalása és i egy index helyfoglalása.

5 Verem, sor és használatuk

5.1 ADT axiomatikus és funkcionális specifikáció

5.1.1 A verem ADT axiomatikus leírása

E alaptípus feletti V verem típus jellemzése

- Műveletek
  - empty: $\rightarrow V$
  - isEmpty: $V \rightarrow L$
  - push: $V \times E \rightarrow V$
  - pop: $V \rightarrow V \times E$
  - top: $V \rightarrow E$
- Megszorítások
  - pop és top értelmezési tartománya: $V \backslash \{\emptyset\}$
- Axiómák
  - $isempty(\emptyset)$
  - $isempty(v) \Rightarrow v = \emptyset$
  - $\neg isempty(push(v, e))$
  - $pop(push(v, e)) = (v, e)$
  - $push(pop(v)) = v$
  - $top(push(v, e)) = e$

5.1.2 A verem ADT funkcionális leírása

- Matematikai reprezentáció
  - a verem rendezett párok halmaza

$$v = \{(e_1, t_1), ..., (e_i, t_i) | t_i \neq t_j\}$$

  - e: a veremben elhelyezett ($push$) érték
  - t: a veremben elhelyezés időpontja
- Megszorítás: az idő értékek külömbözőek
- A pop specifikációja:
  - Állapottér:

$$(v, e) \in V \times E$$

  - Paramétertér:

$$v \text{ és } v'$$

  - Előfeltétel és utófeltétel:
    - $Ef = (v = v' \wedge v' \neq \emptyset)$
    - $Uf = \Big(\big(v = v' \backslash \{(e_j, t_j)\}\big) \wedge (e = e_j) \wedge \big((e_j, t_j) \in v'\big) \wedge \big(\forall i \big((e_i, t_i) \in v' \wedge i \neq j\big) : t_j > t_i\big)\Big)$

5.1.3 A sor ADT axiomatikus leírása

E alaptípus feletti S sor típus jellemzése

- Műveletek:
  - empty: $\rightarrow S$
  - isEmpty: $S \rightarrow L$
  - in: $S \times E \rightarrow S$
  - out: $S \rightarrow S \times E$
  - first: $S \rightarrow E$
- Megszorítások:
  - out és first értelmezési tartománya $S \backslash \{\emptyset\}$.
- Axiómák:
  - $s = empty \Rightarrow isEmpty(s)$
  - $isEmpty(s) \Rightarrow s = empty$
  - $\neg isEmpty(in(s, e))$
  - $\neg isEmpty(s) \Rightarrow out(in(s, e))_2 = out(s)_2$
  - $\neg isEmpty(s) \Rightarrow in(out(s)_1, e) = out(in(s, e))_1$
  - $isEmpty(s) \Rightarrow out(in(s, e))_2 = e$
  - $first(s) = out(s)_2$

Ahol $(s, e)_1 = s$, $(s, e)_2 = e$.

5.1.4 A sor ADT funkcionális leírása

- Matematikai reprezentáció
  - a sor rendezett párok halmaza

$$v = \{(e_1, t_1), ..., (e_i, t_i) | t_i \neq t_j\}$$

  - e: a sorban elhelyezett (in) érték
  - t: a sorban elhelyezés időpontja
- Megszorítás: az idő értékek külömbözőek
- A *out* specifikációja:
  - Állapottér:

$$(s, e) \in S \times E$$

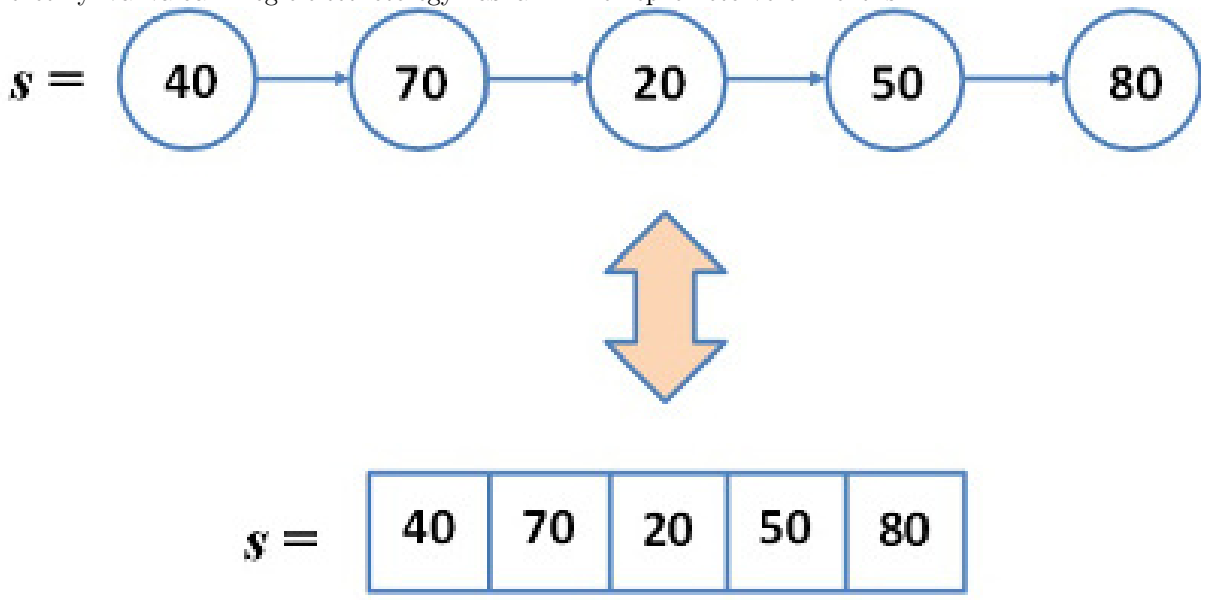
  - Paramétertér:

$$s \text{ és } s'$$

  - Előfeltétel és utófeltétel:
    - $Ef = (s = s' \wedge s' \neq \emptyset)$
    - $Uf = \Big(\big(s = s' \backslash \{(e_j, t_j)\}\big) \wedge (e = e_j) \wedge \big((e_j, t_j) \in s'\big) \wedge \big(\forall i \big((e_i, t_i) \in s' \wedge i \neq j\big) : t_j < t_i\big)\Big)$

5.2 ADS

A sor absztrakt szerkezete elemeinek lineáris struktúrájaként jelenik meg. Az 5.2. ábra úgy szemlélteti a sor ADS-t, mint egy speciális lineáris gráfot. Az ábrán az a megjelenési forma is látható, ahogyan a sor gondolatainkban jelentkezik, ahogyan a szakmai kommunikációban hivatkozunk rá. A két absztrakt szerkezet nyilvánvalóan megfeleltethető egymásnak. Ez elképzelhető veremként is.



5.3 Statikus és dinamikus reprezentáció

5.3.1 Verem statikus reprezentációja

Aritmetikai ábrázolás:

- Egy max hosszú vektor: az elemek tömbje
- A verem tetejének mutatója: $head \in [1, max]$. $head = 0 \Leftrightarrow$ üres a verem.

5.3.2 Verem dinamikus reprezentációja

Láncolt reprezentáció. Minden elemben van egy mutató, ami a következő elemre mutat. A *head* pointer a legfelső elemre mutat.

5.3.3 Sor statikus reprezentációja

Aritmetikai ábrázolás:

- Egy max hosszú vektor: az elemek tömbje
- A sor első elemének mutatója: $heqd \in [1, max]$
- A sor első üres helyének mutatója: $tali \in [1, max]$

Ha a tail utoléri a headet, akkor nem tudjuk, hogy üres, vagy tele a vektor. Megoldás:

- Vagy egy jelző, ami mutatja, hogy tele van-e,
- vagy egy számláló, ami számolja az elemeket.

5.3.4 Sor dinamikus reprezentációja

Láncolt reprezentáció. Minden elemben van egy mutató, ami a következő elemre mutat. A *head* pointer a legelső elemre mutat, míg a tail mutató az utolsóra. Nincs szükség számlálóra.

5.4 Műveletek megvalósítása (algoritmusok)

5.5 Verem

Pszeudokódban:

```
v.empty
    -- uresre allitja a vermet
    v.head <- 0

v.isEmpty
    -- ures-e a verem? Logikai erteket ad vissza
    return (v.head=0)

v.isFull
    -- Csak statikusnal: Tele van a verem? Logikai erteket ad vissza
    return (v.head=max)

v.push(e)
    -- e-t beteszi a v verem tetejere
    if v.isFull
        error "tulcsordulas"
    else
        v.head <- v.head + 1
        v.elements[v.head] = e<-
    end if

v.pop
    -- kiveszi a legfelso elemet es visszaadja
    if v.isEmpty
        error "alulcsordulas"
    else
        v.head <- v.head - 1
        return v.elements[v.head + 1]
    end if

v.top
    -- lekerdezi a legfelso elemet
    if v.isEmpty
        error "alulcsordulas"
    else
        return v.elements[v.head]
    end if
```

5.6 Sor

Pszeudokódban:

```
s.empty
    -- uresre allitja a sort
    s.head <- 1
    s.tail <- 1
    s.empt <- true

s.isEmpty
    -- ures-e a sor? Logikai erteket ad vissza
    return s.empt

s.isFull
    -- Csak statikusnal: Tele van a sor? Logikai erteket ad vissza
```

```
    return ((not s.empt) and (s.head = s.tail))

s.in(e)
    -- e-t beteszi az s sor vegere
    if s.isFull
        error "tulcsordulas"
    else
        s.empt <- false
        s.elements[s.tail] <- e
        if s.tail = max
            s.tail <- 1
        else
            s.tail <- s.tail + 1
        end if
    end if

s.out
    -- kiveszi es visszaadja az s sor elso elemet
    if s.empt
        error "alulcsordulas"
    else
        e <- s.elements[s.head]
        if s.head = max
            s.head <- 1
        else
            s.head <- s.head + 1
        end if

        if s.head = s.tail
            s.empt <- true
        end if
    end if

s.first
    -- vissza adja az s sor elso elemet
    if s.empt
        error "alulcsordulas"
    else
        return s.elements[s.head]
    end if
```

5.7 Kifejezések lengyelformára alakításának és a lengyelforma kiértékelésének algoritmusai

A prefix formát hívjuk lenygel formának, a postfix formát pedig fordított lengyel formának. Az infix kifejezés átalakítható postfix kifejezéssé. Ennek két fontos előnye van:

- A műveleti jelek olyan formában követik egymást, amilyen sorrendben végre kell hajtani azokat.
- A műveleti jel közvetlenül az operandusai után áll.

5.7.1 Lengyelformára alakítás

Az x sorozatot balról jobbra dolgozzuk fel egy y sor, és egy s verem segítségével. Az operátorok, és nyitózárójelek a verembe kerülnek. Az operandusokat írjuk ki.

Talált operátornál legfeljebb a nyitózárójelig kivesszük a veremből az operátornál nagyobb prioritású operátorokat, és kiírjuk azokat, majd berakjuk a verembe a talált operátort.

Csukózárójel esetén írjuk ki a verem teteján lévő elemeket egészen a nyitózárójelig. A zárojeleket elvethetjük, hiszen postfix formában nincs rá szükség.

A kifejezés végére írva írjuk ki a veremben maradt operátorokat.

5.7.2 Kiértékelés

Az y sorozatot balról jobbra dolgozzuk fel, egy v verem segítségével. Az operandusok kerülnek a verembe.

Talált operátor esetén kiveszünk a műveletnek megfelelő mennyiségű operandust a veremből, és ezeken elvégezzük a műveletet, amit a talált operátor jelez, majd ennek az eredményét elhelyezzük a veremben.

Az eredmény a verem tetején lesz megtalálható.

6 Kupacok és prioritásos sor

6.1 Kupac definíció

Egy majdnem teljes (bináris, balról feltölthető) fa heap tulajdonságú, ha

- Üres, vagy
- A gyökerében lévő kulcs nagyobb, mint mindkét gyerekében, és mindkét részfája is heap tulajdonságú.

6.2 Reprezentáció

- Dinamikusan allokált csomópontok és mutatók (mint láncolt lista, fák, stb...)
- Használjunk egy tömböt. A majdnem teljes tulajdonság miatt:
  - a k csomópont gyerekei a $2k, 2k+1$-nél vannak
  - k szülője a $\frac{k}{2}$-nél van
  - ha $k > n$ a csomópont nem létezik.

6.3 Műveletek, algoritmusok (beszúrás, törlés,...)

Pszeudokódban:

```
empty
    -- az ures kupac letrehozasa
    T <- E[Maxmeret]
    Aktmeret <- 0

isEmpty
    -- ures a kupac?
    return (Aktmeret = 0)
```

6.3.1 Beszúrás

1. Helyezzük a következő üres pozícióra e-t
2. Buborékoztassuk fölfele, amíg nagyobb, mint a szülei

Maximum h csere, tehát a hatékonyság $O(\log_2 n)$.

6.3.2 Törlés

1. Az első lépésben a majdnem teljes tulajdonságot hozzuk helyre: felvisszük a legutolsó elemet a gyökérbe.
2. A kapott majdnem teljes fa nem kupac. A helyreállításhoz cseréljük fel a felhelyezett elemet a nagyobb gyermekével addig, amíg van olyan gyermeke, amely nagyobb saját magánál.

Összesen $O(\log_2 n)$.

6.4 Prioritásos sor

Ha a sorban lévő elemeknek van valamifajta rendezése, akkor a rendezési szempontot prioritásnak hívjuk, és úgy kell rendezni a sort, hogy a legnagyobb (legkisebb) priorítású elem legyen a legelső elem.

6.5 ADT axiomatikus specifikáció

E alaptípus feletti P elsőbbségi sor típus jellemzése:

- Egyszerűsítés: csak prioritásokat teszünk bele (N)
- Műveletek:
  - empty: $\rightarrow P$
  - isEmpty: $P \rightarrow L$
  - insert: $P \times N \rightarrow P$
  - delmax: $P \rightarrow P \times N$
  - max: $P \rightarrow N$
- Megszorítások: $delmax$ és max értelmezési tartománya $P \backslash \{\emptyset\}$
- Axiómák:
  - $isEmpty(empty)$
  - $isEmpty(p) \Rightarrow p = empty$
  - $\neg isEmpty(insert(p, n))$
  - $insert(delmax(p)) = p$
  - $max(p) = delmax(p)_2$
  - $delmax(p)_1 \neq empty \Rightarrow max(p) \geq max(delmax(p)_1)$
  - $n \geq max(p) \Rightarrow delmax(insert(p, n))_1 = p \wedge max(insert(p, n)) = n$
  - $n < max(p) \Rightarrow max(insert(p, n)) = max(p)$
  - $delmax(insert(empty, n)) = (empty, n)$
  - $max(insert(empty, n)) = n$

6.6 ADS

Az ADS szint esetünkben nem mutat egyértelműen a megfelelő ábrázolás irányába. Az ADS szintet, mivel ott még nem tudunk egyértelmű absztrakt szerkezetet kialakítani, lényegében átlépjük. Mégis, különleges szerepe és jelentősége lesz az ADS szintnek a kupac adatszerkezet tanulmányozásában.

A kupacot mindig tömbben tároljuk, de egy fastruktúrában gondoljuk el az elemeit! Nem lenne könnyű a kupac műveleteinek működését szigorúan a tömbön megtervezni vagy megérteni. A fa adatszerkezet azonban roppant szemléletes hátteret ad ehhez.

6.7 Reprezentáció

- Rendezetlen tömbökkel: a max műveletigény mindíg egy maxkeresés: $O(n)$
- Rendezett tömbökkel: Bár a hely megkeresése gyors: logaritmikus keresés, a minden jobbra léptetése miatt ez is $O(n)$.
- Heap adatszerkezettel $O(\log_2 n)$

7 Listák

7.1 Szekvenciális adatszerkezetek definíciója

Definíció: A szekvenciális adatszerkezet olyan $\langle A, R\rangle$ rendezett pár, amelynél $R \subseteq (A \times A)$ reláció tranzitív lezártja teljes rendezési reláció.

Az $R \subseteq (A \times a)$ reláció tranzitív lezártja az a reláció, mely tranzitív, tartalmazza R-et, és a lehető legkevesebb elemet tartalmazza.

Megadása:

1. $R' = R \cup (R \circ R)$
2. Ha $R \neq R'$, akkor folytonos 1-nél, különben $R' = R_T$, a tranzitív lezárt

A szekvenciális adatszerkezetben az egyes adatelemek egymás után helyezkednek el, van egy logikai sorrendjük. Az adatok között egy-egy jellegű kapcsolat van, minden adatelem egy helyről érhető el és az adott elemtől csak egy másik látható. Két kitüntetett eleme: az első és az utolsó.

Ez egy homogén adatszerkezet, azaz azonos típusú véges adatelemek sorozata. Jelölése: $L = (a_1, a_2, ..., a_n)$. Ha $n = 0$, akkor $L = ()$ az üres lista.

Minden eleme tartalmaz egy (vagy több) mutatót egy másik adatelemre - "következő" elem logikailag. A lánc első elemének címét a lista feje tartalmazza. A lánc végét az jelzi, ha az utolsó elem mutatója üres.

7.2 Statikus és Dinamikus reprezentáció

7.2.1 Fiy kapacitású ábrázolás

Tömbben ábrázoljuk, a logikai sorrendet indexek mutatják. A szabad helyek is megtalálhatóak a tömbön belül.

7.2.2 Dinamikus, láncolt ábrázolás

Az adatok száma nem ismert előre, így nem foglalunk feleslegesen helyet a memóriában. A feladat dinamikusan változik.

Minden elem röbb elemből áll, egy adatrészből és egy (vagy több) mutatóból. A muatók száma szerint több csoportba oszthatjuk a dinamikus láncolt listákat:

Egyirányú láncolt lista: Minden elem rámutat a következő elemre.

Kétirányú láncolt lista: Minden eleme rámutat az előző és a következő elemre is. A fej elem rámutat az első és az utolsó elemre is.

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

L a lista első elemének mutatója, *Akt* a lista aktuális elemének mutatója.

```
create
    -- ures lista letrehozasa
    L <- NIL
    Akt <- NIL

isEmpty
    -- ures a lista? Logikai ertekkel ter vissza.
    return (L=NIL)

first
    -- elso elemre all
    if L = NIL
        error
    else
        Akt <- L
```

```
    end if

next
    -- kovetkezo elemre all
    if Akt = NIL
        error
    else
        Akt <- (Akt->Pointer)

isEndOfList
    --  a lista vegen allunk-e? Logikai ertekkel ter vissza.
    return (Akt=NIL)

isLast
    -- az aktualis az utolso elem? Logikai ertekkel ter vissza
    return (not(Akt=NIL) and Akt->Pointer=NIL)

getValue
    -- aktualis elem erteke
    if Akt=NIL
        error
    else
        return (Akt->Adat)
    end if

setValue
    -- aktualis elem modositasa
    if Akt=NIL
        error
    else
        (Akt->Adat) <- e
    end if
```

7.3.1 Listaelem beszúrása

1. Deklarálás - p változó, Node típussal
2. Létrehozás - $new(p)$
3. Értékének megadása
4. Új elem befűzése a láncolatba:

- Beszúrás első elemként:

```
insertFirst(e)
    new(p)
    (p->Adat)<-e
    (p->Pointer)<-L
    L<-p
    Akt<-L
```

- Beszúrás utolsó elemként:

```
insertLast(e)
     new(p)
    (p->Adat)<-e
    (p->Pointer)<-NIL
    if L=NIL
        L<-p
        Akt<-L
```

```
        else
            u<-L
            v<-(L->Pointer)
            while not(v=NIL)
                u<-v
                v<-(v->Pointer)
            end while
            (u->Pointer)<-p
            Akt<-p
        end if
```

- Beszúrás az aktuális listaelem elé:

```
insertBefore(e)
    if Akt=NIL
        error
    else
        new(p)
        (p->Adat)<-e
        u<-L
        v<-(L->Pointer)
        while not(v=Akt)
            u<-v
            v<-(v->Pointer)
        end while
        (u->Pointer)<-p
        Akt<-p
    end if
```

7.3.2 Listaelem törlése

1. Törlendő elem megelőzőjének megkeresése
2. Láncolás megváltoztatása
3. Memóriából eltávolítás
4. *Akt* beállítása

```
removeAkt
    if Akt=NIL
        error
    else
        if Akt=L
            p<-L
            L<-(L->Pointer)
            Akt<-L
            delete(p)
        else
            p<-L
            while not(p->Pointer=Akt)
                p<-(p->Pointer)
            end while
            (p->Pointer)<-(Akt->Pointer)
            delete(Akt)
            Akt<-(p->Pointer)
        end if
    end if
```

7.4 Rendezés listával

- Buborék rendezés: A vektor elejétől kezdve felbuborékoztatjuk a legnagyobb elemet, utána megismételjük ugyanezt az egyel rövidebb vektoron... $O(n^2)$
- Maximum kiválasztásos rendezés: j elindul a max-tól. Kiválasztjuk a tömbből a maxot és kicseréljük a j. elemmel, majd csökkentjük j-t egyel. $O(n^2)$
- Beszúró rendezés: Egyszerű beillesztéssel egy-egy elemet a helyére visszükszünk. $O(n^2)$ (Rendezett listákon $O(n)$)
- Quicksort: pivot rendezések. $O(n \log n)$, de vezethet $O(n^2)$-hez.

8 Hierarchikus adatszerkezetek és bináris fák

8.1 Definíciók

Definíció: A hierarchikus adatszerkezet olyan $\langle A, R\rangle$ rendezett pár, amelynél van egy kitüntetett r elem, ez a gyökérelem, úgy, hogy

1. r nem lehet végpont: $\forall a \in A, \neg R(a, r)$.
2. $\forall a \in \big\{A \backslash \{r\}\big\}$ elem egyszer, és csak egyszer lehet végpont, azaz $\forall a \in \big\{A \backslash \{r\}\big\}$-hez $\exists! b \neq a, b \in A : R(b, a)$.
3. $\forall a \in \big\{A \backslash \{r\}\big\}$ elem r-ből elérhető.

Egy elemnek akárhány rákövetkezője lehet, de minden elemnek csak egyetlen megelőző eleme van, azaz az adatelemek között egy-sok jellegű kapcsolat áll fenn.

Definíció: A fa egy hierarchikus adatszerkezet, mely véges számú csomópontból áll, és igazak a következők:

- Két csomópont között a kapcsolat egyirányú, az egyik a kezdőpont, a másika végpont.
- Van a fának egy kitüntetett csomópontja, ami nem lehet végpont. Ez a fa gyökere.
- Az összes többi csomópont egyszer végpont.

Definíció: A fa rekurzív definíciója:

- A fa üres, vagy
- Van egy kitüntetett csomópontja, ez a gyökér.
- A gyökérhez 0 vagy több diszjunkt fa kapcsolódik. Ezek a gyökérhez tartozó részfák.

Az adatszerkezetben

- A fa csúcsai az adatelemeknek felelnek meg
- Az élek az adatelemek egymás utáni sorrendjét határozzák meg
- A gyökérelem a fa első eleme, amelynek nincs megelőzője
- Levélelem a fa azon eleme, amelynek nincs rákövetkezője
- Közbenső elem az összes többi adatelem
- Minden közbenső elem egy részfa gyökerének tekinthető, így a fa részfákra bontható
- Elágazásszám: közvetlen részfák száma
- A fa szintje a gyökértől való távolságot mutatja. A gyökérelem a 0. szinten van.
- A fa szintjeinek a száma a fa magassága
- Csomópontok foka: a csomóponthoz kapcsolt részfák száma
- Fa foka: a fában található legnagyobb fokszám
- Szülő: kapcsolat kezdőpontja
- Gyerek: kapcsolat végpontja
- Minimális magasságú az a fa, amelynek a magassága az adott elemszám esetén a lehető legkisebb
- Egy fa kiegyensúlyozott, ha csomópontjai azonos fokúak és minden szintjén az egyes részfák magasság a nem ingadozik többet egy szintnél.
- Rendezett fa: ha az egy szülőhöz tartozó részfák sorrendje lényeges.

Definíció: A bináris fa olyan fa, amelynek csúcspontjaiból maximum 2 észfa nyílik.

- Egy bináris fa akkor tökéletesen kiegyensúlyozott, ha minden elem bal-, illetve jobboldali részfájában az elemek száma legfeljebb egyel tér el.
- Teljesnek nevezünk egy bináris fát, ha minden közbenső elemének pontosan két elágazása van.
- Majdnem teljes: ha csak a levelek szintjén van esetleg hiány

8.2 Általános és bináris fák reprezentációi, bejárásai, műveletei

8.2.1 Reprezentáció

Általános fa esetén:

- bal gyermek, jobb testvér:
  - Minden csomóponthoz tartozik három mutató: bal gyermek, jobb testvér, és a szülő.
- Multilista:
  - Minden csomóponthoz tartozik egy lineáris lista, amelynek első eleme az adat, a többi a kapcsolatok listája

Korlátos általános fa esetén:

- Aritmetikai ábrázolás
- Láncolt, ahol minden csomópontnak pontosan k db mutatója, maximum k gyereke van.

Bináris fa esetén:

- Aritmetikai ábrázolás: szintfolytonosan egy tömbben
  - $ind(bal(c)) = 2 * ind(c)$
  - $ind(jobb(c)) = 2 * ind(c) + 1$
- Láncolt: mutató a bal és a jobb gyerekre, esetenként a szülőre is

8.2.2 Fák bejárása

A fa csomópontjaiban általában adatokat tárolunk. Ezeket valamilyen sorrendben szeretnénk egymás után elérni.

- **Gyökérkezdő (preorder)**: gyökér, majd a részfák bejárása sorban.
- **Gyökérvégző (postorder)**: részfák bejárása, majd a gyökér.
- **Gyökérközepű (inorder)**: csak bináris fáknál, bal részfa, gyökér, majd jobb részfa bejárása.

8.2.3 Műveletek

Lekérdező műveletek

- Üres-e - logikai értéket ad vissza
- Gyokérelem - visszaadja a gyökér adatelemet
- Keres(e) - adott e adatelemet keres, egy ilyen elem mutatóját adja vissza.

Módosító műveletek

- Üres - létrehoz egy üres fát
- Beszúr(e) - adott e elem beszúrása
- MódosítGyökér(e) - adott e adatelem lesz a gyökér
- Töröl(e) - törli az e adatelemet
- TörölFa - törli az összes elemet

9 Bináris keresőfák

9.1 Definíció

Definíció: A rendezési fa olyan bináris fa adatszerkezet, amelynek 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,
- a csúcs értéke kisebb, mint tetszőleges csúcsé a tőle jobbra lévő leszálló ágon.

A rendezési fa az őt tartalmazó elemek beviteli sorrendjét is visszatükrözi. Ugyanazokból az elemekből különböző rendezési fák építhetők fel.

Fontos tulajdonság: inorder bejárással a kulcsok rendezett sorozatát kapjuk. Egy n csúcsú bináris fa bejárása $O(n)$ ideig tart.

9.2 Műveletek és algoritmusok (beszúrás, törlés, rákövetkező, megelőző, stb)

9.2.1 Keresés

A T fában keressük a k kulcsú elemet. Ha létezik , akkor visszaadjuk az elem címét, egyébként NIL-t.

Rekurzív:

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

Iteratív:

```
keres(x,k)
    while not(x=NIL) and not(k=kulcs[x])
        if k<kulcs[x]
            x <- bal[x]
        else
            x <- jpbb[x]
        end if
    end while
    return x
```

9.2.2 Minimum keresés

Tegyük fel, hogy $T \neq NIL$. Addig követjük a baloldali mutatókat, amíg NIL mutatót nem találunk. Iteratív kód:

```
minimum(T)
    x <- gyoker[T]
    while not(bal[x]=NIL)
        x <- bal[x]
    end while
    return x
```

$O(h)$, ahol h a fa magassága.

9.2.3 Maximum keresés

Tegyük fel, hogy $T \neq NIL$. Addig követjük a jobboldali mutatókat, amíg NIL mutatót nem találunk. Iteratív kód:

```
maximum(T)
    x <- gyoker[T]
    while not(jobb[x]=NIL)
        x <- jobb[x]
    end while
    return x
```

9.2.4 Következő elem

x csúcs rákövetkezőjét adja vissza, ha van, különben NIL-t.

```
kovetkezo(T,x)
    if not(jobb[x]=NIL)
        return minimum(jobb[x])
    end if
    y <- szulo[x]
    while not(y=NIL) and x=jobb[x]
        x <- y
        y <- szulo[x]
    end while
    return y
```

9.2.5 Megelőző elem

x csúcs megelőzőjét adja vissza, ha van, különben NIL-t.

```
megelozo(T,x)
    if not(bal[x]=NIL)
        return maximum(bal[x])
    end if
    y <- szulo[x]
    while not(y=NIL) and x=bal[y]
        x <- y
        y <- szulo[x]
    end while
    return y
```

9.2.6 Beszúrás

A T bináris keresőfába a p csúcsot szúrjuk be. Feltételezzük, hogy a fában még nincs k kulcsú csúcs.

```
beszur(T,p)
    y <- NIL
    x <- gyoker[T]
    while not(x=NIL)
        y <- x
        if kulcs[p] < kulcs[x]
            x <- bal[x]
        else
            x <- jobb[x]
        end if
    end while
    szulo[p] <- y
    if y = NIL
        gyoker[T] <- p
```

```
    else if kulcs[p] < kulcs[y]
        bal[y] <- p
    else
        jobb[y] <- p
    end if
```

9.2.7 Törlés

A T bináris keresőfából a p csúcsot töröljük. Lehetőségek:

- Ha p-nek nincs gyereke: szülőjének mutatóját NIL-re állítjuk
- p-nek egy gyermeke van: A szülője és a gyermeke között építünk ki kapcsolatot
- p-nek két gyermeke van: átszervezzük a fát. Kitöröljük a rákövetkező elemet, aminek az értékét átrakjuk az értékét p-be. Így az előző két típusú esetet kapjuk.

```
torol(T,p)
    if bal[p] = NIL vagy jobb[p] = NIL
        y <- p
    else
        y <- kovetkezo(T, p)
    end if

    if not(bal[y] = NIL)
        x <- bal[y]
    else
        x <- jobb[y]
    end if

    if not(x = NIL)
        szulo[x] <- szulo[y]
    end if

    if szulo[y] = NIL
        gyoker[T] <- x
    else if y = bal[szulo[y]]
        bal[szulo[y]] <- x
    else
        jobb[szulo[y]] <- x
    end if

    if not(y = p)
        kulcs[p] <- kulcs[y]
    end if

    return y
```

10 AVL fák

10.1 Definíció

Jelölje az $m(f)$ az f bináris fa magasságát, ha x az f fa egy csúcsa, akkor $m(x)$ jelöli az x gyökerű részfa magasságát.

Egy bináris keresőfa AVL-fa, ha minden x csúcsára teljesül, hogy

$$|m(bal[x]) - m(jobb[x])| \leq 1.$$

Összefüggés az AVL fa pontszáma és magassága között: A h szintszámú minimális csúcsszámú AVL fa gyökerének egyik részfája $h-1$, a másik $h-2$ szintű. Rekurzió:

$$S_h = 1 + S_{h-1} + S_{h-2}.$$

Tétel: Egy h magasságú AVL fának legalább $F_{h+3} - 1$ csúcsa van. Ahol F a Fibonacci sorozat.

Bizonyítás: Legyen S_h a legkisebb h magasságú AVL fa mérete. Ismert, hogy $S_0 = 0$ és $S_1 = 1$, valamint hogy $S_h = 1 + S_{h-1} + S_{h-2}$.

Indukciót használva $S_h = F_{h+3} - 1$:

$$1 + F_{h+2} - 1 + F_{h+1} - 1 = F_{h+3} - 1.$$

Tétel: Ha F AVL fa, akkor $h(F) < 1.44 * \log_2(n+1)$, ahol n az F pontjainak számát jelöli.

10.2 Műveletek (AVL fában beszúrás utáni kiegyensúlyozás, törlés utáni kiegyensúlyozás) megfelelő esetválasztással

10.2.1 Újrakiegyensúlyozás beszúrásnál

Amikor beszúrunk egy elemet az AVL fa elromolhat. 4 különböző eset lehet, de ebből 2 tükrözi a másik 2-t.

Bevezetünk egy új attribútumot, a kiegyensúlyozási tényezőt:

- -1: bal részfa magasabb egyel
- 0: egyforma magasak a részfák
- 1: jobb részfa magasabb egyel

(++,+) szabály: A beszúrt elem az eredetileg is nagyobb jobb részfa jobb külső gyermekébe szúródott be. A megoldáshoz a jobb részfa gyökerénél kell balra forgatni.

Ennek a tükörképe a (–,-) szabály.

(++,-) szabály: A beszúrt elem az eredetileg is nagyobb jobb részfa bal gyermekébe szúródott be. A megoldáshoz a jobb részfa bal gyermekén kell egy jobbra forgatást, majd az új jobb részfa gyökerén kell egy balra forgatást végrehajtani.

A két forgatás között nincs feltétel vizsgálat. Ennek a tükörképe a (–,+) szabály.

A beszúrás költsége $O(\log_2 n)$.

10.2.2 Újrakiegyensúlyozás törlésnél

(++,+), (++,0) szabályok: Egy + csúcs kap mégegy +-t mert a bal részfájából kitöröltek egy csúcsot. Akkor a jobb részfa gyökerét egyel balra elforgatva megyszületik a megoldás.

Ennek a tükörképe a (–,-), (–,0).

(++,-) szabály: A törlés a bal részfában történik. Ennek a magassága $h+1$ volt és h lett. Az x gyökerű fa magassága $h+3$. A jobb részfa -os, ezért először az x gyökér jobb gyermekének a bal gyermekét jobbra forgatjuk, utána az x gyökér új jobb elemét balra forgatjuk.

Ennek a tükörképe a (–,+) szabály.

A törlés után a törölt elem szülőjétől felfelé haladva a gyökér felé újra számoljuk a csúcsok címkéit ezen az útvonalon. Ha ++, vagy – jön ki, akkor azt forgatással helyre állítjuk. Szélsőséges esetben az adott útvonal minden pontjában forgatni kell.

11 Piros-fekete fák

11.1 Definíció

A piros-fekete fa olyan bináris keresőfa, melynek minden pontja egy extra bit információt hordoz: ez a pont színe, amelynek értékei: piros vagy fekete.

A pontok színezésének korlátozásával biztosítható:

- piros-fekete fában bármely, a gyökértől levélig vezető út hossza nem lehet nagyobb, mint a legrövidebb út kétszerese.
- Megközelítőleg kiegyensúlyozottak.

A szokásos bináris fa minden olyan csúcsát, aminek kevesebb, mint két gyereke van, kiegészítjük további gyerekekkel. Ezek a speciális NIL értéket tartalmazó csúcsok lesznek a fa levelei. A valódi adatot tartalmazó csúcsok nem lesznek levelek.

Egy piros-fekete fa

- Minden csúcs piros, vagy fekete
- A gyökér, és minden levél fekete
- A levelek nem tartalmaznak adatot
- Ha egy csúcs piros, akkor mindkét gyereke fekete
- Minden csúcsból minden út egy levélig ugyanannyi fekete csúcsot tartalmaz.

Az x pont fekete-magassága $fm(x)$, az x pontból induló, levélig vezető úton található x-t nem tartalmazó fekete pontok száma.

A fa fekete-magassága: $fm(gyoker)$.

Lemma: Bármely n belső pontot tartalmazó piros-fekete fa magassága $\leq 2\log_2(n+1)$.

11.2 Műveletek (beszúrás utáni kiegyensúlyozás, törlés utáni kiegyensúlyozás) algoritmusai

11.2.1 Beszúrás

A beszúrt elemet pirosra színezzük. Amyg a beszúrt elem és szülője egyaránt piros, vagy nem értük el a gyökeret, helyreállítási lépéseket kell tenni.

1. eset: Ha a nagybácsi színe piros, cseréljük meg a nagyszülő és a szülő-nagybácsi színét.

2. eset: Ha x nagybácsija fekete, és x és x szülője piros és a szülő ellenkező oldali gyereke a nagyszülőnek, mint x a szülőnek, akkor forgatunk x és szülője körül kifelé, és legyen x x régi szülője.

3. eset: A nagybácsi fekete, x és szülője pirosak, és a szülő azonos oldali gyereke a agyszülőnek, mint x a szülőnek. Ebben az esetben kicseréljük a színeket a szülő és a nagyszülő között és forgatunk egyet a szülő és a nagyszülő mentén.

Addig megyünk felfelé, amíg x szülője piros.

11.2.2 Törlés

z csúcs törlése egy bináris keresőfából.

- Ha z-nek maximum egy gyereke van, akkor töröljük a szülőt, és a szülőt a gyerekkel kapcsoljuk össze.
- Ha z-nek két gyereke van, akkor a következő elem tartalmát átmásoljuk z-be. A következő elemnek vagy 0, vagy 1 gyereke van. Ezt kitöröljük.

Jelöljük y-al a ténylegesen kitörölt elemet. Ha y piros volt, akkor piros-fekete fa marad. Ha a kitörölt elem fekete volt, az elveszett fekete csúcsot pótolni kell.

Úgy képzeljük el, hogy a kitörölt elem gyerekének most van egy extra feketéje. Ha a gyerek színe fekete, akkor most dupla-fekete. Ha piros, akkor piros-fekete. Ezt az extra feketét visszük felfelé a fában, amíg:

- x egy piros-fekete csúcsra mutat. Ekkor ezt feketére színezzük és kész.
- x a fa gyökerére mutat. Ha dupla-fekete, eltávolítunk egy feketét és kész, mert ha a gyökérből viszünk el egy extra feketét, akkor egyforma marad a levelekhez vezető utakon a feketék száma.
- Olyan ponthoz érünk, ahol forgatásokkal és átszínezésekkel el tudjuk távolítani az extra feketét úgy, hogy nem sértjük meg a piros fekete tulajdonságokat.

Lehetséges esetek: Csak azokat vesszük figyelembe, ahol x a balgyerek, ahol jobb, az szimetrikusan adódik.

w jelöli x testvérét.

1. eset: x testvére w piros.

   w-nek van fekete fia. Így w és a x szülőjének színét felcserélve, balra forgatva w és a szülő mentén, az x új testvére fekete lesz. Így egy 2., 3., vagy 4. esetet kapunk.

2. eset: x testvére w fekete, és w mindkét gyereke fekete.

   w piros lesz, és a szülő kap egy extra feketét. Ezután folytatjuk a helyreállítási ciklust a szülőtől.

3. eset: x testvére w fekete, w bal gyereke piros, jobb gyereke fekete.

   w és $bal[w]$ színt cserélnek, majd ezen a két csúcs mentén jobbraforgatással elértük a 4. esetet.

4. eset: x testvére w fekete, w jobb gyereke piros.

   w-t szülő színűre, a szülőt feketére, és w jobb gyermekét feketére színezve balraforgatunk w és a szülő mentén.

12 2-3 fák, B fák

12.1 2-3 fák fogalma

Hatékony keresőfa konstrukció, ami a binárisnál bonyolultabb: egy nem-levélnek 2 vagy 3 gyermeke lehet. A 2-3 fa egy lefelé irányított gyökeres fa, melyre

- A rekordok a fa leveleiben helyezkednek el, a kulcs értéke szerint balról jobbra növekvő sorrendben.
- Egy levél egy rekordot tartalmaz.
- Minden belső csúcsból 2 vagy 3 él megy lefelé.
- Szerkezet: $[m_1, k_1, m2_2, k_2, m_3]$

Logikailag a belső csúcs két féle:

- 3 gyerek
  - Itt m_1, m_2, m_3 mutatók a csúcs részfáira, k_1, k_2 pedig U-beli kulcsok, melyre $k_1 < k_2$.
  - Az m_1 által mutatott részfa minden kulcsa kisebb, mint k_1.
  - Az m_2 által mutatott részfában minden kulcs k_1 és k_2 között van.
  - Végül az m_2 részfában k_2 a legkisebb kulcs
- 2 gyerek
  - $[m_1, k_1, m_2]$
  - Itt m_1, m_2 mutatók a részfákra, és k_1 pedig U beli kulcs.
  - Az m_1 által mutatott részfa mindne kulcsa kisebb, mint k_1.
  - Az m_2 részfában k_1 a legkisebb kulcs.
- A két típus közötti váltást csak logikalilag kezeljük.

Megjegyzés:

$$2^h < n < 3^h \Rightarrow h \leq \log_2(n).$$

12.2 Műveletek 2-3 fákban (keresés, beszúrás, törlés), műveletek költsége

12.2.1 Keresés

Összehasolnítások száma $0, 1, ..., h-1$ szinten 1 vagy 2, h magasságban 1. Tehát

$$T(n) < 2h + 1 < 2\log_2(n) + 1 = O(\log_2(n)).$$

12.2.2 Beszúrás

Kereséssel meghatározzuk a helyét. 2 eset:

1. A legalsó belső pontnak 2 gyereke van, elfér még egy harmadik is.
2. A legalsó belső pontnak 3 gyereke van: Csúcsvágásra van szükség és haladunk felfelé.
   - Ha valahol van kétgyermekes belső csúcs, akkor ott megáll a beszúrás.
   - Ha ezen az úton minden belső pontnak 3 gyereke volt, akkor a csúcsvágás felgyűrűzik a gyökérig. Ekkor fölfelé egy új gyökeret kell tenni, ami megnöveli a fa magasságát.

A művelet költsége: $T(n) = O(h) = O(\log_2(n))$

12.2.3 Törlés

Megkeressük a törlendő kulcsot. 2 eset van:

1. Ha a kulcs szülőjének 3 gyereke van: Elem törlése, esetleg szülő korrelálása.
2. Ha a kulcs szülőjének 2 gyereke van:
   (a) Ha a szülőnek van 3 gyerekes testvére, akkor az egyet átad.
   (b) Ha a szülőnek nincs 3 gyerekes testvére, akkor csúcsot vonunk össze.

Szükség esetén csúcsösszevonásokat hajtunk végre a gyökérig. Így a fa magassága csökkenhet.
A művelet költsége: $T(n) = O(h) = O(\log_2(n))$

12.3 B-fák definíciója

2-3 fa általánosításai. Probléma: nem az összehasonlytás időigényes, hanem az adatok kiolvasása. A fa csúcsai legyenek blokkok.

Feltételezés: A kulcshoz tartozó minden kísérő információ ugyanabban a csúcsban, mint a kulcs. Minden kulcshoz egy pointer arra a lapra, ahol a többi információ megtalálható. Ekkor feltesszük, hogy ha a kulcsot mozgatjuk, ezt a pointert visszük magunkkal.

Definíció:

1. Minden x csúcsnak a következő mezői vannak:
   - $n[x]$ - az x csúcsban tárolt kulcsok darabszáma
   - az $n[x]$ darab kulcs, a kulcsokat monoton növekvő sorrendben tároljuk.
   - $x.level$ egy logikai változó, ami igaz, ha x levél.
2. Ha egy x belső csúcs, akkor tartalmazza a mutatókat x gyerekeire
3. Az x kulcs értékek meghatározzák a kulcsértékeknek azon tartományait, amelyekbe a részfák esnek.
4. Minden levélnek azonos a mélysége, ez az érték a fa h magassága
5. A csúcsokban tárolható kulcsok darabszáma egy also és felső korlát. Ezeket a korlátokat egy $t \geq 2$ egész számmal lehet kifejezni, ezt a számot nevezzük a B-fa minimális fokszámának.

Minden nem gyökér csúcsnak legalább $t-1$ kulcsa van, minden belső csúcsnak így legalább t gyereke van. Minden csúcsnak legfeljebb $2t-1$ kulcsa lehet, tehát egy belső csúcsnak legfeljebb $2t$ gyereke lehet.

Egy csúcs telített, ha pontosan $2t-1$ kulcsa van. Csúcs szétvágásnál a középső elem körül kettő $t-1$ nagyságú nodeot kapunk. A középső elem felkerül a szülőbe.

Tétel: Ha $n \geq 1$ akkor minden olyan T n-kulcsos B-fára, amelynek h a magassága és minimális fokszáma $t \geq 2$, teljesül, hogy

$$h \leq \log_t \frac{n+1}{2}$$

12.4 Műveletek B fában (keresés, beszúrás, törlés)

12.4.1 Keresés

Minden csúcsban $n[x]+1$ lehetőséget kell megvizsgálni. Kvázi rekurzívan kikeresés. Keresésnél a végighaladott blokkokat is megkapjuk (pl.: firebase).

$O(t*h) = O(t*\log_t n)$.

12.4.2 Beszúrás

Két eset:

1. Kevesebb mint $2t-1$ hely van: sima beszúrás
2. $2t-1$ csúcs van: Csúcs szétvágás.

12.4.3 Törlés

Kulcsot nem csak levélből lehet törölni, hanem teszőleges csúcsból. Ügyelni kell arra, hogy a csúcs ne váljon túl kicsivé. Esetek

1. A k kulcs az x levélben van, akkor töröljük k kulcsot x-ből.
2. A k kulcs az x belső csúcsban van:
   (a) Ha x-ben a k-t megelőző gyereknek (y) legalább t csúcsa van, akkor megkeressük az y részfában a k-t közvetlenül megelőző k' kulcsot. Rekurzívan töröljük k'-t és helyettesítjük k-t k'-val az x-ben.
   (b) Szimetrikusan az előzőhöz, ha z gyerek következik az x-beli k után, és z-nek legalább t kulcsa van, akkor megkeressük a z gyökércsúcsú részfában a k-t közvetlenül követő k' kulcsot. Rekurzívan töröljük, majd helyettesítjük k-t k'-vel az $x - ben$.
   (c) Ha mind y-nak, mind z-nek csak $t-1$ kulcsa van, akkor egyesítjük k-t és z kulcsait y-ba, úgy, hogy x-ből töröljük a k-t és a z-re mutató pointert. Ekkor y-nak $2t-1$ kulcsa lesz. Ezután szabadítsuk fel z-t és rekurzívan töröljük k-t az y-ból.
3. Ha a k kulcs nincsen 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 a 3b szerint kell eljárnunk, mivel biztosítani kell, hogy annak a csúcsnak amelyikre lépünk legalább t csúcsa legyen. Ezután rekurzióval megyünk tovább
   (a) Ha $x.c_i$-nek csak $t-1$ csúcsa van, de van közvetlen testvére, amelyiknek legalább t csúcsa van, akkor gyakorlatilag forgatunk egyet úgy, hogy $x.c_i$-nek t csúcsa, közvetlen testvérének $t-1$ csúcsa legyen.
   (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 csúcsot x-ből, ebbe az egyesített kupacba, középre.

13 Hasító táblák

13.1 Fogalma

Adott a K kulcsok halmaza. A K elemeivel azonosított rekordok száma azonban várhatóan jóval kisebb, mint K számossága. Ekkor K-t egy alkalmas h függvénnyel leképezzük az ábrázolás alapját képző kisebb tartományra. Legyen ez a $[0, ..., M-1]$ intervallum.

A $h : K \to [0, M-1]$ függvényt hash függvénynek nevezzük. Mivel $M < |K|$, sőt általában $M << |K|$, ezért h nem lehet injektív, hanem szükségképpen fellép a kulcsütközés: van olyan $k \neq k'$, amire $h(k) = h(k')$.

A hasításos technikák feladata éppen az, hogy megoldást adjanak a kulcsütközésekre.

13.2 Közvetlen hozzáférésű táblák

A közvetlen címzésű táblák a legegyszerűbb eset. Tegyük fel, hogy az elemek kulcsai különböző egész értékek a $[0, m-1]$ intervallumon, és m nem túl nagy.

Használjuk magukat a kulcsértékeket, hogy kiválasszunk egy helyet a T közvetlen címzésű táblázatban, melyben az elemeket tároljuk.

Minden művelet (keresés, beszúrás, törlés) konstans idejű!

13.3 Hash függvények (egyszerű egyenletestől univerzálisig)

Megszorítások:

- Kulcsok egészek kell legyenek
- Kulcsok egyediek kell legyenek
- Kulcsok kis intervallumból kerülhetnek ki
- Hatékonyság miatt a kulcsok sűrűn kell legyenek

13.3.1 Egyszerű egyenletes hasítás

Ideális hash függvény. $P(k)$ annak a valószínűsége, hogy a k kulcs előfordul. Ha van m hely a hasító táblázatban, akkor egy egyenletes hash függvény $h(k)$, biztosítani fogja, hogy

$$\sum_{k|h(k)=0} P(k) = \sum_{k|h(k)=1} P(k) = ... = \sum_{k|h(k)=m-1} P(k) = \frac{1}{m}.$$

A kulcsok száma minden helyre azonos.

13.3.2 Egyenletes hash függvény

Ha a kulcsok a $[0, r)$-en egyenletesen szétszórt egészek, akkor

$$h(k) = floor\left(\frac{mk}{r}\right)$$

egy egyenletes has függvény. legtöbb hash függvény megadható oly módon, hogy a kulcsokat valamely r-re a $[0, r)$-re képezze le.

Most csökkentsük le az intervallumot $[0, m)$-re, ahol m a hash tábla egy elfoglalható mérete:

13.3.3 Osztó módszer

Az intervallum csökkentésre használjuk a mod függvényt:

$$h(k) = k \mod m.$$

M választására: általában a 2-hatványok nem jó választások, mert akkor az utolsó n bit fog kiválasztódni. a 2^n közeli prímek jó választásnak tűnnek.

13.3.4 Szorzó módszer

Szorozzuk meg a kulcsot egy A konstanssal $0 < A < 1$, majd vegyük ki belőle a tört részt, majd szorozzuk meg m-el:

$$floor(m * (kA - floor(kA))).$$

Most m nem kritikus, és 2-hatvány választható. $A = \frac{1}{2}(\sqrt{5} - 1)$ jó választásnak tűnik.

13.3.5 Univerzális hashelés

Legyen H hasító függvények egy véges halmaza, melynek egy adott K kulcsuniverzumot a $[0, m)$ tartományba képeznek le.

A H-t univerzálisnak hívjuk, ha $\forall x, y \in K, x \neq y$ kulcspárra azoknak a $h \in H$ hasító függvényeknek a száma, amelyre $h(x) = h(y) = \frac{|H|}{m}$.

Ez azt is jelenti, hogy egy véletlenül választott $h \in H$ hasító függvényre $\forall x, y \in K, x \neq y$ külcsok közötti kulcsütközési valószínűség pontosan $\frac{1}{m}$.

Tervezés:

- Válasszunk egy $p > m$ prímszámot.
- $Z_p = \{0, 1, ..., p - 1\}$, ls $Z_p^* = \{1, 2, ..., p - 1\}$

$$\forall a \in Z_p^2, b \in Z_p, \qquad h_{a,b}(k) = \big((a * k + b) \mod p\big) \mod m.$$

Az ilyen függvények osztálya:

$$H_{p,m} = \{h_{a,b} : a \in Z_p^*, b \in Z_p\}$$

Tétel: A hasító függvények fenti egyenlőségekkel definiált $H_{p,m}$ osztálya univerzális.

13.4 Kulcsütközések feloldása: láncolt hashelés, túlcsordulási terület, nyílt címzés

13.4.1 Láncolt hashelés

Minden táblaelemhez egy láncolt listát rendelünk.

Keresés: kiszámítjuk $h(i)$-t, kikeressük az i kulcsú elemet a $T[h(i)]$ listában. Ha $NULL$-t találunk, a kulcs nincs a listában.

Beszúrás: kiszámítjuk $h(x.kulcs)$-t, majd beszúrjuk a $T[h(x.kulcs)]$ lista elejére.

Törlés: Hasonlóan a $T[h(x.kulcs)]$ listából.

Hatékonyság: $\alpha = \frac{n}{m}$ az egy láncba fűzött elemek átlagos száma. Lehe kisebb, egyenlő és nagyobb mint 1. Legrosszabb esetben minden egy elem helyére képződik le: $O(n)$.

13.4.2 Túlcsordulási terület

A láncolt listát a tábla egy speciális helyén hozzuk létre - ez a tulcsordulási terület.

Beszúrás: Tegyük fel, hogy $h(k) = h(j)$, és k lett először tárolva. A j beszúrásánál megtaláljuk k-t a $h(j)$ helyen. Megkeressük az első szabad helyet a túlcsordulási területen, oda betesszük j értékét és k "pointerét" erre irányítjuk.

Keresés: Ugyan az, mint a láncolt listánál.

Törlés: Lehet a lista első, vagy utolsó elemét idetenni, de lehet új állapotot bevezetni.

13.4.3 Nyílt címzés

Kiterjesztjük a hash függvény értelmezési tartományát az úgynevezett kipróbálási számmal:

$$h : K \times \{0, ..., m-1\} \to \{0, ..., m-1\}.$$

Megköveteljük, hogy $\forall k$ kulcsra a $(h(k,0), h(k,1), ..., h(k,m-1))$ kipróbálási sorozat a $\{0, ..., m-1\}$ egy permutációja legyen, Így előbb utóbb minden hely szóbajön.

Keresés: Egy elem keresésénél végigmegyünk a táblázaton, amíg meg nem találjuk, vagy el tudjuk dönteni, hogy nincs benne a táblában.

Beszúrás: Kipróbáljuk az összes helyet, amíg nem találunk egy üreset - ez a beszúrandó kulcs függvénye, nem a $0, ..., m-1$ felsorolás..

Törlés: Egy elem törlése bonyolúltabb, mert ha NIL lenne, akkor megállna a keresés. Ezért egy TÖRÖLT flaget kap ez a cella. Keresésnél átugorjuk, beszúrásnál beszúrunk.

14 A rendezés

14.1 Feladata, definíciója

A rendezés feladata, egy n számot tartalmazó $(a_1, a_2, ..., a_n)$ sorozatat olyan permutációjának $(a'_1, a'_2, ..., a'_n)$ megtalálása, hogy $a'_1 \leq a'_2 \leq ... \leq a'_n$, ahol $\leq$ valamilyen rendezési reláció.

Általánosabban: Legyen K egy teljesen rendezett halmaz, a kulcsok halmaza. Legyenek $T_i, i \in [1, m]$-k tetszőleges típusok.

$$E = K \times T_1 \times ... \times T_m.$$

A cél egy $S \in E^*$ rendezése. Legyen $n = |S|$, S rendezett $\Leftrightarrow \forall i \in [1, n-1] : S_i.kulcs \leq S_{i+1}.kulcs$.

- Előfeltétel: $S = S' \in E^*$
- Utófeltétel: S rendezett, és $S \in Perm(S')$

14.2 Rendezők osztályozása

1. $m = 0$: skalár rendezők
2. $m = 1$: rekord rendezők
3. Belső rendezők: központi memória + indexelés
4. Külső rendezők: háttértárolókon
5. Összehasonlításos rendezők: kulcsok értékét hasonlítjuk
6. Edényrendezők: kulcsok értéke szerint szétrakjuk
7. Helyben rendezők: segéd memória konstans
8. Nem helyben rendezők
9. Stabil rendezők: Azonos kulcsú rekordok sorrendje nem változik
10. Nem stabil rendezők
11. Előrendezéshez illeszkedő és nem illeszkedő rendezők: kevesebbet dolgozik-e ha a soroza előrendezett
12. Használt adatszerkezet szerint:
    - lineáris adatszerkezet
    - fa
13. Módszer szerint: például összehasonlításos rendezőknél:
    - Maximális elemet kiválasztó
    - Csererendezők
    - Egy elemet helyre vivők
    - összefuttatásos rendezők

14.3 Négyzetes rendezés algoritmusa és időigénye (bubble sort, beszúró rendezés, maximum kiválasztásos rendezés)

14.3.1 Buborék rendezés

A vektor elejétől kezdve felbuborékoltatjuk a legnagyobb elemet. Utána ugyanezt tesszük az egyel rövidebb vektorral, stb.

```
j <- n
while j >=2
    i <- 1
    while i <= j-1
        if A[i]>A[i+1]
            Csere(A[i],A[i+1])
        end if
        i <- i + 1
    end while
    j <- j-1
end while
```

Legrosszabb időbonyolultság: $O(n^2)$, legjobb időbonyolultság: $O(n)$, átlagos időbonyolultság: $O(n^2)$.

14.3.2 Beszúró rendezés

Egyszerű beillesztéssel egy-egy elemet a helyére viszünk a már rendezett sorozatban. Ha tömbben tároljuk, akkor a mozgatások száma is fontos.

```
j <- 1
while j<=n-1
    w <- A[j+1]
    i<-j
    while i>=1 and A[i]>w
        A[i+1] <- A[i]
        i <- i-1
    end while
    A[i+1] <- w
    j <- j+1
end wile
```

Legrosszabb időbonyolultság: $O(n^2)$, legjobb időbonyolultság: $O(1)$, átlagos időbonyolultság: $O(n^2)$.

14.3.3 Maximum kiválasztásos rendezés

Feltételezzük, hogy a tömb jobb széle már rendezve van. minden lépésben kiválasztjuk az $A[1...j]$ résztömb maximális elemét és kicseréljük a j helyén lévővel, majd j-t csökkentjük. Kezdetben j értéke n

```
j <- n
while j>=2
    Max(A[1..j],ind)
    Csere(A[ind],A[j])
    j <- j-1
end while
```

Legrosszabb időbonyolultság: $O(n^2)$, átlagos időbonyolultság: $O(n^2)$.

15 Quicksort

15.1 Quicksort algoritmusa

Két fézis:

1. Partíciós fázis
2. Rendezési fázis

Válasszunk egy olyan pivotot, és válasszuk rendezzük a sorozatot úgy, hogy minden elem tőle jobbra nagyobb, minden elem tőle balra kisebb legyen.

Ezt az algoritmust alkalmazzuk mindkét félre.

Partícionálás: A résztömb amin dolgozunk, az alsó és a felső indexek között található. A tömb két széléről indul szemben a két indexelés, a jobb és a bal.

A két indexet egymással szemben mozgatjuk addíg, amíg találkoznak egymással. A bal jelzőt addig visszük jobbra, amíg nem igaz, hogy $bal \leq pivot$. A jobb jelzőt addig visszük, amíg nem igaz, hogy $jobb \geq pivot$.

Ha a két index nem kerülte ki egymást, akkor meg kell cserélni a két elemet, mivel a pivot rossz oldalain állnak. Ezt addig ismételjük, amíg szembetalálkoznak. Utolsó lépésként berakjuk a pivotot a jobb és bal értékek közé.

Ezt rekurzívan folytathatjuk a pivot mindkét oldalán.

```
Gyorsrendezes(A, also, felso)
    if also<felso
        q=Feloszt(A, also, felso)
        Gyorsrendezes(A, also, q-1)
        Gyorsrendezes(A, q+1, felso)
    end if

Feloszt(A, also, felso)
    pvt <- A[also]
    bal <- also
    jobb <- felso
    while bal<jobb
        while A[bal] <= pvt and bal<felso
            bal <- bal + 1
        end while
        while A[jobb] >= pvt and jobb>also
            jobb <- jobb - 1
        end while
        if bal<jobb
            Csere(A[bal],A[jobb])
        end if
    end while
    A[also] <- A[jobb]
    A[jobb] <- pvt
    return jobb
```

15.2 Műveletigénye

Partícionálás: $O(n)$, uralkodás: $O(\log_2 n)$, tehát összesen $O(n \log_2 n)$. De van egy gond: Ha az adatok már rendezettek, akkor a futási idő $n * O(n) = O(n^2)$. De ha megváltoztatjuk a pivot választási stratégiát

15.3 Pivot választási stratégia jelentősége

15.3.1 Középső választása

Ha a pivotot a középső elemnek választjuk, az jó a rendezett elemek esetén, de megmutatható, hogy az sem mindíg jó választás.

15.3.2 3-ból a középső választása

Válasszunk ki 3 pivotot (például az elsőt, a középsőt és az utolsót), majd azok közül az érték szerinti középsőt használjuk.

Mivel a rendezett adatok elég gyakoriak, ez jó stratégia.

15.3.3 Véletlen pivot

Pivot véletlen kiválasztása, minden pozíciónál más. Átlagosan rendezett adatokat jól osztja szét. Fontos követelmény: a kiválasztás $O(1)$ idő kell legyen.

Sohasem garantálható az $(On \log_2 n)$, sőt a tetszőleges stratégia vezethet $O(n^2)$-hez is.

16 Kupacos rendezés

16.1 Kupac definíció

Egy majdnem teljes (bináris, balról feltölthető) fa heap tulajdonságú, ha

- Üres, vagy
- A gyökerében lévő kulcs nagyobb, mint mindkét gyerekében, és mindkét részfája is heap tulajdonságú.

16.2 Reprezentáció

- Dinamikusan allokált csomópontok és mutatók (mint láncolt lista, fák, stb...)
- Használjunk egy tömböt. A majdnem teljes tulajdonság miatt:
  - a k csomópont gyerekei a $2k, 2k+1$-nél vannak
  - k szülője a $\frac{k}{2}$-nél van
  - ha $k > n$ a csomópont nem létezik.

16.3 Műveletek, algoritmusok (beszúrás, törlés,...)

Pszeudokódban:

```
empty
    -- az ures kupac letrehozasa
    T <- E[Maxmeret]
    Aktmeret <- 0

isEmpty
    -- ures a kupac?
    return (Aktmeret = 0)
```

16.3.1 Beszúrás

1. Helyezzük a következő üres pozícióra e-t
2. Buborékoztassuk fölfele, amíg nagyobb, mint a szülei

Maximum h csere, tehát a hatékonyság $O(\log_2 n)$.

16.3.2 Törlés

1. Az első lépésben a majdnem teljes tulajdonságot hozzuk helyre: felvisszük a legutolsó elemet a gyökérbe.
2. A kapott majdnem teljes fa nem kupac. A helyreállításhoz cseréljük fel a felhelyezett elemet a nagyobb gyermekével addig, amíg van olyan gyermeke, amely nagyobb saját magánál.

Összesen $O(\log_2 n)$.

16.4 Rendezési algoritmus a kupaccal

Beszúrjuk az elemeket a kupacba, majd amíg ki nem ürül kivesszük a kupacból a maximális elemet, és a rendezett eredménysorba rakjuk.

```
k.empty
while not(s.isEmpty)
    k.insert(s.out)
end while
while not(k.isEmpty)
    s.in(k.delmax)
end while
```

16.5 Műveletigénye

A kupacok hatékony rendezőt adnak.

- n elem beszúrása: $O(n \log_2 n)$
- n elem eltávolítása: $O(n \log_2 n)$

összesen $O(n \log_2 n)$, ignorálva a 2-es konstanst.

17 Összefuttatásos rendezés

17.1 Algoritmusa

17.1.1 Összefuttatás:

Alap: két rendezett sorozat összefuttatása. Legyen $A[1...k]$ és $B[1..m]$ két rendezett vektor. Ekkor $C[1..k+m]$, és két mutatót indítunk a két rendezett vektorban az elejétől. Mindne lépésben összehasonlítjuk a két mutató értékét, és a kisebbet (nagyobbat) belerakjuk az eredményvektorba, valamint a kiválasztott mutatót léptetjük egyel.

Ez addig megy, amíg mindkét mutató el nem éri a megfelelő vektorának a végét.

```
ai <- 1
bi <- 1
ci <- 1
while ai<=k and bi<=m
    if A[ai] < B[bi]
        C[ci] <- A[ai]
        ai <- ai + 1
    else if A[ai] > B[bi]
        C[ci] <- B[bi]
        bi <- bi + 1
    else
        C[ci] <- A[ai]
        ai <- ai + 1
        ci <- ci + 1
        C[ci] <- B[bi]
        bi <- bi + 1
    end if
    ci <- ci + 1
end while
C-be betesszuk a maradekot
```

17.1.2 Összefuttatásos rendezés

Az üres és az egyelemű sorozat biztosan rendezett. Ha $S[1..n], n > 1$ tömböt akarjuk rendezni, akkor szétvágjuk két részre, és ezeket külön-külön rendezzük, majd összefuttajuk.

Általánosan:

```
MergeSort(S)
    if Length(S) > 1
        Split(S,S1,S2)
        MergeSort(S1)
        MergeSort(S2)
        Merge(S1,S2,S)
    end if
```

Tömbre:

```
MergeSort(A,k,v)
    if k<v
        h <- (k+v)/2
        MergeSort(A,k,h)
        MergeSort(A,h+1,v)
        Merge(A[k,h],A[h+1,v],A[k,v])
    end if
```

Az egész tömböt: $MergeSort(A, 1, n)$.

17.2 Műveletigénye

- Összefüttatás: $O(n-1)$, ahol $n = k + m$
- Egyenlő szétvágásnál a bináris fa majdnem teljes, általánosan $h = ceil(\log_2 n)$.

Így összesen $MO_{MS}(n) \leq (n-1) * ceil(\log_2 n) = O(n \log_2 n)$.

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

Az összehasonlítások tekinthetők döntési fának, amik a rendezés során történt összehasonlításokat ábrázolják. Tegyük fel, hogy minden elem különböző. Egy belső csúcsot egy a_i, a_j párral jelölhetünk, ahol $1 \leq i, j \leq n$. A levelek egy-egy permutációnak feleltethetőek meg.

Tétel: Bármely n elemet rendező döntési fa magassága $\Omega(n \log_2 n)$

Bizonyítás: A fának $n!$ levele van, ezek a permutációk. A h mélységű bináris fa leveleinek a száma $\leq 2^h$, ezért

$$n! \leq 2^h \quad \Rightarrow \quad h \geq \log_2(n!),$$

felhasználva, hogy a log függvény monoton nő.

A stirlig formulával:

$$n! \cong \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \Rightarrow \quad n! > \left(\frac{n}{e}\right)^n.$$

Behelyettesítve:

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

Következmények:

1. A kupacrendezés és az összefésüléses rendezés aszimptotikusan optimális összehasonlító rendezések.
2. Nem létezik lineáris idejű összehasonlításos rendezés.

17.4 Batcher-féle páros-páratlan összefésüléses rendezés - algoritmus szövegesen

Ha tudjuk párhuzamos processzorokon futtatni a MergeSortot, akkor az felgyorsulna $O(\log_2 n)^2$-re. Ez lesz a PPMerge. Hasonlóan a MergeSorthoz egy tömb két rendezett feléből összefésüléssel előállítja a tömb rendezett tartalmát rekurzívan.

Az új összefésülés:

1. A páratlan és B páros indexű elemeit fésüli össze U-ba,
2. A páros és B páratlan indexű elemeit fésüli össze V-be.
3. Az U és V sorozatokat nem kell már összefésülni, hanem elég csak az egymás alatti u_i, v_i párokat 1-1 összehasonlítással a helyes sorrenbe rakni. Amennyiben az U és V sorozat hossza eltérő, az utolsó elemet összehasonlítás nélkül kell áttenni.

Párhuzamosítás: 1. és 2. párhuzamosan végezhető.

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

- Egy 2 hosszú tömböt még szétvágunk, két 1 hosszú tömbre, és azokra meghívjuk az összefésülést. Ekkor $k = 1, m = 1$
- Egy egy hosszú tömböt azonban már nem fésülünk össze, hanem felismerjük, hogy az már rendezett, így 0 összehasonlítást igényel. Ekkor $k = 1, m = 0$

17.5 Helyességének belátása

Belátjuk, hogy a leírt eljárás azt a helyes rendezett sorozatot eredményezi, amelyet

$$C = c_1 < c_2 < ... < c_{k+m}$$

jelöl. Esetszétválasztással

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

Általában, ha $1 \leq i \leq floor(\frac{k+m}{2})$,

$$c_{2i-1} = \min\{u_1, v_1\}, \quad c_{2i} = \max\{u_1, v_1\}$$

Ha $k + m$ páratlan, akkor azt is be kell látni, hogy c_{k+m} is helyesen képződik, hiszen erre a képlet nem vonatkozik.

Bizonyítás:

1. Belátjuk, hogy a C sorozatban bármely páros indexű tagig elmenve $c_1, ..., c_{2k}$ között ugyanannyi u_i szerepel, mint v_j, vagyis az U és a V sorozatok szinte cipzár-szerűen építi a C-t.

   - A C sorozat a rendezett A és B sorozatok összefüggésével adódik. Tegyük fel, hogy A-ból az $a_1, ..., A_s$ elemek, a B-ből $b_1, ..., b_{2k-s}$ elemek jönnek összefűzéssel a C sorozat első $2k$ elemébe.
   - $\{c_q, ..., c_{2k}\} = \{a_1, ..., a_s\} \cup \{b_1, ...b_{2k-s}\}$
   - U-ba kerülnek a páratlan indexű elemek A-ból, ezek száma $ceil(\frac{s}{2})$.
   - V-be kerülnek a páros indexű elemek A-ból, ezek száma $floor(\frac{s}{2})$.
   - U-ba kerülnek a páros indexű elemek B-ból, ezek száma $floor(\frac{2k-s}{2})$.
   - V-be kerülnek a páratlan indexű elemek B-ból, ezek száma $ceil(\frac{2k-s}{2})$.
   - Az állítás egyenértékű azzal, hogy

     $$ceil\left(\frac{s}{2}\right) + floor\left(\frac{2k-s}{2}\right) = floor\left(\frac{s}{2}\right) + ceil\left(\frac{2k-s}{2}\right)$$

     Ez nyilván igaz ha s páros. Ha páratlan, akkor az a kérdés, hogy fennáll-e, hogy

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

2. Eszerint:

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

   A két halmaz kivonásával:

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

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

Tehát U és V már egy párhuzamos lépésben összefésülhető.

18 Edényrendezés és radix rendezés

18.1 Edényrendezés

18.1.1 Algoritmus

Tegyük fel, hogy tudjuk, hogy a bemenő elemek ($A[1...n]$ elemei) egy m elemű U halmazból kerülnek ki. Lefoglalunk egy U elemeivel indexelt B tömböt. A B segédtömb elemei lehetnek például láncolt listák.

Első fázis: Végigolvassuk az A-t és az $A[i]$ elemet a $B[A[i]]$ lista végére fűzzük.

Második fázis: Végigmegyünk a B-n elejétől a végéig növő sorrendben, és a $B[i]$ listák tartalmát visszaírjuk A-ba.

18.1.2 Lépésszám

- B létrehozása: $O(m)$
- Első fázis: $O(n)$
- Második fázis: $O(n+m)$
- Összesen: $O(n+m)$, ha $m \leq c * n$, akkor $O(n)$.

18.1.3 Szokásos implementáció

```
Edenyrendezo(S)
    E0,E1,...EM-1 <- eps, eps, ..., eps
    while not(S=eps)
        Out(S,x)
        In(EPhi(x),x)
    end while
    S <- Osszefuzes(E0,E1,...,EM-1)
```

18.2 Radix rendezés

18.2.1 Algoritmus

Tegyük fel, hogy a kulcsok összetettek, több komponensből állnak, $(t_1, ...t_k)$ alakú szavak, ahol a t_i komponens az L_i rendezett típusból való, a rendezés lexikografikus. Radix rendezés:

1. Rendezzük a sorozatot az utolsó, a k-adik komponensek szerint edényrendezéssel.
2. A kapott eredményt rendezzük a $k-1$-edik komponensek szerint edényrendezéssel, stb...

Fontos, hogy az edényrendezésnél az elemeket a lábádab mindig a lista végére tegyük. (Stabil rendezés)

18.2.2 Lépésszám

$2 * d$-szer végigmegyünk az S sorozaton, így

$$T(n) = O(d * |S|).$$

18.2.3 Szokásos implementáció

```
Radix(S)
    i <- 1
    while i<d
        Edenyrendezo(S,Phii)
        i <- i + 1
    end while
```

Szokásos implementáció:

- S fejelemes láncolt lista
- Az edényekez egy head és egy tail mutató ábrázolja
- A szétrakás és az összefűzés az elemek láncolásával is megoldható
- Összefűzéskor nem kell az egyes edények részlistáit végigolvasni, hanem egy darabban lehet őket láncolni

18.3 Leszámláló rendezés algoritmusa

Tegyük fel, hogy van n db bemeneti elem, ezek mindegyike 1 és k közötti egész szám. Alapötlet: meghatározzuk minden egyes x bemeneti elemre azoknak az elemeknek a számát, amelyek kisebbek mint az x, ezután x-et közvetlenül a saját poziciójára lehet helyezni.

Legyen a bemenet az $A[1...n]$ tömb, a kimenet a $B[1...n]$ tömb. Szükség van mégegy $C[1...k]$ tömbre átmeneti munkaterületként.

1. Végigmegyünk az A-n és ha egy elem értéke i, akkor megnöveljük $C[i]$ értékét egyel.
2. $\forall i \in [1, k]$ meghatározzuk, hogy olyan bemeneti elem van, amelynek az értéke $\leq i$.
3. $\forall i \in [n, 1]$ A[i]-t betesszük B megfelelő pozíciójába - ezt a C-ből állapítjuk meg. Ha betettük, akkor $C[A[i]]$ értékét csükkentjük, így a következő vele egyenlő elem már elé kerül. vagyis stabil lesz a rendezés.

18.4 Lépésszáma

Futási idő:

- 2 for ciklus: $O(k)$
- 2 for ciklus: $O(n)$
- Így a teljes időigény $O(k + n)$, ha $k = O(n)$, akkor a teljes rendezés $O(n)$.

19 Külső rendezések

Ha az adatoka háttértérban vannak, akkor a futási idő döntő részét az I/O utasítások teszik ki. Egy I/O egysége az 1 blokk, ami $k * 512$ byte, ahol például $k = 1024, 2048, ...$

A hatékonyságot a szükséges blokk I/O-k számában mérjük. Külső rendezésre igazából csak a merge sort alkalmas.

19.1 2 segédfájlos (4 fájlos)

Merge sort külső tárakon: Adott egy S szekvenciális input file, amely n blokkból áll, minden blokkban adott számú rekorddal. A blokkok tartalma rendezetlen.

Az összefűzést iteratív módon végezzük, úgy, hogy az egyes menetek végén egyre nagyobb darabok, vagyis egyre több szomszédos blokk lesz rendezett.

Az összefűzés meneteként váltakozva az A, B illetve a C, D fileokba végezzük, végül a teljesen rendezett eredményt S-be írjuk. A különböző menetekben lesz 2 output file, mert az összefuttatás eredményét az egyet ide, egyet oda elv alapján írjuk ki.

1. Menet: Beolvassuk S rendezetlen blokkjait, valamilyen belső rendezővel rendezzük, majd kiírjuk felváltva A-ba, illetve B-be.
2. Sorban beolvassuk A és B 1-1 blokkját. Ezek rendezettek. Összefésüljük őket és a rendezett két blokk hosszú adatot felváltva C-be, illetve D-be írjuk.
3. Az A utolsó blokkjának nincs párja, így azt kiírjuk C-be. A C és D fileban rendezett részek hossza 2 blokk, illetve a maradék esetében 1 blokk.

Ezt folytatjuk addig amíg már csak 2 nem összefűzött rendezett blokk van. Ez lehet egy hosszú, és egy egyszem maradék elem.

Ennek a két sorozatnak az összefésülését visszaítjuk S-be. Ha a központi memória mérete kotlátozott és nem képes befogadni az egyre növekvő 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.

$$2^{k-2} < n \quad \Rightarrow \quad k-2 < \log_2 n \leq k-q \quad \Rightarrow \quad k-1 = ceil(\log_2 n)$$

A menetek k száma: $k = ceil(\log_2 n) + 1$. Az összes blokk I/O száma:

$$2 * n * (ceil(\log_2 n) + 1).$$

19.2 3 fájlos

Számítások:

- 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.

Összes blokk I/O száma:

$$2 * n * \left(ceil\left(\log_2 \frac{n}{m} \right) + 1 \right).$$

Lehet az is, hogy több mint kétfelé fésülünk: ha az S file mellett nem $2 * 2$, hanem általában $2 * m$ fájllal dolgozunk, akkor ezzel a ráfordítással hatékonyabb eljárás nyerhető.

A számolás:

- 1. menet eredménye: 1 hosszú futamok
- 2. menet eredménye: m hosszú futamok
- ...
- (k-1). menet eredménye: m^{k-2} hosszú futamok
- k. (utolsó) menet eredménye: $\leq m^{k-1}$ hosszú egyetlen futam.

Összes blokk I/O száma:

$$2 * n * \left(ceil\left(\frac{\log_2 n}{\log_2 m} \right) + 1 \right).$$

Érdekes algoritmushoz jutunk, ha az m-felé fésülésnél nem $2m$ db filet, hanem csak $m + 1$ filet használunk. Ezt $m = 2$ esetére nézzük meg, ami az eredeti eset. Ekkor 3 filet használunk, A, B, C-t.

1. Az első menetben szétszórjuk S immár rendezett blokkjait A-ba és B-be.
2. A második menetben elkezdjük A és B blokkjainak az összefésülését, de most csak 1 fájl tudja fogadni az eredményt, a C file. Ezért C-be fésüljük össze egészen addig, amyg A és B egyike ki nem ürül. Ekkor új menetet kezdünk a két nem üres fileal.

Ez önmagában elég lassú lehet, mert lehet, hogy valamelyik menet végi két futamszámnak 1 a külömbsége.

Ötlet Fibonacci: Első menetben írjunk annyi futamot A-ba és B-be, mint a Fibonacci sorozat klt zomszédos eleme, és lépkedjünk visszafelé a sorozaton az egyes menetekben.

Belátható, hogy így fut le a leggyorsabban.