Problémaosztalyok, algoritmusok

Adatszerkezetek

Adatszerkezetek

- Az adatokat két nagy csoportba oszthatjuk, egyszerű és összetett adatokra.
- Az egyszerű adatot egy érték jellemzi, részekre nem bontható. Ilyenek a numerikus értékek (egész, valós), a logikai értékek , illetve a karakterek.
- Az összetett adatok egyszerű adatokból épülnek fel, melyek között valamilyen strukturális kapcsolat van. Ilyenek a string, a rekord, a tömb, a halmaz.
- Absztrakt adatszerkezetek:
  - a lineáris listák,
  - hasító táblázatok és a fák, bináris fák.

Lineáris listák

- **Definíció**: Egy lineáris lista n >= 0 számú R(1), R(2), ... R(n) (azonos típusú) rekordok halmaza, amelyek szerkezeti tulajdonsága a rekordok egymáshoz viszonyított lineáris elrendezése.
- *Formálisan*, ha n > 0, akkor az R(1) az első, R(n) az utolsó és az i-dik R(i) rekordot megelőzi az R(i-1), ha 1 < i <= n és az R(i)-t követi az R(i+1), ha 1 <= i < n.

A lineáris lista **rekurzív definíciója**:

- Egy lineáris lista lehet:
  - üres, ha n=0, vagy
  - egy R(n) rekord hozzáfűzése egy n-1 elemű lineáris listához, ha n > 0..

Lineáris listák

A lineáris listákkal végezhető műveletek:

- A lista k-dik rekordjának **elérése**, hogy mezőinek tartalmát megvizsgáljuk, esetleg megváltoztassuk.
- A k-dik rekord elé **új rekord beszúrása**
- A k-dik rekord **törlése**
- Két, vagy több lista **egyesítése** egy listává
- Egy lineáris lista **felbontása** több listává
- **Másolat készítése** a listáról
- Egy listán lévő **rekordok számának** meghatározása
- Egy listán lévő **rekordok átrendezése** valamilyen szempont(ok) szerint
- Egy listán lévő rekordok közül **kiválasztani** azokat, amelyek valamilyen tulajdonsággal rendelkeznek
- stb.

Lineáris listák

- A **verem** olyan lineáris lista, ahol minden beszúrás (írás) és törlés (kiolvasás) a listának ugyanazon a végén történik. (LIFO: last in, first out).

- A **sor** olyan lineáris lista, ahol minden beszúrás a lista egyik végén, minden törlés a lista másik végén történik. (FIFO: first in, first out)

- A **kétvégű sor** olyan lineáris lista, ahol minden beszúrás és minden törlés a lista mindkét végén lehetséges.

Lineáris listák tárolási módjai

- A lineáris listákat a számítógépen tárolhatjuk szekvenciálisan és láncoltan.

Szekvenciális helyfoglalás

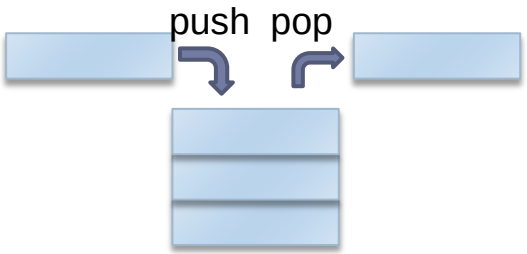
- Egy lineáris listát a legtermészetesebben úgy tárolhatunk számítógépen, ha a lista elemeit **sorban egymás után** helyezzük el.

- A szekvenciális tárolást akkor alkalmazzuk, ha azt akarjuk, hogy a lista logikailag egymás után következő rekordjai a memóriában egymás után következzenek.

- A szekvenciális tárolást megvalósíthatjuk egydimenziós tömbökkel. A tömb elemei rekordok.

Szekvenciális verem

- A szekvenciális tárolás kényelmes megoldást ad a verem kezelésénél.
- A verem kezeléséhez a következő műveleteket kell meghatározni:
  - Letárolás (push)
  - Kiolvasás (pop)
  - Ures-e? (isempty) és tele van-e? (isfull)



- A gyakorlatban a verem mérete maximált, legyen ez VM.
- Szükség van egy változóra, amely a verem tetejét jelzi, az utolsónak tárolt elem helyét.
- Jelöljük ezt VT-vel. Ha VT = 0, a verem üres. Ha VT = VM, a verem megtelt.

Szekvenciális verem

- Jelöljük V-vel a vermet és Y-nal azt az információt, amit tárolni szeretnénk.
- **Letárolás a verembe (*push*):**
  - VT = VT + 1
  - V [VT] = Y
- Ez a művelet nem hajtható végre, ha a verem megtelt. Vagyis ha VT = VM, a VT = VT + 1 túlcsordulást eredményez.
- **Kiolvasás, majd törlés a veremből:**
  - Y = V[VT]
  - VT = VT -1
- Ha a kiolvasás előtt már VT = 0, vagyis a verem üres, akkor ez a művelet nem hajtható végre, nincs mit kiolvasni. Ez a ”lecsordul” eset.
- A VT = 0 nem feltétlenül jelent hibát, ezt programszervezésre is fel lehet használni. Mindaddig olvashatunk a veremből, amíg VT > 0.
- A túlcsordulás viszont programhibát jelent !

Szekvenciális verem, példa

- **Játék**: Legyen egy 100 elemű verem, amiben van 20 szám (1-10). A felhasználó tippel a verem tetején levő számra. Ha kitalálja, háromszor több szám adódik a veremhez, ha nem, akkor kivesszük a számot a veremből. A játékos nyer, ha megtelik a verem, és veszít, ha kiürül.

Szekvenciális verem, példa

Ciklus 1-től 20-ig

    Push: véletlenszam(1-10)

Ciklus vége

Ciklus amig(nem ures a verem&&nem tele a verem)

    Pop: szam

    Be: tipp

    Ha(szam==tipp)

        Ciklus 1-tol 3xszam-ig

            Push: véletlenszam(1-10)

Ciklus vege

Ha(tele a verem)Ki: Nyertél

Különben Ki: Vesztettél

Szekvenciális verem

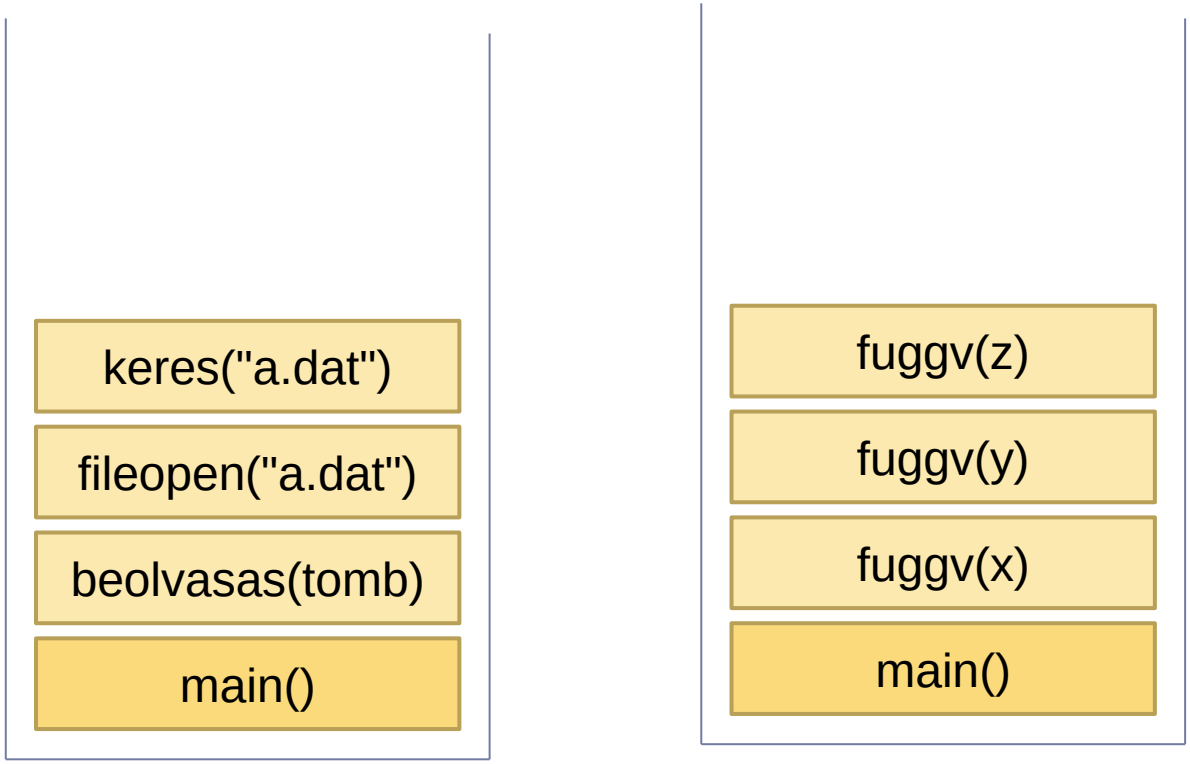
- Veremkezelő függvények C-ben:

```
int verem[100];
int VM=100;
int VT=0;

int isfull()
{
    if(VT>=VM) return 1; else return 0;
}
int isempty()
{
    if(VT<=0) return 1; else return 0;
}
```

```
int push(int szam)
{
    if(isfull) return -1;
    verem[VT]=szam;
    VT++;
}

int pop()
{
    if(isempty()) return -1;
    VT--;
    return verem[VT];
}
```

Szekvenciális verem

- A veremstruktúrát a függvények meghívási sorrendjének a tárolására használják



Szekvenciális sor

- A sor elemeinek letárolására egydimenziós tömböt definiálunk.
- Két mutatót definiálunk, a sor elejére E, a sor végére V mutat.
- **E** mindig az első értékes, még ki nem olvasott elem előtti helyre mutat, míg
- **V** az utoljára beírt elem helyét adja meg.

- Ha a sor üres, akkor E = V = 0.

Szekvenciális sor

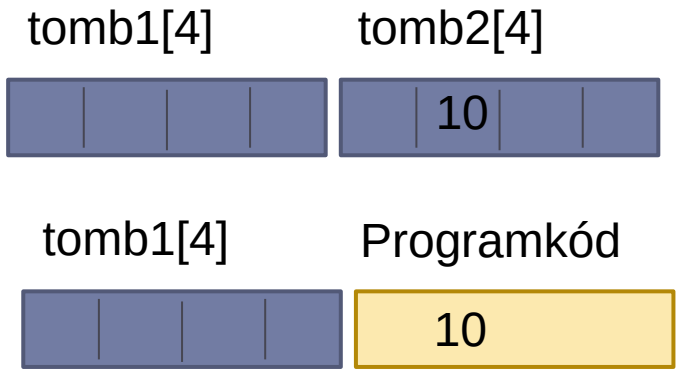
- **Beszúrás a sor végére:**
  - V = V +1 a vége mutató a beírandó elem helyére mutat
  - X(V) = Y tároljuk az információt

- **Olvasás a sor elejéről:**
  - E = E + 1 az eleje mutató a kiolvasandó elemre mutat
  - Y = X (E) kiolvassuk az információt.

- Ha az olvasás után E = V, akkor a sor kiürült, ekkor legyen E = V = 0!
- Ha tudjuk, hogy egyszerre nem kell M elemnél többet tárolni, akkor elegendő egy M elemű tömböt deklarálni a sor elemeinek tárolásához, s az M-dik után ismét az első helyet használni a tárolásra.
- Szekvenciális sorokat legtöbbször a távközlésben (internet, mobiltelefónia) használnak.

Lineáris listák túlcsordulása

- A számítógépes rendszerek egy fontos sebezhetősége.
- Akkor lép fel, ha egy program egy olyan memóriacímre próbál írni, amely nem volt lefoglalva a lista számára
- A C nyelv nem ellenőrzi a túlcsordulást



tomb1[5]=10;

Lineáris listák túlcsordulása

- Ellenőrizetlen túlcsordulás
  - Twilight hack: a Wii konzolt fel lehetett törni úgy, hogy a Legend of Zelda játékban a lónak egy hosszú nevet adtunk (ami bármilyen futtatható kódot tartalmazhatott)
  - Dokumentumokba ágyazott rosszindulatú kódok
    - "Nézd meg ezt a vicces filmet" típusú emailek
    - JPEG of Death kihasználta a GDI+ könyvtár egy sebezhetőségét
  - Polimorf (önmódosító) programok
- Ellenőrzött túlcsordulás
  - A program nem engedi, hogy nem lefoglalt helyre írjunk
  - Ez adatveszteséggel jár
  - Például a DoS támadások aknázzák ki ezt a sebezhetőséget.

Lineáris listák, láncolt sor

- A szekvenciális listák (tömbök) hátrányai:
  - A méretüket az elején meg kell határozni (dinamikus foglalás esetén is)
  - Az elemek törlése és beszúrása sok munkával jár

- A **láncolt helyfoglalás** sokkal rugalmasabb és takarékosabb megoldást ad a lineáris listák tárolására. Ekkor minden rekord tartalmaz egy mutatót is, amely a lista következő rekordjának a helyét adja meg.

- A tárolási egység:
  - Vektorelem: adat
  - Listaelem: adat / mutató

Láncolt listák

- Láncolt lista ábrázolása:

- A lista azonosítására a Start mutató szolgál, az utolsó mutató nem mutat sehova (NULL)

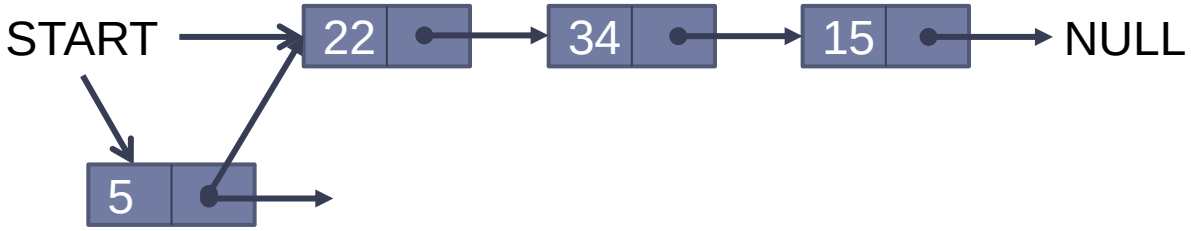
Összehasonlítás

- -- A láncolt lista több tárhelyet foglal (minden adat tartalmaz egy mutatót is)
- + A láncolt lista elemei számára mindig éppen annyi memória foglalható, amennyi az aktuális elemek tárolásához szükséges (dinamikus helyfoglalás).
- -- A lista egy kiválasztott elemére sokkal könnyebb hivatkozni szekvenciális tárolás esetén. Láncolt tárolásnál végig kell menni az elemeken.
- + Elemek beszúrása és törlése egyszerűbb a láncolt tárolásnál. Az elemeket nem kell mozgatni, csak a mutatókat kell átírni.
- + Láncolt listáknál könnyebb két listát egymáshoz csatolni, vagy egy listát két részre bontani. Az elemeket sem kell mozgatni, csak a mutatókat kell átirányítani.
- + A láncolt séma bonyolultabb struktúrák létrehozására is alkalmas.

Láncolt listák kezelése

- Lista:

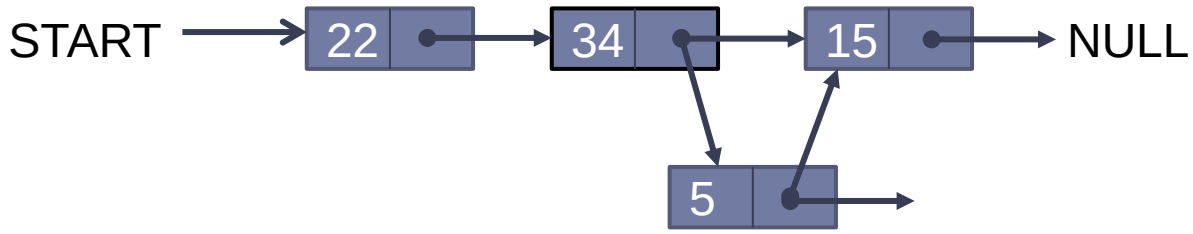
- Elem beszúrása az elejére



  - Új elem létrehozása
  - Az új elem mutatója az első elemre mutat
  - A START mutató az új elemre mutat

Láncolt listák kezelése

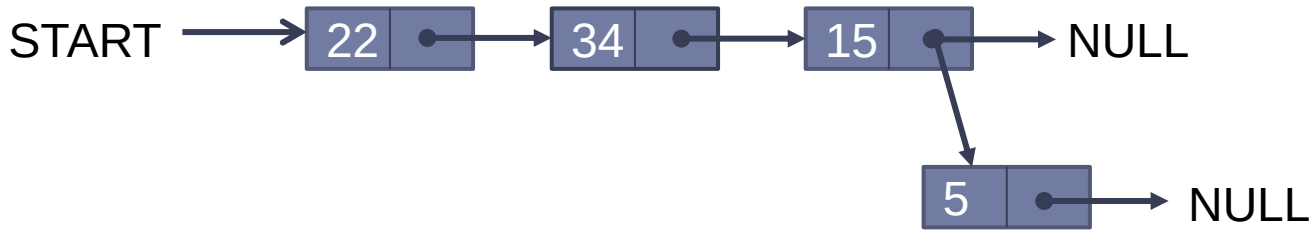
- Beszúrás közbenső elemként



  - Létrehozzuk az új elemet
  - Az új elem mutatóját az aktuális elemet követő elemre állítjuk
  - Az aktuális elem mutatóját az új elemre állítjuk

Láncolt listák kezelése

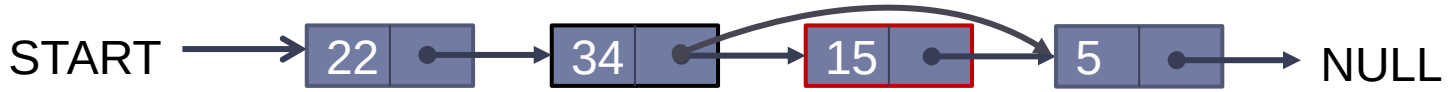
- Hozzáfűzés a lista végéhez



  - Létrehozzuk az új elemet
  - Az új elem mutatóját NULL-ra állítjuk
  - Az aktuális elem mutatóját az új elemre állítjuk

Láncolt listák kezelése

- Elem törlése



  - Az előző elem mutatóját a törlendő elemet követő elemre állítjuk
  - Az aktuális elemet felszabadítjuk
- Listák összefűzése
  - Az első lista utolsó elemének a mutatóját a második lista első elemére állítjuk

Műveletek listákkal

- **Deklarálás**

```
struct lista {
        int adat;
        struct lista *kovetkezo;
}
struct lista *start;
struct lista *elem,*p;
```

- **Elso elem létrehozása**

```
elem=(struct lista*)malloc(sizeof(struct lista));
elem->adat=10;
elem->kovetkezo=null;
start=elem;
```

- **Második elem fűzése a lista végére**

```
p=elem;
elem=(struct lista*)malloc(sizeof(struct lista));
elem->adat=15;
elem->kovetkezo=null;
p->kovetkezo=elem;
```

- **Lista bejárása**

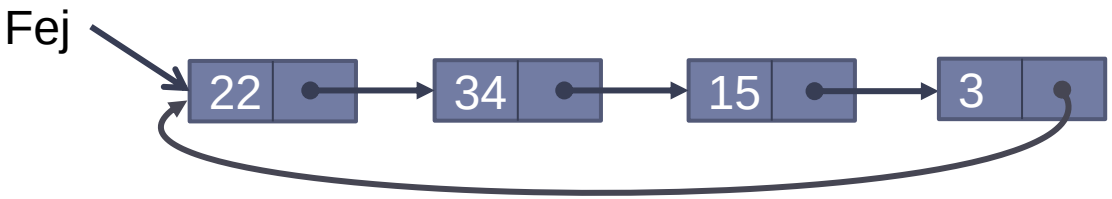
```
elem=start;
while (elem!=NULL)
{
    printf("%d ",elem->adat);
    elem=elem->kovetkezo;
}
```

- **Elem törlése**

```
elem=start;
p=elem->kovetkezo;  //ezt fogjuk torolni
elem->kovetkezo=p->kovetkezo;
free(p);
```

Ciklikus listák

- A ciklikusan láncolt lista, vagy ciklikus lista olyan láncolt lista, amelyben az utolsó elem mutatója nem NULL, hanem az első elemre mutató cím. Ilyen módon a lista bármelyik része bármelyik adott pontból kiindulva elérhető.

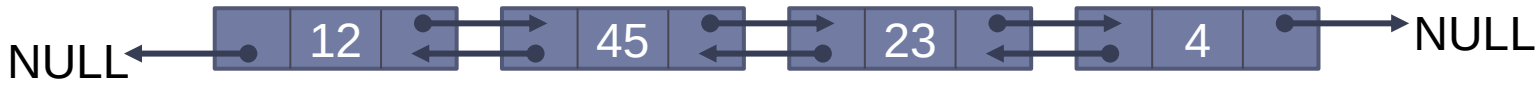


- A listának egy elemét mindig ismerni kell (lista-fej)

Kétszeresen láncolt listák

- **Egyszeresen láncolt lista** esetén mindig az első elemtől kiindulva és csak egy irányban lehet bejárni a lista elemeit. Ha egy adott elemen „állunk”, csak a következő elemre léphetünk, visszafelé nem. (Az elem nem tartalmazza a megelőző elem címét.)

- **Kétszeresen láncolt lista**: minden rekordban két mutatót helyezünk el, az egyik a rekordot megelőző, a másik a rekordot követő rekord címét tartalmazza.



Hasító táblák

- Közvetlenül címezhető egydimenziós tömbök
- A hasító tábláknál minden elem egy kulccsal van azonosítva. A kulcs megegyezik az index-szel.

- A közvetlen címzés akkor előnyös, ha kicsi a kulcsok lehetséges száma.

- Két probléma lehet:
  - Nincs elég hely
  - Az adatok száma kicsi, de az intervallum nagy

Hashing

- ***Az alapgondolat:***
  - ***Használjunk egy olyan eljárást, amely a kulcs alapján azonnal képes előállítani a tömbelemhez egy olyan címet (indexet), ahol az adott tömbelemnek el kell helyezkednie.***
  - ***Ha ezzel az eljárással címezve írjuk be az elemet az "adatbázisba", akkor kereséskor is ugyanott kell megtalálnunk. Erre egy ún. hash függvényt definiálunk, amely a kulcsból címet generál:***

cim = hash_fv(kulcs);

Hashing…

- **Legegyszerűbb címgenerálás:**

  cim = kulcs % TABLAMERET;

  - ez biztosan 0 és TABLAMERET-1 között értékeket szolgáltat, ami tömbcímzésre tökéletes.

- *Gond*: hash_fv(kulcs1) == hash_fv(kulcs2) tehát ha két különböző kulcshoz ugyanaz a cím áll elő. Márpedig x%N és (x+N)%N mindig azonos. Ez a szituáció az ***ütközés***: egy új, más kulcsú adatot egy már foglalt memóriahelyre szeretnénk beírni.

- *Megoldás*: az első üres hely megkeresése (lineáris kereséssel) az ütközött adat számára. Kellemetlen következmény: nem lehet adatot fizikailag törölni, mert lehet olyan adat a táblázatban, ami ütközött vele. Ha kitörlünk egy adott címre először beírt adatot, akkor soha nem fogjuk megtalálni a vele később ütközött adatot.

Hashing …

- **Funkciók**: adatbeírás, adatkeresés, adat logikai törlése, táblázat tömörítése.
  - **Logikai törlés**: egy adatot sem veszünk ki a táblázatból, csak azt jelezzük, hogy érvénytelen.
  - **Tömörítés**: ha egy olyan adat, amivel egy másik ütközött már érvénytelen, akkor az ütközött adatot beírhatjuk a helyére és a korábban ütközött adat által elfoglalt rekeszt felszabadíthatjuk.

- **A tömörítés menete**: A táblázat minden üres eleméről el kell dönteni, hogy a valódi címén van-e. Ha nem, akkor ütközött. Ha ütközött, akkor meg kell nézni a kiszámított címén lévő rekesz státuszát. Ha az érvénytelen, az ütközött adattal az felülírható.

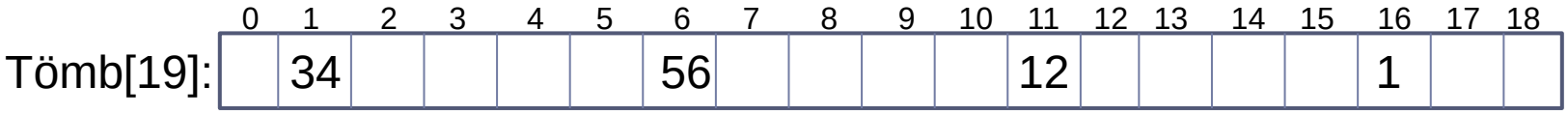
Hashing …

- Tehát célszerű minden rekeszhez egy státusz információt is tárolni.
- Ha például pozitív egészekből áll az adatbáziusunk, akkor a 0 érték jelentheti az üres helyet (empty) ; egy adat érvénytelen (invalid) voltát a negatív előjel jelezheti, maga az adat pedig a rekesz értékének abszolút értéke lehet. Ha pozitív az előjel, akkor az adat érvényes (valid).

- Általában a státusz nyilvántartására használhatunk egy státusz jelzőt.

Ritka (sparse) tömbök

- Olyan tömbök, amelyek sok üres elemet tartalmaznak
- Ebben az esetben a tömb helyett a feltöltött elemek értékét és pozícióját tároljuk.



Sparse[4]: ((1,34); (6,56); (11,12); (16,1))

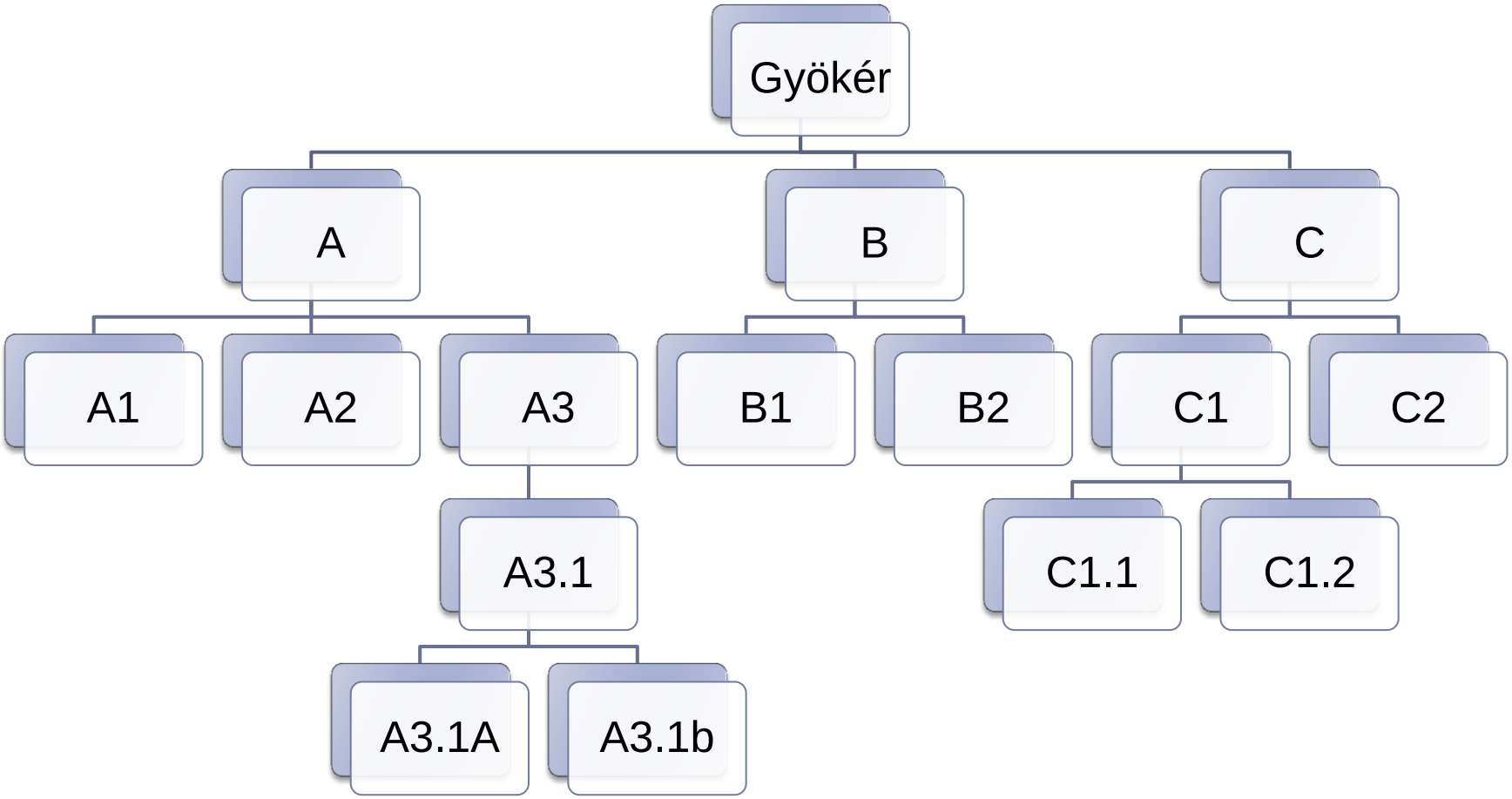
- A ritka tömbök tárolására alkalmasak a láncolt listák
- A ritka tömbábrázolás felgyorsíthatja a ritka adatokon végzendő műveleteket is.

Fák, bináris fák

- A fa a számítógépes algoritmusokban előforduló legfontosabb nem lineáris struktúra. Az azonos típusú rekordokból (C-ben struktúra) a következőképpen alkothatunk fastruktúrát, vagy röviden fát.
- Egy fastruktúra
  - vagy üres,
  - vagy egy kitüntetett csúcshoz (T-hez) kapcsolódó véges sok, közös csúcs nélküli T1, T2, ... Tm, fából áll.
- A kitüntetett T csúcsot a fa gyökerének, míg a T1, T2, ... Tm fákat a gyökér részfáinak nevezzük.

- A lineáris lista tehát egy olyan fastruktura, amelyben minden csúcshoz legfeljebb egy részfa tartozik. A lineáris listákat ezért elfajult fáknak is tekinthetjük.

Fák, bináris fák

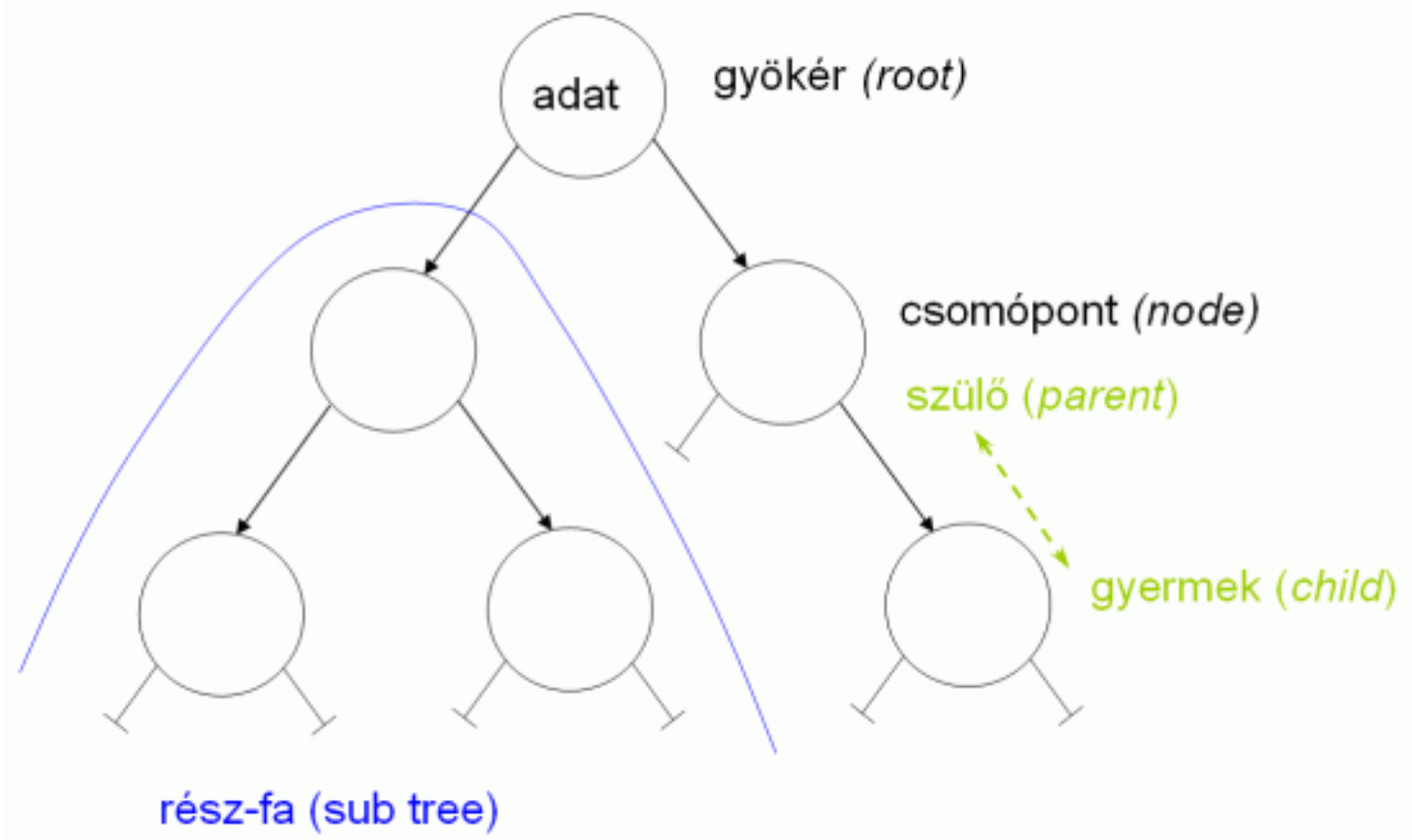


Fák, bináris fák

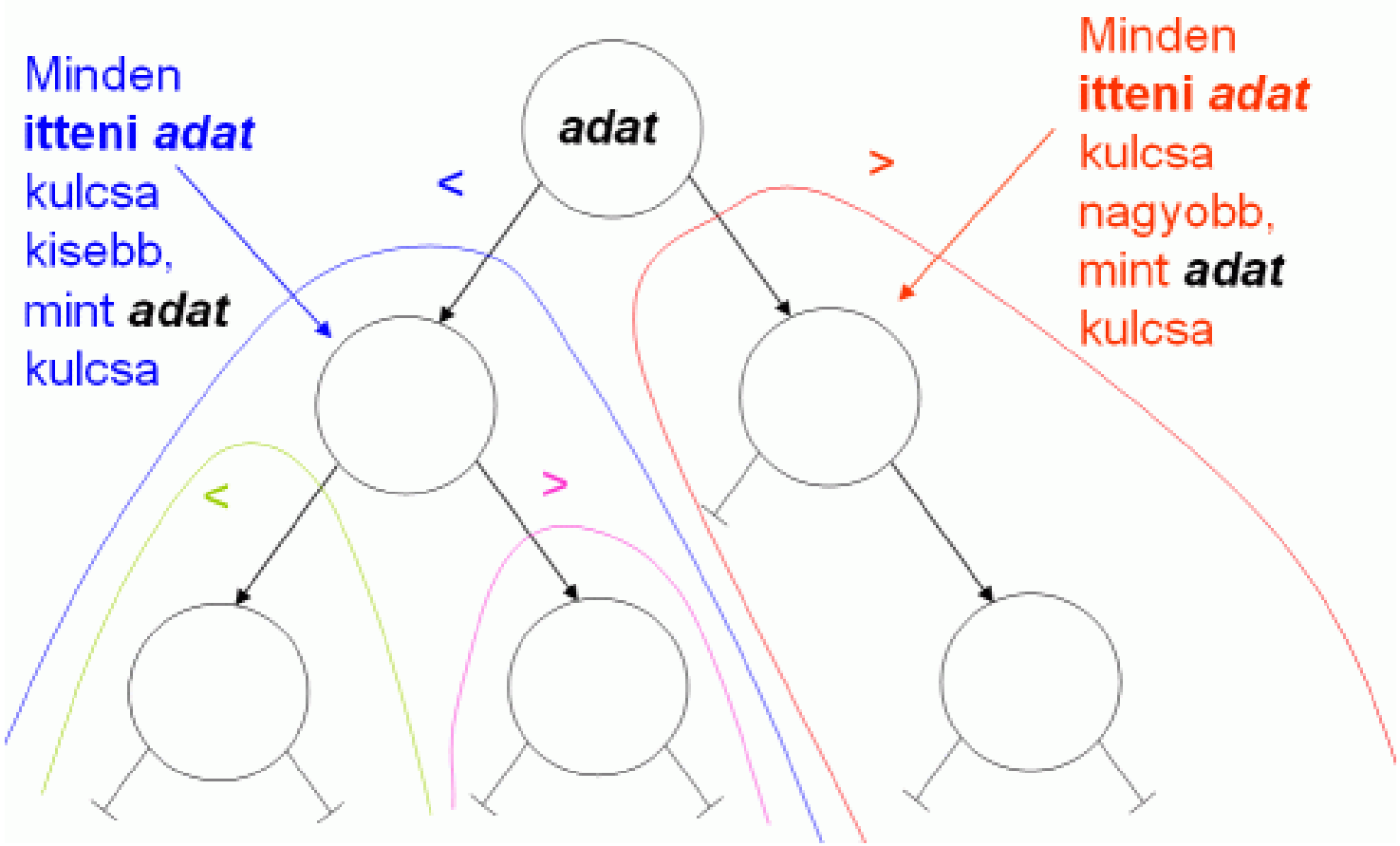
- Abrázolás
  - Gráffal
  - Bekezdéses tagolással
  - Zárójelezéssel

- Bináris fák
  - Minden csomópontnak két ága van. Ezeket bal- illetve jobb részfáknak hívunk

Bináris fák


adat
gyökér (root)
csomópont (node)
szülő (parent)
gyermek (child)
rész-fa (sub tree)

Rendezett bináris fák



Bináris fák deklarálása és kezelése

- **Deklarálás**:

```
struct csomopont {
        int adat;
        struct csomopont *bal;
        struct csomopont *jobb;
}
```

- **Keresés**:
  - Indulás a gyökérből
  - Ha a kulcs egyezik, megvan az elem
  - Ha kisebb, továbblépés a baloldali részfába, ha lehet; ha nagyobb, továbblépés jobbra, ha lehet.
  - Ha már nem lehet a fában a rendezettség szerinti irányba továbblépni, az elem nincs benne a fában. Ha szükséges, az utolsóként bejárt csomópont megfelelő ágára felfűzhető - ezzel a keresett adatot fel is vettük a fába.

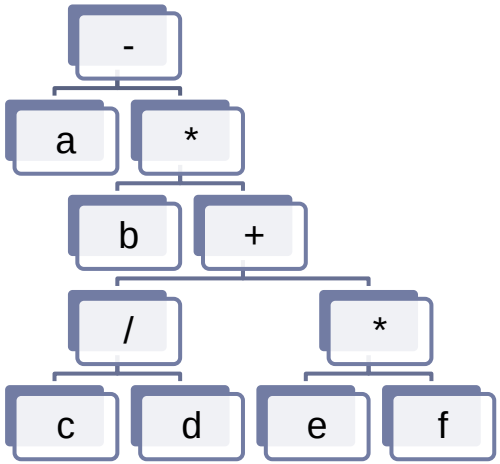
Bináris fák deklarálása és kezelése

- **Új elem beiktatása:**
  - Üres a fa? Ha igen, új elemként felvesszük; ez lesz a gyökér.
  - Ha nem üres a fa, az elem helyét meg kell keresni - lásd fent.
  - Ha megtaláltuk az elemet a fában, nem kell tenni semmit, ha NULL-ba ütköztünk, akkor arra az ágra felfűzzük.Így a fa automatikusan rendezetten épül.
- **Törlés - nem triviális:**
  - *Utód nélküli levél*: nincs gond, a foglalt memóriát felszabadítjuk, a reá mutató pointert-t NULL-lá tesszük.
  - *1 utóda van*: a törlendő elemre mutató pointer értéke felülírandó a törlendő elem egyetlen utódára mutató értékkel, majd a foglalt memóriát felszabadítjuk
  - *2 utóda van*: a törlendő elemre mutató pointer az elem bal oldali utódjára fog mutatni. A jobb oldali utódját hozzáfűzzük a bal oldali utód jobboldali ágának a végére. Végül a foglalt memóriát felszabadítjuk

Bináris fák

- Bináris fákkal ábrázolhatók az aritmetikai kifejezések
- Például:

a-b*(c/d+e*f)



Adatszerkezetek

- **Lineáris listák**
  - **Szekvenciális helyfoglalás**
    - **Szekvenciális verem**
    - **Szekvenciális sor**
  - **Láncolt sorok**
    - **Ciklikusan láncolt listák**
    - **Kétszeresen láncolt listák**
- **Hasító táblázatok**
- **Fák, bináris fák**