Adatszerkezetek és algoritmusok

Jegyzet dr. Juhász István előadása alapján

Készítette Csordás Annamária és Mohai Gábor

A valós világban rendszerekről beszélünk. A dolgok összetevői egymással kölcsönhatásban vannak, viselkednek, tulajdonságaik vannak. Jellemzőjük a viselkedés, ezzel foglalkozik a rendszerelmélet.

Rendszernek nevezzük a közös ismérv alapján összetartozó, egymással meghatározott kapcsolatban lévő elemek jól körülhatárolt együttesét. Az elem a rendszer azon része, amelyet azért választunk ki, hogy vizsgálódást végezzünk rajta. Segítségével jellemezhető a rendszer. Egy elem is lehet rendszer. A kölcsönhatás az elemek közötti olyan kapcsolat, reláció, amely az adott elemeket a rendszer részévé teszi.

A rendszer több mint az elemek együttese. A rendszerek jellemzői:

- *komplexitás:* Sok elem és bonyolult kölcsönhatások jellemzik.
- *nyíltság:* A rendszer nem önmagában létezik, hanem más rendszerekkel együtt. Környezetével állandó kapcsolatban van.
- *dinamikusság:* A rendszer elemeinek különböző állapotai vannak, és a rendszer elemei valamint viselkedésük időben állandóan változnak.

A valós rendszerek túlságosan komplexek, a teljes összetettségükben általában nem tudjuk kezelni őket, így egyszerűsítési módszerekre van szükség. Célunk a komplex rendszerek leegyszerűsítése.

A modellezés lényege az absztrakció, azaz az elemek lényegét, karakterisztikus közös tulajdonságait kiemeljük, a különbözőségeket elhanyagoljuk és a modellbe ezeket a közös tulajdonságokat vonjuk be. A modellben vizsgálunk, kérdéseket teszünk fel, következtetéseket vonunk le, amit a valós rendszerre visszavezetünk. Célja, hogy a valós világról olyan ismereteket szerezzünk, melyek helyesen tükrözik a valós világot. Az ismeretekre azért van szükség, hogy átadhassuk őket. Az ismeret mindig rögzített formában van, és hozzáférhető.

Adatnak nevezzük a tények, elképzelések nem értelmezett, de értelmezhető formában rögzített alakját (pl. 1995).

Az információ az adatokon végzett gondolati művelet eredményeként áll elő (pl. 1995-évszám). Az információ általában viszonylagos. Ugyanaz az adat az egyes emberek számára mást és mást jelent. A modellezés, az információ és a rendszer is viszonylagos.

A rendszerben az elemek kölcsönhatásait, tulajdonságait adatokkal írjuk le. Ezt nevezzük *adatmodell*nek, ami igen bonyolult szerkezetű lehet. Az adatmodellel az Adatszerkezetek és algoritmusok tárgy foglalkozik.

Az elemek viselkedését, a rendszer módosulásait *eljárásmodell*nek nevezzük. Ezzel a Programozás 1. foglalkozik.

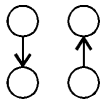
Többfajta irányelv létezik. Van, aki szerint a viselkedésből következik az adatmodell, a másik felfogás szerint az adat tulajdonsága meghatározza az eljárást. Adatcsoportokkal tudok jellemezni egy-egy elemet, és ezek között kemény logikai összefüggés van. A modellben (logikai) adatszerkezetekről *(data structure)* beszélünk.

Az adatszerkezetek adattételekből vagy adatelemekből állnak és közöttük valamilyen kapcsolat van.

Fizikai adat- vagy társzerkezeteknél meg kell különböztetni az operatív tárat, vagy memóriát, amelynek bármelyik byte-ja közvetlenül elérhető, valamint a háttértárakat vagy perifériákat, melyeknél általában van mechanikai mozgás. A mágneslemeznél nem pontos az a kifejezés, hogy közvetlen elérés, mert bár mindegyik elem elérhető, de ez nem ugyanannyi idő alatt történik. Az adatszerkezetek egyik fajtája az állomány *(file)*. Az állomány az az adatszerkezet, amelyik csak periférián létezik.

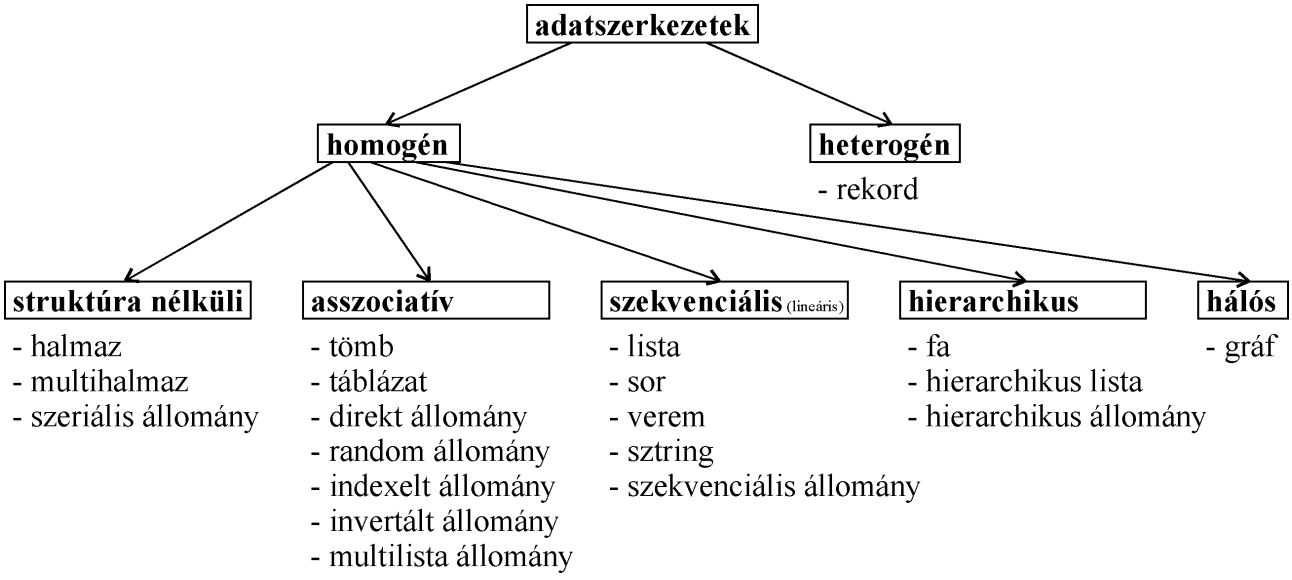
Adatszerkezetek szemléltetése

Egy-egy elem általában adatcsoporttal jellemezhető, melyek adatai mindig összefüggenek. Az adatszerkezetek adattételekből, adatelemekből állnak. Ezek között valamilyen kapcsolat áll fenn.

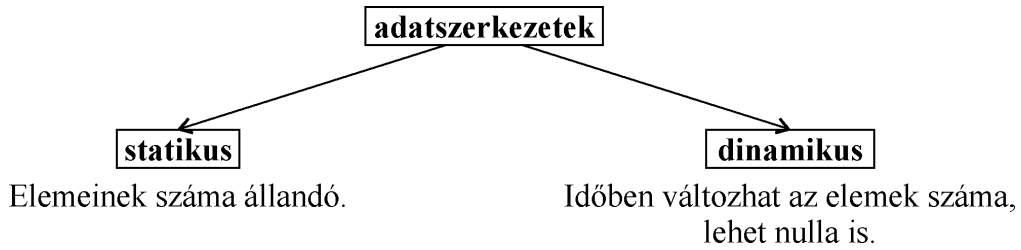
Adatelem: ○ Kapcsolat: 

A kettő nem ugyanazt jelenti.

Adatszerkezetek osztályozása



Homogén adatszerkezet: Az adatelemek azonos típusúak.
Heterogén adatszerkezet: Az adatelemek különböző típusúak.



Bármely adatszerkezet mindig véges számú adatelemet tartalmaz.

Műveletek az adatszerkezetekkel

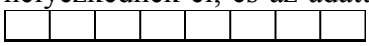
Létrehozás: Nem volt adatszerkezet, létrehozás után lesz.
Módosítás:

- **Bővítés vagy beszúrás** (csak dinamikus adatszerkezeteknél): Új adatelemek kerülnek be az adatszerkezetbe, az adatelemek száma nő.
- **Törlés:** Megkülönböztetjük a logikai és fizikai törlést:
  - fizikai törlés: Csökken az adatszerkezet elemeinek a száma, az addig ott levő eleme(ke)t távolítom el.
  - logikai törlés: Az adatszerkezet elemeinek a száma nem csökken, a törölni kívánt elemeket egy speciális jelzéssel („törölt” jellel) látom el, ezek a további feldolgozásban nem vesznek részt.
- **Csere:** Az elemek száma nem változik, csak valamely elem vagy elemek értéke, mert kicserélem egy másik értékre.

Rendezés
Keresés
Bejárás (elérés): Az adatszerkezet minden elemét érinteni akarom, a kérdés az, hogy ezt milyen sorrendben tehetem meg. A sorrendet alapul véve a bejárás lehet soros, ha az adatszerkezet elemeit valamilyen fizikai sorrend alapján tudjuk megfogni, ahogy le vannak képezve (ld. mágnesszalag), szekvenciális, ha a bejárásnak valamilyen logikai sorrend az alapja (ld. szétszórtan tárolt elemek), vagy közvetlen, ha az elérés független a többi adatelemektől (ld. folytonos tárolás).
Feldolgozás: Az adott adatszerkezet hogyan és mire használható.

Adatszerkezetek tárolása

A logikai adatszerkezeteket le kell képezni a tárba. A memóriában kétféle módon tárolhatunk adatszerkezeteket.

I. **Folytonos (szekvenciális, vektorszerű) tárolással.** A memóriában tárolt elemek egy folytonos tárterületen egymás után helyezkednek el, és az adattételek tárolási jellemzői azonosak (típus, ábrázolási mód, hossz).  Ismerni kell a tárterület kezdőcímét, a tárolási jellemzőket, és az elemek számát. Ilyen tárolásnál minden elemet közvetlenül meg tudok fogni.

Minden adatszerkezet tárolható ilyen módon.

Műveletek:

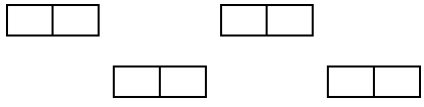
Létrehozás: Az adatelemeket, adattételeket egy-egy tárrészbe helyezem el. A kezdőcímtől kezdve minden tételt folytonosan pakolunk a tárba.
Bővítés: A végére újabb elemeket lehet rakni. Beszúrásnál az adott elem behelyezésének helye után lévő elemeket jobbra kell tolni, át kell mozgatni.
Törlés: Megkülönböztetünk fizikai és logikai törlést:

- Fizikai törlés: A tárrész mögött lévő elemeket rácsúsztatom a törlendő elemre, mindegyiket eggyel balra.
- Logikai törlés: Az elemek száma nem csökken. A törölni kívánt elemet megjelölöm, hogy törölve van („törölt” jellel). Ekkor az adatelem értéke változik meg, így ez valójában egy csere.

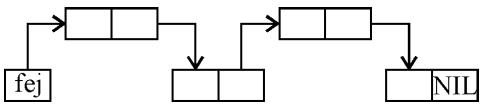
Csere: Bármely elem közvetlenül megfogható és felülírható.
Rendezés: Ha van értelme, akkor mindegyik rendezési algoritmus alkalmazható.
Keresés: Ha van értelme, akkor minden keresési algoritmus alkalmazható.

II. **Szétszórt tárolás.** Az adatelemek tárolási jellemzői különbözőek lehetnek. Minden adatszerkezet tárolható szétszórtan.

A tényleges adatelem mellett létezik a tárrésznek egy úgynevezett mutató része is, amely olyan információt tartalmaz, hogy az adott elem után melyik másik elem következik az adott adatszerkezetben.



1. Egyirányban láncolt lista (linked list)



Minden elemnél a mutatórész a következő elem tárolási címét, az adatrész pedig az adatelem értékét tartalmazza. Tudni kell, hogy melyik a lista első eleme. Ennek a címét a

fej tartalmazza, ez listán kívüli elem, nem tartozik az adatszerkezethez. Az utolsó elem mutatórésze nem mutat sehova sem, a lista végét jelzi.
Üreslista: Csak a fej létezik a speciális, mutató értékkel. NIL

Műveletek:

Létrehozás: Első lépésként létrehozom az üreslistát, utána bővítek.

```
typedef struct elem {
   int adat;
   struct elem *kov;
} ELEM;

ELEM *fej = NULL;
```

Bővítés: A lista bármelyik helyére beilleszthetek egy új tárolási elemet.

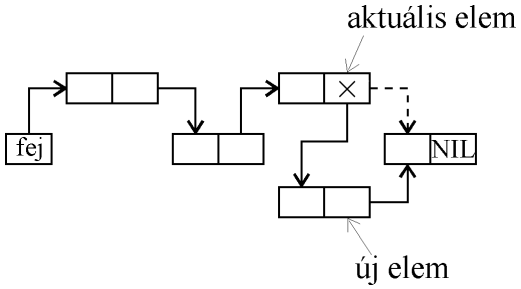
- Bővítés a lista elején: Lefoglalok egy helyet a tárban. Bárhol megtehetem, de tudom a címét. Az új elem az eddigi első elemre mutat, a fej pedig az új elemre.

```
ELEM *bovites_elejere ( ELEM *fej, int mit)
{
   ELEM *uj;

   uj = (ELEM*) malloc (sizeof (ELEM));
   uj->adat = mit;
   uj->kov = fej;
   if (fej == NULL) return uj;
   return fej;
}
```

- Bővítés a lista végén: Lefoglalom a tárhelyet. Az új elem mutatója a lista végét jelző információ, NIL. Az utolsó elem mutatórészébe beírom az új elem címét, az eddigi utolsó elem mutatója az új elemre fog mutatni. Az utolsó elemet meg kell keresni, végig kell menni a listán, tehát szükség van egy bejárásra.

```
ELEM *bovites_vegere ( ELEM *fej, int mit)
{
   ELEM *p = fej, *uj, *seged = NULL;

   uj = (ELEM*) malloc (sizeof (ELEM));
   uj->adat = mit;
   uj->kov = NULL;
   if (fej == NULL) return uj;
   while (p->kov != NULL)
   p = p->kov;
   p->kov = uj;
   return fej;
}
```

- Bővítés a listában (beszúrás): Az elérés szekvenciális. A listánál a keresés teljes keresés, ahol adatértéket keresek. Minden elemre csak úgy tudok eljutni, ha előtte végig mentem az előtte lévő elemeken. Az aktuális elem az elem, amin állok. Az aktuális elem elé, vagy mögé is beszúrhatok.
  - Az aktuális elem mögé bővítek: A megelőző elemekről nincs információm. Az új elem mutatórészébe az aktuális elem mutatórészében lévő címet, az aktuális elem mutatórészébe pedig az új elem címét írom.




```
ELEM *bovites_elem_moge(ELEM *fej, int mit, int mi_moge)
{
  ELEM *uj, *p = fej;

  if ( fej == NULL ) {
    printf("Nem talaltam meg a keresett elemet!\n");
    return fej;
  }
  while (p->adat != mi_moge)
  p = p->kov;
  if (p == NULL) {
    printf("Nem talaltam meg a keresett elemet!\n");
    return fej;
  }
  uj = (ELEM*) malloc (sizeof (ELEM));
  uj->adat = mit;
  uj->kov = p->kov;
  p->kov = uj;
  return fej;
}
```

– Az aktuális elem elé bővítek: Az előző módszerrel nem lehet bővíteni, mert nem tudom megmondani, hogy mi az előző elem címe, ezért meg kell jegyezni az aktuális elem előtti elem címét.

```
ELEM *bovites_elem_ele(ELEM *fej , int mit, int mi_ele)
{
  ELEM *uj, *p = fej, *elozo = NULL;

  if (fej == NULL) {
    printf("Nem talaltam meg a keresett elemet!\n");
    return fej;
  }
  while (p->adat != mi_ele) {
    elozo = p;
    p = p->kov;
  }
  if (p == NULL) {
    printf("Nem talaltam meg a keresett elemet!\n");
    return fej;
  }
  uj = (ELEM*) malloc (sizeof (ELEM));
  uj->adat = mit;
  uj->kov = p;
  if (elozo == NULL)
  return uj;
  elozo->kov = uj;
  return fej;
}
```

Törlés: A bővítés ellentéte.

- Első elem törlése: A lista fejébe az első elem mutatóértékét viszem.

```
ELEM *torles_fejet ( ELEM *fej)
{
  ELEM *p = fej;

  if (fej == NULL) return fej;
  fej = fej->kov;
  free( p);
  return fej;
}
```

- Utolsó elem törlése: Az utolsó előtti elemet megkeresem és a mutatórészébe NIL-t írok.

```
ELEM *torles_veget ( ELEM *fej )
{
  ELEM *p = fej;

  if (fej == NULL) return fej;
  while (p->kov != NULL)
  p = p->kov;
  if (fej == p) {
    free(p);
    return NULL;
  }
  else {
    free(p);
    return fej;
  }
}
```

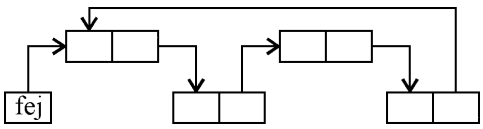
- Ha az aktuális elemet akarom törölni, akkor az aktuális elem mutatórészét átviszem a megelőző elem mutatórészébe.

```
ELEM *torles_elemet( ELEM *fej, int mit)
{
  ELEM *p = fej, *elozo = NULL;

  if (fej == NULL) return fej;
  while (p->adat != mit) {
    elozo=p;
    p=p->kov;
  }
  if (p == NULL) return fej;
  if (elozo == NULL) {
    fej = p->kov;
    free( p);
    return fej;
  }
  elozo->kov = p->kov;
  return fej;
}
```

Csere: Simán végrehajtható.

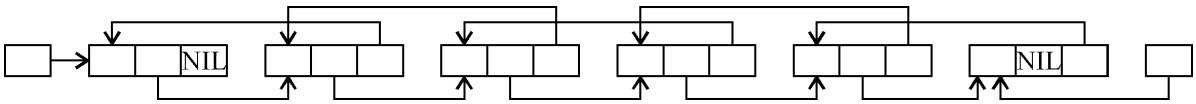
Keresés: Csak teljes kereséssel, amennyiben a lista nem rendezett. Ha rendezett, akkor lineáris kereséssel.
Rendezés: Tegyük fel, hogy van egy listám, és azt saját helyén akarom rendezni. Ez nem célszerű, a gyakoribb az, hogy rendezetten hozok létre egy új listát. Ezt úgy tehetem meg, hogy létrehozom az üres listát, és beszúrásos rendezést alkalmazok. A beszúrandó elem helyét lineáris kereséssel keresem meg.
Bejárás: Minden probléma nélkül megtehető.
Feldolgozás: Adatszerkezet-függő.

2. Körkörös (cirkuláris) lista



Az utolsó elem mutatója az első elem címére mutat. Bármelyik elemből körbe tudok menni, a bejárást egyszerűsíti.

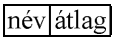
3. Kétirányban láncolt lista



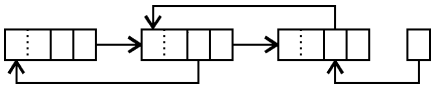
Minden elemnek két mutatója van, amelyek a következő és a megelőző elemről tartalmaznak információt. Általában két fej van, a lista elejére és a végére mutatnak, így két irányban tudok haladni.

4. Multilista

Több értelmezése is van.

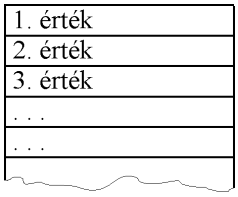
A. Az adatelemek összetettek, pl. 

Kialakítok az egyik és a másik részre is egy listát. Pl. a listaszerkezet tartalmazza a neveket ábécé-sorrendben, az átlagokat csökkenő sorrendben. Több mutatórész van, mindegyik egy-egy adatelem-rész szerint láncol. Annyi mutatórész és annyi fej van, ahány lánc.

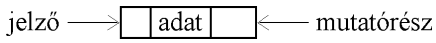


Ez is lehet kétirányban láncolt, illetve körkörös lista.

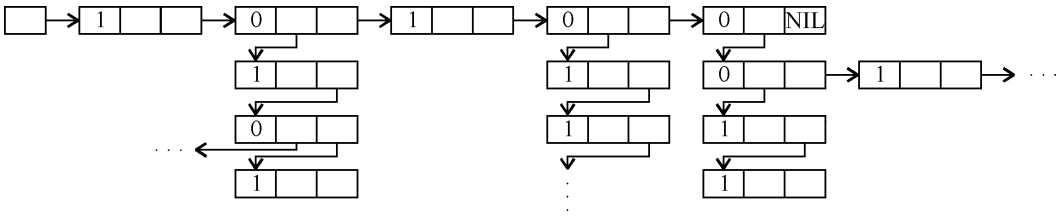
B. Legyenek az adatelemek összetettek, így van egy adatelem sorozatom. Együtt akarom kezelni az azonos értékűeket (pl. a Debrecenben lakókat), tehát olyan részlisták jönnek létre, amelyben azonos értékű elemek vannak. Annyi részlista és fej van, ahány féle érték található. A fejek:



C. Az adatrész is tartalmazhat mutatórészt, amely egy másik listára mutat. Ekkor listákat fűzök fel.



A jelzőrész azt jelzi, hogy a listaelem tényleges adatra, vagy egy másik listára mutat. Jelző: 1 – adatrész, 0 - mutató (bitek).

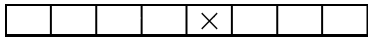


Ezek tárolási szerkezetek. Vannak olyan adatszerkezetek, amelyekhez jobban illik a folytonos tárolás, másokhoz praktikusabb a szétszórt.
Reprezentáció: A tárolási mód és a leképezés együttese.
Implementáció: A reprezentáció és a műveletek együttese.

Törlés…

Amikor adatszerkezetben törlést végzünk, akkor egy tárhely szabadul fel. Kérdés az, hogy hogyan tudunk ezzel a szabad tárhellyel gazdálkodni.

I. … folytonos tárolás esetén:



1. Az egyik megoldás az, hogy a tárhely felszabadulásakor átszervezzük az egészet, azaz a törlendő elemre rácsúsztatjuk a mögötte lévő elemeket. A szabad helyek ilyenkor mindig a lefoglalt tárrész után lesznek. A folyamat igen időigényes, viszont a problémánkat biztosan megoldja.
2. Hulladékgyűjtés *(garbage collection)*: Csak logikai törlés van, a lyukakat otthagyom, folyamatosan írok tovább. Amikor elfogy a rendelkezésre álló tárhely, akkor a rendszer a szabad helyeket rendszeres időközönként átszervezi, átrendszerezi a végére, hogy összefüggő tárterületet kapjunk. Sok operációs rendszer van, amely így kezeli le a törlést.
3. A tárhelyeket nyilvántartjuk egy bitmátrix segítségével. A mátrix elemei: 1 (foglalt), 0 (szabad), ez alapján tudok elemeket elrakni. Az adatelemeket nem kell mozgatni, viszont külön nyilvántartás kell.

II. … szétszórt ábrázolás esetén:

1. A szabadhelyek össze vannak láncolva. Ha új elemet akarok láncolni, akkor a szabadhelyek listájáról leveszem az első elemet, és azt fűzöm az állományhoz. Ha törlést végzek, akkor az adott elem helyét a szabad elemek láncához láncolom. Időben gazdaságos.

   A gond az, hogy különböző méretű tárhelyeket láncolok, így olyan algoritmusokat kell alkalmazni, amelyek ellenőrzik, hogy az új elem befér-e a tárrészbe és meg kell keresni azt a tárhelyet, ahova befér (*first fit:* az első megfelelő tárhelyet foglalom le; *best fit:* a megfelelőek közül a legkisebbet foglalom le). Ha túl kicsi elemet pakolok bele, akkor nem gazdaságos.
2. Szabadhelyek nyilvántartása: Mint folytonos tárolás esetén. Bitmátrix segítségével történik.

3. Hulladékgyűjtési technika, amelyben láncolom a szabadhelyeket. Ha elfogy a szabad hely, akkor láncolok.

Absztrakt adatszerkezetek:

Akkor beszélünk absztrakt adatszerkezetekről, ha az implementációtól eltekintünk. Az implementáció, a reprezentáció mindig rendszerfüggő, az absztrakt adatszerkezet független tőle, csak a tulajdonságai definiálják.

Struktúra nélküli adatszerkezetek

Az egyes adatelemek függetlenek, nincs közöttük kapcsolat. Az elemek között nincs sorrend, mindegy, hogy hol helyezkednek el egymáshoz képest. Az egyes elemekről meg tudom mondani, hogy az adatszerkezetnek elemei-e, még ha megkeverem őket, akkor is.

Halmaz (set)

Matematikai halmazfogalom megjelenése adatszerkezet szinten.

Multihalmaz (multiset, bag)

Olyan speciális halmaz, amely a halmaztól eltérően ismétlődést is megenged, tehát lehetnek benne azonos elemek.

Mindkét adatszerkezet dinamikus és homogén. Beszélhetünk üreshalmazról is, viszont végtelen halmaz, mint adatszerkezet nem létezik.

Speciális adatszerkezeti műveleteik vannak: ($\in$, $\cup$, $\cap$, \ | IN, +, *, –)

- $\in$, IN: Segítségükkel meg lehet állapítani, hogy egy adott elem benne van-e a halmazban, vagy sem. Eredménye mindig logikai érték.
- $\cup$, +: A multihalmazban az unió az összes elem unióját jelenti, tehát ha pl. az egyik halmazban "a" ötször szerepel, a másik halmazban háromszor, akkor uniójukban nyolcszor fog. Ha $A \cup B = C$, akkor $|A| + |B| = |C|$.
- $\cap$, *: Ugyanez igaz a metszetképzésre is, tehát az előbb említett két halmaz metszetében "a" háromszor fog szerepelni.

A következőkben definiált műveletek a multihalmazra is igazak.

Műveletek

Létrehozás:

- Megadhatok egy predikátumot (feltételt). Azok az elemek lesznek a halmaz elemei, amelyek igazzá teszik a feltételt, pl. X={200-nál nem nagyobb pozitív egész számok}
- Felsorolhatom a halmaz elemeit, azaz konkrétan megadom őket, pl. Y={fekete, piros, kék, fehér, sárga}
- Van egy üreshalmazom, és ehhez unióval pakolom hozzá az elemeket, és így jön létre a halmaz.

Bővítés: Unióképzéssel.
Törlés: Különbségképzéssel.
Csere: Nem létezik, mert nem tudom megfogni az elemeket. Szimulált cserét hajthatok végre, ha kitörlök egy elemet, és újat viszek be helyette.
Keresés: Nincs, mert nem tudom megfogni az elemeket.
Rendezés: Nem értelmezhető. A rendezett halmaz nem halmaz, hanem egy másik adatszerkezet. Gyakran nincs is értelme a rendezésnek.
Bejárás: Nincs, mert nem tudom megfogni az elemeket.
Feldolgozás: A definiált műveletek alkalmazása.

A halmazok között értelmezhető az összehasonlítási művelet. Két halmaz egyenlő, ha ugyanazokat az elemeket tartalmazza, egyébként nem. Ha az egyik halmaz részhalmaza a másiknak, akkor az a kisebb: $H \subseteq K \rightarrow H < K$.

A halmazok reprezentációja a karakterisztikus függvényen alapul, általában folytonos tárolást alkalmaznak. A reprezentációs szinten sorrend van, amikor a halmaz elemeit egy sorrendiséggel

rendezett tárhelyre képezzük le. Annyi bitet foglalunk le, amennyi halmazelem lehetséges. Mivel a tár véges, a tárhely kicsi, ezért tudom, hogy maximálisan hány eleme van a halmaznak, és sorrendbe rakom őket. A biteket 1-re állítom, ha az adott elem benne van a halmazban, 0-ra, ha nincs benne. Az üreshalmaznak minden bitje 0, ha pedig minden elem benne van a halmazban, akkor minden bit 1-es. Ez a karakterisztikus függvény.
Pl. S = {hexadecimális számjegyek} Karakterisztikus függvénye: 16 bit = 2 bájt. Legyen S = {1, 4, 8, A}. A halmaz elemeinek feltételezett sorrendje: 0, 1, … , E, F. Ekkor a karakterisztikus függvény:

0	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0

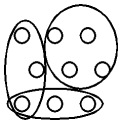
Ez az S halmaz reprezentációja. Az, hogy a hogy a halmaz elemei hol vannak, az implementáció dolga, a felhasználónak nem szabad, nem illik tudni. A karakterisztikus függvény előnye az, hogy az összes halmazművelet visszavezethető logikai műveletre.

$$x \in (H_1 \cap H_2) = x \in H_1 \vee x \in H_2$$
$$x \in (H_1 \cup H_2) = x \in H_1 \wedge x \in H_2$$
$$x \in (H_1 / H_2) = x \in H_1 \wedge x \notin H_2$$

Az „eleme” műveletet a processzor eltolással, shift-tel valósítja meg. Pl. $C \in S$: ekkor a processzor 10 bittel előre tolja a karakterisztikus függvényt, és megnézi, hogy 0 vagy 1 szerepel az adott biten. Ez gyorsan végrehajtható, de hibája, hogy a karakterisztikus függvény nagy tárhelyet foglalhat el. Ezen ok miatt az implementációk százas nagyságrendnél szabják meg a halmaz maximális elemszámát. Pl. Pascalban a halmaznak 256 db eleme lehet, ami 32 bájtot foglalhat el.

Asszociatív adatszerkezetek

Olyan halmaz vagy multihalmaz, amelynek az elemeiből részhalmazokat tudunk kiválasztani, ezek átfedhetik egymást. Lényeges, hogy a részhalmazokat milyen szempont szerint választjuk ki. A részhalmazokat meg kell tudni fogni, különböztetni egymástól.

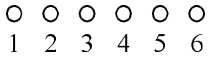


A részhalmazok lehetnek pontosan egyeleműek, így meg tudom fogni bármelyik elemet (pl. tömb) vagy tetszőlegesek, így átfedhetik egymást.

Tömb

A legismertebb, a leggyakrabban alkalmazott statikus adatszerkezet. A részhalmazok egyeleműek és diszjunktak, és az osztályozásba minden elem be van vonva, ezért minden adatelem megfogható.

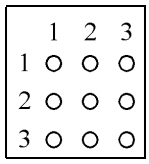
A tömb minden eleme egész számok sorozatán keresztül érhető el (ezek az egész számok a tömbindexek), az indexek alkalmazása teszi azt lehetővé, hogy egyelemű részhalmazokat képezzünk.



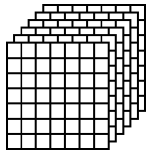
Az index tetszőleges egész szám lehet, általában egytől kezdődik. Van felső és alsó határa, amelyek az indextartományt definiálják. A tömbhöz hozzátartozik, hogy minden dimenzióban rögzítem az indexek kezdő és végértékét, rögzítek egy intervallumot, ez meghatározza az elemek számát. Az egyes dimenziókban lévő elemek számának szorzata adja a tömb elemszámát.

Ha az adatelemekből egy adott elem kiválasztását egy index segítségével meg tudom valósítani, akkor egydimenziós tömbről vagy vektorról beszélünk.

Ha minden elemet két index segítségével lehet kiválasztani, akkor kétdimenziós tömbről, mátrixról beszélünk. A kétdimenziós tömb sorokból és oszlopokból áll. Eleminek indexelésekor előbb a sor-, majd az oszlopindexeket adjuk meg.



Ha minden elemet három index segítségével lehet kiválasztani, akkor háromdimenziós tömbről beszélünk. A háromdimenziós tömb lapokból áll.



Tetszőleges dimenziójú tömbökről beszélhetünk, mint adatszerkezetekről. A tömb olyan adatszerkezetnek tekinthető, amelyben az elemek értékei egymástól függetlenek, és az elemek a szerkezetben elfoglalt helyük szerint vannak definiálva, így rendezettség az elhelyezés szerint van. Az adatszerkezetet értéktől függetlenül a szerkezet definiálja.

Műveletek

Létrehozás: A szerkezet létrehozása, nem az értékeké. Megmondom a dimenziók számát és minden dimenzióban az indextartományt. A szerkezetet hozom létre, és mivel a szerkezet statikus, így már az elején eldől, hogy a tömb hány elemet tartalmaz. Ebbe a szerkezetbe pakolom bele az elemek értékét.

Bővítés: Nem létezik olyan, hogy új helyre új elem bevitele, mivel a tömb statikus, az adatelemek számát létrehozáskor fixáltam. A bővítés szűkebb változata az, hogy az üres szerkezeti helyekre adatelemet viszek be, azaz átvitt értelemben: megadom azoknak az adatelemeknek az értékét, amelyeket eddig még nem adtam meg.

Törlés: Erről sem beszélhetünk, mert az adatelemek számát nem tudom csökkenteni. Csak logikai törlésről beszélhetünk, mikor az adatelemek értéket egy speciális értékre állítom.

Csere: Indexek alapján bármelyik elemet kiválaszthatom, és értékét felülírhatom.

Elérés: közvetlen. Bármelyik elem a többitől függetlenül megfogható, az indexekkel hivatkozom rá.

Rendezés: Általában egydimenziós tömbnél foglalkozunk vele, ekkor az összes rendezési módszer alkalmazható.

Keresés: Mindenféle tömbnél értelmezhető, és értelmezése reprezentáció-függő. Ha a feltételek adottak, akkor egy dimenziós tömbnél (itt van jelentősége) az összes keresési algoritmus alkalmazható.

Bejárás: Értelmezhető, reprezentáció-függő. Többdimenziós tömböknél igazán érdekes.

Feldolgozás: Az indexeken alapul, azon, hogy segítségükkel bármely elem közvetlenül elérhető, feldolgozható.

A tömböknél a folyamatos és szétszórt tárolás egyaránt alkalmazható. Általában folytonos tárolást alkalmaznak.

A tömb leképzése folytonos társzerkezetre a következőképpen történik. Legyen a kiindulási tömb:

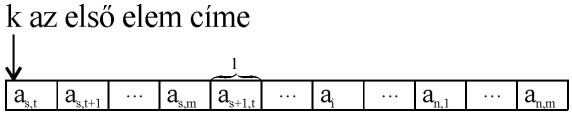
$$\begin{pmatrix} a_{s,t} & a_{s+1,t} & \dots & a_{n,t} \\ a_{s,t+1} & a_{s+1,t+1} & \dots & a_{n,t+1} \\ \vdots & & \ddots & \\ a_{s,m} & & & a_{n,m} \end{pmatrix}$$

A kétdimenziós tömb első indexeinek tartománya: $s...n$ és második indexeinek tartománya: $t...m$. Képezzük le ezt a kétdimenziós tömböt folytonos társzerkezetre. A leképezés sor- vagy oszlopfolytonosan történik. Az implementációk nagy része a sorfolytonosat választja. Ez a következőt jelenti:

- sorfolytonosan: $a_{s,t}$; $a_{s,t+1}$; $a_{s,t+2}$; … ; $a_{n,m-1}$; $a_{n,m}$,
- oszlopfolytonosan: $a_{s,t}$; $a_{s+1,t}$; $a_{s+2,t}$; … ; $a_{n-1,m}$; $a_{n,m}$.

Elhelyezem az első sor elemeit, utána a második sor elemeit és így tovább. Az oszlopfolytonos tárolás ugyanaz, mint a sorfolytonos, csak oszlopokat veszünk sorban. Ha több mint kétdimenziós tömb van, akkor az indexek végigfutnak az indextartományokon. Sorfolytonos tárolásnál a legutolsó index fut végig a leggyorsabban és az első a leglassabban, oszlopfolytonosnál ez fordítva van.

A tömb bejárása és a tömbben való keresés függ attól, hogy a tömb oszlop- vagy sorfolytonosan van tárolva.



Kérdés az, hogy ha ismerem egy adott tömbelem indexeit, akkor hogyan tudom meghatározni a tárbeli helyét. Ismerni kell a tömb kezdőcímét (k), egy elemének hosszát bájtban (l) és az indexeinek alsó és felső határát minden dimenzióban. Így

- egydimenziós tömbnél az i. elem címe : $k + l * (i - s)$,
- kétdimenziós tömbnél az $a_{i,j}$ elem címe: $k + l * (i - s) * (m - t + 1) + l * (j - t)$, sorfolytonos tárolás esetén.

Ezeket nevezzük címfüggvényeknek.

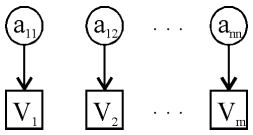
Minden adatszerkezet (a többdimenziós tömb is) szimulálható egydimenziós tömb segítségével.

Mátrixok kezelése:

I. Abban az esetben, ha a mátrix háromszög mátrix, azaz a főátló alatt vagy fölött minden elem nulla, a nullértékű elem tárolását meg lehet spórolni. A mátrixot kezeljük vektorként, képezzük le olyan egydimenziós tömbre, amely csak a nem nulla elemeket tartalmazza. A vektor elemeinek száma: $m = n * (n + 1) / 2$.

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{pmatrix}$$

A mátrix leképezése a következő módon történik (sorfolytonos leképezés):

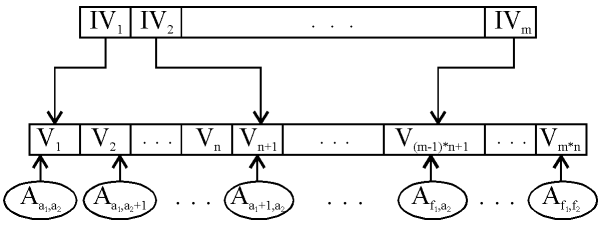


Ekkor $a_{ij} = \begin{cases} V_t & j \geq i \\ 0 & j < i \end{cases}$, $t = \frac{j * (j-1)}{2} + i$.

Általánosan a kétdimenziós tömb leképezése egydimenziós tömbre indirekt indexeléssel történik. Egy A tetszőleges kétdimenziós tömböt képezünk le egy V vektorra. A indexhatárai: (a_1 … f_1, a_2 … f_2). V mérete $m * n$, ahol $m = (f_1 - a_1 + 1)$, $n = (f_2 - a_2 + 1)$.

Hozzunk létre egy IV indexvektort, amely elemeinek száma m.

Az IV k. eleme: $1 + (k - 1) * n$, ez nem más, mint az A mátrix ($a_1 - 1 + k$). sora első elemének V-beli indexe.



Ez egy sorfolytonos leképezés, $A_{i,j} = V_t$, $t = IV_i - a_{1+1} + j - a_2$.

II. Sokszor kell olyan mátrixot használni, amelynek sok eleme nulla, vagy sok azonos értékű eleme van. Ezeket nevezzük ritka mátrixoknak. A kérdés az, hogy ezeket hogyan tudjuk kezelni.

1. Háromsoros reprezentáció

A három egydimenziós tömb azonos elemei írnak le egy-egy elemet sorfolytonos ábrázolásban.

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 6 \\ 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

$$\begin{aligned} \text{SOR} &= (1, 1, 1, \boxed{2}, 5) \\ \text{OSZLOP} &= (1, 2, 6, \boxed{2}, 6) \\ \text{ÉRTÉK} &= (1, 2, 6, \boxed{4}, 2) \end{aligned}$$

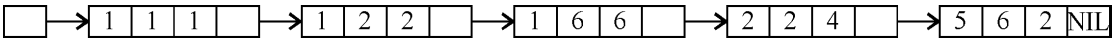
C-ben az *A* mátrix leképezése háromsoros reprezentációra így néz ki:

```
void lekepez()
{
  int i, j, k = 0;

  for ( i = 1; i <= N; ++i)
    for ( j = 1; j <= M; j++)
      if (A[i][j] != 0) {
        ++k;
        SOR[k] = i;
        OSZLOP[k] = j;
        ERTEK[k] = A[i][j];
      }
}
```

Fordítva, amikor visszakeresünk egy elemet: pl. $A_{i,j}$-t

```
int keres(int i,int j)
{
  int l;

  for ( l = 1; l <= k; l++) {
    if (SOR[l] == i && OSZLOP[l] == j)
        return ERTEK[l];
    if (SOR[l] > i) return 0;
  }
}
```

A keresés lineáris, mert vagy megtaláljuk a keresett elemet, vagy nagyobb értékűt találunk és akkor 0 lesz a függvény visszatérési értéke. A probléma az, hogy nem tudjuk a sor, az oszlop, és az érték maximális értékét. A tömb statikus adatszerkezet, ezért több helyet kell foglalni, így marad üresen lefoglalt tárrész. A másik gond az, hogy nem lehet közvetlenül elérni az elemet, keresést kell alkalmazni. Ez nem oldható meg, viszont az első probléma szétszórt tárolás esetén megoldható. A tömb statikus voltát dinamikusra viszem.



Sorok egészben keresésére, mátrix szorzására remek.

2. Négysoros reprezentáció

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 6 \\ 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

SOR = (1, 1, 1, 2, 5)
OSZLOP = (1, 2, 6, 2, 6)
ÉRTÉK = (1, 2, 6, 4, 2)
MUTATÓ = (0, 4, 5, 0, 0)

A mutató vektor dolga, hogy lerendezze az oszlopok kezelését. Értékét úgy állítjuk be, hogy a megfelelő helyre beírjuk a négysoros reprezentáció azon indexét, amelyik megmondja, hogy mely indexnél található az oszlop következő nem nulla eleme. (pl. A második oszlopban a következő nem nulla elem a négyes. A négyes elemet a négysoros reprezentációban a negyedik elem írja le, így valóban négy az index. Ugyanez látható a 6. oszlopnál.) Ennek oszlopfolytonos feldolgozásnál van jelentősége, mert lehetővé teszi, hogy ne kelljen végignézni szekvenciálisan az egészet, hanem megmondja, hogy hol a következő elem az adott oszlopban. A négy sorhoz még két vektort hozzáveszek, melyeket jelöljünk S-el és O-val. Ez a két vektor úgy van feltöltve, hogy a vektor i. eleme megadja, hogy hol helyezkedik el az i. sor, illetve oszlop első nem nulla eleme a SOR illetve az OSZLOP vektorban, így közvetlenül fel tudok dolgozni bármilyen mátrixot.

SOR = (1, 1, 1, 2, 5)
OSZLOP = (1, 2, 6, 2, 6)
ÉRTÉK = (1, 2, 6, 4, 2)
MUTATÓ = (0, 4, 5, 0, 0)
S = (1, 4, 0, 0, 5)
O = (1, 2, 0, 0, 0, 3)

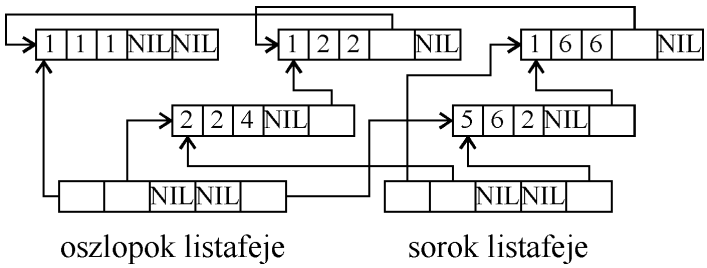
Az S-ből látható, hogy a mátrix 3. és 4. sora csupa nulla.
Az M oszloppal rendelkező ritka mátrix oszlopainak feldolgozása a következőképpen történhet:

```
void bejar()
{
  int i, j;

  for (i = 1; i <= M; i++) {
    j = O[i];
    while (j) {
      feldolgoz(SOR[j],OSZLOP[j],ERTEK[j]);
      j = MUTATO[j];
    }
  }
}
```

Az algoritmus folytonos tárolás esetére vonatkozik. A ritka mátrix tárolásánál is alkalmazható a szétszórt tárolás, láncolás. Ehhez egy multilista szerkezetet használhatunk.

BALRA: sorok láncolása jobbról balra.
FEL: oszlopok láncolása lentről fölfelé.

Listafejek is kellenek, így a példa a következőképpen néz ki:



A tömbök igazán nagy jelentősége az, hogy egydimenziós tömbök segítségével bármely homogén adatszerkezet szimulálható (akár saját maga is).

Minden programozási nyelv ismeri a tömböt. Egydimenziós tömbök segítségével a szétszórt reprezentáció is megvalósítható, ld. program.

Táblázat (table)

Asszociatív, dinamikus adatszerkezet, amelyben minden elem két részből áll.

KULCS	ÉRTÉK

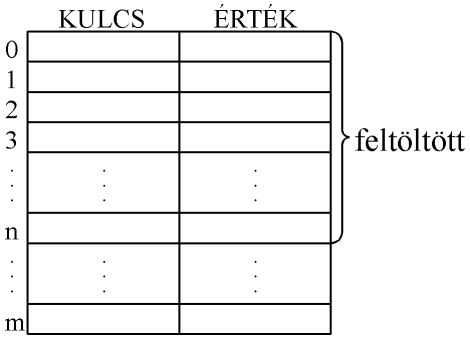
A kulcs és az érték típusa bármilyen lehet, de egy táblázaton belül a kulcsok, értékek típusa megegyező kell, hogy legyen. A kulcs és az érték típusa különböző lehet. Minden táblázati elem kulcsértéke egyedi, így ez a kiválasztási szempont. A táblázat nem más, mint egydimenziós tömb általánosítva, ahol bármely elem a kulcson keresztül érhető el (pl. statikus helyett dinamikus). Reprezentációja lehet folytonos és szétszórt is, de általában folytonos ábrázolást alkalmaznak. Gyakran úgy tárolják, hogy a kulcsot és az értéket leképezik egy-egy egydimenziós tömbre. Különböző szervezésű táblázatok vannak.

I. Soros táblázat

A kezelés szempontjából ez a legegyszerűbb táblázat. Akkor érdemes soros táblázatot használni, ha az egyes elemek feldolgozási valószínűsége közel azonos.

Műveletek

Létrehozás: Az elemek az érkezés sorrendjében kerülnek a táblázatba.
Bővítés: A táblázat végére lehet.
Törlés: Meg lehet oldani logikai törléssel és a szabadhelyek elfogyása után szemétgyűjtéssel, és meg lehet oldani fizikai törléssel is. A fizikai törlés az, amikor a törlendő elem mögött lévő elemeket előretolom. A baj csak az, hogy így sok elemet kellene mozgatni, ezért helyette azt alkalmazzák, hogy kitörlik az elemet és a legutolsó elemet a helyére írják. Így csak egy elemet kell mozgatni.
Csere: Minden elem felülírható. Megkeresem az adott kulcsú elemet és felülírom az értékét. Soros táblázatnál nem jelent gondot, ha az értéket és a kulcsot is felül akarom írni, de figyelni kell, hogy a kulcs az előzőekben ne szerepeljen.
Rendezés: Nem értelmezett.
Keresés: Kulcsra vonatkozó teljes keresés.
Bejárás: Soros (az elemek elhelyezésének sorrendjében).
Feldolgozás: Kulcs alapján.



Keresésre Wirth algoritmustechnikáját alkalmazzuk, amit strázsatechnikának nevezünk. Az algoritmus feltételezi, hogy van még legalább egy szabad hely. A strázsatechnika lényege, hogy *m* méretű tábla esetén az elemek végére (*n* elem esetén az *n* + 1. helyre) elhelyezem a keresett elemet (*k*), és így jutok a tábla végére. A *k*-t így mindig megtalálom: ha a legvégén találom meg, akkor az azt jelenti, hogy nincs benne a táblázatban, ha még a vége előtt megtalálom, akkor benne van. A táblázat elemeinek a száma ezzel a manipulációval nem változott meg, mert ha jön egy új elem, a szabad helyre írom, így törlődik a strázsa (kulcs[*n*+1]).

Teljes keresés algoritmusa:

```
int keres(int k)
{
   int i;

   KULCS[n] = k;
   for ( i = 1; k != KULCS[i]; i++);
   if (i == n) {
      printf("Nincs benne.\n");
      return -1;
   }
   else return i;
}
```

Bővítésnél arra kell vigyázni, hogy az új elem kulcsa olyan legyen, hogy addig még ne szerepeljen a táblázatban.

```
void beszur(int k, int x)
{
   int i;

   if (n > m) {
      printf("Betelt a tablazat!\n");
      return;
   }
   KULCS[n] = k;
   for ( i = 1; k != KULCS[i]; i++);
   if (i == n) {
      ERTEK[i] = x;
      n++;
   }
   else printf("Mar van ilyen kulcsu elem.\n");
}
```

Törlés: A *k* kulcsú elemet akarjuk törölni. Ha megtaláltuk, akkor kitöröljük az elemet és beírjuk a helyére az utolsó elemet.

```
void torol(int k)
{
   int i;

   if (n == 1) {
      printf("Ures a tablazat!\n");
      return;
   }
   KULCS[n] = k;
   for ( i = 1; k != KULCS[i]; i++);
   if (i == n)
      printf("Nincs torlendo elem.\n");
   else {
      KULCS[i] = KULCS[n - 1];
      ERTEK[i] = ERTEK[n - 1];
      n--;
   }
}
```

Soros táblázatok akkor alkalmazhatók jól, ha sok elemet akarunk feldolgozni. Ha feltesszük, hogy minden elemet ugyanolyan gyakorisággal dolgozunk fel, akkor ideális, de ez a gyakorlatban persze, hogy nincs így. Az lenne a jó, ha a gyakrabban feldolgozott elemet gyorsabban elérnénk, vagyis a táblázat elején lenne. Ezen okok miatt jött létre az önátrendező tábla.

II. Önátrendező tábla

Az egyes elemek feldolgozási valószínűsége különböző. Az önátrendező tábla azt a gondolatot valósítja meg, hogy a leggyakrabban használt elemek a tábla elején legyenek. A megoldás nagyon egyszerű. Ha egy elemet feldolgozunk, akkor azt helyezzük a táblázat elejére. Ez a folyamat sok lépés után ahhoz vezet, hogy a legelső helyen a leggyakrabban feldolgozott elem fog állni. Reprezentációjánál szétszórt ábrázolást alkalmaznak, mert itt az átrendezés egyszerű.

III. Rendezett tábla

Ez általában kulcs alapján történő növekvő rendezettséget jelent. A feldolgozást segíti a rendezés. Az elemeket nem azonos gyakorisággal dolgozom fel, és nem mindegyik elemet.

Műveletek

Létrehozás: Beszúrásos rendezéssel történik.
Bővítés: Beszúrásos rendezéssel történik.
Törlés: Meg lehet oldani logikai törléssel, és a szabadhelyek elfogyása után szemétgyűjtéssel, és meg lehet oldani fizikai törléssel is. A fizikai törlés folyamatos tárolásnál az, amikor a törlendő elem mögött lévő elemeket előretolom. Szétszórt ábrázolásnál igen egyszerűen megoldható.
Csere: Megkeresem és felülírom az adott elemet. A kulcs alapján cserélem az értékrészt, a kulcs cseréje nem engedélyezett. Ha azt is meg akarom cserélni, akkor fizikai törlést és bővítést kell alkalmazni.
Rendezés: Nincs, mert a tábla rendezetten jön létre.
Keresés: Mindkét reprezentációnál alkalmazható a lineáris keresés, folytonosnál a bináris keresés is.
Bejárás: Szekvenciális, logikai sorrend szerint.
Feldolgozás: Kulcs alapján történik.

A karbantartási műveletek folytonos ábrázolásnál lassabbak, míg a feldolgozás gyorsabb a bináris keresés miatt. Szétszórt ábrázolásnál a karbantartás (bővítés, törlés) kellemes, de a feldolgozás nyűgös.

Bináris keresés folytonos ábrázolásnál:

```
int keres(int k)
{
   int min = 1, max = n – 1, i = (min + max) / 2;

   while (max >= min && k != KULCS[i]) {
      if (k < KULCS[i]) max = i - 1;
      else min = i + 1;
      i = (min + max) / 2;
   }
   if (k == KULCS[i]) return ERTEK[i];
   else return -1;
}
```

Az algoritmus feltételezi, hogy legalább egy elem létezik. A fordítóprogramok több táblázatot használnak. A memória véges, *n* véges. A soros táblában az elemek elhelyezkedését az idő szabja meg, önátrendezőnél a hivatkozási prioritás, rendezettnél a kulcs értéke. A kulcs tetszőleges lehet.

A táblázatok közvetlen elérést nem tesznek lehetővé, hanem keresést kell alkalmazni. Ha közvetlen elérést akarunk készíteni, az azt jelenti, hogy a kulcsból meg tudjuk mondani az érték címét.

IV. Kulcstranszformációs táblázatok

Mindig folytonos tármegjelenése van. Kulcs: *k*; *f(k)* egy függvény, amelyik a *k*-t leképezi a címre (minden kulcshoz hozzárendel egy címet, meghatározza az elemek helyét a táblában). Az *f* egy hash-függvény, a technika pedig a hashing (randomizálás). Az lenne a jó, ha a függvény kölcsönösen egyértelmű lenne, de ez a gyakorlatban ritka, mint a fehér holló. Általában a függvények egyértelműek, ami azt jelenti, hogy különböző kulcsokhoz ugyanazt a címet rendelhetik hozzá. Azon kulcsokat, amelyekhez ugyanazt a címet rendeli, szinonimáknak (túlcsordulásnak) nevezzük.

A kulcsoknak típusa van, így létezik a kulcsoknak egy elvi darabszáma, amit a kulcs típusa megszab. A gyakorlatban általában az elvi lehetőségek töredékét használjuk fel. Ha az elvi és a gyakorlati lehetőségek száma megegyezik, vagy a gyakorlati lehetőségek száma egyenletesen oszlik meg az elvi lehetőségeken belül, akkor kölcsönösen egyértelmű leképezést alkalmazhatunk. Ez a ritkább, általában az elvi lehetőségek száma nagyságrendekkel nagyobb, mint a gyakorlati lehetőségek száma, és az elhelyezés sem egyenletes. (pl. a telefonszámok hatjegyűek. Egy körzetben 1000000 lehet a telefonszámok száma, de ebből lehet vagy 10000 ténylegesen. Ezen belül hozzá kell számítani, hogy az elhelyezés sem egyenletes.) Ilyenkor egyértelmű hash-függvényt alkalmazunk. Egy hash-függvénytől elvárjuk, hogy:

- A kulcsokat be kell transzformálnia egy sorszámintervallumba ([0 – *n*]), pontosan oda, ahova tárat lefoglalok.
- A kulcsokat lehetőleg egyenletesen szórja szét, ne legyen sok szinonima.

A kulcs lehet numerikus vagy szöveges. Ha szöveges, akkor az első lépésben hozzárendelünk numerikus értéket, pl. a karakterek belső kódját. Ha megválasztottuk a kulcs típusát, az elvi lehetőségek száma eldől. A továbbiakban csak numerikus kulcsokkal foglalkozunk.

A valóságban:

1. Minden kulcs elő is fordul (vagy majdnem mind)
2. Nagyságrendi különbség van az elvi lehetőségek és a ténylegesen előforduló alkalmazott kulcsok között, de a gyakorlatban előforduló kulcsok valamilyen szabály szerint helyezkednek el az elvi lehetőségeken belül.
3. Nagyságrendi különbség és a gyakorlatban szabályszerűtlenség van az elvi lehetőségek és a ténylegesen előforduló alkalmazott kulcsok között.

1., 2.: Kölcsönösen egyértelmű hash-függvényt alkalmazunk. Az elvi lehetőségeket megsorszámozom.

4. Tudnék kölcsönösen egyértelmű függvényt hozzárendelni, de a tárhely üres lenne a nagyságrendi különbség miatt, és a tárolási kapacitás véges. Annyi helyet kell lefoglalnom, ahány kulcs a gyakorlatban előfordul (meg kell becsülni), ezután biztosítani kell, hogy bármilyen elvi kulcsú elemet le tudjunk képezni erre a kisebb területre (más-más kulcshoz ugyanazt a tárhelyet rendeli hozzá). Pl. hatjegyű vidéki telefonszámok.

Az elviekben előforduló bármely kulcshoz egy [0 … *m*] közé eső számot rendelek. Ebbe a 0 ... m tartományba viszonylag egyenletesen képezze le, szórja szét.

Bármely hash-függvény produkál szinonimákat, nincs univerzális hash-függvény. A továbbiakban konkrét kulcstranszformációs módszereket fogunk megnézni.

1. Prímszámmal való osztás

Legyen egy adott kulcs az: 1 2 3 9 9 9 9 1 7. Ez azt jelenti, hogy az elvi intervallum a 0-999999999-ig terjedhet. Tegyük fel, hogy a gyakorlatban 100000 lehetőség van, ezért a leképzés:

$$\underbrace{0 - 999999999}_{\downarrow}$$
$$0 - 99999$$

A gyakorlatban előforduló kulcsok száma *m*, így az intervallum: 0 - *m*. Válasszuk ki az *m*-nél kisebb legnagyobb prímszámot, és jelöljük *p*-vel. A transzformációnál a tényleges kulcsot osszuk el *p*-vel, és sorszámnak tekintsük a maradékot. Így a [0 – (*p*-1)] intervallumot kapjuk. Ez a módszer jól véletlenszerűsít, kevés a szinonima.

2. Szorzás módszere

A kulcsot képezzünk le a 0 - *m* tartományra. Legyen az *m* számjegyeink darabszáma *k*. A transzformációhoz vegyük az eredeti kulcsot és szorozzuk meg egy tetszőleges prímszámmal, vagy önmagával, vagy osszuk ketté és a két felet szorozzuk össze. Az így kapott számnak vegyük *k* db jegyét, és ezt tekintsük sorszámnak. Ez a transzformáció jól véletlenszerűsít, kevés szinonimát produkál.

3. Helyiérték kiválasztás módszere

A legegyszerűbb módszer, de nem véletlenszerűsít jól, sok a szinonima. Az eljárás lényege, hogy a kiindulási kulcsból véletlenszerűen kiválasztok *k* db számjegyet, és ez lesz a sorszám.

1 (2) (3) 9 (9) 9 (9) (1) 7

4. Hajtogatás

Legyen az eredeti kulcs az 5 4 3 2 1 5 4 3 2 1 6 1, és ebből maximum 4-jegyű sorszámot akarok képezni ($k = 4$). A kulcsot fölosztom k elemű csoportokra, és ha kell, a csoportok elejére vagy a végére nullá(ka)t írok. Ezután a csoportokat a középső csoport alá hajtogatom, és összeadom. A kapott összegből veszem az utolsó k db számjegyet, ez lesz a sorszám. Ezt a módszert szöveges kulcsok esetében alkalmazzák.

1 5 4 3
2 3 4 5
1 6 1 2
5 5 0 0

5. Eltolás

Ugyanaz, mint a hajtogatás, csak a csoportokat eltolva írom egymás alá.

1 5 4 3
5 4 3 2
2 1 6 1
9 1 3 6

6. Bázistranszformáció módszere

Legyen az eredeti kulcs a 7 6 1 3 4 5 0. Válasszunk egy 10-nél nagyobb prímszámot, pl. a 3-at, és jelöljük p-vel. Tegyük fel, hogy az adott szám a p számrendszerben van fölírva. Ezután konvertáljuk át p-ből a 10-es számrendszerbe.

$$7*13^6+6*13^5+1*13^4+3*13^3+4*13^2+5*13^1+0 = 36051314.$$

A kapott eredményből az utolsó k jegyet vesszük és az lesz a sorszám (51314). A bázistrsnszformáció egy igen közkedvelt módszer.

Adatbáziskezelőknél beépített hash-függvények léteznek.

Hogyan kezeljük a szinonimákat?

A szinonimák túlcsordult elemek formájában jelennek meg. Vegyük az első elemet, és helyezzük el a táblában. Ezután vegyük a következőt, és azt is helyezzük el. Tegyük fel, hogy a harmadik elemet oda kellene betenni, ahol az első elem van. A kérdés az, hogy mi történik ilyenkor.

	KULCS	ÉRTÉK
0		
1	×	×
2		
3	×	×
⋮	⋮	⋮
n		
⋮	⋮	⋮
m		

1. Nyílt címzés módszere

Ugyanúgy kezeli a túlcsordult elemeket, mint a nem túlcsordultakat. Ha olyan elem érkezik, amelyiknek a helye már foglalt, akkor a túlcsordult elemet megpróbálja a táblázat utána következő első üres helyére bepakolni. Fontos, hogy a táblázatban helyezi el az elemeket. Azon a helyen, ahova a leképezés történt, nincs információ arról, hogy volt-e ott túlcsordult elem, és ha volt, akkor az hova lett elhelyezve. Az első üres helyet általában egyesével lépkedve keressük meg, de haladhatunk másképpen is.

Pl. j = -m: az i. az a hely, ahol túlcsordult az elem.

j = j + 2

((i – 1 + |j|) % m) + 1

Ha a táblában végig nincs üres hely, akkor megyek a tábla elejére. Vagy találok üres helyet, vagy sem. (Általában a nyílt címzés feltételezi, hogy létezik üres hely. Később látni fogjuk, hogy mindig találok, de most feltételezhetjük, hogy létezik olyan eset is, amikor nincs üres hely.) Ha nincs üres hely, akkor visszatérek arra a helyre, amelyről elindultam.

Egy adott elem megtalálása úgy történik, hogy kulcs alapján leképezem, és ha az adott helyen nem az az elem van, akkor túlcsordulást feltételezhetek, így teljes keresést alkalmazok. A keresést megállítja, ha megtaláltam az elemet, vagy ha üres helyet találtam.

Ennek a módszernek az egyik hibája az, hogy a túlcsordult elemek kiszoríthatnak más elemeket, amelyek nem lennének túlcsordultak. A túlcsordult elemeknek nincs közvetlen elérése. Szokás az is, hogy a túlcsordult elemeket félrerakják, elsőnek bepakolják a nem túlcsordultakat, és csak második lépésként rakják be a félrerakott elemeket.

Nyílt címzésnél fizikai törlést nem tudunk megvalósítani, mert az újonnan keletkezett üres hely megállítaná a keresést, ezért logikai törlést alkalmazunk. Így minden elemnek három állapota lehet:

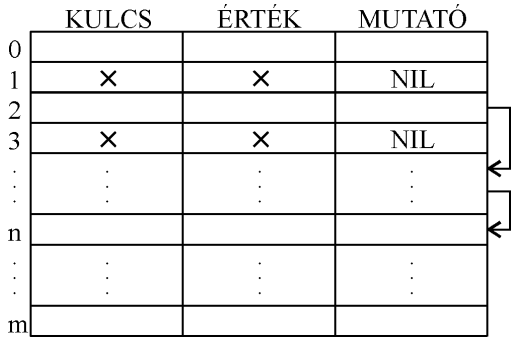
- üres,
- logikailag törölt (a táblázatban van, nem él),
- érvényes (a táblázatban van, él).

Az új elemet a logikailag törölt és az üres elem helyére is írhatom. Fizikai törlés a táblázat újraszervezésével történik (újonnan létrehozom a táblázatot).

Tudunk-e arról valamit mondani, hogy egy túlcsordult táblaelem viszonylag hamar kap-e üres helyet? Ellenkező esetben romlik a hatékonyság a teljes keresés miatt.

A megoldás az, hogy több tárolóhelyet foglalok le, mint amennyire szükség lenne, marad majd üres hely a végén.

2. Láncolás

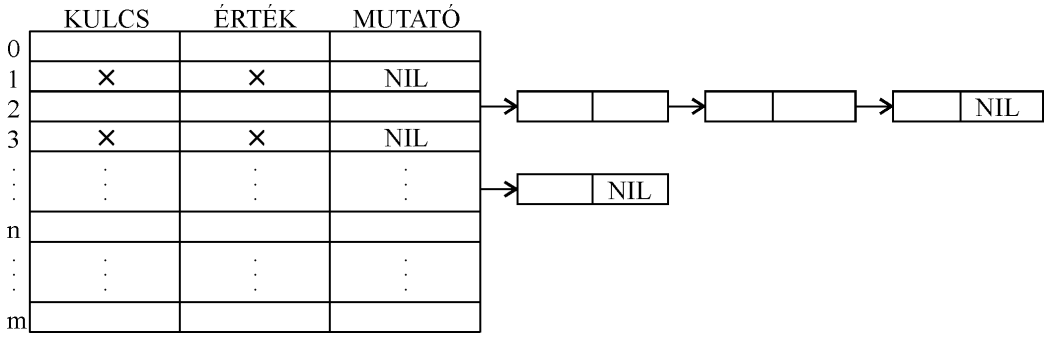


Információt helyez el a túlcsordulás helyén arról, hogy hol van a túlcsordult táblaelem. A KULCS, ÉRTÉK mező mellé egy harmadik mező kapcsolódik, a MUTATÓ mező. Ha az adott helyen nincs túlcsordulás, akkor azt NIL jelzi. Ha van, beírjuk, hogy hol van a túlcsordult táblaelem, így az adott helyről túlcsordult szinonimákat láncoljuk. Keresésnél a túlcsordult elem rögtön elérhető, mert tudjuk a helyét.

Kiküszöböli a teljes keresést, nem kell az összes elemet érinteni. A karbantartás bonyolult, a feldolgozás könnyebb.

3. Túlcsordulási listát alkalmaznak

A túlcsordult elemeket nem ugyanott helyezzük el, mint a nem túlcsordult elemeket.



A MUTATÓ listafej szerepet játszik, az egy adott helyről túlcsordult elemek össze vannak fűzve egy egyirányban láncolt listába. Keveredik a folytonos és a szétszórt ábrázolás, az eredeti elemeket folytonosan, a túlcsordultakat szétszórtan ábrázoljuk.

Egyszerűbb a karbantartás, mert multilista-szerkezetet kell kezelni.

Kulcstranszformációs táblák műveletei

Létrehozás: Meg kell becsülni, hogy a gyakorlatban ténylegesen hány kulcs fog előfordulni (*m*). Ezt követően le kell foglalnom legalább m tárhelyet, de általában 1.2 * *m* tárhelyet biztosítanak, azért hogy az 1. és 2. túlcsordulási technika alkalmazásakor biztosan legyen üres hely. Kiválasztok valamilyen hash-függvényt és valamilyen szinonimakezelési technikát. Ezután bepakolom az elemeket. Létrehozás után léteznek üres helyek és élő elemek.
Bővítés: A létrehozás technikájával kerülnek be a táblázatba az új elemek.
Törlés: Nyílt címzésnél csak logikai, másik kettőnél fizikai törlés is van.
Csere: Kulcs alapján keresem meg az elemet, az értékrészét felülírom.
Rendezés: Nincs.
Keresés: Nincs. Kulcs alapján tudom az elemeket feldolgozni, mert nincs fizikai sorrend. Logikai sorrend sincs. A keresésnek nincs alapja, csak kulcs alapján.
Bejárás: Nincs.
Elérés: Az esetek többségében közvetlen.
Feldolgozás: Kulcs alapján.

A kulcstranszformációs táblák előnyösek, ha viszonylag nagyméretűek, és viszonylag statikusak, azaz kevés a karbantartási művelet. A táblázat kevésbé érzékeny rá, hogy egy vagy több elemet dolgozok fel. A legjobb talán az a technika, amikor a táblázaton belül láncolok. Hátránya az, hogy a táblázaton belül nincs rendezettség, a feldolgozás és a karbantartás lassú. Ez a lassúság legjobban a gyorsan változó tábláknál jelentkezik, hiszen ott igen sok a karbantartási művelet.

Szekvenciális (lineáris) adatszerkezetek

Az adatszerkezet elemei mindig két másik elemmel vannak közvetlen kapcsolatban, kivéve a két szélső elemet.

Lista (list)

Olyan dinamikus adatszerkezet, amelyiknek létezik első és utolsó eleme (kivétel az üres lista) és minden adatelemnek létezik rákövetkezője (kivétel az utolsó elem), továbbá minden elemnek létezik megelőzője (kivétel az első elem). A lista elemeinek rendezettsége van, amelyet az elhelyezésük biztosít. A lista jelölése: $q = [x_1, x_2, \ldots , x_n]$

Speciális listafogalmak, ill. műveletek:

- üres *(empty)* lista: [ ],
- lista feje *(head)*: A lista első eleme.
- lista farka *(tail)*: Az a lista, amely az első elem elhagyásával keletkezik.
- lista vége *(end)*: A lista utolsó eleme.
- lista mérete: A lista elemeinek a száma. $|q| = n$.

A lista alapműveletei:

- Hozzáférés *(access)*: A lista elemei egyértelmű sorrendben vannak felfűzve. Megfogjuk valamely elemét. A listából az i. elem kiválasztása ($q[i] \rightarrow x_i$). Ha i nem eleme az $[1..n]$ intervallumnak, vagyis $i < 1$ vagy $i > n$, akkor a hozzáférés eredménye [], az üres lista.
- Allista *(sublist)* képzés: $q[i..j] \rightarrow [x_i, x_{i+1}, \ldots x_j]$ A listából tetszőleges részt választhatunk ki úgy, hogy megadjuk a kezdő- és a végelemet (i és j, $i < j$). Ha $i < 1$ vagy hiányzik, akkor az allista a lista elejétől indul. Ha j hiányzik vagy $j > n$, akkor az allista a lista végéig megy.
- Konkatenáció *(concetenation)*: Listák egyesítése. Adva van két lista: $q = [x_1, x_2, \ldots , x_n]$ és $r = [y_1, y_2, \ldots , y_m]$. Ekkor $q \,\&\, r = [x_1, x_2, \ldots, x_n, y_1, y_2, \ldots , y_m]$.

A szokásos adatszerkezeti műveletek ezen alapműveletek segítségével értelmezhetőek.

Létrehozás: Az üreslistából indulva konkatenációval rendeljük hozzá az elemeket, mint egyelemű listákat. A hozzáfűzés sorrendje adja a sorrendet a listában.

Bővítés: A q listát a k. elem után bővíteni akarjuk az x elemmel: $q' = q[\ldots, k] \,\&\, [x] \,\&\, q[k+1, \ldots]$. Allistával is bővíthetek, ugyanúgy, ahogy egy elemmel.

Törlés: Fizikai törlés. A k. elem törlése: $q' = q[\ldots , k-1] \,\&\, q[k+1, \ldots]$.

Csere: A k. elemet cserélem x-re: $q' = q[\ldots , k-1] \,\&\, [x] \,\&\, q[k+1, \ldots]$.

Rendezés: Értelmezhető abban az értelemben, hogy a listát az adatelemek értéke szerint bármely módszerrel növekvő vagy csökkenő sorrendbe rendezem a saját helyén, vagy létrehozáskor rendezett listát hozok létre.

Keresés: Teljes keresés, rendezett listánál lineáris vagy bináris keresés.

Elérés: Soros, szekvenciális, közvetlen.

Feldolgozás: A definiált műveletek alapján. Lényegesek azon műveletek formái, amelyek a lista elejét vagy végét érintik.

A szélső elemek kezelésére hat esetet különböztetünk meg:

- Acces head: Hozzáférés a legelső elemhez. $q(1)$.
- Push: Egy elemmel bővítem a listát az elején. $[x] \& q$
- Pop: Törlöm az első elemet. $q[2, \ldots]$

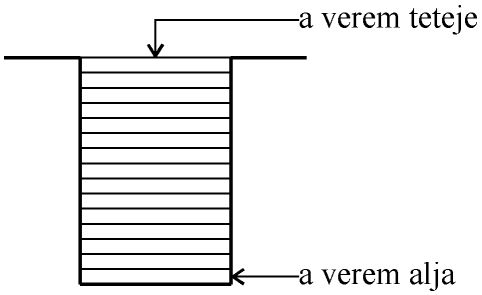
– Access end: Hozzáférés az utolsó elemhez. $q[|q|]$
– Inject: A listát a végén bővítem. $q\&[x]$
– Eject: Törlöm az utolsó elemet. $q[\ldots , |q|-1]$

A lista reprezentációja mind a kétféle módon történhet. A folytonos ábrázolást vektorral, a szétszórtat pedig egy kétirányban láncolt listával oldják meg.

Verem (stack)

A verem a leggyakrabban alkalmazott adatszerkezet a tömb után. A verem egy olyan speciális lista, ahol az előbb definiált műveletek közül csak az 1., 2., 3. van megengedve. Olyan adatszerkezet, amelyet a kezelési módjával definiálunk. A műveletek neve is hasonló:

– PUSH (bővítés az elején)
– POP (első elem törlése)
– ACCESS helyett TOP van (első elem elérése). Vannak olyan felfogások, amelyek a POP-ba beleértik a TOP-ot.



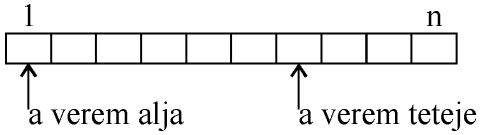
Létezik üresverem, és olyan verem is, amely tele van. Az üreslista vagy az üresverem onnan ismerhető fel, hogy a fej NIL.

Mivel az adatszerkezet homogén, azonos méretű elemeket pakolunk bele. Az elsőnek behelyezett elem a verem aljára kerül, majd erre pakoljuk a többit. Csak az utoljára betett elemhez tudok hozzáférni, az alatta lévőket csak úgy érhetem el, ha a felettük levőket eltávolítottuk. A vermet szokás LIFO *(Last In, First Out)* adatszerkezetnek is nevezni.

A verem reprezentációja lehet:

I. Folytonos

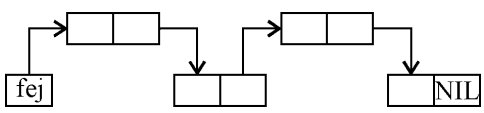
A vermet vektorral kezeljük, ahol az első elem mindig a verem alja. Használni kell egy mutatót, ami a verem tetejét mutatja.



Bővíteni, új elemet bevinni csak a verem tetejére lehet, ha van hely. Hozzáférni is csak a verem tetejéhez lehet. Ha egy belső elemet akarunk elérni, akkor törölni kell a felette levőket.

II. Szétszórt

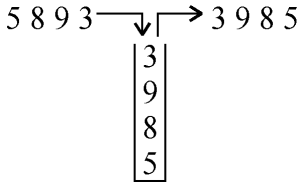
Egyirányban láncolt listával oldják meg. A listafej mindig az aktuális első elemre mutat. Ide kell beszúrni az új elemet adatfelvételnél.



A verem minden komoly szoftverben megjelenik. Általában olyan helyzetben alkalmazzák, amikor egy sorrend megfordítása a cél. A sorrend gyakran időrendi sorrendet jelent, meg akarom

változtatni a folyamatok bekövetkezésének a sorrendjét (pl. a hívási láncon visszafelé lépkedésnél). Rekurzív algoritmusoknál is használják.

Pl.



Műveletek

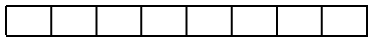
Létrehozás: Van az üresverem és az érkezés sorrendjében pakoljuk bele az elemeket.
Bővítés: PUSH művelet, definíció szerint csak a verem tetején.
Törlés: POP művelet, definíció szerint csak a verem tetején.
Definíció szerinti **keresés**, **rendezés**, **csere**, **bejárás** nincs.
Elérés: TOP, a verem legfelső eleme érhető el.
Feldolgozás: Az alapműveletekkel.

Sor (queue)

FIFO *(First In, First Out)* adatszerkezet. A sor egy speciális lista, ahol csak az 1., 3., 5. művelet van csak értelmezve. A sor tehát olyan adatszerkezet, amelyet kezelésével definiálunk. Vannak speciális elnevezések:

- Inject helyett PUT.
- POP helyett GET. (Felfogás kérdése, hogy a GET-be beletartozik-e az 1. művelet.)

Létezik üres és tele sor is. Új elemet a sor végére lehet bevinni, hozzáférni a sor elejéhez lehet.



A sor reprezentációjánál mindkét módszer megfelelő. Folytonos ábrázolásnál vektorra képezzük le a sort, szétszórtnál egyirányban láncolt listát alkalmaznak.

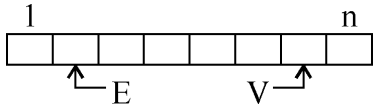
A folytonos ábrázolás megoldásai:

I. Fix kezdetű sor

A sor első elemének indexe 1, tehát rögzített, értéke x. Van egy indexmutató, ami a sor végét jelöli. Létezik üres sor és tele sor. Bővíteni a sor végén, hozzáférni az 1-es indexű elemhez lehet, ekkor az egész sort eggyel balra kell tolni. A többi eleméhez való hozzáférés azt jelenti, hogy az előtte lévő elemeket fizikailag törlöm.

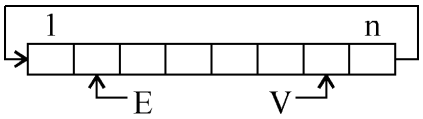
II. Vándorló sor

Két indexmutató van, a sor elejét és végét jelölik. Bővíteni az utolsó, V indexű elem után lehet, hozzáférni az első, E indexű elemhez lehet. Ha az első elem mögött lévő elemet akarom elérni, akkor az E indexűt törölnöm kell. Ha a rendelkezésre álló tárhely végére értem, az egész vektor visszatolódik a tárhely elejére.



III. Ciklikus sor

Két indexe van, a sor elejét és végét jelölik. Nincs nagytömegű adatmozgatás, a sor ciklikusan vándorol körbe. Ha a vektor utolsó eleme már foglalt, nem tolom vissza az egészet, hanem a vektor elején folytatom az új elemek bevitelét.



A sort olyan esetben használjuk, ha az adatokat időrendben való beérkezés szerint akarjuk meghatározott sorrendben feldolgozni. Szerepe kevésbé fontos, mint a veremé. Bármilyen adatszerkezet bejárása nem más, mint egy sorra való leképezése. A sor jelenik meg a puffereknél, ilyen pl. a billentyűzet puffer.

Ciklikus sor egy implementációja: Egydimenziós tömbbel valósítjuk meg. Kezeljük a szélsőséges eseteket. Az ábrát lásd fent.

SOR[1 ... *n*], E, V: indexmutatók. Ha a sor

- üres: E = V = 0,
- tele van: (E = 1 vagy V = N) vagy (E = V + 1).

PUT művelet (bővítés). Vigyük be az *x* elemet n + 1. elemnek!

```
void put(int x)
{
   if (v % n + 1 == e) printf("A sor tele van.\n");
   else {
      v = v % n + 1;
      SOR[v] = x;
      if (e == 0) e = 1;
   }
}
```

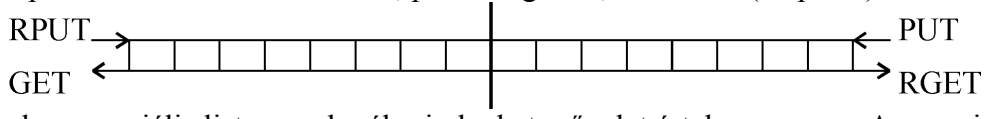
GET művelet. Vegyünk ki egy elemet!

```
int get()
{
   int a;

   if (e == 0) {
      printf("A sor ures.\n");
      return -1;
   }
   a = SOR[e];
   if (e == v) e = v = 0;
   else e = e % n + 1;
   return a;
}
```

Műveletek

Létrehozás: Az üressorból indulunk ki, és az elemek a beérkezés sorrendjében kerülnek be.
Bővítés: Csak a sor végén.
Törlés: Csak a sor elején
A **csere**, a **rendezés**, az **elérés** és a **bejárás** nincs értelmezve.
Feldolgozás: Az 1., 3., 5. műveletek alapján.

Vannak speciálisan kezelhető sorok is, pl. kétvégű sor, kettős sor (dequeue).



Ez egy olyan speciális lista, amelynél mind a hat művelet értelmezve van. A sor mindkét végén lehet bővíteni és mind a két végén lévő elemekhez hozzá lehet férni. A kétvégű sor tekinthető úgy,

mint két darab aljánál összeillesztett verem, így a verem és a sor által megoldható problémák megoldására egyaránt alkalmas. Vannak speciális esetei:

- input korlátozott kettős sor: Nincs értelmezve rá a 2. művelet (RPUT), azaz nem lehet az elején bővíteni, csak a végén. Elemet kivenni az elején és a végén egyaránt lehet.
- output korlátozott kettős sor: Nincs értelmezve rá az RGET művelet, hozzáférni csak a sor első eleméhez lehet, bővíteni viszont mindkét végén lehet.

Reprezentációra, implementációra nézve átvihetőek az előzőek. Memóriakezelési gondok megoldásánál szokták alkalmazni, dinamikusan.

Sztring (string)

A sztring olyan lista, amelynek elemeit az ábécé szimbólumai alkotják. Mi olyan sztringekkel foglalkozunk, amelyeknek elemei karakterek. Sok terület van, ami a sztringekkel foglalkozik:

- formális nyelvek,
- formális rendszerek: Ennél a kettőnél a sztring jelentése a lényeges.
- automaták,
- szövegszerkesztők: A sztring formája a lényeges.
- hipertext rendszerek: A sztringben előforduló szavakat vissza-, ill. kikereshetjük.
- magasszintű nyelvek.

A sztringnél van elérés, minden karakterhez hozzá tudunk férni. A sztringhez kapcsolódó fogalmak:

- allista: Részsztring.
- konkatenáció (egyesítés): Sztringek összefűzése.
- length: A sztring hossza, a sztring karaktereinek a száma.

Lényeges az üressztring kiemelkedő szerepe.

Műveletek

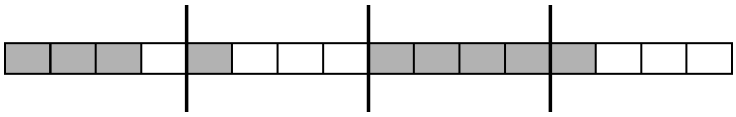
Létrehozás: Megadom a sztring összes karakterét.
Bővítés: Bárhol a sztringben részsztringgel bővíthetek, a listánál megbeszélt konkatenáció művelettel.
Törlés: Fizikai törlés, részsztring törlése.
Csere: A sztring részsztringjét cserélem egy másik részsztringre, azonosra, rövidebbre vagy hosszabbra.
A rendezés, az elérés és a bejárás nem értelmezett.
Elérés: Közvetlen elérés.
Keresés: Értelmezett, de részsztringet keresek.
Feldolgozás: Később látjuk.

A sztringek reprezentációjánál a folytonos és a szétszórt ábrázolás is megjelenik.

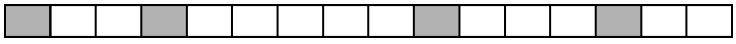
I. Folytonos ábrázolás

Ahány karakter, annyi bájt, belső kód alapján. Egy sztring esetén triviális. A kérdés több sztring tárolásánál a lényeges.

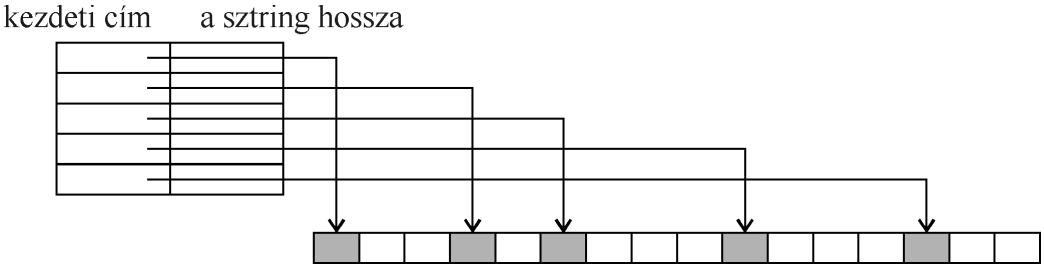
1. Fix hosszon tárolunk. A fix hossz akkora terület kell, hogy legyen, hogy a leghosszabb sztring is elférjen benne. Ez azt jelenti, hogy a rövidebb sztringeket ki kell egészíteni szóközökkel. Kezelése igen kényelmes, de nagyon nagy helyett foglal el, és kihasználatlan tárterületek maradnak.



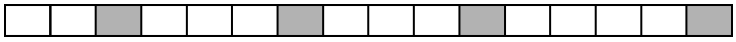
2. Változó hosszúságon tárolok. Minden sztringnek csak a szükséges helyet foglalom le: annyi bájtot, amennyi karakterből áll. A probléma az, hogy nem tudom, melyik meddig tart a sztring. Erre az a megoldás, hogy jelölnöm kell, hogy hol kezdődnek az egyes sztringek, ezért a sztring előtt letárolom a hosszát. Kezelése sokkal nyűgösebb, mint az előzőé, mert tudnom kell, hogy hol vannak a sztringhossz-jelzők, és meg kell tudnom különböztetni a tényleges karakterektől. A Pascal ezt a megoldást vallja.



Ennek a tárolásnak az egyik változata az, amikor a sztring hosszát nem előtte tárolom le, hanem a folyamatosan egymás után írt sztringekről létrehozok egy információs táblázatot, amely megmondja, hogy az egyes sztringek hol kezdődnek.

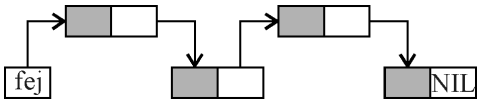


Egy másik változata az, amikor végjelt alkalmazok. Egy sztring végjeltől végjelig tart. Egy-egy sztringbeli elem jelzi a sztring végét.



II. Szétszórt ábrázolás

Minden láncolt lista elem egy-egy karaktert tartalmaz. A sok mutató miatt ez igen nagy helyet foglalna el, ezért ez csak egy elméleti megoldás, a gyakorlatban nem alkalmazzák.



Az előző helyett azt alkalmazzák, hogy karakter helyett részsztringet ábrázolnak. Ekkor a tördelés a probléma, kérdés, hogy mi történik, ha a sztring *n*-nel nem osztható hosszúságú (→ tört).

Mintaillesztés (pattern matching)

A feladat: Van egy sztringem és keresem, hogy egy adott másik sztring előfordul-e benne, és ha igen, akkor hol. Sok helyen előfordulhat. Ennek jelentősége például a képfeldolgozásban, valamint szövegben való keresés és csere esetén van.

Legyen az alapsztring a[1 ... N] (a = 10011101001010001010011100011 1), a keresendő sztring p[1 ... M] (p = 10100111).

Az illesztési algoritmusok:

I. Mezítlábas algoritmus

Vesszük az alap- és a mintasztringet, és karakterenként hasonlítjuk össze őket. Ha a vizsgált karaktereik megegyeznek, akkor mindkét sztringben a következő karakterrel folytatjuk a hasonlítást. Ellenkező esetben az alapsztringben visszalépünk oda, ahonnan a mintasztring

legutolsó hasonlítását kezdtük, és az alapsztring következő, valamint a mintasztring legelső karakterével kezdjük újra a vizsgálatot.

```
int Mintaill1(char A[], char P[])
{
  int i = 0,j = 0;
  while (j < m && i < n) {
    if (A[i] == P[j]) {
      i++;
      j++;
    }
    else {
      i -= j + 2;
      j = 1;
    }
  }
  if (j >= m) return i - m;
  else return i;
}
```

A függvény azzal az indexszel tér vissza, amely karaktertől kezdve a mintasztring először előfordul az alapsztringben, vagy (N + 1)-el, ha nem fordul elő benne. Az alapsztring karakterein az i megy végig, a mintasztring karakterein pedig a j. Ha nincs egyezés, i-t a következő karakterre állítom, j-t pedig 1-re. Ha megtalálok egy mintát, akkor további előfordulásokat kereshetek.

Pl. Az első és a második karakterpárok összehasonlításánál még egyezés van, de a harmadikban már eltérnek.

```
100111010010100010100111000111
10100111
```

Ezután az alapsztringnek a második, a mintasztringnek pedig az első karakterét veszi.

```
100111010010100010100111000111
 10100111
```

Ekkor már az első összehasonlítás után eltérést talál, és mindkét sztringben továbblép.

```
100111010010100010100111000111
  10100111
```

Az alapsztringben a 17. karaktertől kezdődően egyezik meg a részsztring a mintasztringgel.

```
100111010010100010100111000111
                  10100111
```

A mintasztringet addig tolom eggyel jobbra, amíg egyezést nem találok vagy a végére nem érek. A példában a függvény 17-el tér vissza.

Az algoritmus lassú, az összes lehetséges esetet végigvizsgálja.

II. Knuth-Morris-Pratt mintaillesztési algoritmus

Legyen az alapminta egy részlete az $a_{i-j+1} \ldots a_{i-1}\ a_i$ olyan, hogy az a_i kivételével, minden elem megegyezik a mintaelem első (j-1) elemével. Legyen a mintaelem jelölése: $p_1 \ldots p_{j-1}\ p_j$. Az illesztés során, ha tovább keresünk, vissza kellene menni az alapsztringből lévő rész elejére. Viszont az alapsztring részletének elejéről tudunk információt, azaz ismerjük, hogy az nem más, mint a minta eleje. Ezt az ismeretet tehát úgy kell felhasználni, hogy ezeket az elemeket ne vizsgáljam újra végig. Ehhez a következőt kell tenni: Vegyük a mintát, ami legyen nekünk

most p_1 ... p_M, és nézzük meg saját magához való illesztését, azaz megnézzük, hogy van-e a mintában olyan karaktersorozat, amely megegyezik a minta elejével. Ezt tároljuk le és így elérhetjük, hogy nem kell egyesével végiglépkednem a mintán. A letárolást egy KOV[0 ... M + 1] (M + 2) elemű vektorban végezzük, amelyet úgy kell feltölteni, hogy KOV[i] = *j*, ekkor a minta első (i - 1) karakterétől az utolsó (*j* - 1) karakter megegyezik a minta első (*j* - 1) karakterével. A következő keresésnél nem lépek teljesen előre, így lényegesebben kevesebb lépéssel dolgozom.

Első lépésként töltsük fel a vektort:

```
void initkov()
{
   int i, j;

   i = 1;
   j = 0;
   KOV[1] = 0;
   KOV[0] = 0;
   while (i <= m) {
      if (j == 0 || p[i] == p[j]) {
         i++;
         j++;
         KOV[i] = j;
      }
      else j = KOV[j];
   }
}
```

Az algoritmus működése a példán:

i	j	KOV[i]
1	0	0
2	1	1
2	0	1
3	1	1
4	2	2
5	3	3
5	1	3
5	0	3
6	1	1
7	2	2
7	1	2
8	2	2
8	1	2
9	2	2

```
int KMPill()
{
   int i = 1,j = 1;

   initkov();
   while (j <= m && i <= n) {
      if (j==0 || a[i]==p[j]) {
         i++;
         j++;
      }
      else j=KOV[j];
```

```
    }
    if (j > m) return i - m;
    else return i;
}
```

Az algoritmus működése a példán:

```
10011101001010001010011100 0111
10100111
  10100111
   10100111
    10100111
     10100111
          10100111
               10100111
                10100111
```

Az első algoritmus lépésszáma kb. N * M, a KMP-algoritmus lépésszáma N + M. Az algoritmust lehet úgy általánosítani, hogy az összes előfordulást keressem. Nem jelent alapvető problémát az, hogy a sztring egy tömbben van. Az algoritmus úgy is működik, ha a szöveget a perifériáról olvasom be. A mintaszöveg általában kicsi, így könnyen kezelhető.

III. Dömölki-algoritmus

Ez nem sztringkezelő algoritmusnak született meg a 60-as években, hanem az automata-elméleti algoritmusnak. Hatékonyabb algoritmus, mint az előzőek, és könnyebben programozható.

Legyen az alapsztring a = ABAFBADC és a mintasztring p = BAD. Első lépésben hozzunk létre a B logikai mátrixot. Címkézzük sorait azokkal a karakterekkel, amelyek előfordulhatnak a két sztringben, és oszlopait a minta szerinti karakterekkel.

	B	**A**	**D**
A	0	1	0
B	1	0	0
C	0	0	0
D	0	0	1
E	0	0	0
F	0	0	0

Legyen $B_{i,j} = 1$, ha az adott karakter az adott helyen előfordul, egyébként 0. Az algoritmus használ három M elemű logikai vektort: Q-t, U-t és V-t, amelyek induló értékei a következők: Q = (0 0 ... 0), U = (1 0 ... 0), V = (0 0 ... 1). Az algoritmus használ egy vermet is, ami a mintaillesztéshez nem feltétlenül szükséges. Mi a példánkban alkalmazni fogjuk. Jelöljük a vermet S-el. Az alapsztring karaktereit egyenként dolgozzuk fel: elhelyezzük őket a veremben, miközben vizsgáljuk, hogy van-e egyezés az alapsztring és a mintasztring elemei között. A Q vektor az illesztés adminisztrálására szolgál úgy, hogy $Q_l = 1$, ha a verem tetején lévő l db karakter megegyezik a minta első l db karakterével. Nyilván, ha $Q_M = 1$, akkor a verem tetején megvan a minta. Az alapsztring minden egyes karaktere kiválaszt a B-ből egy sort, ami legyen B_i. Értelmezve van egy olyan művelet, az R: $(a_1 \ldots a_M) \rightarrow (0\ a_1 \ldots a_{M-1})$, amely nem tesz mást, mint jobbra shift-el eggyel. Vegyük az alapsztring karaktereit, pakoljuk a verembe, közben Q értéke a következő lesz: $(R(Q) \vee U) \wedge B_i$. Minden lépés után vizsgáljuk, hogy $Q \wedge V$ egyenlő-e V-vel. Ha igen, akkor a minta a verem tetején van. Példa:

Q	**S teteje**	**A következő elem**
(000)	-	A
(000)	A	B
(100)	B	A
(010)	A	F
(000)	F	B
(100)	B	A
(010)	A	D
(001)	D	

Ha Q értéke (100), akkor az azt jelenti, hogy a verem tetején lévő karakter megegyezik a minta első karakterével. A verembe a köv. elemet rakjuk, és közben Q-t számoljuk úgy, hogy Q = (000) $\vee$ (100) & (010) = (000), Q = (000) $\vee$ (100) & (100) = (100) stb. Ha Q = (001), akkor a verem tetején ott van a minta. Ez nem implementáció, hanem algoritmus. Az algoritmust gépi kódban könnyű implementálni, és hosszú sztringeknél is kitűnően működik. Egyik előnye az, hogy míg a többi algoritmusnál annyiszor végig kell menni az alapsztringen, ahány mintát akarok keresni, míg itt egyszerre akárhány mintát illeszthetünk.

Pl. Legyen az alapsztring ABADCBECABF, és a keresett sztringek BAD, BED, CAB, AB. Ekkor U = (10010010010), V = (00100100101).

	BAD	BED	CAB	AB
A	010	000	010	10
B	100	100	001	01
C	000	000	100	00
D	001	001	000	00
E	000	010	000	00
F	000	000	000	00

Q	S teteje
(00000000000)	-
(00000000010)	A
(10010000001)	B
(01000000010)	A
(00100000000)	D
(00000010000)	C
(10010000000)	B
(00001000000)	E
(00000010000)	C
(00000001001)	A
(00000000101)	B
(00000000000)	F

Hierarchikus adatszerkezetek

Olyan adatszerkezetek, amelyek valamilyen értelemben a lista általánosításainak tekinthetőek. A hierarchikus adatszerkezetben egy elemnek akárhány rákövetkezője lehet, de minden elemnek csak egyetlen megelőzője létezik. Van egy kitüntetett eleme, amelynek nincs megelőzője, és több olyan eleme, amelynek nincs rákövetkezője.

Fa (tree)

Dinamikus, homogén adatszerkezet, amelyben minden elem megmondja a rákövetkezőjét.

Alapfogalmak:

A *gyökérelem* a fa azon eleme, amelynek nincs megelőzője. A legegyszerűbb fa egyetlen gyökérből áll. Mindig csak egy gyökérelem van, üres fában egy sem.

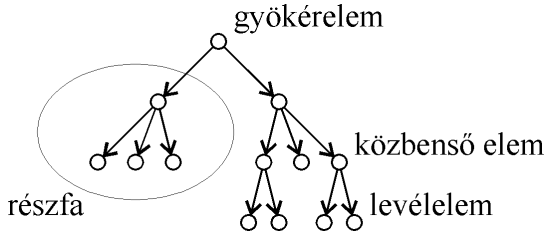
A *levélelemek* a fa azon elemei, amelyeknek nincs rákövetkezőjük. Bármennyi lehet belőlük. Úgy érhetőek el, hogy a gyökérelemből indulva veszem a gyökérelem rákövetkezőjét, majd annak a rákövetkezőjét stb.

A fa *közbenső elemei* a fa nem gyökér- ill. levélelemei, hanem az összes többi eleme.

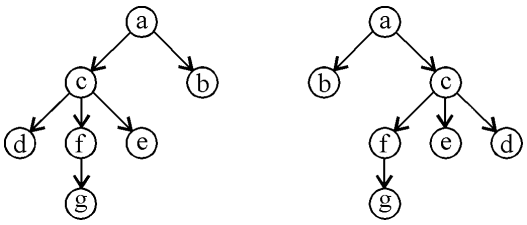
Az *út* a gyökérelemtől kiinduló, különböző szinteken átmenő, és levélelemben véget érő egymáshoz kapcsolódó élsorozat (lista). Az *út hosszán* az adott útban szereplő élek számát értjük. Minden levélelem a gyökértől pontosan egy úton érhető el, út helyett szokás beszélni a fa ágairól is. Egy fában az utak száma megegyezik a levélelemek számával.

A fán belül vannak *szintek*, egy elem szintje megegyezik az elem gyökérelemtől vett távolságával. A 0. szinten a gyökérelem helyezkedik el, az első szinten a gyökérelem rákövetkezői (a gyökértől egy élnyi távolságra lévő elemek), a második szinten ezeknek a rákövetkezői, és így tovább. A maximális szintszámot a *fa magasságának* vagy mélységének nevezzük (a lenti példában 3).

Minden közbenső elem egy *részfa* gyökereként tekinthető, így a fa részfákra bontható. Beszélhetünk *üresfáról* is, ez a fáknak az a szélsőséges esete, amikor a fának egyetlen eleme sincs.

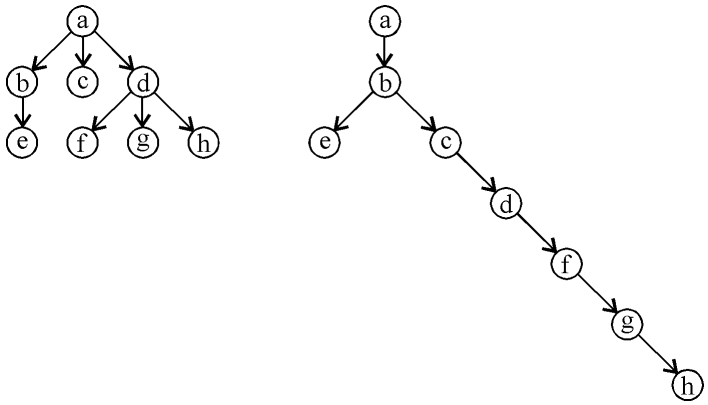


A fát tekinthetjük rendezett vagy rendezetlen módon. Ha rendezetlen módon vizsgáljuk, akkor az egy elemből kiinduló ágaknak nem tulajdonítunk rendezettséget, felcserélhetőek, azaz a sorrendjük tetszőleges. Ha egy fa rendezett, akkor az egy elemből kiinduló éleknek a sorrendje kötött. Tekintsük ezt a két fát:



Ha rendezetlen módon vizsgáljuk, akkor a két fa megegyezik, ha viszont feltesszük, hogy rendezettek, akkor a két fa nem ekvivalens.

Bináris fa: A számítástechnikában kitüntetett szerepe van a bináris fának. A bináris fa olyan fa, amelyben minden elemnek legfeljebb két rákövetkezője lehet. Szigorú értelemben vett bináris fáról akkor beszélünk, amikor minden elemnek 0 vagy pontosan 2 rákövetkező eleme van. Rendezett bináris fáknál a rendezettség miatt az egyértelműség kedvéért beszélhetünk baloldali és jobboldali részfákról. Tetszőleges nem bináris fa reprezentálható bináris fa segítségével a következő módon:



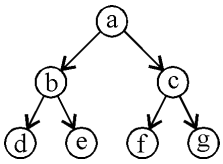
A binárisan ábrázolandó fa gyökéreleme a bináris fában is gyökérelem lesz. Ezek után a bináris fa baloldali részfájának gyökéreleme legyen a következő szinten lévő legbaloldalibb elem. Ehhez láncoljuk hozzá az azonos szinten lévő, közös gyökerű elemeket egymás jobboldali részfáiként. Ezt a folyamatot ismételni kell az egész fára, minden szinten. A bináris fában az eredeti fa levélelemei nem feltétlenül maradnak levélelemek, viszont felismerhetőek arról, hogy nincs baloldali részfájuk. Bármely fa kezelhető bináris faként.

A továbbiakban bináris fákról beszélünk.

Műveletek

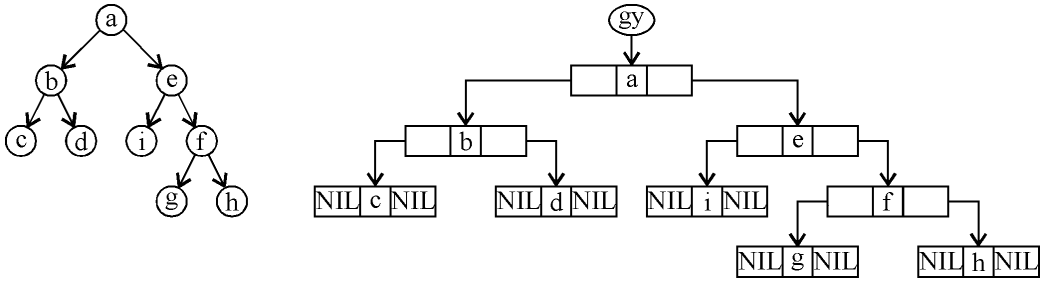
Létrehozás: Létrehozzuk az üres fát, majd egy elemet viszünk be (a gyökérelemet), ezután a létrehozás nem más, mint a fa bővítése.
Bővítés: Bővíteni általában levélelemmel szoktunk, ritkábban részfával. Általában csak a levélelemmel való bővítést értelmezzük, de tetszőleges helyen való bővítés is értelmezhető, a fa átstrukturálásával jár.
Törlés: Bináris fából bármikor törölhetek egy részfát, közbenső elemet általában nem. Ez fizikai törlés, logikai törlést bármikor végezhetek.
Csere: A megfelelő adatelemet bejárással elérem és az értékét bármikor fölülírhatom.
Rendezés, keresés, elérés: Nincs, legalábbis a korábbi értelemben. Elérés helyett egy speciális faművelet, a bejárás értelmezhető:
Bejárás: A bejárás az a tevékenység, amikor a fát, mint hierarchikus adatszerkezet egy sorra, lineáris adatszerkezetre képezzük le. A fa elemeit a bejárás folyamán egyszer, és pontosan egyszer érintjük, az elemek között valamilyen sorrendet állapítunk meg, attól függően, hogy hogyan járjuk be a fát. Három bejárási mód van, ezek közötti különbség abban rejlik, hogy a bejárás folyamán a gyökérelemet mikor dolgozzuk fel.

- *preorder* bejárás: Ha a fa üres, akkor vége a bejárásnak, egyébként vegyük a gyökérelemet és dolgozzuk fel (vagy képezzük le a sor első elemére). Ezután járjuk be preorder módon a baloldali, majd a jobboldali részfát. Pl. a lenti fánál: abdecfg.
- *inorder* bejárás: Ha a fa üres, akkor a bejárás befejeződik. Egyébként járjuk be inorder módon a baloldali részfát, majd dolgozzuk fel a gyökérelemet, és járjuk be ugyanezen módszerrel a jobboldali részfát. Pl. a lenti fánál: dbeafcg.
- *postorder* bejárás: Üres fánál vége, egyébként járjuk be postorder módon a baloldali, majd a jobboldali részfát, végül dolgozzuk fel a gyökérelemet. Pl. a lenti fánál: debfgca.



Feldolgozás: Alapja a bejárás.

A bináris fáknál a leggyakoribb ábrázolás a szétszórt ábrázolás, majdnem kizárólagosan ezt alkalmazzák. Egy fa eleme egy adatrészből és két mutatórészből áll, ezek egyike a bal, a másik a jobb oldali részfát címzi. A gyökérmutató fej jellegű, egy fán kívüli információ. Ha a fej NIL, akkor a fa üres.



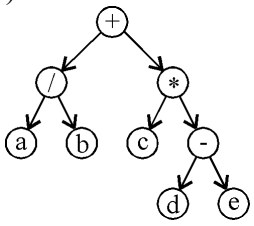
Alkalmazható a folytonos tárolás is, leginkább akkor, ha a fa statikus. Ilyenkor a fát egy bejárással leképezzük egy sorra. Három darab egydimenziós tömbre lesz szükség: Adat, B (a bal oldali részfák indexei, hol van a bal oldali részfa gyökere), J (a jobb oldali részfák indexei, hol van a jobb oldali részfa gyökere). Az Adat vektorban valamilyen módon felsorolom a fa elemeit.

	Adat	B	J
1	a	②	⑤
2	b	3	4
3	c	0	0
4	d	0	0
5	e	6	7
6	i	0	0
7	f	8	9
8	g	0	0
9	h	0	0

Ha valamely i-re B[i] és J[i] is nulla, akkor az Adat[i] levélelem. A bővítést és a törlést nehezebbé teszi, a bejárást könnyebbbé.

Bármely kifejezéshez felírható a megfelelő bináris fa, a kifejezésfája. A kifejezés és a bináris fa között kölcsönösen egyértelmű a megfeleltetés: az operandusokat és a műveleti jeleket a fa egy-egy elemének tekintem (a zárójelekkel nem foglalkozom). A kérdés az, hogy az operandusokhoz képest a műveleti jelek hol helyezkednek el.

Pl. ha a kifejezés az a / b + c * (d - e):



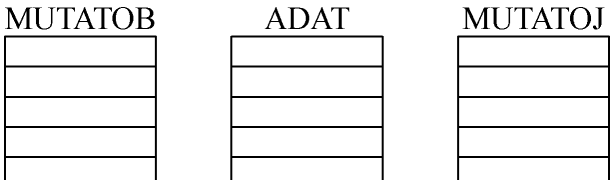
A fát különböző módon bejárva három kifejezést kapunk:

- preorder bejárással (*prefix alak*): + / a b * c – d e,
- inorder bejárással (*infix alak*): a / b + c * d - e,

– postorder bejárással (*postfix alak*): a b / c d e - * +.

A három alak közti különbség az, hogy a műveleti jel az operandushoz képest hol helyezkedik el. Az imperatív nyelvek a középső alakot használják, amely nem egyértelmű, de teljes zárójelezéssel vagy precedenciaszabály alkalmazásával egyértelművé tehető. (Egy infix kifejezés teljesen zárójelezett, ha minden operátor és a hozzá tartozó operandusok zárójelben szerepelnek.) A prefix és a postfix alak egyértelmű. A postfix alakot szokás fordított lengyel formának nevezni, általában ezt favorizálják az egyértelműség miatt. Amikor a fordító felismer egy kifejezést, lefordítja, akkor a kifejezésből épít egy bináris fát, majd bejárja és egy postfix alakot készít belőle.

Például van egy teljesen zárójelezett kifejezésem: ((a / b) + (c * (d - e))). Az operandusok változók, a műveleti jelek és a nevek egykarakteresek, és tegyük fel, hogy a kifejezés szabályos és teljesen zárójelezett. A feladat az, hogy ezen kifejezéshez készítsünk el a neki megfelelő bináris fát. A fa elemeit egydimenziós tömbbel kell szimulálni:



Mindegyik elem áll egy adatrészből és két mutatórészből. A szabad helyek induláskor fel vannak fűzve MUTATOJ szerint. Az algoritmus használni fog egy vermet (*V*).

Bináris fa készítése:

```
typedef struct veremelem {
  int mutato;
  int merre;          /* bal = 1 jobb = 2 */
} veremelem;

int kov_szabad_hely()
{
  int i;

  for ( i = 1; i <= m && ADAT[i] != ' '; i++);
  if (i > m) return 0;
  else return i;
}

int felepit()
{
  char c, i = 0, j, irany, gyoker;
  veremelem VEREM[n + 1];
  int p = 0;
  veremelem ve;

  j = kov_szabad_hely();
  gyoker = j;
  irany = 1;
  i++;
  while (i < m) {
    c = KIF[i];
    if (c == '(') {
      ve.mutato = j;
      ve.merre = irany;
      p++;
      VEREM[p] = ve;
      j = kov_szabad_hely();
      if (irany == 1) MUTATOB[VEREM[p].mutato] = j;
```

```
            else MUTATOJ[VEREM[p].mutato] = j;
            irany = 1;
         }
         else if (c == ')') {
            ve = VEREM[p];
            j = ve.mutato;
            if (ve.merre == 1) irany = 2;
            else p--;
         }
         else {
            ADAT[j] = c;
            irany = 2;
         }
         i++;
      }
      return gyoker;
   }
```

Az első operandust elhelyezi egy tömbben, mint külön elemet, és ezen operandus címét berakja a verembe. Ezután olvassuk tovább a karaktereket, így a következő a / jel. Ezt is elhelyezi, majd megy tovább, addig, míg a bezáró zárójelhez nem ér. Ekkor a veremből kiemeli a felső három elemet, létrehozza a részfát és a gyökerének címét visszahelyezi a verembe.

Ugyanezen fa preorder bejárása, egy nem rekurzív algoritmussal. Használunk egy vermet, amelyet egydimenziós tömbbel implementálok. Ha a verem üres, bejártam a fát. A verem azért kell, hogy mikor a baloldali részfára megyek, tudjam hova kell visszatérnem. Az algoritmusnak akkor lesz vége, mikor a verem kiürül.

```
void bejar_preorder(int gyoker)
{
   veremelem VEREM[n + 1];
   int p = 0, f;
   veremelem ve;

   if (gyoker != 0) {
      f = gyoker;
      do {
         if (f != 0) {
            printf("%c\n",ADAT[f]);
            p++;
            ve.mutato = f;
            ve.merre = 1;
            VEREM[p] = ve;
            f = MUTATOB[f];
         }
         else {
            ve = VEREM[p];
            f = ve.mutato;
            if (ve.merre == 2) p--;
            else VEREM[p].merre = 2;
            f = MUTATOJ[f];
         }
      } while (p != 0);
   }
}
```

A fa inorder bejárása:

```
void bejar_inorder(int gyoker)
{
   veremelem VEREM[n + 1];
   int f, p = 0;
   veremelem ve;

   if (gyoker != 0) {
      f = gyoker;
      do {
         if (f != 0) {
            p++;
            ve.mutato = f;
            ve.merre = 1;
            VEREM[p] = ve;
            f = MUTATOB[f];
         }
         else {
            ve = VEREM[p];
            f = ve.mutato;
            if (ve.merre == 1) {
               printf("%c\n",ADAT[f]);
               VEREM[p].merre = 2;
            }
            else p--;
            f = MUTATOJ[f];
         }
      } while (p!=0);
   }
}
```

A fa postorder bejárásánál fontos, hogy tudjuk, hányszor léptünk vissza. Az első visszalépésnél még nem kell a gyökérelemet feldolgozni, csak a másodiknál.

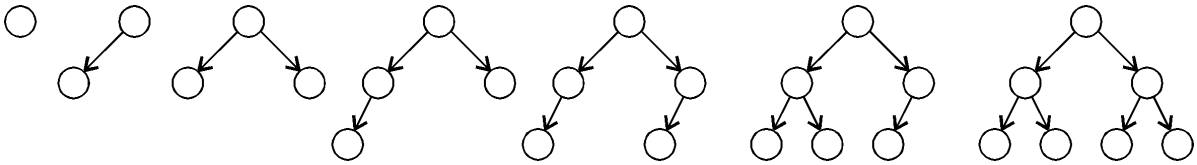
```
typedef struct ref {
   int adat;
   struct ref *bal;
   struct ref *jobb;
} REF;

void preorder ( REF *t)
{
   if (t != NULL ) {
      feldolgoz( t);
      preorder ( t->bal);
      preorder ( t->jobb);
   }
}
```

Kihasználjuk a rekurzív hívás lehetőségét. Ez azt jelenti, hogy a vermet a rendszernek kell kezelnie, és nem mi oldjuk meg. Mi tehát kevesebbet dolgozunk, viszont egy rendszerszinten nagyon lassú algoritmust kapunk. Minden rekurzív algoritmus felírható ciklusok segítségével. A másik két bejárás hasonlóan fölírható.

Minimális magasságú fa: Legyen adva egy adott elemszám. Ez alapján fel lehet építeni egy olyan fát, amelynek a magassága az adott elemszám mellett a lehető legkisebb. Ezt megtehetjük úgy, hogy a legalsó szint kivételével minden szintre a lehető legtöbb elemet helyezzük el. Jelentősége az, hogy az utak minimálisak, bármely levélelem a legrövidebb úton érhető el.

Tökéletesen kiegyensúlyozott fa: Egy fa akkor tökéletesen kiegyensúlyozott, ha minden elem bal- illetve jobboldali részfájában elhelyezett elemek száma legfeljebb eggyel tér el. Mindig minimális magasságú.

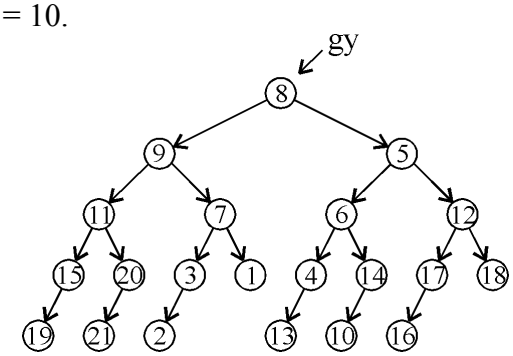
A kérdés az, hogy hogyan lehet létrehozni adott elemszám (n) mellett egy tökéletesen kiegyensúlyozott fát. Ez a következőképpen történik: az elemeket az érkezés sorrendjében vesszük, az első elem a gyökér lesz. A maradék elemből előállítjuk a gyökér $n_b = (n \% 2)$ elemszámú bal oldali részfáját, majd ugyígy előállítjuk a gyökér $n_j = (n - n_b - 1)$ elemszámú jobb oldali részfáját.



Az létrehozás implementációja:

```
REF *fa (int n)
{
   REF *ujelem;
   int x, nb, nj;

   if (n == 0) return NULL;
   else
   {
      nb = n / 2;
      nj = n – nb - 1;
      scanf(„%d”, &x);
      ujelem = (REF*) malloc (sizeof (REF));
      ujelem->adat = x;
      ujelem->bal = fa(nb);
      ujelem->jobb = fa(nj);
      return ujelem;
   }
}
```

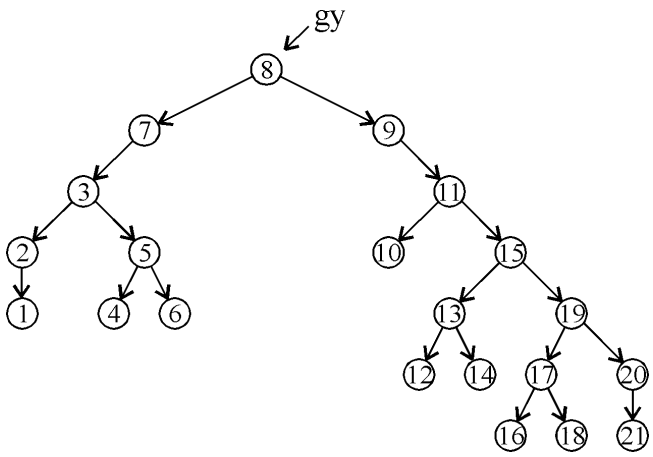
Pl. Ha a fa elemei: 8, 9, 11, 15, 19, 20, 21, 7, 3, 2, 1, 5, 6, 4, 13, 14, 10, 12, 17, 16, 18.
Ekkor $n = 21$, $n_b = 10$, $n_j = 10$.



A megadott számokból egy kiegyensúlyozott fát hoz létre a függvény.

A bináris fákat olyan adathalmaz feldolgozására használják gyakran, amikor az adatelemeknek van egy kulcsrészük (táblázatok) vagy maguk az adatelemek különböző értékűek. Ilyenkor a kulcs a feldolgozás alapja, a rendezett kulcs alapján kell keresni.

Ha adott elemszám mellett a fát úgy építem fel, hogy bármely elemére igaz, hogy az elem baloldali részfájában az összes eleme kulcsa kisebb, a jobboldali részfájában az összes eleme kulcsa pedig nagyobb az adott elem kulcsánál, akkor **keresőfáról** (vagy rendezőfáról) beszélünk.



A keresőfa jelentősége abból áll, hogy ha van egy adott elemünk, amelyet meg akarunk keresni a fában, akkor a gyökértől kiindulva bármelyik kulcsú elem megkereshető úgy, hogy vizsgáljuk, hogy a gyökérelem kulcsa, megegyezik-e a keresett elem kulcsával. Ha igen, megállunk, ha nem megnézzük, hogy attól kisebb vagy nagyobb. Ha nagyobb, akkor a jobboldali részfán (ha van), ha kisebb, akkor a baloldali részfán (ha van) haladok tovább ugyanezzel a módszerrel, mindaddig, míg az elemet meg nem találom, vagy nem jutok el egy olyan elemhez, amelyiknél az adott irányban nincs több elem, ekkor a keresett kulcsú elem nem szerepel a fán. A keresés gyors, mert maximum a fa magassága + 1 hasonlítással vagy megtalálom az elemet, vagy az elem nincs a fában. A keresőfában az elemek olyan sorrendben vannak, hogy ha inorder módon járom be a fát, akkor az elemeknek egy rendezet sorozatát kapjuk.

Egy ilyen fa létrehozása: Vegyük az elemeket az adott sorrendben. Az első elem lesz a fa gyökere. A második elemet véve keressük meg, hogy szerepel-e már a fában. Ha igen, akkor hiba történt (kétszer nem lehet benne ugyanaz az elem), ha nem, akkor döntsünk, hogy az hol helyezkedik el az előzőhöz képest (kisebb vagy nagyobb a gyökérelemnél). Ha nagyobb, akkor a jobboldali részfának lesz az eleme, ha pedig kisebb, akkor a baloldalinak. Beillesztjük az új elemet a részfába: addig megyünk, amíg levélelemet nem találunk, mert mindig levélelemmel bővítünk.

Alkalmazása: Adva van szavaknak egy sorozata, tetszőleges számú szó egymás után. Határozzuk meg, hogy ebben hány szó van, és az egyes szavak milyen gyakorisággal fordulnak elő! Ennek megoldása során a kulcs maga a szó lesz, az elemek másik értéke pedig a szavak gyakorisága. Ezekből keresőfát építek fel, megkeresem az új elem helyét, majd bővítek. Ez a program alkalmazható olyan esetekben, amikor meg akarjuk tudni, hogy a program szövegében lévő szavakat milyen gyakorisággal használjuk. A fordítóprogramok is valahogy ilyenféleképpen kezelik a problémát.

```
typedef struct ref {
   char kulcs[20];
   int szamlalo;
   struct ref *bal;
   struct ref *jobb;
} REF;

REF *gyoker;
char k[20];

void kiir (REF *w , int l)
{
   int i;

   if (w != NULL) {
      kiir(w->bal, l + 1);
      for ( i = 0; i < l; i++) printf(„”);
```

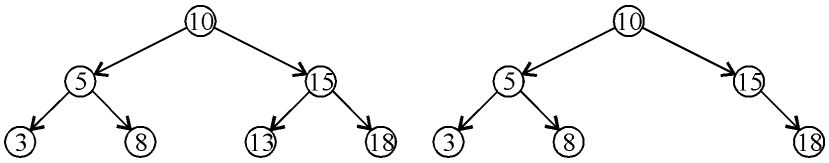
```
      printf(„%s\n”, w->kulcs);
      kiir(w->jobb, l + 1);
      }
   }

void keres ( char x[20], REF *p )
{
   if (p == NULL) {
      p = (REF*) malloc (sizeof (REF));
      p->kulcs = x;
      p->szamlalo = 1;
      p->bal = p->jobb = NULL;
   }
   else if (x < p->kulcs) keres(x, p->bal);
   else if (x> p->kulcs) keres (x, p->jobb);
   else (p->szamlalo)++;
}

main(){
   gyoker = NULL;
   scanf( „%s”, &k);
   keres(k,gyoker);
   kiir(gyoker,0);
}
```

A keresőfából fizikailag bármely elem törölhető. Hogyan tudjuk megvalósítani? A keresés eredménye szerint két eset lehet:

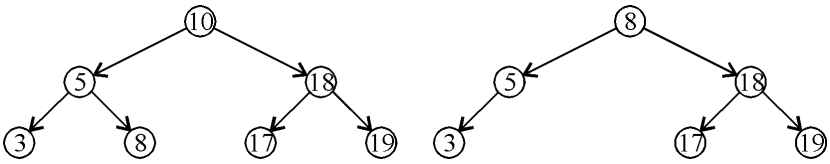
I. A törlendő elem nincs a fában. Ez az eset a legegyszerűbb, mert nincs dolgunk.

II. Az adott kulcsú elem benne van a fában. Ekkor három alapeset lehetséges:

1. A törlendő elemnek nincs rákövetkezője (levélelem): Nincs gond, simán megkeressük és töröljük. Pl. a 22 törlése hiba (mert nincs a fában), a 13 törlése:



2. A törlendő elemnek egy rákövetkezője van: Az elemet törlöm és a baloldali vagy jobboldali részfát (amelyik van) felcsúsztatom a törölt elem helyére (átállítom a mutatóját).
3. A törlendő elemnek két rákövetkezője van (van bal- és jobboldali részfája): Ebben az esetben fölülírom a törlendő elemet: vagy a baloldali részfa legjobboldalibb elemét, vagy a jobboldali részfa legbaloldalibb elemét írom a helyére. Ez is mutatóátállítást jelent. Pl. a 10 törlése.




```
REF *q;

void tor (REF *r)
{
   if (r->jobb != NULL) tor(r->jobb);
   else {
      q->kulcs = r->kulcs;
```

```
      q->szamlalo = r->szamlalo;
      q = r;
      r = r->bal;
   }
}

void torol (char x[20], REF *p)
{
   if (p == NULL) printf(„A szo nincs a faban!”);
   else {
      if (x < p->kulcs) torol( x, p->bal);
      else if (x > p->kulcs) torol( x, p->jobb);
      else {
         q = p;
         if (q->jobb == NULL) p = q->bal;
         else if (q->bal == NULL) p = q->jobb;
         else tor(q->bal);
      }
   }
}
```

A keresőfák jelentősége, hogy nagyon gyors keresést tesznek lehetővé. Ez igazán nagy adathalmaz esetén lényeges. Ha a keresőfa tökéletesen kiegyensúlyozott lenne (ez lenne az optimális eset, a minimális magasságú fa), akkor lenne a leggyorsabb a keresés (a keresés maximum annyi lépésig tart, amennyi a fa magassága), de a módosítások miatt nehéz lenne kezelni, mert mindig át kellene szervezni, hogy tökéletesen kiegyensúlyozott legyen. Nem éri meg tökéletesen kiegyensúlyozni őket, ezért **kiegyensúlyozott fákat** alkalmaznak. (**AVL-fa**). Egy fa kiegyensúlyozott, ha bármely elem esetén a bal és jobb oldali részfák magasságának különbsége legfeljebb eggyel tér el. Kezelése egyszerűbb, mint a tökéletesen kiegyensúlyozott fáké. Ezután kiegyensúlyozott fákkal foglalkozunk.

A kiegyensúlyozott fa magassága elemszámtól függetlenül legfeljebb 45%-kal nagyobb, mint egy olyan tökéletesen kiegyensúlyozott fa magassága, amelyik ugyanannyi elemből épül fel, ezért a kiegyensúlyozott keresőfák játszanak fontos szerepet a gyakorlatban. A keresés hatékonyságát maximum 50%-kal rontja, de a fa kezelését könnyebbé teszi. Bővítésük és a törlés egyszerűen megoldható.

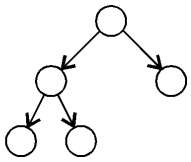
Tökéletesen kiegyensúlyozott fát csak fix elemszámnál érdemes használni, mert sokszor kell benne keresni. Ha az elemek száma dinamikusan változik, akkor kiegyensúlyozott keresőfa marad.

Műveletek

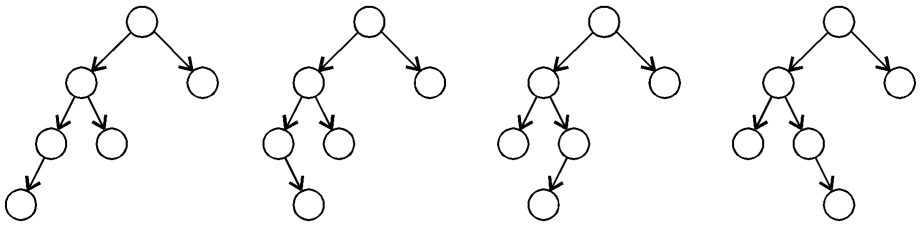
Bővítés: A Keres eljárást használva megkeressük, hogy hova szúrjam be az új elemet. Levélelemmel bővítünk. Tegyük fel, hogy a beszúrás után a baloldali részfa magassága nőtt meg. Ekkor három lehetőség van:

- Balra bővítünk, amikor a jobboldali részfa magassága eggyel nagyobb volt ($m_{bal} < m_{jobb}$ a beszúrás előtt). A fa kiegyensúlyozódik.
- Úgy szúrunk be, hogy korábban tökéletesen ki volt egyensúlyozva a fa ($m_{bal} = m_{jobb}$ a beszúrás előtt). A fa még a beszúrás után is kiegyensúlyozott.
- Balra bővítünk, de a baloldali részfa volt a magasabb ($m_{jobb} < m_{bal}$ a beszúrás előtt), a kiegyensúlyozottság, így ki kell egyensúlyozni a fát mutatócserével. Egy-egy ilyen kiegyensúlyozási lépés két vagy három elemet érint, ezeket forgatni kell. Át kell szervezni a fát, a túl hosszú ágat helyre kell billenteni. Alesetek:

A fa bővítés előtt:



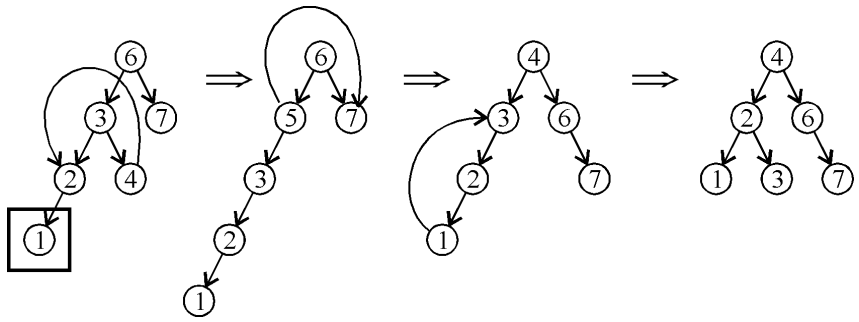
A bővítés alesetei:



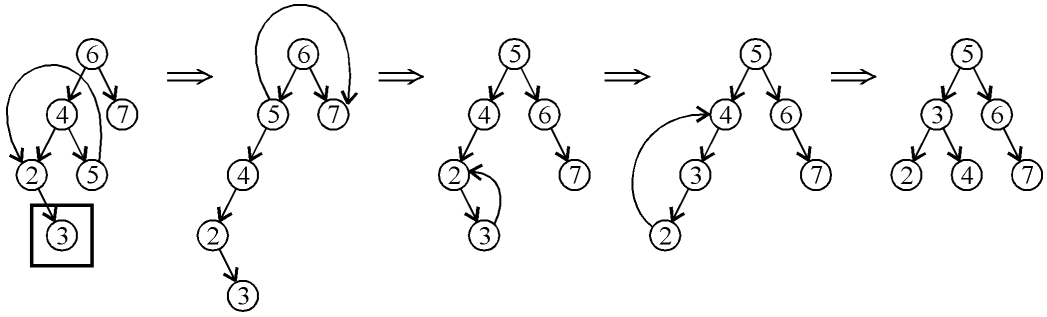
1. 2. 3. 4.

Példák az alesetekre:

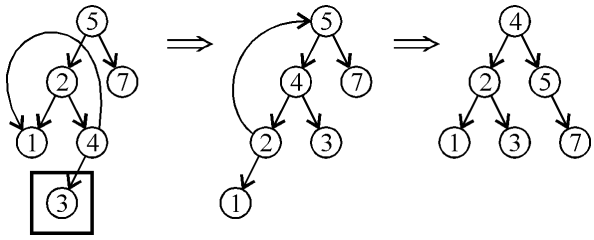
I. 1-el bővítek



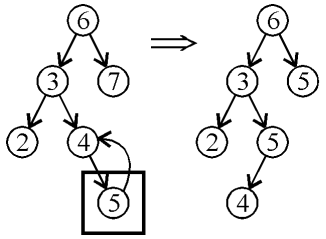
II. 3-mal bővítek



III. 3-mal bővítek

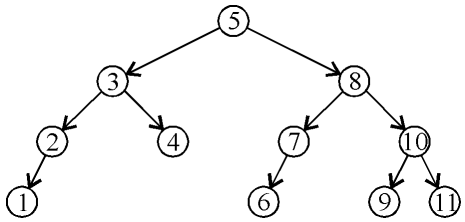


IV. 5-tel bővítek.

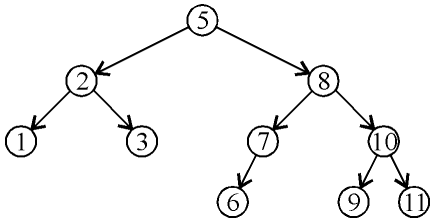


Ez visszavezethető az előző esetre.

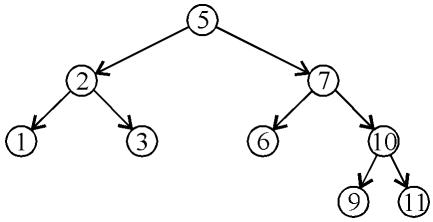
Törlés: A kiegyensúlyozott fában a törléshez a kiegyensúlyozott fában való törlést ki kell egészíteni, és kiegyensúlyozást kell végezni. Legyen a példa a következő:



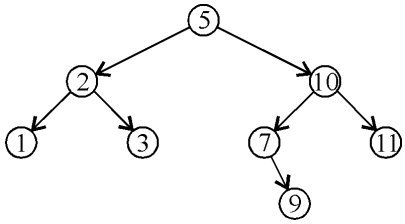
I. Töröljük a 4-es elemet! Az egyensúly felborul, ezért be kell állítani. Az eredmény:



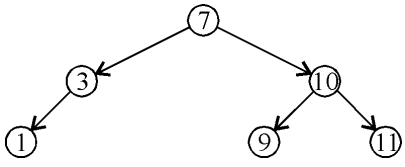
II. Töröljük a 8 elemet az előző fából!



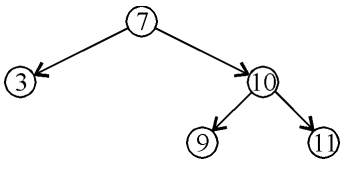
III. Töröljük a 6 elemet az előző fából!



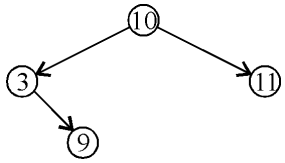
IV. Töröljük az 5-t!



V. Töröljük az 1-et!

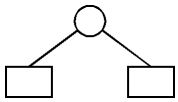


VI. Töröljük a 7-et!



Piros-fekete fa: Speciális keresőfa, minden elemének van egy színe (piros vagy fekete). Ezeknek a színeknek a kiosztásának szabályával biztosítható, hogy a fában minden, a gyökértől levélelemig vezető út maximum kétszer olyan hosszú, mint a legrövidebb út hossza. Tehát az ilyen fák megközelítőleg kiegyensúlyozottak. A piros-fekete fa elemeinek deklarációja, hasonló a bináris keresőfa elemeinek deklarációjához, azaz minden pont tartalmaz egy kulcs, bal és jobb mezőket, de a bináris keresőfáktól eltérően egy szín mezővel bővül a deklaráció.

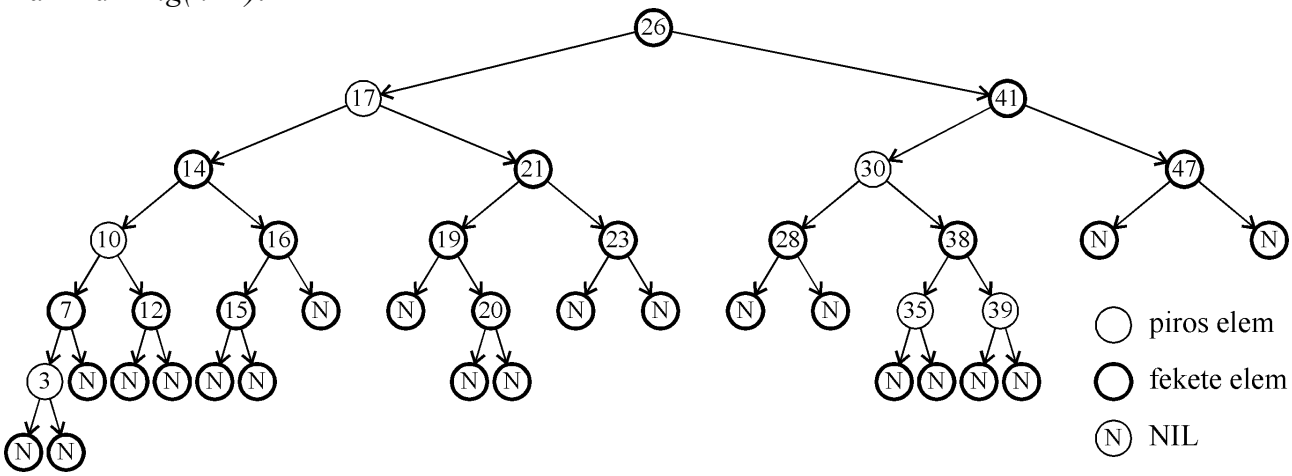
Levélelemei speciálisak, az eddigi levélelemek NIL mutatója egy speciális NIL csomópontra mutat. Ezek a fa külső pontjai, a többiek a belsők.



Egy keresőfa akkor lesz piros-fekete, ha a következő tulajdonságok teljesülnek:

1. A fa minden elemének színe piros vagy fekete.
2. Minden NIL csúcs színe fekete.
3. Minden piros színű elemnek mindkét rákövetkezője fekete.
4. Bármely két, azonos elemből induló és levélig vezető úton ugyanannyi fekete elem van.

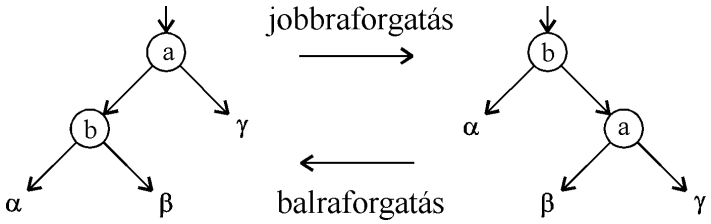
Ezt a négy tulajdonságot teljesítő piros-fekete fánál, amely *n* adatelemet tartalmaz, a magasság maximum *2lg(n+1)*.



Inorder módon bejárva a fát az elemeinek rendezett sorozatát kapom.

A piros fekete fákon ugyanazokat a műveleteket tudjuk elvégezni, mint a bináris keresőfákon, csak bizonyos műveletek esetében szükség van a fa átszínezésére és átszervezésére, pl. törölni és bővíteni ugyanúgy kell, de a fának piros-fekete fának kell maradnia. Ahhoz, hogy a piros-fekete tulajdonságok teljesüljenek, a bővítés illetve a törlés után a csomópontok átszínezésével és forgatásával tudunk eljutni.

A forgatás mutatócserékkel reprezentálható. Módosítja a fa szerkezetét, de megőrzi a keresőfa tulajdonságot.

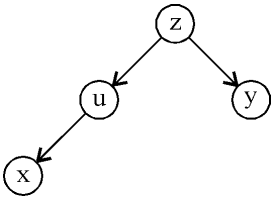


A *b* elem balra forgatása esetén feltételezzük, hogy a *b* jobb oldali fia nem NIL. Ekkor a forgatás a *b* elem és jobboldali gyermeke (*a*) körül történik. A forgatás után az *a* lesz a részfa új gyökere, a *b* az *a* bal fia, és az *a* bal fia a *b* jobb fia. A jobbra forgatás hasonlóan történik, az *a* és a *b* csúcsok, valamint a jobb és a bal oldalak felcserélésével.

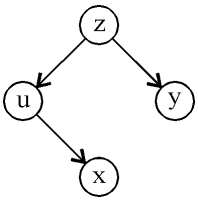
Bővítés: Levélelemmel bővítünk, ugyanúgy, mint a keresőfánál: megkeresem a bővítő elem helyét és berakom. Az új elem színét állítsuk pirosra, és tartozzon két NIL elem hozzá. A 3. tulajdonság sérülhet, ha a bevitt elem megelőzője is piros. Tegyük fel, hogy a gyökér színe fekete!

Az *x* elemet szúrom be, az *x* megelőzője *u*. Az *u* és az *y* azonos szinten vannak, megelőzőjük *z*. Az *x* piros, és megelőzője, *u* is. A fa bővítésénél hat alapeset van, ebből 3-3 szimmetrikus (három esetben *x* szülője jobb, a másik három esetben bal oldali gyermeke az *x* nagyszülőjének). A nagyszülő mindig létezik, mert feltevésünk szerint a gyökér színe fekete, *x* szülője pedig piros.

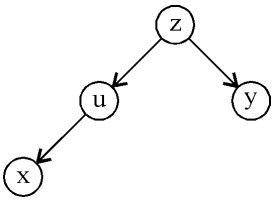
1. eset. Az *u* és az *y* piros, *z* fekete. Ekkor az *u* és az *y* legyen fekete, *z* pedig legyen piros, majd egy szinttel feljebb kell tolni a problémát. (a, b ábra)

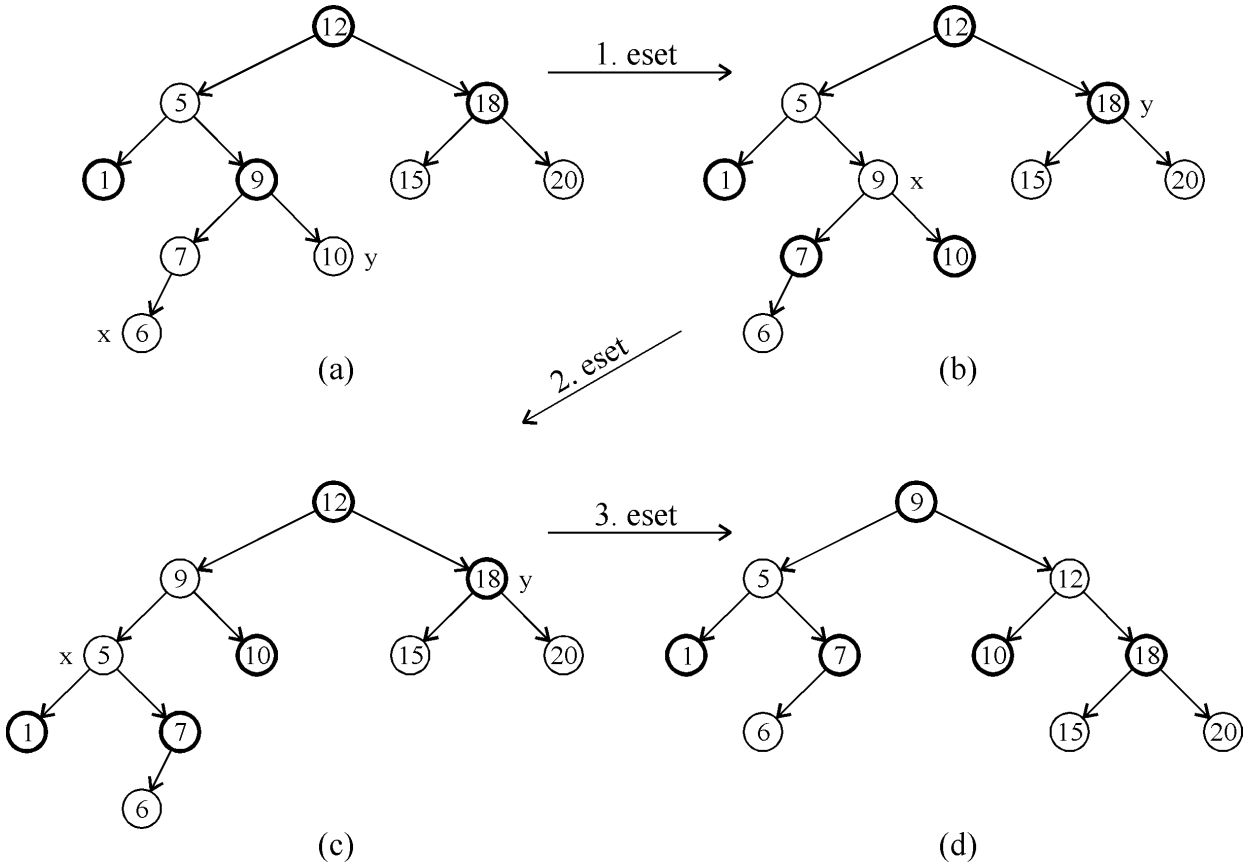


2. eset. Az *y* fekete, az *u* piros, az *x* jobb oldali rákövetkezője *u*-nak. Ekkor hajtsunk végre egy balra forgatást az *u* körül, ezzel előáll a 3. eset. (b, c ábra)



3. eset. Az *y* fekete, az *u* piros, az *x* bal oldali rákövetkezője *u*-nak. Ekkor az *u*, *x* szülője legyen fekete, *z*, az *x* nagyszülője piros, és hajtsunk végre egy jobbra forgatást *u* körül. (c, d ábra)





Az 1. esetben lehet, hogy a gyökér piros lesz. Minden algoritmust úgy zárunk, hogy a gyökér legyen fekete.

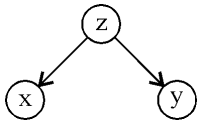
Törlés: Törléskor megkeresem az elemet, és végrehajtom a törlést, ugyanúgy, mint a bináris fáknál. Ha a törlendő elem színe piros, akkor nem sérül a piros-fekete tulajdonság. Viszont ha fekete, akkor a 3. és a 4. tulajdonság is megsérülhet.

Ha a törlendő csúcs színe fekete, akkor minden olyan út, amely átmegy a törlendő ponton, eggyel kevesebb fekete pontot fog tartalmazni. Tehát minden olyan pontra, amely őse a törlendő pontnak, nem teljesül a 4. tulajdonság. Ezt a problémát kiküszöbölhetjük, ha úgy tekintjük, hogy az *x* pont egy extra fekete értéket tartalmaz. Vagyis minden olyan út fekete pontjainak számához hozzáad egyet, mely átmegy az *x* ponton, ekkor teljesülne a 4. tulajdonság. Mikor eltávolítjuk a fekete *y* pontot, akkor a fekete értékeket továbbadjuk az *x* fiának. Az egyetlen probléma akkor van, ha *x* már eredetileg is fekete volt, így kétszeresen fekete lesz, ami sérti a 2. tulajdonságot.

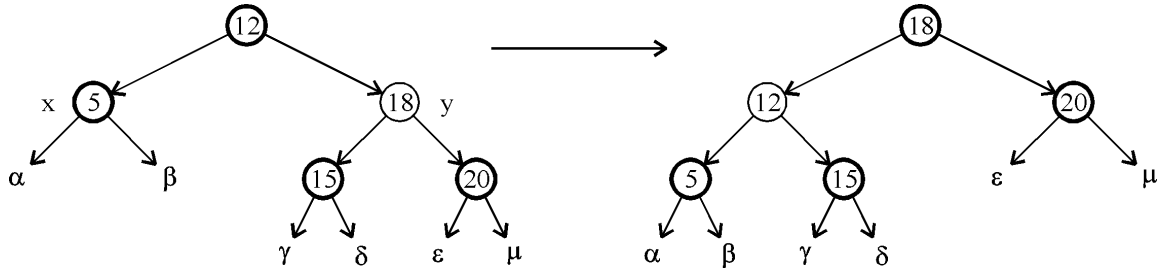
Ha fekete elemet törlök, a törölt elem rákövetkezőjére öröklődik a színe. Ha ez a szín a piros, akkor rendben van, ha fekete, akkor ezt mozgatom felfelé a fában. Ha eljutok a gyökérelemig, akkor a gyökeret ki kell mozgatnom a fából, amíg

1. *x* olyan pontra mutat, amely piros színű, ekkor az utolsó sorban feketére színezzük, vagy
2. *x* a gyökérre mutat, és ekkor az extra fekete egyszerűen elhagyható, vagy
3. alkalmas forgatásokat és átszínezéseket hajtunk végre.

A törlésnek nyolc alapesete van, ebből négy-négy szimmetrikus. A törlendő elem a *z*, a rákövetkezői pedig *x* és *y*.

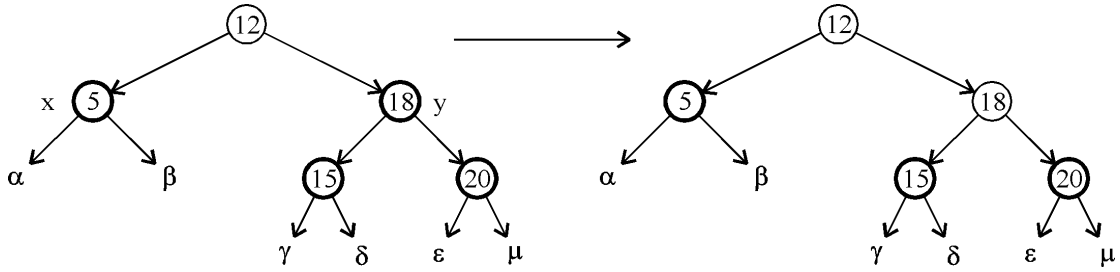


1. eset. A törlendő pont (*y*) piros színű, ekkor létezik neki fekete gyermeke. Felcserélve *y* és *z* pontok színét, majd egy balra forgatást végrehajtva az *z* pontra, a piros-fekete tulajdonság változatlanul teljesülni fog. Az *x* pont új testvére ekkor fekete lesz, így az 1. esetet a 2., 3. vagy 4. esetre transzformáltuk.

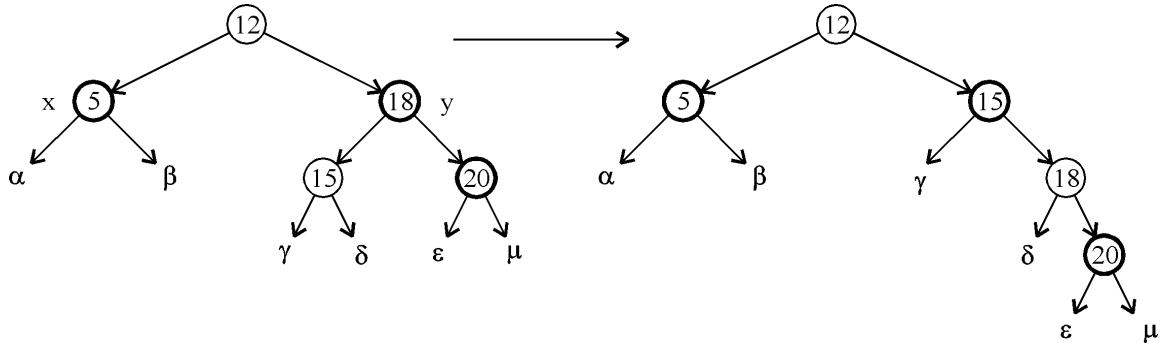


A 2., 3. és 4. esetekben az *y* színe fekete. A további eseteket *y* gyermekeinek színei alapján különböztetjük meg.

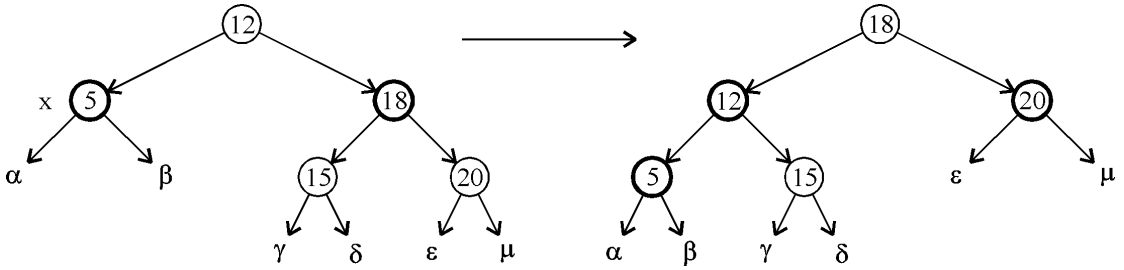
2. eset. Az *y* mindkét gyermeke fekete színű. Mivel *y* szintén fekete, elvetünk egy feketét *x*-től és *y*-tól, így *x* egyszeresen fekete, *y* pedig piros lesz, és *x* új szülőjének pedig átadunk egy extra feketét.



3. eset. Az *y* fekete, bal gyermeke piros és jobb gyermeke fekete. Felcserélve *y* és bal gyermekének színét, majd egy jobbra forgatást végrehajtva az *y* pontra, a piros-fekete tulajdonság változatlanul teljesülni fog. Az *x* pont új *y* testvére fekete és *y*-nak a jobb gyermeke piros lesz, így a 3. esetet a 4. esetre transzformáltuk.

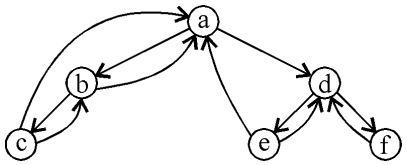


4. eset. Az *y* fekete és jobb fia piros, a bal oldali mindegy, milyen. Az *y*-nak, *x* megelőzőjének (*z*) és *y* jobboldali gyermekének színét megváltoztatva (ellenkezőjére állítva), majd *x* megelőzője körül balra forgatást végrehajtva, eltöröljük az *x*-ről az extra feketét, anélkül, hogy megsértenénk a piros-fekete tulajdonságot. Ezután az *x* felveszi a gyökér értékét.

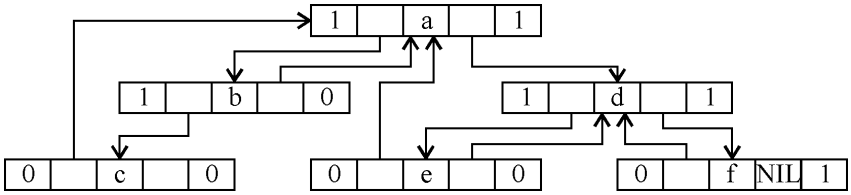


A gyökeret a törlés végén mindig feketére kell változtatni. Piros-fekete fákban a keresés gyors.

A bináris fák bejárása alapvetően rekurzív algoritmusokkal oldható meg, de ezek memória- és időigényesek. A bejárás gyorsítása vezetett ahhoz az ötlethez, hogy bináris fákban, szétszórt ábrázolást feltételezve, vannak olyan elemek (a levélelemek), amelyeknek van 2 NIL mutatójuk. Célunk az, hogy használjuk ezeket a mutatókat arra, hogy visszamutassanak a gyökérre. Ezt nevezzük a vissza- ill. **felfűzött fák**, vagy a kitaposott út módszerének.



Az inorder bejárás szerinti első elem baloldali mutatója mutasson a gyökérelemre. Az inorder bejárás szerinti utolsó elem jobboldali elem mutatója maradjon NIL. Az összes többi esetben állítsuk be a mutatókat úgy, hogy a baloldali mutató mutasson az adott elemet inorder módon megelőző elemre, a jobboldali pedig az adott elemet inorder módon követő elemre. Ezzel felfűztük a fát, így a fában kétfajta mutató lesz: a tényleges (részfára mutató) mutató és visszafűző mutató. Hogy a két mutatót megkülönböztessük, hozzá kell rendelni egy-egy jelzőbitet. Például, ha a bit értéke 1, akkor részfára mutat, ha 0, akkor visszafűző mutató.



A felfűzött fa a preorder és az inorder bejárást segíti, mert ezen mutatók segítségével rekurzió nélkül be tudom járni a fát, viszont a postorder bejárást nem teszi hatékonyabbá. Ez általában statikus fáknál használható jól, mert különben minden változtatásnál a mutatókat is változtatni kellene, és kezelése nehézkessé válna.

A piros-fekete fák reprezentációjánál nagyon gyakran van egy plusz mutató: a megelőző elemre mutat, ezáltal gyorsítja a bejárást.

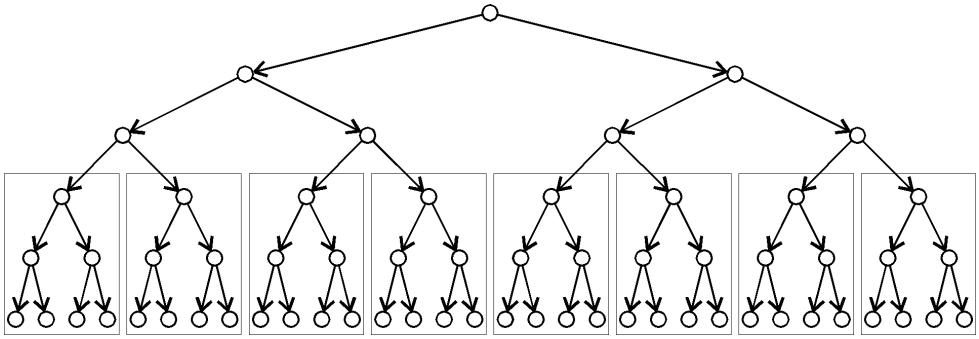
Többágú fák: Többágú rendezett fák esetén is lehet értelmezni a bejárást. Pl. preorder bejárás: Ha a fa üres, akkor vége. Egyébként feldolgozzuk a gyökeret, majd utána preorder módon a legbaloldalibb részfát járjuk be, majd az összes többi részfát rendezettség szerint. Hasonlóan lehet végrehajtani az inorder és a postorder bejárást is.

A reprezentációjára az egyik megoldás a bináris fa. Másik lehetséges módja az, hogy többágú fákat úgy ábrázolok, hogy annyi mutatója legyen minden egyes elemnek, ahány részfa indul ki az adott elemből. Az elemeknél vagy dinamikus számú mutatót kell alkalmazni, vagy fix mutatótömböt, de ekkor a mutatók száma maximált: annyinak kell lennie, hogy minden elem kezelhető legyen. A karbantartása könnyű, de sok helyet foglal. Nincsenek általános elvek ill.

algoritmusok, a megoldása problémafüggő.

Ezek a lemezen megjelenő fáknál játszanak alapvető szerepet, mert az állományok kezelésénél nagy probléma a mozgatás és az átviteli idő. A mutatók lemezcímeket jelentenek, és nem egy tárbeli címet. A cél az, hogy ne egy elemet mozgassunk, hanem adatcsoportokat. A többágú fa a lemezen van. Ha elemenként fogom meg, minden elemet egyenként ki kell vinnem, és ez lassú. A fát osszuk fel lapokra, egy lapon több elem legyen. Lapot mozgassunk a lemez és a gép között, lapot hozzunk be a tárba. A lap elérése periféria sebességű, az elem sebessége pedig társebességű, így lényegesen gyorsabb a művelet.

Tökéletesen kiegyensúlyozott bináris fa lapokra osztása:



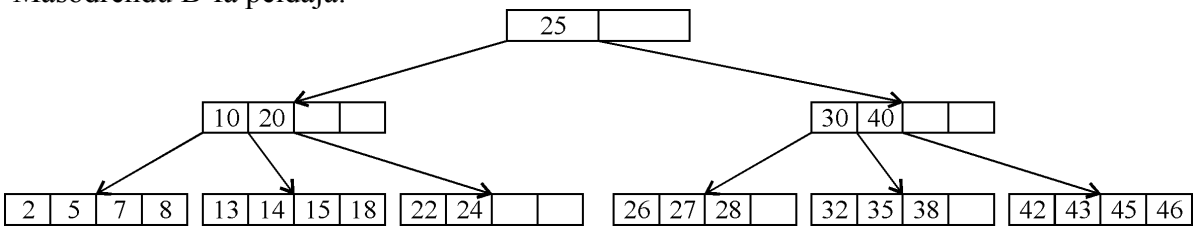
B-fák (balanced): Speciális keresőfák. A kiegyensúlyozott fák, a kereső fák és a lapra történő felosztás elvét viszik tovább. Viszonylag könnyű karbantartani, és gyors a keresés. Jelentőségük az állományoknál vannak.

Mindig adott egy *n* szám, amely a B-fa rendje. Úgy kell felépíteni a fát, hogy minden lapon legalább *n* és legfeljebb *2n* db elem legyen. Az aktuális elemszám egy olyan *m* szám, melyre teljesül, hogy $n \leq m \leq 2n$.

Ha összesen *k* db elemem van, legrosszabb esetben *n / log k* hivatkozással meg tudom keresni az adott elemet. Általában akkor használjuk, ha van kulcs az adatoknál.

A B-fa definíciója: A B-fa elemi lapok összessége. Minden lap maximum *2n* db elemet tartalmaz, és minden lap, kivéve a gyökérlapot, legalább *n* db elemet tartalmaz, a gyökérlap legalább egyet. A B-fában minden levéllap ugyanazon a szinten van. Egy lap vagy levéllap, vagy *m + 1* rákövetkezője van (*m* a lapon elhelyezett adatok száma).

Másodrendű B-fa példája:



A lapokon 2,3,4 elem (kulcs) helyezkedhet el. A fa háromszintű, a levélelemek azonos szinten vannak. Minden elemnek eggyel több rákövetkezője van, mint amennyi elhelyezkedik a lapon. A lapon az elemek kulcsuk szerint növekvő sorrendben helyezkednek el, rendezettek. Ha a fát összenyomnánk, akkor az elemeknek rendezett sorozatát kapnánk.

Egy nem levéllap esetén a lappon kulcsok és mutatók helyezkednek el, a kulcsok nagyság szerint rendezve és a mutatók NIL értékűek. Egy lap szerkezete:

p_0 k_1 p_1 k_2 ... k_n p_n

A fa szerkezete (kulcsok elhelyezkedése a fában): Adott fában az x kulcsot keressük, a gyökérelemből indulva. A kulcsokra vonatkozóan a lapon a kulcsokat végig kell keresnünk, sok

elemnél binárisan. Ha megtaláljuk, rendben van. Ha nem találjuk meg, akkor a lehetséges esetek a következők:

1. $k_i < x < k_{i+1}$: Az x a lapon elhelyezett két kulcs közé esik. Abban a részfában kell keresnünk, amelyre a p_i mutat.
2. $x < k_1$: A keresett kulcs kisebb a lapon lévő legkisebb kulcsnál. A p_0-al címzett, legbaloldalibb részfa gyökerét kell megnéznem.
3. $k_m < x$: A keresett kulcs nagyobb a legnagyobb kulcsnál. A p_n-nel címzett, legjobboldalibb lapon kell keresnem.

Elindulok a gyökérlapon és behatárolom, hogy hol kell keresnem az elemet. Addig keresek, amíg meg nem találom, vagy levéllaphoz nem jutok, és azon sem találom.

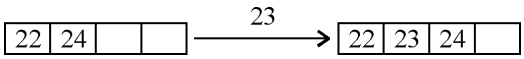
A B-fa szemléletesen úgy adja a kulcsok sorozatát, hogy összenyomjuk a szinteket. Ha a levélelemekről indulunk, be tudjuk rakni a részfa gyökeréből (eggyel magasabb szintről) az elemeket, amelyek beillenek az alattuk lévő lapok közötti résbe.

Műveletek

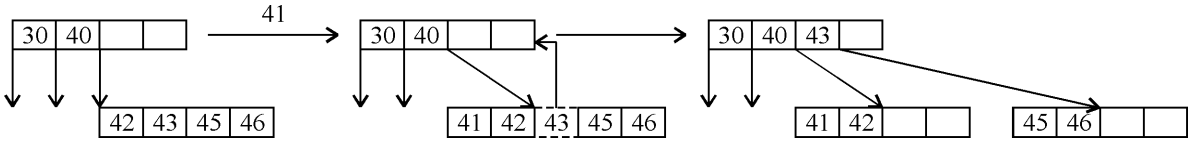
Létrehozás: Üres fa bővítése.

Bővítés: Mindig levéllapon történik. Megkeresem a bővítendő elem helyét.

1. Egyszerű esete az, amikor olyan lapra akarok bővíteni, ahol az elemek száma kisebb, mint a megengedett maximális érték ($m < 2n$). Pl. 23 kulcsú adatot akarom bevinni. Ha nem találom meg, akkor levéllapra kell beilleszteni. A helye a 22 és a 24 között van. Ezen a lapon az elemek száma kisebb, mint $2n$, így a 23 bevihető. Az előbbi keresési módon megkeresem a kulcs helyét.



2. Ha olyan lapra kellene bővíteni, amelyik tele van ($2n$ eleme van), akkor új lapot kell létrehozni. Ha az új elemet is felviszem a lapra, akkor $2n + 1$ elemem lesz. Ezen elemek közül a középső elemet fölviszem a részfa gyökérlapjára, majd ott a megfelelő helyre beillesztem. A maradék $2n$ első felét az első, második felét a második lapra viszem. Ez a vágás. Így részfa gyökérlapon bővítettünk. Ha a felvitt elem befért, akkor nincs probléma, ha nem fért be, akkor ezen a szinten is vágni kell ugyanezzel az algoritmussal. Előfordulhat, hogy a gyökérelemen nem tudunk bővíteni, ezért itt is vágni kell, így a fa magassága eggyel nő, lesz egy új, egyelemű gyökerünk. Ez az egyetlen eset, hogy a fa magassága változhat egy szinttel, egyébként nem változik.
   Pl. 41-el akarok bővíteni.

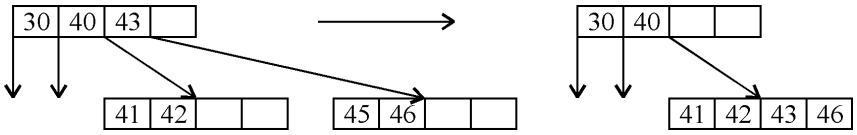


Törlés: Bármely lapról lehet.

Meg kell keresni az adott kulcsú elemet. Ha a törlendő elemet levéllapon találtuk meg, akkor fizikailag törlöm, a mutatókkal nem kell foglalkozni. Ha nem levéllapon találtuk meg, akkor a törlendő elemet felülírom a rendezettségben az őt megelőző vagy a rákövetkező levéllapon lévő elemmel, és azt törlöm.

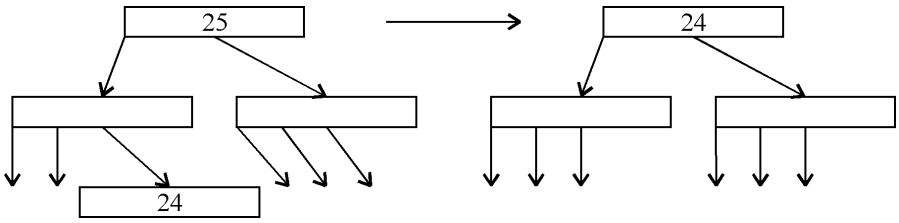
Ha az adott lapon az elemek száma minimálisan n marad, akkor nincs baj. Elem törlése annyit jelent, hogy a tőle jobbra állókat ráhúzom. Lehet, hogy kevesebb lesz az elemek száma, mint n, ilyenkor a szomszédos lapból kölcsönkérünk elemet. Ez csak akkor lehetséges, amikor a szomszéd lapon több mint n db elem van, és ekkor az eggyel fentebbi lapon keresztül áthúzom az adott elemet. Gyakorlatilag ez az előbbi vágás inverze.

Pl. 45 törlése.



Ha két szomszédos lapon az elemek száma $2n$-1, akkor összevonjuk a két szomszédos lap elemeit, megszüntethetjük az egyik (az üres) lapot, és a gyökérből hozzávesszük a „középső” elemet. Ilyenkor az előző szintet is csökkentjük egy elemmel. Lehet, hogy magasabb szinteken is kevés elem van, itt is hasonlóan járunk el. Végiggyűrűzhet a fán, lehet, hogy a gyökérlapot is meg kell szüntetni, ekkor a fa magassága csökken eggyel.

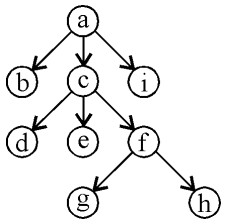
Pl. 25 törlése.



Rendezés: Nincs.
Feldolgozás: Alapja a keresés.

Hierarchikus lista

Tekinthetjük ezt a listát a listák általánosításainak. A listák elemei listák is lehetnek, ezért a listaműveletek értelmezhetőek erre a adatszerkezetre. Pl. (Ez egy ötelemű lista(amelynek utolsó eleme egy(ötelemű lista))). LISP dolgozik ezzel az adatszerkezettel. Tekintsük az alábbi rendezett fát:



Ennek egy reprezentációja: (a(b)(c(d)(e)(f(g)(h)))(i)). Egy baloldali zárójel egy új szintet jelöl, egy bezáró zárójel pedig egy szint végét. Többágú rendezett fákat reprezentál.

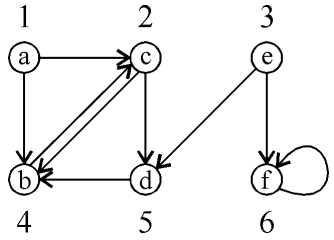
A hierarchikus lista egy fa szimbolikus kezelésére szolgálhat, az összes, a fáknál ismertetett algoritmus alkalmazható rá. Olyan sztring, amely szimbólumokat tartalmaz.

Hálós adatszerkezetek (háló)

A gráf egy homogén, dinamikus adatszerkezet, a fák általánosítása, olyan adatszerkezet, amelyben az elemeknek akárhány megelőzőjük és rákövetkezőjük lehet, beleértve azt is, hogy két elem között olyan kapcsolat van, hogy az egyik a másik megelőzője és rákövetkezője és viszont (kölcsönösen). Egy elem akár önmaga megelőzője és rákövetkezője is lehet. A gráfnak nincs kitüntetet eleme.

A gráf hálózatok modellezésére használható, reprezentálni összefüggő irányított gráf segítségével lehet. A szűkebben vett számítástechnikában csak a hálózatoknál jelenik meg.

Pl.



Gráfok reprezentációjára a szomszédossági listát vagy a szomszédossági mátrixot használják.

A szomszédossági mátrix (csúcsmátrix) egy logikai mátrix, amelynek elemei: 0, 1. Ha a gráfnak n db eleme van, akkor a szmszédsági mátrix $n \times n$-es. Az oszlopokat és a sorokat is a gráf elemeivel címkézzük fel. A mátrix kitöltésének szabálya: a mátrix (i,j) eleme legyen

- 0, ha az i és j csúcsok között nincs rákövetkező vagy megelőző viszony,
- különben 1.

A fenti gráf szomszédossági mátrixa:

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	1	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

A gráf ábrázolására alkalmazható a szétszórt ábrázolás, ahol általában a gráfnak egy multilista felel meg. Mutatótömböket veszek fel az adatelemek mellé. Kezeléshez lehet sorszámozni. Annyi elemű mutatótömböt építek fel, ahány eleme van a hálónak. E tömb elemei mutatótömbök, a mutatótömb elemei a rákövetkezőkre mutatnak.

A fenti gráf szomszédossági listája:

1.	2	4
2.	4	5
3.	5	6
4.	2	
5.	4	
6.	6	

A gráfon belül útnak nevezzük az olyan elemösszességet, amelyek listát alkotnak (pl. a-c-d-b). A gráfban körútnak nevezzük az olyan listát, amikor az utolsó elem az elsőre mutat (pl. b-c-d).

A gráfok szerepéről a következő kérdéseket érdemes feltenni:

- Létezik-e olyan út a gráfban, amelyben az összes elem benne van (maximális út)?
- Egyes csomópontokból melyik másik csomópontokba vezet út?
- Van-e körút a gráfban?
- Melyik a maximális körút?
- Van-e olyan körút, amelyik az összes elemet tartalmazza?
- Hogyan lehet olyan algoritmust találni, amely a gráf elemeit csak egyszer érinti? (bejárás)

A műveleteket speciálisan lehet értelmezni.

Műveletek:

Létrehozás: Az üres háló létrehozása, ezután bővítjük. A szomszédossági mátrixban ez egy új sor és egy új oszlop felvételét jelenti, amelyeket megfelelően ki kell tölteni.
Törlés: Egy elem törlése esetén az elemet reprezentáló sort és oszlopot ki kell törölni.
Csere: Megkeresem az elemet és felülírom.
Rendezés, keresés: Nincs.
Bejárás (keresés, háló): Ezen alapszik a gráf feldolgozása. Létezik szélességi és mélységi bejárás.

I. Szélességi keresés

Az algoritmus a bejárás pillanatnyi állapotát a csúcsok színezésével (fehér, szürke, fekete) tartja számon. Kezdetben minden csúcs fehér, és később szürkére, majd feketére változhat. Egy csúcs elértté válik, amikor először találunk rá a keresés során, és ezután a színe nem lehet fehér. Így a szürke és a fekete csúcsok már elért csúcsok, de a szélességi keresés megkülönbözteti ezeket is: egy fekete csúcs összes szomszédja elért csúcs, de a szürke csúcsoknak lehetnek fehér szomszédjaik, ezek alkotják az elért és a még felfedezetlen csúcsok közötti határt. Tehát eredetileg minden csúcs fehér, kivéve a kezdő csúcsot, amelyik szürke. Egy csúcs akkor válik szürkévé, amikor elértük, és akkor feketévé, ha a belőle kiinduló összes élt már átvizsgáltuk.

A szélességi keresés létrehoz egy szélességi fát, amely kezdetben csak a gyökeret tartalmazza. Kezdőpontnak azt az elemet érdemes választani, amely a legtöbb rákövetkezővel rendelkezik. A fa gyökre az *s* kezdő csúcs. Ha egy fehér *v* csúcsot elértünk egy már elért *u* csúcs szomszédságának vizsgálata során, akkor a fát kiegészítjük a *v* csúccsal és az (*u*,*v*) éllel. Egy csúcsot legfeljebb egyszer érhetünk el.

Az algoritmus kezdetén minden, az *s*-től különböző csúcs színét fehérre kell állítani. A rákövetkezőit a Q sorban tároljuk, ebben kezdetben csak az *s* van, majd elemei a szürke csúcsok lesznek. A Q elemeit egy ciklus dolgozza fel, amely addig fut, amíg el nem fogynak a szürke csúcsok (minden csúcs fekete nem lesz), azaz amíg van olyan csúcs, amelynek szomszédsági listáját még nem dolgoztuk fel. A ciklusban kiválasztjuk a Q sor legelső elemét, és ha ennek van olyan szomszéd csúcsa, amelyet még nem értünk el, azaz fehér, akkor a Q sor végére fűzi a csúcsot. Ha a kiválasztott csúcs minden szomszédját megvizsgáltuk, akkor a csúcsot vegyük ki a sorból, és színét változtassuk feketére.

II. Mélységi keresés

A mélységi keresés során az utoljára elért, új kivezető élekkel rendelkező *v* csúcsból kivezető, még nem vizsgált éleket derítjük fel. Ha v-hez tartozó összes élt megvizsgáltuk, akkor a keresés „visszalép”, és megvizsgálja annak a csúcsnak a kivezető éleit, amelyből *v*-t elértük. Ezt addig folytatja, amíg el nem éri az összes csúcsot, amely elérhető az eredeti kezdő csúcsból. Ha marad olyan csúcs, amelyet nem értünk el, akkor ezek közül valamelyiket kiválasztjuk, mint új kezdő csúcsot, és az eljárást ebből kiindulva megismételjük. Ezt egészen addig folytatjuk, amíg az összes csúcsot el nem értük.

A szélességi kereséshez hasonlóan a csúcsok állapotait ebben az esetben is színekkel különböztetjük meg. Kezdetben minden csúcs fehér, amikor elérünk egy csúcsot, akkor szürkére színezzük azt, és befeketítjük, ha elhagytuk, azaz amikor a szomszédsági

listájának minden elemét megvizsgáltuk. Ez a módszer biztosítja, hogy minden csúcs pontosan egy mélységi fába kerüljön, azaz, hogy ezek a fák diszjunktak legyenek.

A színezés ugyanígy történik, mint a szélességi keresőnél, de alapvető különbség az, hogy a fát szintenként építem fel.

Heterogén adatszerkezetek

Az adatszerkezet elemei más-más típusúak lehetnek.

REKORD (record, tuple)

A rekord, mint a tárban létező adatszerkezet, statikus. Kötött számú és kötött sorrendű mezőből áll, az egyes mezők más-más, tetszőleges típusú értékeket tartalmaznak. A rekordban minden mezőt megnevezünk, és rá közvetlenül a mezőnevével tudunk hivatkozni.

Tárolása lehet folytonos és szétszórt is. Általában folytonos a jellemző, bár ez a kérdés implementáció-, rendszerfüggő (részben hardverfüggő is). A folytonos tárolás esetén a mezők egymás után helyezkednek el.

Műveletek

Létrehozás: Létrehozom a szerkezetet a név, a típus megadásával. Fontos a mezők sorrendje! Az egyes mezőkhöz a nevük alapján értéket rendelek. Előfordulhat az az eset is, hogy nem minden mezőhöz rendelek értéket.
Bővítés: Szűkebb értelemben nem létezik, mert a rekord statikus. Bővítés olyan értelemben létezhet, hogy azon mezőkhöz, melyekhez a létrehozáskor nem rendeltem értéket, most hozzárendelek.
Törlés: Nincs, legfeljebb logikai, olyan hogy a mezőnek speciális értéket adok. A fizikai törlés realizálható cserével.
Csere: Bármely mező értékét felülírhatom.
Rendezés, keresés, bejárás: Nem értelmezhető.
Feldolgozás: A mezőnevek alapján.

Az itt leírtak a tárban lévő rekordok jellemzői, jelentőségük az állományoknál mutatkozik meg.

Egyes nyelvek megengedik, hogy a rekord ne statikus, hanem dinamikus legyen. (pl. Pascal) Az összes homogén adatszerkezet elemeinek az értékei vagy atomiak (tovább nem oszthatók, skalár típusúak), vagy rekordok; tágabb értelemben egy adatszerkezet elemei adatszerkezetek lehetnek. A rekordok mezőinek értéke vagy atomi (a rendszerek nagyobb részében), vagy, modernebb rendszerekben, tetszőleges adatszerkezet típusú lehet (pl. másik rekord). Így a rekord dinamikussá válik, ha a egyetlen mezője dinamikus. Adatbáziskezelő rendszereknél ez a definíció van előtérben (ortogonalitás). A rekordok egymásba ágyazhatóak. A relációs adatbáziskezelő rendszerek rekordja a *tuple*.

Állomány (file)

Az adatok periférián jelennek meg, az állomány fogalom mindig periférián értelmezhető, bármely periférián. A háttértár határozza meg a fizikai struktúrát, mi csak háttértárakon lévő állományokkal fogunk foglalkozni. A valós világ modellezése (az absztrakció első szintje) ugyanúgy történik állományok esetében is, mint adatszerkezeteknél (az állomány kialakítása). Második szinten megjelenik a fizikai állomány. Így létezik egy logikai és egy fizikai állomány.

A logikai állomány fogalmi rendszere

A logikai állomány adatelemekből (adattételekből) épül fel. Az *adatelem* (adattétel) a logikai állomány legkisebb, önállóan értelmezhető része. Jellemzői:

- név,
- típus: milyen típusú értéket vehet fel, tetszőleges,
- hossz: általában karakterekben értendő, lehet:
  - fix hosszúságú (pl. születési évszám (jobb esetben 4 karakter)),
  - változó hosszúságú (pl. születési hely).

A kérdés itt is az, hogy fizikai szinten hogyan jelennek meg ezek.

Az *adatcsoport* az adatelemek önálló névvel ellátott együttese. Típusai:

- vektor adatcsoport: Különböző típusú adatelemeket fog össze valamilyen nagyobb logikai egységbe (pl. a lakcím tartalmazza az irányítószámot, településnevet, utcát, házszámot).
- ismétlődő adatcsoport: olyan adatelemünk van, amelyiknek több értéke lehet (halmazértékű). Az ismétlődések száma lehet 0 is. (pl. gyermekünk neve).
- összetett adatcsoport: Az előző kettő kombinációja: egy vektortípusú adatcsoport valamelyik eleme ismétlődik.

A *logikai rekord* az adatelemek és adatcsoportok adott sorrendű, logikailag összetartozó együttese. Nincs önálló neve.

A *rekordformátumok* a logikai rekordok szerkezetére vonatkozó osztályozást tartalmazzák. Típusai:

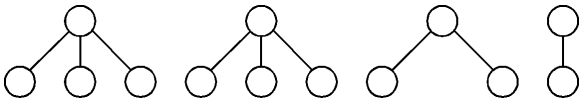
- fix rekordformátum (fix hossz): A logikai rekordok azonos szerkezetűek és azonos hosszúságúak. Ez eleve kizárja a váltakozó hosszúságú adatelemeket és az ismétlődő adatcsoportokat. A legegyszerűbb szerkezetű logikai rekord.
- változó rekordformátum: A logikai rekordok szerkezete azonos, de hosszuk eltérő lehet. Ha van változó hosszúságú adatelem, vagy ismétlődő adatcsoport, akkor ezt alkalmazhatjuk. A leggyakoribb rekordformátum.
- határozatlan rekordformátum: A rekordok hossza és szerkezete is különbözhet. Adatelemek hiányozhatnak az egyes rekordokból. A kérdés az, hogy fizikai szinten ezt hogyan tudom kezelni.

A *logikai rekordazonosítónak* nevezzük azt az adatelemet, vagy az adatelemeknek azon együttesét, amelynek vagy amelyeknek értéke egyedi, azaz minden konkrét rekordban más és más (pl. személyi szám, sorszám stb.). Lehet, hogy létezik, lehet, hogy nem. Ha ez egyetlen adatelem, akkor egyszerű, egyébként összetett rekordazonosítóról beszélünk.

A *logikai állomány* a logikai rekordok valamilyen feldolgozási cél, tartalom, vagy forma szerint összetartozó együttese, amelyet névvel látunk el. Az állomány lehet, hogy azonos szerkezetekből, vagy különböző elemekből áll. A logikai állomány egy absztrakt adatszerkezet. Akkor jön létre, ha a felsorolt jellemzői adottak, és megadom a nevét, valamint az adatokat. Beszélhetünk az állomány szerkezetéről is:

- struktúra nélküli: A logikai rekordok sorrendje tetszőleges.

- asszociatív: Ha elvégezhető a rekordoknak valamilyen egyértelmű, diszjunkt csoportosítása. Pl. a logikai rekordazonosítók alapján végzett csoportosítás, amely 1 elemű csoportokat ad vissza.
- szekvenciális: A logikai rekordok között valamilyen sorrend értelmezett. (pl. azonosító szerinti rendezettség)
- hierarchikus: különböző logikai rekordokat pakolok egy állományba úgy, hogy olyan szerkezetet alkotnak, amely leginkább egy fával jellemezhető (pl. személyi adatok + félévi eredmények)



A fizikai állomány fogalmi rendszere

A *mező* kölcsönösen egyértelműen felel meg az adatelemnek. A különbség az, hogy jellemzője az ábrázolási mód, hiszen megjelenik a háttértáron. A hossz itt byte-okban értendő. Nem fix hosszú adatelemek kezelésénél az adatelem mezőjét fix hosszon kell megjeleníteni.

A *mezőcsoport* megfeleltethető az adatcsoportnak, az adatcsoport megjelenítésére szolgál.

A *blokk* megfeleltethető a logikai rekordnak, a *fizikai állomány* pedig megfeleltethető a logikai állománynak, nem más, mint blokkok sorozata. A fizikai állománynak saját neve van, ennek kezelése operációs szinten zajlik.

A blokk az az adatmennyiség, amely egyszerre mozog a tár és a periféria között. Az adatátvitel egysége, byte-ban értendő. A blokk és a logikai rekord megfeleltetése:

- 1 logikai rekord alkot 1 blokkot,
- több logikai rekord alkot 1 blokkot (szűkebb értelemben vett blokkolás),
- 1 logikai rekord több blokkban jelenik meg (szegmentálás esete).

Kezelése rendszerfüggő. Egyes rendszerek a blokk méretét rögzítik, és a felhasználó nem változtathatja, mások megengedik, hogy a felhasználó szabja meg. Ha a blokk mérete rögzített, akkor elképzelhető, hogy a rendszer nagy blokkoknál automatikus szegmentálást hajt végre. Létezik olyan rendszer, amely megengedi a szegmentálást, de az ilyen rendszer kevés. A felhasználó ritkábban tud tényleges módon szegmentálni.

Hogyan lehet a blokkokat és a logikai rekordokat megfeleltetni? Az adatelem lehet fix és változó hosszúságú. Az első esetben nincs gond, könnyen lehet kezelni. A változó hosszúságnál megkülönböztetünk numerikus és szöveges adatot. Numerikus számábrázolás esetén a hossz az ábrázolási mód megadásával lesz fix. Szöveges típusnál a sztringnél megbeszélt ábrázolási módok mindegyike érvényes.

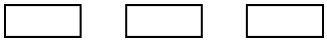
I. Fix rekordformátum

Kezelése a legkönnyebb, minden rendszer tudja kezelni. Ha egy rekord hosszát rögzítettem, akkor jellemzően a többi is ilyen lesz, így nem kell az egyes rekordhoz kötni az információt. A rekordhossz az állomány jellemzőjeként írható le.

1. 1 blokk – 1 rekord: Az összes blokkjellemzőt tudom.
2. Blokkolás: Megmondom, hogy hány rekordot pakolok össze egy blokkba. Ezt nevezzük blokkolási tényezőnek, ami szintén az állományra jellemző. Az utolsó blokk lehet töredékblokk, de a többinek azonosnak kell lennie.
3. Szegmentálás: Meg kell mondani a rekord feltördelését (azonos hosszú byte-sorozatokra bontom szét, de a szerkezetük nem lesz azonos), és ez minden rekordra ugyanaz lesz. Csak egyszer kell tördelni, és az utolsó blokk lehet rövidebb.

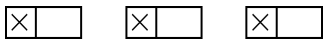
II. Változó hossz

1. 1 rekord-1 blokk
   A. A blokk mérete rögzített, úgy választom meg, hogy a leghosszabb rekord is beleférjen.



Ezért azon rekordok, melyek kisebbek, nem foglalják el teljes egészében a blokkot. A ki nem töltött részt speciális bitkombinációval tölthetjük ki, „szemétbyte”-okkal. Változó lehet egy adatelem, vagy egy ismétlődő adatcsoport. Fizikai szinten ilyenkor plusz információt kell közölni, az ismétlődések számát. Ez a rekordhoz van hozzárendelve, a blokkméret viszont nem függ a rekordtól. Ha nagy a különbség a rekordok között, akkor rossz a helykihasználás, viszont könnyű kezelni.

B. A blokkméret megegyezik a rekordmérettel. A blokk is változó hosszú lesz, a blokk elején jelölni kell a blokk hosszát.



2. Blokkolok

A. A rekordok szabályosan vesznek fel különböző hosszakat, és csak néhány követi egymást (pl. 3 db). Ilyenkor tudok úgy blokkolni, hogy a blokk mérete azonos, a rekord méretét jelölöm. A blokkszerkezetét csak egyszer kell leírni, de a rekordra vonatkozó hosszinformációt a rekord elejének tartalmaznia kell.

B. Fix blokkmérettel dolgozom. A rendszer veszi sorba a rekordokat, és addig pakolja a blokkba, amíg belefér, a megmaradt helyet „szemétbyte”-tal tölti fel. A többi rekordot a következő blokkba rakja.

C. Maximális blokkméretet határoz meg. A rendszer összepakolja a rekordokat mindaddig, amíg belefér a blokkba, de a szemetet elhagyja. Kezdi rakosgatni megint. Ha belefér a maximális blokkméretbe, akkor belerakja. Így változó méretű blokkok keletkeznek, amelyeknél nem csak a rekordok méretét, hanem a blokk hosszát is jelölni kell. Ennek igen jó a helykihasználása, és kell plusz információ, a blokk mérete is.

D. Szegmentálás: Nagyon nyűgös, ugyanúgy, mint a blokkolásnál, plusz információt kell elhelyezni a blokkban. Rendszertől függ, hogy a plusz információt hogyan tárolja. A blokk mérete pl. lehet azonban egy külön táblában is.

III. Határozatlan hossz

Minden rekord a tárolásánál az előzőek mellett olyan plusz információt kell bevinni, amely az adott rekordra vonatkozik, azaz hogy a rekordnak milyen mezői hiányoznak (pl. lánykori név). A rendszerek itt általában nem engedik meg a blokkolást és a szegmentálást, általában nem tudják kezelni a határozatlan hosszat.

Operációs rendszer-függő, hogy hogyan lehet ezeket megoldani. Van olyan rendszer, amely semmit nem enged, de van olyan is, ahol a blokkméretet is be tudom állítani.

Szalagon a blokkok egymás után helyezkednek el. Lemezen a fizikailag össze nem függő blokkok alkotnak egy állományt és bármely blokk közvetlenül elérhető, minden byte-nak van lemezcíme.

A logikai rekordazonosító fizikai szinten az (elsődleges) kulcs, a másodlagos kulcs azon adatelemekből áll, amely(ek) nem tartoznak logikai rekordazonosítóhoz.

A logikai – fizikai állományok kapcsolata tetszőleges lehet:

- 1 fizikai állomány 1 logikai állomány rekordjait tartalmazza: Ez a leggyakoribb.
- 1 fizikai állomány több logikai állomány rekordjait tartalmazza
- 1 logikai állomány több fizikai állományban helyezkedik el.

A fizikai állománynak saját neve van, mellyel hivatkozni lehet rá, kezelése operációs rendszer-függő.

Műveletek (Mindig logikai rekordra vonatkoznak.)

Létrehozás: Az állományt valamelyik háttértáron hozom létre. Ki kell választanom a fizikai állomány szervezési módját, vagyis az állomány tárolási szerkezetét. Az adott szervezési módnak megfelelően kell a logikai rekordokat elhelyeznem a fizikai állományban.

Bővítés: Az állomány rekordjainak darabszáma nő, új rekord kerül be az állományba.

Törlés:

I. Logikai törlés

Az állomány rekordjainak darabszáma nem változik, de a logikailag törölt rekordok meg vannak jelölve, a továbbiakban a feldolgozásban nem vesznek részt. Rendszerfüggő a megvalósítása, a fizikai állományban minden logikai rekordhoz van egy jelző, melynek két állapota van: törölt ill. élő.

1. Törlőbájtot alkalmaznak, hogy eldönthető legyen, hogy tényleges rekord-e, vagy már logikailag törölt. Ez plusz 1byte-on tárolható.
2. Logikai törlést jelentheti valamelyik mező tartalmának felülírását speciális bitkombinációra.

A rendszer feldolgozáskor a leveszi a logikailag törölt rekordot, de van olyan is, amelyik ezt a felhasználóra bízza.

II. Fizikai törlés

Élesen elválik a logikai törléstől. Ilyenkor az állomány rekordjainak darabszáma csökken, a rekordok nem nyerhetőek vissza. Nehezebb végrehajtani. A kérdés az, hogy a rendszer egyáltalán tud-e ilyet. Általános az a válasz, hogy nem.

Csere: A rekord bármely mezőjének (kivéve a rekordazonosítót) értékét cserélhetem valamely más értékre.

Elérés: Olyan tevékenység, mikor az állomány bármely rekordját akarom megfogni. Figyelembe kell-e venni a többi rekordot?

I. soros elérés: Alapja a rekordok fizikai sorrendje (ahogyan elhelyezkednek a rekordok a háttértáron).
II. szekvenciális elérés: Az alapja valamilyen logikai sorrend, az x. rekordot csak az előtt állók elérése után érhetem el.
III. közvetlen elérés: Alapja az elsődleges, vagy másodlagos kulcs, bármely rekordot egyből megtalálok, függetlenül a fizikai és logikai sorrendtől. Ilyen elérés csak lemezen lehetséges.

Keresés: Soros és szekvenciális elérésnél egy kitüntetett rekordot kell megfognom. A szokásos három módszerrel.

Rendezés: A szokásos (kulcs alapján).

Feldolgozás: Egy adott állományban lévő információhoz akarok hozzáférni. Egy-egy szerkezetet az minősít, hogy a feldolgozást mennyire segíti. A feldolgozás alapja a szerkezet és az elérés, keresés.

Újraszervezés: Az állományt a háttértár egy másik területén újra létrehozzuk úgy, hogy közben megváltozik vagy a szerkezete, vagy a tartalma, esetleg mindkettő.

A műveletek szervezési mód függőek. Állományszerkezetek (szervezési módok):

I. egyszerű: Csak a logikai rekordokat tartalmazza.
II. összetett: A logikai rekord adatain túl szerkezethordozó adatok is megjelennek, melyek magáról az állományról adnak információkat. Ezek a feldolgozást segítik.

Egyszerű állományszerkezetek

Egyszerű állományszerkezetnél az osztályozás alapja:

- Van-e kapcsolat a logikai rekordazonosító és a logikai rekord háttértáron elfoglalt helye között?
- Van-e kapcsolat a logikai rekordazonosítók között?

szeriális:	nem	nem
szekvenciális:	nem	igen
direkt:	igen	igen
random:	igen	nem

Szeriális állomány

Olyan állomány, amelynek nincs szerkezete, a rekordok sorrendje tetszőleges. Nem használja fel, hogy van rekordazonosító. Mindenféle háttértáron megjelenhet, mindenféle rekordformátumot tud kezelni, tetszőlegesen lehet blokkolni és szegmentálni.

Műveletek

Létrehozás: A tetszőleges sorrendben jövő rekordok bekerülnek az állományba a beérkezési sorrend szerint.
Bővítés: Mivel nincs rendezettség, így a bővítés nagyon egyszerű, az új rekord az állomány végére íródik.
Törlés: Fix rekordformátum esetén lemezes háttértárnál lehet fizikai törlés: az utolsó rekorddal felülírom a törlendőt. Szalagnál és nem fix rekordformátumnál csak logikai törlésről beszélhetünk.
Csere: A lemezen elhelyezett fix rekordformátumú szeriális állomány tudja csak a cserét, a többi esetben csak újraszervezéssel lehetséges.
Elérés: Csak soros elérésről beszélhetünk, mert csak fizikai sorrend van.
Keresés: Teljes keresés.
Rendezés: Nincs.
Feldolgozás: Az adott fizikai sorrendben végig fel tudom dolgozni az állományt, vagy addig, amíg a feldolgozandó rekordot meg nem találtam (soros elérés). Nem más, mint a rekordok végigolvasása. Viszonylag lassú.
Újraszervezés: A szerkezetátalakítás miatt ez nem merül fel, mert nincs szerkezet. Akkor van rá szükség, amikor logikailag törölt rekordok fizikai törlését akarom megvalósítani, vagy olyan cserét, amikor nem fix a rekordformátum vagy szalagon vagyok.

A szeriális állomány jelentősége az, hogy egyszerűen kezelhető, viszonylag gyors, kivéve, ha egy adott rekordot keresek, mert a teljes keresés időigényes. Minden operációs rendszer, programnyelv stb. tudja kezelni az ilyen típusú állományt. A szeriális állomány jelenik meg az adatbáziskezelő rendszerek táblázatai mögött. Alapvető szerepet játszik olyan eseteknél, amikor a rekordok véletlenszerűen állnak elő, vagy amikor a véletlenszerűen érkező rekordokat átmenetileg gyűjteni kell, hogy az így létrejövő állományból valamilyen szerkezettel rendelkező állományt hozzunk létre. A feldolgozás elkülönülten jelentkezik.

Szekvenciális állomány

Olyan állomány, ahol a rekordok az azonosító (elsődleges kulcs) szerint rendezettek. Ez egy egyértelmű sorrendet ad, egy adott szerkezetet biztosít. Létrehozható mind szalagon, mind lemezen. Minden rekordformátumot tud kezelni, a blokkolás is tetszőleges. A szakma szűkebb értelemben az azonosító szerinti növekvő sorrendben rendezett állományt nevezi szekvenciálisnak, de a csökkenő sorrendű is kielégíti a fogalmakat.

Műveletek

Létrehozás: Egy, vagy két lépésben történhet.

I. Ha egy lépésben történik, akkor nekem kell biztosítani, hogy a rekordok csak növekvő, vagy csökkenő sorrendben kerüljenek be. Ez nagy állományoknál kissé nehézkes.
II. Ha két lépésben történik a létrehozás: Létrehozok a rekordokból egy szeriális állományt, a rendszer ebből elkészít egy szekvenciálist, azaz rendezi azonosító szerint.

Bővítés: Nem hajtható végre közvetlenül, csak újraszervezéssel.
Törlés: Csak logikai törlést realizál, fizikait csak újraszervezéssel.
Csere: Fix rekordformátum esetén lemezen lehet, bármely más esetben csak újraszervezéssel.
Elérés: A szekvenciális állománynál a logikai és a fizikai sorrend egybeesik, ezért a soros és a szekvenciális elérés ugyanazt jelenti.
Keresés: Lineáris keresés, előnyösebb, ha több rekordot keresünk egyszerre.
Feldolgozás: Lineáris kereséssel egy rekordot az állományban vagy meg tudok találni, vagy nincs benne. Tegyük fel, hogy több rekordot keresek. Ilyenkor célszerű a keresendő rekordok azonosítóit ugyanúgy rendezni, mint az állományt, majd így keresni, mert így nem kell mindig az egész állományt végignézni, egy kereséssel meg tudom mondani, hogy az állományban benne vannak-e.
Újraszervezés: Általánosan növekvő sorrendű állományból csökkenő sorrendű állományt lehet létrehozni, vagy fordítva. Újraszervezéssel oldható meg a csere, a fizikai törlés és a bővítés. Nem a szerkezetet változtatom.

Ha egy szekvenciális állományt feldarabolok, akkor szintén szekvenciális állományokat kapok. Ez azért lényeges, mert általában kevés a hely, így a résztartományokkal könnyebb dolgozni. Lényeges lehet az állományok mérete a tárkapacitás miatt. Szekvenciális állományt sok rendszer tud kezelni (nem mindegyik), ezért alapvető szerepet játszik a jelenlegi problémák megoldásánál. Az ilyen típusú állományok az adatbáziskezelő rendszerek mögött is megjelennek.

A többi állományszerkezet bonyolultabb az eddigieknél. A direkt és random állományra a kulcstranszformációs táblázatoknál megbeszéltek vonatkoznak: az elsődleges kulcs és egy rekord lemezcíme között létezik egy leképezés. Ha az elsődleges kulcs olyan, hogy azonos a kulcsok és a rekordok száma, akkor direkt állományról beszélünk, ha ez nem teljesül, akkor random állományról. Itt is érvényesek a hash-függvények, csak itt nem a tárba, hanem a háttértárba képezünk, ami itt csak lemez lehet.

Direkt állomány

Akkor hozható létre, ha a hash-függvény egy-egy értelmű, azaz az azonosítók és az adott azonosítójú rekord lemezen elfoglalt helye között kölcsönösen egyértelmű leképezés van. Ha csak egyértelmű a leképezés, akkor random állomány hozható létre.

Műveletek

Létrehozás: Két lépésben.

I. A lemezen kialakítjuk az állomány üres szerkezetét, a vázat. A direkt állomány csak fix rekordformátummal dolgozik, nem lehet blokkolni és szegmentálni, vagyis egy rekordnak egy blokk felel meg. A szerkezet kialakításakor tehát lefoglalunk adott számú (amennyi rekordot tárolni akarok), azonos méretű üres tárhelyet az összes elvileg előforduló

lehetőségnek, és sorszámozzuk őket. A rekordazonosító elvi lehetőségei és a gyakorlati lehetőségei körülbelül megegyeznek. Megadom a hash-függvényt, amely megmondja, hogy az adott azonosítójú rekordot melyik rekordhelyen kell elhelyeznem. Lehet, hogy abszolút lemezcímet mond, vagy csak egy sorszámot, amelyből kiszámítható az abszolút lemezcím.

II. A rekordok tetszőleges sorrendben érkeznek. A hash-függvény alapján meghatározzuk, hogy melyik rekordhelyre kerülnek, majd elhelyezzük őket. Így a létrehozás végén lesznek tényleges rekordokat tartalmazó rekordhelyek (a bemásolt rekordokkal), és üres rekordhelyek is.

Bővítés: Direkt módon nem létezik. Úgy lehetséges, hogy eddig üres rekordhelyekre helyezünk megfelelő rekordokat. Az üres rekordhely jelzése lehet például egy speciális bitkombináció, pl. minden bit 0-val egyenlő.

Törlés: Az adott rekordhely bitkombinációját a fentebb leírt bitkombinációra állítom. Ez egy speciális keveréke a fizikai és a logikai törlésnek, hiszen tárhely nem szabadul fel, de fizikai törlés, mert a rekordot nem lehet visszanyerni, tartalma elvész.

Csere: Simán működik. Kulcsot nem lehet cserélni.

Feldolgozás: Ez az állománytípus nyújtja a legsokrétűbb és a leggyorsabb feldolgozást. A feldolgozás alapja a közvetlen elérés.

- A fizikai sorrend szerint sorosan feldolgozhatóak, elérhetőek a rekordok.
- Szekvenciálisan, hiszen a sorrend a hash-függvényből következik. Itt a logikai és a fizikai sorrend megegyezik.
- Létezik közvetlen elérés. Ezért hozták létre ezt az állománytípust.
- Szakaszos szekvenciális elérés. Közvetlen módon belépek az állományba, rálépek az állomány egy adott rekordjára, és onnan kezdve szekvenciálisan érem el a többi rekordot.

Mindig figyelnem kell, hogy az éppen feldolgozandó rekord tényleges rekord-e.

Keresés: Ez a kérdés a közvetlen elérés miatt nem igazán merül fel, de ha nagyon akarom, akkor bináris keresést tudok realizálni.

Újraszervezés: Csak akkor merül fel, ha megváltozik az állomány szerkezete. Szigorú értelemben vett újraszervezés nem létezik.

Csak fix rekordformátumot tud kezelni, nincs blokkolás és szegmentálás. Néha előfordulnak olyan rendszerek, amelyek ismerik a blokkolást, és a lefoglalt helyek darabszáma fix.

A rendszereknek és programnyelveknek csak egy része ismeri a direkt állományszerkezetet. Van néhány rendszer, amely a relatív állományt tudja kezelni, ezek megkövetelik, hogy a rekordazonosító sorszám legyen, megkövetelik az üres szerkezet létrehozását. Viszont az állomány bővíthető a végén.

Random állomány

Ha a hash-függvény csak egyértelmű, akkor random állományról beszélünk. Random állománynál túlcsordulás van, a szinonimákat is kezelni kell.

Műveletek

Létrehozás: Első lépésben létrehozzuk az állomány szerkezetét, az üres rekordhelyeket, amit a túlcsordulási technika szab meg, és választunk egy hash-függvényt. A második lépés a rekordok elhelyezése. Jön a rekord és a hash-függvény, mert a túlcsordulási technika szerint bekerül valahova. A létrehozás végén léteznek olyan rekordhelyek, amelyek üresek és amelyek tényleges rekordot tartalmaznak.

Bővítés: Üres rekordhelyre rekord bevitele a rekord kulcsa és a hash-függvény segítségével.

Törlés: Logikai törlésről beszélhetünk, bármely rekord felülírható csupa nullával. Láncolás esetén (ld. szinonimák kezelése) kitörlöm a láncból fizikailag is.
Csere: Fix rekordformátumnál nem probléma, nem fixnél külön nyűg, de megy.
Elérés: Kvázi közvetlen. A nem túlcsordult rekordok közvetlenül elérhetőek, a többit keresnem kell, de a szinonimakezelési technikát jól megválasztva szinte minden rekord közvetlenül elérhető.
Rendezés, keresés: Nincs.
Feldolgozás: A közvetlen elérés alapján.
Újraszervezés: Csak logikai törlés létezik, így a törölt rekordok ott maradnak. A random állomány a túlcsordulási technikától függően nagyon érzékeny a túlcsordult rekordra, ami jelentősen lassíthatja az elérést, így az újraszervezés a túlcsordult rekordok (több mint 10%) és a fizikai törlés miatt következik be. A cél a fizikai törlések és a túlcsordul rekordok számának minimálissá tétele. Egy új hash-függvény vagy szinonimakezelési technika bevezetése a szerkezetet változtatja meg. Hogyan kezeljük a szinonimákat?

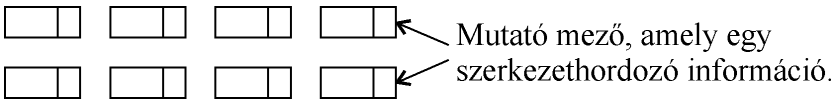
I. Nyílt címzés

A túlcsordult rekordot ugyanott helyezzük el, mint a nem túlcsordult rekordot. Előnye, hogy egyformán tudom kezelni a túlcsordult és a nem túlcsordult rekordokat. Hátránya, azon a helyen, ahonnan túlcsordult a rekord, nincs információm arról, hogy hova került ténylegesen a túlcsordult rekord. Jönnek a rekordok és elhelyezem őket. Ha adott rekordhelyen már nincs hely (egy rekord túlcsordul), akkor elindulok a túlcsordulás helyétől az állomány vége felé és megkeresem az első szabad helyet, és ott elhelyezem. A problémák ugyanúgy fellépnek, mint a táblázatoknál. Minden elemnél meg kell vizsgálni, hogy az adott helyen a helyhez rendelt elem van-e. Egy túlcsordult rekord esetleg olyan rekord helyét foglalja el, amelyik nem lenne egyébként túlcsordult, de így azzá válik. Ezt ki lehet küszöbölni úgy, hogy elsőnek a nem túlcsordult rekordokat helyezzük el, majd a túlcsordultakat. A szükségesnél több rekordhelyet kell, hogy lefoglaljak, hogy a túlcsorduláshoz viszonylag közel legyen üres hely, ez gyorsítja a visszakeresést.

Teljes keresést kell folytatni, ha az adott rekord helyén nem a keresett rekord van. Megáll, ha a keresett rekordot megtaláltuk, vagy az első üres hely esetén, tehát nincs közvetlen elérés.

II. Láncolás

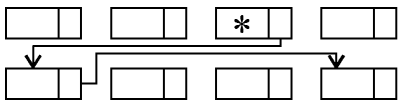
Az állomány szerkezete már összetett jellemzőket hordoz: a rekordok mindegyikénél megjelenik egy mutatómező (szerkezethordozó információ).



Ezt a mutatómezőt használom fel a túlcsordult rekord kezelésére, így a túlcsordulás helyén van információ arról, hogy hova lett elhelyezve a túlcsordult rekord. A nyílt címzést javítja föl keresés szempontjából.

1. Szabad helyek nyilvántartása

Az állományhoz létezik egy, az állománytól független táblázat, amely a rekordhelyek foglaltságát jelzi. Így elhelyezésnél a következő szabad helyet nem az állományban keressük, hanem a táblázatban, ami lényegesen gyorsabb, mert a táblázat a tárban van.



Az egy rekordhelyről túlcsordult rekordokat láncolja össze.

Tehát a szabadhelyet kikeressük a táblázatból, ott elhelyezzük a rekordot, és a címét elhelyezem az eredeti hely mutatójába, azaz a túlcsordulást az adott helyen jelöljük. Ha több túlcsordult rekord van, akkor az előző túlcsordult rekord mutatója kapja meg az új

túlcsordult rekord helyét. Ez a módszer gyorsítja a bővítést és a keresést, a karbantartás viszont bonyolultabb.

2. Független túlcsordulási terület alkalmazása

   A túlcsordult rekord elhelyezése nem ugyanott történik, mint a nem túlcsordult rekordé, hanem egy külön lemezterületen. Beszélünk mutatókról, független túlcsordulási területről, illetve elsődleges területről, amely a nem túlcsordult elemeket tartalmazza. A független túlcsordulási terület, ahová az összes túlcsordult rekord kerül, nem különbözik az elsődleges területtől. Itt ugyanúgy rekordhelyek vannak. Nyilvántartunk egy információt, hogy melyik az első szabad hely ezen a területen. A létrehozás folyamán tehát érkeznek a rekordok. Elhelyezzük őket az elsődleges területre. Ha egy rekord túlcsordult, akkor a független túlcsordulási terület első szabad helyére rakjuk, az elsődleges területnek a túlcsordulás helyén lévő rekordjának mutatója erre a túlcsordult rekordra mutat. A túlcsordulási területre a rekordok a túlcsordulás sorrendjében kerülnek be, egymás után. Ezután itt láncoljuk össze az egy helyről túlcsordult rekordokat. Itt az elsődleges adatterületnél nem kell többletrészt lefoglalni, viszont egy másik tárrészre is szükség van.

3. Osztott szabadhelyek módszere (cilindertúlcsordulási terület alkalmazása)

   Csak olyan lemezek esetén alkalmazható, ahol egy cilinderen belül több sáv van. Itt a sávok közül az első néhányat az elsődleges adatterületnek, míg a többit a túlcsordult adatterületnek tartjuk fenn. Ennél a technikánál a túlcsordult és nem túlcsordult rekordok egy fejmozgással elérhetőek, ezért gyors elérés valósítható meg. A probléma a sávok számának optimális kiválasztása, hogy nem lehet megmondani, hogy hány sávra van szükség a túlcsordult rekordok kezelésére, így lehet, hogy sok szabad területet foglal le.

Ezen szituációk problémafüggők. A fejlettebb rendszerek az utolsó kettőt kombinálják a probléma függvényében.

3. Vyssotzky-módszer

   Úgy kezeljük a túlcsordult rekordokat, hogy ne legyen túlcsordulás. Válasszunk ki néhány különböző hash-függvényt, például 5-öt. Vegyük az első hash-függvényt és az alapján próbáljuk elhelyezni a rekordokat. Ez addig megy, amíg el nem érkeztünk egy túlcsordulásig. Ekkor vegyük a második hash-függvényt és a rekordot próbáljuk meg elhelyezni aszerint. Ha sikerül, akkor azzal menjünk tovább, ha nem, akkor vegyük a 3.-t, majd ha azzal sem sikerül, akkor így tovább. Ha mindegyikkel túlcsordulttá válna a rekord, akkor helyezzük el a rekordot a nyílt címzés módszerével. Általában 4-5 hash-függvény és kb. +10% terület alkalmazásával a gyakorlatban nincs túlcsordulás.

A random állomány mindenféle rekordformátumot tud kezelni, és tud blokkolni.

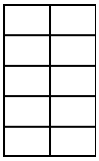
A direkt és a random állományokat jóval kevesebb rendszer képes kezelni, az operációs rendszereknek és a programnyelveknek csak a töredéke. A problémák általában pedig olyanok, hogy nagy szükség lenne rájuk. Az adatbáziskezelő rendszerek mögött majdnem minden esetben van random állomány. A rendszerekben általában beépített hash-függvények vannak. Olyan megkötés is lehet, hogy például a direkt állománynál a rendszer azt mondja, hogy az a sorszám. A direkt állomány kevésbé játszik szerepet.

Összetett állományszerkezetek

Van egy egyszerű szerkezetű alapállomány, és erre rakódik rá egy másik állomány. Minden esetben a feldolgozást segítik. Kialakításuknál két technika van, láncolás és indexelés.

I. Láncolás: Az információhordozó adatok az állományon belül jelennek meg (a rekord mellett mutatómezők jelennek meg – egyirányban, kétirányban láncolás, multilista).
II. Indexelés: Az állományon kívül jelennek meg az információhordozó adatok.

Mindkét technikának az alapja egy egyszerű szerkezetű állomány, amit alapállománynak nevezünk. Erre rakódnak rá a plusz szerkezethordozó információk. Az ilyen állományok (az alapállományok) általában szeriálisak vagy szekvenciálisak. Az indexelés esetén létrejön egy ún. információs tábla:



Két oszlopból áll és annyi sorból, ahány rekordja van az elsődleges állománynak. Az első oszlopban az alapállomány rekordjainak azonosítói vannak növekvő sorrendben. A második oszlop az adott rekord lemezcímeit tartalmazza vagy olyan információ, amiből a lemezcím egyértelműen meghatározható. Ez az elsődleges kulcsra vonatkozó egyszintű, teljes indexelés. Erre azért van szükség, hogy a szeriális, vagy szekvenciális állományra közvetlen elérést alkalmazhassunk. Az indextáblában a kulcs szerint keresek, így rögtön tudom a helyét. Kereséskor az indextáblában, a tárban binárisan keresek. Az indextáblában a kulcs szerint keresek, így rögtön tudom a helyét. Ha benne van a keresett elem kulcsa, akkor a lemezcím alapján közvetlenül meg tudom fogni.

Többszintű indexelésnél a tábla egy szekvenciális állomány, indexelhető, tehát e fölé is felépíthetek egy indextáblát. A második és annál magasabb szinteken nem teljes indexelés van. Ennek a módszernek nagyméretű állománynál van értelme, elősegíti a hatékonyabb keresést.

Nem teljes indexeléskor az indextáblában nem az összes, csak bizonyos rekordazonosítókat tüntetek fel, pl. minden huszadikat. Célja az indextábla méretének csökkentése. Ennek a módszernek azért van jelentősége, mert ha keresünk egy rekordot, akkor az indextábla alapján be tudjuk határolni az állomány egy részét, hogy melyik intervallumban keressük. Az adott intervallumon pedig már lehet szekvenciális keresést alkalmazni.

Hardveres indexelés alkalmazásakor pl. nem minden azonosítót tüntetek fel, csak a sávok utolsó rekordjainak azonosítóját. Így a keresés egy sávra csökken, a sávon belül pedig sorosan keresek. Ennek igen gyakori alkalmazása van.

Másodlagos kulcsra épülő indexelés: Az indextáblát felépíthetem úgy, hogy nem elsődleges, hanem másodlagos kulcs szerint indexelek. Így az összes olyan rekord címe szerepel a 2. oszlopban, és mindegyik … rendelkezik az adott másodlagos kulcsértékkel. Itt csak teljes indexelésnek van értelme az első szinten.

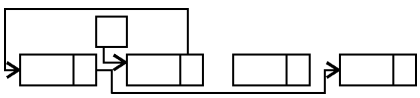
Terminológia: Az elsődleges kulcsra felépített indextáblákat és a (szeriális, vagy szekvenciális) alapállományt úgy hívjuk, hogy indexelt (szeriális, vagy szekvenciális) állomány. A másodlagos kulcsra felépített indextábla és állomány együtt az invertált állomány.

Elsődleges kulcsra épülő összetett állományszerkezetek

Szeriális alapállomány

I. Láncolt szeriális állomány

Adva van a szeriális állomány. Erre építünk rá egy láncszerkezetet, amely a szeriális állomány szekvenciális feldolgozását segíti. Ezt úgy valósítjuk meg, hogy minden rekord mellé az elsődleges állományban beépítünk egy mutatómezőt, melyet arra használunk, hogy növekvő vagy csökkenő azonosító sorrend szerint fűzzük fel a rekordokat. Egyirányú láncolt listát hozunk létre, a fej a legkisebb azonosítóval rendelkező rekordra fog mutatni.



Egyesítjük a szeriális állomány könnyű karbantarthatóságát a szekvenciális állomány könnyű feldolgozásával.

Műveletek

Létrehozás: Két lépésben valósítható meg.

1. Felvisszük a rekordokat, azaz létrehozzuk a szeriális alapállományt (mutatómezőkkel).
2. Az állományra ráillesztem az adatszerkezetet: elkészítjük a láncolást, kitöltjük a mutatómezőket (lemezcímekkel). Itt lemezcímeket fűzünk össze, így létezik egy fizikai sorrend, ahogy a rekordok bekerülnek az állományba, és létezik egy logikai sorrend, amelyet a láncolással alakítunk ki.

Bővítés: Szeriális állomány bővítése, az alapállomány jellemzői miatt az állomány végére bővíthetünk, a fizikai sorrend változatlan. Az új rekordot be kell illeszteni a láncba, ez egy rendezett egyirányban láncolt lista bővítése.
Törlés: Az alapállományban csak logikai törlés van. A logikai törlést kihasználhatom a láncban is, viszont innen fizikailag is törölhetünk.
Csere: Az alapállománynál megbeszéltek igazak itt is, azzal a kikötéssel, hogy azonosítót nem lehet felülírni. Ennek megoldását csak a rekord törlésével és újraírásával lehet megoldani. A csere tehát nem érinti a láncszerkezetet, mert azonosítót nem cserélek.
Elérés: Beszélhetünk soros elérésről (teljes kereséssel), amely a fizikai sorrend szerint történik, és szekvenciális elérésről, amit a logikai sorrend (a láncszerkezet) határoz meg (lineáris kereséssel). Ez a korábbi adatbáziskezelő rendszerek kedvenc alkalmazási technikája volt. Egyesítem a szeriális állomány előnyeit a szekvenciáliséval. A karbantartás egyszerű.
Feldolgozás: Ha az egészet akarom, akkor végigmegyek az állományon, ha csak részeket, akkor a láncon egy lineáris kereséssel könnyű a feldolgozandó elemek megkeresése.
Újraszervezés: Akkor kell alkalmazni, amikor túl sok az alapállományban a logikailag törölt rekord, vagy ellenkező irányú rendezettség szerint akarom felfűzni az állomány rekordjait.

II. Indexelt szeriális állomány

A szeriális állomány mellé, tőle függetlenül felépítek egy indexszerkezetet (közvetlen elérési lehetőséget) az elsődleges kulcsra. Ez az egyik leggyakoribb, hatékony állományszerkezet. Az elsődleges szinten itt csak teljes indexelés jöhet szóba, de egyaránt lehet egy- vagy többszintű indexelést is alkalmazni. Többszintű indexelés esetén sokkal bonyolultabb a kezelés.

Műveletek

Létrehozás: Két lépésben történik.

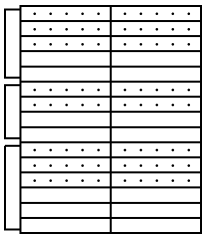
1. Létrehozzák a szeriális alapállományt.
2. Rárakják az indexállományt.

Bővítés: Fizikailag az alapállomány végére bővíthetünk, a szeriális állományszerkezet tulajdonsága szerint. Bármely bővítő rekord esetén aktualizálni kell az indextáblát.

1. Egyszintű indexelés esetén:
   A. Ha az indextábla befér a tárba, a bővítés index szintű kezelése egyszerű, nem jelent problémát, mert egy rendezett táblázatba való beszúrást kell elvégezni.
   B. Ha nem fér el a tárba, akkor az indextábla, mint szekvenciális állomány bővítését kell alkalmazni. Csak újraszervezéssel történhet.
1. A többszintű indexelés esetén bucket technikát alkalmazunk, az összes indexszintet át kell szervezni. A bucketek kialakításánál a szekvenciális állományt több szekvenciális állományra osztjuk. Az indextáblát is azonos méretű részekre osztják fel, ezek a bucket-ek (pl. 50-50). A második szint ezeket a bucketeket indexeli. Az első szinten nem töltjük ki ezeket az indexeket, lesznek bejegyzések és üres helyek. Innentől kedve

a bucketek átszervezéséről lesz szó.

Az indextáblán belül csoportokat képezünk, és ezen csoportokban nem minden helyet töltünk ki. A táblázat egy hézagosan vagy lineárisan kitöltött táblázat, így a csoportokon belül lehet bővíteni. A fő hangsúly az elsődleges indexelésen van.



Törlés: Alapállomány szinten csak logikai törlés van. A felsőbb szinteken, az indextáblában fizikai törlést is alkalmazok, úgy hogy törlöm a megfelelő indexbeírást. Az tehát a további feldolgozásban nem vesz részt.

Csere: Nem probléma, mert nem érinti az indexszerkezetet, csak az alapállományt. Csak olyan csere végezhető el, amely nem érinti az indextáblát, egyébként a láncolásnál megbeszéltek igazak.

Elérés:

1. soros: A fizikai, avagy az alapállománybeli, bekerülési sorrend szerint.
2. szekvenciális: Ha végigmegyek az indextáblán, minden egyes indexű tagot feldolgozok. A logikai sorrend szerinti elérés.
3. közvetlen: Az indextáblában keresek, és csak a keresett rekordot emelem ki.

Feldolgozás: valamely elérés segítségével.

Újraszervezés: Akkor kell, ha sok a logikailag törölt rekord és fizikailag törölni akarok, vagy meg akarom változtatni az indexszerkezetet, pl. egyszintűről többszintűre. Többszintű indexelés esetén az indexszerkezet változatlan.

Minden relációs adatbáziskezelő alapállománya, ahol általában csak egyszintű indexelés van megvalósítva.

Szekvenciális alapállomány

Ez azt jelenti, hogy elsődleges kulcs szerint rendezett, így nincs értelme, hogy arra láncszerkezetet építsünk, de indexelhető, rárakunk egy közvetlen elérési lehetőséget. Nagyon sok formája létezik. Kérdések:

- Milyen a struktúra, amit rárakok (egyszintű vagy többszintű)?
- Indexelési kérdések
- Hova kerülnek az új rekordok? (A szekvenciális sorrendet meg kell tartani.)

A szekvenciális alapállománynál a kialakításkor a logikai és a fizikai sorrend megegyezik, később nem.

Indexelt szekvenciális állomány

A közvetlen elérés megteremtése a cél a szekvenciális állománynál. Különböző kérdések merülnek fel a műveleteknél:

Műveltek

Létrehozás: A fizikai és logikai sorrend megegyezik, így egy lépésben kialakítható az indexállomány. Az alapállománnyal egyidőben jön létre az indextábla. A kérdés az, hogy milyen indexszerkezet alkalmazható: teljes vagy nem teljes, egyszintű vagy többszintű. Alapvető technika a

hardverindexelés. Itt lehet elsődleges szinten is nem teljes indexelés. Milyen bővítési technikát alkalmazzunk? Probléma a bővítés megvalósítása, hiszen az alapállományom rendezett.

Bővítés:

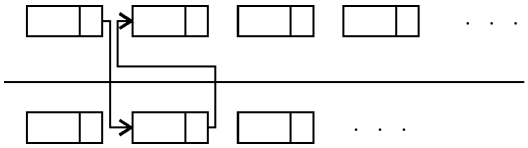
I. Újraszervezéssel bővítek

A szekvenciális alapállomány miatt van erre szükség. Ez csak akkor alkalmazható, ha igen ritkán bővítek, hiszen az átszervezés viszonylag sok időbe kerül. Előnye, hogy nem kell plusz információ a művelet végrehajtásához. A gyakorlatban szinte soha nem fordul elő.

II. Független túlcsordulási terület alkalmazása

1. Láncolás

A random állománynál megbeszéltek érvényesek ide is. A bővítő rekordok kezelését független túlcsordulási területen oldom meg, az alapállomány változatlan. Elválik a logikai és a fizikai sorrend, a bővítést, mint túlcsordulásnak a kezelését valósítják meg. Létrehozáskor a fizikai és a logikai sorrend megegyezik. Jön az első túlcsordult elem, és az a független túlcsordulási területre kerül. Azt, hogy honnan csordult túl, láncolással valósítják meg.



Minden rekord mellett van egy mutatómező, a túlcsordulási terület rekordjainak is, amely segítségével a logikai sorrendet felépítjük. Itt a bővítendő rekordot kezeljük túlcsordult rekordként, és érkezési sorrendben az a független túlcsordulási terület következő üres helyére kerül. Ezután a láncolást úgy kell elvégezni, hogy a logikai sorrendet tükrözze. Elválik a fizikai és a logikai sorrend, de a lánc segítségével őrizzük meg a szekvencialitást. Ez a technika nem érinti az indexszerkezetet, külön kezeljük a két dolgot. Az új rekordhoz tartozó bejegyzés, a bővítő rekord jellemzői az indextáblába kerülnek be a megfelelő helyre. A teljes indexelés közvetlen elérést nyújt, nem teljes indexelés kvázi közvetlen elérést. A láncolás adja a szekvenciális elérés lehetőségét. A bővítés nem egyszerű, lassú. A közvetlen elérés adott, a szekvenciális elérés romlik.

2. Csoportok

Csoportosítás az alapállomány rekordjainál. Nem teljes indexelést alkalmazunk, hanem az elsődleges állomány rekordjait csoportokra osztom és a kapott csoportokat indexelem az elsődleges szinten (pl. kerüljön minden csoportból az utolsó elem elsődleges kulcsa az indextáblába.). Mivel a sáv körbefordul, a kvázi közvetlen elérés nem romolhat. Ezt a technikát alkalmazzák általában, amikor hardveres indexelést végzünk. Egy sáv az alapegység, tehát az egy sávon elhelyezkedő rekordok alkotnak egy csoportot. Így itt a sávokat indexelem, a sávon utolsónak elhelyezkedő rekord kulcsa kerül az indextáblába. A második szinten értelemszerűen a cilindert indexelem. Ezzel a technikával egy sávot érhetek el közvetlenül.

Amikor kialakítom a szerkezetet, szekvenciális állományt alakítok ki, pl. kijelölök egy lemezterületet, és a sávokat növekvő sorrendben feltöltöm rekordokkal. Túlcsordult rekordnak nem a bővítendő rekordot tekintem, hanem a következő módon járok el: A bővítő rekordot a megfelelő csoporton belül beillesztem oda, ahova a rendezettség megkívánja. Hogy ezt megtehessem, az egy csoporton belüli összes mögötte lévő rekordot el kell csúsztatni jobbra, így megmarad a csoporton belüli logikai és a fizikai sorrend egysége. Ha nincs hely, akkor az eddigi utolsó rekord lesz túlcsordult, tehát ez kerül a független túlcsordulási területre, ahol a rekordok a

túlcsordulás sorrendje szerint vannak elhelyezve. Adott csoportból túlcsordult rekordot mindig a csoportjához tartozó lánc elejére helyezem a logikai sorrend megtartása miatt, mert a láncban a szekvencialitást meg kell őrizni.

Az elsődleges állományban nincs láncolás, nincs mutatómező, hanem a túlcsordulás kezeléshez szükséges plusz információt az elsődleges indextáblába írom, amelynek négy bejegyzése lesz a korábbi kettő helyett. A csoportból mindig a legutolsó csordul túl, ez kerül be az indextáblába. Ott fel kell feltüntetnem a csoport címét, az adott csoport kialakításkor meglévő maximális kulcsértékét (ez soha nem változik meg!), az aktuális maximális kulcsértéket, és az adott csoportból elsőnek túlcsordult rekord helyét. Ha a két kulcs eltér, akkor abban a csoportban volt túlcsordulás, ellenkező esetben nem volt.

Pl. A 636. sávon a kezdeti maximális kulcs értéke 335, az aktuális maximális kulcsérték 300. A túlcsordulási lánc a 758. sáv első rekordjánál kezdődik. Bővíteni akarok a 299 kulcsú rekorddal. A következőképpen járunk el:

A 299 kulcsú rekord helye a 636. sávon van, behelyezzük.

A 300 kulcsú rekord túlcsordul, elhelyezzük a független túlcsordulási területen a következő szabad helyre.

Az indexbejegyzés túlcsordult indexét ráállítja, és a 758-ra fog mutatni, az aktuális kulcs 299 lesz.

A létrehozáskor meglévő maximális kulcsok és a csoportok helye nem változik. Ha pl. 303 értékű rekorddal bővítünk, akkor az rögtön túlcsordulttá válik, és a láncot bővíteni kell, de az alapállományt nem változik.

3. Osztott szabad helyek módszere (cilinder-túlcsordulási terület alkalmazása)

A túlcsordult rekordok adott cilinderre kerülnek, használhatok cilinder és független túlcsordulási területet egyszerre. Csak olyan lemezek esetén alkalmazható, ahol egy cilinderen belül több sáv van. A cilinder sávjai közül néhányat az elsődleges adatterületnek, míg másokat a túlcsordult adatterületnek tartjuk fenn. Itt a túlcsordult és a nem túlcsordult rekordok egy fejmozgással elérhetőek, így ezzel a módszerrel gyors elérés valósítható meg. Igazak ide is a fentebb elmondottak (ugyanúgy történik minden, ugyanúgy láncolok, mint a fenti esetben).

Probléma lehet a sávok számának optimális kiválasztása, mert nem lehet megmondani, hogy hány sávra van szükség a túlcsordult rekordok kezelésére, így lehet, hogy sok szabadterület lesz lefoglalva.

A szituációk problémafüggők. A fejlettebb rendszerek az előző két módszert kombinálják, cilinder és független túlcsordulási terület módszert egyszerre használják, a probléma függvényében alkalmazhatóvá válnak.

4. Mutatótömb

Van túlcsordulási terület, amely vagy független, vagy cilinder túlcsordulási terület. Ekkor az alapállomány rekordjai változatlanok és a túlcsordult rekordok is. A túlcsordulást az indextáblán kezelem, a túlcsordult rekordokról minden információ ott van. Ekkor az elsődleges indextábla (teljes vagy nem teljes indexelésű) a következőt teszi:

- teljes indexelés: ekkor a bővítő rekordok csordulnak túl,
- nem teljes indexelés: a csoport utolsó rekordja csordul túl.

Az indextáblában jelzem, hogy a rekord hova csordult túl (itt szerepel a rekord kulcsa, címe, és a túlcsordult rekordok címei), úgy, hogy a bejegyzések sorrendje jelenti a szekvencialitást. Az indextábla egy sorában bármennyi túlcsordult rekord lehet, a mérete a bejegyzések számával együtt változik. Ez egy viszonylag ritkán alkalmazott megoldás.

Törlés: Az indexelt szekvenciális állománynál az alapállomány miatt általában logikai törlés van. Lehetne fizikai törlés is, de nem minden technika teszi lehetővé (ld. csoportosítás), ezért általában nem realizálják.
Csere: Nem probléma, mert nem érinti az indexszerkezetet.
Elérés: Fennmarad a szekvenciális elérés, amikor csak az alapállományt használom. Van közvetlen elérés is, ami a bővítési technikák miatt egy igen tág fogalom, mert a túlcsordult rekordok miatt nehéz ilyenről beszélni. A közvetlen technikák egy részénél kvázi közvetlen. Soros elérés nincs.
Feldolgozás: Olyan szituációban alkalmazzák, ahol az állomány módosítása viszonylag kevés. Bizonyos esetekben szekvenciálisan, bizonyos esetekben közvetlenül kell elérni az állományt. A közvetlen elérés célja az, hogy az állományt ne szekvenciálisan dolgozzuk fel.
Újraszervezés: Lényeges, akkor alkalmazzák, ha:

- sok a túlcsordult rekord (10%), mert lényegesen lelassítja a munkát,
- fizikai törlés esetén (alapállományt és az indexszerkezetet is),
- indexszerkezet megváltoztatása esetén.

Direkt, random állomány

Az egyszerű állományszerkezet az elsődleges kulcsra épül, közvetlen elérés van, ezért összetett állomány szerkesztése nem jöhet szóba.

Másodlagos kulcsra épülő összetett állományszerkezetek

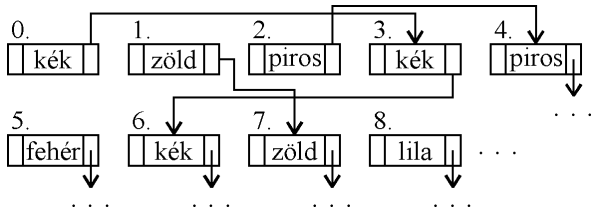
Az egyszerű állományszerkezetek a másodlagos kulcsra épülő állományszerkezeteket nem segítik semmilyen módon, minden rekordot külön meg kell fogni. Mind a négyféle állomány lehet alapállomány. A másodlagos kulcsra épülő adatszerkezetek segítésére az asszociatív adatszerkezetek szolgálnak.

Kérdések:

- Egyszerre egy vagy több másodlagos kulcsot tudnak-e kezelni?
- Ha több másodlagos kulcsot tud kezelni, akkor több szempontú keresést meg tudnak-e valósítani?

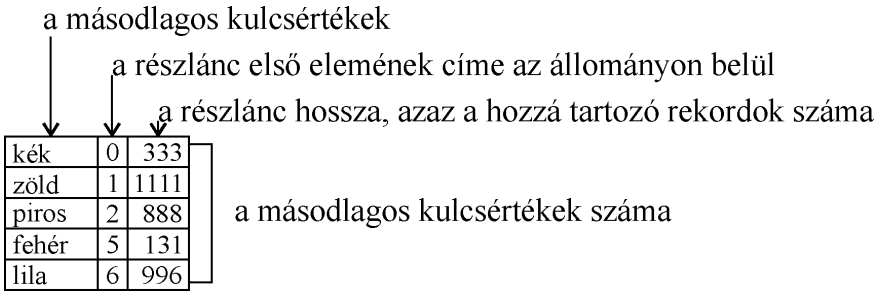
Láncolási technika

Ez a multilista állományszerkezet. A rekordok mellett megjelenik egy mutatómező. Kiválasztom azt a másodlagos kulcsot, amire építeni akarok.



Van egy alapállomány, valamilyen szerkezettel, és van a rekordoknak egy olyan mezője, amely alapján fel akarom dolgozni az állományt. Hogy ezt megtehessem, minden rekordnak rendelkeznie kell ezzel a mezővel. Így olyan kérdésekre is válaszolhatunk, hogy pl. a rekordok között hány van, amelyikben szerepel a lila szín? Minden rekordnak van egy mutatómezője, mely segítségével az azonos másodlagos kulcsértékű rekordokat egy láncba fűzzük. Az alapállományra olyan szerkezetet húzok rá, amely annyi részláncból áll, ahány különböző másodlagos kulcsérték szerepel az állományban. Egy részlánc egy asszociatív csoport, minden részláncnak van feje. Minden egyes rekord pontosan egy részláncban szerepel, és annyi részlánc van, ahány másodlagos kulcsérték előfordul az alapállományban (a másodlagos kulcs értéke bármennyi rekordban lehet azonos). A listafejeket az állománytól külön adom meg: felépül egy olyan táblázat, amely a részlista-fejeket

tartalmazza.



A táblázatnak 3 oszlopa, és annyi sora van, amennyi különböző másodlagos kulcsérték előfordul az állományban.

1.oszlop: A listafejek a másodlagos kulcsértékkel vannak azonosítva. A másodlagos kulcsértékek az előfordulás sorrendjében vagy rendezve kerülnek be a táblázatba.

2.oszlop: Az adott részlánc első elemének a címét adja meg az állományon belül

3.oszlop: Kiegészítő információ, az adott részlánchoz tartozó rekordok száma. A feldolgozásnál fontos.

Garantáltan a tárban kezelhető. A táblázat vizsgálatával (anélkül, hogy az állományhoz hozzányúlnék) megmondhatom:

- Hány darab másodlagos kulcsérték fordul elő?
- Melyek ezek?
- Hány olyan rekord van, amelyik ezzel a másodlagos kulcsértékkel rendelkezik?
- Van-e egy adott másodlagos kulcsértékkel rendelkező rekord?

Ha a táblázatban egy adott másodlagos kulcsértékű rekordot keresek, akkor a táblázatban meg kell keresni a hozzá tartozó sort. Ha megtaláltam, az első előforduló rekordot nézem, és esetleg továbblépek eggyel, csak azokig, amelyekre szükség van.

Műveletek

Létrehozás: (ld. feljebb) Első lépésben létrehozom az alapállományt, majd felépítem erre a láncszerkezetet.

Bővítés: Az elsődleges állomány szabályai mondják meg, hogy fizikailag hova kerül a bővítendő rekord. A bővítő rekordot be kell illesztenem a megfelelő láncba. pl. jön egy fehér. Elrakjuk valahova, majd beillesztjük.

I. Olyan másodlagos kulcsértékkel bővítek, amely már előfordult egyszer. A táblázat mérete nem változik, az új rekordot be kell illeszteni a láncba.
   1. Az eddigi lánc elejére illesztem be az új elemet: egyszerű, mert a táblából mindig meg tudom mondani, hogy hol van az első eleme a részláncnak. Ekkor fizikai és logikai sorrend eltér, a lánc nem azt a fizikai sorrendet tükrözi, ami az állományban van, így igen nagy a fejmozgás, lassú a feldolgozás. A bővítő rekord helye valószínűleg messze van az eredeti fejtől.
   2. A lánc végére illesztem be az új elemet: végig kell menni a láncon, hogy megkeressük a legutolsó elemét, de kisebb a távolság a lánc elemei között. A feldolgozás gyorsul, a karbantartás lassul.
   3. Szokás egy negyedik oszlopot bevezetni, ahol a lánc utolsó elemének címét tároljuk. A bejegyzések változnak, a sorok száma nem.

II. Olyan szituáció is előfordulhat, hogy eddig még nem szereplő másodlagos kulcsértékű rekorddal akarok bővíteni. Ekkor a táblázatba fölveszem ezt a másodlagos elemet és jelölöm az egyelemű láncot. Így a táblázat is bővülhet egy sorral. Ebbe a sorba az új másodlagos kulcsérték kerül, létrejön egy egyelemű részlánc az állományon belül.

Törlés: Logikai törlés van az elsődleges állományban. Multilista szinten használható a logikai törlés és realizálható a fizikai törlés is, de ekkor módosul a harmadik oszlop, és esetleg a második oszlop is. Ekkor a részlánc hossza csökken, első elem törlésénél módosul az első elem bejegyzése is.

Speciális eset, amikor egyelemű részláncból törlök egy elemet. Ilyenkor az adott másodlagos kulcs eltűnik a táblázatból és a multilista állományból, a táblázat adott sorát törölni kell.

Csere: Ha nem másodlagos kulcsot cserélek, nincs gond, az elsődleges állomány szintjén megáll a változtatás, nem jön föl a multilista szintre. Ha másodlagos kulcsot cserélek, akkor az adott rekord az egyik részláncból átkerül egy másik részláncba. Ezt szabályos törléssel és bővítéssel lehet megvalósítani.

Elérés: Az alapállomány és a multilista-állomány elérési lehetőségei együtt. A multilista-állomány elérési lehetőségei: egy-egy csoport közvetlen elérése, a csoportban szekvenciális elérést tesz lehetővé.

Újraszervezés: Az elsődleges állomány szerint újraszervezni az ottaniaknak megfelelően kell. Az egész multilistát újraszervezni akkor kell, ha túl sok a bővítő rekord.

Feldolgozás: A másodlagos kulcsérték száma kevés, ez a táblázat befér a tárba. Pl: lakóhely, a másodlagos kulcs. A táblázat alapján megmondható:

- Hány különböző másodlagos kulcsérték van?
- Egy adott másodlagos kulcsértékhez hány rekord van?
- Egy adott másodlagos kulcs értéke előfordul-e?
- Hol vannak azok a rekordok, amelyeknek a másodlagos kulcs értéke …?

Ezt az állományszerkezetet bármely másodlagos kulcs mezőjére fel tudom építeni, de akár egyszerre az összes mezőre is. Ekkor annyi mutató és annyi táblázat kell, amennyi másodlagos kulcs alapján láncolok.

Hibája, hogy önmagában csak egy másodlagos kulcs kezelését segíti, pedig a másodlagos kulcsértékeknél általában a kombinált lekérdezések a jellemzőek.

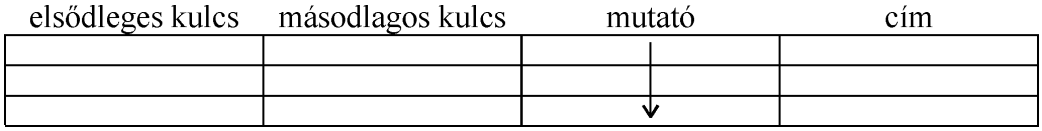
Hány 180 cm magas, kék szemű, felsőfokú állami angol nyelvvizsgával rendelkező ember van az egyetemen? Ezen felsorolásban lehet „vagy” és „és” kapcsolat is. A multilista a „vagy” kapcsolatot segíti. Itt az összes kékszemű embert, az összes 180 cm magas embert és az összes angol felsőfokúval rendelkező embert adni kell. Lehet, hogy valaki többször szerepel, de mindenki benne lesz a halmazban. Itt tehát három táblázatot kell használni.

„VAGY”-kapcsolat: A táblázatból kikeresem a láncokat, ezek adják az összes lehetőséget, legfeljebb többszörözik. Az összes engem érintő rekord benne van.

„ÉS”-kapcsolat: Nem támogatja. Ekkor kiválasztom az „ÉS”-kapcsolatban álló részláncokat, majd ezek közül a legrövidebbet, és ezen végigmenve megvizsgálom a többi tulajdonság teljesülését.

A multilista állományszerkezetet meg lehet valósítani korlátozott hosszú lánccal. Ezt a hardverhez köti, pl. csak olyan részláncokat alakítok ki, amelyek az egy cilinderen elhelyezkedő rekordokat láncolják össze. Így több részlánc alakul ki, de egy részlánc egyetlen fejmozgással elérhető, a feldolgozás könnyebbé válik.

Lehet keverni az indexelési és a láncolási technikát úgy, hogy a láncolást nem az alapállományra építem rá, hanem egy indextáblát építek fel az elsődleges kulcsra, és a láncolást az indextáblára viszem át. Teljes indexelés van, és a táblába beviszem a másodlagos kulcsokra vonatkozó láncolást. Ezt akárhány másodlagos kulcsra megtehetem, egy abszolút hibrid megoldást kapok. A tábla szerkezete és a karbantartás bonyolultabb, de a feldolgozás könnyű. A listafejeknek meg kell lenniük.



Invertált állomány

A definíciónak megfelelő megvalósítása: a tetszőleges, egyszerű alapállomány mellé felépül a másodlagos kulcsra egy teljes indextábla. Az indextábla első oszlopában vannak a másodlagos kulcsértékek. Az indextáblának annyi sora van, ahány különböző másodlagos kulcsérték előfordul az alapállományban, és a 2. oszlopban annyi bejegyzés, ahány rekord rendelkezik az adott sorbeli másodlagos kulcsértékkel.

Változó formátumú szekvenciális állomány. További indexek épülhetnek rá.

Műveletei

Létrehozás:

I. Indextáblázat létrehozása.
II. **Bővítünk** új rekordokkal, az állomány dönti el, hogy az fizikailag hova kerül.
   1. Ha a rekord új másodlagos kulcsértéket hoz be, akkor az indextábla a rendezettségnek megfelelően bővül új sorral, az első oszlopba kerül az új rekord másodlagos kulcsának értéke. A második oszlopnak az új rekord címe lesz az eleme.
   2. A bővítő rekord már szerepel az állományban, ekkor a második oszlopba kerül az új rekord címe. A címek sorrendje lényegtelen.

Törlés: Az alapállományban logikai törlés, az indextáblában fizikai törlés realizálható: az adott rekord címét kiemelem az indextáblából. Ha csak ez eleme volt a sornak a törlés előtt, akkor a teljes sort törölni kell az indextáblából.

Csere: Lehet, hogy nem érinti az indexszerkezetet. Másodlagos kulcsértéket cserélni szabályos törlés és bővítés útján szabad, ekkor az egyik sorból egy másik sorba kerül át a rekord.

Elérés: Másodlagos kulcs szerinti közvetlen elérés van. Ha egy adott másodlagos kulcsértékre vagyunk kíváncsiak, az indextáblában kell keresni. Ugyanúgy, mint a multilista állományszerkezet, ez sem támogatja a több másodlagos kulcs szerinti keresést.

Újraszervezés: Az elsődleges állomány újraszervezésével az indexeinket újra kell szervezni. Az indexünk önmagában nem indokolja az újraszervezést.

Az invertált állomány bitmátrixszal történő megvalósítása

A bitmátrixnak annyi oszlopa van, ahány rekord van az alapállományban. Az oszlopokat az elsődleges állomány rekordjai címével azonosítjuk (vagy az ennek előállításához szükséges információval). Az oszlopok sorrendje igazából lényegtelen, csak az egyértelműség a fontos.

Annyi sora van, ahány különböző másodlagos kulcsérték előfordul. A sorokat ezzel a másodlagos kulcsértékkel azonosítjuk, a másodlagos kulcsértékek sorrendje nem lényeges.

Feltöltésénél 1 kerül oda, ahol az illető rekord tartalmazza az adott másodlagos kulcsot, a többi helyre 0.

0	Budapest	1	0	0	1	1	…
0	Debrecen	0	0	0	0	0	…
1	Ózd	0	1	0	0	0	…
…	…	…	…	…	…	…	…

Egy sorban akárhány egyes lehet, de egy oszlopban csak egy.

Műveletek

Bővítés: Az, hogy a rekord hova kerül, az elsődleges állomány dönti el. A mátrixot is bővíteni kell egy új oszloppal. Mindegy, hogy az új oszlopot hol szerepeltetem. Egy bővítő rekord hozhat új másodlagos kulcsértéket is, ekkor a táblázat bővül egy, az új másodlagos kulcsértéket tartalmazó sorral. Ha nem hoz új másodlagos kulcsértéket a bővítő rekord, nem kerül új sor a mátrixba.
Törlés: Vagy figyelembe veszem az elsődleges állomány fizikai törlését, vagy logikai törlés van. A törölt rekord által meghatározott oszlopot kell eltüntetni a mátrixból. Lehet, hogy olyan másodlagos kulcsértéket törlök ki, amely egyedi volt. Ekkor az adott sor csupa nulla lesz, a sort törölni kell.
Csere: Vagy érint másodlagos kulcsértéket és befolyásolja a mátrixot, vagy sem. Szabályos törlésnek és bővítésnek felel meg. Az oszlopok száma nem változik, a sorok száma vagy nem változik, vagy csökken, vagy nő, vagy csökken és nő (a sorok darabszáma marad, de a sorok nem ugyanazok).
Feldolgozás: A feldolgozáshoz szükséges információ kiolvasható a mátrixból. Ha egy adott másodlagos kulcsértékkel rendelkező rekordhoz akarok hozzáférni, képezek egy oszlopvektort (v, maszkvektor), amely elemeinek a száma egyenlő a másodlagos kulcsértékek számával. A v azon elemeibe, amire kíváncsi vagyok, 1-t írok, máshova 0-t. A v-t, mint maszkot végigviszem a mátrixon (rendre az oszlopokra állítom) és vizsgálom: ha az adott helyen egyes van, akkor a rekord érdekel, különben nem. A hozzáférés így közvetlen hozzáférés lesz.
Keresés: Egyidejűleg bármennyi másodlagos kulcsra felépíthető ilyen mátrix, csak a mátrix sorainak száma változik, ezért egyidejűleg több másodlagos kulcsra is tudok keresni úgy, hogy a maszkvektor megfelelő értékeit egyesre állítom. A kulcsok között a logikai kapcsolat lehet VAGY vagy ÉS kapcsolat.
Újraszervezés: A bitmátrix önmagában újraszervezést nem igényel, csak az alapállomány.

A bitmátrix könnyen kezelhető, nagy állomány esetén is kicsi (ritka mátrix), könnyű implementálni és gyors. A műveletvégzés logikai, ezért nagyon gyors. Egyszerre több másodlagos kulcsértékkel rendelkező rekordhoz tudok hozzáférni (vagy-szerű kapcsolat is létezik). Igazi előnye az, hogy akárhány másodlagos kulcsértékre fel tudom építeni, különböző másodlagos kulcsokat tudok kezelni. A különböző másodlagos kulcsértékek száma jóval kisebb, mint a rekordok száma. A szeriális állományra rárakható egy közvetlen elérés.

Ezt egyetlen programozási nyelv sem kezeli explicit módon (a PL/1 tudja kezelni). Relációs adatbáziskezelőknél alapvető.

VSAM

Az IBM-nél a ’60-as években alakították ki. Komplex állományszerkezet. A PL/1 is ismeri.

Olyan állományszerkezetet, mely több állományszerkezetet egyesít magába, és ahol nem kell foglalkozni a fizikai szerkezettel, csak a logikaival.

A VSAM állományszerkezet tudja kezelni a szeriális, a direkt, az indexelt és az invertált állományszerkezetet.

Lényege, hogy a rekordok elhelyezése úgynevezett kontrollintervallumokban történik. A kontrollintervallum egy logikai tárolóterület, független a fizikai tárolástól. Az intervallumok méretét az állomány kialakítója adja meg (byte-ban), a fizikai tárolóterületre való leképezést a VSAM rendszer rendezi le, és ezért ezt az állományt könnyű más háttértárra átvinni. A rekordok elhelyezése az azonos méretű kontrollintervallumokban folytonosan történik. Minden egyes kontrollintervallum végén rendszerinformációk vannak, amelyekhez a felhasználó nem férhet hozzá. A kontrollintervallumok úgynevezett kontrollterületre szerveződnek, a VSAM állomány kontrollterületek együttese. Ezek szintén azonos méretűek, és a programozó szabja meg, hogy a kontrollterület hány kontrollintervallumot tartalmaz.

Az állomány kialakításakor kérhető, hogy minden egyes kontrollintervallum területének valahány százaléka üres legyen. Megadható az is, hogy a kontrollterületen n darab kontrollintervallum maradjon üresen.

Logikailag a kontrollterületek sorba állíthatók:

- az egy kontrollterületen belüli kontrollintervallumok sorrendje adott,
- az egy kontrollintervallumon belüli rekordok sorrendje adott.

Ez alapján a rekordoknak létezik logikai sorrendje. Mindez lemezen történik, a kontrollintervallumoknak és a kontrollterületeknek rendelkeznek kezdőcímmel. Számunkra ez azt jelenti, hogy bármely byte-nak van relatív címe (a byte-ok meg vannak sorszámozva 0-tól kezdődően), minden rekordhoz hozzárendelem az első byte-jának a sorszámát.

ESDS: Szeriális állományt realizál. Változó hosszúságú rekordokból állhat, ennek az összes jellemzőjével rendelkezik a VSAM állomány. A kontrollterületek folyamatosan töltődnek fel, nincs azonosító. Bővíteni a végén lehet, innentől kezdve szabályos szeriális állomány. Rárakható egy közvetlen elérési lehetőség is, így bármely rekord közvetlenül elérhető a relatív byte-címén keresztül.

RSDS: Direkt állomány. Lefoglalok annyi tárhelyet, amennyi kell, a rekordok fix hosszúságúak. Kialakítom a (fizikai) társzerkezetet. Bármely tárhelynek tudom a relatív byte-sorszámát, ezzel is tudok rá hivatkozni. Definíció szerint nem létezik hash-függvény, ettől eltekintve a direkt állomány sorszámjellegű azonosítóval rendelkezik. A feltöltés után vannak rekordokkal feltöltött helyek és üres lyukak. Csak direkt állományként használható.

KSDS: Indexelt szekvenciális állomány. Változó rekordformátumot tud kezelni. Az alapállomány szekvenciális, és erre épül rá az indexelt szerkezet. Nem teljes, többszintű indexelést alkalmaz, és a szekvenciális állomány karbantartását megoldja (bővítés, törlés, …). Egy kontrollterületen belül a karbantartás miatt nem töltünk fel minden intervallumot, maradnak szabad kontrollintervallumok: van n - x darab feltöltve és x darab kimarad. A szabadon maradók számát adom meg. Minden kontrollintervallumon belül hagyhatunk szabad területet, megmondható, hogy a kontrollintervallum byte-jainak legalább hány százaléka maradjon szabadon. Azonosító szerint növekvő sorrendben kerülnek be a rekordok a szerkezetbe (szekvenciálisan). Vigyáz arra, hogy a megfelelő százalék byte szabadon maradjon az alapállomány kialakításakor. Az indexelés a kontrollterületen történik: minden kontrollterület fölé felépít egy indexrekordot (adatszintű indexrekord), egy kontrollterülethez rendelődik hozzá. Ezeket összeláncolja mindkét irányban. Egy indexrekordhoz annyi bejegyzés van, ahány kontrollintervalluma van.

A nem teljes, többszintű indexszerkezetet a kontrollterületekre építi fel. Elsődleges szinten a kontrollintervallumokat indexeli. Ezek fölé kerül egy egyszintű indexrekord, amelynek annyi bejegyzése lesz, ahány kontrollintervallum van. Az indexrekord tartalmazza az adott kontrollintervallum relatív byte-címét és az adott kontrollintervallumban elhelyezett utolsó rekord azonosítóját, a maximális kulcsot vagy üres kontrollintervallum esetén speciális jelet. Az elsőrendű indexrekordokat kétirányban láncolja. Legfelül egyetlen indexrekord van, ez függ az elsődleges állományban elhelyezett rekordok számától és területétől.

A további indexrekordokat éppúgy a kontrolintervallumban, a kontrollterületen tárolja. E fölé újabb indexrekordokat építhet fel stb. Abban különbözik az elsődleges szintű indexrekordoktól, hogy ezek nincsenek összeláncolva. A második indexszinttől felfelé az intervallum- és területméret azonos!

Bármilyen művelethez az indexszerkezetre kell hivatkozom. Az indexszerkezetre más kontrollintervallum- és kontrollterület-méret adható meg, mint az alapállományra.

Műveletek

Bővítés: A rendszer az indexszerkezet segítségével megkeresi a bővítő rekord helyét. A legfelső indexrekordtól indul, és keresi az első olyan bejegyzést, amely nagyobb-egyenlő, mint a bővítő rekord kulcsa. Így jut el az elsődleges szintre, a kontrollintervallumig.

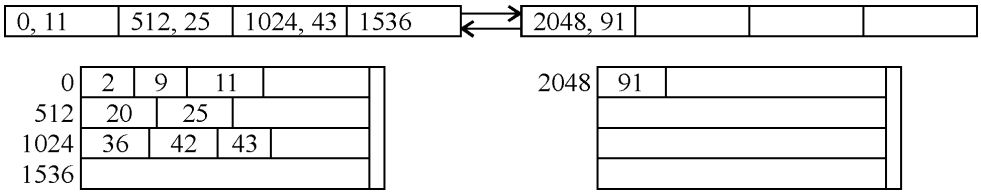
I. Ha az adott kontrollintervallumban van elég szabad byte, lineáris kereséssel megkeresi az elem helyét. Ha már benne volt a bővítő rekord, akkor hiba történt. Egyébként, ha van még

hely a kontrollintervallumban, a beszúrás helye mögött levő rekordokat jobbra tolja, és az új rekordot bepakolja. A változtatás az indexszerkezetet nem érinti, csak ha a legnagyobb rekord után bővítek.

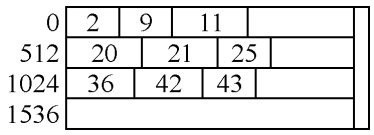
II. Ha nincs elegendő szabad byte a megfelelő kontrollintervallumban:
   1. Ha van szabad kontrollintervallum a kontrollterületen, akkor kontrollintervallum-felezést hajtunk végre.
      A. A teli intervallum első része maradjon a helyén, a második részét vigyük át az első üres kontrollintervallumba. Tegyük ezt úgy, hogy az elfoglalt terület aránya nagyjából megfeleződjön.
      B. Az indexszerkezetet át kell alakítani a módosításnak megfelelően. Az index a logikai sorrendet fogja tükrözni, szétválik a logikai és a fizikai sorrend. Át kell szervezni az indexbejegyzéseket, ezek a kontrollintervallumok logikai sorrendjét fogják tükrözni. A változás átgyűrűzhet a teljes indexszerkezeten.
      C. A keletkezett helyre elvégezhető a beszúrás.
   2. Ha a kontrollintervallum-felezés nem valósítható meg (mert nincs szabad kontrollintervallum), kontrollterület-felezést hajt végre.
      A. Lefoglal egy új kontrollterületet, és a teli kontrollterület (logikailag utolsó) felét átviszi az új kontrollterületre, a kontrollintervallumok szekvencialitása megmarad.
      B. Az indexszerkezet átírja: aktualizálja a bejegyzéseket, átalakítja a láncolást a megfelelő módon, hogy az a logikai sorrendet tükrözze.
      C. Ezután következik kontrollintervallumba való beszúrás, módosul az indexszerkezet.

Bővítés speciális esete, amikor az állomány legutolsó kontrollterületének legutolsó kontrollintervallumán még van hely, akkor ott történik a bővítés (ha fér). Ma már lehet nagyobb kontrollterületet megadni menet közben is, sok bővítés esetén, így nem válik el a fizikai és a logikai sorrend. Általában megadnak egy alapintervallum- és területméretet, de ha területfelezésre kerül sor, meg lehet adni egy növekedési százalékot, és ezt az indexbe is be lehet állítani.

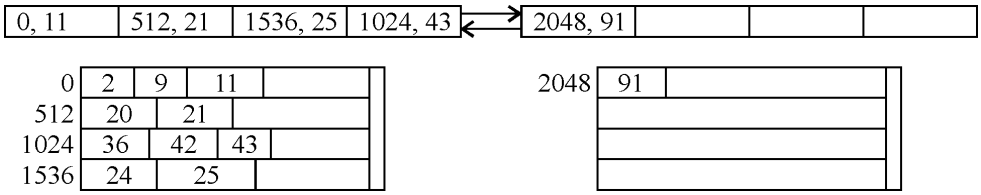
Pl. Legyen az indexrekord és a két kontrollterület a következő:



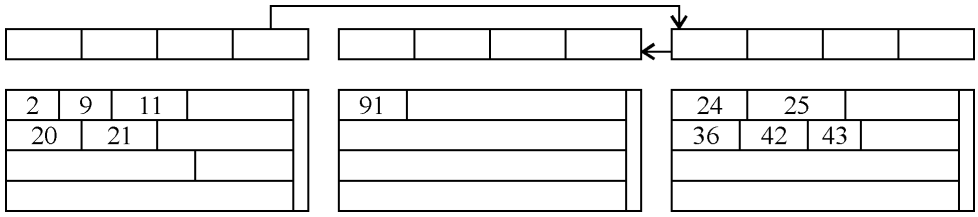
A 21-el akarok bővíteni, megkeresem a helyét. Ha megvan, a szabad helyre tolom át a mögötte levőket, és a 21-t beszúrom. Ha már benne volt a táblában, hiba történt.



Ugyanezt játszanám el a 24-gyel, de a kontrollintervallumban már nincs elég szabad hely a számára, kontrollintervallum-felezést kell végrehajtani, és módosulnak az indexrekord-bejegyzések is a logikai sorrendnek megfelelően.



Ha a 44-t akarom beszúrni, kontrollterület-felezésre van szükség, és az indexrekordok láncolásával megmarad a szekvencialitás.



Törlés: A KSDS állományban van fizikai törlés is, máshol csak logikai. Megkeresem a törlendő rekordot. Ha megtaláltam, akkor a mögötte lévő rekordokat előremozdítom, és gyűjtöm a felszabaduló byte-okat. A fizikai törlés egy esetben érintheti az indexszerkezetet: ha a kontrollintervallum utolsó rekordját kell törölni, ekkor módosul az indexrekord és a rendszerbejegyzés. A módosítás az egész indexrendszeren végigpöröghet.
Csere: Azonosítót (elsődleges kulcsot) nem cserélünk, ezért a csere az indexszerkezetet sosem módosítja.

I. Ha a cserélendő rekord hossza megegyezik az új rekord hosszával, akkor nincs probléma.
II. Ha nagyobb, akkor az üresen maradt helyre ráhúzzuk a mögötte levő rekordokat.
III. Ha kisebb, a szabadon maradt byte-okat használhatom fel (ld. bővítés).

Keresés: Mindig nagyobb-egyenlőre keres.
Feldolgozás:

I. A fenti keresési szisztémával közvetlen elérést tudunk realizálni, nagyobb-egyenlőre végignézi.
II. Szekvenciális alapállományról lévén szó, egy kontrollintervallumon belül szekvenciális elérés van. Ehhez az elsődleges indexrekordokat használom: növekvő vagy csökkenő sorrendben dolgozhatom fel, anélkül, hogy átszervezném (mert kétirányban van láncolva).
III. Az indexszerkezetet figyelembe véve közvetlen módon. Csak kontrollintervallumot tudja közvetlen elérni.

Újraszervezés: Akkor kell, amikor a kontrollterület-osztások felszaporodnak, de a megadott paraméterek is lehetnek rosszak. Célja az, hogy a fizikai és a logikai sorrend közelebb kerüljön egymáshoz. Rendszerfunkció.

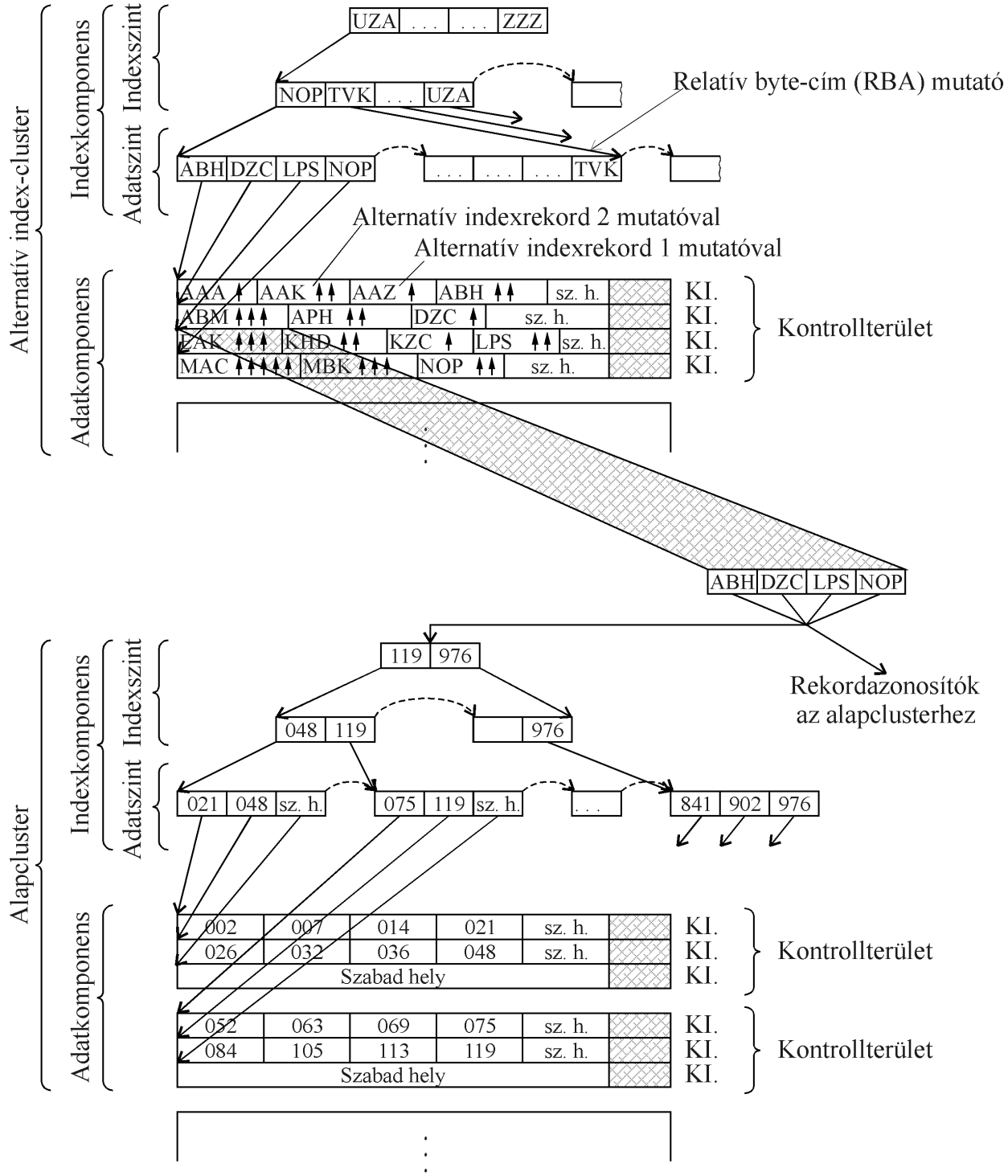
Alternatív indexek

A VSAM a KSDS állományokra automatikusan felépített elsődleges indexen kívül lehetőséget ad arra, hogy minden KSDS és ESDS állományra további – egy vagy több –, ún. alternatív indexet építsen fel. Az alternatív index az előzőekben említett VSAM állományok (KSDS, ESDS) logikai rekordjainak bármely olyan adatmezőjére felépíthető, mellyel külön hozzáférést kívánunk biztosítani ugyanazon állomány rekordjaihoz.

Az alternatív indexek tehát lehetővé teszik, hogy egy VSAM állományt (RRDS-t nem) több szempont szerint is feldolgozhassunk, illetve annak logikai rekordjait különböző utakon elérjük anélkül, hogy több másolatot tartanánk ugyanarról az adatállományról. Például egy dolgozói törzsállományt, amit bérelszámolásnál használunk, a dolgozó száma mint rekordazonosító szerint KSDS szerkezetűre alakítunk ki, de ha a dolgozói törzsrekord egyéb adatmezői szerint is fel akarjuk dolgozni az állományt, mint például a dolgozó neve, születési éve, stb., akkor ezekre a másodlagos kulcsokra alternatív indexeket építhetünk. Ezen alternatív indexek mindegyikét különböző felhasználói programok használhatják.

Egy alternatív indexnek egy ESDS állományban ugyanolyan rendező (rendszerező) funkciója lehet, mint egy KSDS file-ban az elsődleges indexnek. Ezt a gondolatmenetet folytatva látni kell, hogy egy ESDS állomány rekordazonosítójára felépített alternatív index indexelt szeriális struktúrát hoz létre, mely bizonyos esetekben sokkal hatékonyabban felhasználható, mint egy indexelt

szekvenciális file (lásd indexelt szeriális állomány). Tehát ez azért nagy jelentőségű, mert nemcsak az indexelt szekvenciális szerkezetű file-ok rendelkeznek szoftvertámogatással, hanem az indexelt szeriális szerkezetű file-ok is, a VSAM ESDS állomány rekordazonosítójára felépített alternatív indexen keresztül.



Blokkolni nem lehet. Ha a kontrollintervallum mérete olyan, hogy egy olyan rekord nem fér bele, a rendszer automatikusan szegmentál. Fix és változó rekordformátumot tud kezelni, a többszempontú lekérdezést a VSAM nem támogatja.

B^+-fa

Adatbázis-kezelő rendszerek mögött B^+-fák indexelési módszere áll. A B^+-fa a B-fák olyan módosított változata, melyet állományok indexelésére használnak.

Az alapállomány szekvenciális és szeriális is lehet (adatszint). Erre épül fel egy teljes indexelésű, elsődleges indexszint. A B^+-fák adott méretű lapokból állnak, a lapokon indexek, azaz a kulcsok szerepelnek Az indexek elsődleges indexszinten lapokon vannak elhelyezve. Többszintű teljes indexelést tudunk megvalósítani (elsődleges kulcsra), a további indexeknél a lapméret lehet más.

Előírhatjuk, hogy létrehozáskor a lapokon legyenek üres helyek minden szinten. Manapság az összeláncolás is jellemző tulajdonság.

Szerkezetükben különböznek a levél- és a közbenső lapok.

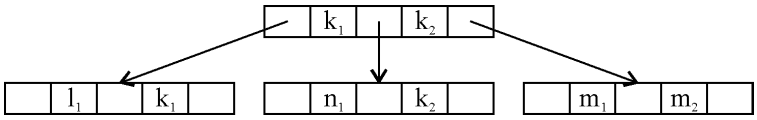
I. A közbenső (nem levéllapok) lapok szerkezete:

Ha a B^+-fa rendje p, akkor a p a maximális értéket jelöli. Itt a rend a korábbinak a kétszerese. A következő információk vannak elhelyezve a lapokon:

$$\boxed{p_1\ k_1\ p_2\ k_2\ \ldots\ k_{q-1}\ p_q} \qquad (q \le p)$$

- $p_{i:}$ egy mutató, amely bármely másik B^+-fa-beli lapra mutathat (famutató),
- k_j: kulcsértékek, $k_1 < k_2 < \ldots < k_{q-1.}$
- Minden p_i mutató által megcímzett lap esetén minden kulcsra igaz:
- $k_{i-1} < x \le k_i$ $(1 < i < q$, x az adott lapon egy tetszőleges kulcsérték)
- $x \le k_i$ $(i = 1)$
- $k_{i-1} < x$ $(i = q)$

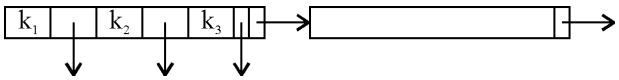
A lapon maximum a B^+-fa rendje darab mutató helyezkedhet el és eggyel kevesebb kulcs. A gyökérlapon kívül minden lap legalább [p/2] mutatót tartalmaz. A gyökérlap legalább 2 mutatót tartalmaz.



II. A levéllapok szerkezete:

$$\boxed{k_1\ p_1\ k_2\ p_2\ \ldots\ k_{q-1}\ p_{q-1}\ p_q} \qquad (q \le p)$$

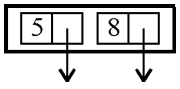
- k_i : kulcsok.
- p_i: $1 \le i < q$ egy olyan mutató, amely az elsődleges állomány azon rekordját címezi, amelyiknek a kulcsa k_i. Ezek az adatmutatók kimutatnak a fából az elsődleges állományba. p_q a fában a tőle közvetlenül jobbra elhelyezkedő levéllapot címezi.



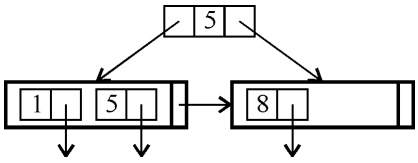
Teljes indexelés van, minden rekord címezve van. A levéllapok azonos szinten helyezkednek el.

Egyes rendszerekben a levéllapok és a nem levéllapok rendje eltérhet, a levéllapé nagyobb lehet (bevezetnek egy p_0 mutatót a láncolás miatt), ez gyorsítja a kezelést. A B^+-fa kialakításánál a gyökérlap levél és közbenső elem is.

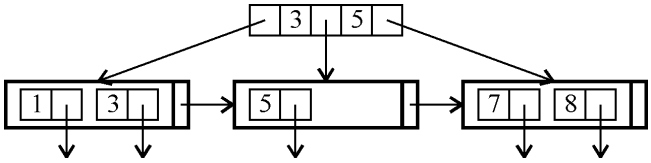
Beszúrás B^+-fába: Legyenek a beszúrandó elemek a 8, 15, 1, 7, 3, 12, 9, 6, és legyen p = 3 a B^+-fa rendje. Lefoglaljuk az első lapot, rápakoljuk az első kulcsot és beállítjuk a mutatót. A következő elemmel ugyanezt tesszük. A jobb szélen levő levéllap mutatója mindig NIL.



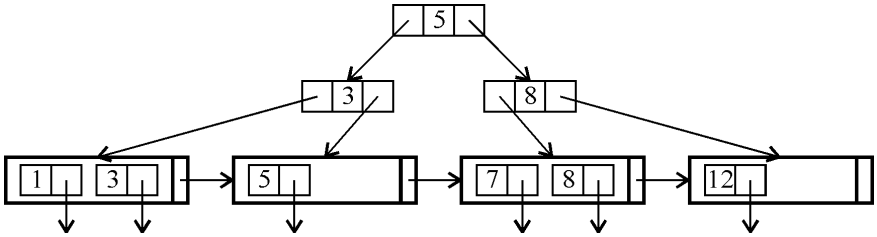
Betelt a lap, az 1 beszúrásánál már laposztás kell:



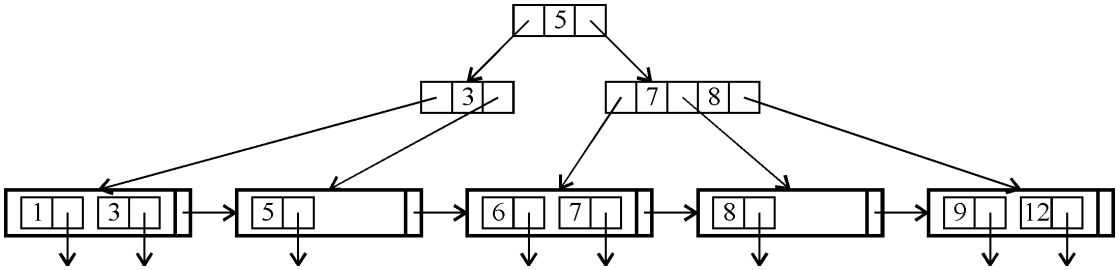
A 3 és a 7 beszúrása után a B^+-fa:



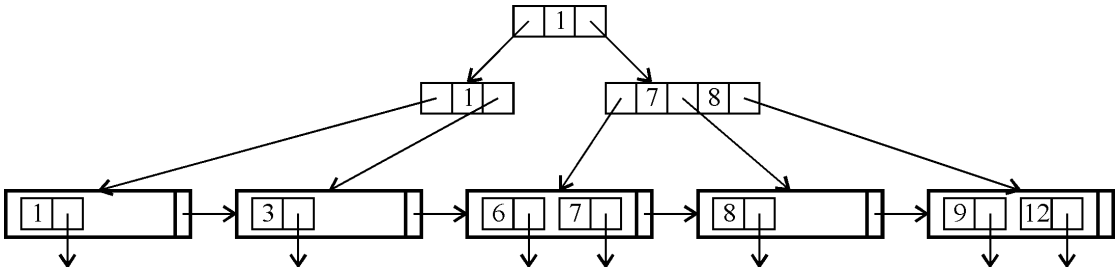
A 12 beszúrása után:



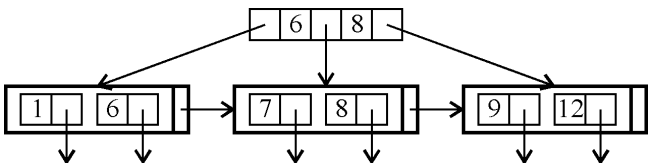
A 6 beszúrása után:



Törlés B^+-fából:
Töröljük 5-öt a fából!



Ebben a fában a 9-et a fa megváltoztatása nélkül törölhetnénk, a 7-et csak indexváltoztatás árán, a 3 törlésekor pedig, mivel balról nem tudok elemet átvenni, két levéllapot kell összevonni, ami végigpörög a fán:



A törlést a B^+-fából alkalmazhatom nem teljes indexelés esetén is, ha az alapállománynál nem a rekordokat, hanem az alapállomány blokkjait indexelem. Blokkolt szekvenciális állománynál nem teljes indexelésre alkalmazhatom a B^+-fát. Az elsődleges állomány fölé felépíthetek egy invertált indextáblát és azt indexelem. Csak a teljes indexelésnek van értelme.
Létezik a B^+-fának olyan megvalósítása, hogy ahogyan nő a B+-fa szintjeinek a száma, úgy nő a rend is. A B^+-fa módosításai:

I. Az egy lapon elhelyezett elemek számát másképp szabályozzák. A lapok jobb kitöltésére szolgál, minden lap minimum 2/3 rendig ki van töltve (B^*-fa).
II. A kitöltöttséget (50-100%) a felhasználó adhatja meg.