Problémaosztályok, algoritmusok

Rendezés, keresés

Rendezési és keresési algoritmusok

- Általános kérdések:
  - Hogyan készíthetők jó algoritmusok ?
  - Hogyan tökéletesíthetők az algoritmusok ?
  - Algoritmusok hatékonyságának vizsgálata matematikai módszerekkel.
  - Ésszerű választás algoritmusok között …
  - Milyen értelemben bizonyulhat egy algoritmus a „legjobbnak”…
  - Hogyan használhatók hatékonyan a külső tárak az adatok kezelésére?

- Irodalom:
  - D.E.Knuth: A számítógép-programozás művészete 3.
  - Rónyi, Ivanyos, Szabó: Algoritmusok

Rendezési és keresési algoritmusok

- Rendezésnek nevezzük azt a folyamatot, amikor egy halmaz elemeit valamilyen szabály szerint sorba állítjuk. A rendezés meggyorsítja az elemek későbbi keresését.

- *Növekvő, monoton növekvő, csökkenő, monoton csökkenő...*
  (telefonkönyv, tartalomjegyzékek, tárgymutató, könyvtárak, áruházak, stb.)

- A rendezés célja:
  - Legnagyobb, legkisebb elem, átlag alattiak, stb…
  - A keresés megkönnyítése
  - Könnyebb összefüzés

Rendezési és keresési algoritmusok

- Rendezési eljárások: az algoritmusok sokfélesége …

- Az algoritmus megválasztása függ a feldolgozandó adatok struktúrájától...
  - tömbök rendezése, (belső rendezés)
  - file-ok rendezése. (külső rendezés).

- Példa:
  - kártyák tömbszerű strukturálása: ha a kártyákat kiterítjük az asztalon: mindegyik lap látható és közvetlenül elérhető.
  - a kártyák file-szerű strukturálása: kártyacsomag, csak a csomagok tetején lévő lapok láthatók.

Rendezési és keresési algoritmusok

- A rendezés legfontosabb alkalmazásai közül néhány:
  - Az „**Összegyűjtési” probléma** (ugyanolyan azonosítójú tételek)
  - Ha két vagy több állományt azonos módon rendeztünk, akkor egyetlen végigolvasással megtalálhatjuk az **összetartozó adatokat**.

„De hát ennyi idő alatt nem tudja az összes rendszámot átnézni. - tiltakozott Drake. Nem is kell, Paul. Csak sorrendbe állítjuk őket és megkeressük az egyformákat.” - PERRY MASON , 1951.

  - A rendezés a **keresés egyik segédeszköze**. (Pl. szótár, telefonkönyv)

Tömbök rendezése.

- Táblázat - legegyszerûbb formája: 1D tömbök. Ez egy adatbázis legprimitívebb modellje. Tovább egyszerűsítünk: a tömbjeink egyszerű típusú adatokat tartalmazzanak!

- Altalában: a táblázat egy eleme egy összetett adattípus (struktúra v. rekord), pl. egy könyvtári katalógus elemei a katalógus cédulák.

- Az adatbázis elemeire ún. **kulcsok** (*key*-k) szerint hivatkozunk, kulcsok szerint keresünk, rendezünk stb. Kulcsok pl. a könyvtári katalógusban: szerző neve, mű címe, a könyv leltári száma, ISBN szám stb.

- A mi primitív modellünkben egyszerű adatokról van szó: a kulcs maga az adat. Pl. egész számok tömbjében a kulcsok maguk a számok.

Tömbök rendezése.

- A rendezőmódszerektől megkívánt fontosabb **követelmények**:
  - gazdaságos tárkihasználás ->
    az elemek átrendezését “saját helyükön” kell végrehajtani.
  - hatékonyság, azaz időtakarékosság:
    mérése: összehasonlítások száma, tételmozgások száma

(Jó rendezési algoritmus: N log N -el arányos idő …)

- A rendezőmódszerek, amelyek eredeti helyükön rendezik át a tömböket:
  - Rendezés beszúrással,
  - Rendezés kiválasztással,
  - Rendezés cserével

Rendezés beszúrással

- Legyen az adatok száma n. A feladat az adatok növekvő sorrendbe rendezése.
- Ezt a módszert a kártyajátékosok is alkalmazzák.
- A tételeket (kártyákat) elvben két részre osztjuk:
  - az **a1, ... ai-1** fogadó sorozatra és az **ai, ... an** leadó sorozatra.
- Minden lépésben, i=2-től egyesével n-ig, kiemeljük a leadó sorozat első elemét és áttesszük a fogadó sorozatba, beszúrva őt a megfelelő helyre.
  - *A megfelelő hely kiválasztása: lineáris kereséssel.*

Rendezés beszúrással

- Példa:
- Kiindulási adatok: **66, 33, 45, 22**
  (kulcsok)
- i = 2 66, **33**, 45, 22
  - 33 beszúrása a {66} fogadó sorozatba
- i = 3 33, 66, **45**, 22
  - 45 beszúrása a {33, 66} fogadó sorozatba
- i = 4 33, 45, 66, **22**
  - 22 beszúrása a {33, 45, 66} fogadó sorozatba.
- A rendezett adatok: 22, 33, 45, 66

Rendezés beszúrással

- i = 4 33, 45, 66, **22**
  - 22 beszúrása a {33, 45, 66} fogadó sorozatba

- *A beszúrás lépései: (i = 4-nél)*

  *33, 45, **22,** 66*

  *33, **22,** 45, 66*

  ***22,** 33, 45, 66*

- *A megfelelő hely megállapításához célravezető az egymás utáni összehasonlítások és mozgatások alkalmazása, a beszúrandó elem “lerostálása”.*

Rendezés beszúrással

- Az összehasonlítás alapú rendező módszerek n elem rendezésekor log2 (n!) összehasonlítást és n*n -el arányos számú mozgatást igényel.

- Az összköltség tehát igen nagy lesz.

Rendezés beszúrással

A közvetlen beszúró-algoritmus egyszerűen javítható:

- az **a1, ... ai-1** fogadó sorozat, ahová az új tételt beszúrjuk, már rendezett. Így a beszúrás helyének megállapításához használhatunk gyorsabb módszert is. Kézenfekvő a **bináris keresés** alkalmazása, amelyben a fogadó sorozat felezését addig ismételjük, amíg a beszúrás helyéhez nem érkezünk.
- A módosított rendezőalgoritmust **bináris rendezésnek** nevezzük.

A digitális számítógépekre a beszúró rendezés nem túl jó módszer (még a bináris beszúrással sem), ugyanis egy tétel beszúrása, amely akár egy egész sor tételnek egyetlen hellyel való odébb tolását váltja ki, **gazdaságtalan** megoldás.

Rendezés közvetlen kiválasztással

- Legyen az adatok száma n. A feladat az adatok növekvő sorrendbe rendezése.

- A módszer elve a következő:
  1. Válasszuk ki a legkisebb elemet.
  2. Cseréljük ki az első elemmel
  3. Ismételjük meg ezeket a műveleteket a megmaradó n-1 majd n-2 tételre, és így tovább addig, amíg egyetlen tétel - a legnagyobbik marad.

- Összehasonlítások száma : 1/2 * n * (n - 1)

Rendezés közvetlen kiválasztással

Példa: Növekvő sorrendbe rendezés

- Kiindulási adatok: **66, 45, 33, 22**
  (kulcsok)
- i = 1 *66, 45, 33, 22*
  - legkisebb elem: 22, csere !
- i = 2 **22**, *45, 33, 66*
  - legkisebb elem: 33, csere !
- i = 3 **22, 33,** *45, 66*
  - legkisebb elem: 45, csere !

Rendezés közvetlen kiválasztással

- Ez a módszer bizonyos értelemben a **közvetlen beszúrás fordítottja**.
- A közvetlen beszúrás ugyanis minden lépésben a leadó sorozat egy következő tételét veszi, és a fogadó sorozat összes tételét áttekintve találja meg a beszúrás helyét,

- míg a közvetlen kiválasztás a leadó sorozat összes tételét áttekintve találja meg a legkisebbet, amit a fogadó sorozat egy soronkövetkező helyére tesz.

- A közvetlen kiválasztás elemzése (a képletek mellőzésével) azt mutatja, hogy az **előnyösebb a közvetlen beszúrásnál**, jóllehet, ha a kulcsok kiinduláskor már rendezettek vagy majdnem rendezettek, a közvetlen beszúrás valamivel gyorsabb.

Cserélő rendezések

- Az előbbi két módszert is tekinthetnénk cserélő rendezéseknek. Most olyan rendezésekkel foglalkozunk, amelyekben
- **két tétel felcserélése a legjellemzőbb művelet az eljárás során**, vagyis alkalmas tétel-párok összehasonlítása és felcserélése során jutunk el az összes elem elrendezéséhez.

- Legyen az adatok száma n. A feladat az adatok növekvő sorrendbe rendezése.

- A rögzített i-dik elemet összehasonlítjuk a nála nagyobb indexű elemekkel, s ha a vizsgált elem nagyobb az i-dik elemnél, az elemeket felcseréljük.

Cserélő rendezések

- Példa:

Kiindulási adatok:		**66,**	**33,**	**45,**	**22**	
i=1	j=2	**66,**	**33**,	45,	22	csere !
	j=3	**33,**	66,	**45**,	22	
	j=4	**33,**	66,	45,	**22**	csere !
i=2	j=3	22,	**66,**	**45,**	33	csere !
	j=4	22,	**45,**	66,	**33**	csere !
i=3	j=4	22,	33,	**66,**	45	csere !

Cserélő rendezések, algoritmus

Függvény rendez_csere

paraméterek: n: egész, a rendezendő adatok száma

adat vektor, a rendezendő adatok

lokális változók: i, j egész tip.

```
ciklus i=1  kezdőértéktől i <= n - 1 végértékig
        ciklus j=i+1  kezdőértéktől  j <= n végértékig
                ha  adat [j] < adat [i]
                        csere (adat [j],  adat [i])
        ciklus vége   //j
 ciklus vége //i
```

rendez_csere fv. vége

Cserélő rendezések - Buborék rendezés

- Ha a tömböt függőlegesen nézzük, a tömböt víztartálynak, a tételeket pedig “kulcsoknak” megfelelő buborékoknak gondolhatjuk. A rendezés során a buborékok felemelkednek ...

- A működés szemléltetése:
- Példa:

Kiindulási adatok:	**66,**	**33,**	**45,**	**22**	
1. menet	**66,**	**33,**	45,	22	66, 33 csere !
	33,	**66,**	**45**,	22	66, 45 csere !
	33,	45,	**66,**	**22**	66, 22 csere !
	33,	**45,**	22,	66	

Cserélő rendezések - Buborék rendezés

- A működés szemléltetése:
- 2. menet **33, 45,** 22, 66

  33, **45, 22,** 66

  45, 22 csere !

  33, 22, **45, 66**

- **3. menet** **33, 22,** 45, 66

  33, 22 csere !

  22, **33, 45,** 66

  22, 33, **45, 66**

Cserélő rendezések - Buborék rendezés

- Algoritmus:

Függvény rendez_buborek

paraméterek:

n: egész, a rendezendő adatok száma

adat vektor, a rendezendő adatok

lokális változók:

i, j egész tip. i a menetek száma

j a lépések száma

Cserélő rendezések - Buborék rendezés

- Algoritmus:

....

```
cilkus i=1  kezdőértéktől i <= n - 1 végértékig  //menet
        ciklus j=1  kezdőértéktől  j <= n - 1 végértékig //lépés
                ha  adat [j] > adat [j+1]
                        csere (adat [j],  adat [j+1])
        ciklus vége   //j
 ciklus vége //i

rendez_buborek fv. vége
```

Javított buborék rendezés:

- Minden menetben figyeljük, hogy történt-e csere. Az első olyan menetnél, ahol nem történt csere, az algoritmus befejezhető.

- Az i-dik menet után a hátsó i darab elem már a helyén van, fölösleges vizsgálni. A belső ciklus (n-1) helyett (n-i)-ig megy.

Javított buborék rendezés:

Függvény rendez_buborek_javitott

paraméterek: n, adat

lokális változók: i, j : i a menetek, j a lépések száma

voltcsere: egész tip, 0, ha nem volt csere...

ciklus i=1 kezdőértéktől i <= n - 1 végértékig, 1 lépésközzel //menet

volt**csere = 0**

ciklus j=1 kezdőértéktől **j <= n - i végértékig** 1 lépésközzel

//lépés

ha adat [j] > adat [j+1]

voltcsere = voltcsere + 1

csere (adat [j], adat [j+1])

ciklus vége //j

ha voltcsere == 0

kiugrás az i ciklusból //Ezt lehetne szebben is !!!

ciklus vége //i

rendez_buborek_javitott fv. vége

Javított buborék rendezés:

Kiindulási adatok:	**66,**	**33,**	**45,**	**22**	**n = 4**
1. menet j = 1	**66,**	**33,**	**45,**	**22**	**66, 33 csere !**
j = 2	**33,**	**66,**	**45,**	**22**	**66, 45 csere !**
j = 3	**33,**	**45,**	**66,**	**22**	**66, 22 csere !**
2. menet j = 1	**33,**	**45,**	**22,**	**66**	
j = 2	**33,**	**45,**	**22,**	**66**	**45, 22 csere !**
3. menet j = 1	**33,**	**22,**	**45,**	**66**	**33, 22 csere !**
Rendezett tömb:	**22,**	**33,**	**45,**	**66**	

Gyorsrendezés (Quicksort)

- **Felosztva cserélő** eljárásnak is nevezik.
- **Alapelv**: nagyobb hatékonyságot lehet elérni az egymástól minél **távolabbi** elemek cseréjével.
- **Az eljárás lényege**: az első elemet (vagy egy tetszőlegesen kiválasztott elemet) a rendezés kulcsának tekintjük abban az értelemben, hogy a tömböt két részre bontjuk úgy, hogy az első felében ne legyenek a kulcsnál nagyobb, a második felében a kulcsnál kisebb elemek.
- A két csoportot külön rendezhetjük.
- A kapott két rendezendő részre ismét alkalmazzuk a leírtakat, tehát egy **rekurzív algoritmussal** van dolgunk. A rekurzió akkor fejeződik be, amikor a rendezendő rész egy elemből áll.

Gyorsrendezés (Quicksort)

- A módszert C. A. R. Hoare találta ki 1960-ban.
- Az alapötlet egy szép példa az ***oszd meg és uralkodj*** elv alkalmazására.

Gyorsrendezés (Quicksort)

- **A gyorsrendezés algoritmusa:**
  - Kiválasztjuk a tömb egy tetszőleges elemét, legyen ez pl. a tömb középső eleme, értéke x
  - Balról keressük az első, x-nél nagyobb elemet, jobbról az első, x-nél kisebb elemet és ezeket felcseréljük.
  - A keresés - csere műveletsort addig folytatjuk, amíg a kétirányú keresés nem találkozik
  - EREDMÉNY: a tömb két részre osztott, bal oldalon az x-nél kisebb, jobb oldalon az x-nél nagyobb elemek állnak.
  - Az eddigi lépéseket most külön-külön, mind a két részre alkalmazzuk, a kapott részeket osztjuk tovább mindaddig, amíg mindegyik rész már csak egyetlen elemből áll.

- Az eljárás minden olyan programozási nyelven nagyon röviden felírható, amely a **rekurziót** támogatja.

Gyorsrendezés (Quicksort)

- A gyorsrendezés várható költsége: n log(n)
- A gyorsrendezés a várható futási időt tekintve az egyik leggyorsabb ismert módszer. Gyakorlati viselkedése alapján a legjobbnak tartott összehasonlítás alapú módszer memóriában lévő állományok (lista, tömb) rendezésére.

Külső tárak tartalmának rendezése

- Rendezés a memóriában: az adatok mozgatásához szükséges idő nem nagy, az összehasonlítások idejével összemérhető nagyságrendű: néhány mikroszekundum…

- Mágneslemezek elérési ideje: ezredmp…
  - A külső tárakon levő adatok kezelésénél ezért komoly figyelmet kell fordítanunk az adat elérés/mozgatás költségeire.

  - A külső tárak másik fontos vonása, hogy egy írás/olvasás során nem csak egy szót mozgatunk, hanem egy nagyobb területet.

- Más módszerek kellenek, pl. összefésüléses rendezés külső tárakban.

Keresési eljárások

- Keresés fileban: ld. később...

- Keresés tömbökben: adott az **N** elemű tömb és egy megadott **K** érték. A keresés feladata megtalálni azt az elemet, amely a megadott értékkel azonos.

- A keresés befejezésekor két lehetőség fordul elő:
  - a keresés sikeres volt, azaz megtaláltuk az elemet, amely a megadott értékkel azonos
  - a keresés sikertelen volt, nem találtunk a tömbben olyan elemet, amely a megadott értékű lett volna.

Keresési eljárások

- **Szekvenciális keresés**
  - **E**z a legegyszerűbb keresés és egyben a leglassúbb is.
  - Ha a tömb elemei nem rendezettek, akkor nincs más lehetőség, mint az elsőtől az utolsóig valamennyi elemet összehasonlítjuk a keresőkulccsal. Az algoritmus így **N** lépésből áll, csak akkor fejeződik be kevesebb lépésben, ha a keresett elemet megtaláljuk és tudjuk, hogy nincs két egyforma elem a tömbben.
  - Ha a tömb elemei rendezettek, pl. növekvő sorrendben, csak akkor van értelme a feladatnak, ha a keresett érték az első és az utolsó elem értéke közötti.
  - Sikertelen keresés esetén a műveletigény N-nel arányos.
  - Rendezett táblázatokban hatékonyabb keresési algoritmusokat is alkalmazhatunk.

Keresési eljárások

- **Szekvenciális (lineáris) keresés strázsával**

- **Strázsa: őrszem (*sentinel*)** - az ötlet: helyezzük el a táblázatban az utolsó érvényes elem mögé a keresendő elemet. Ez lesz a strázsa.

- Az egyszerű lineáris keresésnél mindig két kérdést tettünk fel a ciklusban: megtaláltuk-e a keresett elemet, illetve a táblázat végére értünk-e? A strázsa alkalmazásával elegendő csak azt vizsgáljuk, hogy megtaláltuk-e a keresendő elemet. Tehát a kereső ciklusban elegendő az első feltétel vizsgálni (megtaláltuk-e), így a hasonlítások számát felezhetjük. Ha a kereső ciklusból úgy lépünk ki, hogy az index változónk értéke az aktuális elemszám értékénél kisebb, akkor a keresendő elem benne volt a táblázatban. Ha ez nem igaz, akkor a strázsát találtuk meg, tehát a keresendő elem nem volt benne a táblázatban.

Keresési eljárások

- **Bináris (logaritmikus) keresés**

- Adott az N elemű a1, a2, . . . an tömb, melynek elemei rendezettek, pl. növekvő sorrendben. Keressük azt az elemet, amely azonos egy megadott K értékkel.

- A keresés lényege, hogy a K kulcsot összehasonlítjuk a középső elemmel, Ekkor vagy megtaláljuk az elemet, vagy megállapítjuk, hogy a tömb melyik felében érdemes folytatni a keresést, s ezzel a vizsgálandó elemek számát máris felére csökkentettük.

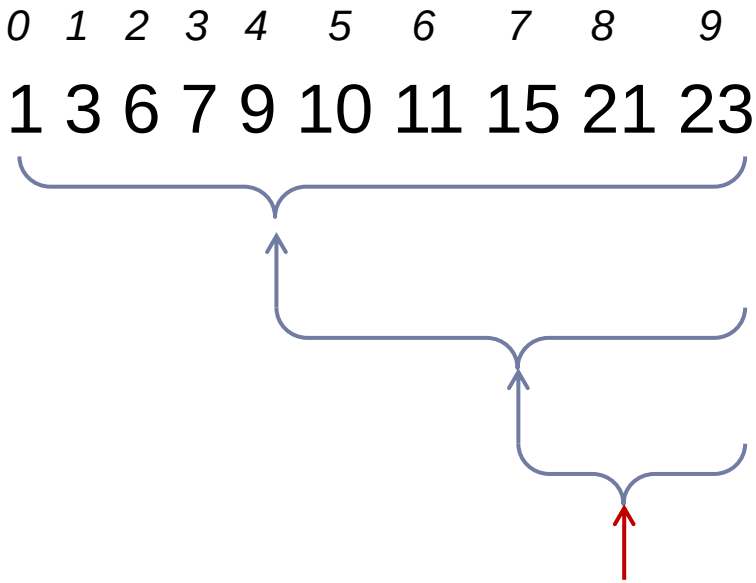
Bináris keresés

- Pszeudokód

```
Függvény binariskeres
        paraméterek: keres: egész
                        első, utolsó: egész
                        tömb: vektor
Ha elso==utolsó         //megállási feltétel
        return -1       //nincs meg
Különben
        középső=(elso+utolso)/2;
        Ha keres==tomb[kozepso]
                        return kozepso;
        kulonben ha keres<tomb[középso]
                        return binariskeres(keres, elso,kozepso,tomb)
        kulonben ha keres>tomb[középso]
                        return binariskeres(keres,kozepso,utolso,tomb)
```

Példa

- Tomb (10 elem):



- Program

binariskeres(21,0,9,tomb)

binariskeres(21,4,9,tomb)

binariskeres(21,7,9,tomb)

return 8;

Tömb rendezése, keresés, példa

```
void main (void)
{
  void adat_beolv (int [], int);
  void adat_kiir (int [], int);
  void adat_rendez (int [], int);
  int adat_keres (int [], int, int);
  int egesz_bevitel (int, int, char *);
  void csere (int *, int *);

  const max_sor    = 100;
  int adat [max_sor];

     ..........
```

Tömb rendezése, keresés, példa

```
void adat_rendez (int a[], int elemszam)
{
  /*  az  elemszam  db adattal feltöltött a  tömb
      elemeit növekvő sorrendbe rendezi
  */
  int i, j, pot;
  for (i=0; i < elemszam - 1; i++)
  {
    for (j = i+1; j < elemszam; j++ )
    {
      if ( a [j] < a [i] )
           {pot=a[j]; a[j]=a[i]; a[i]=pot;}//csere
    }
  }
```

Tömb rendezése, keresés, példa

```
int adat_keres (int a[], int elemszam, int k)
{
 //szekvenciális keresés növekvő sorrendben rendezett adatokon,
 //visszatérési érték a keresett adat indexe,
 //                vagy -1, ha nincs az elem a tömbben
   int i;
   if (k < a [0]  || k > a [elemszam -1])
      return -1;                  //k értéke kívülesik ...
   for (i=0;  i< elemszam; i++)
      if (a [i] == k)
        return (i);             // megvan az elem !
   return -1;                    // nem talált k értékű elemet
}
```

Hashing:

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

cim = hash_fv(kulcs);

Hashing…

- **Legegyszerűbb címgenerálás:**

  cim = kulcs % TABLAMERET;

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

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

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

Hashing …

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

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

Hashing ...

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

- Általában a státusz nyilvántartására használhatunk egy státusz jelzőt.