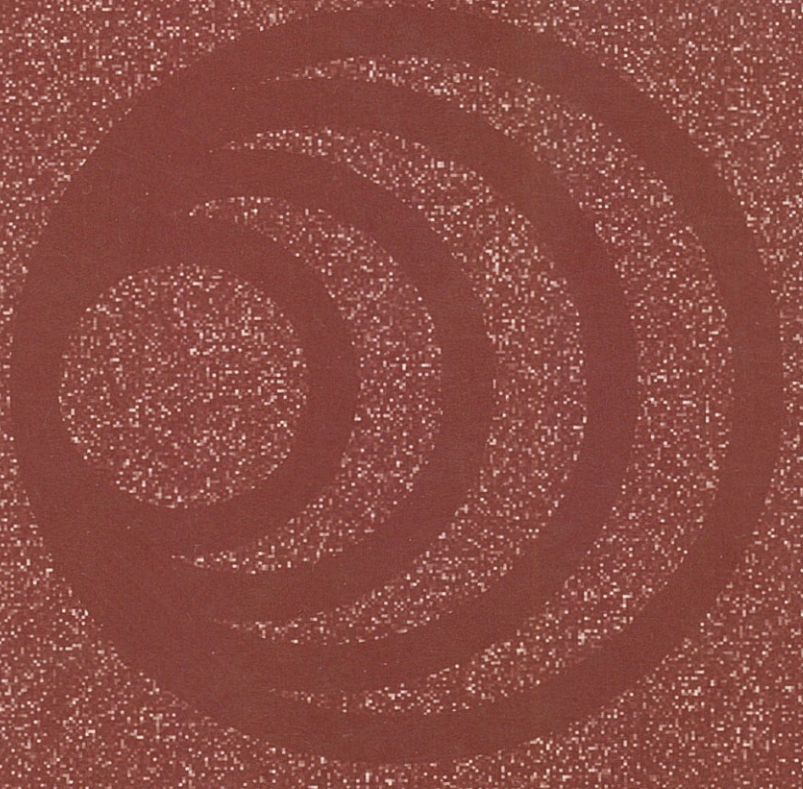
Angster Erzsébet

PROGRAMOZÁS TANKÖNYV II.

Strukturált tervezés Turbo Pascal



Angster Erzsébet

PROGRAMOZÁS TANKÖNYV II.

Strukturált tervezés

Turbo Pascal

Hatodik, javított kiadás, 2000

(javított kiadások: 2., 3. és 5., átdolgozott kiadás: 4.)

Szerkesztette: Angster Erzsébet

Kiadja: 4KÖR Bt.

ISBN 963 450 955 X Ö

ISBN 963 450 957 6 II.K.

Akadémiai Nyomda, Martonvásár
Felelős vezető: Reisenleitner Lajos

Előszó

A Programozás Tankönyv második kötete az első kötet egyenes folytatása. Fő témái az állománykezelés, memóriakezelés és az adatszerkezetek.

A két kötetben igyekeztem összefoglalni azokat az alapvető algoritmusokat, programozási fogásokat, melyek ismerete nem hiányozhat egyetlen programozó tudáskészletéből sem.

A tankönyv távoktatási célra is használható, az anyag fejezetről fejezetre haladva elsajátítható. Minden fejezet végén ellenőrző kérdéseket és feladatokat talál az Olvasó. Ajánlatos a kérdésekre szóban vagy írásban szabatosan válaszolni, a feladatokat pedig ténylegesen megoldani – nem elegendő azokat gondolatban „kipipálni". A feladatok egy részének megoldása megtalálható a kötet végén, a *„Megoldások"* fejezetben.

A könyvben használt strukturált programtervezési módszer a **Jackson** módszer egy helyenként leegyszerűsített, máshol kibővített változata.

A tankönyv programjai a Turbo Pascal, illetve a Borland Pascal 6.0 és 7.0 verziói alatt futtathatók. A forráskódok a könyv hátán megjelölt Internet címen megtalálhatók.

Szeretnék köszönetet mondani a SZÁMALK 1994/95-ös intenzív Számítástechnikai Programozó tanfolyam hallgatóinak az áldozatos hibavadászásért, családomnak a türelemért, szüleimnek pedig a pécsi nyugodt alkotó hetekért.

Kedves Olvasó! Kérem, hogy a könyvvel kapcsolatos észrevételeit, tapasztalatait az alábbi elektronikus postacímre írja meg:

1998. január

angster@okk.szamalk.hu

Tartalomjegyzék

1. REKORD

A rekordban különböző típusú, de összetartozó adatokat tárolunk. Legnagyobb előnye abban áll, hogy ezeket az adatcsoportokat egyszerre tudjuk kezelni és mozgatni. A fejezet az egyszerű rekordokon kívül tárgyalja az egymásba ágyazott és változó hosszúságú rekordokat is. Szó lesz ezenkívül a rekordok egy, illetve több szempont szerinti rendezéséről.

1.1 Rekord adattípus

Sokszor találkozunk olyan feladattal, melyben összetartozó adatokat, adatcsoportokat kell kezelni. Ilyen például a következő:

Feladat
Készítsünk nyilvántartást egy áruházban forgalmazott árukról! Az egyes árukról a következő adatokat kell nyilvántartani:

- Árukód — pontosan 4 karakter: az első betű, a többi szám
- Leírás — maximum 20 karakter
- Egységár — valós

Kérjük be az áruk adatait, és tároljuk el azokat! Feltételezzük, hogy az áruházban nincs 1000-nél több áru.

Vegyünk mintának egy táblázatot, mely a szóban forgó árukat tartalmazza:

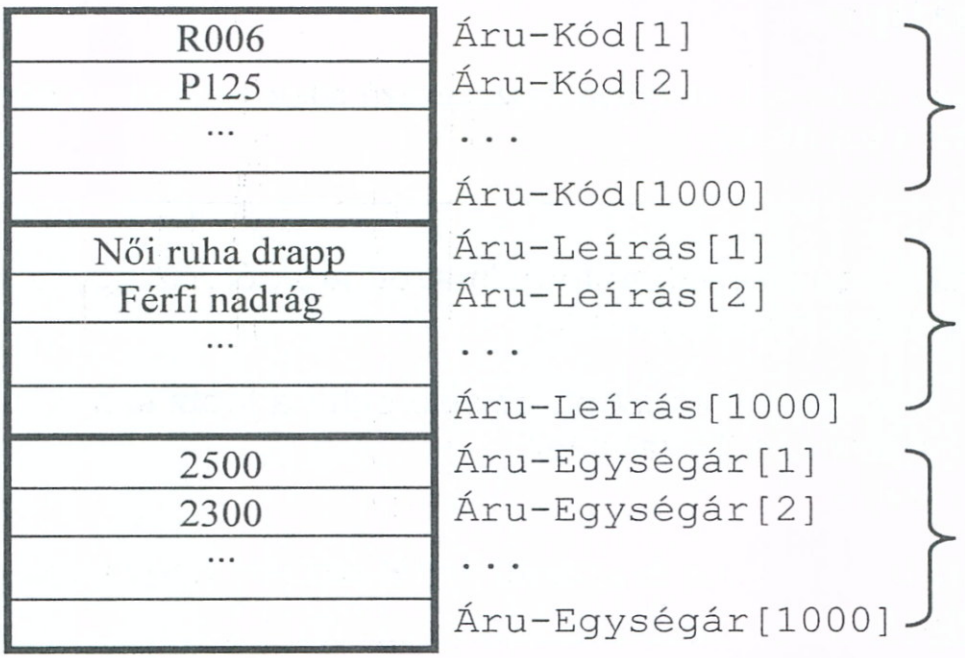
Árukód	*Leírás*	*Egységár*
R006	Női ruha drapp	2500.00
N125	Férfi nadrág	2300.00
C046	Sportcipő fekete	4500.00
P156	Női pulóver	1900.00
B901	Bikini	1650.00
C723	Sportcipő barna	3600.00
Z013	Férfi zokni 5 db	460.00
...	...	...

A tömb adatszerkezet segítségével *azonos típusú* adatokat tudunk tárolni, ahol egy adatot a tömb neve és a hozzátartozó index azonosít. Az előbbi táblázatunk sorai azonos típusúak, így azokból felépíthetnénk egy egydimenziós tömböt. A sorokon belül azonban az egyes árukhoz tartozó adatok (kód, leírás, egységár) különböző típusúak, így azok semmiképpen nem alkothatnak egy beágyazott tömböt. Kérdés: milyen adatszerkezet az, melyben tárolhatjuk adatainkat?

Az egyik megoldás a párhuzamos tömbök használata. Három különböző tömböt veszünk fel: az egyikben az árukódokat, a másikban a leírásokat, a harmadikban pedig az egységárakat tároljuk. Az összetartozó adatokra azonos indexszel hivatkozhatunk:

```
Áru-Kód      : Tömb(1..1000: Szöveg[4])
Áru-Leírás   : Tömb(1..1000: Szöveg[20])
Áru-Egységár : Tömb(1..1000: Valós)
```

Ezek a tömbök a memóriában egymás után, sorfolytonosan helyezkednek el:



Így a memóriában először az 1000 darab árukód szerepel, aztán következnek a leírások, végül az egységárak kerülnek tárolásra. Már ebből is lehet sejteni, hogy egy adatcsoport mozgatása ebben a konstrukcióban csak adatonként lehetséges.

Párhuzamos tömbök esetén az egyes adatcsoportoknak így adhatunk például értéket:

```
Áru-Kód[1]      := 'R006'
Áru-Leírás[1]   := 'Női ruha drapp'
Áru-Egységár[1] := 2500
```

Az összes adat megjelenítése a következőképpen történik:

```
Ciklus I = 1-től Utolsó-ig
   Ki: Áru-Kód[I], Áru-Leírás[I], Áru-Egységár[I]
Ciklus vége
```

A másik megoldás a rekord használata. A rekord a különböző típusú, de összetartozó adatokat összefogja, azokat egy adatként kezeli. Az ilyen adatcsoportok a memóriában egymás mellett helyezkednek el, és az egész adatcsoportra egy névvel lehet hivatkozni. A rekord adatait *mező*knek nevezzük.

A rekord típus deklarációja mondatszerű leírással:

```
TRekord = Rekord(
   Mező1 : Típus1
   Mező2 : Típus2
   ...
   Mezőn : Típusn
)
```

TRekord a rekord típus azonosítója, *Mező1*, *Mező2*, ... *Mezőn* a rekord mezőinek azonosítói. *Típus1*, *Típus2*, ... *Típusn* rendre a mezők típusai. *TRekord* egy előfordulásának deklarálása:

```
Rekord : TRekord ;
```

A rekord mezőire a következőképpen hivatkozhatunk:

```
Rekord.Mező1, Rekord.Mező2 stb.
```

A feladatban definiált áru adatcsoportot így deklarálhatjuk:

```
TÁru = Rekord(
   Kód : Szöveg[4]
   Leírás : Szöveg[20]
   Egységár : Valós
)
```

Az áruház nyilvántartására egy rekordokból álló tömböt veszünk fel:

```
Áruk : Tömb(1..1000: TÁru)
```

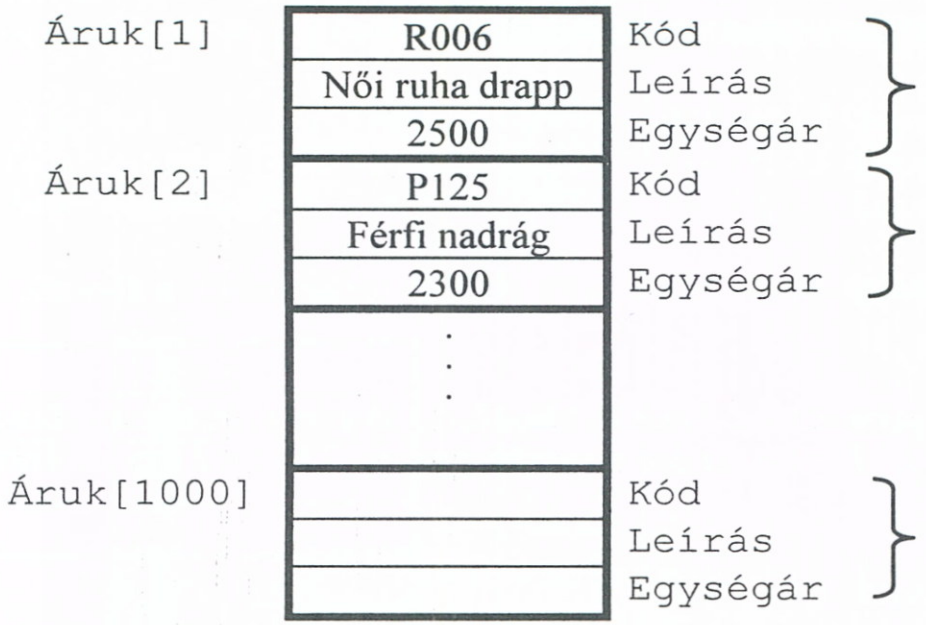
Az első adatcsoportnak most így adunk értéket:

```
Áruk[1].Kód      := 'R006'
Áruk[1].Leírás   := 'Női ruha drapp'
Áruk[1].Egységár := 2500
```

A megjelenítést a következő ciklus végzi el:

```
Ciklus I = 1-től Utolsó-ig
   Ki: Áruk[I].Kód, Áruk[I].Leírás, Áruk[I].Egységár
Ciklus vége
```

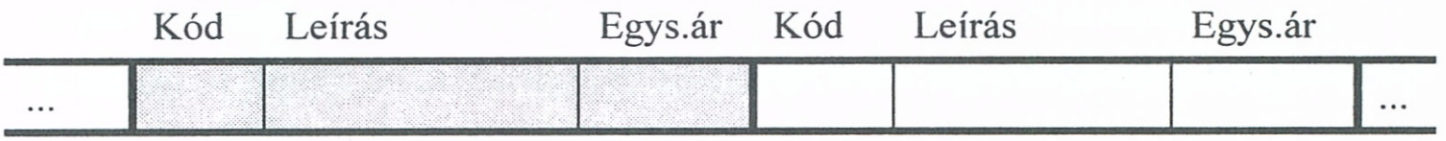
Az *Áruk* tömb elhelyezkedése a memóriában:



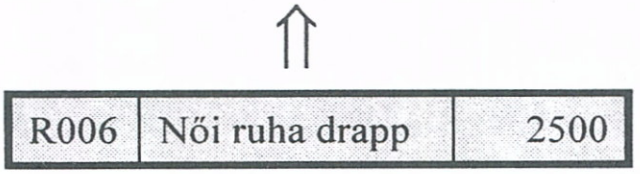
A rekordokból álló tömböt úgy is feltölthetjük, hogy mindig egy új rekordba visszük be az értékeket, majd az ellenőrzött adatrekordot áttesszük a tömb megfelelő helyére:

```
Áru : TÁru ;

Be : Áru.Kód, Áru.Leírás, Áru.Egységár
Áruk[I] := Áru
```

Áruk:



Áru:



A rekord adattípus Turbo Pascal deklarációja a következő:

```
Type
  TRekord = Record
    Mező1 : Típus1 ;
    Mező2 : Típus2 ;
    ...
    Mezőn : Típusn ;
  End ;
```

Ha *Rekord* egy *TRekord* típusú változó, akkor *Rekord.Mezői* egy *Típusi* típusú változó: annak értéket adhatunk, beolvashatjuk stb. – minden olyan művelet elvégezhető vele, ami egy bármilyen más *Típusi* típusú változóval.

Ennek megfelelően az árukat tartalmazó tömböt Turbo Pascalban így deklaráljuk:

```
Type
  TAru = Record
    Kod : String[4] ;
    Leiras : String[20] ;
    Egysegar : Real ;
  End ;

Var
  Aruk : Array[1..1000] Of TAru ;
```

Több rekordtípusban is szerepelhet ugyanaz a mezőnév, de egy rekordon belül a mezőnévnek egyedinek kell lennie. A rekord mezőit az őt tartalmazó rekordnevekkel azonosítjuk, illetve *minősítjük*. Például:

```
Type
  TRendeles = Record
    Kod : String[4] ;
    Db : Integer ;
  End ;

Var
  Rendeles : TRendeles ;

...
Rendeles.Kod := Aruk[I].Kod ;
ReadLn(Rendeles.Db) ;
WriteLn('Fizetendő: ',Rendeles.Db * Aruk[I].Egysegar) ;
```

1.2 Egymásba ágyazott rekordok

Előfordulhat, hogy a rekord mezője szintén rekord típus. Ilyen eset például, ha a rekordban egy dátum szerepel, és a dátum komponenseire (évre, hónapra és napra) külön kell hivatkoznunk:

```
Type
   TNev = String[30] ;
   TDatum = Record
      Ev : Word ;
      Ho : 1..12 ;
      Nap : 1..31 ;
   End ;

   TSzemely = Record
      Nev : TNev ;
      SzulDatum : TDatum ;
      Anya : TNev ;
   End ; { TSzemely }

Var
   Szemely : TSzemely ;
...
```

A születési évre, illetve hónapra így hivatkozhatunk:

```
Szemely.SzulDatum.Ev, Szemely.SzulDatum.Ho
```

Előfordulhat, hogy meg kell jegyezni az anya nevét és születési dátumát is:

```
   TSzemely = Record
      Nev : TNev ;
      SzulDatum : TDatum ;
      Anya : Record
         Nev : TNev ;
         SzulDatum : TDatum ;
      End ; { Anya }
   End ; { TSzemely }
```

Hivatkozás a személy anyjának születési dátumára és annak évére:

```
Szemely.Anya.SzulDatum, Szemely.Anya.SzulDatum.Ev
```

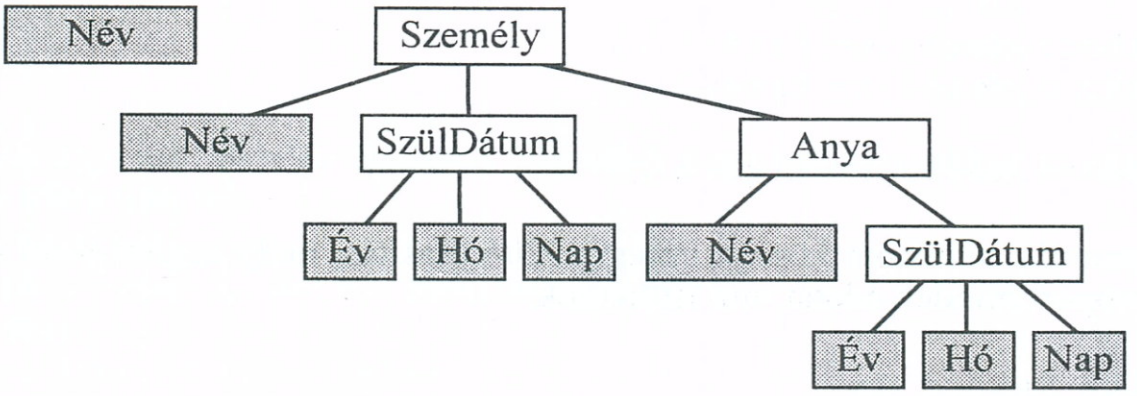
Látható, hogy a rekordokat tetszőlegesen egymásba ágyazhatjuk. A rekordon belüli mezőre mindig úgy hivatkozunk, hogy a rekord azonosítója és a mező közé egy pontot

teszünk. Ha ez a mező rekord, akkor a további mezőkre hasonlóképpen hivatkozhatunk.

Tekintsük a fenti típusok alapján a következő deklarációkat:

```
Var
   Nev : TNev ;
   Szemely : TSzemely ;
```

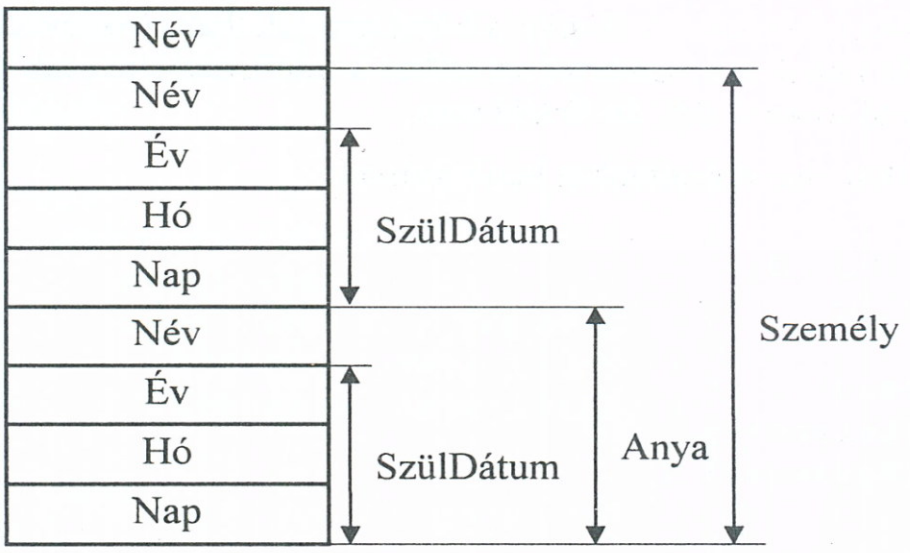
A most deklarált adatok struktúrája a következő:



Az „igazi", illetve elemi adatok a struktúra végein helyezkednek el – a *Személy*, a *Személy.SzülDátum*, *Személy.Anya* és a *Személy.Anya.SzülDátum* csupán csoportnevek. A *Név* változó nem osztható. A *Személy* változó összesen nyolc elemi adatból áll:

1. Személy.Név
2. Személy.SzülDátum.Év
3. Személy.SzülDátum.Hó
4. Személy.SzülDátum.Nap
5. Személy.Anya.Név
6. Személy.Anya.SzülDátum.Év
7. Személy.Anya.SzülDátum.Hó
8. Személy.Anya.SzülDátum.Nap

Nézzük meg, hogy helyezkednek el a deklarált adatok a memóriában:



Óvakodjunk a sok egymásba ágyazástól, mert a program áttekinthetetlen lesz tőle!

1.3 With utasítás

A rekord használata az előnyök mellett egy apró kellemetlenséggel is jár: a rekord mezőit állandóan minősíteni kell, ami az egyes utasításokat rendkívül hosszúvá teheti. Ha például a rekordnak tíz mezője van, és minden mezőnek értéket szeretnénk adni, akkor a mezők minősítéseit tízszer kell leírnunk:

```
Rekord.Mező1 := Érték1 ;
Rekord.Mező2 := Érték2 ;
...
Rekord.Mezőn := Értékn ;
```

A programozók munkáját könnyíti meg a *With* utasítás, mely segítségével egy egész utasításcsoport minden adatát minősíthetünk:

```
With Rekord Do
  Begin
    Mező1 := Érték1 ;
    Mező2 := Érték2 ;
    ...
    Mezőn := Értékn ;
  End ;
```

A With utasítás általános formája a következő:

```
With Rekord-azonosító Do
  Utasítás
```

Utasítás legtöbbször összetett utasítás. A fordító a blokkban található összes azonosítót minősíteni próbálja. Ha ennek nincsen értelme, akkor a minősítés elmarad. A fenti példában *Mezői* minősíthető *Rekord*-dal, de *Értéki* nem.

Példaként töltsük fel az 1000 elemű árutömböt beolvasással:

```
For I := 1 To 1000 Do
  With Aruk[I] Do
    Begin
      ReadLn(Kod) ;
      ReadLn(Leiras) ;
      ReadLn(Egysegar) ;
    End ;
```

Vigyázzunk, nehogy a *For* elé tegyük a *With* utasítást, mert akkor I definiálatlan lenne, és a hivatkozott rekord mindig ugyanaz maradna a ciklusban! A *With* utasításon belül sose változtassuk meg a hivatkozott rekordot!

A With utasításban több rekorddal is minősíthetünk egyszerre. A következő két utasítás ekvivalens egymással:

```
With R1, R2, ... Rn Do
  Utasítás
```

```
With R1 Do
  With R2 Do
    ...
      With Rn Do
        Utasítás
```

Ha *Szemely* egy, a 6. oldalon deklarált *TSzemely* típusú rekord, akkor az anya összes adatát így tölthetjük fel:

```
With Szemely, Anya, SzulDatum Do
  Begin
    Nev := 'Zöld Alma' ;
    Ev := 1899 ;
    Ho := 12 ;
    Nap := 31 ;
  End ;
```

1.4 Változó rekord

Gyakran kényelmes és természetes, ha ugyanaz a tárterület hol ilyen, hol olyan adatcsoportot tud tárolni. A Pascal lehetőséget ad a változó rekordok deklarálására. Ilyenkor a rekord első része mindig egy állandó (fix) rész, melyet különböző változatok leírása követ. A rekord fizikai hosszát (a foglalt memóriahelyet) a leghosszabb változat határozza meg. A programból bármikor bármelyik változatra hivatkozhatunk, a helyes adattárolásra és hivatkozásra a programozónak kell figyelnie. A változó rekord szintaktikája:

```
Type
  TRekord = Record
    Mezőlista
    Case Változó : Típus Of
      Érték1 : (Mezőlista1) ;
      Érték2 : (Mezőlista2) ;
      ...
  End ;
```

Mezőlista és *Változó* együttesen határozzák meg a rekord *állandó (fix) részét,* melyet a rekord változó része követ. *Mezőlista1*, *Mezőlista2* stb. mezői egymásra vannak definiálva. Az egyes mezőlisták mezőire ugyan bármikor lehet hivatkozni, de fontos, hogy az aktuális hivatkozás megfeleljen a konkrét adatnak – egyébként furcsa dolgok történhetnek. A megfeleltetésre a programozónak kell figyelnie. *Változó* ennek segítését célozza: e *szelektormező* értékétől függ, hogy melyik mezőlista van érvényben. Például:

```
Type
   TFajta = (Tegla,Kor) ;
   TIdom = Record
      Terulet : Real ;
      Case Fajta : TFajta Of
         Tegla : (Alap, Magassag: Real) ;
         Kor : (Sugar: Real) ;
   End ;
Var
   K : TIdom ;
...
   K.Fajta := Kor ;
   K.Sugar := 3.9 ;
   K.Terulet := Sqr(K.Sugar)*Pi ;
```

Területe minden idomnak van, de míg a téglának alapja és magassága, addig a körnek sugara. A rekord hosszúságát a változó rész hosszabbik ága, jelen esetben a tégla határozza meg. A teljes hosszúság: *Terület+Fajta+Alap+Magasság*=19 (6+1+6+6) byte.

Előfordulhat, hogy az adattartalomtól függetlenül szeretnénk többféleképpen hivatkozni az adatokra. Ekkor nem adunk meg szelektormezőt, csak egy típust. Példaként olvassunk be egy számot, és írjuk ki a *Szám Div 256* és a *Szám Mod 256* értékeit. Ismeretes, hogy a tárolt szónak előbb az alsó, majd a felső byte-ja kerül tárolásra:

```
Type
   TRekord = Record
      Case Boolean Of
         False : (Egesz : Word) ;
         True : (Also, Felso : Byte) ;
   End ;
Var
   R : TRekord ;
Begin
   ReadLn(R.Egesz) ;                    { 268 }
   WriteLn(R.Felso:3,R.Also:3) ;        { 1 12 }
End.
```

Most nézzünk egy nagyobb feladatot:

Feladat

Definiáljunk egy adatstruktúrát, mely egy publikációt képes tárolni. Publikáció lehet egy könyv vagy egy cikk. A következő adatokat mindkét esetben tárolni kell:

Szerzők száma	egész
Szerzők	maximum 5 szerző, egyenként max. 20 karakter
Cím	maximum 50 karakter
Kiadás éve	1900..2000
Fajta	Könyv/Cikk

Könyv esetén más adatokat kell tárolnunk, mint cikk esetén.
Ha a publikáció könyv:

Kiadó	maximum 22 karakter
Város	maximum 15 karakter

Ha a publikáció cikk:

Folyóirat neve	maximum 20 karakter
Kötet száma	egész
Első oldal, Utolsó oldal	egész

Definiáljunk egy adatstruktúrát, mely maximálisan 300 darab publikáció tárolására alkalmas.

A publikációt egy változó rekordban tároljuk, mely felépítése a következő:

```
Type
   Publ_Fajta = (Konyv,Cikk) ;
   TPublikacio = Record
      SzerzokSzama : Byte ;
      Szerzok : Array[1..5] Of String[20] ;
      Cim : String[50] ;
      KiadasEve : 1900..2000 ;
      Case Fajta : Publ_Fajta Of
         Konyv : (
            Kiado : String[22] ;
            Varos : String[15]) ;
         Cikk : (
            Folyoirat : String[20] ;
            Kotet, ElsoOldal, UtolsoOldal : Word) ;
   End ; {Publikacio }

Var
   Publikaciok : Array[1..300] Of TPublikacio ;
```

Egy *TPublikacio* típusú változó a memóriában a következőképpen helyezkedik el (a rekord teljes memóriafoglalása 199 byte):

<table>
<tr><td colspan="2">SzerzokSzama</td><td colspan="2">1 byte</td></tr>
<tr><td colspan="2">Szerzok</td><td colspan="2">5*21 byte</td></tr>
<tr><td colspan="2">Cim</td><td colspan="2">51 byte</td></tr>
<tr><td colspan="2">KiadasEve</td><td colspan="2">2 byte</td></tr>
<tr><td colspan="2">Fajta</td><td colspan="2">1 byte</td></tr>
<tr><td rowspan="2">Kiado</td><td rowspan="2">23 byte</td><td>Folyoirat</td><td>21 byte</td></tr>
<tr><td>Kotet</td><td>2 byte</td></tr>
<tr><td rowspan="3">Varos</td><td rowspan="3">16 byte</td><td>ElsoOldal</td><td>2 byte</td></tr>
<tr><td>UtolsoOldal</td><td>2 byte</td></tr>
<tr><td colspan="2"></td></tr>
</table>

Hivatkozás a 232. publikáció 2. szerzőjének 1. betűjére:

```
Publikaciok[232].Szerzok[2][1]
```

Ha ez cikk, akkor kiírjuk a folyóirat nevét:

```
With Publikaciok[232] Do
  If Fajta = Cikk Then
    WriteLn(Folyoirat) ;
```

Feladat
A *Publikaciok* tömbbe vigyünk be két rekordot: az egyik egy könyv, a másik egy cikk legyen. A program végén listázzuk ki az adatokat!

```
...
PublikaciokSzama : Word ;
I, J : Word ;
```

```
Begin
  PublikaciokSzama := 2 ;
  With Publikaciok[1] Do
    Begin
      SzerzokSzama := 1 ;
      Szerzok[1] := 'WIRTH, Niklaus' ;
      Cim := 'Algoritmusok+Adatstruktúrák=Programok';
      KiadasEve := 1982 ;
      Fajta := Konyv ;
      Kiado := 'Műszaki könyvkiadó' ;
      Varos := 'Budapest' ;
    End ;

  With Publikaciok[2] Do
    Begin
      SzerzokSzama := 2 ;
      Szerzok[1] := 'Steve Cook' ;
      Szerzok[2] := 'John Daniels' ;
      Cim := 'Designing Object Systems' ;
      KiadasEve := 1994 ;
      Fajta := Cikk ;
      Folyoirat := 'JOOP' ;
      Kotet := 7 ;
      ElsoOldal := 14 ;
      UtolsoOldal := 23 ;
    End ;

  For I := 1 To PublikaciokSzama Do
    With Publikaciok[I] Do
      Begin
        Write(Szerzok[1]) ;
        For J := 2 To SzerzokSzama Do
          Write(' - ',Szerzok[J]) ;
        WriteLn(': ',Cim,KiadasEve:6) ;
        Case Fajta Of
          Konyv :
            WriteLn(Kiado,', ',Varos) ;
          Cikk :
            WriteLn(Folyoirat,' ',Kotet,'. kötet ',
              ElsoOldal:3,'-',UtolsoOldal,' oldalak');
        End ;
      End ;
End.
```

1.5 Rekord konstans

A rekordot – ugyanúgy, ahogy a tömböt – definiálhatjuk típussal rendelkező konstansként. Ekkor a rekord mezőinek úgy adunk kezdőértéket, hogy zárójelben felsoroljuk az egyes mezők azonosítóit és kezdőértékeit. Az értékadásokat pontosvesszők választják el egymástól. Például:

```
Type
   Pont = Record
      X, Y, Z : Integer ;
   End ;

   TSzemely = Record
      Nev : String[20] ;
      IQ : Byte ;
   End ;

Const
   Origo : Pont = (X:0; Y:0; Z:0) ;
   Szemely : TSzemely = (
      Nev : 'Bolond Botond' ;
      IQ : 100) ;
```

1.6 Rekordok rendezése

Az első kötetben több rendezési algoritmussal is megismerkedtünk. A rendezendő elemek tömbben helyezkedtek el, és a rendező algoritmusok mindegyike az elemek cserélgetésével rendezett. A csere mindig két elem összehasonlításának alapján történt.

A rendezendő elemek lehetnek adatcsoportok, vagyis rekordok is. Ebben az esetben az összehasonlítás alapja egy egyszerű vagy összetett feltétel, és ügyelni kell arra, hogy a cserélgetésnél az összetartozó adatokat együtt mozgassuk.

Legyen a feladat a következő:

Feladat

Kérjünk be egymás után személyi adatokat:

Név	maximum 20 karakter
Születési év	4 egész
Foglalkozás	maximum 15 karakter

A bevitelnek akkor van vége, amikor névnél „*" -ot ütnek. Ekkor írjuk ki a bevitt adatokat név szerint rendezetten a következő formában:

```
     Név                        Szülév   Foglalkozás
XXXXXXXXXXXX...XXXXXXXXX        9999     XXXXXXXXXXXXXXX
XXXXXXXXXXXX...XXXXXXXXX        9999     XXXXXXXXXXXXXXX
...
XXXXXXXXXXXX...XXXXXXXXX        9999     XXXXXXXXXXXXXXX
```

A listakép megtervezésénél a kiírandó szöveg helyére X karaktereket, a kiírandó számok helyére pedig 9-eseket szokás írni a megadott hosszúságban. A feladat megoldásához most fel kell vennünk egy rekordokból álló tömböt. A rendezést név szerint növekvőleg kell elvégezni, ami azt jelenti, hogy két rekord akkor van jó sorrendben, ha az első rekordban szereplő név kisebb, mint a másodikban. Rendezéshez a minimumkiválasztásos módszert alkalmazzuk. A rendezési szempont megadására írunk egy logikai függvényt (JóSorrend), mely igaz értéket ad vissza, ha az átadott rekordok jó sorrendben vannak (Rekord1.Nev <= Rekord2.Nev).

Nézzük a Turbo Pascal programot:

```
Program RekRend ;
Uses
  Crt ;

Const
  Max = 1000 ;

Type
  TNev = String[20] ;
  TFogl = String[15] ;

  TSzemely = Record
    Nev : TNev ;
    Ev : Word ;
    Fogl : TFogl ;
  End ;

Var
  NevekSzama : Integer ;
  Szemelyek : Array[1..Max] Of TSzemely ;

{ A vizsgálandó rekordoknak csak a címét adjuk át, mert
a rekordok másolása a veremre sok időt venne igénybe: }
Function JoSorrend(Var Sz1, Sz2: TSzemely) : Boolean ;
  Begin
    JoSorrend := Sz1.Nev <= Sz2.Nev ;
  End ;
```

```
{ Két TSzemely típusú rekord kicserélése: }
Procedure Csere(Var Sz1, Sz2: TSzemely);
  Var
    Seged : TSzemely ;
  Begin
    Seged := Sz1 ;
    Sz1 := Sz2 ;
    Sz2 := Seged ;
  End ;

Var
  Nev : TNev ;
  I, J, MinIndex : Integer ;

Begin { Program }
  { Személy rekordok bevitele: }
  ClrScr ;
  NevekSzama := 0 ;
  Write('Név      : ') ; ReadLn(Nev) ;
  While (Nev <> '*') And (NevekSzama < Max) Do
    Begin
      Inc(NevekSzama) ;
      Szemelyek[NevekSzama].Nev := Nev ;
      Write('Szülév  : ') ;
      ReadLn(Szemelyek[NevekSzama].Ev) ;
      Write('Foglalk.: ') ;
      ReadLn(Szemelyek[NevekSzama].Fogl) ;
      WriteLn ;
      Write('Név      : ') ;
      ReadLn(Nev) ;
    End ;

  { Minimumkiválasztásos rendezés. Ha nem jó a sorrend,
  megjegyezzük az indexet, majd cseréljük a rekordokat:}
  For I := 1 To NevekSzama-1 Do
    Begin
      MinIndex := I ;
      For J := I+1 To NevekSzama Do
        If Not JoSorrend(Szemelyek[MinIndex],Szemelyek[J])
        Then
          MinIndex := J ;
      If MinIndex <> I Then
        Csere(Szemelyek[I],Szemelyek[MinIndex]) ;
    End ;
```

```
    { Rekordok kiírása: }
    ClrScr ;
    WriteLn(' Név                  Szül. év   Foglalkozás') ;
    WriteLn('--------------------  ----  ---------------');
    For I := 1 To NevekSzama Do
      With Szemelyek[I] Do
        Begin
          Write(Nev,'':22-Length(Nev)) ;
          WriteLn(Ev:4,' ',Fogl) ;
        End ;

End. { Program vége }
```

A rendezésben itt az a nagyszerű, hogy a rendezési algoritmust függetlenítettük a rendezési szempontól is, és a cserélési mechanizmustól is. Megmaradt a „tiszta" rendezési algoritmus. A *JóSorrend* függvényt, illetve a *Csere* eljárást más rendezési algoritmus esetén is alkalmazhatjuk.

1.7 Rendezés több szempont szerint

A rekordok rendezésénél most különféle szempontokat fogunk figyelembe venni:

- Lehet, hogy nem növekvőleg szeretnénk az adatokat rendezni, hanem csökkenőleg.
- Lehet, hogy keresztnév szerint óhajtjuk a rendezést.
- Lehet, hogy nem név szerint akarunk rendezni, hanem születési év szerint.
- Egyforma elemek esetén lehet, hogy egy másik szempontot is figyelembe akarunk venni, például év és azon belül név szerint növekvőleg szeretnénk a sorrendet (egymásba ágyazott rendezés).

Rendezésnél a rendezési szempontot tetszőlegesen megválaszthatjuk – a lényeg az, hogy a szempontot meg tudjuk fogalmazni. Rendezésnél egy sorrendet kell megállapítani, vagyis ha két tetszőleges adatról, illetve adatcsoportról a szempont alapján egyértelműen el tudjuk dönteni, hogy melyik kerüljön előrébb, akkor a rendezés elvégezhető. A következő példákban Csoport1 és Csoport2 mindegyike az előző pontban megadott TSzemély típusú rekord. Azt kell tehát minden esetben megfogalmaznunk, hogy mikor kell Csoport1-nek előbb szerepelnie, mint Csoport2.

- Ha a rendezési szempont az, hogy *név szerint növekvő sorrendben* legyenek az adatcsoportok, akkor Csoport1 előbb van, mint Csoport2, ha a Csoport1-ben lévő név kisebb vagy egyenlő, mint a Csoport2-ben lévő név. Ha a megfogalmazott feltétel nem igaz, akkor a sorrend nem jó. Ha a két név egyforma, akkor a sorrend mindegy ugyan, de a feltétel megfogalmazásánál egyértelműen meg kell adnunk, legyen-e csere, vagy nem. A jó sorrend feltétele tehát:

  Csoport1.Név <= Csoport2.Név

- *Név szerint csökkenő rendezettség* esetén a jó sorrend megfogalmazása a következő:

  Csoport1.Név >= Csoport2.Név

- *Keresztnév szerint növekvő rendezettség* esetén egy kicsit bonyolultabb feltételt kell megfogalmaznunk. A nevekből ki kell emelnünk a keresztneveket, és a csoportok hasonlításában ezeket kell szerepeltetni. A keresztnevet természetesen minden elemnél ki kell „hámoznunk" a névből:

  Csoport1.Keresztnév <= Csoport2.Keresztnév

  Most az egyszerűség kedvéért tökéletes neveket veszünk alapul, így az első szóköz utáni részlánc lesz a keresztnév. Turbo Pascal nyelven:
  Copy(Csoport1.Név,Pos(' ',Csoport1.Név)+1,255) <=
  Copy(Csoport2.Név,Pos(' ',Csoport2.Név)+1,255)

- Ha a *név második betűjére* szeretnénk rendezni *növekvőleg* az adatcsoportokat, akkor ez a jó sorrend:

  Csoport1.Név[2] <= Csoport2.Név[2]

- *Év szerint növekvő rendezettség* esetén a jó sorrend megfogalmazása a következő:

  Csoport1.Év <= Csoport2.Év

- Legyen a rendezettség év szerint, de most a szokásostól egy kicsit eltérő: *előbb szerepeljenek századunk évszámai növekvőleg, aztán pedig az 1900-nál előbbi évszámok csökkenőleg*! Ez is egy jól meghatározott rendezettség – a jó sorrend megfogalmazása a következő:

  (Csoport1.Év >= 1900) és (Csoport2.Év < 1900)

  vagy

  (Csoport1.Év >= 1900) és (Csoport2.Év >= 1900) és
  (Csoport1.Év <= Csoport2.Év)

  vagy

  (Csoport1.Év < 1900) és (Csoport2.Év < 1900) és
  (Csoport1.Év >= Csoport2.Év)

- Sok egyforma év esetén megadhatunk egy második szempontot is. Fogalmazzuk most meg azt, hogy *születési év szerint, és azon belül név szerint növekvő* a rendezettség. A második szempontnak természetesen csak akkor van értelme, ha az első szempont alapján a helyes sorrend nem dönthető el, illetve nem egyértelmű. Ha tehát a születési évek egyenlőek, döntsenek a nevek:

  (Csoport1.Év < Csoport2.Év) vagy

  (Csoport1.Év = Csoport2.Év) és (Csoport1.Név <= Csoport2.Név)

- A *születési év szerint csökkenő, azon belül név szerint növekvő* sorrendet így fogalmazzuk meg:

  (Csoport1.Év > Csoport2.Év) vagy

  (Csoport1.Év = Csoport2.Év) és (Csoport1.Név <= Csoport2.Név)

- A *születési év szerint növekvő, azon belül név szerint csökkenő* sorrend megfogalmazása a következő:

  (Csoport1.Év < Csoport2.Év) vagy

  (Csoport1.Év = Csoport2.Év) és (Csoport1.Név >= Csoport2.Név)

Lehetne természetesen három vagy több szempontunk is. Miért ne lehetne például adataink közt egyforma születési évek mellett két egyforma név is? Ha a sorrendet sem a születési év, sem a név nem tudja eldönteni, döntsön a foglalkozás:

- A *születési év szerint növekvő, azon belül név szerint csökkenő, azon belül foglalkozás szerint növekvő* rendezettséghez ez a jó sorrend:

  (Csoport1.Év < Csoport2.Év)

  vagy

  (Csoport1.Év = Csoport2.Év) és (Csoport1.Név > Csoport2.Név)

  vagy

  (Csoport1.Év = Csoport2.Év) és (Csoport1.Név = Csoport2.Név) és (Csoport1.Fogl <= Csoport2.Fogl)

Az egymásba ágyazásokat a végtelenségig folytathatnánk. Figyelembe vettük, hogy a Pascalban az és kapcsolat erősebb, mint a vagy kapcsolat. Ha nem így lenne, akkor a részfeltételeket zárójelbe kellene tenni!

Ne felejtsük el a beágyazott szempont megfogalmazásánál az addigi egyenlőségeket leírni, mert könnyen csapdába eshetünk. Ha elhagynánk az egyenlőségeket, akkor abban az esetben, ha *Csoport1.Fogl < Csoport2.Fogl*, az előbbi háromszorosan összetett feltétel igaz lenne attól függetlenül, milyen sorrendben vannak az évek.

Az adatcsoportnak azt az adatát, mely szerint a rendezést elvégezzük, az adatcsoport *rendezési kulcs*ának szokás nevezni. A kulcs lehet az adatcsoport egyetlen adata is, de lehet valamilyen algoritmus szerint összetett adat is. Fontos, hogy a kulcs egyértelműen megállapítható legyen az adatcsoport adataiból. Szokás az adatcsoporthoz tartozó kulcsképző függvényt írni – ekkor a kulcsokat kellene egymáshoz hasonlítani:

```
...
Ha Kulcs(J) < Kulcs(MinIndex) akkor
  MinIndex := J
...
```

Változtassuk most meg a 14. oldalon megfogalmazott feladatban a rendezési szempontot: rendezzünk most születési év szerint csökkenőleg, azon belül név szerint növekvőleg! A programban mindössze a *JóSorrend* függvényt kell megváltoztatnunk:

```
Function JoSorrend(Var Sz1, Sz2: TSzemely) : Boolean ;
  Begin
    JoSorrend := (Sz1.Ev > Sz2.Ev) Or
      (Sz1.Ev = Sz2.Ev) And (Sz1.Nev <= Sz2.Nev) ;
  End ;
```

Végül teszteljük a programot! Tegyük fel, hogy a következő adatokat ütik be (oszlopfolytonosan):

Nemoda Buda	Bor Ivó	Boldog Karácsony	Fekete Farkas
1975	1975	1976	1973
Gázszerelő	Programozó	Programozó	Programozó
Kerti Viola	Bor Virág	Esti Hajnal	Fehér Holló
1975	1976	1975	1976
Mészáros	Kertész	Kertész	Asztalos
Györgyi György	Szomorú Szilveszter	Bolond Istók	Bőrönd Ödön
1973	1974	1976	1975
Asztalos	Bohóc	Mészáros	Bőröndös
			*

A program a következő listát fogja produkálni:

```
  Név                 Szül. év   Foglalkozás
--------------------   ----     --------------
Boldog Karácsony       1976     Programozó
Bolond Istók           1976     Mészáros
Bor Virág              1976     Kertész
Fehér Holló            1976     Asztalos
Bor Ivó                1975     Programozó
Bőrönd Ödön            1975     Bőröndös
Esti Hajnal            1975     Kertész
Kerti Viola            1975     Mészáros
Nemoda Buda            1975     Gázszerelő
Szomorú Szilveszter    1974     Bohóc
Fekete Farkas          1973     Programozó
Györgyi György         1973     Asztalos
```

1.8 Csoportos adatbevitel

Számtalan esetben van szükség arra, hogy a felhasználótól egy egész adatcsoport bevitelét kérjük – ilyen eset például, ha egy rekord mezőit szeretnénk kitöltetni. Adatcsoport bevitele általában egy párbeszédablakban történik. A program a párbeszédablakon keresztül „beszélget" a felhasználóval: a felhasználó számára adatokat szolgáltat úgy, hogy azokat a képernyőre „vetíti", és a felhasználótól adatokat gyűjt be a billentyűzetről, vagy más eszközökről, miközben folyamatosan kijelzi a bevitt adatokat. A párbeszédablak feladata tehát a felhasználó és a program adatai közötti kapcsolat megteremtése, mégpedig oly módon, hogy az a felhasználó szempontjából a lehető legkényelmesebb legyen, a program számára pedig helyes adatokat biztosítson. A párbeszédablak egy *felhasználói illesztő* (user interface) a program adataihoz. Ebben a pontban egy ilyen felhasználói illesztőt próbálunk elkészíteni. Természetesen az illesztésnek számtalan módja lehetséges, de vannak bizonyos általános szabályok, *szabványok*, melyeket a programozónak be kell tartania. A felhasználó az összetartozó adatokat általában egyszerre szeretné látni a képernyőn, és alapvető igény az is, hogy egy adat vagy adatcsoport az abból való kilépésig szerkeszthető, módosítható legyen. A program csak akkor „cselekedhet", ha a kész adat vagy adatcsoport előállításáról megkapta a megfelelő jelet.

A csoportos adatbevitelt most egy konkrét rekord kitöltése kapcsán tárgyaljuk – legyen ez a következő személy rekord:

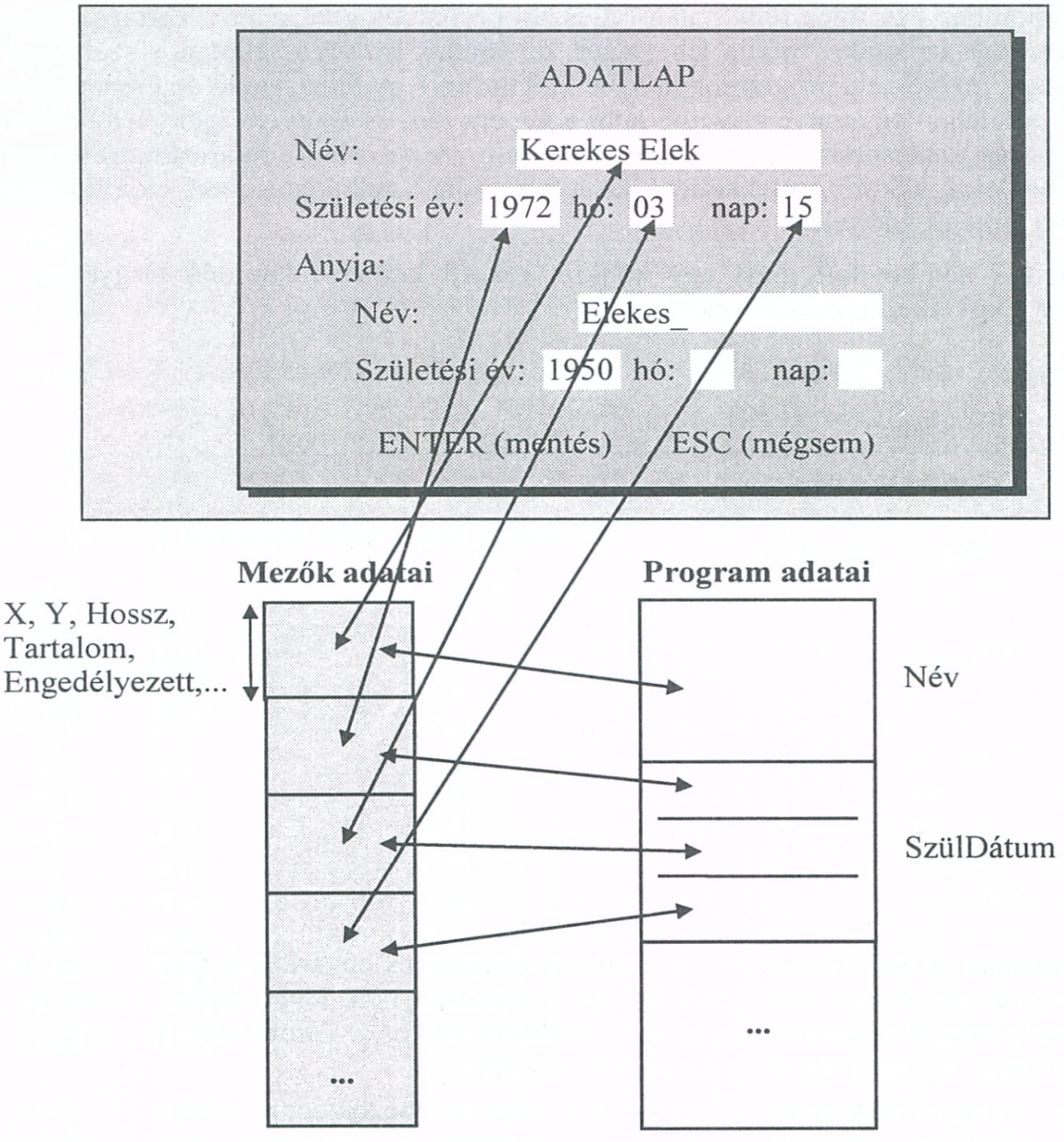
```
Type
  TNev = String[30] ;
  TDatum = Record
    Ev : Word ;
    Ho : 1..12 ;
    Nap : 1..31 ;
  End ;

  TSzemely = Record
    Nev : TNev ;
    SzulDatum : TDatum ;
    Anya : Record
      Nev : TNev ;
      SzulDatum : TDatum ;
    End ; { Anya }
  End ; { TSzemely }
```

A programnak el kell készítenie egy beviteli ablakot, és egyértelmű kapcsolatot kell létesítenie az ablak és a rekord között. Ez a kapcsolat nem lehet direkt, hiszen ha a felhasználó mégsem szeretné az ablakot kitölteni, akkor a rekordnak változatlannak kell maradnia. A kapcsolatot két lépcsőben oldjuk meg.

- Az egyik kapcsolat a képernyő és a képernyőmezők adatai között lesz, ahol a képernyő mezőiről a következő információkat jegyezzük: mező pozíciója,

hossza, aktuális tartalma, bevihető karakterek halmaza stb. A mezőkhöz tartozó adatok a ciklikus szerkesztés miatt egyforma típusúak, a mezők aktuális értékeit a szerkesztés befejeztéig egy-egy *String* típusú változó tartalmazza. Ez a kapcsolat direkt, a mezők a képernyőre ki vannak vetítve. Az adatok megjelenését meghatározó adatok kizárólag a felhasználói illesztő részei.

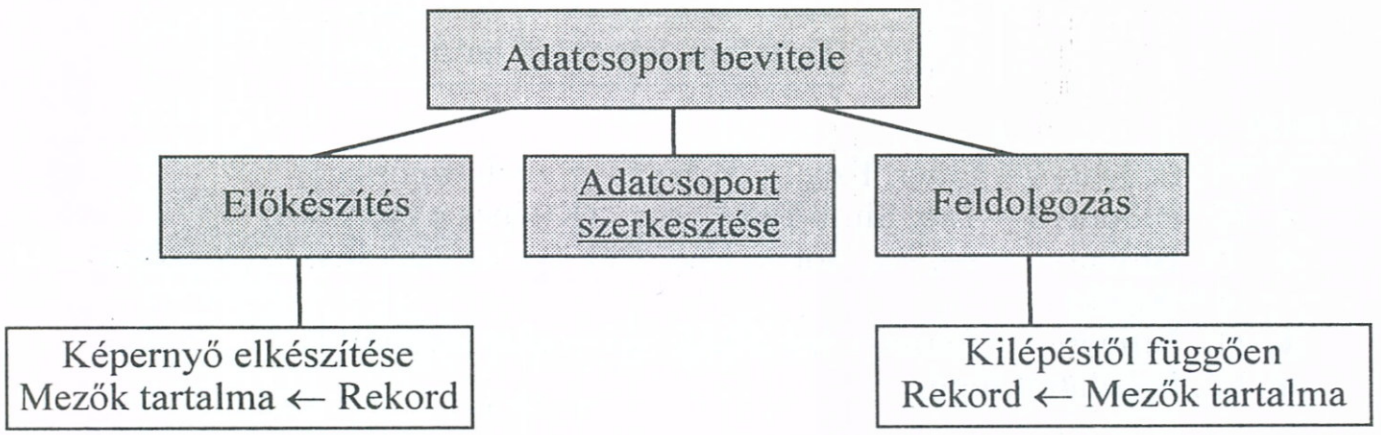
- A másik direkt kapcsolat a szerkesztés alatt álló mezők konkrét tartalmai és a program adatai között van, mely már független a képernyőmezők konkrét megjelenéseitől (pozíció, hossz, szín stb.). A bevitel elején a program adatait át kell tölteni a közvetlen kapcsolatot létesítő mezőkbe, a bevitel végén pedig – a szerkesztésből való kilépés függvényében – a megszerkesztett adatokat vissza kell tölteni a program adataiba:



Az adatcsoport szerkesztésekor mindig van egy aktuális mező – ebbe kerülnek az éppen leütött megjeleníthető karakterek. Az adatcsoport szerkesztése közben általában meg kell adni a lehetőséget a mezők közötti lépegetésre, hiszen kitöltés közben előfordulhatnak elgépelések, tévedések. A mezők közti váltás általában a *Tab*, illetve *Shift-Tab* billentyűk leütésére történik. Még jobb megoldás, ha programunk az egér működésére is reagál, és azt a mezőt teszi aktuálissá, amelyen a felhasználó kattint egyet. Ettől azonban most ismeretek hiányában eltekintünk.

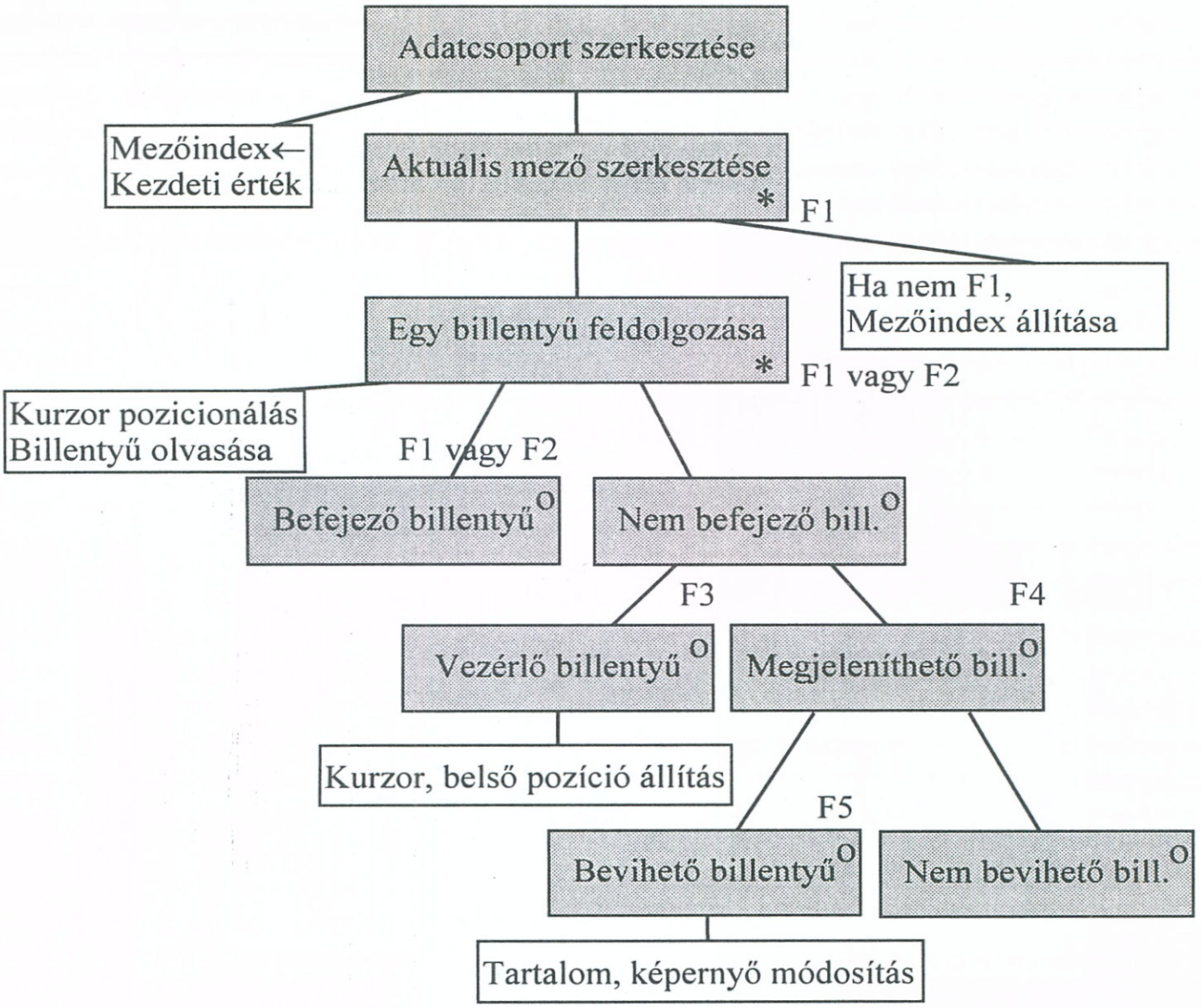
Az adatcsoport szerkeszthető mindaddig, amíg a felhasználó nem utasítja a programot annak befejezésére. Ez az utasítás minden esetben legalább kétféle: adatok elfogadása, illetve elvetése, melyet általában az *Enter* vagy *Esc* gombok leütésével lehet érvényesíteni. Ha a felhasználó az adatcsoport elfogadását kérte, akkor a program elvégezhet bizonyos ellenőrzéseket, és ennek megfelelően a bevitt adatcsoportot feldolgozza, vagy a hibára utaló üzenet kíséretében rááll arra a mezőre, amelyben a hibát találta. A menekülésre mindig meg kell adnunk a lehetőséget, hiszen kitöltés közben bárki meggondolhatja magát. Ilyen esetben az adatok részleges kitöltése semmilyen következményt nem vonhat maga után.

Adatcsoport szerkesztésekor leüthetünk vezérlőbillentyűket vagy megjeleníthető billentyűket. A vezérlőbillentyű vezérelheti a mező szerkesztését, vagy utasíthat a mezőből való kilépésre. Ez utóbbiak is lehetnek olyanok, melyek nem csak a mezőből, hanem az egész párbeszédablakból való kilépést jelentik. A megjeleníthető karakterek, mint az elnevezés is sugallja, elvileg megjelenhetnek a mezőben. Jó dolog azonban a karakterek előzetes szűrése, hiszen egy numerikus egész mező esetén felesleges a felhasználó által leütött betű karaktereket elfogadni.



Adatcsoport szerkesztése:

Mezők közötti lépegetés, módosítás. Lásd következő oldal.



Feltételek:

- F1: Billentyű = Enter, Esc, ... (Kilépés az adatcsoportból)
- F2: Billentyű = Tab, Shift-Tab,↑, ↓, ... (Kilépés a mezőből)
- F3: Billentyű = ←, →, Home, End, Del, Backspace, Ins, ... (Mezőszerkesztő billentyűk)
- F4: Billentyű = Bármi, a kontroll karaktereket leszámítva
- F5: Feladattól függ

F3 (Mezőszerkesztő billentyűk) részletesebb kifejtése:

- ←: Kurzor balra, ha az nem az első karakteren áll
- →: Kurzor jobbra, ha az nem az utolsó karakteren áll
- *Home:* Kurzor a mező elejére áll
- *End:* Kurzor a mező végére áll
- *Del:* Kurzor alatti karakter törlése, ha nem üres a mező – a kurzor pozíciója változatlan marad
- *Backspace:* Kurzor előtti karakter törlése, ha van – kurzor eggyel visszalép
- *Ins:* Beszúró/felülíró üzemmód

A mezők adatait a *Mezők* tömbben tároljuk:

```
Type
  Karakterhalmaz = Set Of Char ;

  TMezo = Record
    X,Y,Hossz: Byte ;
    Tartalom: String ;
    Engedelyezett: Karakterhalmaz ;
  End ;

Const
  Mezok: Array[1..8] Of TMezo =
    ((X:20 ; Y:5 ; Hossz:30 ; Tartalom: '' ;
    Engedelyezett: [' ','a'..'z','A'..'Z']),
    (X:18 ; Y:6 ; Hossz:6 ; Tartalom: '' ;
    Engedelyezett: ['0'..'9']),

    ... ) ;
```

Most nézzük a feladat egy durva algoritmusát. *Poz* jelenti a kurzor aktuális pozícióját egy mezőn belül nullától számítva:

```
Rekord értékeinek beállítása
Képernyő elkészítése. Információs szövegek, üres mezők.
Rekord mezőinek átvitele a Mezők tömb Tartalom mezőibe,
kijelzés

{ Adatcsoport szerkesztése: }
MezőIndex ← Kezdeti érték (első szerkesztendő mező indexe)
Ciklus
  { Aktuális mező szerkesztése: }
  Ciklus
    { Egy billentyű feldolgozása: }
    Pozicionálás a képernyőn
    Billentyű olvasása

    Ha Befejező billentyű, akkor
      Esetleges ellenőrzések, figyelmeztetések, állítások
    Egyébként { Nem befejező billentyű }
      Elágazás
        { Vezérlő billentyűk: }
        ←: Ha Poz > 0, akkor Poz csökkentése
        →: Ha Poz < Akt. hossz, akkor Poz növelése
        Home: Poz ← 0
        End: Poz ← Aktuális hossz
```

```
            Del: Ha Poz < Akt. hossz, akkor karakter
              törlése, kijelzés
            Backspace: Ha Poz > 0, akkor karakter törlése,
              kijelzés
            Ins: Beszúrómód ← Not Beszúrómód
          Egyébként
            { Megjeleníthető billentyűk: }
            Ha Bevihető billentyű akkor
              Ha Poz < Hossz, akkor
                Beszúrómódtól függően karakter betétele a
                mezőbe, kijelzés
              Elágazás vége { Poz < Hossz }
            Elágazás vége { Bevihető billentyű }
          Elágazás vége { Vezérlő billentyű }
        Elágazás vége { Befejező billentyű }
      mígnem Befejező billentyűt ütöttek (Enter, Esc, Tab,
      Shift-Tab, le vagy fel billentyű)
      Ciklus vége

      Ha Nem adatcsoport kilépő billentyűt ütöttek, akkor
        Mezőindex állítása
      Elágazás vége
    mígnem Adatcsoport kilépő billentyűt ütöttek (Enter vagy
    Esc)
    Ciklus vége

    Ha Adatcsoport elfogadva (Enter), akkor
      Mezők tömb Tartalom mezőinek átvitele a Rekord mezőibe
    Elágazás vége
```

Kérdések

1. Mit jelentenek a következő fogalmak?
   a) Rekord, rekord mező
   b) Egymásba ágyazott rekord
   c) Mező minősítése
   d) Változó rekord, szelektor mező
   e) Rekord konstans
   f) Felhasználói illesztő
   g) Csoportos adatbevitel
   h) Adatcsoport szerkesztése, mező szerkesztése
2. Mit csinál a With utasítás?
3. Hogy lehet egy rekordnak a deklarációs részben kezdőértéket adni?

Feladatok

A *-gal jelzett feladatok megoldásai a *Megoldások* című függelékben megtalálhatók.

1. Állítson össze egy dátum rekord típust, mely tartalmazza az évet, a hónapot és a napot!
   a) Készítsen egy *ElőbbiDátum* függvényt, mely összehasonlít két dátumot, és a visszaadott érték *True*, ha az első dátum előbbi, mint a második!
   b) Készítsen egy *EgyenlőDátum* függvényt is!
   c) Írjon egy eljárást, mely egy adott dátumot egy nappal megnövel!
2. Definiáljon egy *Babák* adatszerkezetet, mely babák születési adatait tartalmazza:
   - Név
   - Anyja neve
   - Születési dátum (év, hó, nap)
   - Pontos idő (óra, perc, másodperc)

   Töltse fel a *Babák* adatszerkezetet. Minden baba után kérdezze meg, van-e még felvitelre váró baba!
   Írja ki a babák adatait születési dátum+idő szerint rendezetten!
3*. Állítson össze egy adatszerkezetet, mely memóriában tárolja maximum 100 tanuló következő adatait:
   - Név
   - Félévi osztályzatok az egyes tantárgyakból (tantárgyanként egy osztályzat)
   - Tantárgyak, melyekből a tanuló felvételizni akar

   A lehetséges tantárgyakat a program rendezetten tárolja, azok nem változtathatók. Egy tanulóhoz legfeljebb 5 felvételi tárgyat lehet tárolni.
   a) Hány byte-ot foglal le összesen a tanulókat tároló adatszerkezet a memóriában?
   b) Olvassa be a tanulók adatait! A beolvasásnak akkor van vége, ha a tanuló nevénél nem írnak be semmit. Az egyes tanulók felvételi tárgyait „*" végjelig olvassuk!
   c) Írja ki a tanulók adatait!

   A rekordok lehetőleg minimális helyet foglaljanak el, vagyis a felvételi tantárgyakat kódolva tárolja – legyen a tárolt kód a tantárgy sorszáma!
   - Írjon függvényt, mely visszaadja a megadott tantárgy kódját!
   - Írjon függvényt, mely visszaadja a megadott kódhoz tartozó tantárgy nevét!
4. Állítson össze egy rekordot, mely egy személyről a következő adatokat tartja nyilván:
   - Név
   - Nem (Férfi, Nő)
   - Születési dátum
   - Családtagok száma
   - Családi állapot (Házas, Elvált, Egyedülálló, Özvegy)

- A következő adat a családi állapottól függ:
  Házas: a házastárs neve;
  Elvált: válás dátuma;
  Egyedülálló: semmi;
  Özvegy: házastárs elhalálozásának dátuma.

Vegyen fel egy tömböt, mely maximum 50 személy adatát tárolja. Olvasson be adatokat addig, amíg névnél egy végjelet nem ütnek be.

a) Írja ki az összes elvált asszony nevét és korát kor szerint növekvőleg rendezve!
b) Írja át egy másik tömbbe azokat a személyeket, akik 30 évnél fiatalabbak, és a családtagok száma nem több, mint 2!

5. A fenti feladatok bármelyikét úgy oldja meg, hogy a bevitel párbeszédablak segítségével történjen!

Érdemes tanulmányozni

Angster Erzsébet-Kertész László: Turbo Pascal feladatgyűjtemény I.:
Tömbök fejezet, Tomb14 program (Mezőszerkesztés)

2. ÁLLOMÁNYOK

Egy külső állomány (external file) valamely másodlagos tárolóeszközön helyezkedik el, legtöbbször lemezen. Az ide kiírt adatok a program futásának végeztével megmaradnak – azokat legközelebb is lekérdezhetjük, újabb adatokat írhatunk hozzájuk stb. A lemezre vitt adatokat egy másik számítógépre is átvihetjük, ott felhasználhatjuk, módosíthatjuk. Ebben a fejezetben egy általános áttekintést kap az Olvasó az állományokról, azok szervezéséről és feldolgozásairól. Megvizsgáljuk, hogy egy felhasználó logikailag hogyan „látja" az adatokat, és azt is, hogy ezek az adatok fizikailag hogyan helyezkednek el a külső tárolóeszközön. A fejezet végén áttekintjük a Turbo Pascal által nyújtotta lehetőségeket.

Eddig az adatokat kizárólag memóriában tároltuk. Adatok megőrzésére azonban a memória nem alkalmas a következők miatt:

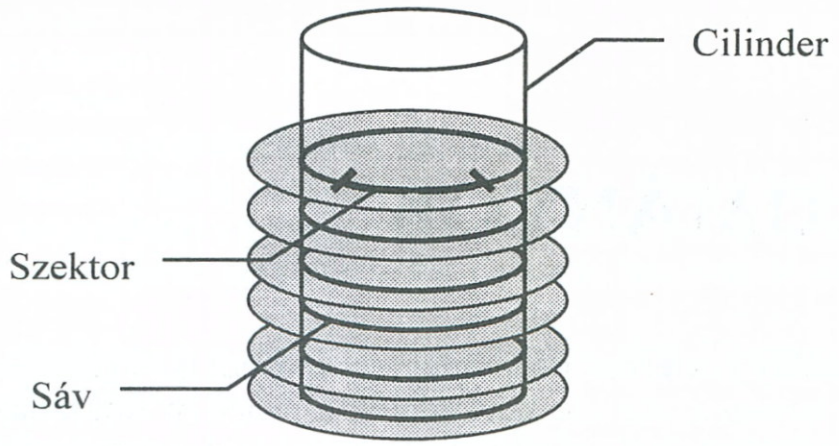
- A programban tárolt adatok a program futásának végeztével mindenképpen elvesznek, de ez bekövetkezhet valamilyen nem várt esemény (pl. programhiba, áramkimaradás, számítógép kikapcsolása) hatására is.
- Az adatok tárolásához szükséges memória nagysága általában többszörösen meghaladja a rendelkezésre álló memória nagyságát.

A bevitt adatokra később is szükségünk lehet – az áruházban forgalmazott áruk adatait nem csak a bevitel napján, hanem esetleg évekig használni szeretnénk. Az adatok tárolását biztonsággal kell megoldanunk.

2.1 Fizikai állomány, fizikai rekord

*Fizikai állomány*nak nevezzük a másodlagos tárolón elhelyezett adatok önálló névvel ellátott halmazát. Ez az önálló név lemezen az állományspecifikáció.

*Másodlagos tár*nak (külső tár, háttértár) nevezzük azokat a számítógéphez kapcsolt egységeket, amelyek alkalmasak adatok és programok hosszú távú tárolására. Vannak címezhető, illetve nem címezhető háttértárak. A nem címezhető tárakon tárolt adatokat csak sorban lehet feldolgozni, míg a címezhető tárak adatait direkt módon el lehet érni. A leggyakrabban használt címezhető külső tároló a mágneslemez, mely fizikai felépítése a következő: Egy lemezkötet (volume) egy vagy több egymás felett álló lemezből áll. Címzés szempontjából azonban a lemezkötet cilinderekből, a cilinderek sávokból, a sávok pedig szektorokból állnak. Az író/olvasófejek a lemezek között helyezkednek el. Egy cilindert a fizikailag egymás felett elhelyezkedő sávok alkotják, melyek az író/olvasófejek mozgása nélkül egyszerre elérhetők.



A lemezre való fizikai írás, illetve olvasás egysége a *fizikai rekord*, vagy más néven *blokk*. Ennél kevesebb adatot a ki-, illetve bevitelnél nem lehet kezelni. A fizikai állomány fizikai rekordokból áll. A fizikai rekord nem tévesztendő össze az előző fejezetben tárgyalt rekorddal, mely egy logikai egységet alkot. Egy fizikai rekordba legtöbbször több logikai rekordot helyeznek el – ezt nevezik *blokkolás*nak, a logikai rekordok számát a fizikai rekordon belül a *blokkolási tényező* adja meg. Amikor a lemezre adatokat írunk, akkor a tárnak egy fizikai címét kell megadnunk akár közvetlenül, akár közvetett módon. A cím megadását lényegében a következő címzési módok egyikével tehetjük meg:

- *Abszolút címzés:* Megadandó a lemezegység azonosítója, a cilinder, a sáv és a szektor fizikai címe.
- *Relatív címzés:* Megadandó egy sávcím és azon belül a fizikai rekord relatív sorszáma.
- *Logikai címzés:* Megadandó a fizikai állományon belül a fizikai rekord relatív sorszáma.

Az abszolút címet a bemeneti/kimeneti vezérlő rendszer határozza meg bármelyik címzési módból.

2.2 Logikai állomány, logikai rekord

Egyed, egyedtípus

Az egyed adott tulajdonságokkal bíró „valami", mely az információfeldolgozás tárgyát képezheti, és mely egyértelműen azonosítható – például egy személy, egy étel stb. Az egyed fogalma egy absztrakció eredménye. Az egyedeket egy adott feladat kapcsán állapítjuk meg és osztályozzuk. A konkrét egyedet *egyed-előfordulás*nak vagy egyszerűen *egyed*nek, míg annak általánosítását *egyedtípus*nak nevezzük. A *Személy* egyedtípus előfordulásai például az *Olvasó* és *én*, az *Étel* egyedtípus előfordulásai a *Töltöttkáposzta*, *Mákosguba* stb.

Tulajdonság, tulajdonságtípus

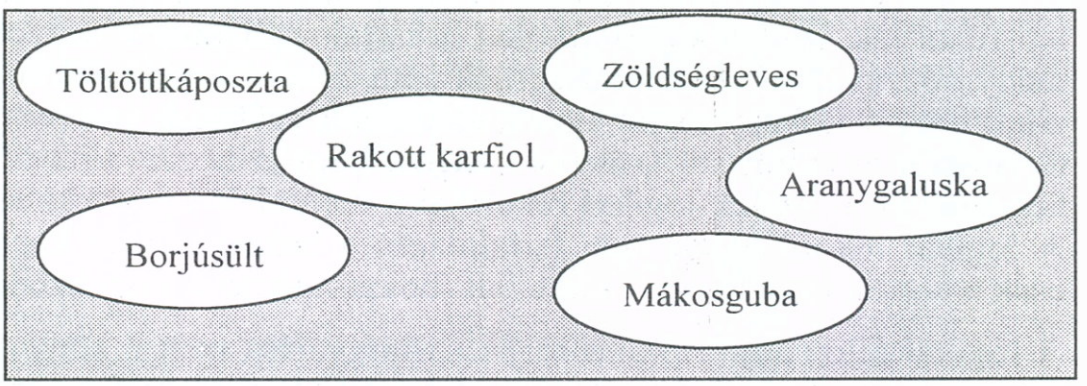
Az egyedek *tulajdonság*okkal rendelkeznek – például egy személynek van személyi száma, neve, végzettsége stb. A most felsorolt azonosítók tulajdonságtípusok. A *Személyi szám* előfordulásai a *25901019965*, *16612010547* stb., a *Végzettség* előfordulásai például az *Egyetem*, *Főiskola*.

Logikai állomány

A logikai állomány egyed-előfordulások önálló névvel ellátott halmaza, mely olyan tulajdonságtípusok előfordulásait tartalmazza, mely egy adott feladat szempontjából lényeges. Például az *Étel* logikai adatállomány konkrét ételeket tartalmazhat. Elképzelhető, hogy egy étterem számítógépen szeretné nyilvántartani ételeinek receptjeit, jellemzőit. A szakács szerint az ételek legfontosabb tulajdonságtípusai többek között a következők:

- Étel neve
- Hozzávalók
- Elkészítési idő
- Elkészítés pontos leírása
- Előállítási ár
- Eladási ár
- stb.

Étel
logikai állomány



Logikai rekord

Az egyed-előfordulásokat logikai rekordokkal írjuk le. A feladattól függően ugyanahhoz az egyedhez más és más logikai rekord is tartozhat. Egy kórházat például valószínűleg nem az ételek receptjei érdeklik, hanem azok kalóriamennyisége és az, hogy milyen betegek fogyaszthatják. A felhasználó általában egyszerre egy logikai rekordot szeretne látni. A logikai rekord adattételekből és adatcsoportokból áll.

Adattétel (elemi adat)

Az adattétel logikailag elkülöníthető, önálló elnevezéssel rendelkező legkisebb adat. Egy logikai adatstruktúra adattételekből építkezik. Az adattétel a felhasználó szempontjából tovább nem bontható tulajdonságtípus konkrét előfordulása. Például az ételek tulajdonságai közül az *Étel neve*, vagy az *Elkészítési idő*.

Adatcsoport

Az adatcsoport adattételek olyan névvel ellátott együttese, melyek logikailag egymáshoz kapcsolódnak. Például: *Hozzávalók = 1. hozzávaló, 2. hozzávaló* stb., vagy *Születési dátum = év, hónap, nap*.

Logikai rekordazonosító

A logikai rekordazonosító a logikai rekord azon tulajdonsága, mely az egész állományban csak egyszer szerepel. A rekordokat az azonosító különbözteti meg egymástól, az azonosító a rekordokat azonosítja. A logikai rekordazonosító lehet egy adattétel, de lehet egyéb, a rekordhoz rendelt egyértelmű érték is. Ilyen például az *Étel neve*. Az ételt az elkészítési idő nem azonosítja.

A logikai állomány a felhasználó igényeit elvben kielégíti. Megtervezésénél a hardver és szoftver körülményeket figyelmen kívül hagyjuk. Összegyűjtjük azokat az adatokat, melyekre a felhasználónak szüksége lehet, és meghatározzuk azokat az igényeket, melyeket az állománynak ki kell tudni elégíteni. Ezek az igények két fő csoportba sorolhatók:

- *Karbantartás*: új adatok felvitele, régiek törlése, módosítása.
- *Lekérdezések*: az állományba felvitt adatok vagy adatcsoportok visszanyerése különböző szempontok szerint.

A logikai állomány az itt leírtaknál valójában sokkal bonyolultabb szerkezet. A logikai rekordok *logikai kapcsolat*okat is tartalmazhatnak – nézzük például az *Étel* állomány hozzávalóit. Ha a rendszernek nem csak a hozzávalók mennyiségére, hanem az adott hozzávalóhoz tartozó egyéb adatokra is szüksége van (egységár, beszerzési hely, C-vitamin tartalom stb.), akkor világos, hogy a hozzávalók külön egyedeket fognak képezni, és az ételek logikai rekordjai a megfelelő egyedekkel lesznek kapcsolatban. A logikai rekord ekkor nem tartalmazza a hozzávalókkal kapcsolatos összes adatot, csak a kapcsolatot létesítő adattételt (hozzávaló neve vagy azonosítója).

A logikai állománystruktúra részletes tárgyalása nem e könyv feladata. Nagyobb információs rendszerek készítéséhez komolyabb rendszerfejlesztési és állományszervezési ismeretekre van szükség.

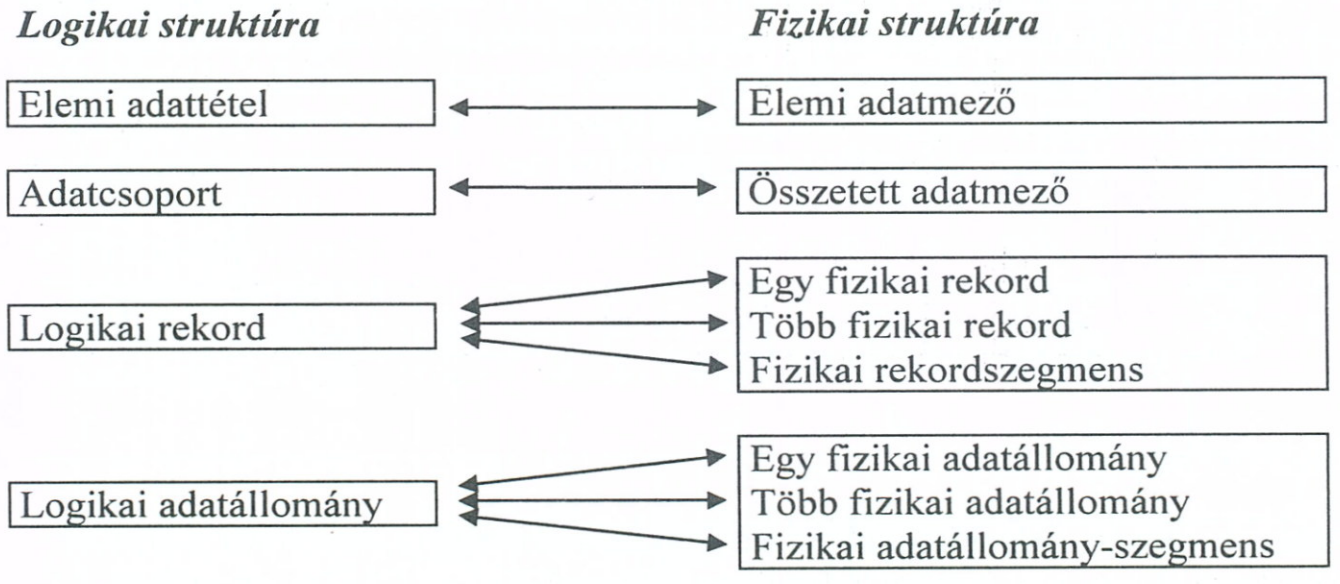
2.3 Logikai állomány leképezése fizikaira

A logikai adatállomány és a benne lévő kapcsolatok csak a képzeletben kezelhetők. Tegyük fel például, hogy a szakács ki szeretné válogatni a mintegy 500 ételből azokat, amelyeknek C-vitamin tartalma egy adott érték felett van. Ha az ételek adatai papíron vannak nyilvántartva, akkor a szakács valószínűleg le fog mondani az eredményről. Számítógép segítségével azonban a kérdéses ételek azonnal produkálhatók. Ahhoz, hogy a kívánt adatokat a felhasználó igényeinek megfelelően gyorsan „elő tudjuk halászni", ahhoz először is a logikai állományt le kell képezni a számítógép valamilyen fizikai tárolójára. S mivel a fizikai tárolásnak megvannak a maga lehetőségei és korlátai, ez a leképezés egy nagyon alapos meggondolás eredménye kell, hogy legyen.

A fizikai állományra való leképezést egy adott hardver, illetve szoftver környezetben végezzük. Mivel logikai szinten a felhasználó igényei az egyedekre és azok tulajdonságaira vonatkoznak, a leképezést az egyedek, illetve a logikai rekordazonosítók és valamilyen fizikai cím között kell elvégezni. Ha egy tár nem címezhető, akkor a megfeleltetésen nem kell sokat gondolkodni, hiszen a logikai rekordokat csak egyféleképpen, fizikai cím szerint növekvő sorrendben lehet elhelyezni, és azokat úgy is lehet visszaolvasni. Ekkor a rekordokat nem is fontos „beazonosítani". Ha viszont egy tár címezhető, akkor a lehetőségek tágak. Mivel azonban a fizikai címzés legjobb esetben is csak a rekordok sorszámára vonatkozhat, a rekordazonosítók pedig általában nem sorszámok, különféle „fondorlatokra" van szükség ahhoz, hogy a megfelelő adatokat kevés lemezművelet árán elérhessük. Egy leképezés alkalmával a célok a következők:

- A lemezműveletek számának, vagyis az író/olvasófej mozgásának minimálisra csökkentése.
- A foglalt lemezterület minimálisra csökkentése.
- A lemezhibákból eredő adatvesztések minimalizálása. Ha például az adatokat csak sorban, egymás után tudjuk elérni, akkor az első fizikai rekord meghibásodása miatt az egész állomány olvashatatlanná válik.
- A rendszer, illetve szoftverhibákból eredő adatvesztések minimalizálása. Ha például az adatok írása közben összeomlik a rendszer, akkor a „félig módosított" adatállomány komoly következményeket vonhat maga után.
- Hordozhatóság biztosítása. Például ha az állomány rekordjai fizikai címhez kötöttek, akkor az az állomány nem vihető át bármely más háttértárra.

A leképezés eredményeként nem biztos, hogy egy logikai állománynak pontosan egy fizikai állomány feleltethető meg. A gyors és ésszerű feldolgozás érdekében lehet, hogy a logikailag összetartozó dolgokat fizikailag szét kell vágni, vagy éppen ellenkezőleg, össze kell azokat vonni. A logikai struktúra és a fizikai struktúra közti megfeleltetést a következő ábra mutatja:



Egyszerűbb feladatoknál a logikai állomány és a fizikai állomány egybeesik. Sokszor a logikai rekord és a fizikai rekord is megfeleltethető egymásnak.

A logikai állomány alapján a fizikai állomány struktúrájának kialakítása, valamint a megfelelő karbantartó és lekérdező algoritmusok meghatározása az *állományszervezés* feladatkörébe tartozik.

2.4 Állomány szervezése

A logikai adatszerkezetet olyan fizikai adatszerkezetre kell leképezni, amely a felhasználói igényeket hatékonyan képes kielégíteni. A fizikai állomány szerkezetét elsősorban a hozzáférési igények határozzák meg. Hozzáférési igény például:

- A logikai állomány összes adatának elérése (feldolgozása) logikai sorrendtől függetlenül
- A logikai állomány összes adatának elérése egy adott logikai (rendezési) sorrendben
- Adott azonosítójú rekord elérése
- Adott tulajdonságú rekord vagy rekordok elérése

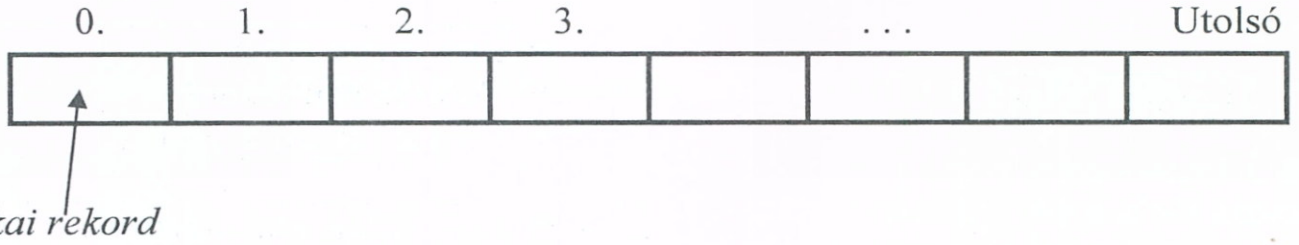
Ennek szellemében kell megtervezni az alapvető karbantartási funkciókat is (új rekord felvitele, régi rekord törlése, módosítása). A fizikai állomány szerkezetének kialakításakor lényeges szempont a rekordok logikai azonosítója és a tárolás fizikai címe közti összefüggés. Az állományok szervezésük szerint alapvetően az alábbi csoportokba sorolhatók:

Soros szervezés

A logikai rekordazonosító és a tárolási cím között nincs kapcsolat. Az állományban lévő adatokat csak a fizikai tárolás sorrendjében tudjuk felvinni, illetve elérni.

Direkt szervezés

A logikai rekordazonosító és a fizikai cím között kölcsönösen egyértelmű megfeleltetés van. Az állomány akármelyik elemét közvetlenül el tudjuk érni, azt írhatjuk és olvashatjuk. A rekord címét elvileg bármelyik címzési móddal megadhatjuk, de a szoftverek többsége a logikai címzést támogatja. Az állományon belüli relatív sorszámot vagy közvetlenül, vagy közvetve, egy előre definiált algoritmussal adjuk meg.



Például: időszámításunk kezdete óta jegyezni szeretnénk bizonyos adatokat – ezeket az adatokat egy-egy logikai rekordban helyezzük el. Az 1526. év adatait úgy tudjuk elérni, hogy egyszerűen beolvassuk az 1526. rekordot. Ha ugyanezt az adatgyűjtést csak századunkra szeretnénk elvégezni, akkor az algoritmus: Év-1900. Az 1995. év rekordja a 95. relatív sorszámon található. Ha a logikai rekordazonosító egy dátum, akkor ezeket a rekordokat például a lehetséges időintervallum napjainak sorszámára

lehet leképezni. Az algoritmus természetesen ennél sokkal bonyolultabb is lehet, de fontos, hogy a leképezés egyértelmű legyen, vagyis két különböző logikai rekordhoz ne rendeljük ugyanazt a sorszámot.

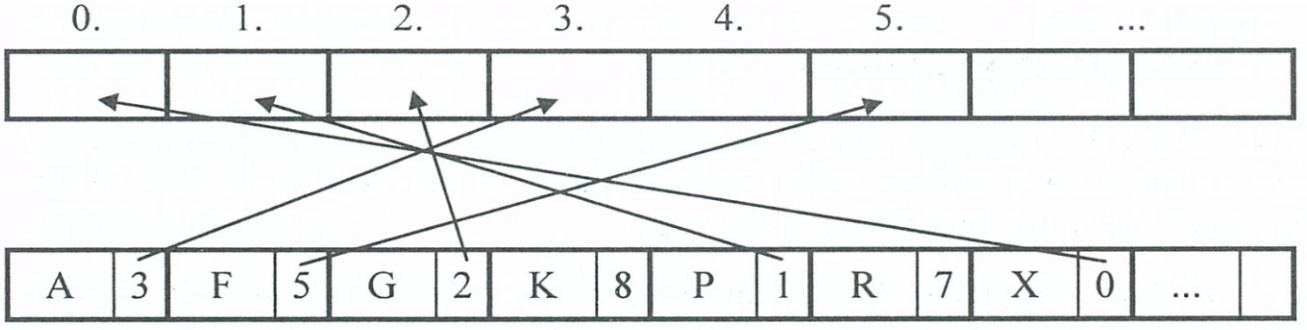
Véletlen szervezés

A rekordok azonosítói általában nem képezhetők egy jól meghatározott algoritmus szerint – azok legtöbbször véletlenszerűek. Vannak azonban olyan esetek, ahol az azonosítók szórása nem túl nagy, és lehet egy olyan algoritmust találni, mely a logikai rekordokat aránylag egyenletesen „teríti". Ha algoritmusunk egy második logikai rekordot véletlenül ugyanarra a helyre képezne le, akkor megfelelő korrekciós algoritmusokat vethetünk be, illetve a szóban forgó rekordokat ún. *túlcsordulási terület*re tehetjük. Az ilyen állományokat véletlen szervezésű állományoknak nevezzük. Példának vegyünk egy néhány száz fős személyzeti nyilvántartást. A logikai rekordokat azonosíthatja a személyi szám, melynek utolsó 4 számjegye várhatóan egyenletesen oszlik el, és feltehetően nem fordul elő sokszor, hogy két ember esetén ez a 4 szám megegyezik. A logikai rekordnak megfelelő fizikai rekordban természetesen jelezni kell, hogy ennek a rekordnak van még *szinonimája*, mely a túlcsordulási területen található. E szervezés egyik hibája, hogy túl sok lehet a fel nem használt lemezterület. A véletlen szervezésű állományokkal könyvünk nem foglalkozik.

Indexelt szekvenciális szervezés

A legtöbb feladat kapcsán a rekordokat közvetlenül el szeretnénk érni, vagyis egy adott azonosítójú rekordra minél kevesebb energia befektetésével rá szeretnénk pozicionálni. Általános felhasználói igény továbbá a rekordok valamilyen logikai sorrend szerinti elérése. A direkt vagy véletlen szervezés csak bizonyos kulcseloszlások esetén gazdaságos, és a megfelelő sorrend is nehezen állítható elő. Mesterséges sorrendet a következő módokon lehet például felállítani:

- Az állományt rendezzük
- A rekordokat indexekkel (mutatókkal) egészítjük ki, ahol a rekord mindig a logikailag őt követő rekordra mutat. Egy rekordnak elvileg egyszerre több mutatója is lehet, és így többféle logikai sorrend is felállítható.
- A rekordok azonosítóit kigyűjtjük, és egy külön állományban, vagy egyéb helyen tároljuk. Az azonosítókhoz szorosan hozzátartozik az „anyarekord" állománybeli indexe, vagyis relatív sorszáma. E sorszám szerint a logikai rekordok direkt módon elérhetők. Így egy rekord keresésekor az azonosítót először az indexállományban, illetve indextáblában keressük, és a megtalált azonosító melletti sorszám alapján a logikai rekord feldolgozható:



Az ilyen fajta állományszervezés a leggyakoribb, hiszen a rekordok azonosítói általában teljesen „kiszámíthatatlanok", azokban semmiféle törvényszerűség nem észlelhető. Ilyen például az *Étel neve*.

Az indexállomány lehet többszintű, ennek részletezésébe itt nem megyünk bele – erről az *„Adatszerkezetek"* fejezetben részletesebben szó lesz.

2.5 Hozzáférések (elérések)

Az állomány rekordjaihoz való hozzáféréseket (a rekordok eléréseit) a következőképpen osztályozhatjuk:

- *Soros hozzáférés*: A rekordok elérése a fizikai tárolás sorrendjében történik.
- *Szekvenciális hozzáférés*: A rekordok elérése egy logikai sorrend szerint történik (például rendezettség). A szekvenciális feldolgozhatóságot elérhetjük az állomány megfelelő szervezésével vagy valamilyen szoftver eszközzel, mint például rendezéssel.
- *Közvetlen hozzáférés*: A megadott rekordhoz közvetlenül hozzáférünk anélkül, hogy a rekordokat sorban be kellene olvasnunk. A rekordokhoz való közvetlen hozzáférést szintén szervezéssel biztosíthatjuk.

2.6 Pufferelés

A bevitel, illetve kivitel egysége, mint tudjuk, a fizikai rekord. A fizikai lemezművelet a memóriaműveletekhez képest rengeteg időt vesz igénybe különösen akkor, ha a beolvasásokat és kiírásokat kis egységekben oldjuk meg. Két okból is jobban járhatunk, ha egyszerre több fizikai rekordot olvasunk be, illetve írunk ki. Egyrészt azért, mert a legtöbb időt az író/olvasófej felgyorsulása és a megfelelő címre való ráállás veszi igénybe, másrészt pedig a lemezegység beviteli/kiviteli rendszere programunktól függetlenül, párhuzamosan működik, és így a lemezről való olvasás közben már feldolgozhatjuk az előzőleg beolvasott rekordot, illetve a lemezre írás közben már folytathatjuk is programunkat, és készíthetjük elő a következő kiírandó rekordot. A rekordok egy ún. pufferben „sorakoznak". Bizonyos file-kezelők megengedik a puffer méretének megadását, mások maguk határozzák azt meg. A puffer használata leginkább szekvenciális feldolgozásoknál érdekes, mert ott a következő rekord megjósolható, ellentétben a rekordok direkt feldolgozásaival, ahol a következő beolvasott rekordot sosem lehet „látni".

2.7 Állomány típusa

Az állományokat típusaik szerint is osztályozhatjuk. Az osztályozás aszerint történik, hogy az állományt milyen céllal hoztuk létre, mi annak a funkciója. Alapvetően a következő típusokat különböztetjük meg:

- *Törzsállomány:* A programrendszerben kulcsfontosságú adatállomány, mely viszonylag hosszú ideig él. A törzsállományt általában több feldolgozó prog-

ram is használja. Ilyen például egy áruház áruit nyilvántartó adatállomány. A törzsállományt napra kész állapotban kell tartani, azt folyamatosan aktualizálni kell (karban kell tartani) – új áruk jöhetnek az áruházba, régi áruk megszűnhetnek stb. A törzsállomány törzsadatokat tartalmaz.

- *Tranzakciós állomány:* A törzsállomány karbantartását lehet interaktív módon, a felhasználóval történő párbeszédes formában elvégezni, de lehet úgy is, hogy a karbantartási (tranzakciós) igényeket egy állományba összegyűjtve egy menetben végrehajtjuk. Az összegyűjtött tranzakciókat a tranzakciós állomány tartalmazza. A tranzakciós állomány tehát egy ideiglenesen létrehozott állomány, mely alapján egy törzsállományt aktualizálunk.
- *Tábla állomány:* Statikus adatokat tartalmaz, melyet csak ritkán módosítanak. Szótárnak is használható.
- *Riport állomány:* Szöveget tartalmazó állomány, például egy jelentés.
- *Kontroll állomány:* Olyan adatokat tartalmaz, mellyel a program futása ellenőrizhető.
- *Előzmény állomány:* Ezen állomány alapján a rendszer egy előző állapota visszaállítható.
- *Átmeneti vagy munkaállomány:* Olyan állomány, mely a programrendszer működése során ideiglenesen jön létre a feldolgozás elősegítésére.
- *Biztonsági állomány:* Másolat a programrendszer valamely fontos állományáról, mint például egy törzsállományról. A folytonosan változó állományokról rendszeresen kell biztonsági másolatot készíteni.

Egy állománynak szervezésén és típusán kívül egyéb jellemzői is vannak, mint például az állományra jellemző számadatok:

- *Állomány mérete*
- *Rekord mérete*
- *Elérési, feldolgozási idők* stb.

2.8 Turbo Pascal állományok

Állományok használatára a Turbo Pascal alapvetően három lehetőséget kínál:

Típusos állomány

Direkt szervezésű állomány. A kiírás, illetve beolvasás egysége a komponens. A komponensek egyforma típusúak, mely típus az állományra jellemző. A komponens hosszát a típus határozza meg. A komponensek sorszámozva vannak 0-tól. Akármelyik sorszámú elemre közvetlenül pozicionálhatunk, azt beolvashatjuk, illetve kiírhatjuk. A komponens helyét a rendszer a sorszám és a komponens hosszúsága alapján meg tudja határozni. A típusos állományt szekvenciálisan is feldolgozhatjuk a komponensek fizikai sorrendjében, vagyis a sorszámok szerint.

Típusnélküli állomány

Direkt szervezésű állomány. Abban különbözik a típusos állománytól, hogy itt a komponensek hosszúsága tetszőlegesen megadható, azt nem a típus határozza meg. Álta-

lában akkor használjuk, amikor csak az számít, hogy hány byte-ot írunk ki, illetve olvasunk be egyszerre – az adatok típusa érdektelen.

Szöveges állomány

Soros szervezésű állomány. A szöveges állomány sorokból, a sorok karakterekből és egy „sor vége" jelből áll. A kiírt, illetve beolvasott adatok változó hosszúságúak, azok fizikai címét nem lehet sorszám alapján megállapítani. Ezért a szöveges állományban található adatokat nem lehet direkt módon elérni.

Kérdések

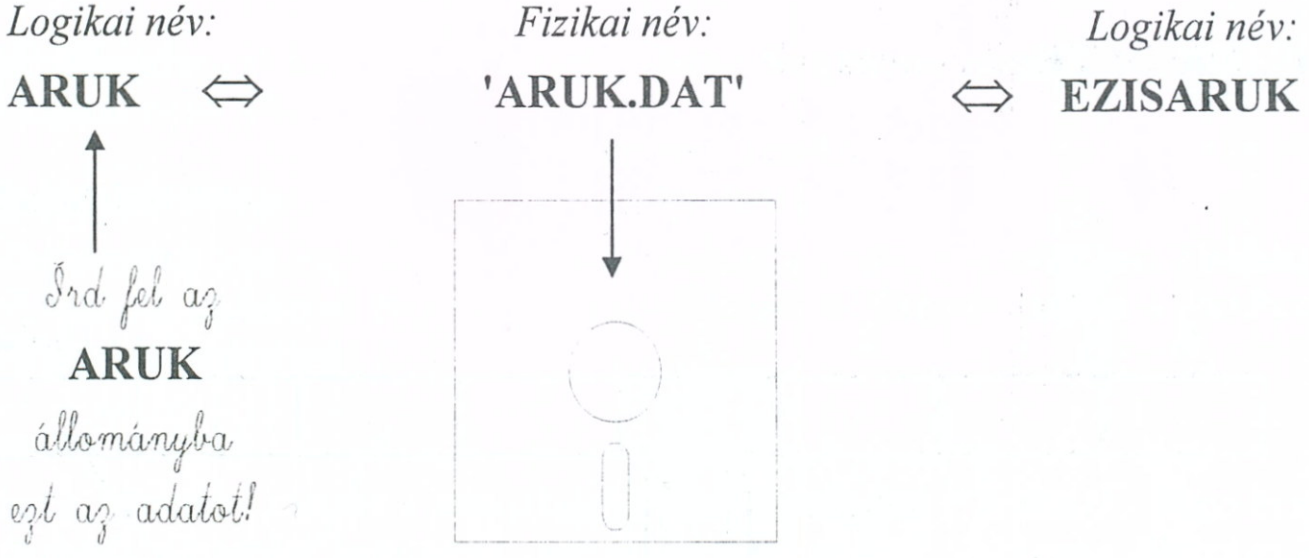
1. Mit jelentenek a következő fogalmak?
   a) Fizikai állomány, fizikai rekord
   b) Blokk, blokkolás, blokkolási tényező
   c) Másodlagos tár, címzési módok, cilinder, sáv, szektor
   d) Egyed, egyedtípus, tulajdonság, tulajdonságtípus
   e) Adattétel, adatcsoport
   f) Logikai állomány, logikai rekord, logikai rekordazonosító
   g) Karbantartás, lekérdezés
   h) Logikai állomány leképezése fizikai állományra
   i) Állomány szervezése
   j) Soros, direkt, véletlen, illetve indexelt szekvenciális szervezés
   k) Állomány hozzáférések
   l) Pufferelés
   m) Állomány típusa
   n) Törzsállomány, tranzakciós állomány, tábla állomány
   o) Riport állomány, kontroll állomány, előzmény állomány, átmeneti állomány, biztonsági állomány
   p) Típusos állomány, típusnélküli állomány, szöveges állomány

3. TÍPUSOS ÁLLOMÁNY

A Turbo Pascal legalapvetőbb állománytípusa a típusos állomány. Segítségével azonos típusú adatokat lehet lemezre írni, illetve onnan visszaolvasni. Megismerkedünk a típusos állomány felépítésével, majd az állományt írjuk, olvassuk, bővítjük. Ezután több állományt kezelünk egyszerre, és megnézzük, hogyan lehet egy adott komponenst megkeresni, illetve módosítani.

3.1 Fizikai név, logikai név

Minden külső állománynak van egy *fizikai neve*, mely egyértelmű, és a lemez katalógusában megtalálható. A fizikai név egy állományspecifikáció, tehát a név és kiterjesztés előtt tartalmazhatja a lemezegység és útvonal azonosítóját is. Ez utóbbiak hiányában az állomány az aktuális lemezegység aktuális katalógusára értendő. A programban az állománynak egy *logikai nevet* adunk, melyet összekapcsolunk egy fizikai névvel. Az állományokkal kapcsolatos műveleteknél mindig a logikai névre hivatkozunk, de a művelet az összekapcsolás miatt mindig a megadott fizikai állományra vonatkozik. A logikai név használata azért is előnyös, mert a fizikai név változása esetén a programban csak az összekapcsolást végző utasítást kell kicserélni.



A logikai állománynév a program egy azonosítója (állományváltozó):

```
Var
  Aruk : ...
```

A logikai név megadása független a fizikai névtől, mégis célszerű azokat hasonlóra választani (Aruk és 'Aruk.Dat').

Az *állományt* még használat előtt *meg kell nyitni*. Ekkor a program ellenőrzi a fizikai állományt, és megteszi a szükséges előkészítő lépéseket az íráshoz, illetve olvasáshoz. Az állományt használat után *le kell zárni*. Ennek hatására a program „befejezi" az állományt, aktualizálja a katalógust.

A program minden állományváltozóhoz egy memóriaterületet rendel, ahol jegyzi az állomány adatait (például fizikai nevét) és aktuális állapotát (nyitva van, zárva van, állomány pozíciója stb.). E területet a programozó közvetlenül nem módosíthatja. Ahhoz, hogy egy fizikai állományon dolgozzunk, mindenekelőtt el kell végezni a logikai név - fizikai név hozzárendelést:

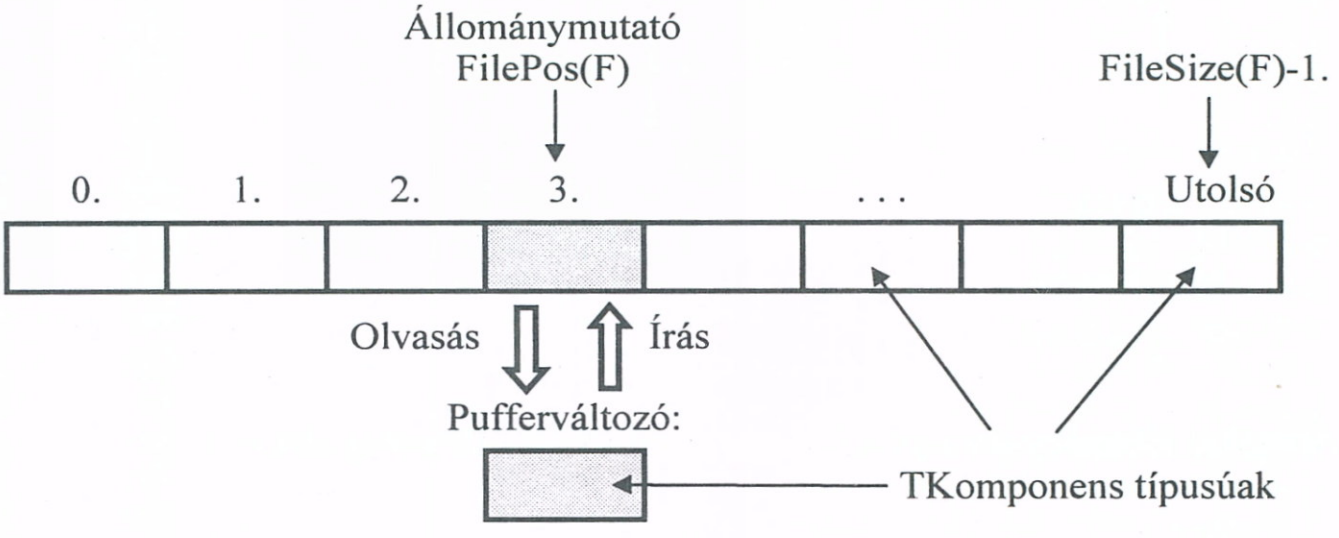
```
Assign(Aruk,'Aruk.Dat') ;
```

Ettől kezdve a programban csak a logikai névre hivatkozunk (Aruk). A hozzárendelés csak egy memóriabeli értékadás, az állomány nyitásakor már konkrét lemezműveletek is történnek.

3.2 Típusos állomány felépítése

A típusos állomány direkt szervezésű állomány. Egyforma típusú komponensekből áll, melyeket írni is, olvasni is lehet. Az írás és olvasás egysége a komponens. A komponensek sorszámozva vannak, az első komponens sorszáma 0. A típusos állomány felépítését a 3.1. ábra mutatja. A *komponens típus*át (alaptípust) az állomány deklarálásánál meg kell adni.

```
Var
   F : File Of TKomponens ;
```



3.1. ábra. Típusos állomány felépítése

A komponens típusa meghatározza annak hosszát. Ha például TKomponens = Integer, akkor az állomány 2 byte-os, Real esetén 6 byte-os egységekből áll. A típusos állománynak a Turbo Pascal-ban – ha a lemez mérete engedi, – több, mint 2 milliárd komponense lehet (az állomány mutatója LongInt típusú).

A programban meg kell adni legalább egy *puffervált*ozót, melybe a megfelelő komponenst beolvassuk, illetve azt onnan kiírjuk. A pufferváltozó típusa meg kell, hogy egyezzen a komponensek típusával. Például:

```
Var
   Egeszek : File Of Integer ;
   Egesz : Integer ;

   Valosak : File Of Real ;
   Valos : Real ;
```

A típusos állomány alaptípusa szinte bármelyik Pascal típus lehet – például egész, karakterlánc, tömb vagy rekord. Egy szintaktikai szabályt azonban be kell tartani: az állomány típusának és a változónak, melybe a komponenst majd beolvassuk, *azonos típusú*aknak kell lenniük. Ebből következik, hogy az állomány típusát előre kell definiálni:

```
Type
   TKomponens = ... ;

Var
   F : File Of TKomponens ;
   PufferValtozo : TKomponens ;
```

Minden állománnyal kapcsolatos eljárás, illetve függvény első paramétere az állomány logikai neve (F).

Állomány mutatója

Az állománynak van egy *LongInt* típusú mutatója, mely tetszőlegesen állítható a *Seek(F,Mutató)* eljárással. Nyitáskor a mutató értéke 0. Aktuális értéke a *FilePos(F)* függvénnyel lekérdezhető.

Állomány mérete

A típusos állomány mérete a komponensek száma, mely a *FileSize(F)* függvénnyel lekérdezhető. Az utolsó komponens sorszáma *FileSize(F)-1*. A típusos állomány mérete a lemezen a komponensek számának és egy komponens méretének a szorzata: *FileSize(F) * SizeOf(TKomponens)* byte.

Állomány nyitása

Nyitás előtt az állományváltozóhoz hozzá kell rendelni a fizikai nevet. A típusos állományt kétféleképpen nyithatjuk meg:

A *Rewrite(F)* eljárással egy új állományt hozunk létre. Ha ilyen nevű állomány már létezett a lemezen, akkor azt az eljárás törli. Nyitás után az új állomány üres, mutatójának értéke 0:

```
Assign(F,' ... ') ;
Rewrite(F) ;
```

A *Reset(F)* egy már létező állományt nyit meg. Futási hibát eredményez, ha nincs ilyen nevű állomány a lemezen. Az állomány mutatója kezdetben 0:

```
Assign(F,' ... ') ;
Reset (F) ;
```

Állomány lezárása

Az állományt feldolgozás után a *Close(F)* eljárással zárhatjuk le.

Megsérülhet az állomány, ha azt megváltoztatjuk, és a program befejezése előtt nem zárjuk le!

Olvasás

A típusos állományból a Read eljárással olvashatunk be egy komponenst: *Read(F, Pufferváltozó)*. Az eljárás azt a komponenst olvassa be *Pufferváltozó*ba, amelyiken az állomány mutatója áll. A megadott pufferváltozó típusának meg kell egyeznie az állomány alaptípusával. Beolvasás után az *állomány mutatója eggyel tovább lép*, elősegítve ezzel a szekvenciális feldolgozást.

Ha a mutató nem létező elemen áll, olvasási hiba következik be.

Írás

A típusos állományba a Write eljárással írhatunk ki egy komponenst: *Write(F, Pufferváltozó)*. Az eljárás *Pufferváltozó* tartalmát kimásolja a lemezre arra a helyre, ahol az állomány mutatója áll. Kiírás után az *állomány mutatója eggyel tovább lép*, elősegítve ezzel a szekvenciális feldolgozást.

Ha egy elemet módosítani szeretnénk, akkor az elem kiírása előtt vissza kell pozicionálni, mert olvasásnál a mutató továbblép.

Állomány vége

Ha az állomány mutatója az utolsó komponens után áll, akkor vége van az állománynak. Ekkor az *Eof(F)* Boolean típusú függvény *True* értéket ad vissza.

3.3 Létrehozás, szekvenciális írás

Feladat
Készítsünk nyilvántartást egy áruházban forgalmazott árukról! Az egyes árukhoz a következő adatok tartoznak:

- Árukód pontosan 4 karakter
- Leírás maximum 20 karakter
- Egységár valós

Vigyük fel lemezre az áruk adatait – a felvitelnek akkor legyen vége, ha az árukódnál nem ütnek be semmit!

A szekvenciális felvitel általános algoritmusa a következő:

```
Állomány nyitása
Ciklus amíg még akarunk komponenst kiírni
   Komponens összeállítása
   Komponens kiírása az állományba
Ciklus vége
Állomány zárása
```

Feltételezzük, hogy még nincsen árukat nyilvántartó állományunk, tehát azt most hozzuk létre. A rekordok mezőit terminálról kérjük be. Ha az árukód végjelet tartalmaz, akkor nem írunk fel több rekordot az állományba. Ezért az összeállításhoz a kódot előolvassuk. Az összeállított rekordokat sorban felvisszük a *Write* utasítással. Mivel az állománymutató 0-ról indul és minden írás után eggyel továbblép, a rekordok sorszámai rendre 0, 1, 2 stb. lesznek. A Pascal kód a következő:

```
Program Letrehoz ;
Type
   TKod = String[4] ;
   TAru = Record
      Kod : TKod ;
      Leiras : String[20] ;
      Egysegar : Real ;
   End ;
Var
   Aruk : File Of TAru ;
   Aru : TAru ;
Begin
   Assign(Aruk,'Aruk.Dat') ;
   Rewrite(Aruk) ;
   Write('Árukód:') ; ReadLn(Aru.Kod) ;
   While Aru.Kod <> '' Do
      Begin
         Write('Leírás:') ; ReadLn(Aru.Leiras) ;
         Write('Egys.ár:') ; ReadLn(Aru.Egysegar) ;
         Write(Aruk,Aru) ;
         Write('Árukód:') ; ReadLn(Aru.Kod) ;
      End ;
   Close(Aruk) ;
End.
```

3.4 Szekvenciális olvasás

Feladat
Listázzuk ki az előző pontban létrehozott 'Aruk.Dat' állomány tartalmát!

Amikor egy típusos állományt olvasni szeretnénk, tudnunk kell, hogy az állomány milyen típusú komponensekből áll. Az elemeket általában ugyanolyan típusú változóba akarjuk beolvasni, mint amilyenből azt annak idején kiírtuk. A lemezes állomány csupán egy byte-halmaz, íráskor a memória pontos képe kerül ki a lemezre. A lemezen lévő byte-sorozatnak nincs típusa, azt a program „húzza rá" az adatokra. Ha az adatokat más típusú memóriaterületre töltjük rá, mint amilyenek azok kiíráskor voltak, akkor először is furcsa dolgok történhetnek, másodszor pedig az állomány utolsó rekordjánál a program lemezolvasási hibával le fog állni, ha nem találja a kért adatmennyiséget.

Az állományt most a *Reset* eljárással nyitjuk meg, mert egyébként az adatok törlődnének.

A teljes állományt kétféleképpen listázhatjuk ki a komponensek fizikai sorrendjében. Első megoldásként addig olvasunk, amíg az állomány végére nem érünk. Ciklusunk tehát addig megy, míg *Eof(Aruk)* értéke *False*:

```
Program Kiir ;
{ Deklaráció ugyanaz, mint létrehozásnál. }
Begin
  Assign(Aruk,'Aruk.Dat') ;
  Reset(Aruk) ;
  While Not Eof(Aruk) Do
    Begin
      Read(Aruk,Aru) ;
      { A #9 karakterrel tabulálunk a képernyőn: }
      WriteLn(Aru.Kod,#9,Aru.Leiras,
        #9,Aru.Egysegar:8:2) ;
    End ;
  Close(Aruk) ;
End.
```

Második megoldásként az elemeket *For* ciklussal olvassuk be, hiszen tudjuk, hogy pontosan *FileSize(F)* darab elem van az állományban:

```
For Mutato := 1 To FileSize(Aruk) Do
  Begin
    Read(Aruk,Aru) ;
    WriteLn(Mutato:2,'. ',Aru.Kod,
      #9,Aru.Leiras,#9,Aru.Egysegar:8:2);
  End ;
```

Ha kihasználjuk, hogy a ciklusváltozó egyben az állomány mutatója is, akkor a növekményes ciklus *0*-tól *FileSize(F)-1* -ig megy!

3.5 Bővítés

Egy típusos állományt úgy tudunk bővíteni (ahhoz elemeket hozzáírni), hogy az utolsó komponens utáni helyre pozicionálunk, és onnan folytatjuk a szekvenciális írást. Írjunk fel az 'Aruk.Dat' állományba egy utolsó rekordot:

```
...
Seek(Aruk,FileSize(Aruk)) ;
With Aru Do
  Begin
    Kod := 'Z999' ;
    Leiras := 'Utolsó' ;
    Egysegar := 0 ;
  End ;
Write(Aruk,Aru) ;
```

Megtehetjük, hogy jóval az utolsó elem után pozicionálunk, és felírunk egy elemet. Ekkor az állomány a megfelelő méretre bővül. Ha előre lefoglaljuk az állomány számára a várható lemezterületet, akkor az állomány a későbbiekben nem lesz „felszabdalva".

Arra vigyázni kell, hogy ne olvassunk olyan helyről, ahová előzőleg nem írtunk ki semmit. Az ilyen komponensekben „szemét" van!

3.6 Direkt elérés

A típusos állomány akárhányadik elemére pozicionálhatunk, az elemet onnan beolvashatjuk. A következő mintaprogramban terminálról sorban bekérünk számokat, és kiírjuk az azon a pozíción található rekord mezőit. A ciklus addig megy, amíg létező pozíciót ütnek be. Tájékoztatásul kiírjuk az állomány méretét:

```
Reset(Aruk) ;
WriteLn('Méret: ',FileSize(Aruk)) ;
Write('Pozíció?') ; ReadLn(Pozicio) ;
While (Pozicio >= 0) And (Pozicio < FileSize(Aruk)) Do
  Begin
    Seek(Aruk,Pozicio) ;
    Read(Aruk,Aru) ;
    With Aru Do WriteLn(Kod,#9,Leiras,#9,Egysegar:8:2);
    Write('Pozíció?') ; ReadLn(Pozicio) ;
  End ;
Close(Aruk) ;
```

3.7 Állomány létezésének vizsgálata

Sokszor van szükség arra, hogy egy adott állományról megállapítsuk, létezik-e egyáltalán. Normális esetben, ha egy nem létező állományt próbálunk *Reset*-tel megnyitni, a program futási hibával leáll. Ha kikapcsoljuk az input/output ellenőrzést ($I- fordítási direktíva), akkor a program hiba esetén is fut tovább – ekkor azonban az *IOResult* függvénnyel kötelezően le kell kérdeznünk a művelet helyességét. Bővítsük most az *Aruk.Dat* állományt. Ha nem létezik, akkor előbb hozzuk létre:

```
Assign(Aruk,'Aruk.Dat') ;
{$I-} Reset(Aruk) ; {$I+}
If IOresult = 0 Then
  Seek(Aruk,FileSize(Aruk))
Else
  Rewrite(Aruk) ;

{ Áruk szekvenciális felvitele }
```

3.8 Két vagy több állomány egyszerre

A programból egyszerre több állományt is kezelhetünk. Minden egyes állománynak adnunk kell egy logikai nevet, melyekhez egy-egy fizikai nevet rendelünk. A program mindegyik logikai névhez lefoglal egy memóriaterületet, melyben az ahhoz az állományhoz tartozó információkat „jegyzi". Minden állomány „tudja magáról", hogy éppen nyitva van-e vagy zárva, hol tart az írásban, illetve olvasásban stb. Az egyszerre nyitva tartható állományok számának az operációs rendszer szab határt.

Feladat
Válogassuk szét az előzőekben létrehozott árukat tartalmazó állományt: az 'Olcsok.Dat' állományba kerüljenek a 100 Ft-nál olcsóbb áruk, a 'Draga.Dat'-ba pedig a többi!

```
Procedure Szetvalogat ;
  Var
    Aruk, OlcsoAruk, DragaAruk : File Of TAru ;
    Aru : TAru ;
  Begin
    Assign(Aruk,'Aruk.Dat') ;
    Assign(OlcsoAruk,'Olcsok.Dat') ;
    Assign(DragaAruk,'Dragak.Dat') ;
    Reset(Aruk) ;
    Rewrite(OlcsoAruk) ;
    Rewrite(DragaAruk) ;
```

```
    While Not Eof(Aruk) Do
      Begin
        Read(Aruk,Aru) ;
        If Aru.Egysegar < 100 Then
          Write(OlcsoAruk,Aru)
        Else
          Write(DragaAruk,Aru) ;
      End ;
    Close(Aruk) ;
    Close(OlcsoAruk) ;
    Close(DragaAruk) ;
End ;
```

3.9 Állomány törlése, átnevezése

Amikor egy nagyobb állományt átalakítunk – például rendezzük, vagy elemeit törlés miatt lejjebb toljuk –, ajánlatos a régi változatot az átalakítás végeztéig megtartani. Előfordulhat ugyanis, hogy egy nem várt esemény miatt az algoritmus megszakad, és az új állomány *még*, az eredeti *már* nem használható. Ha az átalakítás rendben befejeződött, kitöröljük a régi állományt, majd átnevezzük az újat az eddig megszokott névre. Az új állomány tehát egy másik lemezterületen keletkezik, és hiba esetén is egészen biztos, hogy legalább a régi megmarad. Törölni, illetve átnevezni csak lezárt állományokat szabad:

```
Assign(Regi,'Regi.Dat') ;
Assign(Uj,'Uj.Dat') ;
...
Close(Regi) ;
Close(Uj) ;
Erase(Regi) ;                { Állomány törlése }
Rename(Uj,'Regi.Dat') ;      { Állomány átnevezése }
```

3.10 Keresés a rendezetlen állományban

Egy típusos állományban a komponenseket véletlenszerűen csak sorszámuk szerint lehet elérni. Egy konkrét adat megkereséséhez hasonló keresési módszereket kell alkalmaznunk, mint tömbök esetén. Ha az állomány rendezetlen, akkor szekvenciálisan keresünk. Minden esetben végig kell gondolni, hogy a megadott tulajdonságú elemből hány lehet az állományban. Ha csak egy van, vagy minket csak az első ilyen érdekel, akkor az elem megtalálása után abbahagyjuk a keresést. Több elem esetén a keresést az állomány végéig folytatjuk.

A *Van* függvény egy árukat tartalmazó rendezetlen állományban megkeresi az első, megadott kódú árut. A kódot paraméterként adjuk meg. A függvény visszatérési értéke

logikai, mely igaz, ha van az állományban ilyen kód – ekkor az áru többi adatát az *Aru* rekordban, a rekord állománybeli pozícióját a *Poz* változóban kapjuk vissza:

```
Function Van(Kod:TKod; Var Aru:TAru; Var Poz:LongInt):
  Boolean ;

  Var
    Ok : Boolean ;
  Begin
    Seek(Aruk,0) ;
    Ok := False ;
    Poz := -1 ;
    While Not Eof(Aruk) And Not Ok Do
      Begin
        Inc(Poz) ;
        Read(Aruk,Aru) ;
        Ok := Kod = Aru.Kod ;
      End ;
    Van := Ok ;
  End ;
```

- Ha az állomány rendezett, akkor ne keressünk az állomány végéig! Nagy, rendezett állomány esetén egyáltalán ne keressünk szekvenciálisan!
- Külön vigyázzunk, hogy a keresési eljárás szélsőséges esetekben is működjön, például ha az állomány történetesen üres!

Feladat
Olvassunk be folyamatosan árukódokat, és keressük meg azokat az árukat tartalmazó állományban! Ha a megadott kódot nem találjuk, adjunk hibaüzenetet!

A feladat megoldásához az előzőleg megírt *Van* függvényt használjuk:

```
Procedure Kerdez ;
  Var
    Aruk : File Of TAru ;
    Aru : TAru ;
    Poz : LongInt ;
    Kod : TKod ;

  Begin
    Assign(Aruk,'Aruk.Dat') ;
    Reset(Aruk) ;
    Write('Kód: ') ; ReadLn(Kod) ;
```

```
While Kod <> '' Do
  Begin
    If Van(Kod,Aru,Poz) Then
      WriteLn(Aru.Leiras,' ',Aru.Egysegar:8:2)
    Else
      WriteLn('Nincs ilyen áru!') ;
    Write('Kód: ') ; ReadLn(Kod) ;
  End ;
Close(Aruk) ;
End ;
```

3.11 Módosítás

Feladat
Engedjük meg egy adott kódú áru adatainak módosítását!

Most csak egyetlen kód beolvasásáról van szó, melyet szekvenciálisan megkeresünk az állományban. A módosított adatokat terminálról bekérjük, majd a rekordot visszaírjuk az állományba a régi rekord helyére:

```
Write('Kód: ') ; ReadLn(Kod) ;
If Van(Kod,Aru,Poz) Then
  Begin
    WriteLn(Aru.Leiras,' ',Aru.Egysegar:8:2) ;
    Write('Leírás:') ; ReadLn(Aru.Leiras) ;
    Write('Egys.ár:') ; ReadLn(Aru.Egysegar) ;
    Seek(Aruk,Poz) ;
    Write(Aruk,Aru) ;
  End
Else
  WriteLn('Nincs ilyen áru!') ;
```

A következő feladatban módosítani kell az állomány összes – adott feltételnek eleget tevő – elemét:

Feladat
Az áruház összes árujának egységárát, melyek 100 Ft felett vannak, emeljük meg 10%-kal!

A feladatot kétféleképpen fogjuk megoldani. Az első megoldásban a módosított rekordokat visszaírjuk eredeti helyükre. A rekordokat sorban beolvassuk. Minden megváltoztatott rekordot ugyanarra a helyre kell kiírni, mint ahonnan azt beolvastuk. Mivel beolvasás után az állomány mutatója eggyel tovább lép, visszaírás előtt a pozíciót újra

állítani kell. A kiírás megint állítja a mutatót, de ez nem baj, mert úgyis a következő rekordot szeretnénk beolvasni.

```
Procedure Modosit1 ;
  Var
    Aruk : File Of TAru ;
    Aru : TAru ;
    I : LongInt ;
  Begin
    Assign(Aruk,'Aruk.Dat') ;
    Reset(Aruk) ;
    For I := 0 To Filesize(Aruk)-1 Do
      Begin
        Read(Aruk,Aru) ;
        With Aru Do
          If Egysegar > 100 Then
            Begin
              Egysegar := Egysegar * 1.1 ;
              Seek(Aruk,I) ;
              Write(Aruk,Aru) ;
            End ;
      End ;
    Close(Aruk) ;
  End ;
```

A második megoldásban az összes rekordot átmásoljuk egy új állományba, miközben a megfelelő rekordokat módosítjuk. A másolás végén az eredeti állományt töröljük. Ez utóbbi megoldás biztonságosabb.

```
Procedure Modosit2 ;
  Var
    Aruk, UjAruk : File Of TAru ;
    Aru : TAru ;
  Begin
    Assign(Aruk,'Aruk.Dat') ; Reset(Aruk) ;
    Assign(UjAruk,'Uj.Dat') ; Rewrite(UjAruk) ;
    While Not Eof(Aruk) Do
      Begin
        Read(Aruk,Aru) ;
        With Aru Do
          If Egysegar > 100 Then
            Egysegar := Egysegar * 1.1 ;
        Write(UjAruk,Aru) ;
      End ;
    Close(Aruk) ;
    Close(UjAruk) ;
```

```
    Erase(Aruk) ;
    Rename(UjAruk,'Aruk.Dat') ;
End ;
```

Az állományt szükség esetén csonkíthatjuk: ráállunk egy komponensre, és onnan kezdve az állományt egyszerűen levágjuk. Példaként csonkítsuk meg az *Aruk* állományt úgy, hogy új mérete 10 legyen:

```
Seek(Aruk,10) ;
Truncate(Aruk) ;
```

A típusos állomány felfogható egy tömbnek, mely sokkal nagyobb, mint a memóriatömb, és az elemeket permanensen tárolja. Elvileg alkalmazható a tömböknél tárgyalt összes algoritmus – például keresések, rendezések, karbantartás, összeválogatás.

Kérdések

1. Mit jelentenek a következő fogalmak?
   a) Fizikai név, logikai név, hozzárendelés
   b) Komponens
   c) Pufferváltozó
   d) Állománymutató
2. Típusos állomány esetén mi az írás/olvasás egysége?
3. Hogyan lehet egy típusos állományt
   a) létrehozni?
   b) kilistázni?
   c) bővíteni?
   d) módosítani?
   e) levágni?
   f) szekvenciálisan feldolgozni?
   g) direkt módon feldolgozni?
   h) törölni, átnevezni?
4. Hány állományt lehet egyszerre nyitva tartani?

Feladatok

1. Olvasson be valós (Real) számokat, és vigye fel azokat az aktuális katalógus 'Szamok.Dat' nevű állományába. Ezután végezze el a következő feladatokat:
   a) Listázza ki az állományt!
   b) Írja ki a számok átlagát!
   c) Írja ki azt a számot és annak állománybeli indexét, amely legjobban eltér az átlagtól (ha több ilyen van, írja ki az összeset)!
   d) Írja ki, hogy az állomány hány számból, és hány byte-ból áll!
   e) Bővítse az állományt újabb számokkal!
   f) Válogassa szét a számokat. Az egész számokat vigye fel egy 'Egeszek. Dat', a törteket pedig egy 'Tortek.Dat' állományba! Az egészeket tartalmazó állomány komponensei ne Real, hanem LongInt típusúak legyenek. Feltételezzük, hogy egyik szám sem nagyobb MaxLongInt-nál.
   g) Keressen meg a 'Tortek.Dat' állományban egy számot!
   A feladatot kétféleképpen oldja meg: első esetben feltételezzük, hogy minden számból csak egy lehet, második esetben lehet több is!

2*. Adott a következő rekordkép:

♦ Név	Maximum 20 karakter
♦ Irányítószám	4 számjegy. Budapesti irányítószám esetén az első jegy 1-es, a 2-3. jegyek a kerület.
♦ Cím	Maximum 30 karakter
♦ Fizetés	Egész

   a) Hozzon létre egy ilyen rekordokból álló állományt az aktuális alkönyvtárban 'Szemely.Dat' néven! Ha már létezett ilyen állomány, akkor bővítse!
   b) Írja ki az állományból az B betűvel kezdődő neveket!
   c) Írja át egy 'Nagyfiz.Dat' állományba azon rekordokat, amelyekben a fizetés > 100000!
   d) Törölje ki az állományból a nem budapesti személyek adatait (sűrítse az állományt)!
   e) Készítsen egy táblázatot, mely megadja budapesti kerületenként az átlag fizetést! Azokat a kerületeket, ahol nem volt személy, nem kell kiírni!

3. Írjon programot, mellyel rögzíthetjük a heti lottószelvények adatait: lottó száma, tippek, név és cím (ha van a szelvényen). A tippeket halmazként tárolja! A felvitel után
   a) készítsen listát a lottószelvényekről.
   b) írja ki, hány darab volt a különböző találatokból!
   c) keresse meg és írja ki a megadott nevű és című egyén tippjét! Ha nincs ilyen, adjon üzenetet!

Érdemes tanulmányozni

Angster Erzsébet-Kertész László: Turbo Pascal feladatgyűjtemény I.:
Típusos állományok fejezet, Tipall1-9 programok

4. ÁLLOMÁNYOK RENDEZÉSE, ÖSSZEVÁLOGATÁSA

E fejezetben kis és nagy állományokat fogunk rendezni. Szó lesz ezenkívül a rendezett állományok összeválogatásáról is, mely nagy állományok esetén a rendezés szerves része.

A lemezes állomány rendezése sokkal nehezebb feladat, mint egy memóriában elhelyezkedő tömbbé. Megtehetjük ugyan, hogy az állományt egyszerűen tömbnek nézzük, és a rendezést valamelyik tömbre megadott klasszikus algoritmussal elvégezzük, de az állomány komponenseinek cseréje minden szempontból sokkal „drágább" művelet, mint a memóriaváltozóké. Hogy a rendezés minél kevesebb időt és energiát vegyen igénybe, a lemezműveletek számát minimálisra kell csökkenteni. Legjobb megoldás, ha az egész állományt beolvassuk a memóriába, és rendezés után visszaírjuk a lemezre. Probléma akkor van, ha az állomány komponenseinek számát nem tudjuk előre behatárolni, vagy semmiképpen nincs elegendő memóriánk. Ilyenkor különböző trükköket kell bevetni. A rendezéseket a következő feladat kapcsán fogjuk bemutatni:

Feladat

A *Raktar.Dat* típusos állomány a raktáron levő áruk mennyiségeit tartalmazza:

- Árukód pontosan 4 karakter (1 betű, 3 szám)
- Mennyiség egész (darabszám)

Rendezzük az állományt árukód szerint növekvőleg!

4.1 Típusos állomány rendezése memóriában

Ha be tudjuk határolni a komponensek számát és azok befférnek a memóriába, akkor a rendezést végezzük el a memóriában. Tegyük fel, hogy biztos nincs 3000-nél több áru az állományban. Ebben az esetben a rendezésnek három fázisa van. Az elsőben beolvassuk az összes rekordot a memóriába, a másodikban a beolvasott rekordokat rendezzük, végül a rendezett rekordokat lemezre írjuk. A rendezéshez bármely ismert algoritmust használhatjuk – mi most a minimumkiválasztásos rendezést választjuk. A rendezés Pascal kódja a következő:

```
Type
  TKod = String[4] ;
  TAru = Record
    Kod : TKod ;
    Mennyiseg : LongInt ;
  End ;

Procedure RendezMem ;
  Var
    Raktar : File Of TAru ;
    T : Array[1..3000] Of TAru ;
    N, I, J, Min : LongInt ;
    Temp : TAru ;
  Begin
    { A rendezetlen rekordok beolvasása: }
    Assign(Raktar,'Raktar.Dat') ;
    Reset(Raktar) ;
    N := 0 ;
    While Not Eof(Raktar) Do
      Begin
        Inc(N) ;
        Read(Raktar,T[N]) ;
      End ;
    Close(Raktar) ;

    { A rekordok rendezése minimumkiválasztásos
    módszerrel: }
    For I := 1 To N-1 Do
      Begin
        Min := I ;
        For J := I+1 To N Do
          If T[J].Kod < T[Min].Kod Then
            Min := J ;
        If Min <> I Then
          Begin
            Temp:=T[I]; T[I]:=T[Min]; T[Min]:=Temp;
          End ;
      End ;

    { A rendezett rekordok kiírása az állományba: }
    Rewrite(Raktar) ;
    For I := 1 To N Do
      Write(Raktar,T[I]) ;
    Close(Raktar) ;
  End ;
```

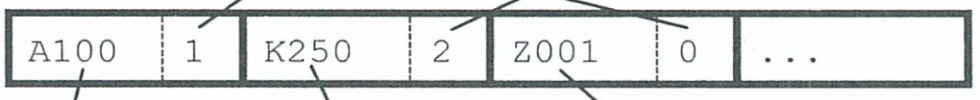
Ha n az állomány komponenseinek száma, akkor a lemezről n-szer olvasunk, és n-szer is írunk oda. Ennél kevesebb lemezművelettel egy rendezés nem oldható meg.

4.2 Rendezés indextömb segítségével

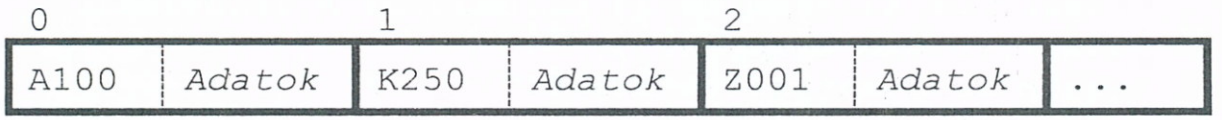
Sok memóriát tudunk spórolni, ha minden egyes komponensből csak azokat a „kulcs" adatokat olvassuk be a memóriába, melyek részt vesznek a rendezési sorrend megállapításában. A rendezendő tömb elemei most ún. kulcsrekordok lesznek. A kulcsrekord a kulcsadatok mellett tartalmaz egy indexet, mely a kulcshoz tartozó komponens állománybeli pozíciója. Az indexeket és kulcsokat tartalmazó tömböt indextömbnek vagy kulcstömbnek szokás nevezni. A kulcstömb rendezése után az indexek alapján beolvassuk az eredeti, rendezetlen állományból a rekordokat, melyeket szekvenciálisan felírunk az új, rendezett állományba. Ezt a módszert akkor alkalmazhatjuk, ha a rendezendő állomány összes rekordja számára nincs hely a memóriában, de a kulcsrekordok még mind elférnek.

Rendezetlen állomány:

Kulcsrekordok a memóriában:



Rendezett állomány:




```
Procedure RendezKulcs ;
  Type
    TKulcs = Record
         Kod : TKod ;
         Poz : LongInt ;
       End ;

  Var
    Raktar, RendRaktar : File Of TAru ;
    Aru : TAru ;
    Kulcsok : Array[1..3000] Of TKulcs ;
    Temp : TKulcs ;
    N, I, J, Min : LongInt ;
```

```
Begin
  { Rekordok kódjainak beolvasása: }
  Assign(Raktar,'Raktar.Dat') ;
  Reset(Raktar) ;
  N := 0 ;
  For I := 0 To FileSize(Raktar)-1 Do
    Begin
      Inc(N) ;
      Read(Raktar,Aru) ;
      Kulcsok[N].Kod := Aru.Kod ;
      Kulcsok[N].Poz := I ;
    End ;

  { Kulcsrekordok rendezése: }
  For I := 1 To N-1 Do
    Begin
      Min := I ;
      For J := I+1 To N Do
        If Kulcsok[J].Kod < Kulcsok[Min].Kod Then
          Min := J ;
      If Min <> I Then
        Begin
          Temp := Kulcsok[I] ;
          Kulcsok[I] := Kulcsok[Min] ;
          Kulcsok[Min] := Temp ;
        End ;
    End ;

  { Rekordok kiírása az új állományba a rendezett
  kulcsrekordok alapján: }
  Assign(RendRaktar,'RendRakt.Dat') ;
  Rewrite(RendRaktar) ;
  For I := 1 To N Do
    Begin
      Seek(Raktar,Kulcsok[I].Poz) ;
      Read(Raktar,Aru) ;
      Write(RendRaktar,Aru) ;
    End ;
  Close(Raktar) ;
  Close(RendRaktar) ;
  Erase(Raktar) ;
  Rename(RendRaktar,'Raktar.Dat') ;
End ;
```

Ha n az állomány komponenseinek száma, akkor a lemezről való olvasások száma összesen $2n$, az írások száma pedig n.

4.3 Rendezés lemezen

Megtehetjük, hogy valamelyik rendezési algoritmust közvetlenül alkalmazzuk az állományra. Az ilyen rendezés egyetlen előnye, hogy akármekkora állomány esetén működik. Óriási hátránya viszont, hogy annyi a lemezművelet, hogy nagyobb állomány esetén ez az algoritmus kivárhatatlan és „lemeznyűvő".

Nézzük például a minimumkiválasztásos módszert. A külső ciklusban megkeressük az *I.* helyre a még rendezetlen állományrész legkisebb elemét. Ehhez beolvassuk az *I.* komponenst, és innen kezdve végignézzük az állományt, van-e ennél előrébb való elem. Ha van és annak indexe *Min*, akkor az *I.* és *Min.* komponenseket felcseréljük a lemezen:

```
Procedure RendezLemez ;
  Var
    Raktar : File Of TAru ;
    IAru, JAru, MinAru : TAru ;
    I, J, Min : LongInt ;
  Begin
    Assign(Raktar,'Raktar.Dat') ;
    Reset(Raktar) ;
    For I := 0 To FileSize(Raktar)-2 Do
      Begin
        Seek(Raktar,I) ;
        Read(Raktar,IAru) ;
        Min := I ;
        MinAru := IAru ;
        For J := I+1 To FileSize(Raktar)-1 Do
          Begin
            Seek(Raktar,J) ;
            Read(Raktar,JAru) ;
            If JAru.Kod < MinAru.Kod Then
              Begin
                Min := J ; MinAru := JAru ;
              End ;
          End ;
        If Min <> I Then
          Begin
            Seek(Raktar,Min) ;
            Write(Raktar,IAru) ;
            Seek(Raktar,I) ;
            Write(Raktar,MinAru) ;
          End ;
      End ;
    Close(Raktar) ;
  End ;
```

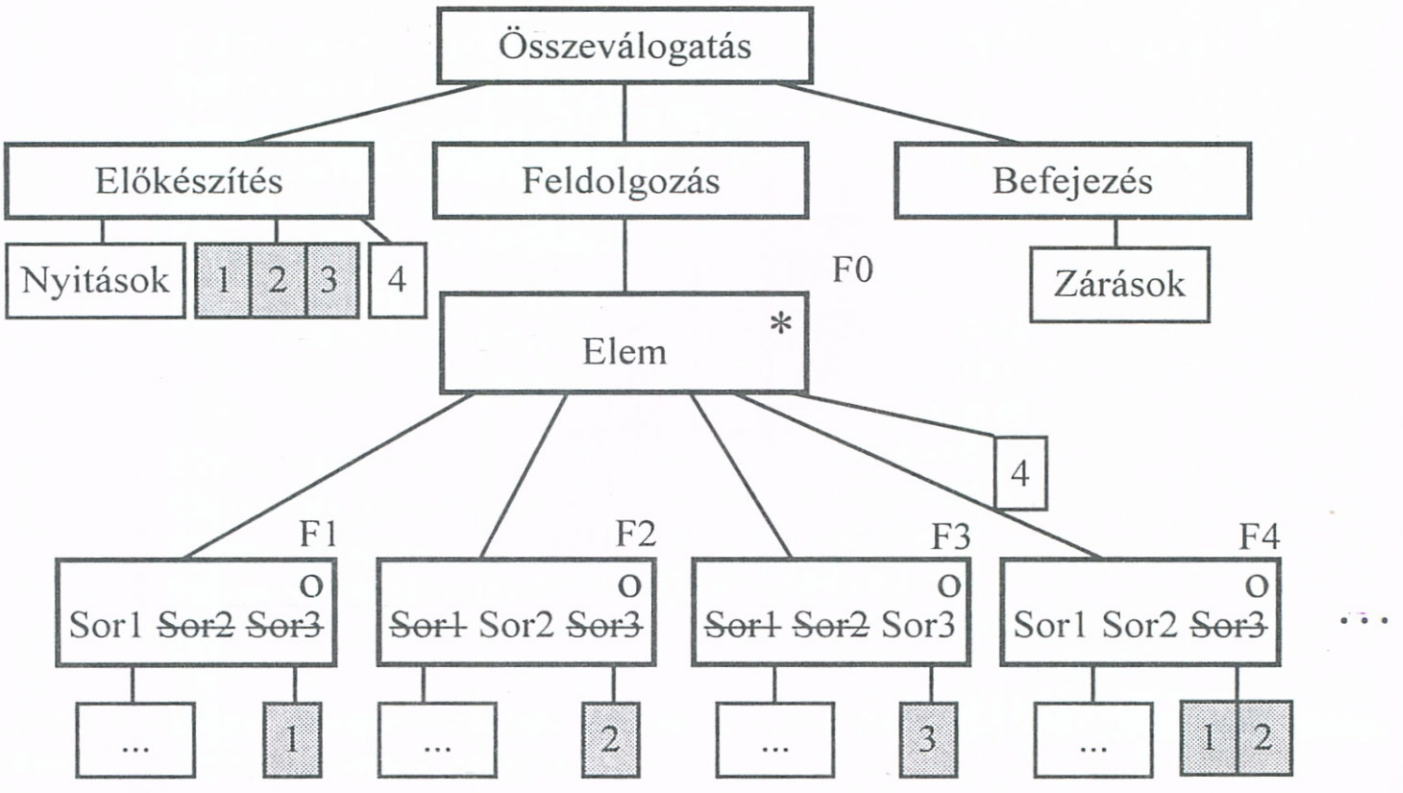
Ha n az állomány komponenseinek száma, akkor a lemezről való olvasások száma nagyságrendileg $n^2/2$, hiszen minden legkisebb elem megkereséséhez átlagban $n/2$-ször kell végigolvasni az állományt. Az írások száma $2n$, hiszen majdnem mindegyik külső ciklusban elemcsere van, ami 2 lemezírást jelent.

4.4 Állományok összeválogatása

Az 1. kötet *„Sorozatok feldolgozása"* fejezetében már szó volt rendezett sorozatok összeválogatásáról. Az ott tárgyalt algoritmusok természetesen nem csak tömbökre alkalmazhatók, hanem minden olyan esetre, ahol a bemenő sorozatok rendezettek. Legyenek most ezek a sorozatok típusos állományok!

Rendezett sorozatok összeválogatásának lényege a következő: minden rendezett sorozatból hozzáférhetővé tesszük a soron következő elemet. A „készenlétben lévő" elemekből kiválasztjuk a sorrendben következőt (legkisebbet) – ilyenből lehet egy vagy több. Ezekből bizonyosakat feldolgozunk (akár az összeset), és a feldolgozott elem(ek) helyére léptetjük a következő várakozó eleme(ke)t. A ciklus addig megy, amíg el nem fogy az összes sorozat. Mivel a rendezett sorozatokból mindig a legkisebb elemeket dolgozzuk fel, ezért az eredménysorozat is rendezett lesz. Ismeretes, hogy az összeválogatás egyik „legkényesebb" része a sorozatok végeinek figyelése. Egy megoldás a következő: ha egy sorozatból elfogytak az elemek, akkor annak készenléti (aktuális) elemét maximumra állítjuk, s ezzel kényszerítjük az algoritmust a többi sorozat (rendezettségben előrébb álló) elemeinek feldolgozására.

Nézzük meg az összeválogatás általános programtervét abban az esetben, ha a bemenő sorozatok rendezettek, és azok egyikében sincs két egyforma elem. Három sorozat esetén az algoritmus nagy vonalakban így néz ki:



Tevékenységjegyzék:

1: Be(Sor1): Elem1. Ha nincs elem, akkor Elem1 ← Ütköző
2: Be(Sor2): Elem2. Ha nincs elem, akkor Elem2 ← Ütköző
...
4: Elem meghatározása: Elem ← Minimum(Elem1, Elem2, ...)

Feltételjegyzék:

F0: Van még elem valamely sorozatban (Elem < Ütköző).
F1: Elem Sor1-ben van, Sor2-ben nincs, Sor3-ban nincs.
F2: Elem Sor1-ben nincs, Sor2-ben van, Sor3-ban nincs.
F3: Elem Sor1-ben nincs, Sor2-ben nincs, Sor3-ban van.
F4: Elem Sor1-ben van, Sor2-ben van, Sor3-ban nincs.
...

A program fel kell, hogy dolgozza az összes bemenő sorozatot, mégpedig úgy, hogy az azonos elemek egyszerre legyenek „kéznél". A fő ciklus tehát legyen olyan, hogy minden különböző elemre létezzen egy ciklusmag. Ebben a ciklusmagban aztán el kell döntenünk, hogy az éppen terítéken levő (nagyság szerint következő) elem mely sorozatokban van jelen: a feldolgozás ugyanis ennek függvénye. A szelekciókat úgy kell összeállítani, hogy minden lehetséges eset külön ágként szerepeljen. A lehetséges esetek száma *n* bemenő állomány esetén 2^n-1. Azért nem 2^n, mert az „egyik sorozatban sincs" eset értelmetlen (a következő elemet mindig a bemenő sorozatokból vesszük). A lehetséges esetek száma két bemenő állomány esetén 3, három bemenő állomány esetén pedig 7.

Egy ciklus lefutásakor az összes egyforma elemet feldolgozzuk. Minden ciklus elején meg kell határoznunk a várakozó elemek közül a legkisebbet (4-es tevékenység). Feldolgozás után fontos, hogy pótoljuk a megfelelő elemeket: ha az adott ciklusban egy sorozatban volt feldolgozandó elem, ott a következő elemre kell lépni. Ha nincs több elem, akkor oda egy ütközőt kell állítani, mely nagyobb az összes lehetséges elemnél (1-es, 2-es, ... tevékenység). Mivel a feldolgozás addig megy, amíg az aktuális elem nem az ütköző, egészen biztos, hogy az összes elemet feldolgozzuk.

Nézzünk most egy konkrét feladatot:

Feladat

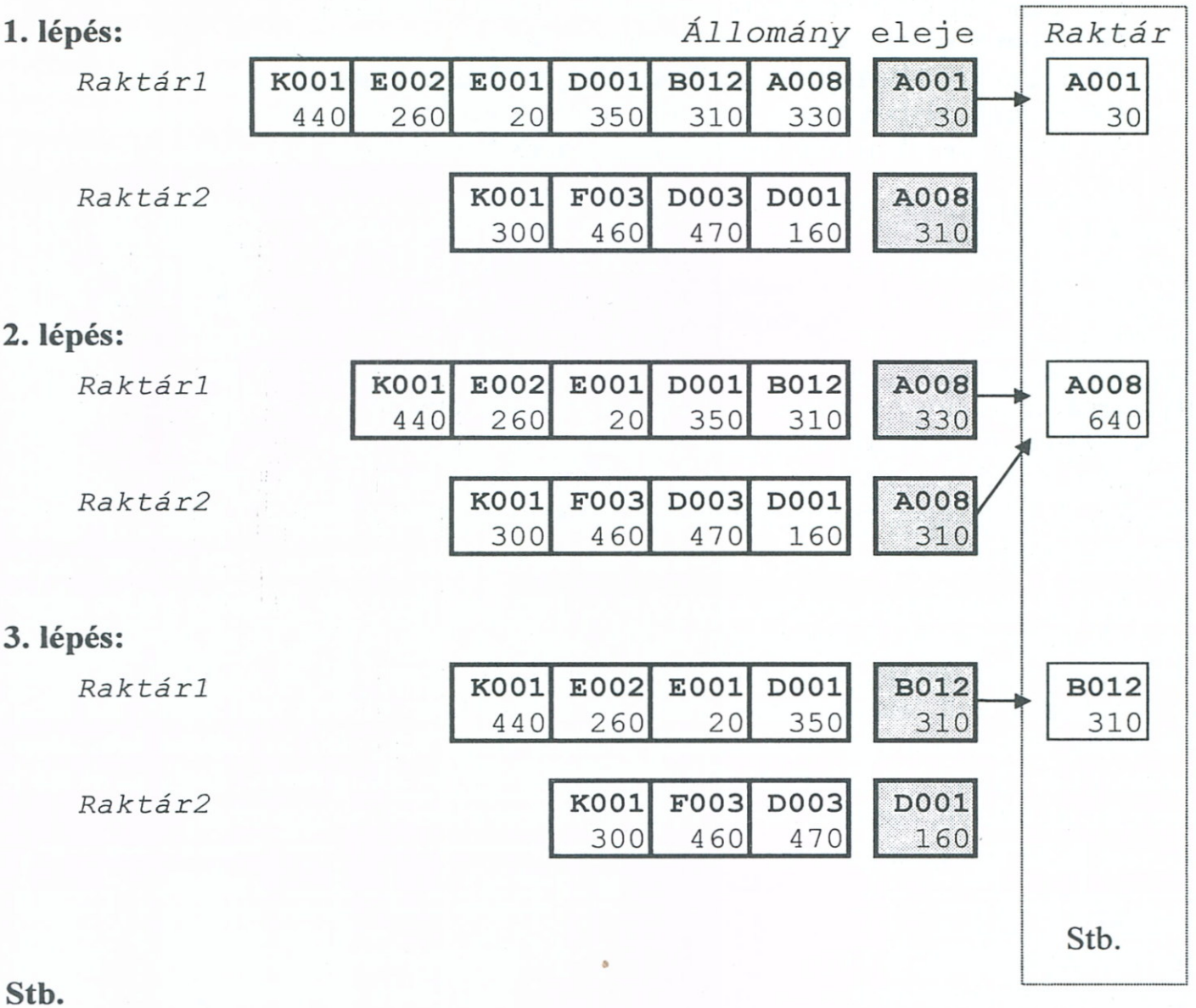
Egy kiskereskedésben egy új raktárépületet rendeznek be, miközben két régit felszámolnak. Ezért a két régi raktár áruit át kell tenni az újba. A *Raktar1.Dat* és a *Raktar2.Dat* állományok tartalmazzák a régi raktárak áruihoz tartozó adatokat. A rekordkép a következő:

- Árukód pontosan 4 karakter (1 betű, 3 szám)
- Mennyiség egész

Mindkét állomány árukód szerint rendezett, és az árukód az áru azonosítója (egyikben sincs két egyforma árukód). Válogassuk össze a két állományt úgy, hogy a *Raktar.Dat* eredményállomány is rendezett legyen árukód szerint, és az árukód ott is egyedi legyen. A régi raktárakban természetesen lehetnek egyforma áruk is – ilyenkor az áruk mennyiségei összeadandók.

Először mindkét állományból beolvasunk egy áru rekordot. Azt az árut írjuk fel az új állományba, amelyiknek kódja kisebb. A feldolgozott rekordot pótoljuk, azaz beolvassuk a következőt az állományból. Ha a kódok egyformák, akkor csak egy rekordot írunk fel az összeadott mennyiségekkel, és mindkét bemenő állományból pótolunk.

Nézzünk meg példaként két bemenő állományt, *Raktár1* és *Raktár2*-t:



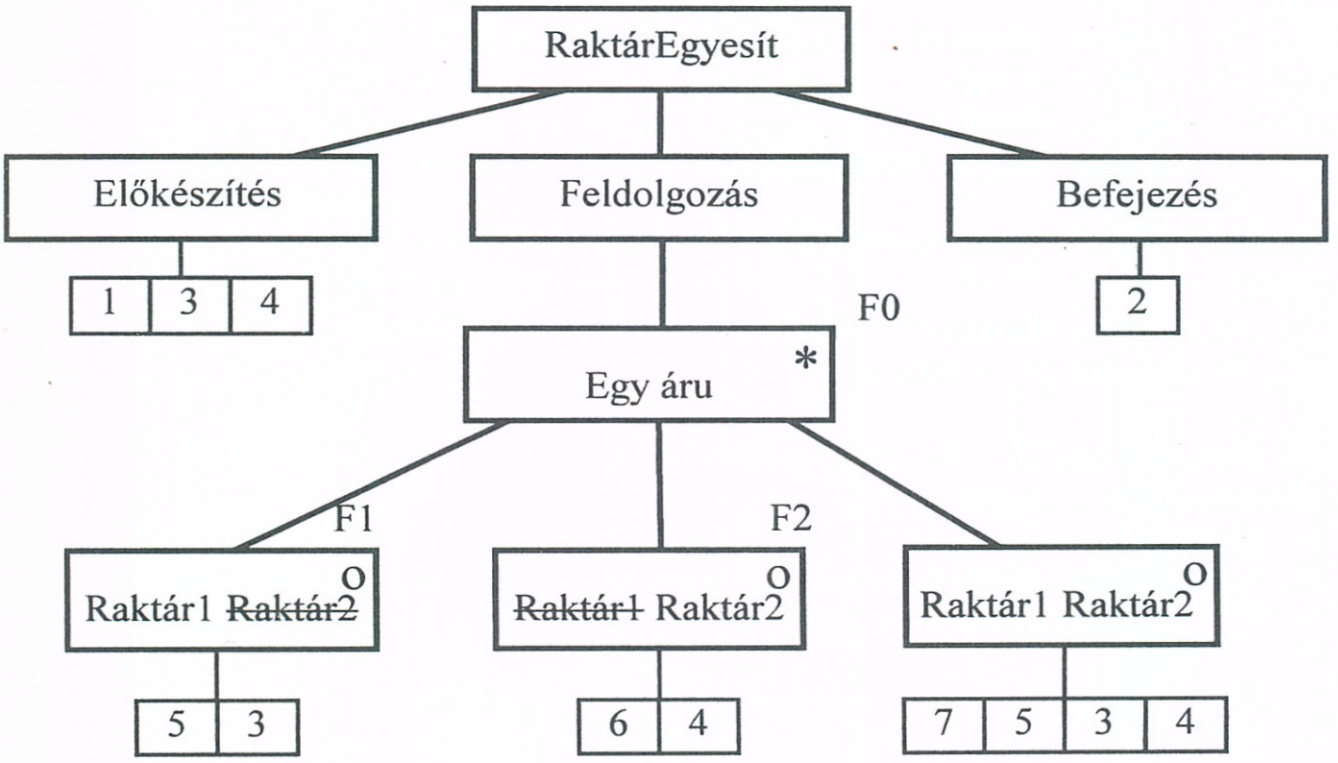
Az összeválogatás eredménye a következő lesz:

Raktár: *Állomány eleje*

K001	F003	E002	E001	D003	D001	B012	A008	A001
740	460	260	20	470	510	310	640	30

Az állományból csak akkor olvasunk, ha még van beolvasandó rekord, vagyis az *Eof* függvény *False* értéket ad vissza. Ha már nincs több, akkor a kódot 'ZZZZ'-re állítjuk – ennél nagyobb kód biztosan nem lesz, hiszen annak 2-4. karakterei számjegyek.

A program Jackson terve a következő:



Adatok:

TÁru = Rekord (Kód, Mennyiség)
Raktár1, Raktár2: Bemenő állományok
Raktár: Kimenő állomány
Áru1, Áru2: TÁru – Raktár1 és Raktár2 pufferterületei

Feltételjegyzék:

F0: Áru1.Kód <> 'ZZZZ' vagy Áru2.Kód <> 'ZZZZ'
F1: Áru1.Kód < Áru2.Kód
F2: Áru1.Kód > Áru2.Kód

Tevékenységjegyzék:

1: Raktár1 és Raktár2 nyitása olvasásra, Raktár nyitása írásra.
2: Raktár1, Raktár2 és Raktár zárása.
3: Be(Raktár1): Áru1. Ha vége: Áru1.Kód ← 'ZZZZ'
4: Be(Raktár2): Áru2. Ha vége: Áru2.Kód ← 'ZZZZ'
5: Ki(Raktár): Áru1
6: Ki(Raktár): Áru2
7: Áru1.Mennyiség $\overset{+}{\leftarrow}$ Áru2.Mennyiség

A Turbo Pascal kód:

```
Program RaktarEgyesit ;
Type
  TKod = String[4] ;
  TAru = Record
    Kod : TKod ;
    Mennyiseg : LongInt ;
  End ;

Var
  Raktar1, Raktar2, Raktar : File Of TAru ;
  Aru1, Aru2 : TAru ;

{ Pótlás az 1. állományból: }
Procedure Olvas1 ;
  Begin
    If Not Eof(Raktar1) Then
      Read(Raktar1,Aru1)
    Else
      Aru1.Kod := 'ZZZZ' ;
  End ;

{ Pótlás a 2. állományból: }
Procedure Olvas2 ;
  Begin
    If Not Eof(Raktar2) Then
      Read(Raktar2,Aru2)
    Else
      Aru2.Kod := 'ZZZZ' ;
  End ;

{ Összeválogatás: }
Begin
  Assign(Raktar1,'Raktar1.Dat') ; Reset(Raktar1) ;
  Assign(Raktar2,'Raktar2.Dat') ; Reset(Raktar2) ;
  Assign(Raktar,'Raktar.Dat') ; Rewrite(Raktar) ;

  Olvas1 ;
  Olvas2 ;

  While (Aru1.Kod <> 'ZZZZ') Or (Aru2.Kod <> 'ZZZZ') Do
    Begin
      If Aru1.Kod < Aru2.Kod Then
        Begin
          Write(Raktar,Aru1) ;
          Olvas1 ;
        End
```

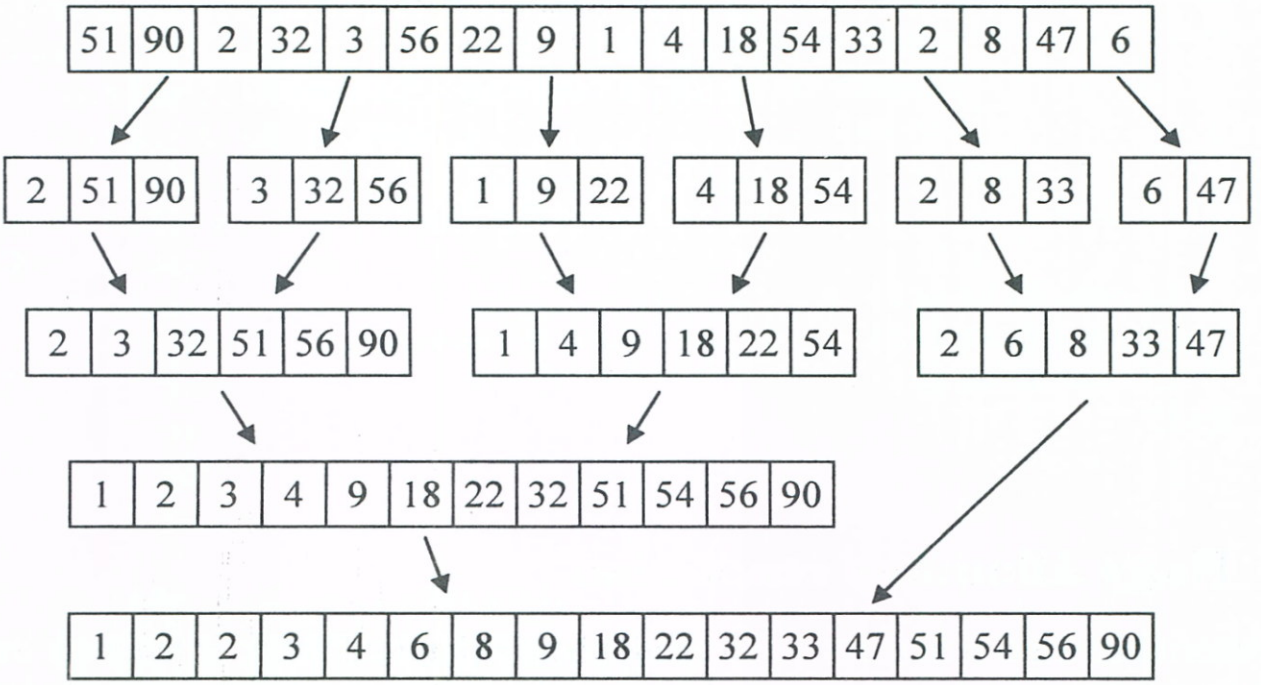
```
        Else If Aru1.Kod > Aru2.Kod Then
          Begin
            Write(Raktar,Aru2) ;
            Olvas2 ;
          End
        Else { Aru1.Kod = Aru2.Kod }
          Begin
            Aru1.Mennyiseg:= Aru1.Mennyiseg+Aru2.Mennyiseg;
            Write(Raktar,Aru1) ;
            Olvas1 ;
            Olvas2 ;
          End ;
      End ;

    Close(Raktar1) ;
    Close(Raktar2) ;
    Close(Raktar) ;
End.
```

4.5 Nagy állomány rendezése

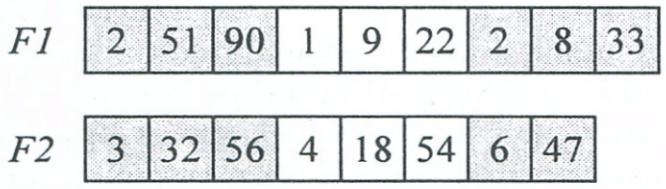
A memóriában való rendezést belső rendezésnek (internal sort), a lemezen történőt külső rendezésnek (external sort) szokás nevezni. A legnépszerűbb külső rendezés az ún. rendezés/összeválogatás (sort/merge). Ennek lényege, hogy a nagy állományt darabonként rendezzük a memóriában, majd a rendezett részeket összeválogatjuk. Első lépésként az állományt szakaszokra osztjuk – hogy mekkora egy szakasz, az főleg a rendelkezésre álló memória nagyságától függ. A szakaszokat egymás után beolvassuk a memóriába, rendezzük, és kiírjuk lemezre. Mivel következő lépésként a rendezett szakaszokat össze kell válogatnunk, a kiírás minimum két állományba kell, hogy történjen. Szélsőséges esetben annyi állományt készítünk, amennyi a rendezett szakaszok száma. Például ha az állomány mérete 20000 rekord, és egyszerre 3000 rekord fér be a memóriába (a szakasz mérete 3000), akkor a szakaszok száma kezdetben 20000/3000=6,666..., vagyis 7. Megtehetjük, hogy készítünk 7 állományt a 7 rendezett szakasszal, majd azokat egyetlen menettel összeválogatjuk. De mit csinálunk 100000 rekord esetén, amikor a szakaszok száma 34? Ennyi állományt általában nem lehet egyszerre nyitva tartani, nem beszélve arról, hogy ezek nyitása, csukása is rengeteg időt vesz igénybe, és az összeválogatás egy ciklusára eső összes hasonlítások száma is számottevő. Ha kevesebb rendezett szakaszt válogatunk össze egyszerre, akkor az összeválogatásnak több menete lesz. Több menetben az állományokat természetesen többször kell végigolvasnunk, ami idő és energia. Meg kell tehát találni egy egészséges kompromisszumot, melyben sem az állományok, sem a menetek száma nem nagy.

Két-utas rendezés/összeválogatás

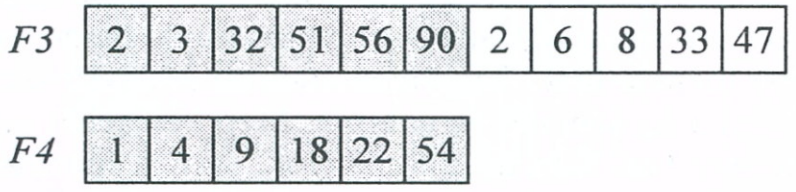
Két-utas megoldásról beszélünk, ha egyszerre mindig csak két rendezett szakaszt válogatunk össze. Példánkban a rendezendő rekordok száma 17, melyből m=3 rekord fér be egyszerre a memóriába, ennyi a kezdeti szakasz mérete. Kezdetben a szakaszok száma n=(áll.méret+m-1) div m, vagyis (17+2) div 3=6. A kezdeti rendezés, szétosztás után az összeválogatások következnek:



Belső rendezés csak az első lépésben van. Mivel mindig csak két szakaszt válogatunk össze egyszerre, elegendő a szakaszokat két állományba szétosztani – azokat hol *F1*-be, hol *F2*-be írjuk ki felváltva:



Most a párhuzamos szakaszokat összeválogatva dupla hosszúságú rendezett szakaszok keletkeznek, melyeket újabb két állományba osztunk szét. Példánkban ez így fest:

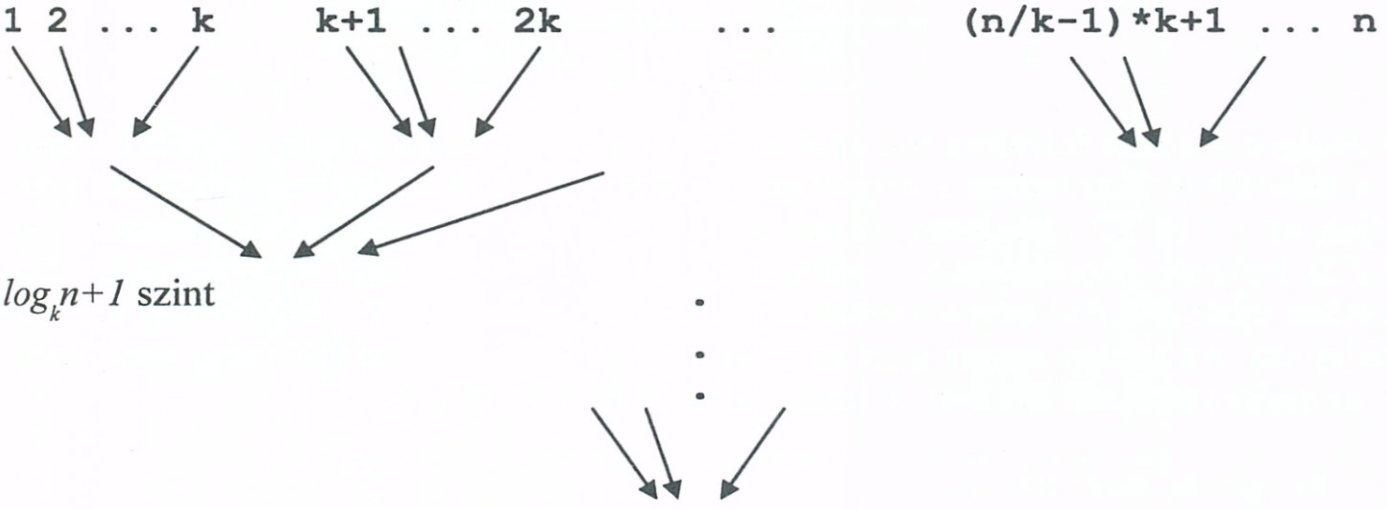


Következő lépésben *F1*-be kerül a két első 6+6 szakasz, *F2*-be pedig a maradék 5. Utolsó, 4. lépésként *F3*-ban kész a rendezett állomány.

E módszer előnye, hogy nem kell annyi állományt nyitva tartani, és a hasonlítások száma egy-egy ciklus esetén minimális. Ha n a kezdeti szakaszok száma, akkor az összeválogatás $log_2 n+1$ menetben történik. Példánkban a menetek száma $log_2 6+1=4$.

K-utas rendezés

K-utas rendezésről beszélünk, ha egyszerre k darab szakaszt válogathatunk össze. Kezdetben a szakaszok száma n:



Az ilyen rendezéshez k darab állományra van szükség. Előnye a 2-utas módszerrel szemben, hogy a menetek száma kevesebb. k=3 esetén az előző példánkban szereplő 6 szakasz kezdeti szétosztás után így néz ki:

F1	2	51	90	4	18	54
F2	3	32	56	2	8	33
F3	1	9	22	6	47	

Az első összeválogatás után már csak 2 darab, maximum 9 hosszúságú rendezett szakaszt kell újra összeválogatni, és kész a rendezett állomány. Így a menetek száma $log_3 6+1=3$.

A most következő feladatban egy nagy állományt két-utas módszerrel rendezünk:

Feladat

A *Nagy.Dat* állomány rekordképe a következő:

- Árukód pontosan 4 karakter (1 betű, 3 szám)
- Mennyiség egész

Az állomány mérete feltehetően nagyon nagy. Rendezzük az állományt!

A következő adatokat a program elején rögzítjük:

- MMax: A memóriába beférő rekordok száma (most 5000)
- FN : Beviteli, illetve kiviteli állományok száma (most 2)

A beviteli, illetve kiviteli logikai állományokat az *F* kétdimenziós tömbben tároljuk:

		1	2 (*FN*)
Be	1	*F1.Tmp*	*F2.Tmp*
Ki	2	*F3.Tmp*	*F4.Tmp*

A feldolgozás során felváltva hol az első sort fogjuk beviteli állományokként kezelni, hol a másodikat. Kezdetben a *Nagy.Dat* állományt szakaszonként rendezzük, és szétosztjuk az *F1* és *F2* ideiglenes állományokba. Az első összeválogatási menetben az *(F1,F2)* beviteli állományokból válogatunk az *(F3,F4)* kiviteli állományokba. Ezt követően tehát *(F3,F4)* beviteli állományokból válogatunk az *(F1,F2)* kiviteli állományokba stb. Az utolsó menet után az első kiviteli állományt tartjuk meg, mely most már az egész rendezett állományt tartalmazza. A többit töröljük.

```
Program SortMerge ;
Type
   TKulcs = String[4] ;
   TRek = Record
      Kulcs : TKulcs ;
      Mennyiseg : LongInt ;
   End ;
   TFile = File Of TRek ;

Const
   FN = 2 ; { Input, illetve output állományok száma }
   MMax = 5000 ; { Egyszerre beférő rekordok száma }
   MaxKulcs = 'ZZZZ' ;

Var
   { Rendezendő állomány: }
   FNagy : TFile ;
   { Ideiglenes állományok és jellemzőik: }
   F : Array[1..2,1..FN] Of TFile ;
   FileVege : Array[1..FN] Of Boolean ;
   Rek : Array[1..FN] Of TRek ;

   { Egy szakasz a memóriában és jellemzői: }
   Szakasz : Array[1..MMax] Of TRek ;
   SzakaszHossz : LongInt ;
   SzakaszVege : Array[1..FN] Of Boolean ;
   SzakaszIndex : Array[1..FN] Of LongInt ;
```

```
  { Segédváltozók }
  Menet, Futam : Word ;
  M, Be, Ki, Hova, I : Word ;

{ Szakasz rendezése gyors rendezéssel: }
Procedure Quick(Bal,Jobb: Word) ;
  Var
    I, J : Word ;
    Kozepso : TKulcs ;
    Temp : TRek ;
  Begin
    I := Bal ;
    J := Jobb ;
    Kozepso := Szakasz[(I + J) Div 2].Kulcs ;
    While I <= J Do
      Begin
        While Szakasz[I].Kulcs < Kozepso Do
          Inc(I) ;
        While Szakasz[J].Kulcs > Kozepso Do
          Dec(J) ;
        If I <= J Then
          Begin
            Temp := Szakasz[I] ;
            Szakasz[I] := Szakasz[J] ;
            Szakasz[J] := Temp ;
            Inc(I) ;
            Dec(J) ;
          End ;
      End ;
    If Bal < J Then
      Quick(Bal,J) ;
    If I < Jobb Then
      Quick(I,Jobb) ;
  End ;

{ Olvasás az I. Input állományból: }
Procedure Olvas(I : LongInt) ;
  Begin
    FileVege[I] := Eof(F[Be,I]) ;
    SzakaszVege[I] :=
      (SzakaszIndex[I] >= SzakaszHossz) Or FileVege[I] ;
    If Not SzakaszVege[I] Then
      Begin
        Inc(SzakaszIndex[I]) ;
        Read(F[Be,I],Rek[I]) ;
      End
```

```
      Else
        Rek[I].Kulcs := MaxKulcs ;
    End ;

Begin { Főprogram }
    Assign(FNagy,'Nagy.Dat') ;
    Assign(F[1,1],'F1.Tmp') ;
    Assign(F[1,2],'F2.Tmp') ;
    Assign(F[2,1],'F3.Tmp') ;
    Assign(F[2,2],'F4.Tmp') ;
    Reset(FNagy) ;

    { Kezdeti rendezés és szétosztás F1 és F2-be: }
    Ki := 1 ;
    Rewrite(F[Ki,1]) ;
    Rewrite(F[Ki,2]) ;
    { 1 futamban 1 szakaszt beolvasunk, rendezünk, és
    kiírunk valamelyik output file-ba. A futamok addig
    mennek, amíg el nem fogy az input állomány: }
    Futam := 0 ;
    While Not Eof(FNagy) Do
      Begin
        Inc(Futam) ;
        M := 0 ;
        While Not Eof(FNagy) And (M < MMax) Do
          Begin
            Inc(M) ;
            Read(FNagy,Szakasz[M]) ;
          End ;
        Quick(1,M) ;
        { Hova értékei rendre 1,2,1,2,1,... lesznek: }
        Hova := (Futam-1) Mod FN + 1 ;
        For I := 1 To M Do
          Write(F[Ki,Hova],Szakasz[I]) ;
      End ;
    Close(FNagy) ; Close(F[Ki,1]) ; Close(F[Ki,2]) ;

    { Összeválogatások. Kezdetben a szakasz hossza MMAx,
    mely minden menetben kétszereződni fog. Először az input
    állományok az F1 és F2, output állományok pedig az
    F3 és F4. Ezek minden menetben cserélődnek: }
    SzakaszHossz := MMax ;
    Be := 1 ; Ki := 2 ;

    { Az összeválogatás több menetből áll. Amíg a futamok
    száma > 1, addig van még menet: }
```

```
Menet := 0 ;
Repeat
     { Input és output állományok nyitása: }
     Inc(Menet) ;
     Reset(F[Be,1]) ; Reset(F[Be,2]) ;
     Rewrite(F[Ki,1]) ; Rewrite(F[Ki,2]) ;
     For I := 1 To FN Do FileVege[I] := False ;
     Futam := 0 ;
     Repeat
          Inc(Futam) ;
          Hova := (Futam-1) Mod FN + 1 ;
          { Egy-egy szakasz összefuttatása, írás a Hova
          állományba: }
          For I := 1 To FN Do
            Begin
              SzakaszIndex[I] := 0 ;
              SzakaszVege[I] := False ;
            End ;
          Olvas(1) ;
          Olvas(2) ;
          While Not SzakaszVege[1] Or Not SzakaszVege[2] Do
            Begin
              If Rek[1].Kulcs < Rek[2].Kulcs Then
                Begin
                  Write(F[Ki,Hova],Rek[1]) ;
                  Olvas(1) ;
                End
              Else
                Begin
                  Write(F[Ki,Hova],Rek[2]) ;
                  Olvas(2) ;
                End ;
            End ;
     Until FileVege[1] And FileVege[2] ;
     Close(F[Be,1]) ; Close(F[Be,2]) ;
     Close(F[Ki,1]) ; Close(F[Ki,2]) ;
     SzakaszHossz := SzakaszHossz * 2 ;
     If Futam > 1 Then
       Begin
         If Be = 1 Then
           Begin Be := 2 ; Ki := 1 ; End
         Else
           Begin Be := 1 ; Ki := 2 ; End ;
       End ;
Until Futam = 1 ;
```

```
    WriteLn('A menetek száma ',Menet,' volt') ;
    { Eredeti és segédállományok törlése, eredmény állomány
    átnevezése: }
    Erase(FNagy) ;
    Erase(F[Be,1]) ; Erase(F[Be,2]) ;
    Erase(F[Ki,2]) ;
    Rename(F[Ki,1],'Nagy.Dat') ;
End.
```

Legyen példának az állomány rekordjainak száma 67000. Mivel egyszerre 5000 rekord fér be a memóriába, a szakaszok száma kezdetben 14. A rendezés $log_2 14+1=4$ menetes lesz.

4.6 Keresés rendezett állományban

Egy rendezett állományban kereshetünk akár szekvenciálisan, akár binárisan. A rendezett sorozatokban való kereséseket az 1. kötet *„Rendezések, keresések, karbantartás"* fejezetében tárgyaltuk. Alkalmazzuk most ezeket az algoritmusokat rendezett állományokra!

Szekvenciális keresés esetén természetesen nem csak az állomány végénél hagyjuk abba a keresést, hanem akkor is, ha az állomány komponense rendezettségben a keresett elem után következik:

```
Function Van(Kulcs: TKulcs; Var Rek: TRek;
  Var Poz: LongInt): Boolean ;
  Var
    Tovabb : Boolean ;
  Begin
    Van := False ;
    Tovabb := True ;
    Seek(F,0) ;
    Poz := -1 ;
    While Not Eof(F) And Tovabb Do
      Begin
        Inc(Poz) ;
        Read(F,Rek) ;
        Tovabb := Kulcs > Rek.Kulcs ;
        Van := Kulcs = Rek.Kulcs ;
      End ;
  End ;
```

A *bináris keresés* külső tárolás esetén annyival hatékonyabb, hogy rendezett állományban szinte „vétek" szekvenciálisan keresni. Nézzük az algoritmust:

```
Function BVan(Kulcs: TKulcs; Var Rek: Trek;
  Var Poz: LongInt): Boolean ;
  Var
    Elso, Utolso, Kozepso: LongInt ;
    Ok: Boolean ;
  Begin
    Elso := 0 ;
    Utolso := FileSize(F)-1 ;
    If Utolso = -1 Then
      Begin
        BVan := False ;
        Exit ;
      End ;

    Ok := False ;
    While (Elso <= Utolso) And (Not Ok) Do
      Begin
        Kozepso := (Elso+Utolso) Div 2 ;
        Seek(F,Kozepso) ;
        Read(F,Rek) ;
        If Kulcs > Rek.Kulcs Then
          Elso := Kozepso + 1
        Else If Kulcs < Rek.Kulcs Then
          Utolso := Kozepso - 1
        Else
          Ok := True ;
      End ;
    Poz := FilePos(F)-1 ;
    BVan := Ok ;
  End ;
```

Tegyünk egy összehasonlítást a két keresés közt egy 67000 komponensből álló rendezett állomány esetén. Ha szekvenciálisan keresünk, akkor átlagban 33500-at kell olvasnunk ahhoz, hogy az elemet megtaláljuk, vagy kiderüljön, hogy nincs az állományban. Bináris keresés esetén 10-17 olvasásból mindig „megússzuk" a keresést.

Ha több, adott tulajdonságú elemet keresünk, akkor szekvenciális keresés esetén tovább kell folytatnunk a keresést. Bináris keresésnél nem tudjuk, hogy az egymás mellett szereplő egyforma elemek közül melyiket „találtuk el", ezért a további keresést mindkét irányban el kell végeznünk.

Kérdések

1. Mit jelentenek a következő fogalmak?
   a) Állomány rendezése
   b) Rendezés memóriában

c) Indextömb
d) Állományok összeválogatása
e) Belső rendezés, külső rendezés
f) Sort/merge
g) 2-utas rendezés/összeválogatás
h) K-utas rendezés/összeválogatás
i) Szekvenciális keresés az állományban
j) Bináris keresés az állományban

2. Milyen szempontokat kell figyelembe venni egy állomány rendezésénél?

3. Miért nem ajánlatos az állomány rendezését kizárólag lemezen elvégezni?

Feladatok

1. Készítsen egy valós, véletlen számokból álló állományt. A komponensek száma először *1 000*, majd *1 000 000* legyen!
   a) Rendezze növekvőleg az állományt!
   b) Keressen meg a rendezett állományban egy adott számot! Írja ki az összes ilyen szám állománybeli pozícióját! Végezze el a keresést szekvenciálisan és binárisan is. Hasonlítsa össze a keresési időket!
   c) Listázza ki az állományt két megadott érték közt!

2. Adott a következő rekordkép:
   - Vevőkód 2 betű, 4 számjegy
   - Dátum 6 karakter. ÉÉHHNN alakú
   - Összeg Valós

   a) Hozzon létre egy ilyen rekordokból álló állományt az aktuális alkönyvtárban *Eladas.Dat* néven! A komponenseket véletlenszámgenerátorral állítsa össze, az állomány mérete legalább *10 000* legyen!
   b) Rendezze az állományt dátum szerint növekvőleg! Számolja ki a megadott naphoz tartozó eladások teljes összegét!
   c) Rendezze az állományt vevőkód szerint növekvőleg, azon belül dátum szerint csökkenőleg! Írja ki, egy adott vevő összes vásárlásait!

Érdemes tanulmányozni

Angster Erzsébet-Kertész László:
Turbo Pascal feladatgyűjtemény I.: Típusos állományok fejezet, Tipall11 program
Turbo Pascal feladatgyűjtemény II.: Turbo Sort fejezet

5. KARBANTARTÁS

E fejezet a karbantartási funkciókkal kapcsolatos alapvető kérdéseket tárgyalja. Az egyszerű karbantartási algoritmusok megismerése után az indextömbös karbantartásról lesz szó, végül tranzakciós állomány segítségével kötegelt karbantartást végzünk.

5.1 Alapgondolatok

Egy adatállománynak olyannak kell lennie, hogy a benne lévő adatok keresése és manipulálása a felhasználó igényeit messzemenőkig kielégítse. Mit szeretne a felhasználó?

- A megadott tulajdonságú adatokat (rekordokat) pillanatokon belül és a legkényelmesebb módon megtalálni;
- Az adatokat más és más rendezettségben listázni, illetve pásztázni;
- Az állományt folyamatosan napra kész állapotra hozni, vagyis *karbantartani* – bármikor újabb adatokat felvinni, a már meglévőket szükség esetén törölni, módosítani.

Ezek mind jogos felhasználói igények. Nem kis feladat ezeket az igényeket kielégíteni, különösen nagy adatállományok esetén. Az, hogy a fenti funkciók milyen gyorsak és mennyire megbízhatóak, az a hardver feltételeken túl leginkább az állomány szervezésén múlik. A szervezési mód megadásánál egyrészt rögzítjük az *állomány struktúráját* (azokat a szabályokat, melyek szerint az adatok elhelyezkednek az állományban), másrészt definiáljuk az *állománykezelési műveletek* algoritmusait. Ez utóbbiak alapvetően négy csoportba oszthatók:

- Létrehozás
- Visszakeresés
- Karbantartás
- Újraszervezés

A *létrehozás* az új állomány előkészítési műveleteit foglalja magában. A *visszakeresés* különböző algoritmusokat ad a megadott tulajdonságú adatok minél rövidebb idő alatt történő elérésére. A *karbantartási funkciók* a következők:

- Új rekord felvitele
- Meglévő rekord törlése
- Meglévő rekord módosítása

A karbantartási műveleteket legtöbb esetben megelőzi egy keresés: új rekordot csak akkor vihetünk fel az állományba, ha ilyen azonosítójú rekord még nem létezik, törlés, illetve módosítás esetén pedig meg kell keresni a kérdéses rekordot. Általában az a jó, ha az állomány valamilyen szempont szerint rendezett, ugyanis egy rendezett adathalmazban mindig gyorsabb a keresés, és egy rendezett listát is csak így lehet azonnal produkálni. A rendezettséget természetesen akkor is meg kell tartanunk, amikor egy új adatot felírunk az állományba, illetve egy régit megszüntetünk. Rendezettségen nem mindig fizikai rendezettség értendő, közvetítheti azt valamilyen kiegészítő adatstruktúra is, mint például egy indextömb. Az indextömb(ök) használatának óriási előnye, hogy így egy adatállomány több szempont szerint is rendezett lehet egyszerre. Több index karbantartása persze sok időt vesz igénybe, de kereséskor ezek az idők megtérülhetnek. Az állomány szerkezeti felépítését, szervezését mindig az adott körülmények és követelmények ismeretében döntjük el. Mérlegelni kell, hogy az állomány élete során mennyit és milyen szempont szerint fogunk keresni, hányszor fogunk új adatot felvinni, illetve már meglévőt törölni. Új rekord felvitele a legkényesebb művelet, hiszen az új rekord bekapcsolási módjával lényegében az is eldől, hogyan tudjuk majd az egyes rekordokat visszakeresni. Ha sokat keresünk és a felvitel nem túl gyakori, akkor felvitelkor mindent be kell vetnünk, kerül amibe kerül, hiszen ez kereséskor bőven megtérül. Ha viszont rengeteg adatot kell felvinnünk, ahol számít a felviteli idő, ráadásul csak néhányszor kell az adatokat visszaolvasni, akkor nem érdemes a szervezést elbonyolítani. Bizonyos szervezéseknél előfordulhat, hogy a sok felvitel és törlés miatt az adatkapcsolatok fizikai megvalósítása elbonyolódik. Sok esetben a gyorsaság kedvéért a felvitel, illetve törlés ideiglenesen valósul meg. Ezeket az állományokat időnként *újra* kell *szervezni*.

5.2 Azonosító, kulcs, elsődleges kulcs

A karbantartási és lekérdezési funkciók végrehajtásához mindig pontosan tudnunk kell, melyik rekordról van szó. Valahogyan el kell döntenünk, mi az a tulajdonság, illetve mik azok a tulajdonságok, melyek a rekordot egyértelműen azonosítják. Egy személyzeti nyilvántartásban például lehet két *Kis József* is az állományban, de ha törlésről van szó, akkor nem mindegy, hogy azt a személyt töröljük, aki a II. kerületben lakik, vagy azt, aki a XXII.-ben. Ha netán mindketten a II.-ban laknak, akkor nem mindegy, hogy azt töröljük, akinek 12000 Ft, vagy azt, akinek 100000 Ft a fizetése. A választás csak abban az esetben közömbös, ha a két *Kis József* összes adata megegyezik. A személyzeti nyilvántartásban – mivel sem a nevektől, sem egyéb személyes adatoktól nem várható el azok egyedisége, – mindenkinek kell, hogy legyen egy azonosító száma (törzsszáma), mely dolgozónként szigorúan más. A törzsszám a dolgozót egyértelműen azonosítja. A személyzeti nyilvántartás rekordjai tartalmazhatják például a következő mezőket:

- Törzsszám
- Név
- Beosztás
- Lakcím
- stb.

Azonosítónak (identifier) nevezzük azt a tulajdonságot, mely minden egyes rekordban előfordul, és melynek értéke minden rekordban más. Az azonosítót *egyedi azonosító*nak is szokás nevezni. A rekord azonosítója általában a rekord egy mezője, de összeállítható több mezőből is. Az azonosító elvileg lehet a rekord állománybeli pozíciója is, de ennek alkalmazása általában nem ajánlatos, mivel ilyen esetben az egyes rekordok fizikai helyhez kötöttek, következésképpen nem „hordozhatóak".

Sokszor nem csak egy szempont szerint kell az adatokat visszakeresni. Kereshetjük például a személyzeti nyilvántartásban azokat a rekordokat, melyben a név *Kis József*, vagy azokat, melyekben a beosztás *osztályvezető*. A keresési ismérvet *kulcs*nak (key) hívjuk. A kulcs lehet a rekord egy mezője, mint példánkban a *Név* vagy a *Beosztás*. Az azonosítót *elsődleges kulcs*nak (primary key) is szokás nevezni – ilyen például a *Törzsszám*.

5.3 Egyszerű karbantartás

Az 1. kötet „*Rendezések, keresések, karbantartás*" fejezetében szó volt az egydimenziós tömb karbantartásáról. A karbantartás lemezen sokkal „drágább", mint memóriában, hiszen az adatok mozgatásához az író/olvasófejeket állandó mozgásra kényszerítjük.

1. eset – Nem rendezett állomány, felvitel sorrendje nem számít

Ha az állomány nem rendezett, akkor az új rekordot mindig az állomány végére írjuk. Előtte ellenőrizni kell, volt-e már ilyen azonosítójú rekord az állományban. Törlés esetén meg kell keresni a törlendő rekordot, és ha van, akkor az utolsó rekordot a törölt helyére írjuk, és csonkítjuk az állományt. A karbantartásokat most „olcsón" megússzuk, de a keresés sokáig fog tartani, hiszen sikeres esetben az állományt átlagosan félig, sikertelen esetben pedig (mint például beszúrás előtt várhatóan,) az egész állományt végig kell olvasni.

2. eset – Nem rendezett állomány, felvitel sorrendje számít

Ha számít a felvitel sorrendje, akkor az új rekord felírása ugyanolyan jó, mint 1. esetben volt. Most azonban nem törölhetünk olyan „trükkösen", mint az előbb, hiszen ezzel a felvitel sorrendjét változtatnánk meg. A törlésre most két megoldás kínálkozik:

- Lejjebb mozgatjuk az egész állományt – drága megoldás.
- A rekordot csak logikailag töröljük. Ez azt jelenti, hogy a szóban forgó rekordot nem töröljük ki fizikailag a fél állomány lejjebb mozgatásával, hanem azt csak megjelöljük, és a legközelebbi kereséskor egyszerűen nem vesszük figyelembe.

A logikai törléshez minden rekordot kiegészítünk egy mezővel, mely azt az információt tartalmazza, hogy a rekord érvényes-e vagy törölt. Ez a mező – ha egyéb információt nem kell tartalmaznia, – általában logikai típusú.

Felvitelkor az *Érvényes* mezőt igazra kell állítani, törléskor pedig hamisra. Így kereséskor, listázáskor stb. meg tudjuk állapítani, hogy a rekord „élő", vagy sem. Ha nem

túl sok a logikailag törölt rekordok száma, akkor ez a keresést nem lassítja lényegesen. A logikailag törölt rekordokat az állomány újraszervezésével időnként fizikailag is törölni kell. Ilyenkor a logikailag törölt rekordok kihagyásával az állományt sűrítjük: az érvényes rekordokat átírjuk egy új állományba. Ha volt logikailag törölt rekord az állományban, akkor az új állomány rövidebb lesz:

```
Régi állomány nyitása
Új állomány létrehozása
Ciklus amíg nincs vége a Régi állománynak
   Következő rekord beolvasása a Régi állományból
   Ha Rekord.Érvenyes akkor
      Rekord kiírása az Új állományba
Ciklus vége
Régi állomány törlése
Új állomány átnevezése
```

3. eset – Rendezett állomány

Ha az állomány állandóan rendezett kell, hogy legyen, akkor beszúrás, illetve törlés esetén átlagosan a fél állományt eggyel feljebb, illetve lejjebb kell írnunk. Ez a megoldás rendkívül nehézkes, ezért kerülendő. Ha bevezetjük a logikai törlést, akkor a beszúrás még nehézkesebbé válik. Az állomány szervezésekor át kell gondolni, milyen módon akarjuk a felvitt adatokat visszakeresni. Ha az adatokat mindig egy konkrét rendezettségben szeretnénk feldolgozni, és a karbantartások száma a visszakeresésekhez képest elenyésző, akkor érdemes az állományt rendezett állapotban tartani, és az ezzel járó nehézségeket vállalni.

4. eset – Rendezett állomány, direkt szervezés, törlés nem lehetséges

Az adatok visszanyerése akkor a legegyszerűbb, ha a rekord azonosítója és állománybeli pozíciója közt egyértelmű megfeleltetés van, mert ekkor keresés nélkül, direkt módon a kérdéses rekordra pozicionálhatunk. Ha például a 23 kerület mindegyikéhez tartoznak adatok, akkor kézenfekvő, hogy az *1.* kerület adatait a *0.* rekordban, a *2.*-ét az *1.*-ben, a *23.* kerület adatait a *22.* rekordban tároljuk. Így a keresett rekordot közvetlenül elérjük, hiszen az *n.* kerület adatainak visszanyeréséhez egyszerűen az *n-1.* helyre pozicionálunk, és beolvassuk a rekordot. Ha feláldozzuk a 0. rekordot, akkor még transzformációra sincs szükség. Fizikai törlésről itt szó sem lehet, hiszen a rekordok mozgatásával elromlanának azok egyedi azonosítói. Új rekord felvitele is csak akkor lehetséges, ha annak azonosítója eggyel nagyobb, mint az eddigi legnagyobb. Ellenkező esetben figyelni kell az állományban lévő „lyukakat", hiszen ekkor nem minden azonosítóhoz tartozik rekord az állományban.

A direkt szervezés külön előnye, hogy a rekordok automatikusan állandó rendezettségben vannak, elősegítve ezzel a szekvenciális feldolgozásokat.

Nézzünk most egy feladatot a direkt szervezésre:

Feladat

Egy lakásépítő szövetkezet meghirdeti lakásait – összesen 100 darab egyforma lakás megvásárlására lehet jelentkezni. A jelentkezők adatait (név, lakcím) és a jelentkezési dátumot a jelentkezés sorrendjében jegyezni kell. Jelentkezéskor mindenki kap egy sorszámot, melyet a lakásakció során nem lehet többször kiadni. A végső elosztás előtt bárki visszaléphet, de a jelentkezők adatait az akció végéig nem lehet törölni. Módosítani a sorszám és a jelentkezési dátum kivételével minden adatot lehet, de ha valaki visszalépett, annak adatai nem módosíthatók, és csak új sorszámmal jelentkezhet újra. Sorszám alapján a jelentkezők adatai visszakereshetők, valamint kérésre a program listát készít arról a 100 jelentkezőről, akik megkapnák a kérdéses lakásokat.

Szervezzük meg az adatállományt, és adjuk meg a keresési és karbantartási funkciók stratégiáját!

A feladat alapján egy-egy jelentkezőről a következő adatokat kell nyilvántartanunk:

- Sorszám (1-től)
- Visszalépett (logikai)
- Jelentkezés dátuma
- Név
- Lakcím

Mivel a sorszámot a program adja és nem törölhető, az összeköthető a rekord állománybeli pozíciójával. Ha a sorszám egyről indul, akkor annak értéke *pozíció+1*. A biztonság kedvéért azonban a sorszámot adatmezőként betesszük a rekordba, hiszen könnyen elképzelhető például, hogy a nyerteseket át kell másolni egy másik állományba úgy, hogy az előzőleg kapott sorszámuk megmaradjanak. Állománybeli pozíciójukat a rekordok nem viszik magukkal.

Az állomány szervezése tehát legyen direkt: az egyedi azonosító mindig eggyel nagyobb, mint a rekord állománybeli pozíciója. A keresési és karbantartási algoritmusok pedig legyenek a következők:

Keresés sorszám szerint: A *Sorszám-1.* rekordra pozícionálunk, és beolvassuk a rekordot.

Új rekord felvitele: A rekord kitöltésekor *Sorszám* az állomány aktuális mérete, a *Jelentkezés dátuma* pedig a napi dátum lesz. A *Visszalépett* mezőt hamisra állítjuk. A *Név* és *Lakcím* adatokat a felhasználótól kérjük be. A rekord összeállítása után az állomány végére pozícionálunk, és felírjuk a rekordot.

Rekord törlése: Nem lehetséges.

Rekord módosítása: Bekérjük a sorszámot. Ha létező sorszámot adnak meg, akkor beolvassuk a rekordot az állományból. Ha a jelentkező visszalépett, akkor a módosítást nem végezzük el, egyébként kívánság szerint módosítjuk a rekordot. Visszalépés esetén a *Visszalépett* mezőt igazra állítjuk, egyébként bekérjük a *Név*, illetve *Lakcím* adatokat. A rekordot ugyanarra a pozícióra írjuk vissza.

100 nyertes listája: Az állományt sorosan olvassuk elölről. Csak azokat a rekordokat listázzuk és számoljuk, melyekben a *Visszalépett* mező értéke hamis. 100 jelentkező után megállunk.

5. eset – Rendezett állomány, direkt szervezés, törlés lehetséges

Az esetet egy feladaton keresztül mutatjuk be:

Feladat

Egy áruházban nyilvántartják a forgalmazott árukat. Az egyes árukhoz a következő adatok tartoznak:

- Árukód pontosan 4 karakter
- Leírás maximum 20 karakter
- Egységár valós

Az árukód a rekord egyedi azonosítója. Az árukódok A000, A001, A002 ... alakúak, és azok szórása kicsi, vagyis majdnem minden lehetséges kód ki van osztva. Az állományt folyamatosan karban kell tartani: új árukat felvinni, már létezőket törölni, illetve módosítani. Listát árukód szerinti rendezettségben gyakran, leírás szerinti rendezettségben pedig csak ritkán kell készíteni.

Szervezzük meg az állományt, és adjuk meg a szükséges algoritmusokat!

Mivel az árukódok és az egész számok (állomány pozíciója) közt létezik egy kölcsönösen egyértelmű megfeleltetés, ezért a rekordok árukód szerint direkt módon elérhetők:

A000 → 0
A001 → 1
...

Az állományban azonban elképzelhetők „lyukak". Ha például az A009-es árukód nem szerepel a nyilvántartásban, akkor az állomány 9. pozíciója „üres". A „lyukakat" nyilván kell tartani, hiszen tudni kell, hogy azon a helyen van-e egyáltalán rekord. Vegyünk fel egy *Érvényes* mezőt, és az állományt előre alakítsuk ki a várható méretben úgy, hogy az összes rekord *Érvényes* mezőjét hamisra állítjuk. Felvitel esetén e mezőt igazra, törléskor újra hamisra állítjuk. Egy adott árukódú rekord csak akkor létezik az állományban, ha *Érvényes* mezője igaz értékű.

Az állomány szervezése tehát direkt lesz.

Létrehozás: A várható méretben felviszünk logikailag törölt rekordokat.

Keresés: Elvégezzük a kulcstranszformációt. Ha a pozíció nem esik bele az állomány tartományába, akkor nincs ilyen rekord, egyébként pozícionálunk a megadott rekordra. Ha a rekord érvényes, akkor van, egyébként nincs ilyen rekord.

Felvitel: Ha a megfeleltetett pozíció nagyobb, mint az állomány mérete, akkor a szükséges méretben bővítjük az állományt érvénytelen rekordokkal, és felvisszük a rekordot. Egyébként keresünk, és ha van, hibaüzenetet adunk, egyébként *Érvényes* mezőt igazra állítjuk, és felvisszük a rekordot.

Törlés: Keresés. Ha van, *Érvényes* mezőt hamisra állítjuk, egyébként hibaüzenet.

Módosítás: Keresés. Ha van, akkor módosítjuk a rekordot és visszaírjuk, egyébként hibaüzenetet adunk. Rekordazonosító módosítása természetesen nem lehetséges.

Lista árukód szerint: Állomány végigolvasása szekvenciálisan. Az érvénytelen rekordokat nem vesszük figyelembe.

Lista leírás szerint: Az állomány érvényes rekordjait valamilyen módon egy új állományba rendezzük leírás szerint. A rendezett állományt kilistázzuk.

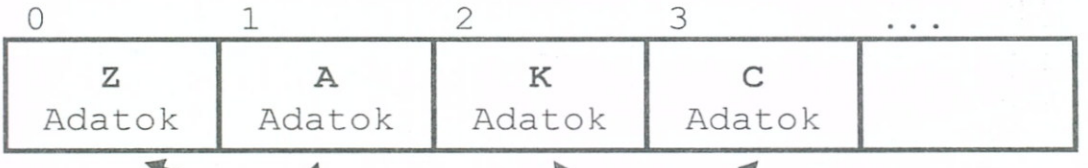
5.4 Karbantartás indextömb segítségével

Állományok direkt szervezése aránylag ritka, hiszen a kulcsok legtöbb esetben teljesen véletlenszerűek. Egy nagy állományban való soros keresés hatékonysága nem kielégítő, főleg akkor, ha az nem rendezett. Rendezett állomány esetén a karbantartás is nagy gondot okoz. Ha még több szempont szerint is kell keresnünk, akkor egyenesen lehetetlenné válhat helyzetünk. Az indextömb használata ugyan még távol áll a „profi" állományszervezési fogásoktól, de már lényeges előrelépés a keresési és karbantartási műveletek hatékonysága felé, és már magában foglalja azokat az alapvető meggondolásokat, melyek a komolyabb állományszervezési módszerek megértéséhez feltétlenül szükségesek.

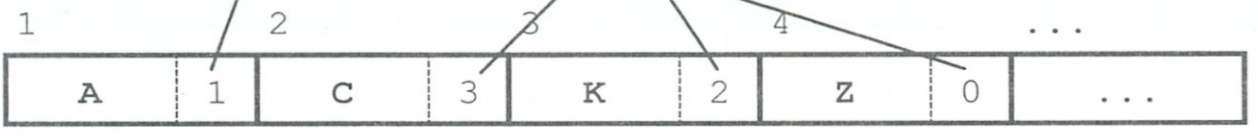
Indextömbről, illetve kulcstömbről már szó volt az *„Állományok rendezése, összeválogatása"* című fejezet 2. pontjában, amikor az állomány rendezését indextömb segítségével végeztük el. Most ugyanezt az ötletet használjuk fel a karbantartáshoz is. Mivel az adatállomány (törzsállomány) rekordjait nehézkes mozgatni, azok fizikai helyét az állományban a felírás után nem változtatjuk meg. A rendezést közvetett módon, *indextömb* segítségével érjük el. Az indextömb a memóriában helyezkedik el (feltételezzük, hogy befér oda), mely csak a rendezettség meghatározásában résztvevő adatokat (kulcsokat) tartalmazza. Minden kulcs mellett szerepel a rekord többi adatának állománybeli pozíciója (indexe). Az indextömb kulcs szerint rendezett.

Példaként tekintsünk egy olyan állományt, melyben a kulcsok karakterek:

Adatállomány:



Indextömb:

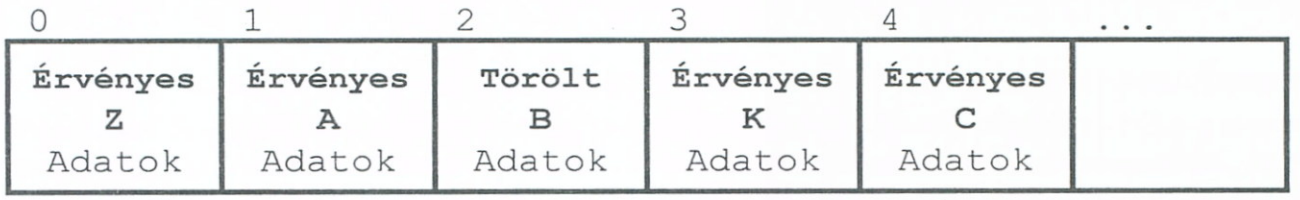


Ettől kezdve a kereséseket a rendezett indextömbben végezzük el. Ha a keresett kulcsot megtaláljuk, akkor az index alapján beolvashatjuk a kulcshoz tartozó adatokat a törzsállományból. A keresés memóriában történik, így az nem vesz sok időt igénybe. Ennek persze alapfeltétele, hogy az indextömb beférjen a memóriába. Új rekord felvitelénél az adatállomány végére írunk, és a kulcsrekordot rendezetten beszúrjuk az indextömbbe. Törlés esetén a kulcsrekordot egyszerűen kitöröljük az indextömbből. Mivel mindig az indextömbben keresünk, a kitörölt kulcsot soha többé nem találjuk meg, tehát olyan, mintha az a rekord már nem is létezne. A rekord azonban továbbra is ott foglalja a helyet a lemezen, ezért az állományt időnként újra kell szervezni. Ez jelen esetben azt jelenti, hogy a meglévő indextömb alapján az élő rekordokat beolvassuk, és azokat egy új állományba sorban kiírjuk. Eközben a pozíciók megváltozása miatt az indextömböt is módosítjuk.

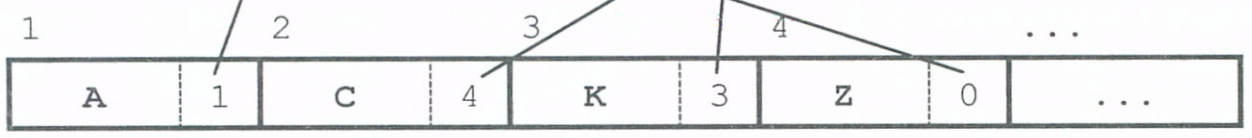
Az indextömböt a memóriában tároljuk, és azon folyamatosan változtatásokat eszközölünk. Ahhoz, hogy az indextömböt legközelebb is használhassuk, azt a programból való kilépés előtt lemezre kell másolnunk, majd a legközelebbi felhasználáskor ismét be kell töltenünk a memóriába. Az indextömb lemezre mentett változatát *indexállomány*nak nevezzük. Elvileg jó megoldás lenne, ha a használat elején a törzsállományból építenénk fel az indextömböt. Ez azonban most lehetetlen, hiszen a rekordok között ott vannak a töröltek is, melyeket nem jelöltünk meg, és így csak az indextömbből lehet információnk az élő rekordokat illetően. De e megoldásnak van egy másik óriási veszélye is. A memóriában tárolt indextömb sérülékeny, hiszen a memória tartalma egy programleállás vagy újraindítás esetén elvész. Az indextömb ezenkívül programozói hiba miatt is elromolhat. Fontos szabály tehát, hogy a törzsállományból ne hiányozzék semmilyen információ. Az indextömb csak a kereséseket gyorsító kisegítő adatszerkezet, mely az adatállomány alapján bármikor újra felépíthető. Ezért

- a törzsrekordnak tartalmaznia kell a kulcsot, és
- a törölt rekordokat meg kell jelölni.

Adatállomány:



Indextömb:



Most nézzük meg az állomány struktúráját, valamint az állománykezelési műveletek algoritmusait:

Az adatállomány struktúrája: A törzsrekordokat típusos állományban tartjuk nyilván. A rekordok adatai tartalmaznak egy *Érvényes* logikai változót, mely jelzi, hogy a rekord érvényes-e vagy törölt. Az adatállományhoz tartozik egy indextömb, mely kulcsrekordokat tartalmaz. Minden kulcsrekord tartalmaz egy kulcsot, és a hozzá tartozó törzsrekord állománybeli pozícióját (indexét).

Létrehozás: Ha az adatállomány még nem létezik, akkor azt létrehozzuk, és az indextömböt inicializáljuk (Kulcsok száma=0). Ha létezik, akkor annak nyitásával egyidejűleg az indextömböt beolvassuk a memóriába. Ha az indextömb a lemezen nem található, akkor azt felépítjük az adatállomány alapján.

Keresés: A keresést az indextömbben végezzük. Mivel az indextömb kulcs szerint rendezett, a keresést akár szekvenciálisan, akár binárisan elvégezhetjük. Ha megtaláltuk a keresett kulcsot, akkor a hozzátartozó index alapján beolvashatjuk a törzsrekordot.

Felvitel: Először bekérjük a felhasználótól a felviendő rekord azonosítóját. Rákeresünk a kulcsra az indextömbben. Ha megtaláltuk, akkor a felvitelt nem tudjuk elvégezni, erről üzenetet adunk a felhasználónak. Ha eddig még nincsen ilyen elsődleges kulcs az állományban, akkor bekérjük a rekord többi adatát. Az *Érvényes* mezőt igazra állítjuk. Az összeállított rekordot az állomány végére írjuk. A kulcsrekordot, melybe betesszük a kulcsot és az állománybeli pozíciót, kulcs szerinti rendezettségben beszúrjuk az indextömbbe.

Törlés: Bekérjük a felhasználótól a törlendő rekord azonosítóját. Rákeresünk a kulcsra az indextömbben. Ha a kulcsot nem találjuk, akkor a törlést nem tudjuk elvégezni, erről üzenetet adunk a felhasználónak. Ha van ilyen kulcs, akkor először az index alapján beolvassuk a törzsrekordot, az *Érvényes* mezőt hamisra állítjuk, és visszaírjuk a lemezre. Ezután a kulcsrekordot kitöröljük az indextömbből.

Módosítás: Bekérjük a felhasználótól a módosítandó rekord azonosítóját. Rákeresünk a kulcsra az indextömbben. Ha nincs meg, akkor a módosítást nem tudjuk elvégezni, erről üzenetet adunk a felhasználónak. Ha megvan, akkor megengedjük a felhasználónak, hogy módosítsa a rekordot. Most két eset lehetséges:

- Nem engedjük meg az elsődleges kulcs megváltoztatását. Ez az egyszerűbb eset, ekkor nincs más teendő, mint a megváltoztatott rekordot visszaírni az eredeti pozícióra.
- Megengedjük az elsődleges kulcs megváltoztatását is. Ebben az esetben a törzsrekord visszaírása után az indextömbből ki kell törölni az eredeti kulcsrekordot, és be kell szúrni egy újat az új kulccsal, de eredeti állománypozícióval. Az új kulcs nem lehet olyan, ami már létezik az állományban.

Lista elsődleges kulcs szerinti rendezettségben: Szekvenciálisan végigolvassuk az indextömböt, és a megfelelő indexek alapján sorban beolvassuk a törzsrekordokat. A megadott mezőket kilistázzuk.

Újraszervezés (sűrítés): Az újraszervezéssel mind a törzsállomány, mind pedig az indextömb meg fog változni. A feladat kétféleképpen is elvégezhető:

- Ha a logikailag törölt rekordok száma nem túl nagy, akkor egyszerűbb a törzsállományt sorosan végigolvasni, és az érvényes rekordokat sorban felírni az új állományba. Az új állomány annyival lesz rövidebb, amennyi logikailag törölt rekord volt a régiben. Közben az indextömböt építjük: a kulcsokat és a hozzá tartozó indexeket sorban, rendezve beszúrjuk.
- Ha nagyon sok a logikailag törölt rekord, akkor érdemesebb a beolvasást a régi indextömb alapján elvégezni, hiszen ekkor a törölt rekordok nem kerülnek beolvasásra. Az indextömbben az indexeket módosítjuk.

Az újraszervezés az állomány nagyságától és bonyolultságától függően hosszabb ideig is eltarthat, ezért nem mindegy, hogy azt mikor végezzük el – ügyeljünk arra, hogy a felhasználót lehetőleg ne várassuk. Legjobb, ha a programból való kilépéskor szervezzük az állományt, és akkor is csak szükség esetén – például ha a logikailag törölt rekordok száma elér egy bizonyos százalékot. A logikailag törölt rekordok száma megállapítható, hiszen mind a törzsállomány fizikai mérete, mind az indextömb kulcsainak száma lekérdezhető.

Az állomány szervezésénél nagyon fontos szempont, hogy a törzsállomány ne tudjon megsérülni. Minél „fontosabb" egy állomány, annál több biztonsági intézkedést kell ennek érdekében tenni. Nem tehetjük meg például, hogy egy kisebb állományt, amely befér a memóriába, a program elején beolvasunk, a karbantartást memóriában végezzük el, majd a program befejeztével visszaírjuk lemezre. Nem engedhetjük meg, hogy egy esetleges programhiba, vagy egy nem várt esemény miatt elvesszen a törzsállományunk. Memóriába csak gyors megtekintésre vihetjük be az állományt, változtatást azon nem szabad eszközölni!

A fent leírt elméletet most egy konkrét programmal valósítjuk meg. A feladat a következő:

Feladat

Írjunk programot, mely az árukat tartalmazó adatállomány karbantartását indextömb segítségével végzi el!

Az egyes árukhoz a következő adatok tartoznak:

- Árukód pontosan 4 karakter
- Leírás maximum 20 karakter
- Egységár valós

Az árukód a rekord egyedi azonosítója.

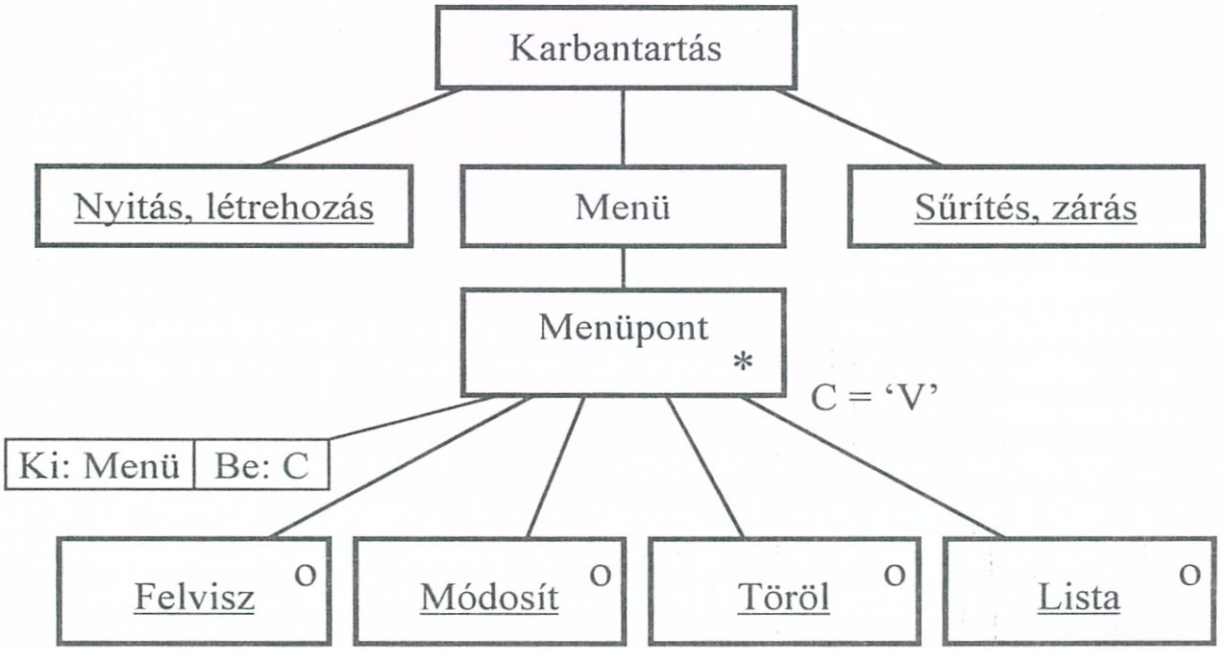
Menüből a következő funkciók hívhatók:

- Felvitel
- Módosítás
- Törlés
- Árukód szerinti lista

Először a programtervet készítjük el nagy vonalakban:

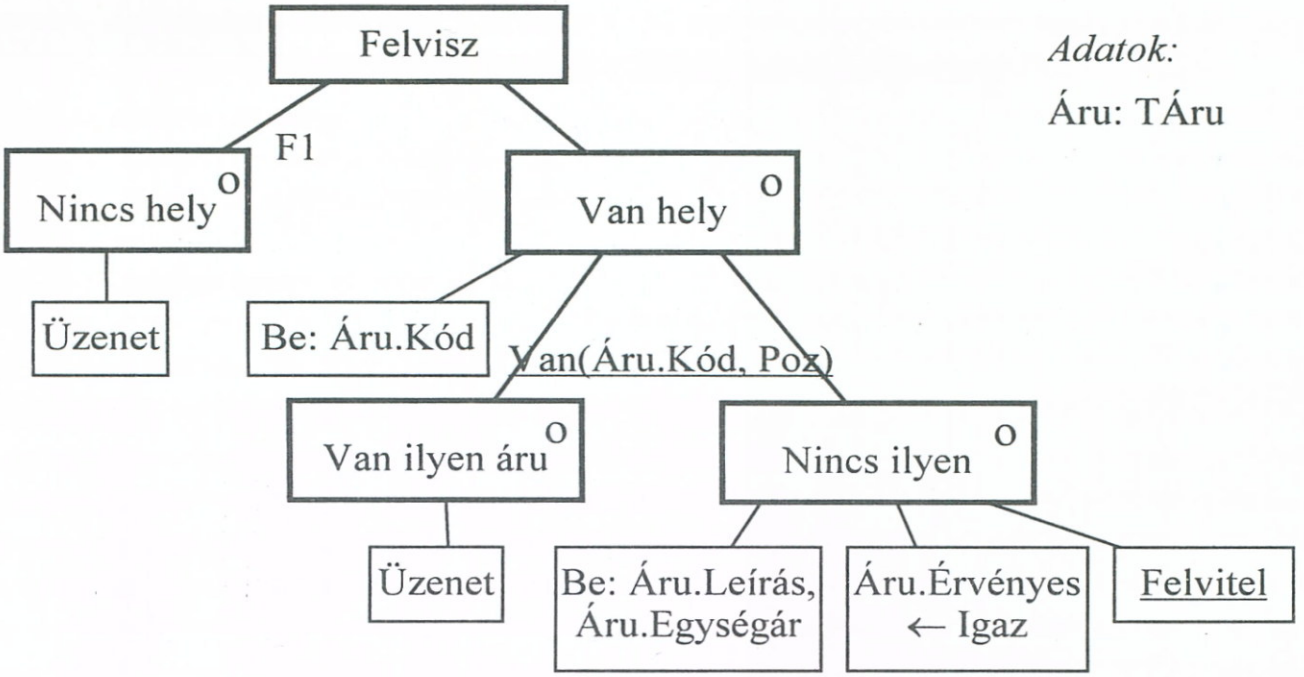
Adatok:

- TÁru = Rekord (Érvényes, Kód, Leírás, Egységár)
- TKulcsRek = Rekord (Kód, Index)
- KulcsSzám: Érvényes kulcsok száma.
- MaxKulcsSzám = 1000. Maximálisan felvihető kulcsok száma.
- Áruk: Állomány (TÁru). Törzsállomány.
- IndexTömb: Tömb(1.. MaxKulcsSzám: TKulcsRek)
- IndexÁll: Indextömb másolata lemezen.



Tevékenységjegyzék:

- Nyitás, létrehozás: Cél, hogy legyen egy megnyitott Áruk állomány, és legyen hozzá egy IndexTömb a memóriában. Elsősorban a már létező Áruk állományt nyitjuk meg, és beolvassuk a hozzátartozó IndexÁll-t. Ha nincs Áruk, létrehozzuk. Ha nincs meg hozzá az IndexÁll, akkor az IndexTömb-öt felépítjük Áruk-ból (IndexFelépít eljárás). KulcsSzám beállítása.
- Sűrítés, zárás: Áruk állományt lezárjuk. Ha túl sok a törölt rekord (>80%), akkor előbb sűrítjük. Kiírjuk az indexállományt lemezre (IndexKiír eljárás).
- Felvisz: Lásd következő oldal.
- Módosít, Töröl, Lista: Ezen menüpontoknak csak a forráslistáját adjuk meg.



Feltételjegyzék:

- F1: KulcsSzám >= MaxKulcsSzám
- Van(Kód,Poz): Logikai. Visszaadja, hogy van-e IndexTömb-ben Kód. Ha igen, akkor Poz a Kód pozíciója, ha nem, akkor Poz az a pozíció, ahova be kellene szúrni Kód-ot.

Tevékenységjegyzék:

- Felvitel: Ki(Áruk): Áru. Az összeállított Áru rekordot felviszi az Áruk állományba az utolsó, Index. helyre. A megfelelő kulcsrekordot (Kód, Index) beszúrja az Indextömbbe a Poz. helyre. (KulcsBeszúr eljárás).

Turbo Pascal kód:

```
Program Karbantartas ;
Uses Crt ;
Const
  MaxKulcsSzam = 1000 ;
Type
  TKod = String[4] ;
  TAru = Record
    Ervenyes : Boolean ;
    Kod : TKod ;
    Leiras : String[20] ;
    Egysegar : Real ;
  End ;

  TKulcsRek = Record
    Kod : TKod ;
    Index : LongInt ;
  End ;
```

```
Var
  Aruk : File Of TAru ;
  IndexAll : File Of TKulcsRek ;
  IndexTomb : Array [1..MaxKulcsSzam] Of TKulcsRek ;
  KulcsSzam : LongInt ;
```

Az indextömböt normális esetben úgy építjük fel, hogy a lemezre mentett változatot (az indexállományt) beolvassuk. A kulcsokat közben számoljuk. Az indexállományt a főprogramban nyitottuk meg, amikor annak létezését vizsgáltuk. A kulcsok beolvasása után lezárjuk az állományt, arra csak a program végén lesz újra szükségünk.

```
Procedure IndexBeolvas ;
  Begin
    KulcsSzam := 0 ;
    While Not Eof(IndexAll) Do
      Begin
        Inc(KulcsSzam) ;
        Read(IndexAll,IndexTomb[KulcsSzam]) ;
      End ;
    Close(IndexAll) ;
  End ;
```

Az indextömböt a feldolgozás végén egyszerűen lemezre másoljuk:

```
Procedure IndexKiir ;
  Var
    I : LongInt ;
  Begin
    Rewrite(IndexAll) ;
    For I := 1 To KulcsSzam Do
      Write(IndexAll,IndexTomb[I]) ;
    Close(IndexAll) ;
  End ;
```

A *Van* függvény a rendezett indextömbben keresi a megadott kódot. Ha van, akkor a függvény igaz értéket ad vissza, egyébként hamisat. *Poz* értéke a megtalált kód pozíciója a tömbben, illetve ha nincs ilyen kód, akkor *Poz* az a pozíció, ahova a kódot be kellene szúrni.

```
Function Van(Kod: TKod; Var Poz: LongInt): Boolean ;
  Begin
    Poz := 1 ;
    While (Poz<=KulcsSzam) And (Kod>IndexTomb[Poz].Kod) Do
      Inc(Poz) ;
    Van:=(Poz<=KulcsSzam) And (Kod=IndexTomb[Poz].Kod);
  End ;
```

A *KulcsBeszur* eljárás a megadott kódot és indexet (kulcsrekordot) beszúrja az indextömb *Poz*-adik helyére. A megadott kódnál nagyobb kulcsokat eggyel feljebb tolja, és a kulcsok számát eggyel megnöveli:

```
Procedure KulcsBeszur(Kod: TKod; Index, Poz: LongInt) ;
  Var
    I : LongInt ;
  Begin
    For I := KulcsSzam Downto Poz Do
      IndexTomb[I+1] := IndexTomb[I] ;
    IndexTomb[Poz].Kod := Kod ;
    IndexTomb[Poz].Index := Index ;
    Inc(KulcsSzam) ;
  End ;
```

Az indextömb felépítésére akkor lesz szükség, ha az adatállomány megvan, de az indexállomány valamilyen ok folytán elveszett. Az index felépítését sok esetben felhasználói kérésre végzi a program olyan esetekben, amikor az indexállomány akár programhiba akár felhasználói gondatlanság miatt megsérült. A sérülést a felhasználó az adatok furcsa viselkedéséből gyaníthatja:

```
Procedure IndexFelepit ;
  Var
    I, Poz : LongInt ;
    Aru : TAru ;
  Begin
    KulcsSzam := 0 ;
    For I := 0 To Filesize(Aruk)-1 Do
      Begin
        Read(Aruk,Aru) ;
        If Aru.Ervenyes And Not Van(Aru.Kod,Poz) Then
          KulcsBeszur(Aru.Kod,I,Poz) ;
     End ;
  End ;
```

Újraszervezésre akkor van szükség, ha már túl sok a logikailag törölt rekord. Az eljárás egy új törzsállományt épít fel a törölt rekordok elhagyásával, és ezzel párhuzamosan módosítja az indextömböt:

```
Procedure Ujraszervez ;
  Var
    I : LongInt ;
    UjAruk : File Of TAru ;
    Aru : TAru ;
```

```
  Begin
    Assign(UjAruk,'Uj.Dat') ;
    Rewrite(UjAruk) ;
    For I := 1 To KulcsSzam Do
      Begin
        Seek(Aruk,IndexTomb[I].Index) ;
        Read(Aruk,Aru) ;
        Write(UjAruk,Aru) ;
        IndexTomb[I].Index := I-1 ;
      End ;
    Close(Aruk) ;
    Close(UjAruk) ;
    Erase(Aruk) ;
    Rename(UjAruk,'Aruk.Dat') ;
  End ;
```

A *Felvisz* eljárás egy új áru adatait kéri be terminálról, és viszi fel a törzsállományba. Ezzel párhuzamosan az új kulcsot az indextömbbe is beszúrja. A felvitel nem hajtható végre, ha nincs hely az indextömbben, vagy a megadott kód már létezik:

```
Procedure Felvisz ;
  Var
    Aru : TAru ;
    Index, Poz : LongInt ;
  Begin
    If KulcsSzam >= MaxKulcsSzam Then
      Begin
        Write('Nincs több hely a memóriában!');
        Exit;
      End ;

    Write('Árukód: ') ; ReadLn(Aru.Kod) ;
    If Van(Aru.Kod,Poz) Then
      Write('Van már ilyen áru!')
    Else
      Begin
        Write('Leírás: ') ; ReadLn(Aru.Leiras) ;
        Write('Egységár: ') ; ReadLn(Aru.Egysegar) ;
        Aru.Ervenyes := True ;
        Index := FileSize(Aruk) ;
        Seek(Aruk,Index) ;
        Write(Aruk,Aru) ;
        KulcsBeszur(Aru.Kod,Index,Poz) ;
      End ;
  End ;
```

Módosításkor csak a törzsrekordot változtatjuk meg és írjuk vissza a lemezre. A módosítás csak akkor hajtható végre, ha a megadott árukód létezik az indextömbben. A kulcs módosítását nem engedjük meg:

```
Procedure Modosit ;
  Var
    Aru: TAru ;
    Poz : LongInt ;
  Begin
    ReadLn(Aru.Kod) ;
    If Van(Aru.Kod,Poz) Then
      Begin
        Seek(Aruk,IndexTomb[Poz].Index) ;
        Read(Aruk,Aru) ;
        With Aru Do
          Begin
            WriteLn(Kod:5,Leiras:22,Egysegar:10:2);
            Write('Leírás: '); ReadLn(Aru.Leiras);
            Write('Egységár: '); ReadLn(Aru.Egysegar);
          End ;

        Seek(Aruk,IndexTomb[Poz].Index) ;
        Write(Aruk,Aru) ;
      End
    Else
      WriteLn('Nincs ilyen áru!') ;
  End ;
```

Törléskor – ha létezik a megadott kód, – az index alapján beolvassuk a törzsrekordot, abban az *Ervenyes* mezőt False-ra állítjuk, majd visszaírjuk a lemezre. Az indextömbből egyszerűen kitöröljük a kulcshoz tartozó rekordot, és a kulcsok számát eggyel csökkentjük.

```
Procedure Torol ;
  Var
    I, Poz : LongInt ;
    Aru: TAru ;
  Begin
    ReadLn(Aru.Kod) ;
    If Van(Aru.Kod,Poz) Then
      Begin
        Seek(Aruk,IndexTomb[Poz].Index) ;
        Read(Aruk,Aru) ;
        Aru.Ervenyes := False ;
        Seek(Aruk,IndexTomb[Poz].Index) ;
        Write(Aruk,Aru) ;
```

```
            For I := Poz To KulcsSzam-1 Do
              IndexTomb[I] := IndexTomb[I+1] ;
            Dec(KulcsSzam) ;
          End
        Else
          WriteLn('Nincs ilyen áru!') ;
    End ;
```

Az árukód szerinti lista úgy készül, hogy az indextömböt szekvenciálisan végigolvasva a kulcsokhoz tartozó indexek alapján olvassuk be a törzsrekordokat. A rekordok adatait soronként kilistázzuk:

```
Procedure Listaz ;
  Var
    I : LongInt ;
    Aru: TAru ;
  Begin
    For I := 1 To KulcsSzam Do
      Begin
        Seek(Aruk,IndexTomb[I].Index) ;
        Read(Aruk,Aru) ;
        With Aru Do
          WriteLn(Kod:5,Leiras:22,Egysegar:10:2) ;
      End ;
  End ;
```

A főprogram három részből áll:

- Az első rész feladata a törzsállomány és az indextömb inicializálása. Ha a törzsállomány nem létezik, akkor egy üreset hozunk létre a lemezen – ekkor az indexállományt meg sem nézzük, hiszen úgysem tudunk mit kezdeni vele. Ha létezik, akkor beolvassuk a hozzátartozó indexállományt. Ha az indexállomány nem létezik, akkor felépítjük az indextömböt a törzsállományból. A törzsállomány az egész program során nyitva lesz.
- A második részben menüből hívhatók a különböző karbantartási és listázási funkciók.
- Befejezésként sűrítjük az állományt, ha a logikailag törölt rekordok száma meghaladja a 20%-ot. Végül kiírjuk az indextömböt a lemezre.

```
Var
  C : Char ;

Begin
  { Nyitás, illetve létrehozás: }
  Assign(Aruk,'Aruk.Dat') ;
  Assign(IndexAll,'Kod.Idx') ;
  {$I-} Reset(Aruk) ; {$I+}
  If IOResult = 0 Then
```

```
      Begin
        {$I-} Reset(IndexAll) ; {$I+}
        If IOResult = 0 Then
          IndexBeolvas
        Else
          IndexFelepit ;
      End
    Else
      Begin
        Rewrite(Aruk) ;
        KulcsSzam := 0 ;
      End ;

    { Menü: }
    Repeat
        WriteLn('F(elvitel)') ;
        WriteLn('M(ódosítás)') ;
        WriteLn('T(örlés)') ;
        WriteLn('L(ista)') ;
        WriteLn('V(ége)') ;
        C := Upcase(ReadKey) ;
        Case C Of
          'F' : Felvisz ;
          'M' : Modosit ;
          'T' : Torol ;
          'L' : Listaz ;
        End ;
    Until C = 'V' ;

    { Szükség esetén sűrítés, majd zárás: }
    If FileSize(Aruk) * 0.8 > KulcsSzam Then
      Ujraszervez
    Else
      Close(Aruk) ;
    IndexKiir ;
End.
```

5.5 Karbantartás tranzakciós állománnyal

Az eddigiekben kizárólag interaktív (párbeszédes) karbantartásról volt szó. Interaktív karbantartás esetén a karbantartó műveleteket a törzsállományon azonnal átvezetjük, hiszen a felhasználó az adatokon eszközölt változtatást rögtön látni szeretné. Nem mindig van azonban szükség az azonnali változtatásra, sőt, van olyan eset, amikor ez nem is lehetséges. Előfordul, hogy az óhajtott változtatásokat gyűjteni kell, és az összegyűjtött karbantartó műveleteket a törzsállományon egyszerre kell végrehajtani. A karbantartó műveletet *tranzakció*nak is szokás nevezni. Ezért a lemezen összegyűjtött

karbantartó műveleteket *tranzakciós állomány*nak hívjuk. A tranzakciós állomány tranzakcióit egy menetben vezetjük át a törzsállományon. A teljesíthetetlen tranzakciós kéréseket hibalistára szokás írni.

Mikor végezzük a karbantartást tranzakciós állománnyal?

- Ha a tranzakciók rögzítésekor nincs „kéznél" a törzsállomány;
- Ha nem szükséges az azonnali átvezetés.

Ha elegendően sok a tranzakciók száma, akkor a tranzakciós állományból történő karbantartás lényegesen felgyorsulhat az interaktív végrehajtáshoz képest, mert ha a tranzakciós állományt rendezzük, akkor az a törzsállománnyal párhuzamosan feldolgozható. Így a törzsállomány és a tranzakciós állomány egyszeri végigolvasásával a feladat végrehajtható. Az állományok összeválogatását az *„Állományok rendezése, összeválogatása"* fejezetben részletesen tárgyaltuk.

A tranzakciós rekord mindig tartalmaz egy mezőt, mely a *tranzakciós kód*ot tartalmazza. A tranzakciós kódnak ki kell fejeznie, hogy a tranzakciós igény felvitel, törlés vagy módosítás.

A következő ábra egy törzsállomány aktualizálását szemlélteti tranzakciós állomány segítségével. A törzsállomány és a tranzakciós állomány azonosító szerint rendezett. A tranzakciós kódok a következők:

- F : Felvitel
- T : Törlés
- M : Módosítás

Az új törzsállomány mutatja az aktualizálás utáni állapotot. Hibalistára azok a tranzakciós rekordok kerülnek, melyeket nem lehet átvezetni:

- Felviteli kérelem, ha már létezik a megadott azonosító;
- Törlési, illetve módosítási kérelem, ha nem létezik a megadott azonosító.

Törzsállomány:

Azonosító →	A	B	D	K	L	M	P	R
Adatok →	Aadat	Badat	Dadat	Kadat	Ladat	Madat	Padat	Radat

Tranzakciós állomány:

Tranz.kód →	T	M	F	F	T	M
Azonosító →	B	D	E	K	P	Q
Adatok →	-	TDadat	TEadat	TKadat	-	TQadat

Új törzsállomány:

Azonosító →	A	D	E	K	L	M	R
Adatok →	Aadat	TDadat	TEadat	Kadat	Ladat	Madat	Radat

Hibalista:

Tranz.kód	*Azonosító*	*Adatok*	*Üzenet*
F	K	TKadat	Van már ilyen azonosító
M	Q	TQadat	Nincs ilyen azonosító

Nézzük most a feladat megoldásának általános algoritmusát:

```
Törzs és Tranz állományok nyitása olvasásra
Újtörzs nyitása írásra
Olvasás Törzsből
Olvasás Tranzból

Ciklus amíg van Törzs rekord vagy van Tranz rekord
  Elágazás
    Törzs.Azonosító < Tranz.Azonosító esetén
      { Nincs Törzs rekordhoz Tranz rekord }
      Törzs rekord írása Újtörzsbe
      Olvasás Törzsből
    Törzs.Azonosító > Tranz.Azonosító esetén
      { Nincs a Tranz rekordhoz Törzs rekord }
      Ha Tranz.Kod = 'F' akkor
        Újtörzs rekord összeállítása Tranz alapján,
        és felírása Újtörzsbe
      egyébként {ha Tranz.Kod = 'T' vagy 'M' }
        Hiba, nincs ilyen azonosítójú rekord a törzsben
      Elágazás vége
      Olvasás Tranzból
    Egyéb esetben (Törzs.Azonosító = Tranz.Azonosító)
      { Van a Tranz rekordhoz Törzs rekord }
      Elágazás
        { Tranz.Kód = 'T' esetén
          Törzs rekordot nem visszük át Újtörzsbe }
        Tranz.Kód = 'M' esetén
          Törzs rekord módosítása Tranz alapján, felírása
          Újtörzsbe
        Tranz.Kód = 'F' esetén
          Hiba, van már ilyen azonosítójú Törzs rekord
          Törzs rekord felírása Újtörzsbe
      Elágazás vége
      Olvasás Törzsből
      Olvasás Tranzból
  Elágazás vége
Ciklus vége
Állományok lezárása
```

Kérdések

1. Mit jelentenek a következő fogalmak?
   a) Állományszervezés, állomány struktúrája
   b) Állománykezelési műveletek
   c) Létrehozás, visszakeresés, karbantartás, újraszervezés
   d) Felvitel, módosítás, törlés
   e) Azonosító, kulcs, elsődleges kulcs
   f) Logikai törlés, fizikai törlés
   g) Indextömb, indexállomány
   h) Törzsállomány, tranzakciós állomány
2. Milyen szempontokat kell követni egy állomány szervezésénél?
3. Milyen célt szolgál a logikai törlés, és miért hasznos annak bevezetése?

Feladatok

1. Egy kórházban a betegeket számítógép segítségével tartják nyilván. A betegekről a következő adatokat kell tudni:
   - Felvételi sorszám (egyedi azonosító, a program adja, nem ismétlődhet)
   - Név
   - Kórterem, ágy
   - Felelős orvos
   - Diagnózis
   - Lakcím
   - Felvétel napja
   - Távozás napja

   A programban lehetőség van új beteg felvételére, régi beteg adatainak módosítására (a felvételi sorszámot kivéve). A beteget törölni interaktív módon nem lehet, csak a távozás napját lehet beírni, ami kezdetben üres.
   A program a napi feldolgozás végén az összes beteget, aki 8 napnál régebben eltávozott, átírja egy archív állományba, és innen kitörli őket.

   Kérésre a program listát ad
   - az összes betegről felvételi sorszám szerint rendezetten;
   - az összes betegről névsor szerint rendezetten;
   - egy adott orvos felügyelete alá tartozó összes betegről;
   - egy adott kórterem betegeiről.
2. Egy autókereskedelem telephelyén annyi autó van, hogy a kereskedők nem győzik az adminisztrációt számítógép nélkül. Készítsen egy programot a kereskedők számára, mellyel az autókat nyilván tudják tartani, a nyilvántartott adatokat módosítani tudják, és különböző szempontok szerint le tudják kérdezni az adatokat. Az autókról a rendszámot, típust, az árat és a nyilvántartásba vétel idejét min-

denképpen nyilván kell tartani. A rendszám egyedi azonosító. Kérésre a program produkálja a következő listákat:

- az összes autó listája rendszám szerint
- az összes autó listája típus szerint
- egy adott típushoz tartozó összes autó listája
- egy adott árkategóriába eső összes autó listája

Próbálja megoldani úgy a feladatot, hogy a kereskedő minél nagyobb örömét lelje a programban.

3*. Egy bank az ügyfelek adatait az *Ugyfelek.Dat* törzsállományban tartja nyilván. A rekordkép a következő:

- Számlaszám String[10]
- Név String[30]
- Lakcím String[40]
- Egyenleg Real
- Utolsó mozgás dátuma String[6], EEHHNN alakú

A törzsállományban az egyenleg lehet negatív is.

Az állományhoz tartozik egy indexállomány: *Szamla.Idx*, mely számlaszám szerint rendezett, és a duplikáció nem megengedett (a számlaszám elsődleges kulcs). Az indexállomány rekordképe a következő:

- Számlaszám String[10]
- Index LongInt (a törzsrekord rekordszáma)

Az indexállomány akkora, hogy feltételezhetően nem fér be a memóriába!

A *Mozgas.Dat* az aznapi mozgásokat tartalmazza, rekordképe:

- Számlaszám String[10]
- Mozgáskód Char, B vagy K (betét vagy kivét)
- Összeg Real

Ha a mozgáskód *B*, akkor a számlára befizettek, *K* esetén levették a megadott összeget. A mozgás állomány számlaszám szerint növekvőleg rendezett.
Aktualizáljuk a törzsállományt a mozgás állomány alapján! Ha a mozgáskódhoz tartozik számlaszám a törzsállományban, akkor betét esetén tegyük rá a megadott összeget a számlára, kivét esetén vegyük le róla! Az utolsó mozgás dátuma az aznapi dátum legyen, melyet a program elején kérjünk be. Ha a mozgáshoz nem tartozik számlaszám a törzsállományban, akkor írjuk azt hibalistára!

Érdemes tanulmányozni

Angster Erzsébet-Kertész László: Turbo Pascal feladatgyűjtemény I.:
Típusos állományok fejezet, Tipall10 és Tipall14 programok

6. CSOPORTVÁLTÁS

A fejezetben először tisztázzuk a csoportváltás fogalmát, majd egy-, illetve kétszintű csoportváltásos, illetve összegfokozatos listákat fogunk készíteni.

6.1 A fogalom tisztázása

Gyakran találkozunk olyan feladattal, melyben az állomány rekordjait csoportosítani kell a rekord valamely mezőjének különböző értékei szerint. Tegyük fel például, hogy egy személyzeti nyilvántartás alapján listát kell készíteni az összes alkalmazottról foglalkozás szerinti bontásban. Az egyforma foglalkozású dolgozók a listán legyenek jól elkülönítve úgy, hogy minden csoport elején szerepeljen a közös foglalkozás, a csoport végén pedig a csoportba tartozó dolgozók száma. A lista végén írjuk ki az összes dolgozó számát.

Képzeljük el a problémát konkrét adatokkal. A dolgozók adatait a *Dolgozok.Dat* állomány tartalmazza, ahol az egyes rekordok a következő mezőkből állnak: *Törzsszám, Név, Foglalkozás, Szül.év* és *Fizetés*.

Egy ilyen feladatot egy menetben csak úgy lehet megoldani, ha a kérdéses állomány a csoportosítási szempont szerint rendezett, hiszen a listán ugyanolyan sorrendben jelennek meg a rekordok, mint ahogyan azok a feldolgozandó állományban szerepelnek. Mivel a csoportosítást foglalkozás szerint szeretnénk elvégezni, ezért a beviteli állományt a *Foglalkozás* mező szerint rendezzük. Az áttekinthetőség miatt a foglalkozásokon belül a rendezést névre is elvégezzük, bár ez a csoportok kialakításában nem játszik szerepet. Legyen az állomány konkrét tartalma a következő:

Törzsszám	*Név*	*Foglalkozás*	*Szül.év*	*Fizetés*
77244	Boldog Tihamér	Fizikus	1970	90000
12000	Keres Judit	Fizikus	1951	31000
81112	Pálinkás Elek	Fizikus	1970	55000
85466	Rendes Erzsébet	Matematikus	1960	50000
11111	Földes Mária	Nyelvész	1946	93000
94444	Néhai Bálint	Nyelvész	1951	35000
99924	Boros Éva	Programozó	1956	25000
12222	Dolgos Zsófia	Programozó	1956	28000
22133	Hegyes József	Programozó	1970	25000
23111	Szomorú Dávid	Programozó	1956	61000
90000	Utolsó Tamás	Programozó	1970	62000

A listában most elmarad a foglalkozás oszlop, hiszen ezt az adatot „kiemeltük" a csoport elejére. A *Dolgozok.Dat* állomány alapján tehát a lista így fog kinézni:

Dolgozók listája foglalkozásonként

Fizikus:

Törzsszám	*Név*	*Szül.év*	*Fizetés*
77244	Boldog Tihamér	1970	90000
12000	Keres Judit	1951	31000
81112	Pálinkás Elek	1970	55000

Fizikus összesen: 3

Matematikus:

Törzsszám	*Név*	*Szül.év*	*Fizetés*
85466	Rendes Erzsébet	1960	50000

Matematikus összesen: 1

Nyelvész:

Törzsszám	*Név*	*Szül.év*	*Fizetés*
11111	Földes Mária	1946	93000
94444	Néhai Bálint	1951	35000

Nyelvész összesen: 2

Programozó:

Törzsszám	*Név*	*Szül.év*	*Fizetés*
99924	Boros Éva	1956	25000
12222	Dolgos Zsófia	1956	28000
22133	Hegyes József	1970	25000
23111	Szomorú Dávid	1956	61000
90000	Utolsó Tamás	1970	62000

Programozó összesen: 5

Dolgozó összesen: 11

Lista vége

Az ilyen feladatok megoldásához tehát először is rendezni kell a beviteli állományt a csoportosítási szempont szerint. Ez általában a rekord egy mezőjét jelenti, de elképzelhető összetett mező, csonkított mező vagy számolt érték is. A rendezett beviteli állományt végigolvassuk, s közben *figyeljük* a rekordokat, hogy a csoportot meghatározó érték *megváltozott-e*. Ha a beolvasott rekord már egy következő csoporthoz tartozik, akkor az előző értékhez tartozó csoportot le kell zárni, és egy új csoportot kell nyitni. Példánkban a figyelt mező a *Foglalkozás*.

A *csoportváltás* elnevezés onnan ered, hogy a csoportokat figyelni, és a váltásra reagálni kell. Szokásos még a *kontrollváltás* elnevezés is, mert egy adott mező értékeit állandóan ellenőrizzük (kontrolláljuk). Egy olyan csoportváltást, melyben csak egy

értéket kell figyelni, egyszintű csoportváltásnak nevezünk. A csoportváltás azonban több szintű (fokozatú) is lehet. Képzeljük el például, hogy foglalkozáson belül születési évenként is csoportosítani kell a dolgozókat. Ekkor a rendezést foglalkozáson belül születési évre kell elvégezni. Az áttekinthetőség kedvéért születési évenként természetesen most is rendezhetjük az állományt név szerint. Itt a csoportváltás két szintű, hiszen a foglalkozáson belül a születési évet is figyelni kell, és változás esetén a szükséges teendőket el kell végezni. A következő listán az egyforma foglalkozású dolgozók születési évek szerint is csoportosítva vannak. A születési évet most az első oszlopba tesszük, és az egykorú dolgozók csoportjaihoz a születési évet csak egyszer írjuk ki:

Dolgozók listája foglalkozásonként és születési évenként

Fizikus:

Szül.év	*Törzsszám*	*Név*	*Fizetés*
1951	12000	Keres Judit	31000
1970	77244	Boldog Tihamér	90000
	81112	Pálinkás Elek	55000

Fizikus összesen: 3

Matematikus:

Szül.év	*Törzsszám*	*Név*	*Fizetés*
1960	85466	Rendes Erzsébet	50000

Matematikus összesen: 1

Nyelvész:

Szül.év	*Törzsszám*	*Név*	*Fizetés*
1946	11111	Földes Mária	93000
1951	94444	Néhai Bálint	35000

Nyelvész összesen: 2

Programozó:

Szül.év	*Törzsszám*	*Név*	*Fizetés*
1956	99924	Boros Éva	25000
	12222	Dolgos Zsófia	28000
	23111	Szomorú Dávid	61000
1970	22133	Hegyes József	25000
	90000	Utolsó Tamás	62000

Programozó összesen: 5

Dolgozó összesen: 11

Lista vége

A csoportváltásos feladat nem feltétlenül listázást jelent, sőt, az sem fontos, hogy az input egy állomány legyen. E fogalom csupán azt jelenti, hogy *egy rendezett beviteli sorozatot úgy dolgozunk fel, hogy közben figyelünk bizonyos értékeket, és váltás ese-*

tén megtesszük a szükséges intézkedéseket. Azokat a csoportváltásos feladatokat, melyek csoportonként összegzéseket végeznek, *összegfokozatos feladatok*nak nevezzük.

6.2 Egyszintű csoportváltás

Próbáljuk most megoldani az előző pontban már bevezetett egyszintű csoportváltásos feladatot:

Feladat
Adott a *Dolgozok.Dat* állomány a következő rekordképpel:

- Törzsszám String[5] (számjegyek)
- Név String[20]
- Foglalkozás String[15]
- Szül. év Word
- Fizetés LongInt

Az állomány rendezett foglalkozás, azon belül név szerint. Listázzuk ki az állományt foglalkozásonként csoportosítva. A lista képernyőre készüljön, a listakép legyen a következő:

```
                Dolgozók listája foglalkozásonként

Foglalkozás
Törzsszám    Név           Szül.év    Fizetés
99999        XXXX...XXXX    1999      9999999
99999        XXXX...XXXX    1999      9999999
99999        XXXX...XXXX    1999      9999999
...
Foglalkozás összesen: 999

Foglalkozás
Törzsszám    Név           Szül.év    Fizetés
99999        XXXX...XXXX    1999      9999999
99999        XXXX...XXXX    1999      9999999
99999        XXXX...XXXX    1999      9999999
...
Foglalkozás összesen: 999
.
.
.
Dolgozó összesen: 9999
                      Lista vége
```

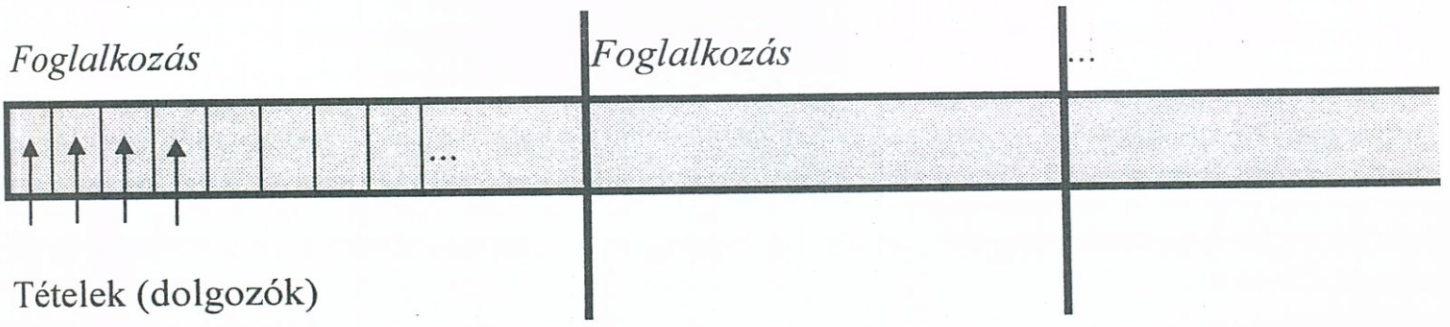
Első megközelítésben eltekintünk attól, hogy a lista nem fér ki egy képernyőlapra – a listát soronként, folyamatosan fogjuk készíteni.

Próbáljuk végiggondolni a megoldást lépésről lépésre az előző pontban megadott adatokkal. A program menete a következő lesz:

1. Kiírjuk a lista címét, nullázzuk az összesen gyűjtőt, majd beolvassuk az első dolgozó rekordját. Mivel Boldog Tihamér egy fizikus, tudjuk, hogy egy fizikus csoport kezdődik.
2. A csoport elé tehát már ki is írjuk a foglalkozást és a dolgozók adatainak fejlécét. A foglalkozás szerinti gyűjtőt nullázzuk. Kiírjuk a beolvasott dolgozó adatait is.
3. Beolvassuk a következő dolgozót. Mielőtt kilistáznánk, megvizsgáljuk, hogy ő is fizikus-e még. Keres Judit szintén fizikus, tehát adatait kilistázzuk Boldog Tihamér után. Ugyanez történik Pálinkás Elekkel is. A foglalkozás szerinti gyűjtőt közben növeljük. A ciklusnak akkor van vége, ha észrevesszük, hogy a beolvasott rekord Foglalkozás mezője nem fizikus. Rendes Erzsébet már matematikus, így lezárjuk a fizikus csoportot: kiírjuk a gyűjtött létszámot és egy üres sort írunk. A fizikusok gyűjtött létszámát hozzáadjuk az összesen gyűjtőhöz. A matematikusokkal hasonlóképpen járunk el, mint a fizikusokkal, tehát visszamegyünk a 2. ponthoz. A ciklusnak akkor lesz vége, ha nincs több dolgozó.
4. Ha nincs több dolgozó, akkor lezárjuk az utolsó csoportot, és az egész listát is befejezzük. Kiírjuk az összesen gyűjtő tartalmát, és a „lista vége" szöveget.

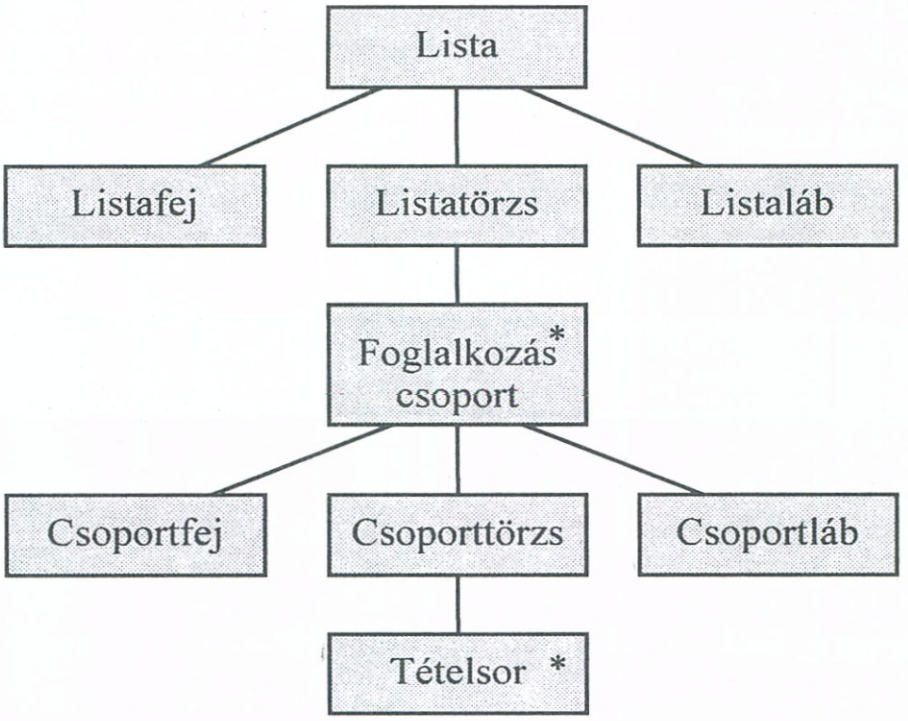
A feladatot **Michael Jackson** programtervezési módszerével oldjuk meg. Határozzuk meg először a feladat megoldásának durva algoritmusát! Nyilvánvalóan a programban olyan ciklusokat kell elhelyezni, melyek a Dolgozók állományt végigolvassák, és a fenti listát produkálják. Ezért első dolgunk, hogy megvizsgáljuk a feldolgozandó állomány és a lista szerkezetét.

A *Dolgozok.Dat* állomány sematikus rajza a következő:



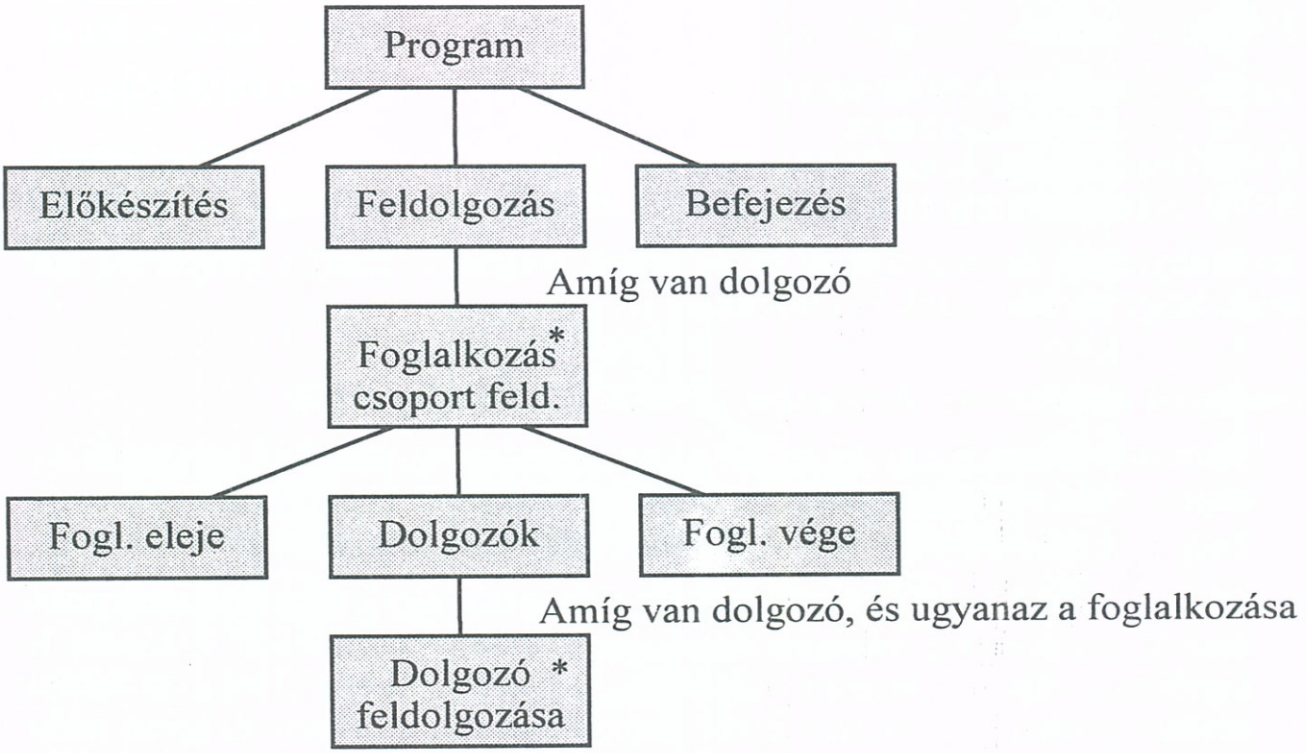
Az állomány szerkezetét Jackson jelöléssel így ábrázoljuk:

Most nézzük a készítendő listát. Látható, hogy a lista elején egyszer szerepel a *„Dolgozók listája foglalkozásonként"* cím – ezt a *lista fejléc*ének nevezzük. A lista végén egyszer szereplő kiírás a *lista lábléc*e – ez az összes dolgozó létszáma, valamint a lista végének jelzése. Ha leszámítjuk a lista elején és végén kiírt információkat, akkor a *lista törzs*ét kapjuk. A listatörzs logikailag egyforma csoportokból áll. Egy csoport szerkezete a következő: a csoport elején szerepel a *csoport fejléc*e, melyet a *csoport törzs*e követ, és a *csoport lábléc*e zár be. A csoport fejlécéhez hozzátartozik az aktuális foglalkozás, valamint az oszlopok elnevezéseinek kiírása. A lábléc a foglalkozás összesen értékének kiírása. A csoporttörzs logikailag egyforma sorokból áll, mindegyik sor egy dolgozó adatait tartalmazza. Mivel egy sor a legkisebb kiírásra kerülő logikai egység, ezeket a sorokat *tételsor*oknak nevezzük. A fejlécet és láblécet egyszerűen fejnek, illetve lábnak is nevezhetjük. A lista szerkezetét Jackson jelöléssel a következőképpen ábrázolhatjuk:



A lista fejlécének kiírása a program elején, láblécének kiírása pedig a program végén egy egyszer végrehajtandó programrészben fog szerepelni. A csoportfej, illetve csoportláb kiírása foglalkozás-csoportonként egyszer szerepel. Gyanítható, hogy a lista szerkezete tökéletes lesz arra, hogy programszerkezetként felhasználva elkészítse listánkat. Jackson programtervezésének egyébként pont ez a lényege: az adatszerkezetek segítenek bennünket a program szerkezetének kialakításában. Sőt, a bemenő, illetve kimenő adatszerkezetek összefésüléséből legtöbb esetben egyszerűen adódik a program szerkezete. A csoportváltásos feladattípus ennek egyik legfényesebb példája.

Mivel a kimeneti adatszerkezet a bemeneti adatszerkezetet tartalmazza, ezért jelen esetben a program szerkezete a következő lesz:



Ha megvan a programszerkezet, akkor meg kell fogalmazni a feltételeket, és meg kell vizsgálni azok kiértékelhetőségét. A külső ciklus addig tart, amíg van csoport, vagyis *amíg van dolgozó* a beviteli állományban. Ugyanis ha van dolgozó, akkor az beletartozik egy csoportba. A belső ciklus addig tart, amíg a dolgozó foglalkozása ugyanaz, mint az előzőé. Természetesen a ciklus akkor is befejeződik, ha nincs több dolgozó. Általános szabályként megfogalmazhatjuk a következőt: *csoportváltásos feladatoknál a belső ciklusfeltételek mindig öröklik a külső feltételeket.* Ez természetes, hiszen a külső csoport befejeződése mindig maga után vonja a belső csoport lezárását is – a nagy csoport kisebb csoportokból áll, amely újabb csoportokból áll stb.

Nézzük most meg, ki tudjuk-e értékelni a megfogalmazott feltételeket?

- *Amíg van dolgozó:* Ha az állománynak vége van, akkor nincs több dolgozó, egyébként van. Ezt a feltételt tehát ki tudjuk értékelni.
- *Amíg van dolgozó, és ugyanaz a foglalkozása:* Ahhoz, hogy egy dolgozó foglalkozását összehasonlíthassuk egy előző dolgozó foglalkozásával, ahhoz az előző dolgozó foglalkozását meg kell jegyeznünk. A döntéshez tehát egy előkészítő műveletet kell végrehajtanunk. De ha ezt megtesszük, akkor a feltételt ki tudjuk értékelni. Ezt a dolgot azonban át kell majd újra gondolni akkor, amikor az állományból való olvasás és döntéselőkészítés helyét meghatározzuk a programban.

Következő lépésként gyűjtsük össze a végrehajtandó elemi tevékenységeket, és tegyük a megfelelő „dobozok" alá. A tevékenységeket számokkal fogjuk jelölni, mert szövegesen nem férnének el az ábrán. A tevékenységeket jellegük szerint a következő csoportokba oszthatjuk[1]:

- Állományok előkészítése, befejezése, olvasás az állományokból
- Döntéselőkészítések
- Számítások és azok előkészítései
- Írások és azok előkészítései

A tevékenységek összegyűjtését és helyének meghatározását a csoportok szerint végezzük, hogy ne felejtsünk ki semmit:

Állományok előkészítése, befejezése, olvasás az állományokból:

1. *Dolgozók állomány nyitása:* Mivel ezt egyetlen egyszer kell megtenni a program elején, ezért e tevékenységet az *Előkészítés* dobozba tesszük.
2. *Dolgozók állomány zárása:* A *Befejezés* dobozba tesszük.
3. *Olvasás a Dolgozók állományból:* Ez a program egyik legkényesebb pontja. Sokszor hasznos dolog végiggondolni, hogy az egyes dobozok összesen hányszor hajtódnak végre a program folyamán. A legalsó doboz (Dolgozó feldolgozása) például pontosan annyiszor kerül végrehajtásra, mint ahány dolgozó rekord van a *Dolgozók* állományban. Ezért kizárt például, hogy a *Fogl. eleje* dobozban olvasás történjen, hiszen akkor foglalkozásonként még pontosan egy plusz olvasás hajtódna végre. Az olvasást tehát egészen biztos a *Dolgozó feldolgozása* dobozban kell elhelyezni. De ha csak itt van olvasás, akkor a felette álló feltétel induláskor kiértékelhetetlen, hiszen azon a ponton még egyetlen rekordot sem olvastunk be, következésképpen nincs megjegyzett foglalkozás. Kénytelenek vagyunk ezért egy rekordot előolvasni az *Előkészítés* dobozban. Ezután már természetes, hogy a *Dolgozó feldolgozása* doboznak a végére kell tennünk a további olvasásokat (ezek előolvasások lesznek a következő ciklusokhoz). Összességében tehát eggyel több olvasás lesz, mint ahány rekord van a beviteli állományban. Az utolsó olvasáskor már nem olvasunk be rekordot – az egy sikertelen kísérlet lesz.

[1] A tevékenységek meghatározásának és elhelyezésének sorrendjét a francia Jean-Dominique Warnier alkalmazta először saját programtervezési módszerében.

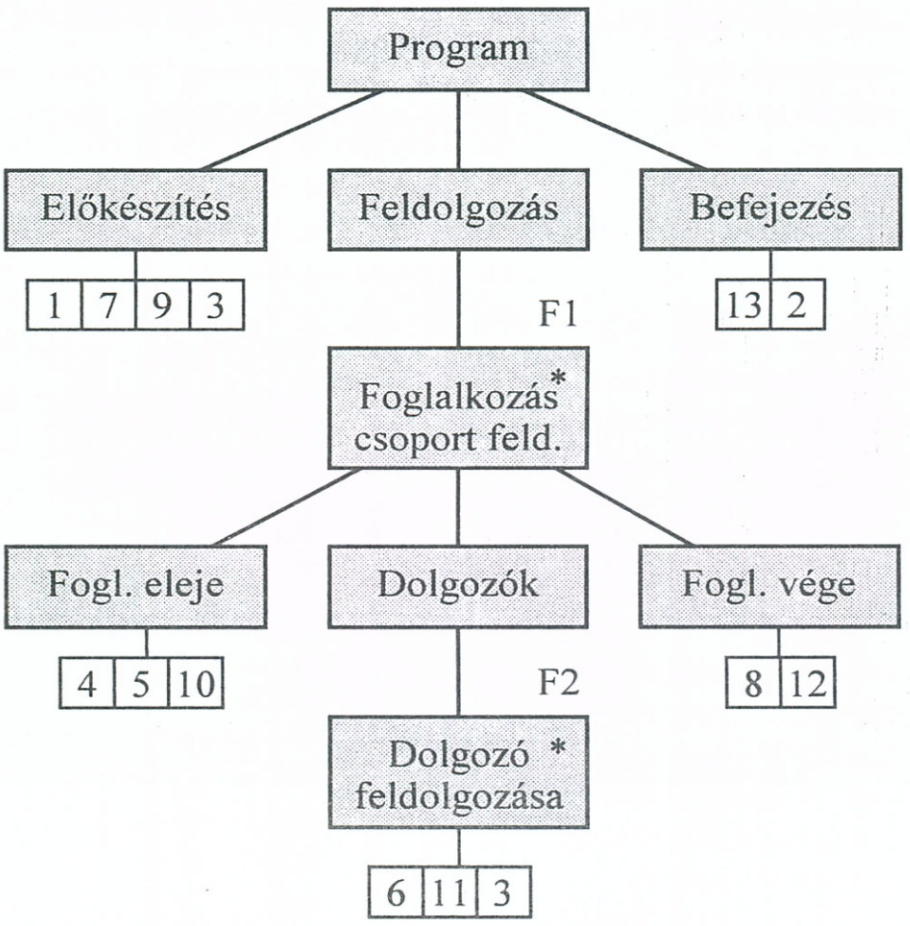
Döntéselőkészítések:

4. *Aktuális foglalkozás ← Dolgozó.Foglalkozás:* Mivel ezt a tevékenységet foglalkozásonként egyszer kell végrehajtani, e tevékenységnek a *Fogl. eleje* dobozban van a helye.

Számítások és azok előkészítései:

Végig kell nézni a listát, milyen számolt értékek kell, hogy megjelenjenek. A programnak természetesen ezeket az értékeket kell kiszámolnia.

5. *Foglalkozás összesen gyűjtő nullázása:* Foglalkozásonként egyszer, *Fogl. eleje* dobozban.
6. *Foglalkozás összesen gyűjtő növelése:* Dolgozónként egyszer, a *Dolgozó feldolgozása* dobozban. Ezen a dobozon ugyanis minden dolgozó „keresztül megy".
7. *Összesen gyűjtő nullázása:* A program elején egyszer, az *Előkészítés* dobozban.
8. *Összesen gyűjtő növelése:* Foglalkozásonként egyszer, a *Fogl. vége* dobozban. Ekkor vagyunk ugyanis a foglalkozás összesen végleges eredményének birtokában.



Írások és azok előkészítései:

Végig kell nézni a lista sorait, és egyenként megvizsgálni, hogy azok a program mely részében kerülhetnek kiírásra.

9. *Listafej kiírása: Előkészítés* dobozban.
10. *Foglalkozás elejének kiírása:* A foglalkozás az *Aktuális foglalkozás*, a második sor konstans. Ezt a *Fogl. eleje* dobozban kell kiírni.
11. *Tételsor kiírása:* A *Dolgozó feldolgozása* dobozban.
12. *Foglalkozás végének kiírása:* Itt a foglalkozás gyűjtő értékét kell kiírni, természetesen a *Fogl. vége* dobozban. Az érték elé írjuk az *Aktuális foglalkozás* értékét is.
13. *Listaláb kiírása:* Itt írjuk ki az összesen gyűjtő értékét, és a *Lista vége* szöveget – természetesen a *Befejezés* dobozban.

Bizonyos tevékenységek elhelyezésére a dobozokon belül van egy ajánlott sorrend, melyet érdemes betartani:

- Döntéselőkészítések
- Számításelőkészítések
- Számítások
- Íráselőkészítések
- Írások
- Olvasások (csak előolvasás esetén)

A programszerkezet elkészítése után most készítsük el a terv kódját. A teljes programkódot majd a következő pontban, a kétszintű csoportváltásra fogjuk megadni. Nézzük most a megoldás mondatszerű leírását:

```
Dolgozók állomány nyitása
Összesen gyűjtő nullázása
Listafej kiírása
Olvasás a Dolgozók állományból
Ciklus amíg Van még dolgozó
   Aktuális foglalkozás ← Dolgozó.Foglalkozás
   Foglalkozás összesen gyűjtő nullázása
   Foglalkozás elejének kiírása
   Ciklus amíg Van még dolgozó és
      (Aktuális foglalkozás = Dolgozó.Foglalkozás)
      Foglalkozás összesen gyűjtő növelése
      Tételsor kiírása
      Olvasás a Dolgozók állományból
   Ciklus vége
   Összesen gyűjtő növelése
   Foglalkozás végének kiírása
Ciklus vége (Van még dolgozó)
Listaláb kiírása
Dolgozók állomány zárása
```

6.3 Kétszintű csoportváltás

Oldjuk meg most az első pontban már bevezetett kétszintű csoportváltásos feladatot:

Feladat

Adott a *Dolgozok.Dat* állomány a következő rekordképpel:

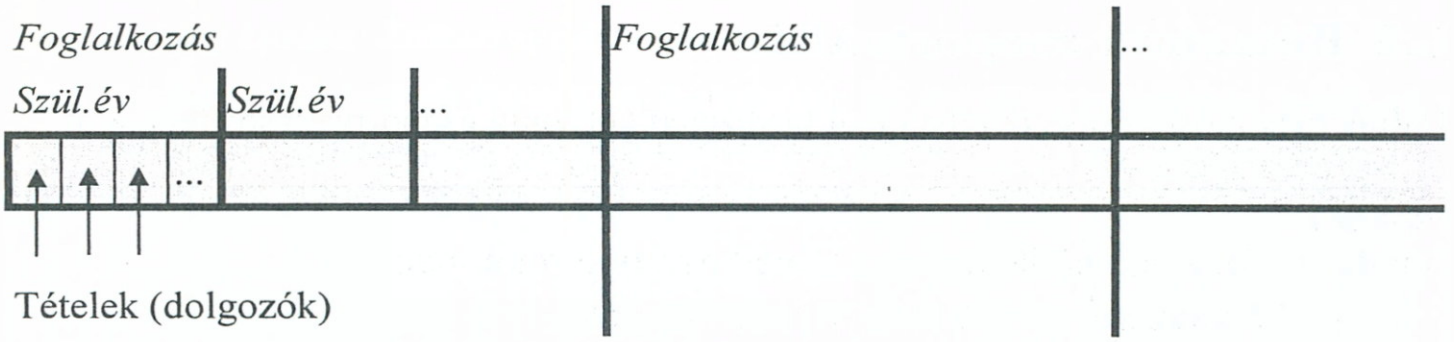
- Törzsszám String[5] (számjegyek)
- Név String[20]
- Foglalkozás String[15]
- Szül. év Word
- Fizetés LongInt

Az állomány rendezett foglalkozás, azon belül születési év, azon belül pedig név szerint. Listázzuk ki az állományt foglalkozásonként, azon belül születési évenként csoportosítva. A születési évet csak abban az esetben írjuk ki, ha új csoport kezdődött! A lista képernyőre készüljön úgy, hogy minden 20 kiírt sor után megállunk. A lapokat számozzuk!

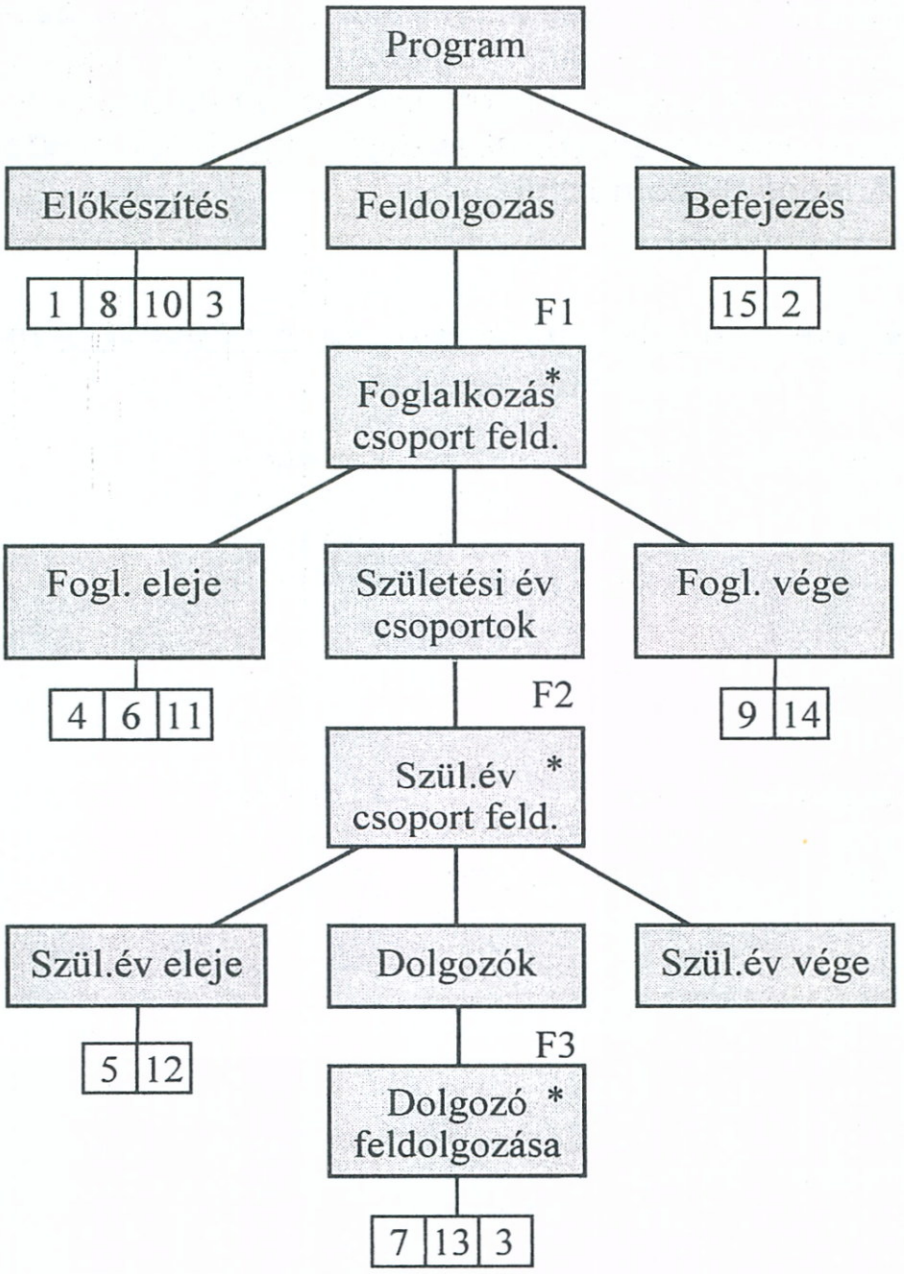
A listakép legyen a következő:

```
Dolgozók listája foglalkozásonként és születési évenként

Foglalkozás
Szül.év   Törzsszám    Név           Fizetés
 1999     99999        XXXX...XXXX   9999999
          99999        XXXX...XXXX   9999999
          99999        XXXX...XXXX   9999999
...
Foglalkozás összesen: 999

Foglalkozás
Szül.év   Törzsszám    Név           Fizetés
 1999     99999        XXXX...XXXX   9999999
          99999        XXXX...XXXX   9999999
 1999     99999        XXXX...XXXX   9999999
...
Foglalkozás összesen: 999
.
.
.
Dolgozó összesen: 9999
                  Lista vége
```

Születési év szerint most csak azért van csoportváltás, mert annak értékét csak a csoport elején kell kiírni. A *Dolgozok.Dat* állományban most a foglalkozás csoportok születési év csoportokat tartalmaznak:



Nézzük most a program tervét. Az előző feladathoz képest a szintek száma eggyel növekszik – a két ciklus közé egy születési év szerinti iteráció ékelődik be:



Ha jól belegondolunk, a lista szerkezetének felállításánál a tételsor most nem egyértelmű – az vagy tartalmaz születési évet, vagy nem. A tételsor tehát lehet ilyen is, meg olyan is. De úgy is felfoghatjuk a lista idevonatkozó részét, hogy a *Születési év* csoport elején van egy *Születési év* adat, majd ezután következnek a *Törzsszám, Név* és *Fizetés* adatcsoportok. Ez utóbbi csoportokat csak úgy tudjuk egységesen kiírni, ha a kiírás előtt pozicionálhatunk. Képernyő esetén ezt minden további nélkül megtehetjük, így a születési évet a *Szülév eleje* dobozban, a dolgozó többi adatát pedig a *Dolgozó feldolgozása* dobozban írjuk ki.

A képernyőn való lapozást a következőképpen oldjuk meg: ha Jackson szigorú útmutatásait követnénk, akkor a listának lapozás szerint is lenne egy struktúrája. A csoportokra és a lapokra való tördelést egyszerre azonban lehetetlen ábrázolni. Ezt a problémát Jackson a struktúraütközések feladatkörébe sorolja. Ennek tárgyalása azonban meghaladja e könyv lehetőségeit, ezért érjük most be a megoldás egyszerű közlésével: a lapozást úgy intézzük el, hogy minden egyes kiírás előtt megnézzük, ráfér-e a kiírandó sor a lapra, vagy sem. Ha igen, akkor kiírjuk, egyébként a lapot „befejezzük", várunk egy billentyű leütésére, majd új lapot kezdünk. Ha hagyunk annyi helyet a lapon, hogy a befejező sorok arra ráférjenek, akkor elegendő csak a fejléceknél és a tételsoroknál figyelni a sorok számát. Ez azért is jó, mert így nem kerül új lapra egy befejező információ.

Feltételjegyzék:

F1 Amíg van dolgozó

F2 Amíg van dolgozó és (Aktfoglalkozás = Dolgozó.Foglalkozás)

F3 Amíg van dolgozó és (Aktfoglalkozás = Dolgozó.Foglalkozás) és (Aktszülév = Dolgozó.Szülév)

Tevékenységjegyzék:

1. Dolgozók állomány nyitása
2. Dolgozók állomány zárása
3. Olvasás a Dolgozók állományból
4. Aktfoglalkozás ← Dolgozó.Foglalkozás
5. Aktszülév ← Dolgozó.Szülév
6. Fogl. összesen ← 0
7. Fogl. összesen $\overset{+}{\leftarrow}$ 1
8. Összesen ← 0
9. Összesen $\overset{+}{\leftarrow}$ Fogl. összesen
10. Listafej kiírása
11. Foglalkozás elejének kiírása
12. Aktszülév kiírása
13. Törzsszám, Név, Fizetés kiírása
14. Foglalkozás végének kiírása
15. Listaláb kiírása

A *Szülév vége* dobozba végül is nem került semmi, ezért azt el is hagyhatnánk. Ha születési évenként átlagfizetést kellett volna például számolni, akkor annak kiírása itt lenne esedékes.

A programterv alapján most elkészítjük e feladat Turbo Pascal forráskódját. Az olvasás kódolásánál a következőre kell vigyázni: a Turbo Pascal-ban állomány vége akkor következik be, amikor még az utolsó rekordot be sem olvastuk. A mi megoldásunknál viszont az lenne jó, ha az állomány vége feltétel csak akkor állna be, ha már egy eredménytelen beolvasást végeztünk. Ezt úgy tudjuk a kódban realizálni, hogy az állomány vége feltételt is eggyel késleltetjük. Lássuk most a kódot:

```
Program Lista ;

Uses
  Crt ;

Type
  TDolgozo = Record
    Torzsszam : String[5] ;
    Nev : String[20] ;
    Fogl : String[15] ;
    Szulev : Word ;
    Fizetes : LongInt ;
  End ;

Var
  Dolgozok : File Of TDolgozo ;
  Dolgozo : TDolgozo ;
  Vege : Boolean ;
  Osszesen,
  FoglOsszesen : LongInt ;
  AktFogl : String[15] ;
  AktSzulev : Word ;
  Lapszamlalo,
  Sorszamlalo : Word ;

Procedure Olvas ;
  Begin
    Vege := Eof(Dolgozok) ;
    If Not Vege Then
      Read(Dolgozok,Dolgozo) ;
  End ;
```

```
Procedure UjLap ;
  Begin
    ClrScr ;
    Inc(Lapszamlalo) ;
    WriteLn(Lapszamlalo,'. lap') ;
    WriteLn ;
    Sorszamlalo := 2 ;
  End ;

Procedure Lapvizsgal ;
  Begin
    If Sorszamlalo > 20 Then
      Begin
        WriteLn ;
        Write('ENTER - Tovább') ;
        Repeat
        Until ReadKey = #13 ;
        UjLap ;
      End ;
  End ;

Begin { Főprogram }
  Assign(Dolgozok,'Dolgozok.Dat') ;
  Reset(Dolgozok) ;
  Osszesen := 0 ;
  Lapszamlalo := 0 ;
  UjLap ;
  WriteLn('Dolgozók listája foglalkozásonként' +
    ' és születési évenként') ;
  WriteLn ;
  Inc(Sorszamlalo,2) ;
  Olvas ;
  While Not Vege Do
    Begin
      AktFogl := Dolgozo.Fogl ;
      FoglOsszesen := 0 ;
      Lapvizsgal ;
      WriteLn(AktFogl) ;
      WriteLn('Szül.év Törzsszám     Név' +
      '                    Fizetés') ;
      Inc(Sorszamlalo,2) ;
```

```
        While Not Vege And (AktFogl = Dolgozo.Fogl) Do
          Begin
            AktSzulev := Dolgozo.Szulev ;
            Lapvizsgal ;
            Write(AktSzulev:5) ;
            While Not Vege And (AktFogl = Dolgozo.Fogl)
              And (AktSzulev = Dolgozo.Szulev) Do
              Begin
                Inc(FoglOsszesen) ;
                Lapvizsgal ;
                GotoXY(10,WhereY) ;
                With Dolgozo Do
                  WriteLn(Torzsszam,'':6,
                    Nev,'':22-Length(Nev),Fizetes:8) ;
                Inc(Sorszamlalo) ;
                Olvas ;
              End ;
          End ;
        Inc(Osszesen,FoglOsszesen) ;
        WriteLn(AktFogl,' összesen: ',FoglOsszesen) ;
        WriteLn ;
        Inc(Sorszamlalo,2) ;
      End ;
    WriteLn('Dolgozó összesen: ',Osszesen) ;
    Write('Lista vége - ENTER') ;
    Close(Dolgozok) ;
    Repeat
    Until ReadKey = #13 ;
  End.
```

Kérdések

1. Mit jelentenek a következő fogalmak?
   a) Csoportváltás, kontrollváltás
   b) Összegfokozatos lista
   c) Egyszintű csoportváltás
   d) Többszintű csoportváltás
   e) Listafej, listatörzs, listaláb
   f) Csoportfej, csoporttörzs, csoportláb
   g) Tételsor

Feladatok

1*. Adott egy tömb, mely N darab számot tartalmaz növekvőleg rendezve. A sorozatnak lehetnek egyforma elemei is. Készítsünk egy listát, mely megadja, hogy melyik számból összesen hány darab szerepel a tömbben!

2*. Olvassunk be neveket „*" végjelig. Készítsünk egy listát, mely a neveket ABC sorrendben tartalmazza kezdőbetűk szerinti bontásban:

A
Albert Gáspár
Andor Aladár

B
Bálint Béla
Benedek Botond
Boldizsár Árpád
Budavári Róbert

C
...

3. Adott egy név és születési év adatpárosokból álló állomány, mely születési év s azon belül név szerint növekvően rendezett. Listázzuk ki az állományt úgy, hogy születési évenként névsorba írjuk az embereket. Minden születési év csak egyszer jelenjen meg a képernyőn, az első ember neve előtt. Írjuk ki csoportonként az emberek számát, végül az összes ember számát!

4. Hozzunk létre egy állományt, mely dolgozónként a következő adatokat tartalmazza: *főosztálykód, osztálykód, dolgozó neve, személyi szám.* Rendezzük az állományt főosztálykód, azon belül osztálykód, azon belül pedig nem és név szerint. (A dolgozó neme a személyi szám első karakteréből megállapítható.) Készítsünk kimutatást főosztályonként, s azon belül osztályonként a férfi és női dolgozókról, és azok átlag életkoráról!

5. Egy raktárból különböző árukat szállítanak a kerületi boltokba. A szállítások adatait a *Szallitas.Dat* állományban rögzítették a szállítások sorrendjében. Természetesen bármelyik bolt bármikor szállíthatott akármelyik áruból akármilyen összegben. A rekordkép a következő:

- Kerület Byte (1..22)
- Boltazonosító Word
- Tételazonosító String[10]
- Összeg Real

Készítsen kimutatást, mely megadja, hogy összesen, valamint kerületenként, azon belül boltonként és azon belül tételenként milyen összegben történt a raktárból szállítás. Az egyes tételeket tehát a listában boltonként összevonjuk. Tervezze meg a listát úgy, hogy az a felhasználónak tetszetős és áttekinthető legyen!

Érdemes tanulmányozni

Angster Erzsébet-Kertész László: Turbo Pascal feladatgyűjtemény I.:
Típusos állományok fejezet, Tipall12 program

7. SZÖVEGES ÁLLOMÁNYOK

A szöveges (text) állomány onnan kapta elnevezését, hogy tartalma egy olvasható szöveg. Szövegek nem csak lemezen lehetnek, hanem egyéb perifériális eszközökön is, mint például terminálon vagy nyomtatón. Bizonyos eszközökről csak olvashatóak, másokra csak írhatóak a szövegek. Ebben a fejezetben a szöveges állományok kezelését, és azok különböző eszközökön való megjelenéseit tárgyaljuk.

7.1 A szöveges állomány felépítése

A szöveges állomány sorokból áll, a sorok pedig karakterekből. Minden sor végén egy-egy „sor vége" jel (<EOLN> = End Of LiNe) található. Az állományt az „állomány vége" jel (<EOF> = End Of File) zárja:

E	L	S	Ő	<EOLN>	2	<EOLN>	U	T	O	L	S	Ó	<EOLN>	<EOF>

A fenti állomány három soros: az első sor négy betűből áll: „ELSŐ", a második sor egyetlen számból áll, míg a harmadik és egyben az utolsó sor az „UTOLSÓ" szöveget tartalmazza.

Az <EOLN> és az <EOF> logikai jelölések, fizikai megvalósításuk a következő:

- <EOLN>
  CR és LF karakterek: CR = #13 (Carriage Return = kocsi vissza); LF = #10 (Line Feed = soremelés).
- <EOF>
  Ctrl-Z = #26 karakter.

Fizikai megvalósítás:

Állománymutató

↓

E	L	S	Ő	#13	#10	2	#13	#10	U	T	O	L	S	Ó	#13	#10	#26

Az állomány szervezése soros, vagyis az egyes sorokat, illetve karaktereket csak egymás után lehet elérni, és az állományt nem lehet egyszerre írni és olvasni. Deklarációnál csak a szöveges állomány tényét kell megadni:

Var T : Text ;

A szöveges állomány mutatója egy konkrét karakteren áll, és azt csak a *Read*, illetve *Write* eljárások tudják állítani.

Nyitás, zárás

A szöveges állományt ugyanúgy hozzá kell rendelni egy fizikai állományhoz, mint bármely más állományt. Három megnyitási mód létezik:

- *Reset(T)*
  Nyitás után az állomány mutatója az első karakteren áll. Az állományt csak olvasni lehet sorosan, az első pozíciótól kezdődően.
- *Rewrite(T)*
  Nyitáskor az állomány törlődik. Mutatója az első karakteren áll. Az állományba csak írni lehet sorosan, az első pozíciótól kezdődően.
- *Append(T)*
  Szöveges állomány bővítése. Az állomány mutatója az <EOF> jelre áll. Innen kezdődően az állományt csak írni lehet.

Az állományt a *Close(T)* eljárással zárjuk le. Ha az állomány írásra volt megnyitva, akkor az eljárás még zárás előtt felír egy <EOF> jelet az állomány végére.

Szöveges állományban nincs értelme a pozicionálásnak, mert a sorok változó hosszúságúak. Így a rendszer nem tud következtetni az N. sor helyére a lemezen. Egy adott sort sem tudunk „kicserélni" egy másikra, hiszen az újonnan felírt sor más hosszúságú is lehet.

A szöveges állományban nem használhatók a *Seek*, a *FilePos*, illetve a *FileSize* függvények.

A szöveges állományba kiírhatunk különböző típusú adatokat úgy, mint képernyőre való íráskor. Ugyanúgy a beolvasás is különböző típusú változókba történhet. Nézzük meg először, hogyan tudjuk a szöveges állományt karakterenként kezelni.

7.2 Írás karakterenként

Write(T,Karakter)

Ha *Karakter* egy *Char* típusú kifejezés, akkor ezzel az eljárással egyetlen karaktert írhatunk ki a *T* szöveges állományba arra a helyre, ahol az állomány mutatója áll. Írás után a mutató eggyel tovább lép.

WriteLn(T)

A *WriteLn* (Write Line) eljárás hatására egy <EOLN> jelet írunk a *T* állományba. Az állomány mutatója a felírt CR és LF karakterek mögé, azaz az új sor elejére áll.

Feladat
Kérjünk be a billentyűzetről mondatokat. Egy mondatnak akkor van vége, ha leütötték az ENTER gombot. A beütött karaktereket egy szöveges állományba írjuk fel! ESC leütésére fejezzük be a bevitelt.

A mondatokat tartalmazó állományt *Rewrite*-tal nyitjuk meg, hiszen azt most szeretnénk létrehozni. A mondatokat alkotó karaktereket a *Readkey* függvénnyel olvassuk közvetlenül a billentyűzetről, és azonnal felvisszük az állományba. Csak a CR (ENTER) karaktert kell figyelni, mert annak bevitelekor egy teljes <EOLN> jelet kell felírni, vagyis a CR+LF karaktereket. A sor vége jelet a *WriteLn* eljárás írja fel. Az állomány lezárásakor az <EOF> jel automatikusan felíródik:

```
Program Mondatok ;
Uses Crt ;
Var
   Szoveg : Text ;
   Kar : Char ;
Begin
   Assign(Szoveg,'Mondatok.Txt') ;
   Rewrite(Szoveg) ;
   Kar := ReadKey ;
   While Kar <> #27 Do
      Begin
         If Kar = #13 Then
            WriteLn(Szoveg)
         Else
            Write(Szoveg,Kar) ;
         Kar := ReadKey ;
      End ;
   Close(Szoveg) ;
End.
```

Más megoldás

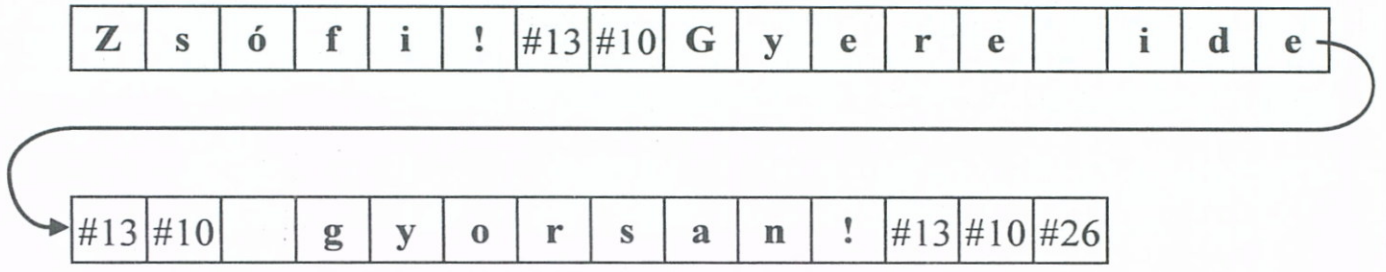
Írjuk fel az összes beütött karaktert, de ha a karakter CR, akkor írjunk utána egy LF karaktert is:

```
Kar := ReadKey ;
While Kar <> #27 Do
   Begin
      Write(Szoveg,Kar) ;
      If Kar = #13 Then
         Write(Szoveg,#10)
      Kar := ReadKey ;
   End ;
```

Nézzük meg, mi történik konkrét tesztadatok esetén. Legyenek a beütött karakterek:

```
Zsófi!<ENTER>
Gyere ide<ENTER>
 gyorsan!<ENTER><ESC>
```

Az állomány összesen 31 byte hosszúságú lesz, és így fog kinézni:



A szöveges állomány a lemezen csak egy byte-halmaz, úgy mint a többi – a katalógusban nincsen megkülönböztetve a többi állománytól. Miről tudjuk akkor a szöveges állományt felismerni? Hogy az állományt szövegként hozták-e létre, vagy sem, nem lehet megmondani – erre legfeljebb következtetni tudunk bizonyos jelekből, például:

- Az állomány kiterjesztése TXT, DOC, BAT, PAS stb.
- Sok benne az olvasható karakter. Erről meggyőződhetünk, ha a TYPE DOS paranccsal az állományt képernyőre írjuk, vagy egy bármilyen szövegszerkesztőbe behozzuk.

A szöveges állományt létrehozhatták akár Turbo Pascal programmal is *File Of Byte* típusú állományként – ez nincs az állományra „ráírva". A *Text* típus használata azonban rendkívüli módon megkönnyíti a szövegek feldolgozását. Természetesen a szabályokat be kell tartani. Vigyázni kell például arra, hogy a szöveg közé ne kerüljön fel egy <EOF> jel, hiszen ebben az esetben a soros olvasás ezen a ponton megszakad.

7.3 Olvasás karakterenként

Feladat
Jelenítsük meg az előző feladatban létrehozott szöveges állományt!

Ahhoz, hogy az előzőleg felírt mondatokat visszaolvassuk, újabb eljárásokat, illetve függvényeket kell megismernünk:

Read(T,Karakter)

Karakter egy Char típusú változó. Az eljárással egyetlen karaktert olvasunk be a *T* szöveges állományból arról a helyről, ahol az állomány mutatója áll. Olvasás után a mutató eggyel tovább lép.

ReadLn(T)

Az állomány mutatója a következő sor elejére áll. Bárhol is áll a mutató, átugrik minden karaktert a legközelebbi sorvégjelig, és a sorvégjel után megáll. Ha közben állomány végjelet talál, akkor azon áll meg. A *ReadLn(T,Karakter)* előbb beolvas egy karaktert a megadott változóba, és aztán áll a következő sor elejére.

EoLn(T) : Boolean

A függvény *True* értéket ad vissza, ha a sornak vége van (End Of LiNe), vagyis az állomány mutatója sorvégjelen áll. Az *If EoLn(T) Then ReadLn(T)* utasítással átugorhatjuk a sorvégeket.

Eof(T) : Boolean

Ezzel a függvénnyel kérdezhetjük le, hogy vége van-e az állománynak. *Eof(T)* értéke akkor *True*, ha a *T* szöveges állomány mutatója az <EOF> jelen áll.

Az állományból folyamatosan olvashatjuk a karaktereket és írhatjuk ki a képernyőre, hiszen a CR+LF karakterpáros a képernyőn is sort fog emelni. Mivel az állomány vége jelet már nem akarjuk feldolgozni, a ciklus addig tart, amíg az *Eof* függvény értéke *False*, vagyis van még beolvasandó karakter az állományban:

```
Program Olvas ;
Var
  Szoveg : Text ;
  Kar : Char ;
Begin
  Assign(Szoveg,'Mondatok.Txt') ;
  Reset(Szoveg) ;
  WriteLn('A szöveg: ') ;
  While Not Eof(Szoveg) Do
    Begin
      Read(Szoveg,Kar) ;
      Write(Kar) ;
    End ;
  Close(Szoveg) ;
End.
```

Az előző pontban felvitt tesztadatok esetén a képernyőn ez jelenik meg:

```
Zsófi!
Gyere ide
 gyorsan!
```

A kiírás után a kurzor az utoljára kiírt sor alatt, a sor elején áll.

Feladat
Írjuk ki az előzőleg létrehozott *Mondatok.Txt* állomány minden sorának első szavát! Az első szó az első szóközig tart. Ha a sor üres, azt is írjuk ki!

```
Program ElsoSzo ;
 Var
   Szoveg : Text ;
   Kar : Char ;
Begin
   Assign(Szoveg,'Mondatok.Txt') ;
   WriteLn('A szöveg első szavai: ') ;
   Reset(Szoveg) ;
   While Not Eof(Szoveg) Do
      Begin
         Read(Szoveg,Kar) ;
         If Kar = ' '
         Then
            Begin
               ReadLn(Szoveg) ;
               WriteLn ;
            End
         Else
            Write(Kar) ;
      End ;
   Close(Szoveg) ;
End.
```

7.4 Írás, olvasás soronként

Szöveges állományból a következő típusú adatokat lehet beolvasni, illetve oda kiírni:

- Karakter
- Karakterlánc
- Szám

Karakter kezeléséről az előző pontokban volt szó, a számokat a következő pontban tárgyaljuk. Ha *a Read* eljárás paramétere *String* típusú, akkor a *Read* mindenképpen a sor végéig olvas. A behozott sorból annyi karaktert tesz be a változóba, amennyi belefér, illetve amennyit talált.

Feladat
Írjuk képernyőre az *ElsoSzo.Pas* forrásprogramot soronként!

```
Program Sorok ;
Var
   F : Text ;
   Sor : String[80] ;
Begin
   Assign(F,'ElsoSzo.Pas') ;
   Reset(F) ;
   While Not Eof(F) Do
      Begin
         ReadLn(F,Sor) ;
         WriteLn(Sor) ;
      End ;
   Close(F) ;
End.
```

- Karakterlánc beolvasásánál feltétlenül a *ReadLn* eljárást használjuk, mert a *Read(F,Sor)* beolvassa a teljes sort, de sorvégjelen megáll. Ezért a következő *Read* eljárás üres karakterláncot fog beolvasni. A következő ciklus végtelen, mert az állomány mutatója egy helyben áll, és sohasem éri el az <EOF> jelet:

```
   While Not Eof(F) Do
      Begin
         Read(F,Sor) ; { Hibás! }
         WriteLn(Sor) ;
      End ;
```

Ha az előbbi programban *Sor* típusát *String[6]*-ra változtatjuk, akkor a képernyőn a program jobbról le lesz vágva (minden sorból csak 6 karakter jelenik meg).

A szöveges állományba egy egész sort egyszerre a *WriteLn(F,Sor)* eljárással írhatunk, ahol *Sor* egy *String* típusú változó vagy kifejezés.

Feladat
Írjuk át a *Mondatok.Txt* szöveg nem üres sorait a *Masolat.Txt* állományba!

```
Program Masol ;

Var
   Szoveg,
   Masolat : Text ;
   Sor : String ;
```

```
Begin
  Assign(Szoveg,'Mondatok.Txt') ;
  Reset(Szoveg) ;
  Assign(Masolat,'Masolat.Txt') ;
  Rewrite(Masolat) ;
  While Not Eof(Szoveg) Do
    Begin
      ReadLn(Szoveg,Sor) ;
      If Sor <> '' Then
        WriteLn(Masolat,Sor) ;
    End ;
  Close(Szoveg) ;
  Close(Masolat) ;
End.
```

7.5 Hozzáfűzés a szöveghez

Mint már említettük, egy szöveges állományt úgy tudunk bővíteni, hogy azt az *Append* eljárással nyitjuk meg. Az állomány mutatója ekkor egyszerűen rááll az <EOF> jelre, s az írás innen folytatódik.

Feladat
A *Mondatok.Txt* állományhoz fűzzünk hozzá egy utolsó sort. A sor tartalma az *„Ez az utolsó sor"* legyen! Ha az állomány nem létezik, akkor hozzuk azt létre!

```
Program Hozzafuz ;
Var
  Szoveg : Text ;
Begin
  Assign(Szoveg,'Mondatok.Txt') ;
  {$I-} Append(Szoveg) ; {$I+}
  If IOResult <> 0 Then
    Rewrite(Szoveg) ;
  WriteLn(Szoveg,'Ez az utolsó sor') ;
  Close(Szoveg) ;
End.
```

7.6 Számok írása, olvasása

A *Read*, illetve *Write* eljárásnak több paramétere is lehet ugyanúgy, mint képernyő esetén. Az eljárások szintaktikája is ugyanaz, csak szöveges állomány esetén első paraméterként meg kell adnunk a logikai állományváltozót.

Ha a *Read* eljárás paramétere szám, akkor a bevezető szóköz, TAB és sorvégjel karaktereket az eljárás átugorja, és beolvas egy karaktersorozatot a legközelebbi szóköz, TAB, sorvégjel, illetve állományvégjelig. Ezt a karaktersorozatot aztán a *Val* eljáráshoz hasonlóan megpróbálja átalakítani a paraméterként megadott numerikus változóba. Ha a beolvasott karaktersorozat nem numerikus formátumú, futási hiba lép fel. Numerikus változók kiírása hasonlóképpen történik, mint azt a képernyőnél tárgyaltuk. Eszerint használható például a *Write(T,Szám:N:M)* alak.

Összefoglalásként nézzünk néhány példát különböző típusú adatok kiírására és azok visszaolvasására:

```
Var
   F : Text ;
   W : Word ;
   R : Real ;
   C : Char ;
   S : String[10] ;
```

A következő programrészlettel felírunk egy sort a szöveges állományba:

```
C := 'K' ;
W := 600 ;
R := 4.9 ;
S := 'Sorvége' ;
WriteLn(F,W,R:4:1,' ',C,S) ;
```

6	0	0		4	.	9		K	S	o	r	v	é	g	e	#13	#10

Olvassuk most vissza ezeket az adatokat többféleképpen!

- `ReadLn(F,W,R,C,S) ;`
  → `W=600; R=4.9; C=' '; S='KSorvége';`
  R beolvasása után az állomány mutatója a szóközön áll meg, így *C* értéke ' ' lesz. Ha viszont a 4.9 után nem lenne szóköz, akkor *R* beolvasásánál a program futási hibával leállna.
  Az utasítás egyenértékű a következő 5 utasítással:
  `Read(F,W) ; Read(F,R) ; Read(F,C) ; Read(F,S) ;`
  `ReadLn(F,Sor) ;`

- `ReadLn(F,R,C,S) ;`
  → `R=600; C=' '; S='4.9 KSorvé';`
  R-be most a 600 kerül. A mutató a szóközön áll meg, ezért *C* tartalma a szóköz lesz. Mivel *S* karakterlánc, ezért az eljárás a 4-estől kezdve minden karaktert beolvas. *S*-be azonban csak 10 karakter fér be.

Írjuk fel most az adatokat másképpen:

```
C := 'K' ;
W := 600 ;
R := 4.9 ;
S := 'Sorvége'
WriteLn(F,W,R:3:1,C,S) ;
```

6	0	0	4	.	9	K	S	o	r	v	é	g	e	#13	#10

Mivel nem írtunk fel szóközöket az egyes adatok közé, visszaolvasáskor lehetetlen megállapítani, melyik adat meddig tart. A

```
ReadLn(F,W,R,C,S) ;
```

eljárás most futási hibával leáll, hiszen *W*-be a legközelebbi sorvégjelig próbálja a karaktersorozatot átalakítani. A futási hiba akkor is bekövetkezne, ha csak az első szóköz felírását hagytuk volna el, hiszen *W* nem valós változó.

7.7 Szabványos eszközök

Bizonyos periferiális eszközöket szöveges állományként lehet használni. Ilyen eszköz a *terminál, a nyomtató, a soros port* és a *nyelő*. Ezen eszközök mindegyikének van egy jól meghatározott fizikai neve:

- CON (Console): Írás esetén a képernyő, olvasás esetén a billentyűzet.
- LPT1, LPT2, LPT3 (Line PrinTer): Három nyomtató csatlakoztatható a számítógéphez, ezek bármelyikére lehet írni. A PRN (PRiNter) eszköznév egyenértékű az LPT1 névvel.
- COM1, COM2 (COMmunication port): Soros kommunikációhoz használatos csatornák. Ezeken az eszközökön keresztül lehet kapcsolatot tartani az egérrel, modemmel, faxszal, különböző műszerekkel stb. Az AUX eszköznév egyenértékű a COM1 névvel.
- NUL: Ez egy „kitalált" eszköz, amely elnyeli a karaktereket. Általában tesztelési célokra használatos.

A Turbo Pascal az egyes fizikai eszközökhöz logikai állományokat rendel, elősegítve ezzel azok kezelését. A következő logikai állománynevek a nyelv előre definiált változói:

- `Input : Text ;`
  Szabványos beviteli állomány. A rendszer alapértelmezésben a CON perifériához rendeli, és megnyitja olvasásra. A hozzárendelés a *System* egységben történik:
  ```
  Assign(Input,'CON') ;
  Reset(Input) ;
  ```

Az *Assign(Input,'')* alapértelmezés szerint szintén a CON-hoz rendeli *Input*-ot.
Az *Input* állomány használatakor az *Input* szó a *Read* utasításból elhagyható. Ezért olvasunk terminálról a *Read(Input,Változó)* helyett a *Read(Változó)* utasítással.

- `Output : Text ;`
  Szabványos kiviteli állomány. A rendszer alapértelmezésben a CON perifériához rendeli, és megnyitja írásra. A hozzárendelés a *System* egységben történik:
  ```
  Assign(Output,'CON') ;
  Rewrite(Output) ;
  ```
  Az *Assign(Output,'')* alapértelmezés szerint szintén a CON-hoz rendeli *Output*-ot.
  Az *Output* állomány használatakor az *Output* szó a *Write* utasításból elhagyható. Ezért írunk terminálra a *Write(Output,Kifejezés)* helyett a *Write(Kifejezés)* utasítással.
- `Lst : Text ;`
  Szabványos kiviteli állomány. A rendszer alapértelmezésben a PRN perifériához rendeli, és megnyitja írásra. A hozzárendelés a *Printer* egységben történik:
  ```
  Assign(Lst,'PRN') ;
  Rewrite(Lst) ;
  ```

Feladat
Készítsünk egy listát úgy, hogy az kérés szerint akár képernyőre, akár állományba kerüljön! A lista álljon most 20 darab 0 és 100 közötti véletlen valós számból!

```
Program Hova ;
Uses Crt ;

Var
  Lista : Text ;
  I : Word ;

Begin
  WriteLn('Képernyőre/Állományba?') ;
  If Upcase(ReadKey) = 'K' Then
    Assign(Lista,'CON')
  Else
    Assign(Lista,'Szamok.Txt') ;
  Rewrite(Lista) ;
  Randomize ;
```

```
    For I := 1 To 20 Do
      WriteLn(Lista,Random*100:10:4) ;
    Close(Lista) ;
End.
```

7.8 Nyomtató

Ha a nyomtatóra szeretnénk írni, akkor valamelyik LPT vagy a PRN perifériát meg kell nyitnunk írásra:

```
Var
  Nyomtato: Text ;
...
Assign(Nyomtato,'PRN') ;
Rewrite(Nyomtato) ;
WriteLn(Nyomtato,'Ez most a nyomtatóra ment!') ;
```

Ha a *Printer* egységet hozzászerkesztjük programunkhoz, akkor használhatjuk az előre definiált *Lst* logikai állományt, melyet a rendszer automatikusan megnyit számunkra:

```
WriteLn(Lst,'Ez is a nyomtatóra ment!') ;
```

A nyomtatóra csak írni lehet. A normál, megjeleníthető karaktereken kívül a nyomtatóra *vezérlőkarakter*eket is ki lehet küldeni. Az ilyen karaktereket az adott nyomtató megérti, és hatásukra a beletáplált módon viselkedik. Vannak összetett vezérlőkarakterek is. Ez esetben a hatást csak a karaktersorozat tagjainak egymás utáni kiküldése éri el. Nézzünk néhány példát lehetséges vezérlőkarakterekre, illetve sorozatokra:

- #10 (Line Feed): Soremelés
- #12 (Form Feed): Lapdobás
- #13 (Carriage Return): Kocsi vissza
- #27#51: Dőlt nyomtatás kezdése
- #27#52: Dőlt nyomtatás befejezése
- #27#45#49: Aláhúzott nyomtatás kezdése
- #27#45#48: Aláhúzott nyomtatás befejezése

E könyvnek nem célja egyetlen nyomtató működésének az ismertetése sem. Egy nyomtató alapos megismerése azonban a programozói munka elengedhetetlen feltétele, hiszen alig akad programrendszer nyomtatott listák nélkül.

A nyomtató használata előtt tanulmányozni kell annak leírását, mert azok egyedi módon működnek: egyik többet tud, másik kevesebbet, de az is előfordulhat, hogy egyik nyomtató nem kompatíbilis a másikkal, vagyis ugyanazon vezérlőkarakter kiküldése más hatást ér el egyiken, illetve másikon.

Feladat
Nyomtassuk ki a kért szöveges állományt úgy, hogy minden lapra fejlécet írunk. A fejléc tartalmazza aláhúzottan az oldalszámot és az állomány nevét!

```
Program Nyomtat ;

Uses
   Printer ;

Var
   T : Text ;
   Sor : String[80] ;
   AllNev : String[40] ;
   Lapsz, Sorsz: Byte ;

Procedure Lapfej ;
   Begin
      WriteLn(Lst,#27'-1',AllNev,', ',Lapsz,'. lap',
         #27'-0') ;
      WriteLn(Lst) ;
      Sorsz := 2 ;
   End ;

Begin
   Write('Az állomány neve: ') ;
   ReadLn(AllNev) ;
   Assign(T,AllNev) ;
   Reset(T) ;
   Lapsz := 1 ;
   Lapfej ;
   While Not Eof(T) Do
      Begin
         ReadLn(T,Sor) ;
         If Sorsz > 20 Then
            Begin
               Write(Lst,#12) ;
               Inc(Lapsz) ;
               Lapfej ;
            End ;
         WriteLn(Lst,Sor) ;
         Inc(Sorsz) ;
      End ;
   Close(T) ;
   Write(Lst,#12) ;
End.
```

7.9 Logikai eszközök átirányítása

A szabványos fizikai eszközöket (CON, PRN, ...) bármilyen logikai eszközzel (Input, Output, Lst) össze lehet kötni. Alapértelmezésben az *Input* logikai eszközt például a CON fizikai eszközhöz rendelték, de az bármikor átrendelhető egy másik fizikai eszközhöz vagy állományhoz. Az átirányítás sokszor nagyon hasznos lehet – nézzünk erre egy konkrét példát:

Feladat
Töltsük fel adatokkal a *Dolgozok.Dat* állományt, melynek rekordképe a következő:

- Törzsszám String[5] (számjegyek)
- Név String[20]
- Foglalkozás String[15]
- Szül. év Word
- Fizetés LongInt

A feltöltést később terminálról kell végezni, de most a program teszteléséhez az adatokat egy előre elkészített *Adatok.Txt* szöveges állományból vegyük!

Képzeljük csak el, hogy az adatok terminálról való begépelése közben valamilyen programhiba miatt mindig a 10. rekordnál történik valami hiba. Hogy ezt újra kipróbálhassuk, az előző kilenc rekordot megint csak be kell ütnünk. Amikor ezt a tesztelési műveletet már sokadszor ismételjük, akkor az *Input* átirányítása igazi megváltás lehet. Ekkor ugyanis a beviteli adatokat előre elkészítjük egy szöveges állományban, és a *Read* eljárás nem a terminálról, hanem a szöveges állományból veszi az adatokat.

A szabványos logikai eszközök átirányítása esetén dolgunk végeztével az eszközöket irányítsuk vissza, mert különben a szóban forgó eszköz ettől kezdve elérhetetlen. Esetünkben például a program végén nem lehet terminálról olvasni. Ez most azért nem probléma, mert a program úgyis befejeződik, és egy újabb programindításkor megint az alapértelmezések lépnek életbe.

```
Program Atiranyit ;

Type
   TDolgozo = Record
      Torzsszam : String[5] ;
      Nev : String[20] ;
      Fogl : String[15] ;
      Szulev : Word ;
      Fizetes : LongInt ;
   End ;
```

```
Var
   Dolgozok : File Of TDolgozo ;
   Dolgozo : TDolgozo ;
Begin
   Assign(Input,'Adatok.Txt') ;
   Reset(Input) ;
   Assign(Dolgozok,'Dolgozok.Dat') ;
   Rewrite(Dolgozok) ;
   ReadLn(Dolgozo.Torzsszam) ;
   While Dolgozo.Torzsszam <> '*' Do
      Begin
         ReadLn(Dolgozo.Nev) ;
         ReadLn(Dolgozo.Fogl) ;
         ReadLn(Dolgozo.Szulev) ;
         ReadLn(Dolgozo.Fizetes) ;
         ReadLn ;
         Write(Dolgozok,Dolgozo) ;
         ReadLn(Dolgozo.Torzsszam) ;
      End ;
   Close(Dolgozok) ;
   Close(Input) ;
End.
```

A programhoz elkészítjük az *Adatok.Txt* szöveges állományt. Az áttekinthetőség kedvéért a rekordok között üres sorokat hagyunk, ezért a programban minden rekord után ki kell adnunk egy üres *ReadLn* utasítást:

```
77244
Boldog Tihamér
Fizikus
1970
90000

12000
Keres Judit
Fizikus
1951
31000

*
```

Kérdések

1. Mit jelentenek a következő fogalmak?
   a) Szöveges állomány

b) Sor, sorvégjel, állományvégjel
c) Szabványos fizikai eszköz
d) Szabványos logikai eszköz
e) Nyomtató, vezérlőkarakter

2. Milyen a szöveges állomány szervezése?

3. Hányféleképpen lehet egy szöveges állományt megnyitni?

4. Milyen típusú adatokat lehet a szöveges állományba írni, illetve onnan visszaolvasni?

5. Hogyan lehet logikai eszközöket átirányítani?

Feladatok

1*. Kérjen be mondatokat úgy, hogy az a képernyőn is megjelenjen. Egy mondat bármilyen hosszú lehet, végét az ENTER leütése jelzi. Írja fel a mondatokat egy szöveges állományba úgy, hogy minden mondat végéhez hozzáfűzi a „*Stop*" szöveget! A felvitel akkor fejeződjék be, ha leütötték a Ctrl-Z billentyűt! Ha az állomány már létezett, akkor bővítse azt!

2*. Az előbb létrehozott állományból törölje ki a szóközzel kezdődő és az üres sorokat!

3. Írja ki az előbbi szöveges állományt képernyőre lassítva!

4. Írja ki az előbbi szöveges állományt nyomtatóra
   - NLQ (Near Letter Quality – közel levélminőségű) írásmóddal;
   - Condensed (sűrített) írásmóddal!

   Tegyen sorszámokat a sorok elejére, és a 80 karakternél hosszabb sorokat vágja le – a levágást valamilyen folytatásjellel jelezze! Egy lapra legfeljebb 55 sort írjon, és minden lap aljára írja ki a lapszámot, a szöveges állomány nevét és az aznapi dátumot aláhúzva!

5*. Hozzon létre a Turbo Pascal szövegszerkesztővel egy szöveges állományt, mely sorokból áll, és a sorok számokat tartalmaznak. A számokat legalább egy szóköz vagy TAB választja el. Írjon programot, mely az előbbi szöveges állományban található számokat soronként átlagolja, és kiírja az eredményeket a sor számával együtt!

Érdemes tanulmányozni

Angster Erzsébet-Kertész László: Turbo Pascal feladatgyűjtemény I.:
Szöveges állományok fejezet

8. MEMÓRIAKEZELÉS

Egy programozónak legalább nagy vonalakban tudnia kell, hogy programja és adatai hogy helyezkednek el a tárban, és futás közben milyen változások következnek be. A memória közelebbi ismeretével sokkal hatékonyabb programokat írhatunk, és a hibákat is könnyebben megtalálhatjuk. E fejezet feltérképezi a memóriát, bemutatja a tárkezelési lehetőségeket, majd tisztázza a dinamikus tárkezelés alapfogalmait.

8.1 A memória címzése

Az I. kötet „*Számítógép és a program*" fejezetében már szó volt a központi tár címzéséről. Emlékezhetünk, hogy a memóriát byte-onként címezzük, és a címzéshez regisztereket használunk. 2 byte (=16 bit) nagyságú regiszterrel összesen 2^{16} (=65536) byte, vagyis 64 KB címezhető meg. Több MB memória címzéséhez természetesen 1 regiszter nem elég. 32 bites címzési módot alkalmazva 2^{32} byte címezhető meg, ami 4096 MB. A DOS alatt futó programok azonban egész más címzési módot használnak. Mielőtt belemennénk ennek részleteibe, tisztázzunk néhány alapfogalmat:

- *Abszolút cím:* A memória valamely byte-jának (a memória elejétől számított) fizikai címe.
- *Relatív cím:* A memória valamely pontjától számított cím.
- *Paragrafus:* A memória egy 16 byte-os szelete. A paragrafus mindig egy 16-tal osztható fizikai címen kezdődik. Az *n.* paragrafus a *16*n.* byte-on kezdődik.
- *Szegmens:* Egy vagy több paragrafusból álló folytonos memóriaterület. A szegmens mindig paragrafushatáron kezdődik, és nagysága nem haladhatja meg a 64 KB-ot. Egy szegmens kezdőcímét kétféleképpen adhatjuk meg: paragrafusokban vagy abszolút módon. Mivel paragrafusokban általában egy szegmens kezdetét szokás megadni, ezért ezt a címet *szegmenscím*nek is nevezik.

A memória logikai felosztását a 8.1 ábra mutatja. A besatírozott rész egy 3 paragrafusnyi, 48 byte méretű szegmens, szegmenscíme 1.

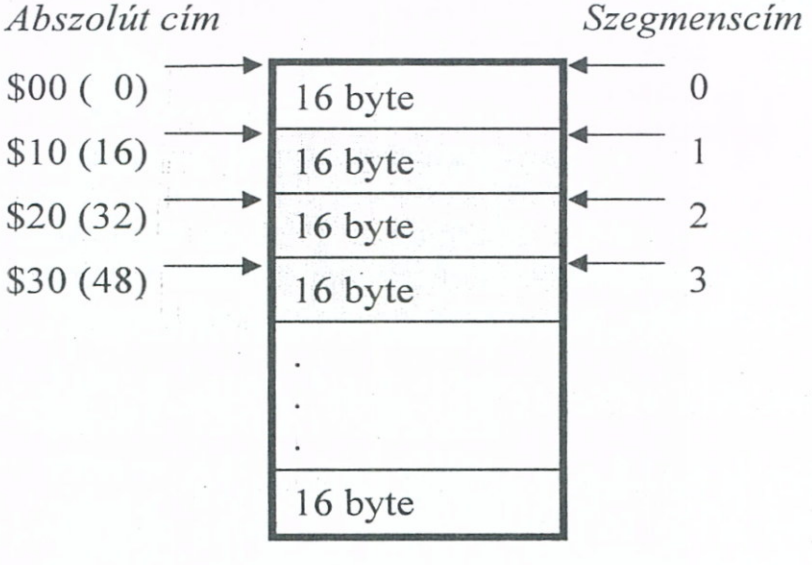
A DOS operációs rendszer alatt futó programok *kétkomponensű címzési módszer*t használnak. Egy fizikai címet egy *szegmenscím* és egy *ofszetcím* határoz meg:

Szegmens:Ofszet

- *Szegmenscím:* egy szegmens kezdőcíme paragrafusban.
- *Ofszetcím (eltolás):* a szegmensen belüli relatív cím.
- Mindkét cím egy-egy regiszter, azaz 4-4 hexadecimális jegy. Legkisebb értékük $0000, legnagyobb értékük pedig $FFFF lehet. A cím megadásakor a két érték közé kettőspontot teszünk. A címeket általában hexadecimálisan szokták megadni.
- Több *Szegmens:Ofszet* is meghatározhat egy memóriacímet.
  Ha a szegmenscímet megszorozzuk 16-tal, és hozzáadjuk az ofszetcímet, akkor megkapjuk, hogy a fizikai cím a memória hányadik byte-ja. A szorzást 2^4=16-tal a hardver végzi. Például: $0111:$0023 = $01133, mert

```
  $01110
+$00023
=$01133
```

- Egy cím *normált*, ha az eltolás érték 0 és 15 közé esik. Normált cím esetén a címzés egy-egyértelmű: egy címet csak egyféleképpen lehet megadni.



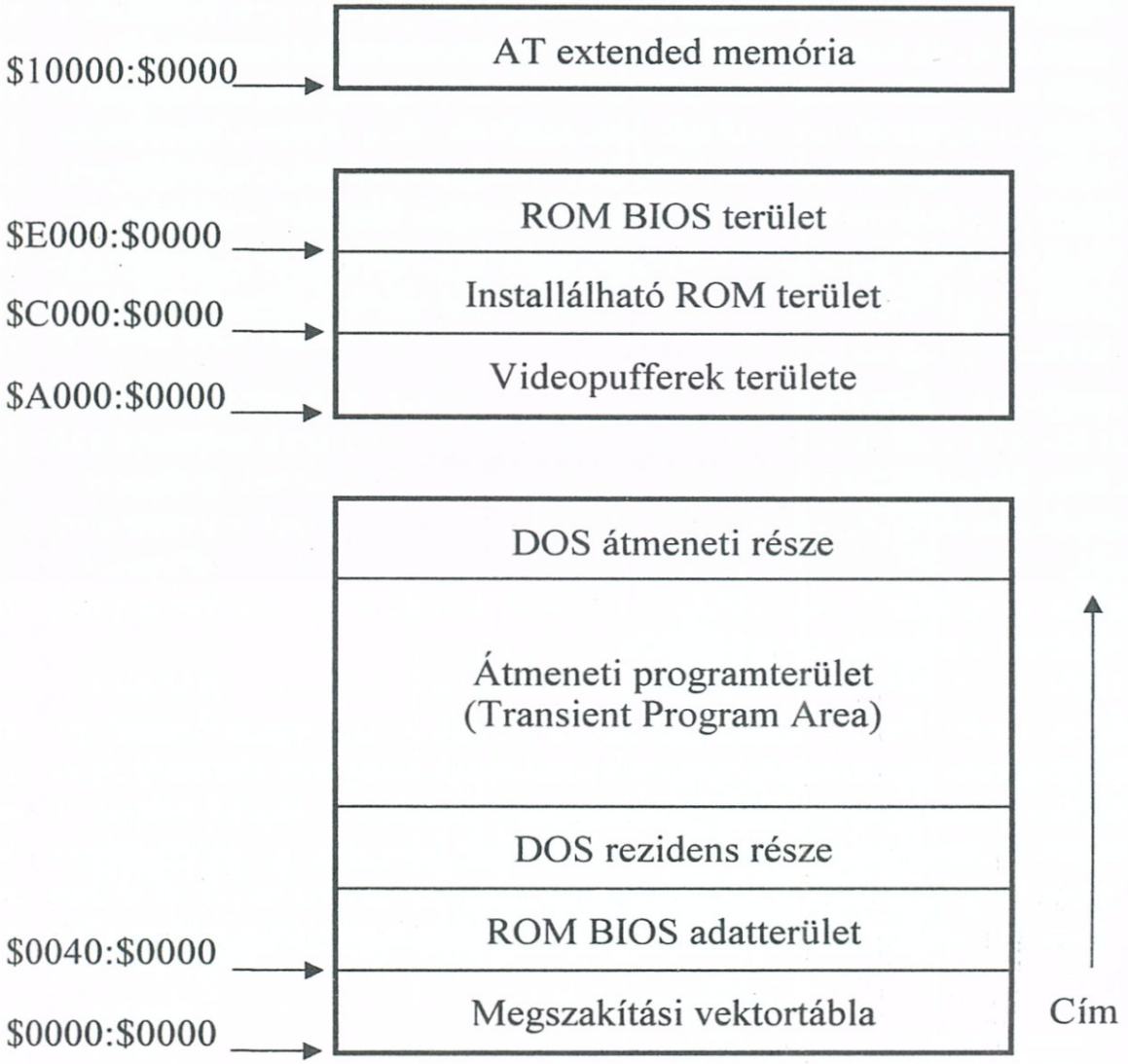
8.1. ábra. A memória logikai felosztása

Példák:

	Cím	*Byte sorszáma 0-tól*
1.	$0001:$0021	16+33=49
2.	$0002:$0011	32+17=49
3.	$0003:$0001	48+1=49
4.	$FFFF:$000F	65535*16+15= 1 048 575 (1 MB teteje)

Az első három cím ugyanazt a byte-ot határozza meg, mindegyik a memória 49. ($00031.) byte-ja.

8.2 A memória felosztása



8.2. ábra. Az IBM PC vázlatos memóriafelosztása DOS alatt

A számítógép memóriájának vázlatos felosztását a 8.2. ábra szemlélteti. A cím alulról megy felfelé, tehát a memória $0000:$0000 címén találhatók a megszakítási vektorok stb. Nézzük meg az egyes részeket kicsit közelebbről:

- *Megszakítási vektortábla:* A táblában hardver, illetve szoftver megszakítási rutinok címei vannak. A megszakítási rutinok olyan végrehajtható eljárások, melyeket főleg a rendszer használ anélkül, hogy a programozó tudna róla. Vannak rutinok, melyeket egyes hardver egységek egy esemény bekövetkezésére – programunkat megszakítva – hajtanak végre. Ha például leütünk egy billentyűt, akkor programunk megszakad, és a megfelelő hardver megszakítási rutin elhelyezi a billentyűnek megfelelő kódot a billentyűzet pufferben. A vektortábla rutinjait a programozó is felhasználhatja saját céljaira.

- *ROM BIOS adatterület:* Rendszerterület, amely a hardver vezérléséhez szükséges adatokat tartalmazza. Ezt a területet átírni csak a rendszer nagyon alapos ismerete esetén szabad!
- *DOS rezidens része:* Itt vannak a DOS alaprutinjai, melyek indításkor betöltődnek a tárba, és az operációs rendszer futása alatt ott is maradnak.
- *Átmeneti programterület:* Ez a terület a 640 KB legnagyobb része, ide kerülnek a felhasználói programok. Ide töltődik be a Turbo Pascal keretrendszer, és itt lesz az általunk írt, lefordított program is futtatható állapotban – a következő részben majd ezt a programot bontjuk tovább.

  Vigyázat! Az átmeneti programterület nem túl nagy, ezért ha memóriaproblémája van, nézze meg, nincs-e túl sok rezidens (a memóriában maradó) program a tárban!
- *Videopufferek:* Képernyőmemóriák. A rendszer a képernyőmemória tartalmát kivetíti a monitorra. Különöző típusú videokártyák esetén a videopufferek e területen belül különböző címeken helyezkedhetnek el.
- *Az installálható ROM terület*et és a *ROM BIOS terület*et szintén a rendszer foglalja.
- *Extended (bővíthető) memória:* Az 1 MB feletti rész, melyet a processzor csak védett üzemmódban tud elérni.

Mint említettük, a Turbo Pascal keretrendszer, s benne a szövegszerkesztő program (editor), fordító program (compiler), programszerkesztő program (linker) valamint a hibakereső program (debugger) az átmeneti programterületen foglal helyet. Ezen a területen keletkezik a keretrendszerben írt, lefordított program is. A következőkben azt vizsgáljuk meg, hogy az általunk írt program hogyan helyezkedik el a memóriában, és milyen további memóriaterületeket használ. E területek lényegében a következők:

- PSP
- Akárhány kódszegmens
- Adatszegmens
- Veremszegmens
- Átfedési (overlay) terület (nem kötelező)
- Heap (nem kötelező)

A 8.3. ábra a Turbo Pascal program által használt memória térképét ábrázolja. Nézzük meg közelebbről az egyes részeket:

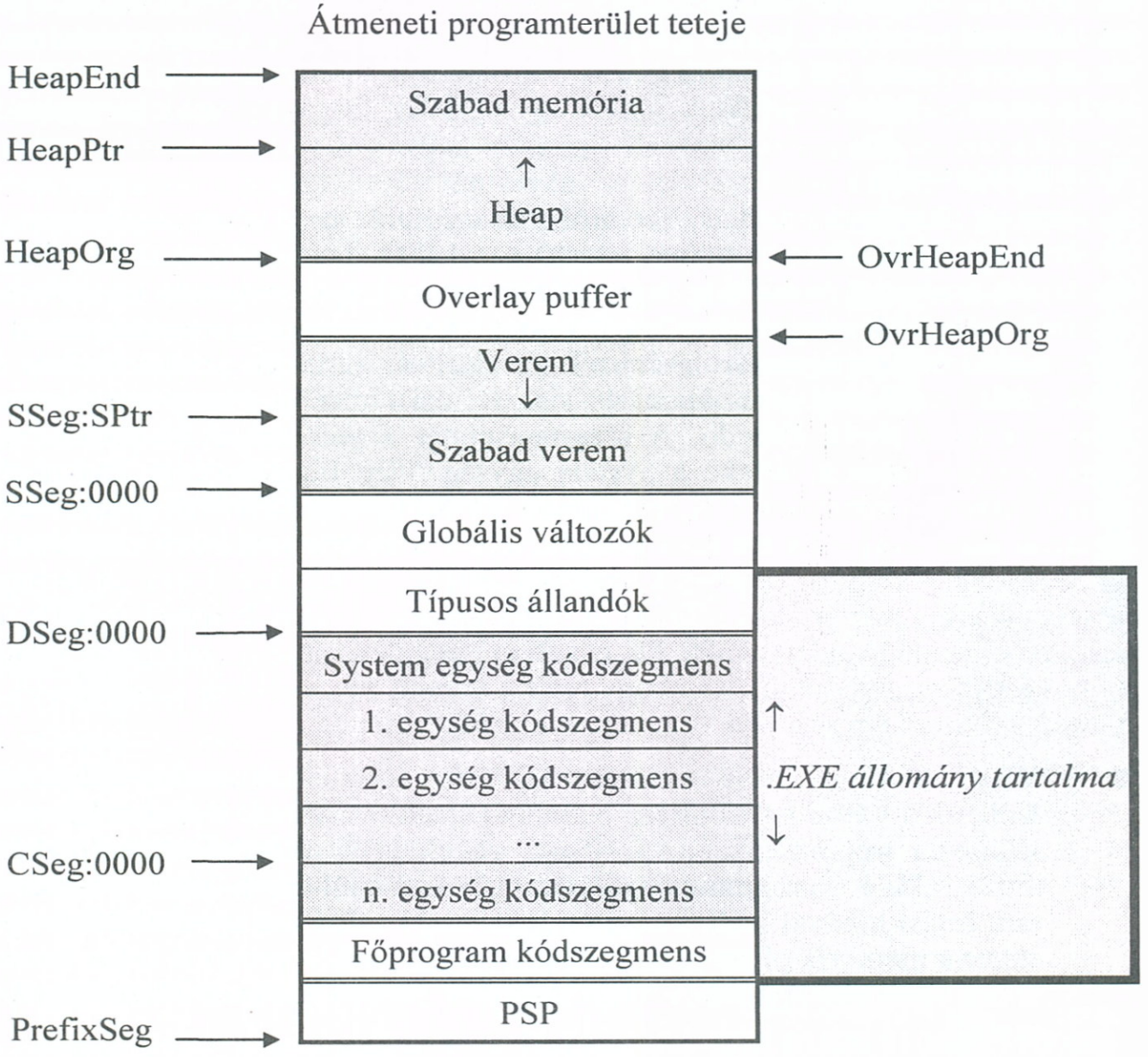
PSP

Program Segment Prefix. Ez egy 256 byte-os memóriaterület, melyet az operációs rendszer állít elő az .EXE állomány betöltésekor. A PSP szegmenscímét a *System* egység *PrefixSeg* változója tartalmazza.

Kódszegmens

A program jól elkülöníthető programrészekből áll. Egy Turbo Pascal programnak mindig van egy főprogramja, mely különböző modulokat (egységeket) használ. Ilyen modul például a *System*, *Crt* vagy a *Printer* egység (unit), melyek a gyártó cég által

szállított szabványos egységek. Írhatunk azonban mi programozók is egységeket, melyeket hozzászerkeszthetünk programjainkhoz. Egységek írásáról a *„Programszegmentálás, kapcsolat az operációs rendszerrel"* fejezetben lesz szó. Minden egység lefordított kódja egy külön szegmens, melyek egymás után helyezkednek el a tárban: legelöl a főprogram kódszegmense, melyet a programhoz szerkesztett egységek kódszegmensei követnek, mégpedig fordított sorrendben, mint ahogyan azokat a *Uses* kulcsszó után megadtuk. A *System* egység az utolsó kódszegmens, mely minden programhoz automatikusan hozzászerkesztődik. Egy kódszegmens maximális mérete 64 KB, a programban használt egységek száma elvileg korlátlan. A program futásakor a vezérlés mindig más kódszegmensben van, az aktuális kódszegmens szegmenscímét a processzor *CS* (Code Segment) regisztere tartalmazza – értéke a *System* egység *CSeg* függvényével lekérdezhető.



8.3. ábra. A Turbo Pascal program által használt memória

Adatszegmens

Az adatszegmens egy közös, állandó adatterület. A főprogram és az összes egység ezt az adatterületet használja statikus (az egész program futása során létező) adatainak tárolására. Az adatszegmens tartalmazza tehát a globális változókat és az összes típusos állandót (a lokálisan deklarált típusos konstansok is itt keletkeznek, hiszen azok az eljárás végrehajtása után sem veszíthetik el értékeiket). Mint minden szegmens, az adatszegmens mérete is csak maximum 64 KB lehet. Ebből következik, hogy összességében ennél nagyobb terület nem áll rendelkezésünkre globális adataink számára. Nem deklarálhatunk például két darab *Array[1..9000] Of LongInt* típusú változót, hiszen ezek mindegyike 36000 byte memóriát foglalna le. Tudni kell ezenkívül azt is, hogy 64 KB-nál nagyobb területet elfoglaló adatszerkezetet egyáltalán nem adhatunk meg programunkban, és hogy a gyári egységekben (System, Crt, ...) deklarált adatok szintén az adatszegmensből foglalják a helyet. Az adatszegmens méretét a deklarált adatok meghatározzák, így az már a fordítás során eldől. Az adatszegmens szegmenscímét a processzor *DS* (Data Segment) regisztere tartalmazza, mely a program futása során szintén állandó – értéke a *System* egység *DSeg* függvényével lekérdezhető.

Veremszegmens

A rendszer által karbantartott veremről már szóltunk az I. kötet *„Eljárások, függvények"* című fejezetében. A verem (stack) egy LIFO (Last In First Out) jellegű adatszerkezet, mely úgy működik, hogy adatot betenni csak a verem tetejére lehet, és onnan mindig csak a legutoljára betett adatot lehet kivenni. A veremszegmenst a rendszer az adatok ideiglenes tárolására használja: ide kerülnek az eljárások és függvények lokális változói, paraméterei és visszatérési címei. A verem szegmenscíme és mérete a futás során állandó. A szegmenscímet a processzor *SS* (Stack Segment) regisztere tartalmazza, értéke a *System* egység *SSeg* függvényével lekérdezhető. A verem méretét (minimum 1 K, maximum 64 K) fordítás előtt meg lehet adni ($M fordítási direktíva) – ha ezt nem tesszük, akkor a keretrendszerben megadott alapértelmezés lép érvénybe, amelynek szokásos értéke 16 KB körül van. A verem lefelé növekszik (feje tetejére van állítva). A verem-mutató (Stack Pointer) mutatja a következő elhelyezendő adat címét. Ez a verem szegmenscímétől számított relatív cím, mely kezdetben maximális, és ahogy telik a verem, úgy csökken az értéke. A veremmutató a *System* egység *SPtr* függvényével lekérdezhető.

Vigyázat! Ha rekurzív hívásokat végzünk vagy túl sok lokális változót használunk, akkor a verem könnyen betelhet (Stack overflow). Ilyenkor két eset lehetséges: ha programunkat az *S+* (Stack checking) fordítási direktíva mellett fordítottuk, akkor a programba egy ellenőrző kód épül be, és betelés esetén a program futási hibával leáll. *S-* esetén igaz, hogy valamivel gyorsabb a program, de ekkor a memória olyan részeit írhatjuk felül, mely a program teljes lemerevedéséhez vezethet. Ajánlatos tehát tesztelési időben a verem ellenőrzését bekapcsolni.

Átfedési puffer

Ez a terület csak akkor része programunknak, ha átfedési (overlay) technikát alkalmazunk – egyébként a verem után közvetlenül a heap következik. Az overlay technikát nagy programok esetén szokás használni, és lényege a következő: nem tartjuk bent egyszerre a tárban a teljes lefordított programot, hanem annak csak az éppen futó részeit – a futó programrészek átfedik egymást. A legkisebb átfedhető egység a Turbo Pascal egység (unit). A betöltéseket az *Overlay egység* automatikusan elvégzi a hivatkozott programelemek alapján. Az overlay technikát a „*Programszegmentálás...*" című fejezet fogja részletesebben tárgyalni. A puffer elejét a *System* egységben található *OvrHeapOrg*, végét az *OvrHeapEnd* függvénnyel kérdezhetjük le.

Heap

A heap egy dinamikus (állandóan változó) adattároló, jelentése halom, rakás. A program futása közben dinamikus változókat lehet itt létrehozni, illetve megszüntetni. A teljes heap bármekkora lehet (itt nincs az egy szegmensnyi korlát), de egy dinamikus változó mérete itt sem lehet több 64 KB-nál. A heap minimális, illetve maximális méretét a *$M* fordítási direktívával adhatjuk meg – ha ezt nem tesszük, akkor a keretrendszerben megadott alapértelmezés lép érvénybe, mely szerint a heap lefoglalja a teljes szabad memóriát. A heap veremszerűen működik, felfelé növekszik. A területfoglalásokat és felszabadításokat, valamint a foglalt és szabad területek felügyeletét a *heap-kezelő* (heap manager) végzi, mely a *System* egység része. A heap-mutató (heap pointer) alatti részben vannak a program által lefoglalt dinamikus változók, a felette lévő rész szabad. A heap alját a *System* egység *HeapOrg* változója, tetejét a *HeapEnd,* a heap-mutatót pedig a *HeapPtr* mutatja. A heap-pel a fejezet „*Dinamikus tárkezelés*" pontjában részletesen foglalkozunk.

A verem és heap méretének beállítása

$M *Veremméret, HeapMin, HeapMax*

Veremméret-tel a verem pontos méretét állítjuk be, értéke minimum 1024, maximum 65520. *HeapMin*-nel (0..655360) adhatjuk meg azt a minimális heap méretet, melyre a programnak mindenképpen szüksége van, és amely nélkül a program nem tud futni. *HeapMax* (HeapMin..655360) az a méret, amelynél több heap-et nem szeretnénk, ha a program lefoglalna, még akkor sem, ha van annyi hely (lehet, hogy egyéb célra is szeretnénk területet használni, például más programok futtatására). A *$M* direktívát a főprogram elején kell megadni, elhagyása esetén a keretrendszerben megadott értékek a mérvadóak. Hogy a heap konkrétan mekkora lesz, az futáskor derül majd ki. Ha a *HeapMin*-ben megadott terület futáskor nem áll rendelkezésre, akkor a program nem indul el. Nézzünk néhány példát:

A verem mérete 16000, a heap mérete 0 legyen:	{$M 16000,0,0}
A verem mérete 65520, a heap mérete 64 K legyen:	{$M 65520,$10000,$10000}
A verem mérete 1024, a heap mérete minimum 10 K, de akár az egész szabad memória legyen:	{$M 1024,10240,655360}

Feladat
Írjuk ki a szegmensek kezdőcímeit és méreteit!

```
Program CimekPrg ;
Uses Crt ;
Type
   Str4 = String[4] ;

Function HexToStr(W:Word):Str4 ;
   Const
      HexChars : Array[0..$F] Of Char
        = '0123456789ABCDEF' ;
   Begin
      HexToStr := HexChars[Hi(W) Shr 4] +
                  HexChars[Hi(W) And $F] +
                  HexChars[Lo(W) Shr 4] +
                  HexChars[Lo(W) And $F] ;
   End ;

Begin
   ClrScr ;
   WriteLn('Cimek:') ;
   WriteLn('Program kezdete (PSP címe) : ',
      HexToStr(PrefixSeg),'h') ;
   WriteLn('Kódszegmens címe   : ',HexToStr(CSeg),'h') ;
   WriteLn('Adatszegmens címe  : ',HexToStr(DSeg),'h') ;
   WriteLn('Veremszegmens címe : ',HexToStr(SSeg),'h') ;
   WriteLn ;
   WriteLn('Méretek:') ;
   WriteLn('Teljes kód mérete : ',
      (DSeg-CSeg)*16:6,' byte') ;
   WriteLn('Adatszegmens mérete : ',
      (SSeg-DSeg)*16:6,' byte') ;
   WriteLn('Verem teljes mérete : ',
      (OvrHeapOrg-SSeg)*16:6,' byte') ;
   WriteLn('Verem aktuális mérete: ',
      (OvrHeapOrg-SSeg)*16-Sptr:6,' byte') ;
End.
```

Seg és Ofs függvények

E függvények paramétere bármi lehet, aminek a memóriában címe van – változó, rutin stb. A visszaadott érték *Word* típusú: a kérdéses objektum szegmens, illetve ofszet címe. Például *Seg(HexChars)* vagy *Ofs(HexToStr)*.

8.3 Abszolút változó, rádefiniálás

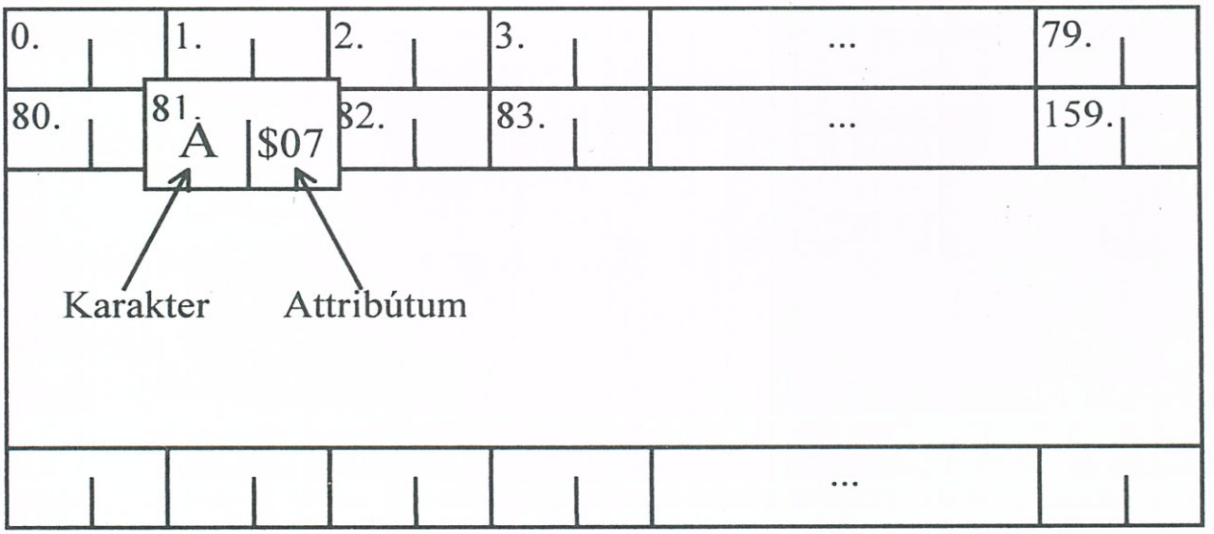
Egy globális változó deklarálásakor a fordító a változó számára „kioszt" az adatszegmensben egy memóriaterületet. Lokális változó esetén az eljárás, illetve függvény bevezető kódja intézi el a területfoglalást a veremszegmensben. A programozónak tehát semmi dolga a változó helyének, illetve címének meghatározásával, azt a rendszer automatikusan elvégzi. Egy magasszintű nyelvben csak a deklarációban megadott változó-azonosítóra kell hivatkozni, és a rendszer a foglalt memóriaterület címét behelyettesíti a program kódjába. Elképzelhető azonban olyan eset, amikor a programozó maga akarja meghatározni a változó fizikai címét – például a ROM BIOS adatterület egy adatát szeretné programjából megváltoztatni, vagy a képernyőmemóriát akarja direkt módon elérni. Ekkor a deklaráció végén meg kell adni a változó címét:

```
Var
  Változónév: Típus ABSOLUTE Szegmens:Ofszet ;
```

Abszolút változó megadásakor nem történik memóriafoglalás, a változóhoz tartozó terület kezelésének felelőssége teljes mértékben a programozóra hárul.

Abszolút változót csak indokolt esetben használjon, mert a memória bizonyos részeinek megváltoztatása esetén a rendszer összeomolhat!

Nézzük meg példaként, hogyan tudunk programunkból a képernyő memóriájába közvetlenül írni. A színes képernyő memóriája a $B800 szegmenscímen kezdődik. Az I. kötet *„Billentyűzet, képernyő"* fejezetében már szó volt a képernyő felépítéséről. Egy karakterhez a memóriában két byte-nyi információ tartozik: az első a karakter kódja, második pedig attribútuma. A karakterek tárolása sorfolytonosan történik – a képernyő memóriaképét a 8.4. ábra mutatja.



8.4. ábra. A képernyő memóriaképe

A színes képernyőmemória első karakterének abszolút deklarációja:

```
Var
  ElsoKar : Array[1..2] Of Byte Absolute $B800:0 ;
```

Feladat
Írjunk programot, mely a képernyőt teleírja szürke csillagokkal!

```
Program Csillag1;
Type
  ScrTip = Array[1..25,1..80,1..2] Of Byte ;
Var
  Scr: ScrTip Absolute $B800:0 ; {Mono esetén $B000:0}
  I, J : Byte ;

Begin
  For I := 1 To 25 Do
    For J := 1 To 80 Do
      Begin
        Scr[I,J,1] := Ord('*') ;
        Scr[I,J,2] := 7 ;
      End ;
End.
```

Más megoldás:

```
Program Csillag2 ;
Type
  Karhely = Record
    Kar : Char ;
    Attr : Byte ;
  End ;
Var
  Scr : Array[1..25,1..80] Of KarHely Absolute $B800:0 ;
  Sor, Oszlop : Byte ;

Begin
  For Sor := 1 To 25 Do
    For Oszlop := 1 To 80 Do
      Begin
        Scr[Sor,Oszlop].Kar := '*' ;
        Scr[Sor,Oszlop].Attr := 7 ;
      End ;
End.
```

Egy abszolút változó deklarálásakor a konkrét fizikai cím helyett írhatunk egy másik változónevet is:

```
Var
  Változó1 : Típus1 ;
  Változó2 : Típus2 ABSOLUTE Változó1 ;
```

Ekkor *Változó2* címe megegyezik *Változó1* címével. A rendszer *Változó1* számára helyet foglal, de *Változó2* számára nem – annak felügyelete teljes egészében a programozó dolga. *Változó2*-t rádefiniáltuk *Változó1*-re.

Vigyázni kell a rádefiniált változókkal, mert ha például *Típus2* több memóriát foglal, mint *Típus1*, akkor *Változó2* belelóg a *Változó1*-et követő változóba.

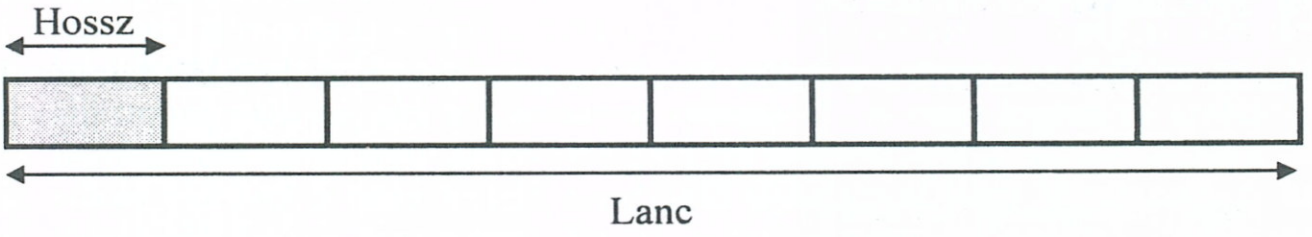
A rádefiniálást sokszor kényelmi szempontból szokás alkalmazni. Ugyanarra a változóra például több típust is rá tudunk húzni. Példának olvassunk be egy *Word* típusú értéket, majd írjuk ki annak felső, majd alsó byte-ját:

```
Var
  W : Word ;
  B : Array[1..2] Of Byte Absolute W ;
...
  ReadLn(W) ;
  WriteLn(B[2]:3, B[1]:3) ;
```

Most olvassunk be egy karakterláncot, majd írjuk ki annak hosszát:

```
Var
  Lanc : String[10] ;
  Hossz : Byte Absolute Lanc ;
...
ReadLn(Lanc) ;
WriteLn(Hossz) ;
```

A *WriteLn(Hossz)* utasítás itt egyenértékű a *WriteLn(Length(Lanc))*, illetve a *WriteLn(Ord(Lanc[0]))* utasításokkal.



Feladat
Írjunk programot, mely egy rekordba beolvas adatokat, majd megjeleníti a rekordot bájtonként! (Angster-Kertész: Feladatgyűjtemény I., Mem1.Pas)

```
Program Mem1 ;
Var
   Rek :  Record
      B : Byte ;
      W : Word ;
      S : String[10] ;
   End ;

   RekKep : Array [1..SizeOf(Rek)] Of Byte Absolute Rek ;
   I : Byte ;

Begin
   Write('B= ') ; ReadLn(Rek.B) ;      { 26 }
   Write('W= ') ; ReadLn(Rek.W) ;      { 260 }
   Write('S= ') ; ReadLn(Rek.S) ;      { Kakukk }

   For I := 1 To SizeOf(Rek) Do
      Write(RekKep[I]:5) ;
   WriteLn ;
End.
```

Rek: I:

B: W: S:

26	260	K	a	k	u	k	k	15

RekKep:

26	4	1	6	75	97	107	117	107	107	0	0	0	0	15

8.4 Memóriatömbök

A memória direkt elérésére a Turbo Pascal „egész memóriát lefedő" tömböket definiál:

```
MEM[Szegmens:Ofszet]
MEMW[Szegmens:Ofszet]
MEML[Szegmens:Ofszet]
```

A *MEM* tömb segítségével a memória egy adott címén található *byte*-ot írhatjuk, illetve olvashatjuk. *MEMW* esetén a hivatkozás *Word* típusú, *MEML* esetén pedig *LongInt*. A hivatkozások állhatnak az értékadó utasítás mindkét oldalán – baloldal esetén memóriaírás történik, egyébként olvasás.

Írjuk most tele a képernyőt csillagokkal memóriatömb segítségével:

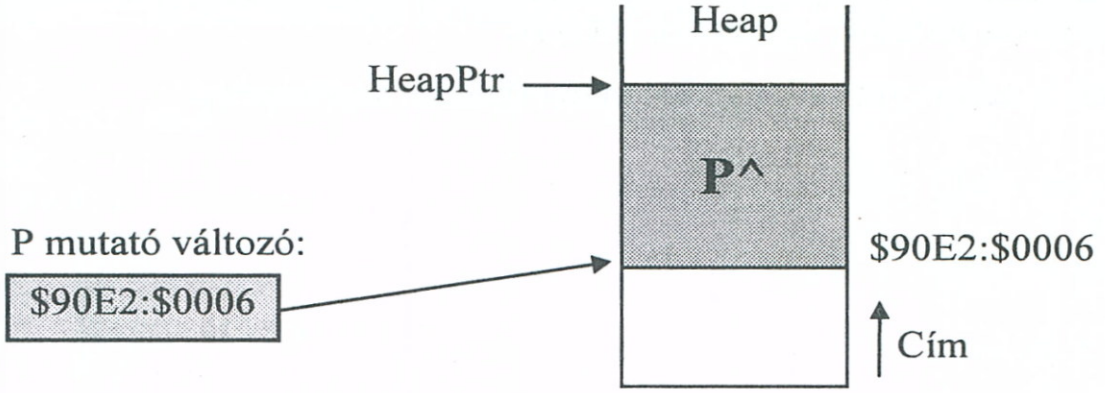
```
Program Csillag3 ;
Const
  Kep = 4000 ; {25*80*2}
Var
  I : Word ;

Begin
  I := 0 ;
  While I < Kep Do
    Begin
      MEM[$B800:I] := Ord('*') ;
      MEM[$B800:I+1] := 7 ;
      Inc(I,2) ;
    End ;
End.
```

8.5 Mutatók, dinamikus tárkezelés

Tegyük fel, hogy kapunk egy olyan feladatot, mely több funkciót kell, hogy ellásson, és mindegyik funkció végrehajtásához szükség van több, mintegy 60 KB méretű tömbre. A tömbök különböző típusúak, és azokra csak a funkció végrehajtása alatt van szükség. Az adatszegmensben még két ilyen tömb sem fér el, hiszen az összesen 64 KB nagyságú. Ekkora tömböt a veremszegmensbe sem ajánlatos betenni (lokális változóként deklarálni), hiszen a verem pillanatokon belül túlcsordulhat. Elképzelhető, hogy a heap-ben sincs az összes tömb számára hely. Jó lenne tehát, ha mindig csak azok a tömbök lennének a memóriában, amelyekre éppen szükségünk van. A dinamikus tárkezelés erre a problémára ad megoldást.

Egy dinamikus változót a program futása közben hozunk létre, illetve szüntetünk meg. A létrehozott változóra mutatóval hivatkozunk, mely mutató egy 4 byte-os memóriacímet tartalmaz: a mutatott változó fizikai címét. Programunk bármely részén deklarálhatunk mutató típusú változót, mely értékadás után egy címet fog tartalmazni. Ha *P* egy mutató típusú változó, akkor a mutatott változó *P^*. A heap foglalt és szabad területeit a *heap-kezelő* tartja nyilván és felügyeli. A heap-kezelő a *System* egység része. Új dinamikus változót a heap-kezelő megfelelő eljárásával lehet létrehozni. A heap-ben elhelyezkedő összes dinamikus változó a heap-mutató (*HeapPtr*) által mutatott cím alatt van, vagyis *HeapPtr* a heap első szabad byte-jára mutat. A HeapPtr *Pointer* típusú változót a System egység deklarálja. A heap-ben létrehozott változókat meg is

lehet szüntetni, ezért a heap-ben „lyukak" keletkezhetnek. Ha van megfelelő méretű lyuk, akkor az új változó abban kap helyet.



Kétféle mutató létezik: típusos és típus nélküli. Típusos mutató esetén a mutatott változónak jól meghatározott típusa van, míg egy típus nélküli mutató által mutatott memóriaterületre nem jellemző a típus.

```
Var
   P1 : ^Típus ;     { Típusos mutató }
   P2 : Pointer ;    { Típus nélküli mutató }
```

Mutatókkal nem csak dinamikus változókra lehet mutatni, hanem bármely memóriaterületre, például egy már létező statikus változóra. A mutatókat értékadással át lehet irányítani egyik memóriacímről a másikra. Mutató típusú változót sem beolvasni, sem pedig kiírni nem lehet.

Címek képezhetők a @ operátorral, az *Addr*, valamint a *Ptr* függvényekkel:

@X operátor
Addr(X): Pointer

Az eredmény mindkét esetben Pointer típusú: az X változó vagy rutin címe.

```
Var
   P : Pointer ;
   W : Word ;
...
   P := @W ;
   P := Addr(W) ;
```

Ptr(Szegmens,Ofszet): Pointer

A *Ptr* által visszaadott mutató érték a megadott *Szegmens*:*Ofszet* címre mutat.

```
Var
   Kep : Pointer ;
...
   Kep := Ptr($B800,0) ;
```

Mutató konstans

Egy konstans címet a *Ptr* függvénnyel állíthatunk elő: *Ptr(Seg,Ofs)*. A *Ptr($B800,0)* például a képernyőmemória kezdetére mutat. A *NIL* mutató konstans értéke *Ptr(0,0)*. *NIL* nem mutat semmilyen változóra sem, azt kizárólag arra szokás használni, hogy megállapítsuk: mutat-e valahová a mutató, vagy sem. Szokás azt is mondani, hogy a *NIL* nem mutat sehová.

Típusos mutató

Egy típusos mutató deklarálásakor megadjuk a mutatott változó típusát, például:

```
Type
   TTomb = Array[1..60000] Of Byte ;
Var
   W : ^String ;
   T : ^TTomb ;
```

W és *T* mutató-változók, mindkettő 4 byte helyet foglal el a memóriában. *W^* egy 256 byte méretű *String* típusú változó. *T* egy *Array[1..60000] Of Byte* típusú tömbre mutat, a mutatott tömb *T^*, mérete 60000 byte. *T^* változóval minden olyan művelet elvégezhető, ami egy statikusan deklarált hasonló típusú változóval:

```
Var
   T : ^TTomb ;
   Tomb : TTomb ;
...
   T^ := Tomb ;
   Tomb[2] := T^[2] ;
```

A mutatónak értéket kell adni, mielőtt hivatkozunk rá! Ellenkező esetben az valamilyen véletlenszerű memóriahelyre mutat, tehát a rá való hivatkozás futási hibát eredményezhet.

Egy típusos mutatónak kétféleképpen adhatunk értéket:

- Mutató értékadással
- Dinamikus változó létrehozásával (New eljárás)

Értékadás

Egy típusos mutatónak vagy egy típus nélküli mutatót, vagy egy ugyanolyan típusú típusos mutatót lehet értékül adni, például:

```
Var
   W1 : ^Word ;
   W2 : ^Word ;
   P : Pointer ;
...
   W1 := W2 ;
   W1 := P ;
```

Dinamikus változó létrehozása

Egy dinamikus változót a *New* eljárással hozhatunk létre:

New(P) ;

P egy típusos mutató-változó, típusa *^Típus*. Az eljárás lefoglal egy területet a heapből, a lefoglalt terület nagysága byte-okban: *SizeOf(Típus)*. *P* az új változóra fog mutatni.

- A *New* eljárásnak átadott aktuális paraméter típusos mutató kell, hogy legyen
- A lefoglalt terület nagysága a mutató típusától függ
- Hivatkozás az új változóra: P^
- Ha nincs elég hely a heap-ben, futási hiba lép fel
- A foglalást a heap-kezelő végzi
- Ha a heap tetején volt csak szabad hely (nem volt lyuk), akkor a *HeapPtr* feljebb mozog:
  P ← HeapPtr
  HeapPtr ← HeapPtr + P^ mérete

```
Var
   P : ^Integer ;
   I : Integer ;
...
   New(P) ;
   ReadLn(P^) ;
   I := Sqr(P^) ;
```

Feladat
Egy 64 KB nagyságú, byte-okból álló tömböt töltsünk fel véletlen értékekkel, majd számítsuk ki az elemek átlagát!
(Angster-Kertész: Feladatgyűjtemény I., Mem5.Pas)

64 KB nagyságú tömböt nem is tudnánk máshová elhelyezni, mint a heap-be. Hogy biztosan legyen ennyi heap futáskor, annak minimális méretét 64 KB-ra, azaz $FFFF-re állítjuk. (Turbo Pascalban egy adatstruktúra maximális nagysága pontosabban 64K-1 byte, azaz 65535 byte lehet):

```
Program Mem5 ;
{$M $4000,$FFFF,655360}
Type
   TombTip = Array[1..$FFFF] Of Byte ;
Var
   Tomb  : ^TombTip ;
   I     : Word ;
   Osszeg : LongInt ;
```

```
Begin
  New(Tomb) ;
  Randomize ;

  WriteLn('Várj, dolgozom!') ;
  For I := 1 To $FFFF Do
    Tomb^[I] := Random(256) ;

  Osszeg := 0 ;
  For I := 1 To $FFFF Do
    Inc(Osszeg,Tomb^[I]) ;
  WriteLn('A tömb átlaga: ',Osszeg/$FFFF:8:2) ;
  Dispose(Tomb) ;
End.
```

Dinamikus változó felszabadítása

Egy dinamikus változót a *Dispose* eljárással szüntethetünk meg.

Dispose(P) ;

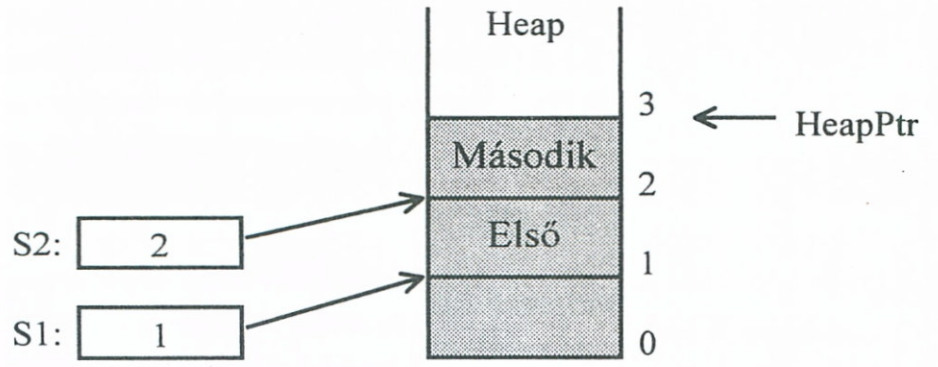
Az eljárás megszünteti a P által mutatott dinamikus változót. Ha P típusa *^Típus*, akkor a felszabadított terület nagysága *SizeOf(Típus)*.

- Az aktuális paraméter egy típusos mutató kell, hogy legyen
- A felszabadított terület nagysága a mutató típusától függ
- Ha a mutató nem a heap-be mutatott, akkor futási hiba áll elő
- A felszabadított terület elejére a heap-kezelő egy mutatót tesz, mely a következő szabad helyre mutat
- Legközelebbi heap-foglaláskor a heap-kezelő a most keletkezett lyukat felhasználhatja

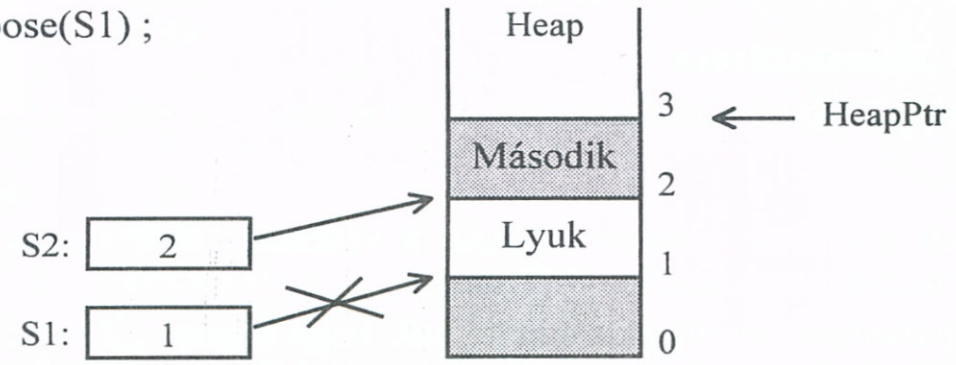
A következő példát egy ábrán próbáljuk szemléltetni:

```
Var
  S1, S2 : ^String[10] ;
...
  New(S1) ;
  New(S2) ;
...
  S1^ := 'Első' ;
  S2^ := 'Második' ;
...
  Dispose(S1) ;
  Dispose(S2) ;
...
```

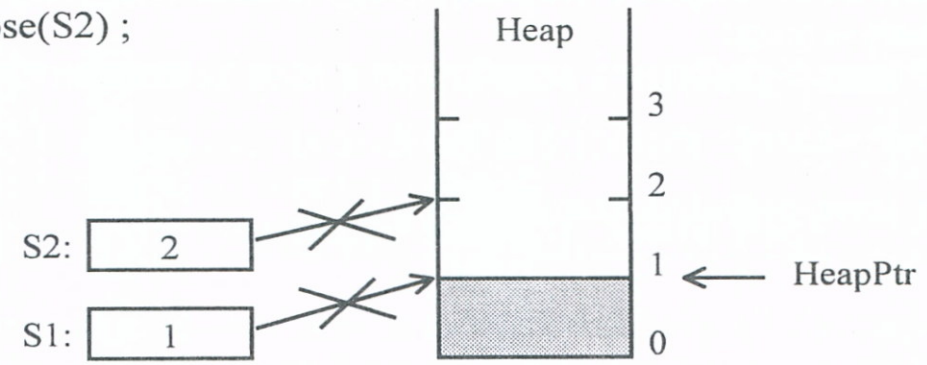
Területfoglalások, értékadások:



Dispose(S1) ;



Dispose(S2) ;



Heap felszabadítása egy adott címtől

Megtehetjük, hogy a heap tetejét felszabadítjuk egy adott pontig. Ehhez először meg kell jelölnünk a heap-nek azt a pontját, ameddig törölni akarjuk az összes addig létrehozott dinamikus változót.

Mark(P) ;

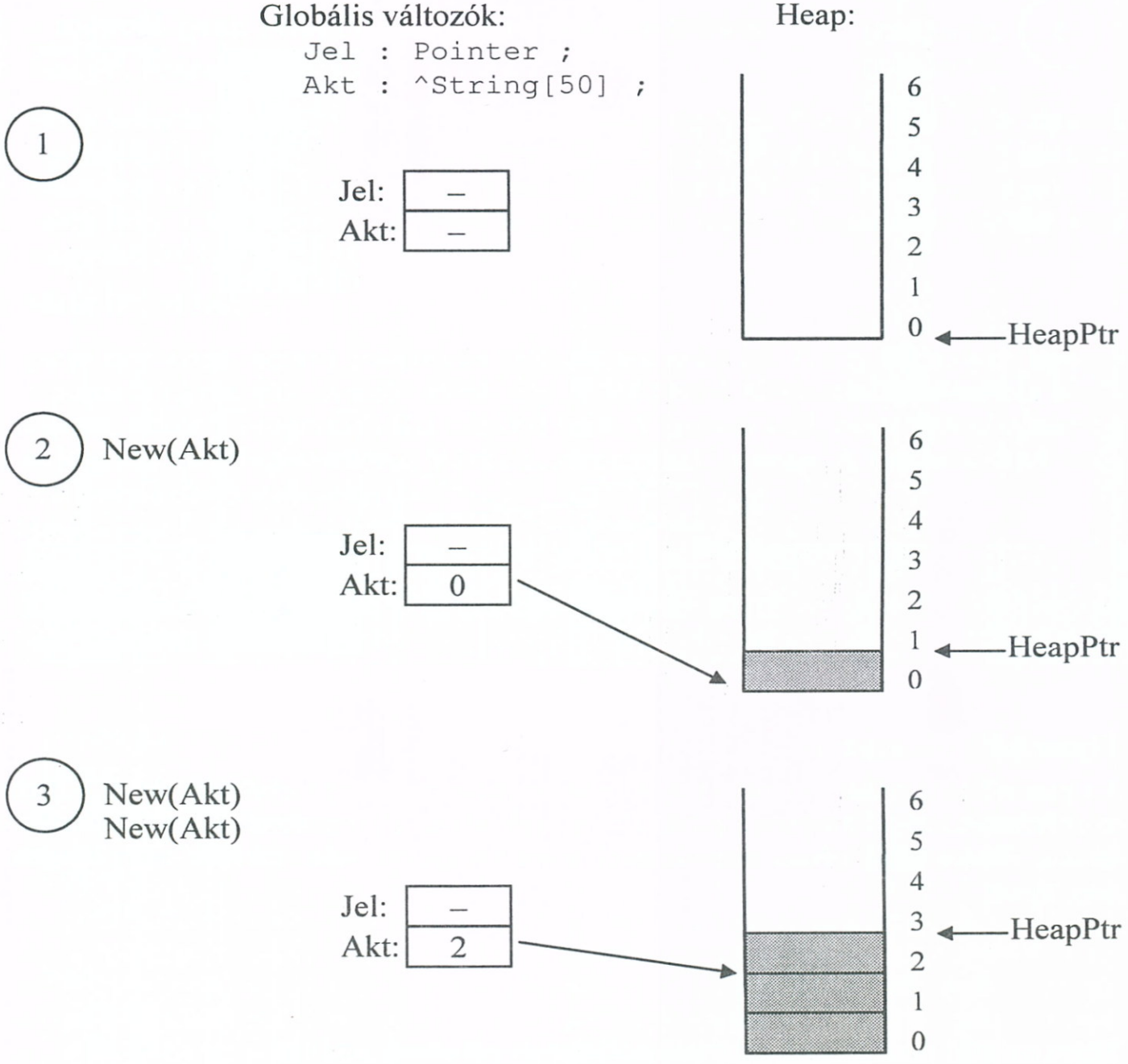
Az eljárás a *HeapPtr* értékét elmenti a *P* mutatóba.

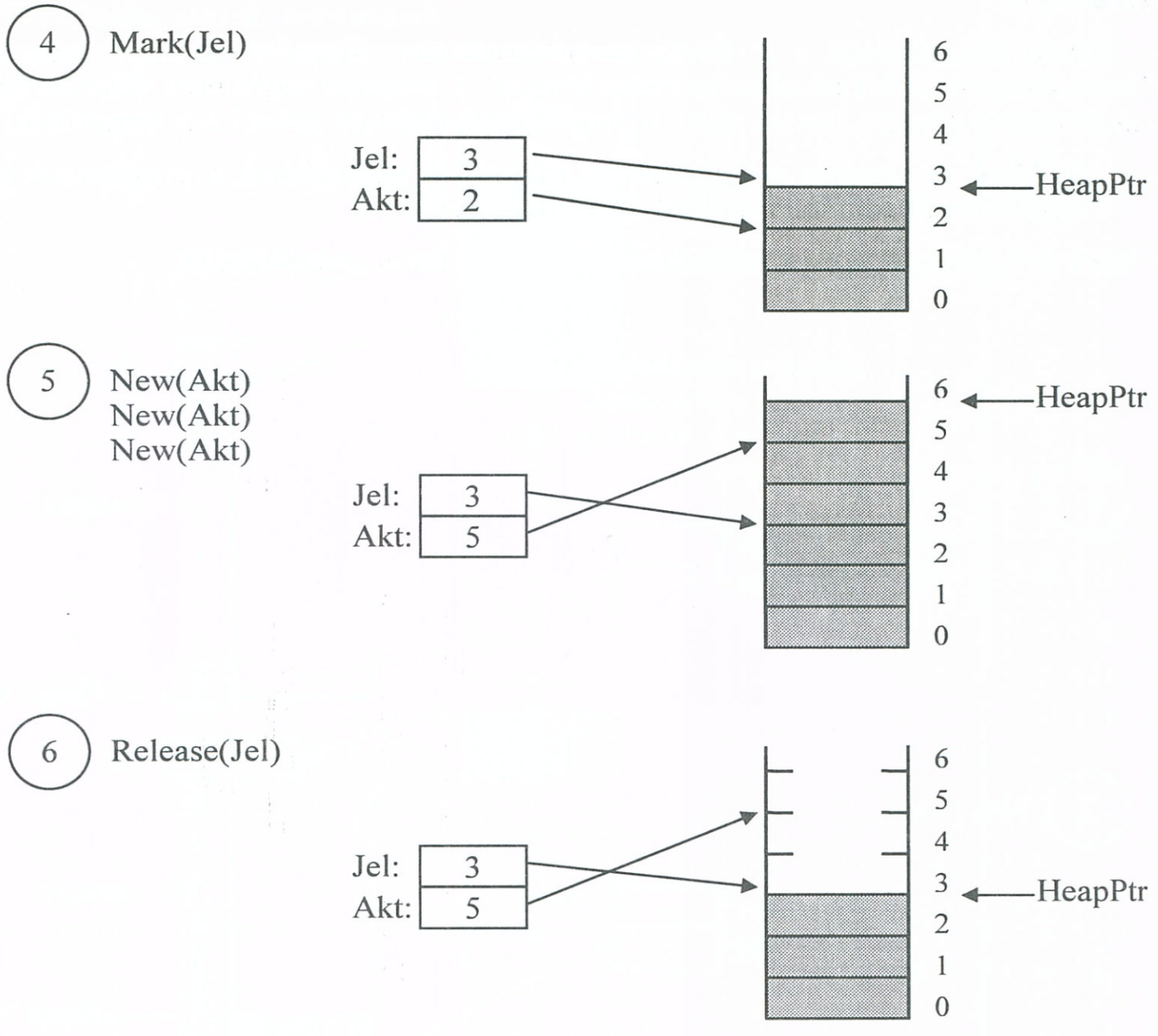
Release(P) ;

Az eljárás felszabadítja a P feletti memóriaterületet. *HeapPtr* értéke *P* lesz. A *Release(HeapOrg)* felszabadítja az egész heap-et. A program végén nem érdemes felszabadításokat végezni, hiszen ekkor a programmal együtt az egész heap megszűnik.

A *Mark* és *Release* eljárásokat ne használjuk ugyanabban a programban az egyéb felszabadító eljárásokkal (*Dispose*, *FreeMem*)!

A *Mark* és *Release* eljárások működését szintén egy ábrán szemléltetjük:





Típus nélküli mutató

A típus nélküli mutató szintén egy 4 byte-os memóriacímet tartalmaz, de itt a mutatott memóriaterületnek nincsen típusa. *P^*-ot csak olyan eljárásoknak lehet paraméterként átadni, illetve csak olyan műveleteket lehet elvégezni vele, melyek nem követelnek meg típust.

```
Var
   P : Pointer ;
```

Típus nélküli mutatóknak a következőképpen adhatunk értéket:

- @ operátorral, *Addr* illetve *Ptr* függvényekkel
- Mutató értékadással
- *GetMem* eljárással

Értékadás

A jobboldalon bármilyen típusos vagy típusnélküli mutató állhat.

```
Var
   P1, P2 : Pointer ;
   W : ^Word ;
...
   P2 := P1 ;
   P1 := W ;
```

Adott méretű heap lefoglalása, illetve felszabadítása

GetMem (P, Méret) ;

Az eljárás lefoglalja a megadott méretű helyet a heap-ben. *P* egy bármilyen típusos vagy típusnélküli mutató változó, *Méret* egy Word típusú kifejezés. *P* a most létrehozott dinamikus változóra fog mutatni. A mutatott változóra *P*^-pal hivatkozhatunk. Ha nincs elég hely a heap-ben, a program futási hibával leáll. A foglalandó terület nem lehet nagyobb 64 KB-nál.

FreeMem (P, Méret) ;

Az eljárás egy adott méretű dinamikus változót szabadít fel. *P* egy bármilyen típusos vagy típusnélküli mutató változó, *Méret* egy Word típusú kifejezés. Ha *P* nem a heap-be mutatott, akkor a program futási hibával leáll. Vigyázni kell, hogy *Méret* pontosan akkora legyen, mint foglaláskor, különben „elromolhat" a heap. A felszabadított területeket a heap-kezelő nyilvántartja.

Feladat
Gombnyomásra cseréljük ki a képernyő első két sorát az utolsó két sorával! ESC-re fejezzük be a játékot!

```
Program SorCsere ;
Uses
   Crt ;

Const
   Sor = 2 ;

Var
   Kep1,
   Kep2,
   Ment : Pointer ;
   Meret : Word ;
```

```
Begin
  {... }
  Meret := Sor*160 ;
  Kep1 := Ptr($B800,0) ;
  Kep2 := Ptr($B800,4000-Meret) ;
  GetMem(Ment,Meret) ;
  While ReadKey <> #27 Do
    Begin
      Move(Kep1^,Ment^,Meret) ;
      Move(Kep2^,Kep1^,Meret) ;
      Move(Ment^,Kep2^,Meret) ;
    End ;
  FreeMem(Ment,Meret) ;
End.
```

Műveletek mutatókkal

A mutatók között értelmezve vannak az = és a <> operátorok. Két mutató akkor és csakis akkor egyenlő, ha szegmens és ofszet címük megegyezik.

Vigyázat! A *$B001:$0000* és a *$B000:$0010* címek ugyanazt a memóriacímet határozzák meg, de definíció szerint nem egyenlőek.

A *New* és *GetMem* eljárások normalizált címet adnak (0 <= Ofs <= $F).

Kompatibilitás

Két mutató típus szerint kompatíbilis, ha

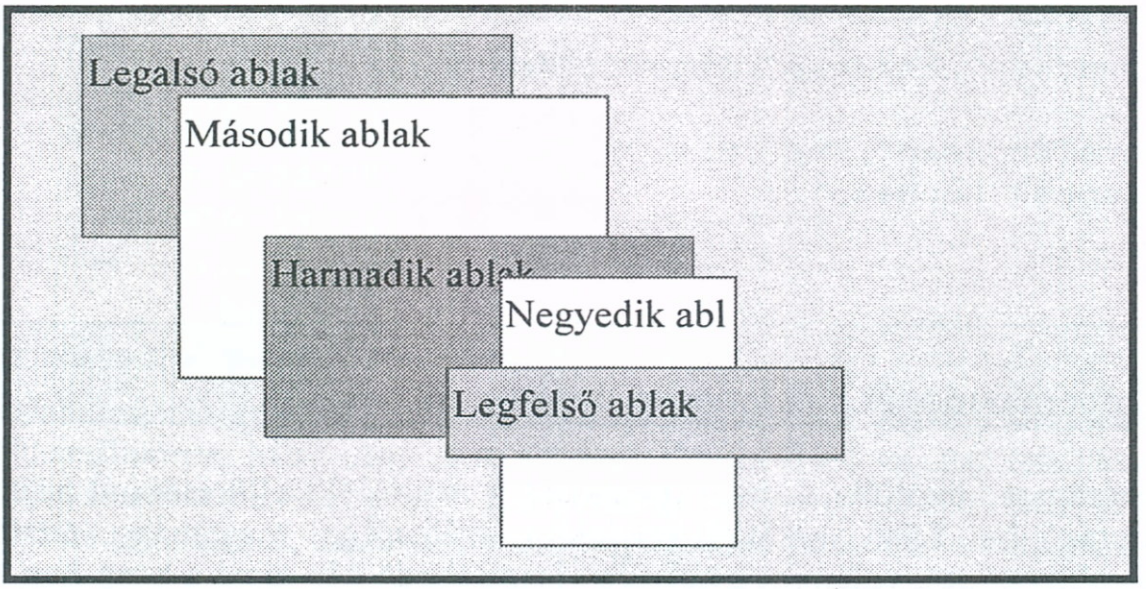
- a két típus azonos, vagy leírása ugyanaz;
- az egyik típusnélküli, a másik bármilyen típusú mutató

Két mutató akkor kompatíbilis értékadás szerint, ha azok típus szerint kompatíbilisek. A mutatott változók értékül adhatók egymásnak, ha azok értékadás szerint kompatíbilisek.

8.6 Ablaktechnika

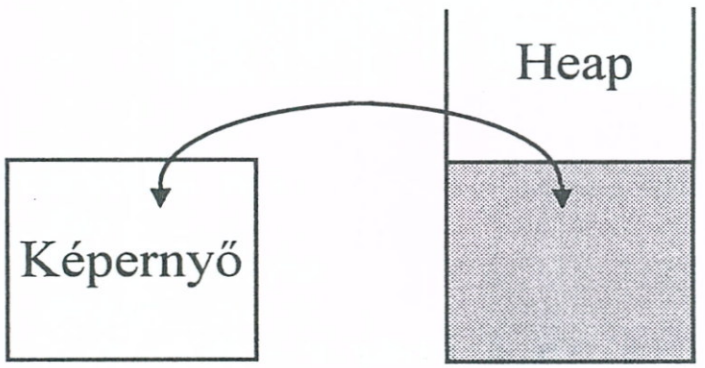
Ma már szinte elképzelhetetlen egy felhasználói program lebomló menük és ablakok nélkül. A legegyszerűbb üzenetet is úgy szokás kiírni, hogy a képernyőre „rátesszük" az információs ablakot, majd az üzenet nyugtázása után azt „levesszük" onnan. A leemelt ablak alatti kép természetesen az eredeti kell, hogy maradjon. Az ablak képernyőre tevése nem olyan egyszerű, mint egy papírlap elhelyezése az asztalon, hiszen a képernyő nem három dimenziós: ha ráírunk valamit, akkor az ott lévő információ elvész. Ezért még kiírás előtt az ablak helyét el kell menteni, majd alkalmasint vissza kell oda írni. Ha például egy adatbeviteli ablakot szeretnénk a képernyőre tenni, akkor az ablak alatti területet el kell mentenünk, majd ha végeztünk az adatbevitellel, akkor az elmentett területet vissza kell tennünk a képernyőre, mintegy „betömve" az ott keletkezett lyukat. A mentéseket dinamikusan kell végezni, hiszen az aktuálisan létrehozandó ablak a felhasználó által választott funkciótól függ, és az ablakok helye és

mérete különböző lehet. Az is elképzelhető, hogy az egyik ablakból hívható egy következő ablak, melyből megint hívható egy ablak stb. Mindig az aktív ablak van legfelül, és mindig csak a legfelső ablakot lehet „levenni". Az elmentendő területet mindig a heap-be tesszük, visszatételkor a helyét felszabadítjuk.



Mielőtt megírnánk a lebomló ablakok programját, nézzük meg, hogyan lehet az egész képernyőt a heap-be menteni:

Feladat
Mentsük el a teljes képernyőt a heap-be, majd később töltsük azt vissza!




```
Type
  KarHely = Record
    Kar: Char ;
    Attr: Byte ;
  End ;

  TKep = Array[1..25,1..80] Of KarHely ;
```

```
Var
   Kep, KepHeap: ^TKep ;

Begin
   Kep := Ptr($B800,0) ;
   ...
   New(KepHeap) ;
   Move(Kep^,KepHeap^,SizeOf(TKep)) ;
   ...
   Move(KepHeap^,Kep^,SizeOf(TKep)) ;
   Dispose(KepHeap) ;
End.
```

Feladat
Írjunk egy lebomló menüt megvalósító programot!

A teljes programot közöljük, a magyarázatokat a megfelelő eljárásoknál adjuk. Egy ablakra jellemző bal felső sarkának pozíciója, szélessége, magassága, attribútuma (háttér és karakterszíne), valamint a helyére visszateendő kép címe a heap-ben. Ezeket a tulajdonságokat fogja össze a TAblak típus.

```
Program MenuPr ;

Uses Crt ;

Const
   Esc = #27 ;
   Enter = #13 ;

Type
   KarHely = Record
      Kar : Char ;
      Attr : Byte ;
   End ;

   TKep = Array[1..25,1..80] Of KarHely ;

   TAblak = Record
      X,Y,Szel,Mag,Attr: Byte ;
      Cim: Pointer ;
   End ;

Var
   Kep : ^TKep ;
```

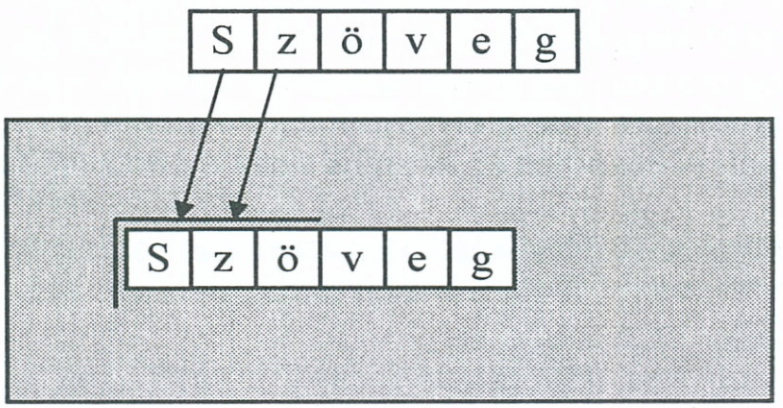
Cím növelése egy adott értékkel

A feladat megoldásához szükségünk lesz egy eljárásra, mely egy mutatót megnövel egy egész értékkel. A visszakapott mutató normált legyen – ez egyrészt az összehasonlításoknál fontos, másrészt ofszet túlcsordulása esetén hamis eredményt kapnánk:

```
Function Mutato(P :Pointer ; Rel :LongInt): Pointer;
  Var
    L : LongInt ;
  Begin
    L := LongInt(16) * Seg(P^) + Ofs(P^) + Rel ;
    Mutato := Ptr(L Shr 4,L And $0F) ;
  End ;
```

Szöveg írása a képernyőmemóriába (Write)

A következő rutin egy karakterláncot közvetlenül beír a képernyőmemóriába az (X,Y) relatív koordinátára. Mint ismeretes, az aktív ablak (Window) bal felső koordinátája a *WindMin* változóból kalkulálható. Közvetlenül a képernyőmemóriába való írás gyorsabb a *Write* utasításnál, hiszen a *Write* bonyolultabb is, és BIOS rutinokat hív.




```
Procedure MemIras(X,Y: Byte; St: String) ;
  Var
    L,I : Byte ;
  Begin
    Y := Y + WindMin Shr 8 ;
    X := X + WindMin And $FF ;
    L := Length(St) ;
    For I := 1 To L Do
      Begin
        Kep^[Y,X - 1 + I].Attr := TextAttr ;
        Kep^[Y,X - 1 + I].Kar := St[I] ;
      End ;
  End ;
```

Ablak törlése (ClrScr)

Ebben az eljárásban nem csinálunk mást, mint szóközökkel feltöltjük az egész képernyőt. Az összes karakterhely attribútuma a *TextAttr* változó értéke lesz.

```
Procedure AblakTorol ;
  Var
    X, Y: Byte ;
  Begin
    For Y := WindMin Shr 8 + 1 To
             WindMax Shr 8 + 1 Do
      For X := WindMin And $FF + 1 To
               WindMax And $FF + 1 Do
        Begin
          Kep^[Y,X].Kar := ' ' ;
          Kep^[Y,X].Attr := TextAttr ;
        End ;
  End ;
```

Ablak elmentése a heap-be

Az *Ablak* paraméterben megadott ablak alatti részt mentjük el a heap-be. Az összefüggő területet (Szélesség*Magasság*2 byte) az eljárás foglalja le, melyre az *Ablak* rekord *Cim* változója fog mutatni. Az átvitel soronként történik – az aktuális sor a kép elejétől számított *Mut* távolságra van, melyet minden alkalommal meg kell növelnünk az átvitt byte-ok számával. A heap-ben az aktuális sorra a *Mutato(Cim,Mut)* mutat.

```
Procedure AblakMent (Var Ablak: TAblak) ;
  Var
    I, Atvitel: Byte ;
    Mut : Word ;
  Begin
    With Ablak Do
      Begin
        Atvitel := Szel * 2 ;
        GetMem(Cim,Atvitel * Mag) ;
        Mut := 0 ;
        For I := Y To Y + Mag - 1 Do
          Begin
            Move(Kep^[I,X],
              Mutato(Cim,Mut)^,Atvitel) ;
            Inc(Mut,Atvitel) ;
          End ;
      End ;
  End ;
```

Elmentett ablak visszatétele a képernyőre

Az eljárás az ablak elmentésének pontosan a fordítottja. A paraméterben megadott *Cim* által mutatott helyről visszateszi a paraméterként megadott helyre a paraméterként megadott méretű ablakot. A kép helyét az eljárás végén felszabadítjuk:

```
Procedure AblakVissza (Var Ablak: TAblak) ;
  Var
    I, Atvitel: Byte ;
    Mut: Word ;
  Begin
    With Ablak Do
      Begin
        Atvitel := Szel * 2 ;
        Mut := 0 ;
        For I := Y To Y + Mag - 1 Do
          Begin
            Move(Mutato(Cim,Mut)^,
              Kep^[I,X],Atvitel) ;
            Inc(Mut,Atvitel) ;
          End ;
        FreeMem(Cim,Atvitel * Mag) ;
      End ;
  End ;
```

Ablak inicializálása

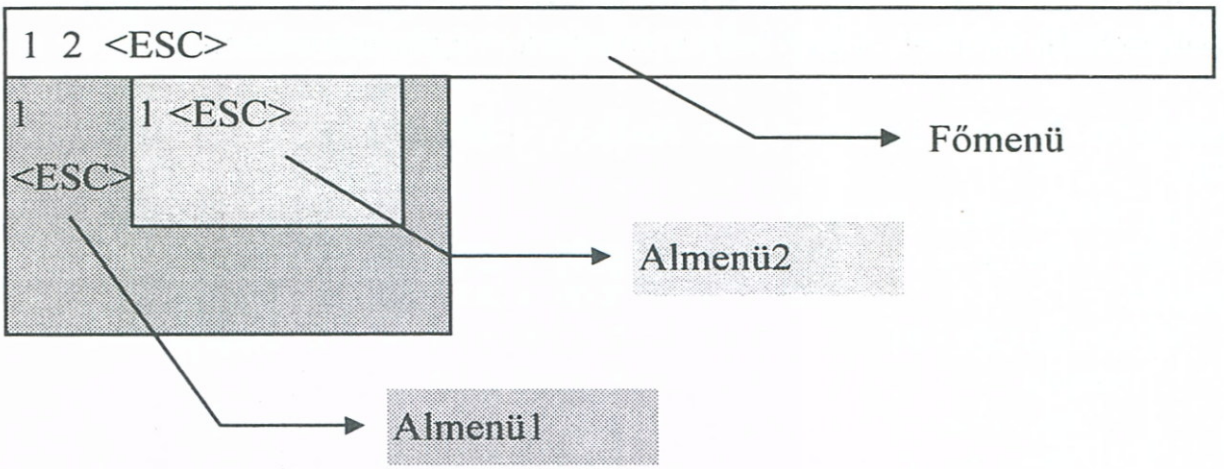
Az eljárás az *Ablak* paraméterben megadott pozícióban, mérettel és attribútummal egy üres ablakot hoz létre. Az ablak alatti területet előbb a heap-be menti:

```
Procedure AblakInit (Var Ablak: TAblak) ;
  Begin
    AblakMent(Ablak) ;
    With Ablak Do
      Begin
        Window(X,Y,X+Szel-1,Y+Mag-1) ;
        TextAttr := Attr ;
      End ;
    AblakTorol ;
  End ;
```

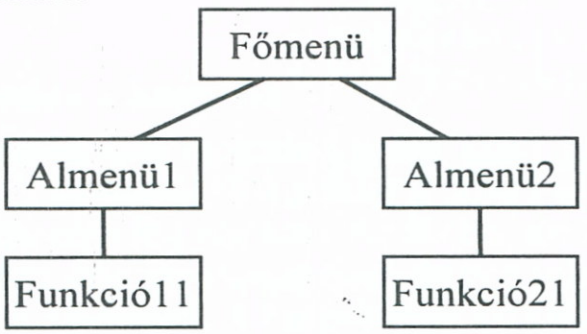
Almenük, funkciók

Most következnek a funkciók, almenük és a főmenü – mindegyiket egy-egy eljárásba helyeztük. Kihasználjuk, hogy az eljárások lokális változói sértetlenek maradnak újabb eljárások hívásai esetén is. Ezért minden ablakot abban az eljárásban deklarálunk, melyre az jellemző. Minden hívott eljárás a hozzátartozó ablak inicializálásával

kezdődik. Ezeket az ablakokat közvetlenül az eljárásból való kilépés előtt szüntetjük meg:



A menüstruktúra:




```
Procedure Funkcio21 ;
  Const
    Ablak : TAblak =
    (X:1; Y:2; Szel:50; Mag:20; Attr: $67; Cim:Nil) ;

  Var
    I : Byte ;

  Begin
    AblakInit(Ablak) ;
    For I := 1 To 100 Do
      Write('Funkció21 ') ;
    Write('<Enter>') ;
    Repeat
    Until ReadKey = Enter ;
    AblakTorol ;
    AblakVissza(Ablak) ;
  End ;
```

```
Procedure Funkcio11 ;
  Const
    Ablak : TAblak =
    (X:15; Y:8; Szel:25; Mag:3; Attr: $70; Cim:Nil) ;

  Begin
    AblakInit(Ablak) ;
    MemIras(1,1,'<Esc>') ;
    Repeat
    Until ReadKey = Esc ;
    AblakVissza(Ablak) ;
  End ;

Procedure Almenu1 ;
  Const
    Ablak : TAblak =
    (X:1; Y:2; Szel:30;Mag:15; Attr: $40; Cim:Nil) ;

  Var
    C : Char ;
  Begin
    AblakInit(Ablak) ;
    MemIras(1,1,'1') ;
    MemIras(1,2,'<Esc>') ;
    Repeat
         C := ReadKey ;
         Case C Of
           '1' : Funkcio11 ;
         End ;
    Until C = Esc ;
    AblakVissza(Ablak) ;
  End ;


Procedure Almenu2 ;
  Const
    Ablak : TAblak = (X:15; Y:2; Szel:15; Mag:8;
                      Attr: $17; Cim:Nil) ;
  Var
    C : Char ;

  Begin
    AblakInit(Ablak) ;
    MemIras(1,1,'1<Esc>') ;
```

```
      Repeat
           C := ReadKey ;
           Case C Of
             '1' : Funkcio21 ;
           End ;
      Until C = Esc ;
      AblakVissza(Ablak) ;
    End ;


Procedure Fomenu ;
  Const
    Ablak : TAblak = (X:1; Y:1; Szel:80; Mag:1;
                      Attr: $30; Cim:Nil) ;
  Var
    C : Char ;
  Begin
    AblakInit(Ablak) ;
    MemIras(1,1,'12 <Esc>') ;
    Repeat
         C := ReadKey ;
         Case C Of
           '1' : Almenu1 ;
           '2' : Almenu2 ;
         End ;
    Until C = Esc ;
    AblakVissza(Ablak) ;
  End ;

Begin
  Window(1,1,80,25) ; { WindMin és WindMax beállítása }
  Kep := Ptr($B000 + Ord(LastMode <> Mono) * $800,0) ;
  Fomenu ;
End.
```

Kérdések

1. Mit jelentenek a következő fogalmak?
   a) Kétkomponensű címzés
   b) Paragrafus, szegmens
   c) Abszolút cím, relatív cím
   d) Kódszegmens, adatszegmens, veremszegmens, overlay terület, heap
   e) Abszolút változó, rádefiniálás
   f) Memóriatömb

g) Mutató, típusos mutató, típusnélküli mutató
h) Dinamikus változó
i) Heap-kezelő
j) Ablaktechnika
k) Lebomló menü

2. A programnak mely szegmenseit határozhatjuk meg közvetlen módon, és hogyan? Hogyan adható meg a heap mérete?
3. Növelhető-e a veremszegmens nagysága futás közben?
4. Hol helyezkedik el a képernyőmemória és milyen felépítése?
5. Hogyan tudunk közvetlenül a képernyőmemóriába írni?
6. Hogyan lehet egy dinamikus változót létrehozni/megszüntetni?
7. Mekkora helyet foglal el egy dinamikus változó?
8. Mekkora helyet foglal el egy típusos mutató?
9. Hogyan lehet a heap-et egy adott címig felszabadítani?

Feladatok

1*. Jelenítse meg a főprogramban deklarált változók tartalmát byte-onként először decimálisan, majd karakteresen (a nem megjeleníthető karakterek helyén pont szerepeljen). A tár egy részének ilyetén kiírását *dump*nak nevezik (dump= lerak, lezúdít)!

2*. Írjon eljárást, mely a képernyő egy adott pozíciójába tesz egy karaktert közvetlenül a képernyőmemóriába való írással, a Write eljárás használata nélkül! A pozíciót, a karaktert és annak színét az eljárás paraméterként kapja meg!

3. Írjon programot, mely a képernyő egy megadott ablakát teleírja a megadott karakterekkel közvetlenül a képernyőmemóriába való írással!

4. Írjon programot, mely a képernyő egy adott ablakának tartalmát kinyomtatja!

5*. Írjon eljárást, mely a képernyőt megfordítja: a felső sor legyen az alsó, az alsó a felső stb. A középső sor marad a helyén!

6*. Írjon egy eljárást, mely
a) az aktuális képernyőt a megadott nevű állományba menti;
b) a megadott nevű képet betölti a lemezről!

7. Hozzon létre (illetve bővítsen) egy típusos állományt. Az állomány első rekordja tartalmazza a felhasználó cég nevét és az állomány utolsó használatának dátumát. Ezt a dátumot rögtön az állomány nyitásakor írja felül a mai dátummal. A program végén listázza ki az állományt úgy, hogy a fent említett adatok, valamint a mai dátum a lista tetején szerepeljen!

8. Írja ki a program adat-, verem- és kódszegmensének méretét. A használt verem nagyságát írja ki a program különböző pontjain, főprogramban, eljárásban. A program paramétereit és lokális változóit közben cserélgesse!

9. Írjon programot, mely megjeleníti a memória egy részét egy adott címtől kezdve 20 paragrafus hosszúságban! Egy paragrafus a képernyőn egy sor legyen. A memóriabájtokat először hexadecimálisan, majd alatta karakteresen írja ki (a nem megjeleníthető karakterek helyén pont szerepeljen)!

10. Készítsen 4 darab tömböt, melyek mindegyike 10000 darab 4 bájtos véletlen számból áll, és írja ki az átlagukat! A tömbök feltöltésére eljárást, átlagolásukra pedig függvényt készítsen! Oldja meg ugyanezt a feladatot 10 tömbbel!

11*. Vegyen fel egy tömböt és írja tele értékekkel! Mentse el a tömböt a heap-be, majd miután az értékeket megváltoztatta, töltse vissza az eredeti változatot!

12. Mentse el a képernyő aktuális tartalmát a heap-be, majd képernyőtörlés után töltse azt vissza!

13. Készítsen különböző képernyőket, és mentse el azokat a heap-re. Ezután menüből választhatóan töltse vissza azokat!

14. Mentsen el egy megadott méretű A ablakot a heap-be. Törölje le A-t pirossal, majd egy billentyű leütésére hozza vissza A helyére az eredeti ablakot!

Érdemes tanulmányozni

Angster Erzsébet-Kertész László: Turbo Pascal feladatgyűjtemény I.:
Memóriakezelés fejezet

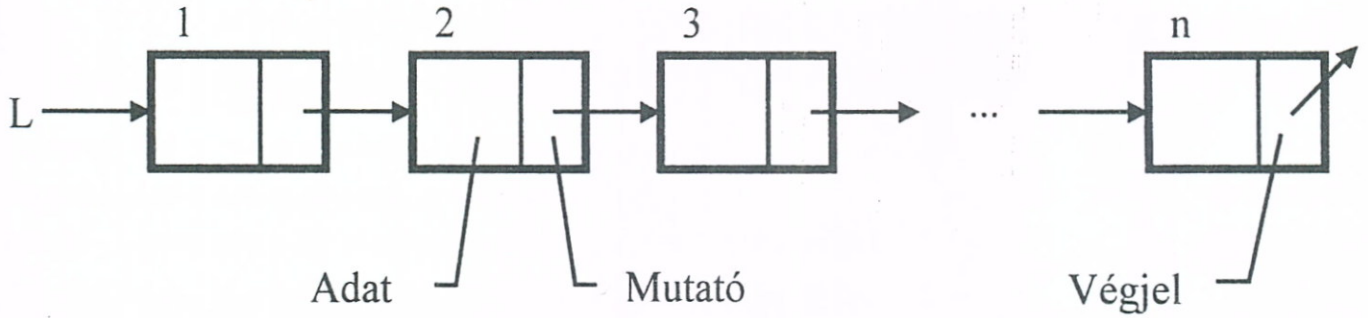
9. DINAMIKUS LISTA

Ebben a fejezetben a listával, mint absztrakt tárolóval és annak a heap-ben való megvalósításával foglalkozunk.

9.1 Lista

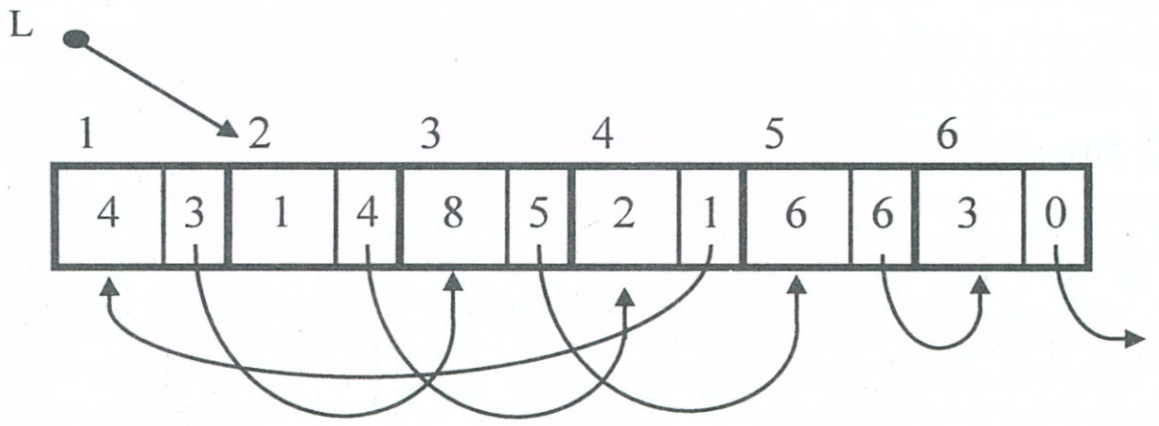
A listára az jellemző, hogy a benne lévő adatelemek össze vannak kapcsolva: minden listaelem valamilyen módon tartalmazza az őt követő elem mutatóját. A lista a mutatók révén elejétől a végéig bejárható. A listára új elemek kapcsolhatók rá, a már meglévők pedig lekapcsolhatók. A listát az első listaelem mutatója (L) meghatározza. Az utolsó elem mutatója általában egy végjel, mely egyértelműen felismerhető, és természetesen nem létező listaelemre mutat.

A listát *kapcsolt* vagy *láncolt listá*nak, vagy egyszerűen *lánc*nak is szokás nevezni. A legegyszerűbb lista az *egyirányú* vagy *lineáris lista*, ahol a mutatók csak egy irányban léteznek:



Egy listaelem egy *adat* és egy *mutató* részből áll. A lista elemei fizikailag bárhol elhelyezkedhetnek, erre semmiféle megkötés nincsen. Ábrákon a logikailag egymást követő elemeket általában egymás után szokás rajzolni. A lista egy elvi, absztrakt tároló, tehát létezhet számítógép nélkül is, például a képzeletünkben. A listának rengeteg fizikai megvalósítása lehetséges, ilyen például az életben a teendők listája: sorban intézzük a listára írt teendőket, miközben újabbakat is listára írunk, másokat pedig kitörölünk onnan. Mi most természetesen a listának a számítógépes megvalósításaival foglalkozunk. A lista számítógépen is elképzelhető háttértárolón vagy memóriában, és a memóriában is lehet statikus, vagy dinamikus. Statikus lista lehet például egy tömb, melynek elemei rekordok – minden rekord az adatokon kívül tartalmaz egy indexet, mely az őt követő elem tömbbeli indexe:

```
Type
   Listaelem = Record
      Adat : ... ;
      Mutató : Index ;
   End ;

Var
   Lista : Array[1..Max] Of Listaelem ;
   L : Index ;
```



Ez a lista a 2 indexű elemmel kezdődik. 2 mutat 4-re, 4 mutat 1-re, 1 mutat 3-ra, 3 mutat 5-re, 5 pedig 6-ra. Így az elemek bejárási sorrendje a következő: **2,4,1,3,5,6.** A lista adatai sorban: **1,2,4,8,6,3.**

Ha az *I* által mutatott listaelem adatát *Adat(I)*-nek, mutatóját pedig *Mutató(I)*-nek nevezzük, akkor a listát az *I* mutatóval a következőképpen járhatjuk be:

```
I := L
Ciklus amíg I <> Végjel
   Adat(I) feldolgozása
   I := Mutató(I)
Ciklus vége
```

Mindig az *I* által mutatott adatot dolgozzuk fel.

A statikus lista óriási hátránya, hogy

- egyrészt a lista maximális mérete előre rögzített, és így a rendelkezésre álló memória hamar elfogyhat,
- másrészt pedig a lekapcsolt elemek révén felszabadult helyeket a felhasználónak kell karbantartania, ha azokat újra fel akarja használni.

Definíciók

- *Dinamikus lista*: a lista elemeit szükség szerint hozzuk létre, illetve szüntetjük meg. A dinamikus lista a memóriából annyi helyet foglal el, amennyire éppen szüksége van, átengedve ezzel a felesleges memóriahelyeket más tárolószerkezetek számára.

- *Rendezett lista:* a lista elemei valamilyen szempont szerint jól meghatározott sorrendben követik egymást.
- *Kétirányú (szimmetrikus) lista:* a lista oda-vissza mutatókkal van felszerelve, és így az elemek előre és visszafelé is feldolgozhatók.
- *Nyíltvégű lista:* A lista utolsó vagy első eleme egy nem létező listaelemre mutat (végjel).
- *Cirkuláris (zárt) lista*: a lista utolsó eleme az első elemre mutat, vagyis annak mutatója nem a végjel. A cirkuláris lista mutatói körbeérnek.

A listákon végezhető műveleteket és azok különböző megvalósításait dinamikus listákon mutatjuk meg.

9.2 Egyirányú, nyíltvégű dinamikus lista karbantartása

A dinamikus lista elemei dinamikusak, vagyis a heap-ben vannak, s azokra mutatókkal lehet hivatkozni. Minden listaelem tartalmaz egy ilyen mutatót, mely a következő listaelemre mutat:

```
Type
  PListaelem = ^TListaelem ;
  TListaelem = Record
    Adat : ... ;
    Kov : PListaelem ;
  End ;
```

Az első elem mutatója a listát egyértelműen azonosítja:

```
Var
  L : PListaelem ;
```

Megjegyzés: *^Típus* szerepelhet a deklarációban akkor is, ha *Típust* előzőleg még nem deklaráltuk. Ha a Pascal ezt nem engedné meg, nem lehetne listát definiálni, hiszen a rekordban a mutatóra, a mutatóban pedig a rekordra kell hivatkoznunk (22-es csapdája). *Típus*t még ugyanabban a blokkban kötelező deklarálni!

Feladat
Írjunk programot, mely 0 végjelig bekér számokat, s azokat egy listára felfűzve a heap-ben tárolja. Írjuk ki a listát, és oldjuk meg az alapvető keresési és karbantartási funkciókat!

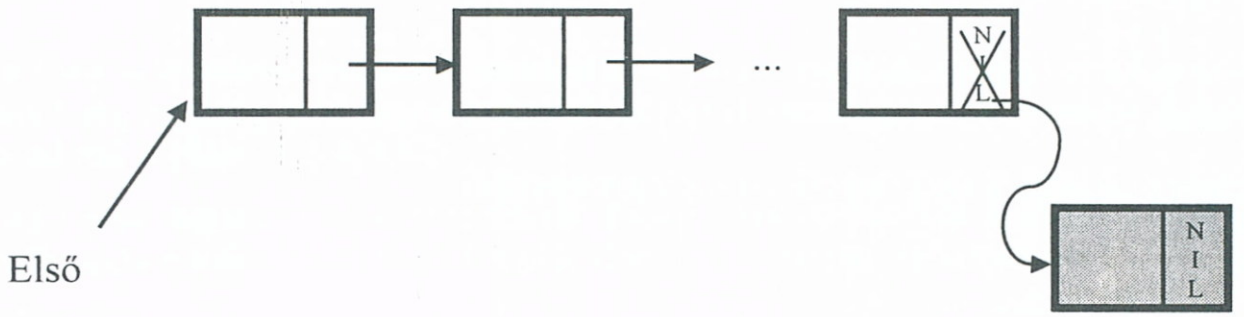
A számokat többféleképpen fogjuk felfűzni. Először a bevitel sorrendjében, vagyis az új számot mindig a lista végéhez fűzzük hozzá. Ezután a sorrend fordított lesz, a felfűzések elölről történnek. Végül a számokat rendezetten fogjuk elhelyezni a listában. A megoldás kapcsán bemutatjuk a lista szekvenciális feldolgozását, és szó lesz a rendezetlen, illetve rendezett listában való keresésről is.

A lista és a megoldáshoz szükséges segédváltozók deklarációja legyen a következő:

```
Type
   PElem = ^TElem ;
   TElem = Record
      Szam : LongInt ;
      Kov : PElem ;
   End ;

Var
   Elso : PElem ;
   Uj, Akt, Elozo, Torlendo : PElem ; { Segédmutatók }
   Szam : LongInt ;
```

A dinamikus lista bővítésének menete a következő: először lefoglaljuk az új elem számára a helyet. Ezután az adatokat kitöltjük, és az elemet felfűzzük a megadott listára (amely történetesen üres is lehet). E két utóbbi tevékenység (kitöltés és felfűzés) sorrendje elvileg lényegtelen. Felfűzéskor külön végig kell gondolnunk az elölről és a hátulról való kapcsolódást, és azon belül is meg kell gondolnunk a szélsőséges eseteket.

Elemek felfűzése – mindig a lista végére



- *Kapcsolódás elölről:* a most létrehozott új listaelemre az eddigi utolsó elem mutatója fog mutatni, feltéve, hogy a lista nem volt üres. Ha üres volt, akkor az új elem lesz az első.
- *Kapcsolódás hátulról:* mivel az új elemet most mindig utolsónak tesszük, ezért az ő mutatója minden esetben *Nil* lesz.

```
Elso := Nil ;
ReadLn(Szam) ;
While Szam <> 0 Do
   Begin
      { Helyfoglalás, adatok kitöltése: }
      New(Uj) ; Uj^.Szam := Szam ;

      { Kapcsolódás elölről: }
      If Elso = Nil Then
         Elso := Uj
      Else
         Akt^.Kov := Uj ;
```

```
    { Megjegyezzük a mindenkori utolsót, hogy a követ-
    kező ciklusban ehhez fűzhessük hozzá az új elemet:}
    Akt := Uj ;

    {Kapcsolódás hátulról: }
    Uj^.Kov := Nil ;

    ReadLn(Szam) ;
  End ;
```

Megjegyzés: Ha a felvitel nem folyamatos, akkor az új elem kapcsolása előtt meg kell határozni az utolsó elemet!

Szekvenciális feldolgozás - elemek kiírása

```
WriteLn('A számok:') ;
Akt := Elso ;
While Akt <> Nil Do
  Begin
    Write(Akt^.Szam:8) ;
    Akt := Akt^.Kov ;
  End ;
```

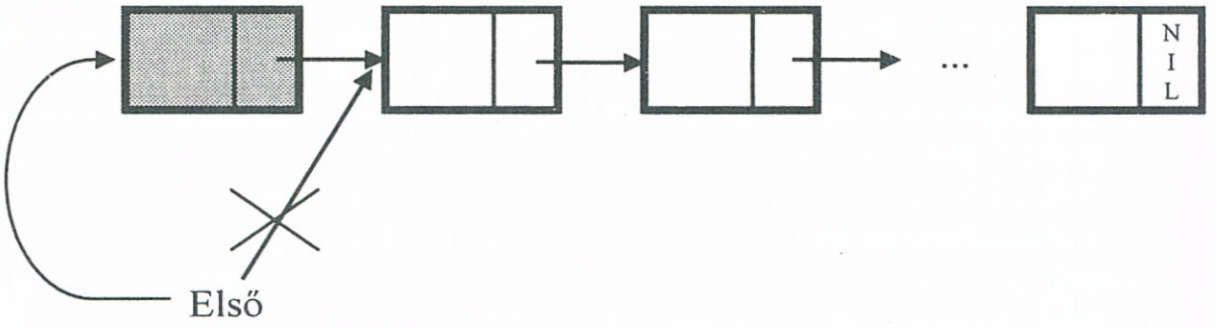
Sose hivatkozzunk nem létező listaelemre, és külön ügyeljünk arra, hogy a lista elejét és végét korrekt módon dolgozzuk fel. Ne mutassunk sehova egy olyan mutatóval, melynek értéke *Nil*. Ha például a lista üres, akkor az *Akt:=Elso* utasítás után *Akt= Nil*. Ekkor *Akt*^-ra való hivatkozás futási hibát eredményezhet!

Keresés a rendezetlen listában

```
WriteLn('Keresendő:') ;
ReadLn(Szam) ;
Akt := Elso ;
{$B-}
While (Akt <> Nil) And (Akt^.Szam <> Szam) Do
  Akt := Akt^.Kov ;

If Akt = Nil Then
  WriteLn('Nincs')
Else
  WriteLn('Van') ;
```

Figyeljük meg a hasonlatosságot a rendezetlen tömbben való kereséshez! A keresés itt is elölről történik, és addig megy, amíg a keresett számot meg nem találjuk, illetve a lista végére nem érünk.

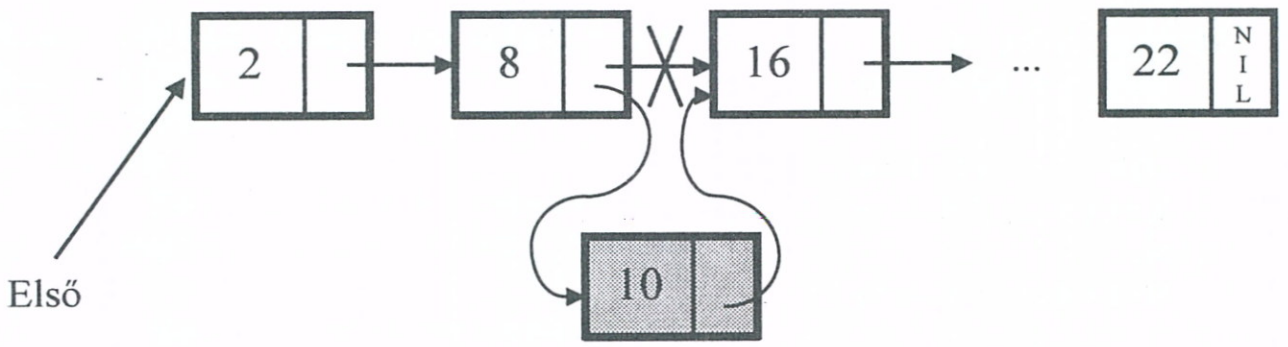
Elemek felfűzése - mindig a lista elejére



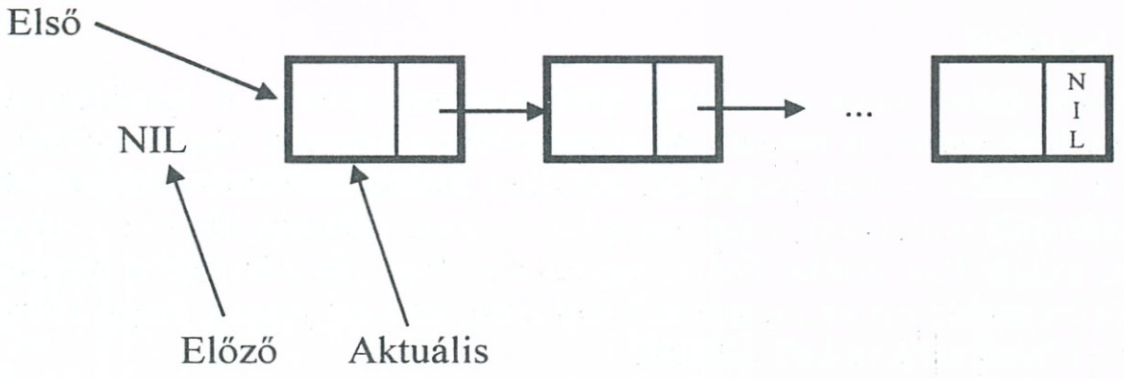
- *Kapcsolódás elölről:* az új listaelem lesz az első.
- *Kapcsolódás hátulról:* az új listaelem az eddigi első elemre fog mutatni, akkor is, ha az *Nil* volt, vagyis üres volt a lista. Vigyázni kell a műveletek sorrendjére, nehogy felülírjuk az *Első* mutatót addig, amíg azt át nem írtuk az új listaelembe!

```
Elso := Nil ;
ReadLn(Szam) ;
While Szam <> 0 Do
   Begin
      { Helyfoglalás, adatok kitöltése: }
      New(Uj) ;
      Uj^.Szam := Szam ;

      { Felfűzés: }
      Uj^.Kov := Elso ;
      Elso := Uj ;

      ReadLn(Szam) ;
   End ;
```

Elemek felfűzése rendezetten



Ez a legbonyolultabb eset, hiszen most felfűzés előtt meg is kell keresni az elem helyét. Ha például az eddigi rendezetten felfűzött számok: 2,8,16,22, akkor a 10-est a 8-as és 16-os elemek közé kell beiktatnunk. Ezt az ember persze ránézésre megmondja, de mit tápláljunk be a programba? Ha a lista még üres, a legelső elemet most is egyszerű felfűzni. A másodikat már lehet, hogy a meglévő szám elé, lehet, hogy mögé kell beszúrni. Egy tetszőleges elemet vagy a lista elejére, vagy a végére, vagy pedig két elem közé kell betenni. Tegyük fel, hogy egy *Aktuális* mutatóval végigmegyünk a rendezett listán, keresve az új elem helyét. A 2 kisebb, mint a 10, ezért továbblépünk. 8 is kisebb, megint továbblépünk. Mutatónk most a 16-oson áll, ami már nagyobb a 10-esnél, így megállunk. Az új számot tehát a 16-os elé kellene beszúrni, vagyis az *Előző* és az *Aktuális* közé. A program már nem tudja, mi volt az előző, hiszen már a 16-oson áll, és visszafelé nincs mutató. Lépésről lépésre meg kell tehát jegyeznünk az előző elem mutatóját, hogy az az adott pillanatban „kéznél legyen". Amikor a lista elején állunk, akkor *Előző* legyen *Nil*, aktuális pedig legyen az *Első*. A lista végén *Előző* az utolsó listaelemre mutat, míg az *Aktuális* értéke *Nil*. Így az *Előző* és az *Aktuális* mutatók mindenképpen közrefogják majd az új elemet:



- *Kapcsolódás elölről:* ha *Előző=Nil*, vagyis az új elem az összes elemnél kisebb, akkor az új elem lesz az első, egyébként az előző elem mutatóját állítjuk rá az újra.
- *Kapcsolódás hátulról:* az új listaelem következője az *Aktuális* lesz. Ha *Aktuális=Nil*, akkor az új elem lesz az utolsó.

```
Elso := Nil ;
ReadLn(Szam) ;
While Szam <> 0 Do
  Begin
    { Helyfoglalás, adatok kitöltése: }
    New(Uj) ;
    Uj^.Szam := Szam ;

    { A szám helyének keresése: }
    Elozo := Nil ;
    Akt := Elso ;
```

```
    While (Akt<>Nil) And (Akt^.Szam<Szam) Do
      Begin
        Elozo := Akt ;
        Akt:= Akt^.Kov ;
      End ;

    { Felfűzés: }
    If Elozo = Nil
    Then
      Elso := Uj
    Else
      Elozo^.Kov := Uj ;
    Uj^.Kov := Akt ;

    ReadLn(Szam) ;
  End;
```

Törlés a listából

Ha a törlendő szám nincs az elemek közt, üzenetet adunk. A feladatot rendezett és rendezetlen lista esetén is megadjuk. A megoldás a keresést végző eljárást leszámítva ugyanaz, a keresést külön megadjuk mindkét esetre. A *Keresés* eljárás beállítja az *Előző* és *Aktuális* mutatókat, valamint a *Van* logikai változót. Ha nincs a keresett elem a listában, akkor *Van* értéke *False*, egyébként *True*, és *Aktuális* a megtalált elemre mutat – törlés esetén ezt az elemet kell kiiktatni, ami azt jelenti, hogy *Előző* mutatóját az *Aktuális* következőjére kell állítani.

A törlés menete a következő: először a törölt elemet le kell kapcsolnunk a listáról – a memória felszabadítását csak ezután szabad elvégezni, hiszen a lekapcsoláshoz szükségünk van a törlendő elem *Következő* adatára.

A felszabadított területre hivatkozni nem szabad, mert azt a heap-kezelő saját adataival írja felül!

```
WriteLn('A törlendő:') ;
ReadLn(Szam) ;

Kereses ; { Elozo és Akt beállítása }
If Van
Then
  Begin
    { Lekapcsolás: }
    If Elozo = Nil
    Then
      Elso := Akt^.Kov
    Else
      Elozo^.Kov := Akt^.Kov ;
```

```
      { Hely felszabadítása: }
      Dispose(Akt) ;
    End
  Else
    WriteLn('Nincs');
```

Vigyázat! A listáról való lekapcsolás nem szabadítja fel a helyet, arról külön kell gondoskodnunk!

Keresés, ha a lista nem rendezett:

```
Elozo := Nil ;
Akt := Elso ;
While (Akt <> Nil) And (Akt^.Szam <> Szam) Do
  Begin
    Elozo := Akt ;
    Akt := Akt^.Kov ;
  End ;
Van := Akt <> Nil ;
```

Keresés, ha a lista rendezett:

```
Elozo := Nil ;
Akt := Elso ;
While (Akt <> Nil) And (Akt^.Szam < Szam) Do
  Begin
    Elozo := Akt ;
    Akt := Akt^.Kov ;
  End ;
Van := (Akt <> Nil) And (Akt^.Szam = Szam) ;
```

Teljes lista megszüntetése

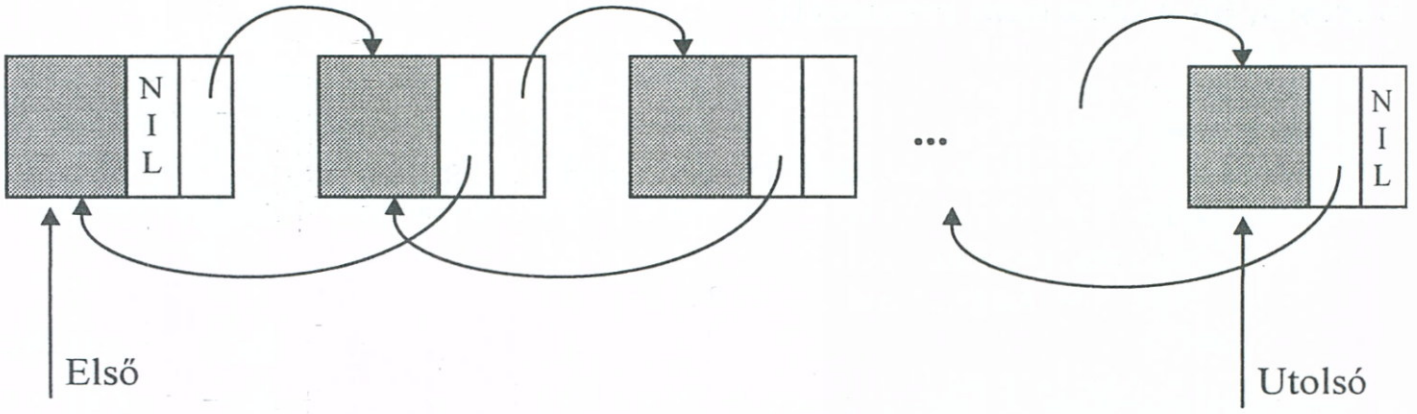
Ha az összes listaelemet sorban fel akarjuk szabadítani, akkor nem szükséges lekapcsolásokat végezni, elegendő a helyeket felszabadítani, végül a lista mutatóját *Nil*-re állítani. Az elemeket sorban, a kapcsolódások mentén szabadítjuk fel:

```
Akt := Elso ;
While Akt <> Nil Do
  Begin
    Torlendo := Akt ;
    Akt := Akt^.Kov ;
    Dispose(Torlendo) ;
  End ;
Elso := Nil ;
```

Vigyázni kell, hogy még felszabadítás előtt „kiszedjük" a törlendő elemből azt az információt, hogy melyik elem követi őt – egyébként a lista megszakad!

9.3 Kétirányú lista

A *kétirányú*, más néven *szimmetrikus lista* minden eleme tartalmazza az őt követő és az őt megelőző elem mutatóját is. Az elemek tehát mindkét irányban fel vannak fűzve:



A visszafelé olvasáshoz most a listát az utolsó elemmel kell kezdeni, ezért szükségünk van egy *Utolsó* mutatóra. Ha a listát csak az első elemen keresztül érnénk el, akkor az utolsó megállapításához az egész listát végig kellene olvasnunk. Az utolsó elem mutatóját folyamatosan karban kell tartani. Legyenek most a rendezetten felfűzendő adatok szövegek – a deklaráció a következő:

```
Type
   TSzoveg = String[40] ;
   PElem = ^TElem ;
   TElem  = Record
      Kov : PElem ;
      Elozo : PElem ;
      Szoveg : TSzoveg ;
   End ;

Var
   Elso, Utolso : PElem ;
   Szoveg : TSzoveg ;
```

A mutatók és adatok deklarálási sorrendje elvileg lényegtelen, mégis szokásosabb azokat a listaelem elejére tenni. A lista inicializálásakor most az *Első* és *Utolsó* mutatókat is be kell állítani. A felfűzéseket végezze el most a *Felfűz* eljárás, mely paraméterként megkapja a lefoglalt területet, és a szöveget.

A *New* eljárás a Turbo Pascal-ban függvényként is hívható – ekkor annak paramétere egy *típusos mutató* típus. A *New* függvény lefoglalja a típusnak megfelelő nagyságú területet a heap-ben, a visszaadott mutató értéke erre a területre mutat:

```
Elso := Nil ;
Utolso := Nil ;

Write('Szöveg: ') ; ReadLn(Szoveg) ;
While Szoveg <> '*' Do
  Begin
    Felfuz(New(PElem),Szoveg) ;
    Write('Szöveg: ') ; ReadLn(Szoveg) ;
  End ;
```

Felfűzés:

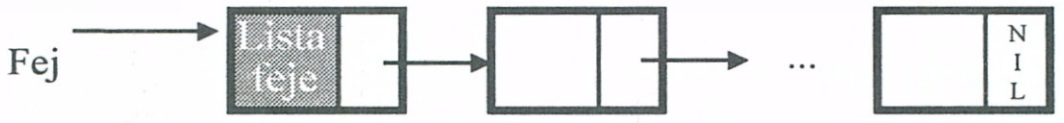
```
Procedure Felfuz(Uj: PElem; Szoveg: TSzoveg) ;
  Var
    Elozo, Akt : PElem ;
  Begin
    Elozo := Nil ;
    Akt := Elso ;
    While (Akt <> Nil) And (Szoveg > Akt^.Szoveg) Do
      Begin
        Elozo := Akt ;
        Akt := Akt^.Kov ;
      End ;

    If Elozo = Nil
    Then
      Elso := Uj
    Else
      Elozo^.Kov := Uj ;

    If Akt = Nil
    Then
      Utolso := Uj
    Else
      Akt^.Elozo := Uj ;

    Uj^.Szoveg := Szoveg ;
    Uj^.Kov := Akt ;
    Uj^.Elozo := Elozo ;
  End ;
```

Lista hátrafelé:

```
Akt := Utolso ;
While Akt <> Nil Do
   Begin
      WriteLn(Akt^.Szoveg) ;
      Akt := Akt^.Elozo ;
   End ;
```

9.4 Fejelt lista

Látható, hogy az üres lista, illetve a lista végének lekezelése sok esetben bonyolítja az elvégzendő feladatot. A gyakorlatban jól bevált módszer, ha a lista elejére felveszünk egy listafejet, amely mindig létezik. Ha csak a lista feje van meg, akkor a lista üres:



Fejelt kétirányú lista

Fejeljük meg az előző pontban tárgyalt kétirányú listát, és oldjuk meg úgy a feladatot! A lista inicializálásakor az *Első* mutató helyett vegyünk fel egy *PElem* típusú *Fej* mutatót, és állítsuk *Fej*^ mutatóit mindkét irányban *Nil*-re:

```
Var
   Fej : Pelem ;
...
New(Fej) ;
Fej^.Elozo := Nil ;
Fej^.Kov := Nil ;
Utolso := Nil ;
```

A keresésnek csak az indítása más:

```
Elozo := Fej ;
Akt := Fej^.Kov ;
```

Mivel a lista sosem lesz üres (*Fej* nem lehet *Nil*), az elölről való felfűzéskor szélsőséges esetet nem kell kezelnünk:

```
Elozo^.Kov := Uj ;
```

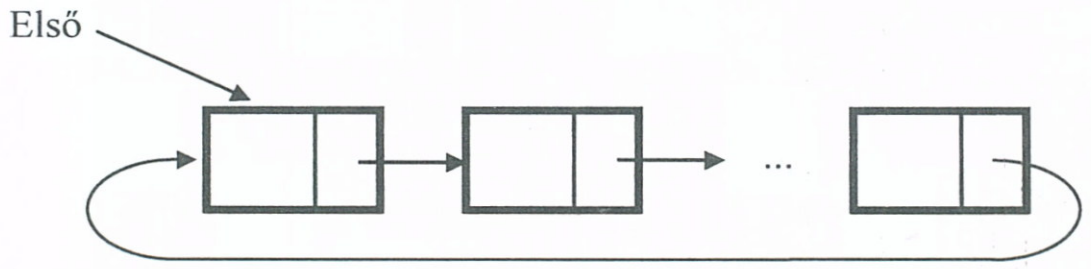
Lista előre:

```
Akt := Fej^.Kov ;
While Akt <> Nil Do
   Begin
      WriteLn(Akt^.Szoveg) ;
      Akt := Akt^.Kov ;
   End ;
```

A fejnek van még egy jó tulajdonsága: az adat részében tárolhatunk mindenféle – a listával kapcsolatos – információt (mint például a lista aktuális méretét), amely gyorsíthatja a feldolgozást.

9.5 Cirkuláris lista

Az olyan listát, amelyben az utolsó elem mutatója végjel, *földelt* vagy *nyílt listá*nak is szokás nevezni. A *cirkuláris*, más néven *gyűrűs* vagy *zárt lista* esetén a mutatók körbeérnek, vagyis az utolsó elem mutatója az elsőre mutat. A mutatólánc ekkor zárt:



Cirkuláris, kétirányú fejelt lista

Természetesen a kétirányú és a fejelt lista is lehet cirkuláris. Nézzük meg, hogy alakul a szövegek felfűzése *cirkuláris fejelt lista* segítségével. A listára most egyedül a *Fej* mutat. Az utolsó elemre nincs szükségünk, mert azt könnyedén megkaphatjuk egyetlen lépéssel: Fej^.Előző. A lista inicializálása most a fej létrehozásából és mutatóinak önmagára állításából áll:

```
Var
   Fej : Pelem ;
...
New(Fej) ;
Fej^.Elozo := Fej ;
Fej^.Kov := Fej ;
```

A felfűzés még egyszerűbb lesz, mert az utolsó elemmel sem kell bajlódni.

Vigyázat! Ha a cirkuláris lista fej nélküli, egészen más a helyzet!

Keresés:

```
Elozo := Fej ;
Akt := Fej^.Kov ;
While (Akt <> Fej) And (Szoveg > Akt^.Szoveg) Do
  Begin
    Elozo := Akt ;
    Akt := Akt^.Kov ;
  End ;
```

Felfűzés:

```
Elozo^.Kov := Uj ;      { elölről }
Uj^.Elozo := Elozo ;
Uj^.Kov := Akt ;        { hátulról }
Akt^.Elozo := Uj ;
```

Lista előre és hátra:

```
Akt := Fej^.Kov ;
While Akt <> Fej Do
  Begin
    WriteLn(Akt^.Szoveg) ;
    Akt := Akt^.Kov ;
  End ;

Akt := Fej^.Elozo ;
While Akt <> Fej Do
  Begin
    WriteLn(Akt^.Szoveg) ;
    Akt := Akt^.Elozo ;
  End ;
```

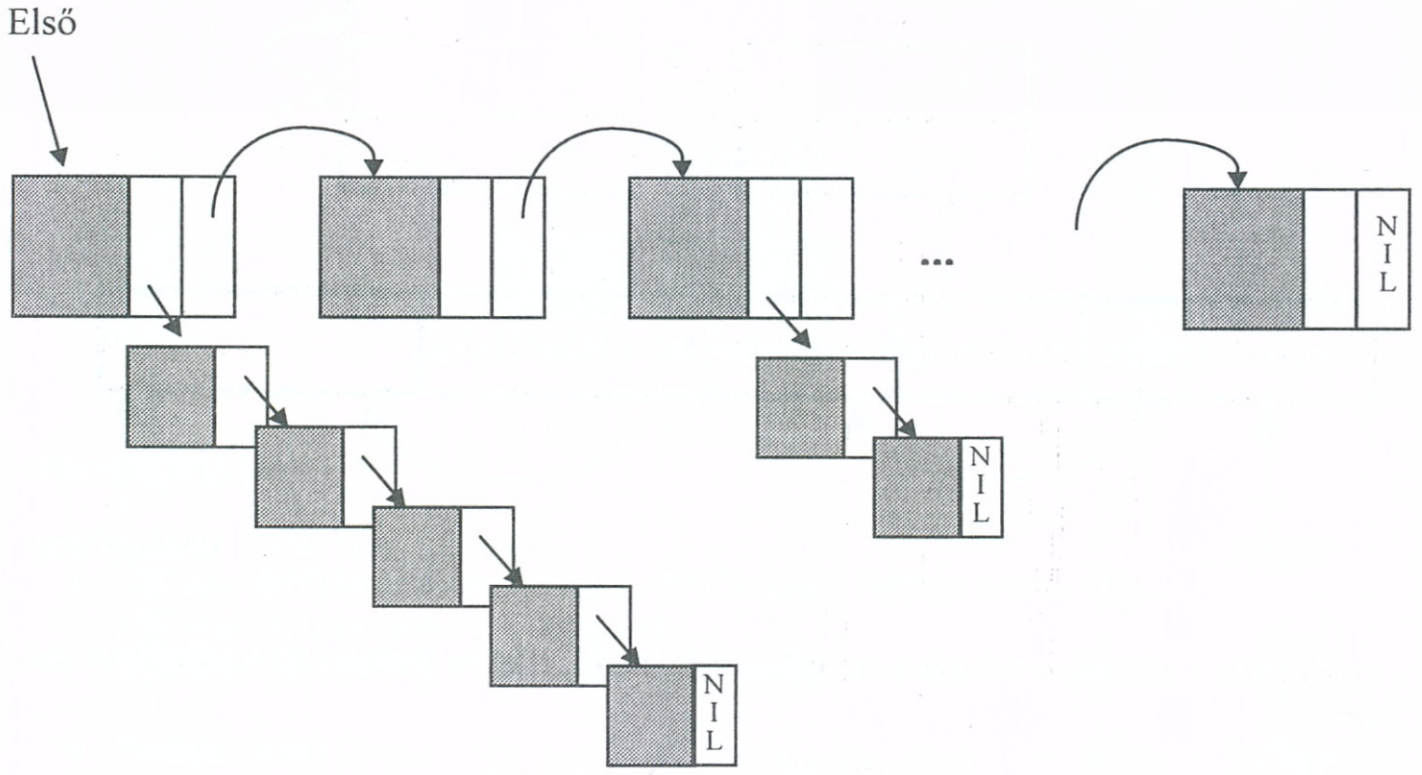
A fejben lévő információkat egy másik típus rákényszerítésével manipulálhatjuk:

```
Type
  PElem = ^TElem ;
  TElem  = Record
    Kov, Elozo : PElem ;
    Szoveg : TSzoveg ;
  End ;

  PFej = ^TFej ;
  TFej = Record
    Kov, Elozo : PElem ;
    N : Word ;
  End ;
```

```
Var
   Fej : PElem ;
...
PFej(Fej)^.N := 0 ;
Inc(PFej(Fej)^.N) ;
For I := 1 To PFej(Fej)^.N Do
   ...
```

9.6 Multilista

Az olyan listát, amelyben az elemek újabb listák kiinduló pontjai lehetnek, multilistának, illetve többszörös listának nevezzük:



A multilistát egy feladaton keresztül mutatjuk be:

Feladat
Olvassunk be csoportszámokat végjelig! Minden csoporthoz olvassuk be a hallgatók névsorát! Az adatokat a beolvasás sorrendjében tároljuk!

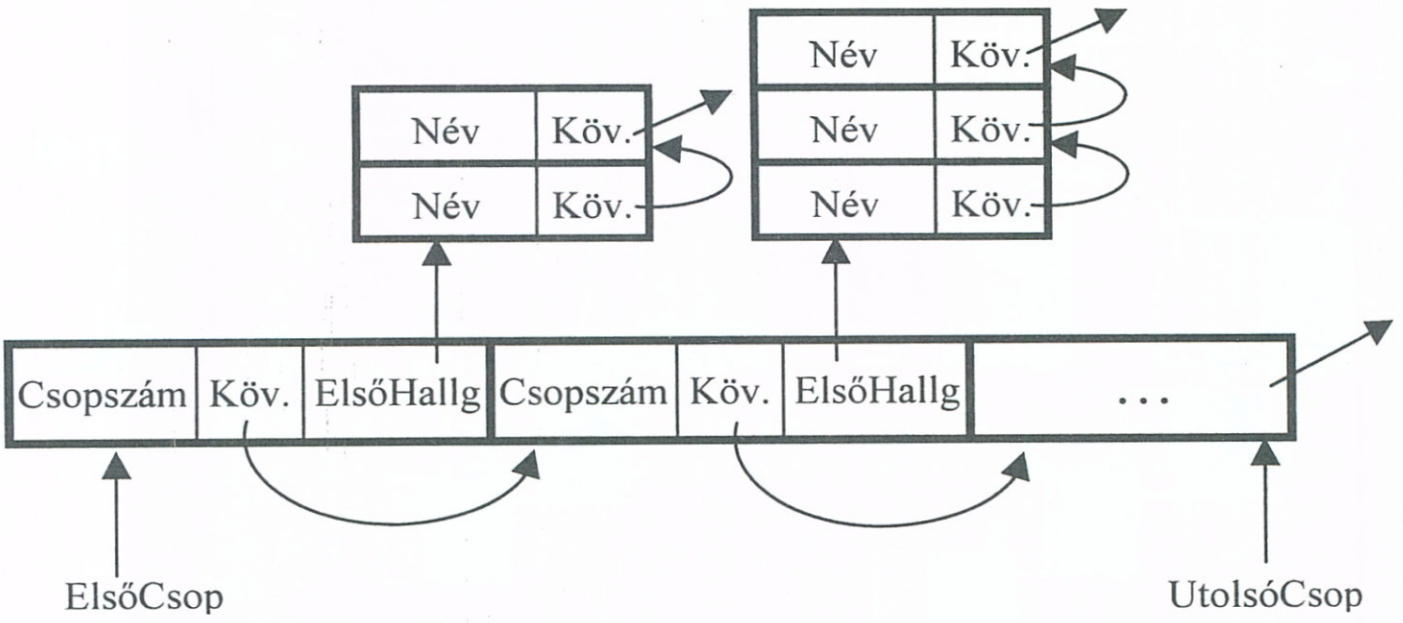
- Listázzuk ki a csoportszámokat és az azokhoz tartozó hallgatókat!
- Listázzuk ki az összes csoportszámot!
- Írjuk ki az összes hallgató nevét!
- Töröljük ki egy megadott csoport összes hallgatóját!

A feladat megoldásához először a globális adatszerkezeteket kell meghatároznunk. Rögtön adódik, hogy adatainkat multilista segítségével kell tárolnunk, hiszen az első szinten csoportokat kell felfűzni, és minden csoportnak kell, hogy legyen egy allistája, melyekre a hallgatók kerülnek. Kérdés most már csak az, milyen típusú listákat válasszunk. Legyen most mindkét fajta lista (a csoportlista és a hallgatói lista) *egyszerű (nem fejelt, nem cirkuláris) és rendezetlen*. A csoportokat és azon belül a hallgatókat „érkezési" sorrendben tároljuk el.

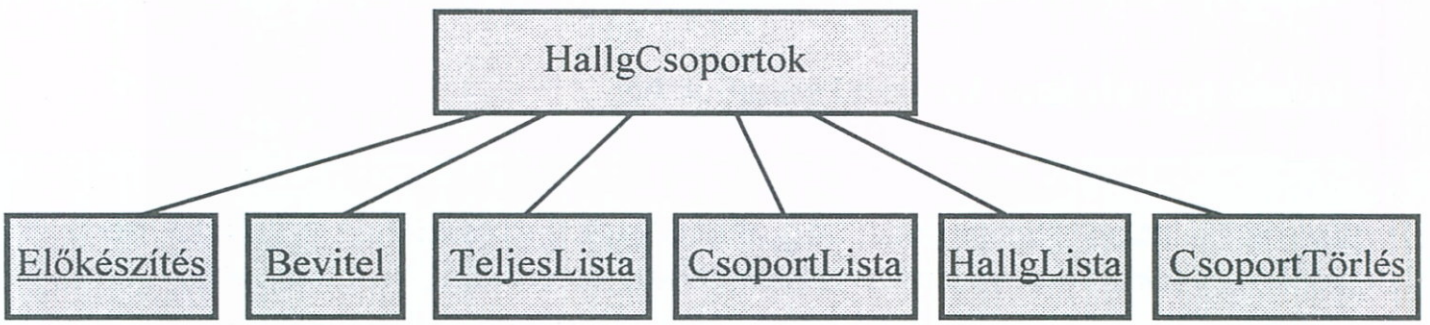
Adatszerkezetek:

THallgató = Rekord (Név: Szöveg(30), Köv: PHallgató)

TCsoport = Rekord (Csopszám: Szöveg(10), Köv: PCsoport, ElsőHallg: PHallgató)



A program Jackson terve ezt a multilistát építi fel, illetve listázza ki különböző módokon:

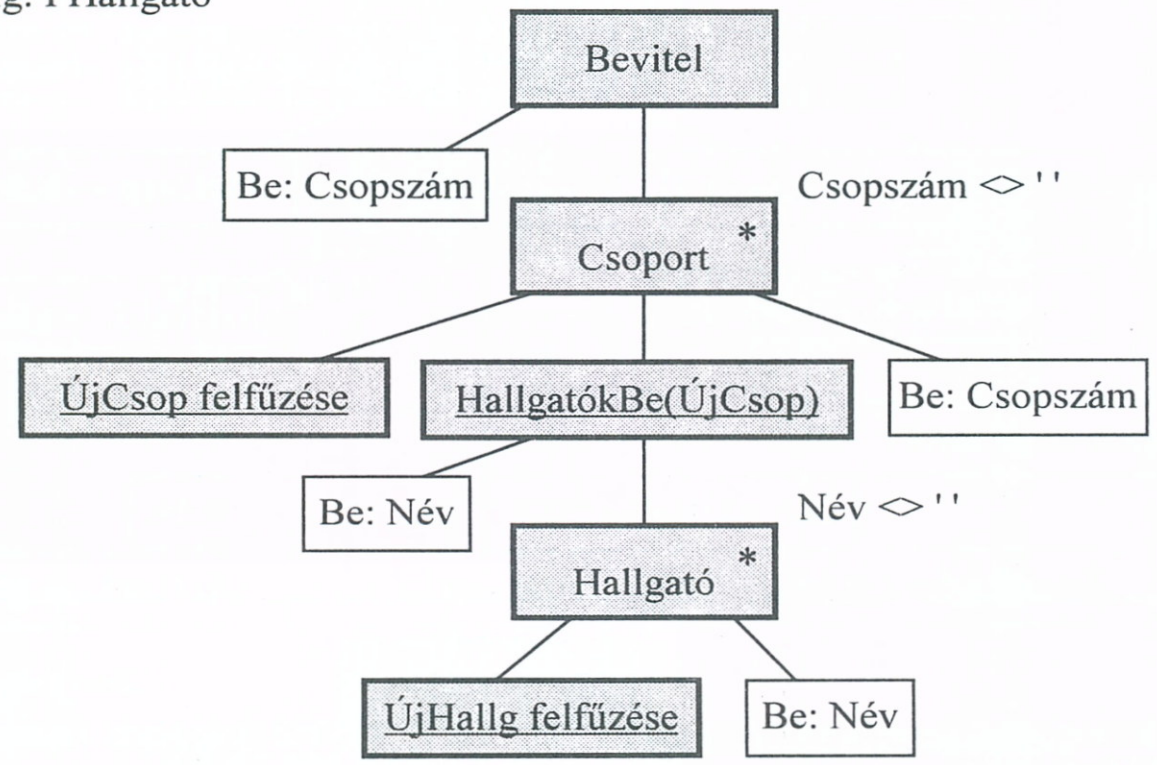


Tevékenységjegyzék:

- Előkészítés: ElsőCsop ← Nil : UtolsóCsop ← Nil
- Bevitel, TeljesLista, CsoportLista, HallgLista, CsoportTörlés: lásd később.

Adatok:

Csopszám, Név: Szöveg
ÚjCsop: PCsoport
ÚjHallg: PHallgató

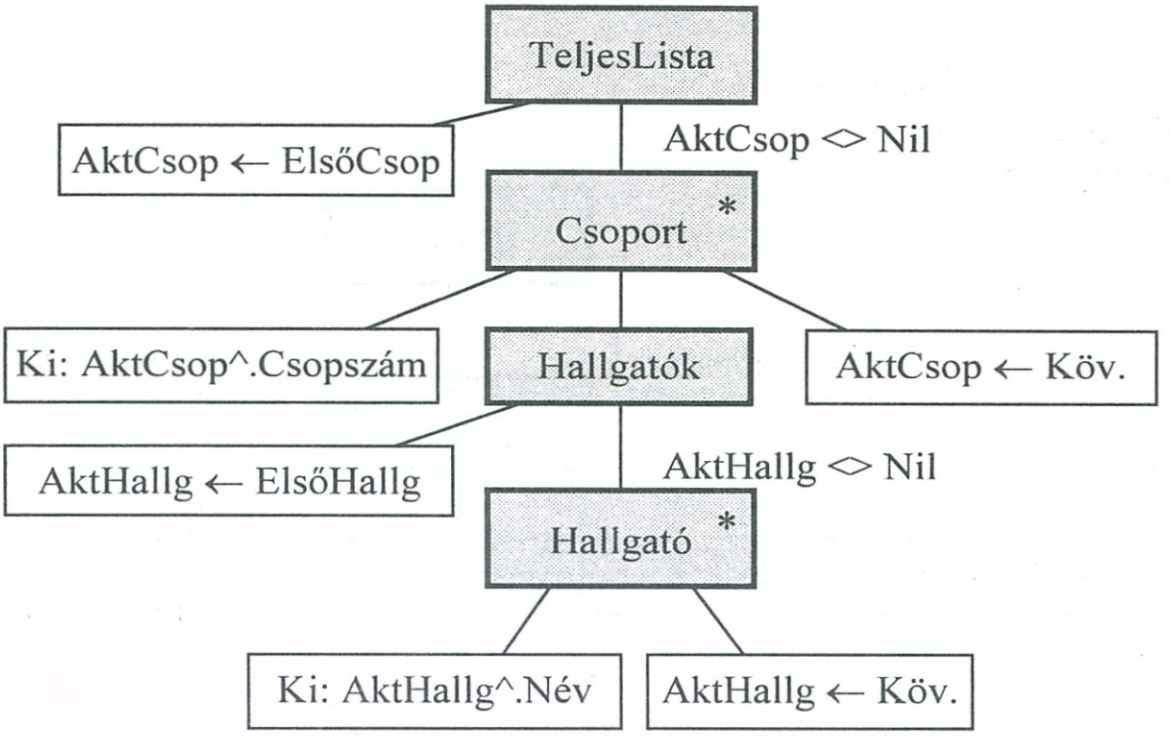


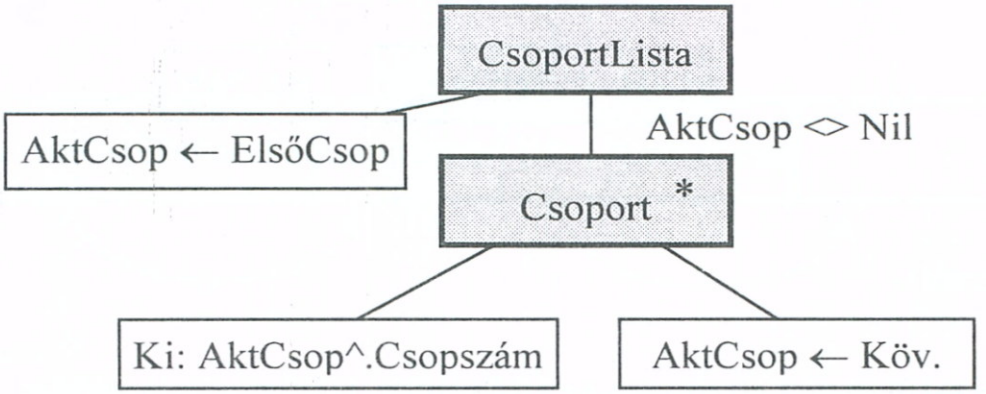
Tevékenységjegyzék:

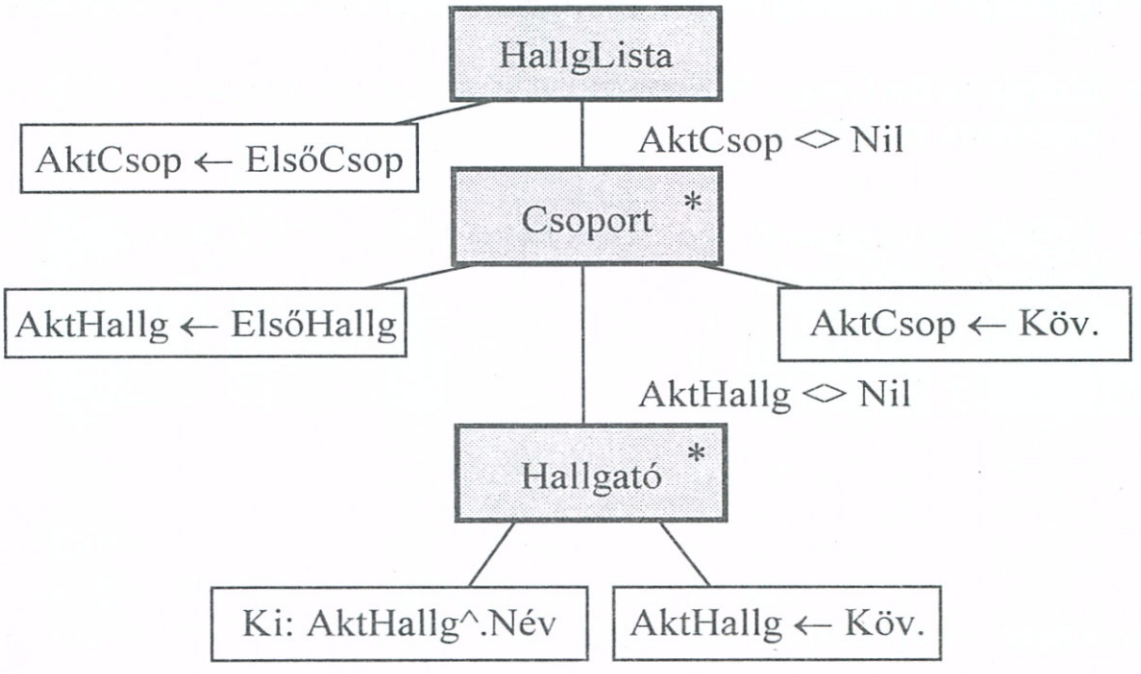
- ÚjCsop felfűzése: Létrehozza az ÚjCsop-ot, felfűzi a lista végére.
  ÚjCsop^.Csopszám ← Csopszám
  ÚjCsop^.ElsőHallg ← Nil; (hallgatók még nincsenek a csoportban)

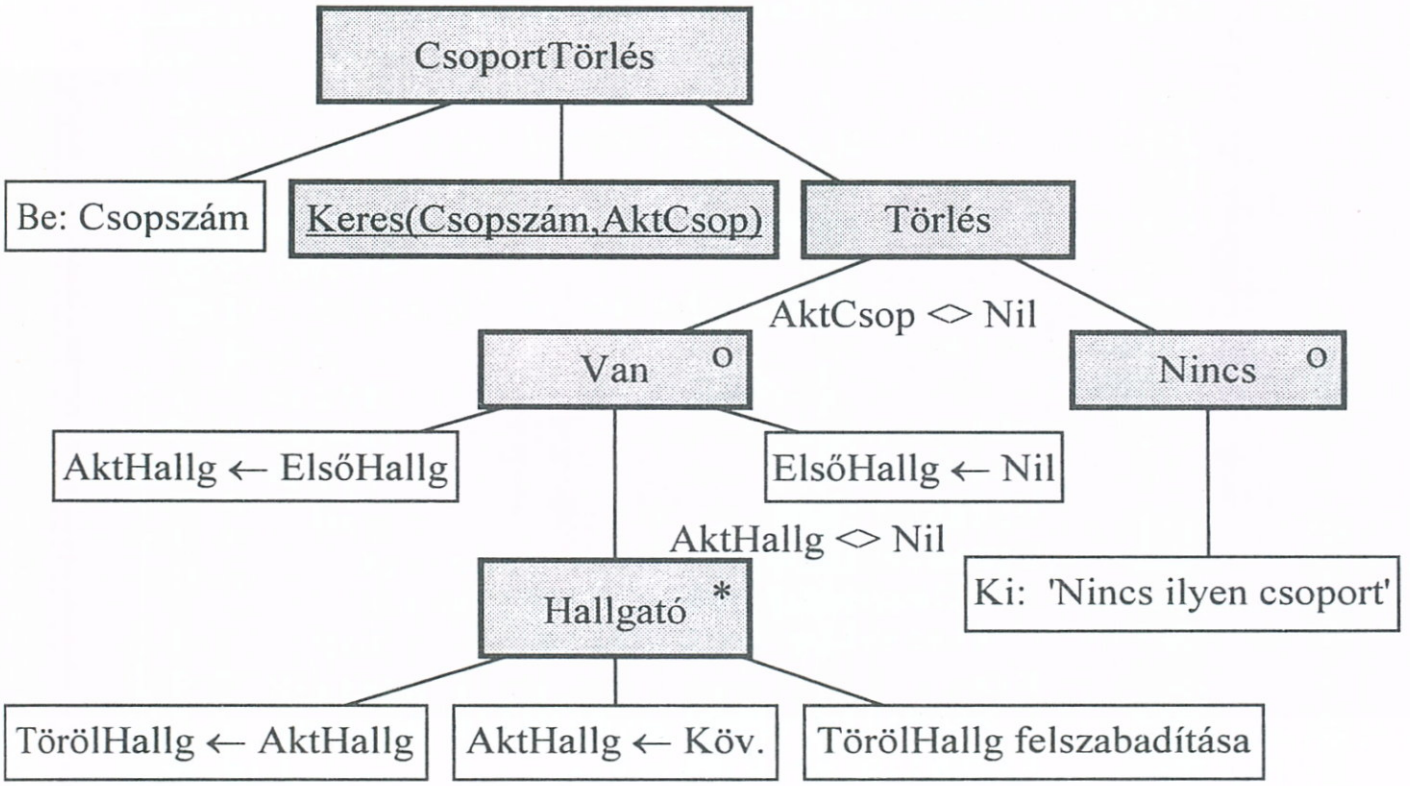
- HallgatókBe(Csop: PCsoport): A Csop által megadott csoport hallgatóit beolvassa és felfűzi.

- ÚjHallg felfűzése: A beolvasott nevet felfűzi az ÚjCsop^.Első által mutatott hallgatói lista végére.


TeljesLista
AktCsop ← ElsőCsop
AktCsop <> Nil
Csoport *
Ki: AktCsop^.Csopszám
Hallgatók
AktCsop ← Köv.
AktHallg ← ElsőHallg
AktHallg <> Nil
Hallgató *
Ki: AktHallg^.Név
AktHallg ← Köv.


CsoportLista
AktCsop ← ElsőCsop
AktCsop <> Nil
Csoport *
Ki: AktCsop^.Csopszám
AktCsop ← Köv.


HallgLista
AktCsop ← ElsőCsop
AktCsop <> Nil
Csoport *
AktHallg ← ElsőHallg
AktCsop ← Köv.
AktHallg <> Nil
Hallgató *
Ki: AktHallg^.Név
AktHallg ← Köv.



Tevékenységjegyzék:

- Keres(Csopszám: Szöveg, Var Csop: PCsoport): Megkeresi a csoportot Csopszám alapján. Ha megvan, Csop a csoport mutatója, egyébként Nil.

És végül elkészítjük a program kódját:

```
Program HallgCsoportok ;

Type
  PHallgato = ^THallgato ;
  THallgato = Record
    Nev : String[30] ;
    Kov : PHallgato ;
  End ;

  PCsoport = ^TCsoport ;
  TCsoport = Record
    Csopszam : String[10] ;
    Kov : PCsoport ;
    ElsoHallg : PHallgato ;
  End ;

Var
  ElsoCsop,
  UtolsoCsop: PCsoport ;
```

```
Procedure HallgatokBe(Csop: PCsoport) ;
  Var
    Nev : String[30] ;
    UjHallg, AktHallg : PHallgato ;
  Begin
    With Csop^ Do
      Begin
        WriteLn(Csopszam,' hallgatói') ;
        ReadLn(Nev) ;
        While Nev <> '' Do
          Begin
            New(UjHallg) ;
            UjHallg^.Nev := Nev ;
            UjHallg^.Kov := Nil ;
            If ElsoHallg = Nil Then
              ElsoHallg := UjHallg
            Else
              AktHallg^.Kov := UjHallg ;
            AktHallg := UjHallg ;
            ReadLn(Nev) ;
          End ;
      End ;
  End ;

Procedure Bevitel ;
  Var
    Csopszam : String[10] ;
    UjCsop : PCsoport ;
  Begin
    Write('Csoportszám: ') ; ReadLn(Csopszam) ;
    While Csopszam <> '' Do
      Begin
        New(UjCsop) ;
        UjCsop^.Csopszam := Csopszam ;
        UjCsop^.ElsoHallg := Nil ;
        UjCsop^.Kov := Nil ;
        If ElsoCsop = Nil Then
          ElsoCsop := UjCsop
        Else
          UtolsoCsop^.Kov := UjCsop ;
        UtolsoCsop := UjCsop ;
        HallgatokBe(UjCsop) ;
        Write('Csoportszám: ') ; ReadLn(Csopszam) ;
      End ;
  End ;
```

```
Procedure TeljesLista ;
  Var
    AktCsop : PCsoport ;
    AktHallg : PHallgato ;
  Begin
    WriteLn('Csoportok, hallgatók:') ;
    AktCsop := ElsoCsop ;
    While AktCsop <> Nil Do
      Begin
        WriteLn(AktCsop^.Csopszam) ;
        AktHallg := AktCsop^.ElsoHallg ;
        While AktHallg <> Nil Do
          Begin
            WriteLn('    ',AktHallg^.Nev) ;
            AktHallg := AktHallg^.Kov ;
          End ;
        WriteLn ;
        AktCsop := AktCsop^.Kov ;
      End ;
    ReadLn ;
  End ;

Procedure CsoportLista ;
  Var
    AktCsop : PCsoport ;
  Begin
    WriteLn('Csoportok:') ;
    AktCsop := ElsoCsop ;
    While AktCsop <> Nil Do
      Begin
        WriteLn(AktCsop^.Csopszam) ;
        AktCsop := AktCsop^.Kov ;
      End ;
    ReadLn ;
  End ;

Procedure HallgLista ;
  Var
    AktCsop : PCsoport ;
    AktHallg : PHallgato ;
  Begin
    WriteLn('Hallgatók:') ;
    AktCsop := ElsoCsop ;
    While AktCsop <> Nil Do
      Begin
        AktHallg := AktCsop^.ElsoHallg ;
```

```
        While AktHallg <> Nil Do
          Begin
            WriteLn('    ',AktHallg^.Nev) ;
            AktHallg := AktHallg^.Kov ;
          End ;
        AktCsop := AktCsop^.Kov ;
      End ;
    ReadLn ;
  End ;

Procedure CsoportTorles ;
  Var
    Csopszam : String[10] ;
    AktCsop : PCsoport ;
    AktHallg, TorolHallg : PHallgato ;
  Begin
    WriteLn('Melyik csoport törlendő?') ;
    ReadLn(Csopszam) ;
    AktCsop := ElsoCsop ;
    While (AktCsop <> Nil) And
      (AktCsop^.Csopszam <> Csopszam) Do
      AktCsop := AktCsop^.Kov ;
    If AktCsop <> Nil Then
      Begin
        AktHallg := AktCsop^.ElsoHallg ;
        While AktHallg <> Nil Do
          Begin
            TorolHallg := AktHallg ;
            AktHallg := AktHallg^.Kov ;
            Dispose(TorolHallg) ;
          End ;
        AktCsop^.ElsoHallg := Nil ;
      End
    Else
      WriteLn('Nincs ilyen csoport') ;
  End ;

Begin { HallgCsoportok főprogram }
  ElsoCsop := Nil ;
  UtolsoCsop := Nil ;
  Bevitel ;
  TeljesLista ;
  CsoportLista ;
  HallgLista ;
  CsoportTorles ;
End.
```

Kérdések

1. Mit jelentenek a következő fogalmak?
   a) Lista, kapcsolt lista, láncolt lista, egyirányú lista, nyíltvégű lista
   b) Statikus lista, dinamikus lista
   c) Rendezett lista
   d) Kétirányú lista, fejelt lista, cirkuláris lista
   e) Multilista
2. Hogyan keres meg egy elemet a különböző listákban? Adja meg mindegyikre az algoritmust (rendezetlen, rendezett, kétirányú, fejelt, cirkuláris, illetve multilistára)!
3. Hogyan dolgozza fel sorban a különböző listák elemeit? Adja meg mindegyikre az algoritmust!
4. Hogyan fűz fel egy új elemet a különböző listák esetén? Adja meg mindegyikre az algoritmust!
5. Hogyan töröl ki egy elemet a különböző listákból? Adja meg mindegyikre az algoritmust!

Feladatok

1. Adott egy számokból álló lista, melyet az *Elso* mutató azonosít. Az utolsó elem mutatója *Nil*. Fűzze a listához a 99-es számot utolsó elemként!
2. Olvasson be folyamatosan karakterláncokat, és fűzze fel azokat egy heap-listára a bevitel sorrendjében.
   a) Írja ki a listát!
   b) Keressen meg egy nevet, írja ki, hogy van, vagy nincs! Ha több van, mind írja ki!
   c) Keressen meg egy nevet. Ha van, akkor írja ki az előtte és utána következő neveket. Ha nincs előtte, vagy utána név, akkor azokat 3 pont helyettesítse!
   d) Törölje ki a listából az összes „K" betűvel kezdődő nevet!

3*. Egy egyszerű lineáris listának csak az első mutatója ismert.
   a) Írjon függvényt, mely visszaadja
      - a lista utolsó elemének mutatóját!
      - egy adott mutató előző mutatóját a listában!
   b) Írja ki a listát visszafelé!
   c) Fordítsa meg a listát! Az eddigi utolsó elem legyen az első, és mindegyik következő mutató az eddigi előzőre mutasson!
4. Olvasson végig egy rekordokból álló típusos állományt, és fűzze fel a rekordokat egy rendezett dinamikus listára! A rendezés bármely mező szerint történhet. Nyomtassa ki a listát előre és visszafelé!

5*. Ha van a lemezen egy *Nevek.Txt* állomány, akkor fűzze fel annak sorait egy rendezett, fejelt, cirkuláris dinamikus listára!

a) Listázza ki a neveket!
b) Fűzzön a listához újabb neveket!
c) Mentse el a listát lemezre a *Nevek.Txt* állományba, hogy legközelebb megint visszaolvashassa!
d) Írja ki a leghosszabb nevet!
e) Törölje ki a 10 karakternél rövidebb neveket a listából!

6. Az előző feladatban megadott listát tartsa karban: újabb név felvitele, adott név törlése. Ne engedje meg két egyforma név felvitelét!

7. Olvasson be adatcsoportokat:

- Számlaszám
- B/K – betét vagy kivét
- Dátum
- Összeg

A bevitelnek akkor van vége, ha számlaszámnál <ENTER>-t ütnek. Tárolja ezeket az adatokat egy heap-listában. A lista számlaszám szerint legyen rendezett. Egyforma számlaszámok esetén először szerepeljenek a betétek, aztán a kivétek, s azon belül is dátum szerint legyen rendezett a lista!

a) Minden adatcsoporthoz csak következő mutatót tároljon. Listázzon előre, majd visszafelé!
b) Minden adathármashoz tároljon előző és következő mutatót. Listázzon előre és visszafelé!
c) Készítsen a számlákról egy rendezett listát nyomtatóra. A lista számlaszámonként tartalmazzon fejlécet, és betét, illetve kivét csoportonként is legyen tördelve!

8. Olvassa be folyamatosan egy iskola osztályainak névsorait, s minden gyerekhez annak tanulmányi átlagát! Az osztálynévsor végét '/' jelezze, a teljes bevitel végét pedig a '///'. Az osztályoknak egyértelmű azonosítóik vannak, mint 1a, 1b stb. Ha már több osztály számára nincs hely, akkor azt jelezzük! Végül írja ki a következőket:

a) Osztályonkénti átlagot és az iskola átlagát
b) Adott osztály névsorát
c) Adott gyerek melyik osztályba jár, és mennyi az átlaga
d) Melyik osztályba jár a legtöbb gyerek
e) Melyik osztály átlaga a legjobb
f) Találjon még ki három feladatot!

Érdemes tanulmányozni

Angster Erzsébet-Kertész László: Turbo Pascal feladatgyűjtemény I.:
Dinamikus adatszerkezetek fejezet

10. PROGRAMSZEGMENTÁLÁS, KAPCSOLAT AZ OPERÁCIÓS RENDSZERREL

E fejezet egyik témája, hogy hogyan lehet egy programot szegmentálni, vagyis darabokra szedni, illetve több részből összerakni, és hogy hogyan tudnak ezek a programrészek egymással kommunikálni. Szó lesz az egységek készítéséről és a külső programok hívásáról, valamint az átfedési technika használatáról. A másik téma az operációs rendszerrel való kapcsolatteremtés, mely kapcsán megemlítünk néhány fontosabb – a Turbo Pascal által támogatott, illetve nem támogatott – rendszerhívást.

10.1 Egységek készítése

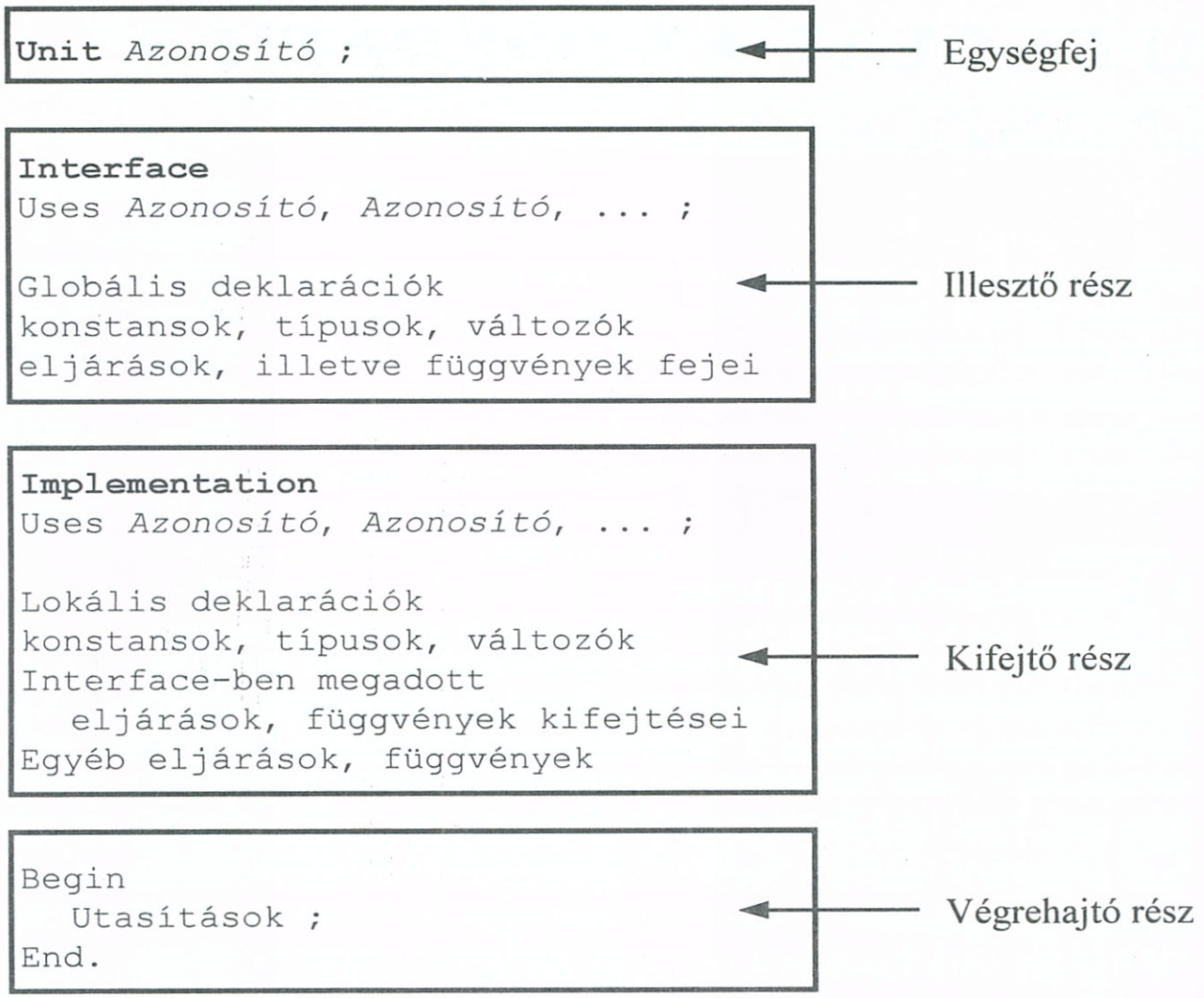
Az egység (modul, unit) a főprogramhoz hozzászerkeszthető tárgykód, mely a lemezen TPU kiterjesztéssel van nyilvántartva. Egységet használtunk már, hiszen a *ClrScr*, *GotoXY* eljárások hívásához a *Crt* egységet programunkhoz kellett szerkeszteni. A *„Memóriakezelés”* című fejezetben szó volt arról, hogy sem a főprogram, sem az egység kódja nem lehet 64 KB-nál nagyobb. Ez ugyan egy fizikai kényszer az egységek írására, de nem ez az egyetlen ok, amiért szegmentáljuk programjainkat.

Mikor írunk egységeket?

- Ha akkora a főprogramunk, hogy annak lefordított kódja meghaladná a 64 KB-ot.
- Ha akkora a főprogramunk, hogy annak forráskódja áttekinthetetlenné, sőt egyenesen kezelhetetlenné válik.
- Ha bizonyos részleteket nem is akarunk „látni”, mert zavaró. Vannak bizonyos „jó öreg” eljárások, függvények, amelyeket már régen megírtunk, és használjuk azokat. Már minden tesztelési fázison túlestek, működnek, tökéletesek. Most már csak használni szeretnénk azokat, éppen úgy, mint a *Crt* egység *ClrScr* eljárását. Mit szólna a kedves Olvasó, ha programjának áttekintésekor, kiprintelésekor minden esetben ott szerepelne a *ClrScr és társai* forráskódja?
- Ha rutinjaink csoportosításától programunk szerkezete világosabbá, áttekinthetőbbé válik.
- Bizonyos programrészek forráskódját nem akarjuk közszemlére bocsátani, mert azt a szerzői jog védi.

Természetesen az egységre bontásnak csak úgy van értelme, ha azt ésszerűen, alaposan átgondolva végezzük el. Úgy csoportosítsuk rutinjainkat, deklarációinkat, hogy azok hovatartozásán azonnal kiismerjük magunkat. Egy egységnek legyen egy jól meghatározott feladatköre, mellyel az egység használójának szolgálatában áll.

Az egység felépítése



Az egységfej az egységet azonosítja – erre az azonosítóra kell hivatkozni majd, ha az egységet használni akarjuk. A lemezre is ilyen néven kell elmentenünk, PAS kiterjesztéssel.

Az egységgel az őt használó program, illetve másik egység az illesztő (interface) részen keresztül kommunikálhat. Ez az egység „kirakata", ami ezen kívül van, az az egység magánügye, azokra a dolgokra nem lehet hivatkozni, és nem is lehet azokat megváltoztatni. Az illesztő részben megadott eljárások, függvények meghívhatók, az ott megadott konstansok, típusok használhatók, a változók lekérdezhetők, megváltoztathatók.

A kifejtő (implementation) rész az egység legnagyobb része. Nem csinál mást, mint az illesztő részt kiszolgálja. Olyan ez az egész, mint egy kis üzlet vagy cukrászda, ahol kiraknak a kirakatba krémeseket, tortákat, fagylaltokat. Az ember a készterméket

megvásárolhatja, megnézheti, megszagolhatja és megeheti. De hogy az a torta hogyan készül, azt legtöbbször homály fedi. A felhasználó megítélheti, hogy a torta tetszik vagy nem, ízlik, vagy nem. A tortát a kulisszák mögött készítik, ahhoz a vevőnek „semmi köze". Persze ha a tortának valamilyen hibája menetközben kiderül, akkor a vevő nem megy oda legközelebb. Így van ez a szoftverekkel is. Megvásárolok egy egységet, mondjuk a *Crt* egységet. Ettől kezdve használhatom annak változóit (*TextAttr, WindMin*), rutinjait (*ClrScr, GotoXY*). Ha nem tetszik, mert nehézkes a kezelése, akkor veszek egy másikat – ha kapok. Végül azok a szoftverek terjednek el, amelyek arra érdemesek – legalábbis így kellene ennek lennie. A kifejtő rész tehát a kulissza, ahol írott vagy íratlan szabályok szerint megy a munka. Persze általában mindenhol azok járnak jobban, akik valamilyen előre megadott szabály szerint, jól átgondoltan, szervezetten dolgoznak, ez nem vita tárgya. Az összes megírt egység tehát része a jól megtervezett programnak.

A végrehajtó (inicializáló) rész egyetlen egyszer fog lefutni akkor, amikor a főprogramot futtatjuk. Ide olyan utasításokat szokás tenni, mellyel az egység valamely részét előkészítjük (inicializáljuk), vagy bármi mást, ami az egység feladatkörébe tartozik, és egyszer szeretnénk végrehajtatni. Sem a kifejtő részt, sem a végrehajtó részt nem kötelező megadni. Előfordul, hogy egy egységnek csupán az a feladata, hogy néhány változót vagy típust definiáljon – ekkor tehát nincs mit kifejteni. Az *Implementation* kulcsszót mindig meg kell adni, a végrehajtó rész *Begin*-je az utasításokkal együtt elhagyható.

A rendszer *Printer* egységének forráskódja például lényegében így fest (a Borland cég kisbetűvel írja a kulcsszavakat):

```
unit Printer;
{$I-,S-}

interface
var
  Lst: Text;

implementation

begin
  Assign(Lst,'LPT1');
  Rewrite(Lst);
end.
```

Feladat
Készítsünk egységet, mely segítségével lejátszhatunk egy dallamot. A dallam megadása minél egyszerűbb legyen, és az egység minden technikai részletet rejtsen el!

Legjobb, ha az egységnek először a használatát tervezzük meg ugyanúgy, mint az eljárások esetében. A dolog úgy is felfogható, hogy megrendelünk egy egységet, mely a

következőket tudja... Természetesen egy egység nem tudhat mindent – a nyelvi lehetőségek és korlátok rá is vonatkoznak. Tervezzük meg tehát most is először a programot:

Az egységben definiálni kell egy *TZene* típusú változót, mely meghatározza a dallamot. A dallam inicializálása után adjuk meg sorban a hangokat, melyek mindegyike egy hangmagasságból és egy időintervallumból áll. Az előbbit a szokásos c,d,e,f,g, a,h,c betűkkel adjuk meg, az utóbbit pedig egy időegység többszöröseként. Az egység egy nagyon rövid időintervallumot jelent. Még a dallam lejátszása előtt meg lehet adni annak tempóját – ez alapértelmezésben 100 (az egység végrehajtó része állítja be), és minél gyorsabban akarjuk lejátszani, annál nagyobb értéket kell megadnunk. Mint látják, a lejátszandó népdal itt a „Rózsa, rózsa, százlevelű rózsa". Elnézést kell kérnem azoktól az Olvasóimtól, akiknek ez a téma nem a legjobb választásom, de ha eltekintünk a zenei háttértől, akkor látható, hogy a feladat pusztán a következő: fel kell fűznünk a megadott (hang, ritmus) párokat egy listára, majd kérésre végig kell szaladni rajta, és a megfelelő hangokat a megfelelő időintervallumokig a *Sound* eljárással meg kell szólaltatni. A hang betűinek megfelelő frekvenciaértékeket egy táblázatból vesszük. Ez a feladat azért nagyon szép példája az egységkészítésnek, mert a technikai részletek, programozói fogások mind az egységben maradnak, s így azok nem zavarják a dallam önfeledt készítését. Nézzük először a főprogram, majd az egység forráskódját:

```
Program Nepdal ;
Uses
   UZene ;

Var
   Rozsa: TZene ;

Begin
   Init(Rozsa) ;
   Hozza(Rozsa,'a',4) ; { la }
   Hozza(Rozsa,'c',4) ; { do }
   Hozza(Rozsa,'e',4) ; { mi }
   Hozza(Rozsa,'c',4) ; { do }
   Hozza(Rozsa,'h',6) ; { ti }
   Hozza(Rozsa,'d',2) ; { re }
   Hozza(Rozsa,'f',2) ; { fa }
   Hozza(Rozsa,'e',6) ; { mi }
   Hozza(Rozsa,'a',8) ; { la }
   Hozza(Rozsa,'a',4) ; { la }
   Hozza(Rozsa,' ',4) ; { szün }
   Tempo := 50 ;
   Lejatszik(Rozsa) ;
End.
```

```
Unit UZene ;

Interface

Type
```

{ Ezt a deklarációt sajnos ki kell raknunk a kirakatba, mert a TZene típusnak szüksége van rá. Ez az egységkészítésnek még egy gyengéje: }

```
  St3 = String[3] ;
  PHang = ^THang ;
  THang = Record
    Kov : PHang ;
    Hang: St3 ;
    Ritmus : Byte ;
  End ;
```

{ A TZene típus egy mutatót tartalmaz a hangok listájára, mely egy rendezetlen, egyirányú, fejelt cirkuláris lista. Az elemek felfűzése mindig a lista végére történik, ezért megjegyezzük a mindenkori utolsó hangot is: }

```
  TZene = Record
    Fej: PHang ;
    Utolso : PHang ;
  End ;

Var
  Tempo : Word ;

Procedure Init(Var Zene: TZene) ;
Procedure Hozza(Var Zene: TZene; Hang: St3; Ritmus: Byte) ;
Procedure Lejatszik(Zene: TZene) ;

Implementation
Uses Crt ;
```

{ Ezek itt a zenei hangok a szokásos megnevezéseikkel és a hozzájuk tartozó frekvencia értékekkel (helyhiány miatt csak 10 db): }

```
Const
  HangokSzama = 10 ;
  Hangok : Array[1..HangokSzama] Of St3 =
    ('a','h','c','d','e','f','g','a1','h1','c1') ;

  Frekv : Array[1..HangokSzama] Of Word =
    (220,247,262,294,330,349,392,440,494,523) ;
```

```
Procedure Init(Var Zene: TZene) ;
  Begin
    With Zene Do
      Begin
        New(Fej) ;
        Fej^.Kov := Fej ;
        Utolso := Fej ;
      End ;
  End ;

Procedure Hozza(Var Zene: TZene; Hang: St3; Ritmus: Byte);
  Var Uj : PHang ;
  Begin
    New(Uj) ;
    Uj^.Hang := Hang ;
    Uj^.Ritmus := Ritmus ;
    Zene.Utolso^.Kov := Uj ;
    Uj^.Kov := Zene.Fej ;
    Zene.Utolso := Uj ;
  End ;

Procedure Lejatszik(Zene: TZene) ;
  Var
    P : PHang ;
    Sorszam : Word ;
  Begin
    With Zene Do
      Begin
        P := Fej^.Kov ;
        While P <> Fej Do
          Begin
            With P^ Do
              Begin
                Sorszam := 1 ;
                While (Sorszam <= HangokSzama) And
                  (Hangok[Sorszam] <> Hang) Do
                  Inc(Sorszam) ;
                If Sorszam <= HangokSzama Then
                  Sound(Frekv[Sorszam]) ;
                Delay(10000 Div Tempo * Ritmus) ;
                NoSound ;
              End ;
            P := P^.Kov ;
          End ;
      End ;
  End ;
```

```
Begin
  Tempo := 100 ; { alapértelmezés }
  WriteLn('Most zenélni fogok!') ;
End.
```

Szabályok

- Egy programrendszerben ugyanazt az egységet több helyen is használhatjuk, ilyenkor azt a fordító csak egyszer szerkeszti programunkhoz.
- A főprogramban és az összes egységben az adatszegmens, a veremszegmens és a heap közös.
- Az egység azonosítói az egység nevével minősíthetők.
  Például *System.Str*, *UZene.THang*
  A minősítés nélküli azonosító mindig a legutóbb felsorolt egység azonosítóját jelenti.

Egység elkészítése, használata

- Megírjuk az egységet, mintha program lenne. Lemezre mentjük, kiterjesztése PAS lesz.
- Az egységet lemezre fordítjuk, kiterjesztése TPU (Turbo Pascal Unit) lesz.
- Megírjuk az egységet használó programot/egységet.
  USES után megadjuk az egység nevét.
  Egység esetén lásd előzőek.
  A Uses után megadott egység a programhoz/egységhez szerkesztődik.
- A program futtatható: Ctrl-F9.
  Lemezre fordítás esetén az EXE állomány DOS-ból is futtatható.

Egységkönyvtár (TURBO.TPL)

TPL = Turbo Pascal Library

- A TURBO indításakor automatikusan a tárba töltődik.
- Általában tartalmazza a *System, Crt, Dos* stb. egységeket.
- A könyvtárba bevihetünk TPU állományokat, illetve kivehetjük azokat a TPUMOVER programmal.
  Ha egy egység végleges, bevihetjük az egységkönyvtárba. Ekkor az egységet használó program/egység szerkesztése gyorsabb lesz, viszont az egységkönyvtár több memóriát foglal.

10.2 Program paraméterezése

Már egészen biztos, hogy több olyan programmal találkozott az Olvasó, melynek híváskor paramétert adott át. Ilyen például a *Turbo.Exe* is, melynek megadhatjuk az először betöltendő program nevét:

```
C:\BP\BIN>Turbo Nepdal
```

Adtak már paramétert valószínűleg néhány DOS parancsnak is, például:

```
Print Nepdal.Pas
Type Nepdal.Pas
Copy Nepdal.Pas D:\Mentes
```

Feladat
Az előző pontban megírt, népdalt lejátszó programunkat paraméterezzük a tempóval. Ha nem adunk meg paramétert, akkor a tempó legyen alapértelmezett! *Nepdal.Exe* hívásai például:

```
Nepdal 200
Nepdal 50
Nepdal
```

Egy programnak akárhány paraméter átadható, melyeket szóközökkel vagy TAB karakterekkel választhatunk el egymástól. Ezek a paraméterek a programból a *System* egység *ParamStr* függvényével kérdezhetők le:

ParamStr(*Index:Word*):String

Index-szel a paraméter sorszámát adjuk meg. A 0. paraméter mindig a futó program állományspecifikációja, a többi a felhasználó által futáskor megadott karakterláncok. A paraméterek számát a *ParamCount* Word típusú függvény adja meg. A nem létező paraméter mindig üres lánc.

Példánkban a programnak 0 vagy 1 paramétere lehet – ha nem adnak meg paramétert, akkor a tempót alapértelmezésnek vesszük. Ezek alapján *Nepdal* programunk így alakul:

```
Program Nepdal ;
Uses UZene ;
Var
   Rozsa : TZene ;
   Kod : Integer ;

Begin
   Init(Rozsa) ;
   If ParamStr(1) <> '' Then
      Val(ParamStr(1),Tempo,Kod) ;

   Hozza(Rozsa,'a',4) ;
   Hozza(Rozsa,'c',4) ;
   { Stb. }

   Lejatszik(Rozsa) ;
End.
```

10.3 Külső program hívása

Megtehetjük, hogy programunkból meghívunk egy másik kész, lefordított programot. Az *Exec* (Execute = végrehajt) eljárásnak ekkor paraméterként kell megadnunk a futtatandó program nevét és paramétereit:

Exec(*Programspecifikáció:String; Paramétersor:String*) ;

A programspecifikáció tartalmazhat lemezegységnevet és útvonalleírást is. Paramétersor pontosan az a karakterlánc, melyet a program operációs rendszerből való futtatásakor adnánk. Fontos tudnivalók az *Exec* használatához:

- Az *Exec* eljárást a *Dos* egység definiálja.
- A hívott program csak akkor töltődik be a tárba, ha van számára hely. Mivel alapértelmezésben minden program lefoglalja a teljes memóriát, ezért külső programot csak akkor tudunk futtatni, ha a heap méretét lejjebb vesszük, átengedve ezzel a helyet a futtatandó program számára. Lásd *„Memóriakezelés"* fejezet – A memória felosztása – A verem és heap méretének beállítása.
- Az *Exec* hívása előtt és után ajánlatos meghívni a *SwapVectors* eljárást, amely elmenti a megszakítási vektorokat. Ezzel biztosítjuk, hogy a meghívott program nem használja a hívó program megszakítási vektorait és fordítva.

Feladat
Írjunk programot, mely az előző pontokban elkészített paraméterezhető *Nepdal.Exe* programot egy másik programból futtatja!

```
Program Futtato ;
{$M 16384,2024,2024}
Uses Dos ;
Var
   Tempo: String[5] ;
Begin
   WriteLn('Milyen gyorsan játsszam?') ;
   WriteLn('Normál tempó = 100, * = Vége:') ;
   ReadLn(Tempo) ;
   While Tempo <> '*' Do
      Begin
         Swapvectors ;
         Exec('Nepdal.Exe',Tempo) ;
         Swapvectors ;
         WriteLn('Milyen gyorsan játsszam?') ;
         ReadLn(Tempo) ;
      End ;
End.
```

10.4 Overlay technika

Ha nagyon nagy a programunk és memóriagondokkal küszködünk, akkor van egy megoldás, mellyel óriási területeket nyerhetünk: ez az overlay (átfedéses, átlapolásos) technika használata, melynek lényege, hogy a programnak mindig csak az a része töltődik be a tárba, melyre hivatkozás történik. A külső program hívásával is nyerhetünk memóriát, de az sokkal rugalmatlanabb megoldás, mert ott egyrészt csak komplett programot lehet futtatni, másrészt nincs meg hozzá az a fejlett memóriagazdálkodást biztosító szoftver eszköz, mint az overlay-kezelő.

Ha az overlay technikát használjuk, akkor a teljes lefordított kód akár többszöröse lehet az elérhető memória méretének. Ha az egyszerre betöltendő programrészeket (Turbo Pascal-ban az egységeket) ügyesen választjuk meg, akkor a lemezműveleteket is minimálisra lehet csökkenteni.

A *„Memóriakezelés"* fejezetben már szó volt arról, hogy az overlay egységek a verem és a heap között elhelyezkedő overlay-puffer területére kerülnek. A puffer mérete alapértelmezésben a legnagyobb betöltendő egység méretével egyezik meg, de ez a méret nagyobbra állítható, és futás közben is növelhető (feltéve, hogy a heap üres). Az az egység, melynek eljárására vagy függvényére hivatkozás történik, bekerül a pufferbe, kilökve esetleg egy már bentlévőt.

Az Overlay-be szervezhető egységek alig különböznek a „normális" egységektől – megírásukkor csupán a következőkre kell figyelni:

- engedélyezni kell az overlay-be szervezést: $O+
- távoli hívással kell fordítani: $F+
- lemezre kell fordítani

Engedélyezzük példaként az előbbi pontokban tárgyalt *UZene* egység overlay-be szervezését:

```
Unit UZene ;
{$O+,F+}

Interface
Type
   St3 = String[3] ;
{ Ugyanaz, mint az eredeti... }
```

Vegyünk még két másik overlay-be szervezhető egységet. Legyenek például ezek a Feladatgyűjtemény I. *Alap14* és *Alap19* programjai, melyeket most átalakítunk egységekké, és az egységfej után elhelyezzük a megfelelő fordítási direktívákat:

```
Unit Alap14 ;
{$O+,F+}

Interface
Procedure Jelmozgat ;

Implementation
Uses Crt ;
Procedure Jelmozgat ;
  Const
    Jel  = ^B ;
    Ures = ' ' ;
  Begin
    { ... }
  End ;
End.


Unit Alap19 ;
{$O+,F+}

Interface
Procedure Szorzotabla ;

Implementation
Uses Crt ;
Procedure Szorzotabla ;
  Var
    I : Byte ;
  Begin
    { ... }
  End ;
End.
```

Attól még, hogy engedélyeztük az egység overlay-be szervezését, még nem kell overlay-be szervezni. A szervezést a főprogram fogja elvégezni, ahol a *Uses* kulcsszó után, még bármelyik átfedési egység előtt fel kell sorolni az *Overlay* szabványos egységet – ez tartalmazza az overlay-kezelőt. Egy egység overlay-be szervezése a *$O egységnév* fordítási direktíva hatására történik meg. Az így megadott egységeket a fordító nem szerkeszti hozzá az EXE állományhoz, hanem azokat egy külön állományba összegyűjti, és a főprogram nevén, OVR kiterjesztéssel eltárolja a lemezen. A program elején az overlay-kezelőt az *OvrInit* eljárással inicializálnunk kell!

Nézzük a főprogram kódját:

```
Program Atfedes ;
{$F+}

Uses Overlay, UZene, Alap14, Alap19 ;

{$O UZene }
{$O Alap14 }
{$O Alap19 }

Var
   Zene : TZene ;
Begin
   OvrInit('Atfedes.Ovr') ;
   Szorzotabla ;
   Jelmozgat ;
   Init(Zene) ;
   Hozza(Zene,'c',10) ;
   Tempo := 150 ;
   Lejatszik(Zene) ;
End.
```

Az *Atfedes.Pas* nevű program fordítása után tehát két állomány keletkezik:

- *Atfedes.Exe*, mely a futtatható kód állandóan memóriában levő részét tartalmazza;
- *Atfedes.Ovr*, melybe a három overlay-be szervezett egység (UZene, Alap14, Alap19) kódja kerül. Ez az állomány futás közben a lemezen (vagy kérés szerint az EMS-ben) van, az overlay-kezelő innen olvassa be a pufferbe a szükséges egységet.

Fontos, hogy a főprogramot és az összes többi egységet is távoli (FAR) hívással fordítsuk, hogy a lefordított címek ne csak szegmensen belüli relatív (NEAR) címek legyenek.

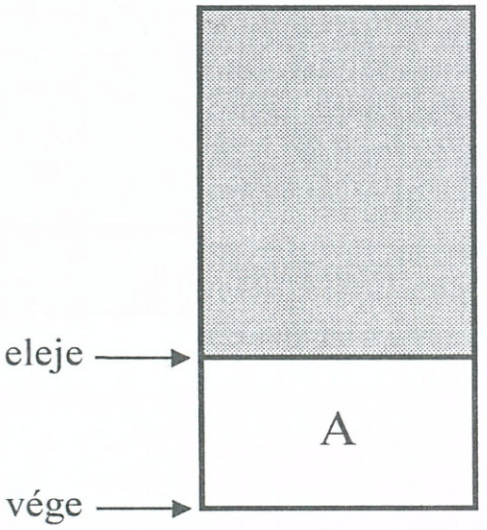
Szabályok összefoglalása

- Overlay-t csak a DOS *Real* üzemmódjában lehet használni, mert *Protected* üzemmódban a run-time manager végzi ezt a feladatot.
- A főprogramhoz hozzá kell szerkeszteni az *Overlay* egységet még bármely más, overlay-be szervezendő egység előtt.
- Csak olyan egység szervezhető overlay-be, melyet $O+ és $F+ fordítási direktívával lemezre fordítottunk.
- Az egész programot az összes egységével együtt $F+ mellett kell fordítani.
- Az overlay-be szervezett egységek nem tartalmazhatnak végrehajtó részt (tapasztalat).
- A szabványos egységek közül csak a *Dos* egység szervezhető overlay-be, a többi nem.

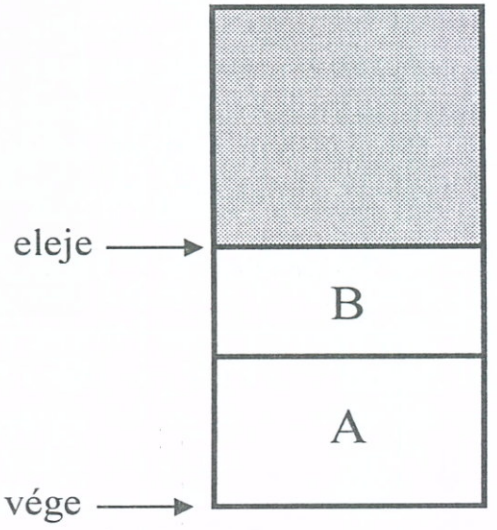
Az overlay-kezelő működése

Az overlay terület egy körkörös (cirkuláris) puffer, mely alapértelmezésben a következőképpen működik: a puffernek van két mutatója, melyek a puffer elejére, illetve végére mutatnak. Az újonnan betöltendő egységek mindig a puffer elejére kerülnek, majd ha a puffer megtelik, vagyis a következő egység számára már nincs elegendő hely, akkor a puffer végénél levő egy vagy több egység helye felszabadul:

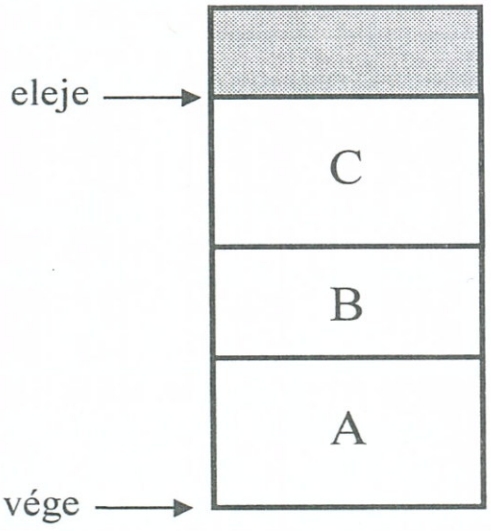
1. lépés – A befér



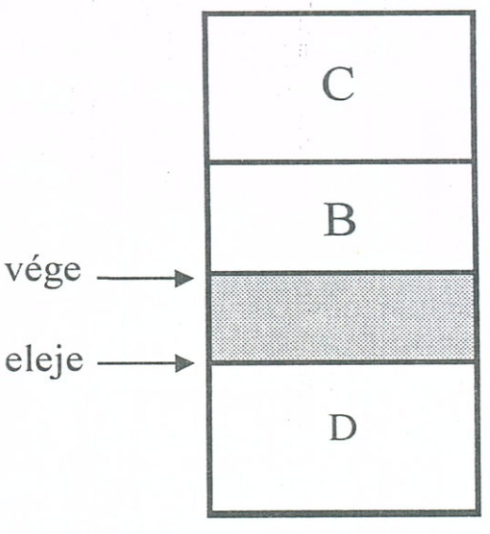
2. lépés – B befér



3. lépés – C befér



4. lépés – D nem fér be, kilöki A-t



Az overlay-kezelőnek van egy ún. optimalizálási funkciója is. Képzeljük el, hogy az *A* átfedési egységre nagyon gyakran történik hivatkozás. Ennek ellenére a fent leírt módszer értelmében az overlay-kezelő az *A* egységet ugyanúgy kilöki a pufferből, ha az kerül sorra, mint bármelyik másik egységet. Az *A* egység valószínűleg egy pillanat múlva már megint a pufferben lesz. Annak érdekében, hogy a gyakran használatos

egységek bent maradhassanak a tárban, az overley-kezelő egy ún. próbaterületet tart nyilván a puffer végén. Ha az egység erre a területre érkezik, akkor tesztelés alá kerül: ha a tesztelési idő alatt az egység bármely rutinjára hivatkozás történik, az egység a puffer elejére kerül, elkerülve ezzel a felszámolást. Ha az egység a próbát nem állja, akkor az érdemes az eltávolításra.

10.5 Megszakítások

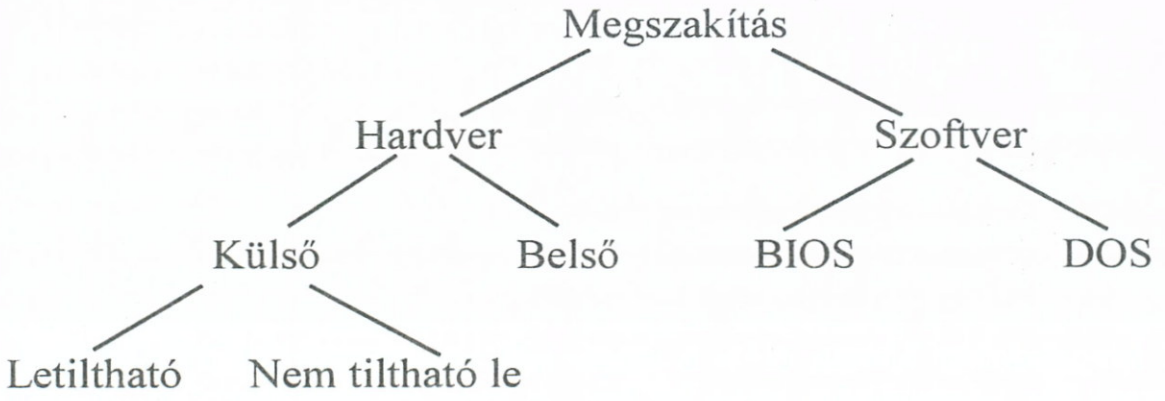
Előfordul, hogy a processzor utasítást kap a program végrehajtásának megszakítására. Ilyenkor a futást felfüggeszti, végrehajtja a kért *megszakítási rutint*, majd visszatér a megszakadt programba, és ugyanonnan folytatja a végrehajtást, ahol azt abbahagyta. A megszakítási rutinok címei a memória elején ($0000:$0000 címtől kezdődően) egy 1 KB méretű táblázatban vannak összegyűjtve. A *megszakítási vektortábla* összesen 256 darab 4 byte-os címet tartalmaz, melyek nullától kezdve sorszámozva vannak:

Megsz. száma	*Memóriacím*	*Memória tartalom (minta)*	*Rutin címe*	*Megjegyzés*
$00	$0:0000	0C 01 49 13	$1349:010C	Osztás 0-val
$01	$0:0004	3A 26 50 10	$1050:263A	Lépés...
$02	$0:0008	12 00 F4 CC	$CCF4:0012	NMI
$03	$0:000C	43 26 50 10	$1050:2643	Töréspont
$04	$0:0010	31 02 70 00	$0070:0231	Túlcsord.
$05	$0:0014	54 FF 00 F0	$F000:FF54	Hardcopy
...				
$10	$0:0040	A1 28 50 10	$1050:28A1	Kurzor
...				
$16	$0:0058	2E E8 00 F0	$F000:E82E	Billentyűzet
$17	$0:005C	D2 EF 00 F0	$F000:EFD2	Nyomtató
...				

A 0. megszakítás címe a memória 0-3. byte-jain, az 1. megszakítás a 4-7. byte-jain stb. helyezkedik el. Példánkban az 5. megszakítás címe a $14 (dec. 20) címen található, mely $F000:$FF54 (a címek tárolása mindig fordítva történik). Az 5-ös megszakítási rutin tehát valahol a memória 640 KB-os részének a végén van, az ún. ROM BIOS területen. A legtöbb megszakítási rutin a géphez tartozik, az alaplapba be van „égetve" – a számítógép ezek nélkül nem képes működni. Minden rutinnak megvan a jól meghatározott feladata: van, amelyik a billentyűzet olvasását végzi el, egy másik a nyomtatóra küld ki karaktereket. Ezek a rutinok gépenként különböznek, hiszen a számítógépeket állandóan fejlesztik, az adott funkciót elvégző rutin kódja más és más.

A megszakítási tábla használata a gépek közötti hordozhatóságot valósítja meg, hiszen megállapodás szerint a billentyűzet olvasását például minden esetben a $16-os (dec. 22) megszakítás végzi el. A BIOS megszakítások a számítógép sajátjai. Van azonban az adott operációs rendszertől függő rutinkészlet is, melyeket a $21 megszakításon keresztül alfunkciók hívásával érhetünk el. Ezek a DOS rutinok az operációs rendszer betöltésével kerülnek be a tárba.

A megszakítási rutinok közönséges eljárások, végrehajtásuk után a megszakítást kérő program visszakapja a vezérlést. A megszakítást kiváltó ok viszont többféle lehet. A megszakítás történhet a program akaratából, ezeket a megszakításokat *szoftver megszakítás*oknak nevezzük. Ilyenkor a programozó egyszerűen meghív egy megszakítási rutint ugyanúgy, mintha azt ő írta volna. A *hardver megszakítás*t – mint neve is sugallja, – a számítógép valamely hardver egysége váltja ki. A különböző periferiális egységek a program futása közben is jelt adhatnak magukról, és kérhetnek egy megszakítást. Biztos megfigyelte már az Olvasó, hogy a billentyűzet ütögetésével a puffer a program futása közben is telik, sőt előbb vagy utóbb betelik, melyet egy jellegzetes hang kísér. A billentyűzet minden egyes billentyű leütésekor „kéri" a megszakításvezérlő egységet: „Figyelj, leütötték egy billentyűmet, kérem a billentyű lekezelését!". És a megszakításvezérlő beosztja a feladatot, és amint lehet, elvégzi azt. Meghívja a $16-os rutint, melyet a billentyűzet lekezelésére írtak meg az adott számítógépre úgy, hogy az betegye a leütött billentyűt a billentyűzetpufferbe. A hardver megszakítások lehetnek *külső* és *belső megszakítás*ok. Míg a külső megszakítást egy külső egység, addig a belső megszakítást valamely segédprocesszor váltja ki rendhagyó működés esetén. Belső megszakítás például a nullával való osztás is. Külső megszakítások esetén megkülönböztetünk *letiltható* (maszkolható) és *nem letiltható* megszakításokat. Megtehetjük például, hogy a billentyűzet megszakítását letiltjuk, melynek az lesz a következménye, hogy a megszakításvezérlő ezt a megszakítást mindaddig nem hívja meg, amíg a tiltást fel nem oldjuk. A program a letiltás ideje alatt nem vesz tudomást a billentyűzet létezéséről. Nem tiltható le például a memória hibája miatt meghívandó 2-es megszakítás. A megszakítások osztályozását a következő ábra mutatja:



Futás közben egy program rengeteg megszakítási rutint hív meg, hiszen egyébként nem kommunikálhatna egyetlen periferiális egységgel sem. A magasszintű programnyelvek utasításaiba ezek a megszakításhívások be vannak építve (a ReadLn eljárás például nagyon sokszor hívja meg a $16-os megszakítást). Előfordul azonban, hogy a

programozó olyan funkciót szeretne használni, melyet az adott programozási nyelv nem kezel le. Ilyenkor „kénytelen" meghívni az adott megszakítást. A Borland Pascalban például ilyenek a következő funkciók:

- *Nyomtató figyelése:* Ha a nyomtató nincs bekapcsolva és arra írni akarunk, akkor a program futási hibával leáll. Ezt elkerülendő, a nyomtatót még kiírás előtt ellenőrizni kell. Az *IOResult* függvény nem alkalmas erre a célra, mert az a nyomtatót hosszú ideig teszteli, hogy rendben van-e már, és addig nem adja vissza a programnak a vezérlést. A legtöbb felhasználó ilyenkor idegesen kikapcsolja a gépet, látva, hogy bármit leüt, nem történik semmi. A megoldás itt is a megfelelő megszakítás hívása, mely pillanatok alatt visszaadja a nyomtató állapotát.
- *Kurzor figyelése, módosítása:* eltüntetése, megjelenítése, pozíciójának lekérdezése, beállítása, nagyságának változtatása stb.
- *Egér figyelése, módosítása:* eltüntetése, megjelenítése, pozíciójának lekérdezése, beállítása stb.

Természetesen elképzelhető, hogy az adott magasszintű nyelv egy legközelebbi verzió kibocsátásával újabb funkciókat épít be a nyelvi elemek közé, de amíg ez nem történik meg, addig a programozónak kell megírnia a számára szükséges elemeket. Egy dolgot azonban minden programozási nyelvnek biztosítania kell: a megszakítások hívására meg kell adni egy biztonságos eljárást. A Borland Pascalban a megszakításokat az *Intr* (interrupt = megszakítás) eljárással hívhatjuk meg:

Intr(*Megszakítás száma: Byte;Var Regiszterek: Registers)* ;

Az *Intr* eljárást és az ehhez szükséges *Registers* típust a *Dos* egység definiálja:

```
Registers = record
  case Integer of
    0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Word);
    1: (AL,AH,BL,BH,CL,CH,DL,DH: Byte);
  end;
```

Látható, hogy *Registers* egy változó rekord típus, mely a processzor regisztereit térképezi fel. Az első felosztás szavankénti hivatkozást tesz lehetővé, míg a második felosztás segítségével az első 4 regiszterre byte-onként is hivatkozhatunk (felső, alsó byte). A megszakítások a processzor regisztereit használják az input, illetve output adatokra. A megszakítás hívása előtt a megfelelő regisztereknek értéket kell adni, majd a megszakítás végrehajtása után itt kapjuk meg az eredményt is. A nyomtató ellenőrzéséhez például a $17-es megszakítást kell meghívni, de előtte az *A regiszter* felső byte-jába be kell tölteni az alfunkció számát (2=nyomtató státuszának lekérdezése), a *D regiszter*ben pedig a nyomtató számát kell megadnunk (0=LPT1). A megszakítás az *A regiszter* felső byte-jába teszi le az állapotot:

- 0. bit: idő lejárt
- 1-2. bit: nem használatos
- 3. bit: I/O hiba
- 4. bit On line
- 5. bit: kifogyott a papír

Ez azt jelenti, hogy a nyomtató akkor OK, ha a 4. bit 1-es, a többi szóba jövő bit pedig 0. Ez akkor igaz, ha *AH And $39 = $10*:

```
Program NyomtatoEllenorzes ;
Uses Dos ;

Var
  Reg : Registers ;

Begin
  With Reg Do
    Begin
      AH := 2 ;
      DX := 0 ;
      Intr($17,Reg) ;
      If AH And $39 <> $10 Then
        WriteLn('Nyomtató hiba!') ;
    End ;
End.
```

Kényelmi okokból a $21-es megszakításokra (DOS megszakítások) a Borland Pascal egy külön eljárást definiált:

MsDos(*Var Regiszterek: Registers*) ;

Az *MsDos(Regiszterek)* eljárás egyenértékű az *Intr($21,Regiszterek)* eljárással. A DOS megszakításnak rengeteg alfunkciója van, a $2C (dec. 44) alfunkció például az aktuális időt „mondja meg". Az *A regiszter* felső byte-jába tesszük az alfunkció számát (AH:=$2C, vagy AX:=$2C00), és a funkció hívása után az időadatokat a *C*, illetve *D* regiszterekben találjuk meg:

```
Program MsDosHivas ;
Uses Dos ;

Var
  Reg : Registers ;
  Ora,
  Perc,
  Mp,
  Mp100 : Word ;
```

```
Begin
  With Reg Do
    Begin
      AX := $2C00 ;
      MsDos(Reg) ;
      Ora := CH ;
      Perc := CL ;
      Mp := DH ;
      Mp100 := DL ;
    End ;
  WriteLn(Ora:5,Perc:5,Mp:5,Mp100:5) ;
End.
```

A megszakítási vektortábla átírható, vagyis a címek átirányíthatók más, nem szabványos, „saját" rutinokra. Ehhez azonban különösen nagy körültekintés szükséges. Az átírást és az esetleges visszaállítását a *GetIntVec*, illetve *SetIntVec* eljárásokkal végezhetjük el. A felhasználó által írt rutinokat az *Interrupt* fordítási direktívával meg kell különböztetni a többi rutintól. Az 5-ös megszakítás normális esetben kinyomtatja az aktuális képernyő tartalmát – ez minden olyan alkalommal meghívásra kerül, ha leütik a *Shift-Print Screen* billentyűt. A következő program egy időre átirányítja e megszakítási rutin címét a *Kiir* eljárásra, s ekkor nyomtatás helyett egy felkiáltójel fog megjelenni a képernyőn. Kisvártatva – a program második felében, – helyreáll a rend.

```
Program Atiranyit ;

Uses Dos ;

Procedure Kiir ; Interrupt ;
  Begin
    Write('!') ;
  End ;

Var
  Cim : Pointer ;
  I : LongInt ;

Begin
  GetIntVec(5,Cim) ;
  SetIntVec(5,@Kiir) ;
  For I := 1 To 10000000 Do ;

  SetIntVec(5,Cim) ;
  For I := 1 To 10000000 Do ;
End.
```

10.6 Rendszerszolgáltatások

Bizonyos megszakítások meghívására a nyelv magasszintű eljárást definiál, elősegítve annak rövidebb, egyszerűbb használatát. Ilyenek például a következők, melyeket többnyire a *Dos* egység deklarál[1]:

Dátum, idő lekérdezése

Sokszor előfordul, hogy programunkban a mai dátumra van szükség, vagy például időt kell mérni. Ezekre az igényekre írták meg a dátumot és időt lekérdező függvényeket:

GetDate(Var Év, Hónap, Nap, Hétnapja: Word) ;

GetTime(Var Óra, Perc, Mp, Mp100: Word) ;

Az eljárás hívása után a megfelelő változókba az aktuális dátum, illetve idő adatai kerülnek. A *SetDate*, illetve *SetTime* fordítva működik, a változókban megadott adatok alapján az aktuális dátumot illetve időt beállítja.

Lemez kapacitásának, szabad területének lekérdezése

Nagyon kényelmes dolog a rendelkezésre álló lemezterületet programból lekérdezni. A teljes kapacitást, illetve a rendelkezésre álló szabad lemezterületet a következő függvények adják meg:

DiskSize(Meghajtó száma: Word): LongInt

DiskFree(Meghajtó száma: Word): LongInt

Meghajtó száma: 0=aktuális; 1=A; 2=B; stb.

Állománykezelés

Az egyes állományok attribútumai lekérdezhetők, illetve beállíthatók *Dos* egység *GetFAttr* illetve a *SetFAttr* eljárásaival:

GetFAttr(Var F; Var Attr: Word) ;

SetFAttr(Var F; Attr: Word) ;

F egy logikai állománynév, melyet előzőleg egy fizikai állományhoz kell rendelnünk. Az állomány nem lehet nyitva.

Állomány törlésére az *Erase(F)*, átnevezésére pedig a *Rename(F)* eljárás használható, ahol *F* a logikai állomány neve. Az állomány egyik esetben sem lehet nyitva. Ezek az eljárások a *System* egységben találhatók.

---

[1]Az adott nyelvi támogatásokat részletesen a referenciakönyvben találhatja meg az Olvasó.

Könyvtárkezelés

Feladatunkhoz sokszor tartozik hozzá a lemezen való keresgélés. Meg kell adnunk például a lehetőséget a felhasználó számára, hogy a lemezen található állományokból válasszon, s programunk majd a kiválasztott állományt tölti be, s azon fog dolgozni. Az is előfordul, hogy katalógust szeretnénk váltani.

Könyvtárak közötti mozgásokra a Borland Pascal a következő eljárásokat szolgáltatja (*System* egység):

ChDir(Katalógus: String) ;

Katalógusváltás.

MkDir(Katalógus: String) ;

Katalógus létrehozása.

RmDir(Katalógus: String) ;

Katalógus megszüntetése.

GetDir(Meghajtó száma; Var Katalógus: String) ;

Aktuális katalógus lekérdezése. Meghajtó száma: 0=aktuális; 1=A; 2=B; stb.

A katalógus állományainak végigolvasása pedig a következőképpen lehetséges: a *Dos* egység definiál egy ún. keresőrekordot:

```
SearchRec = Record
   Fill: Array[1..21] of Byte;
   Attr: Byte;
   Time: Longint;
   Size: Longint;
   Name: String[12];
End;
```

A *FindFirst* eljárás arra való, hogy a megadott katalógus első, adott tulajdonságú állományát megállapítsuk, és annak bejegyzését beolvassuk:

FindFirst(Útvonal: String; Attribútum: Word; Var Keresőrekord: SearchRec) ;

*Útvonal*ban egy állománycsoportot adhatunk meg, például **.Pas*. *Attribútum* az állomány attribútumát definiálja, mely lehet csak olvasható, rejtett, rendszer állomány, kötetcímke, katalógus vagy archivált. Az eljárás a keresőrekordban elhelyezi a katalógusban talált első ilyen állomány adatait: attribútumát, létrehozási idejét, méretét és nevét. A név itt csak a szűkebb értelemben vett név, mely útvonalat nem tartalmaz, csak kiterjesztést. A *Fill* mezőt a programozó nem használhatja. A *FindNext(Keresőrekord)* a következő (a *FindFirst*-ben megadott tulajdonságú) állomány adatait olvassa be a katalógusból. A *FindNext* előtt mindig kötelező a *FindFirst*

használata. Sajnos ezek az eljárások nem csak a megadott attribútumú állományokat, hanem az összes normál (Attr=0) és archív (Attr=$20) állományt is visszaadja. A *Dos* egység legtöbb eljárásának sikeressége a *DosError* változóval lekérdezhető. Ha *DosError* 0, akkor minden rendben van. A *FindFirst* és *FindNext* esetében *DosError* értéke 18, ha nincs több ilyen állomány a lemezen.

Feladat
Listázzuk ki az aktuális könyvtár összes *Exe* kiterjesztésű állományát!

```
Program KatalogusOlvas ;
Uses Dos ;

Var
   Bejegyzes : SearchRec ;

Begin
   FindFirst('*.Exe',AnyFile,Bejegyzes) ;
   If DosError <> 0 Then
      WriteLn('Nincs ilyen állomány, vagy lemezhiba!')
   Else
      Begin
         While DosError = 0 Do
            Begin
               WriteLn(Bejegyzes.Name) ;
               FindNext(Bejegyzes) ;
            End ;
      End ;
End.
```

Feladat
Listázzuk ki a paraméterként megadott könyvtár alá tartozó összes alkönyvtárat!

A feladatot rekurzióval oldjuk meg, hiszen a könyvtárak kilistázásának algoritmusai megegyeznek. Miután kiírjuk a könyvtár nevét, végigolvassuk a könyvtárat, s ha abban alkönyvtárat találunk (attribútuma *Directory*), akkor arra meg kell hívni a könyvtárat kilistázó eljárást, vagyis önmagát. Ha már nincs több alkönyvtár, akkor az előzőleg félbehagyott eljárások sorban befejeződhetnek. Az egy, illetve két ponttal jelölt alkönyvtárakat nem szabad figyelembe venni.

```
Program KatList ;

Uses
   Crt, Dos ;
```

```
Procedure KonyvtarLista(Konyvtar: DirStr) ;
  Var
    Bejegyzes : SearchRec ;

  Begin
    WriteLn(Konyvtar) ;

    FindFirst(Konyvtar + '\*.*',Directory,Bejegyzes) ;
    While DosError = 0 Do
      Begin
        If WhereY > 23 Then
          Begin
            ReadLn ;
            ClrScr ;
          End ;

        If (Bejegyzes.Attr And Directory = Directory)
        And (Bejegyzes.Name[1] <> '.') Then
          KonyvtarLista(Konyvtar+'\'+ Bejegyzes.Name);
        FindNext(Bejegyzes) ;
      End ;
  End ;

Var
  Konyvtar : DirStr ;

Begin
  ClrScr ;
  Write('Könyvtárnév: ') ;
  ReadLn(Konyvtar) ;
  If Konyvtar[Length(Konyvtar)] = '\' Then
    Delete(Konyvtar,Length(Konyvtar),1) ;
  KonyvtarLista(Konyvtar) ;
End.
```

Kérdések

1. Mit jelentenek a következő fogalmak?
   a) Egység, modul
   b) Egységfej, illesztő rész, kifejtő rész, végrehajtó rész
   c) Egységkönyvtár
   d) Program paramétere
   e) Külső program
   f) Overlay technika, overlay-puffer, overlay-kezelő

g) Megszakítás, hardver megszakítás, szoftver megszakítás
h) Külső megszakítás, belső megszakítás
i) Letiltható megszakítás, nem letiltható megszakítás
j) Rendszerszolgáltatás

2. Hogyan kell egy egységet elkészíteni? Milyen szempontokat kell figyelembe venni?
3. Hogyan lehet a programnak paramétert adni, és azt a programból levizsgálni?
4. Hogyan lehet programból egy külső programot meghívni?
5. Soroljon fel rendszerszolgáltatásokat, melyeket a Borland Pascal támogat!
6. Hogyan tud programból egy állományt megkeresni a lemezen?

Feladatok

1. Írjon egy *Rutinok* nevű egységet a következőképpen:
   a) Az egység tartalmazzon egy eljárást, mely egy valós számot tud beolvasni a képernyő egy adott pozíciójáról, adott hosszon, adott számú tizedesjegyekkel. A beolvasás végén igazítsa a számot a mezőben a tizedespontra! Az eljárás ne tudjon futási hibával leállni, és működése felhasználóbarát legyen!
   b) Az egység tartalmazzon egy *Str* függvényt (mert nem tetszik, hogy azt eljárásként írták meg a System egységben)!
   c) Az egység inicializáló részében készítsen egy címlapot, mely majd a program elején lesz látható, és kiírja a program készítőjének nevét piros alapon fekete betűkkel. Az inicializáló rész végén ne felejtse el a szöveges attribútumot az eredetire visszaállítani!
   d) Használja programjában a fent definiált egységet, pl. olvasson be a képernyő közepéről valós számokat a fenti beolvasó eljárással 0 végjelig, majd jelenítse meg az összes számot és azok összegét egyszerre $a+b+c+... = s$ alakban!

2*. Írjon egy egységet, mely segítséget ad a matematikusoknak, fizikusoknak, mérnököknek a komplex számokkal való számoláshoz! Egy komplex szám $v+k*i$ alakú, ahol v a valós, k pedig a képzetes rész. A komplex számot a (v,k) valós számpár meghatározza.

   Két komplex szám összege: (v1,k1) + (v2,k2) = (v1+v2,k1+k2)
   Két komplex szám különbsége: (v1,k1) - (v2,k2) = (v1-v2,k1-k2)
   Két komplex szám szorzata: (v1,k1) * (v2,k2) = (v1*v2-k1*k2,v1*k2-k1*v2)
   Stb.

3. Írjon egy programot, mely a paraméterként megadott valós és képzetes részekkel megadott komplex szám négyzetét kiírja! A program hívása tehát:

   Sqr v k

   A feladathoz használja az előbb megírt egységet!

4. Szervezze az 1. és 2. pontban elkészített egységeket overlay-be!
5. Válasszon ki öt darab programot, mely a lemezen *Exe* formában szerepel. Menüből választhatóan futtassa ezeket a programokat!
6. Állítsa be a gép óráját a saját órája szerinti időre, és írja ki másodpercenként az aktuális időt. Végül egy billentyű leütésére állítsa vissza az eredeti órát!
7*. Mennyi idő alatt tud a gép elvégezni 1 millió szorzást?
8. Addig bővítsen egy állományt, amíg van hely a lemezen!
9. A parancssor-paraméterben megadott, illetve az aktuális alkönyvtárból válogassa ki azokat az állományokat, melyekben megtalálható a 'PCK' karakterlánc!
10. Az aktuális könyvtár leghosszabb állományát rejtse el!
11. Írjon programot, mely sorban felkínálja az aktuális könyvtár állományait és megkérdezi, hogy törölje-e azt. Igen válasz esetén törölje az állományt. Végül készítsen egy könyvtárlistát!
12. Írja ki a parancsor-paraméterben megadott lemezegység összes rejtett állományát az azokat tartalmazó könyvtárak nevével együtt!
13. Melyik a három legterjedelmesebb könyvtár a lemezen és mekkorák? Állapítsa meg ezt egy program!

Érdemes tanulmányozni

Angster Erzsébet-Kertész László: Turbo Pascal feladatgyűjtemény II.:
Rendszerközeli programozás, Overlay technika és Egérkezelés fejezetek

11. ADATSZERKEZETEK

Ebben a fejezetben olyan klasszikus adatmodellekről és azok viselkedéséről lesz szó, melyek a számítástechnikában alapvető szerepet játszanak, és melyek ismeretét és helyes alkalmazását egyetlen programozó sem nélkülözheti.

11.1 Adatmodell, eljárásmodell

A világ bonyolult – minél jobban megfigyelünk egy dolgot, annál bonyolultabbnak látjuk azt. Minthogy a komplexitás mélysége gyakorlatilag végtelen, az ember gondolkodása, érdeklődése pedig véges, a lebontást egy bizonyos szinten abba kell hagynunk. Hogy hol hagyjuk abba, és mit veszünk figyelembe, az attól függ, hogy számunkra mi a fontos: azokat a részleteket, amelyek a vizsgálandó rendszer szempontjából érdektelenek, egyszerűen elhagyjuk.

A modellezés lényege, hogy a valós világ bizonyos jegyeit kiragadva egy olyan rendszert állítunk fel, mely kinézetét, illetve működését tekintve hasonlít a valós világhoz. Lényegében két fajta modell létezik: *fizikai modell* és *absztrakt modell*. Egy fizikai modell a vizsgálandó tárgy kicsinyített és leegyszerűsített mása, mint például egy tervezés alatt álló városrész makettje. Az absztrakt modell a valós világ elemeit absztrakt módon, például ábrákkal, adatokkal írja le. Absztrakt modell például Magyarország autótérképe.

A szoftverrel is a valós világot modellezzük. A szoftver-fejlesztés eredménye – a programrendszer – egy olyan absztrakt modell, melyet számítógéppel működtetünk. A modellt alkotó objektumokat adatokkal azonosítjuk, melyek a modell működése során állandóan változnak. Egy számítógépes rendszer alapvetően kétféle modellből áll: adatmodellből és eljárásmodellből, melyek azonban nem választhatók el egymástól, mindkettő a rendszer szerves része. Az adatmodell a rendszer elemeinek adatokkal kifejezhető tulajdonságait írja le, és felállítja az elemek közötti kapcsolatokat is. Az eljárásmodell az adatmodellen bekövetkező változásokat írja le. Működése során a rendszer jeleket vesz fel környezetéből, melyek hatására adatmodelljén az eljárásmodellben leírt módon változtatásokat eszközöl, és jeleket továbbít a rendszer más részei, illetve a külvilág felé. Vannak elméletek, melyek az adatmodellt tartják fontosabbnak az eljárásmodellel szemben, mások az eljárásmodellt tekintik elsődlegesnek a rendszer kiépítése során. Ma már eldöntött dolog, hogy a két modell egyformán fontos, és egyiket sem lehet a másik elé helyezni. Mindkét modell a valós világ szerves része, következésképpen a valós világot csak az adatok és eljárások együttes analizálása során lehet hűen modellezni. Ezt a felismerést próbálja szem előtt tartani az objektum-

orientált programozási és rendszerfejlesztési szemlélet is, mely már meghatározója egész számítástechnikánknak.

Egy rendszer kialakításának első lépése az adat, illetve eljárásmodell elkészítése logikai szinten. Bizonyos *klasszikus adatmodellek és a rajtuk értelmezett eljárásmodellek* elméletei a számítástechnikában már régen megszülettek. Ezen adatmodellek szerkezetének és működésének szabályait leírták, a modelleket rendszerezték. A modellkészítő feladata, hogy készítendő rendszerében ezeket a modelleket felismerje, és helyesen alkalmazza – ehhez természetesen elegendő szakmai ismerettel, absztrakciós készséggel, valamint szoftverépítő tapasztalattal kell rendelkeznie.

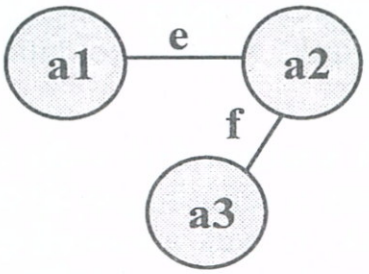
Számítógépes rendszer esetén a logikai modell fizikai megvalósítása számítógépen történik. Ezért a modellt – a hardver lehetőségeit figyelembe véve – a számítógépre le kell képezni. Az adatmodell elemei a központi tárban, illetve külső tárolóeszközökön helyezkednek el, az eljárásmodellnek gépi kódú utasítások felelnek meg.

Mindenekelőtt tisztázzuk a számítástudományban gyakran használatos jelölésrendszer, a *gráf* fogalmát:

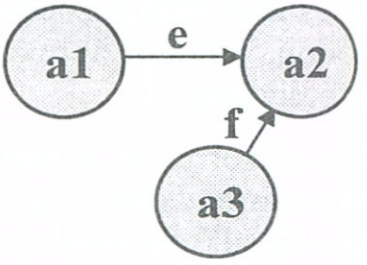
Gráf

A gráf csúcsokból és a csúcsokat összekötő élekből álló véges, nem üres halmaz. Ha az *e* él az *a1* és *a2* csúcsokat köti össze, akkor azt mondjuk, hogy *a1* és *a2* illeszkedik *e*-re, valamint *a1* és *a2* szomszédos csúcsok. Az *e* él a rendezetlen *(a1,a2)* párral is jellemezhető. A gráfot irányítottnak nevezzük, ha az élekhez irányok is tartoznak. Az irányokat nyilakkal ábrázoljuk. Az irányított gráf *e* éle a rendezett *(a1,a2)* párral jellemezhető:

Gráf:



Irányított gráf:



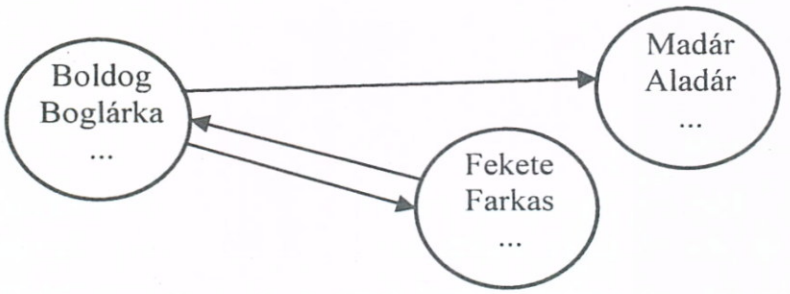
11.2 Adatszerkezetek rendszerezése

Az adatszerkezet egymással kapcsolatban álló adatok, objektumok összessége. A kapcsolatokban részt vevő struktúraelemeket csomóponti adatoknak szokás nevezni. Hogy a csomópontok pontosan mit tartalmaznak, az a modell szempontjából lényegtelen. A csomópontokat körökkel, a csomópontok közötti kapcsolatokat pedig nyilakkal reprezentáljuk.

Tekintsünk például egy hallgatói névsort. A csomópontokban a hallgatók lesznek azok összes, a rendszer szempontjából fontos tulajdonságaikkal. A hallgatók közötti kap-

csolatokat az ABC rend határozza meg. Ezek szerint *Boldog Boglárka* után *Fekete Farkas* következik, *Fekete Farkast* pedig *Madár Aladár* követi:

Ugyanezeket a hallgató objektumokat egész más szempont alapján is össze lehetne kötni, például a „szereti" kapcsolat alapján: *Fekete Farkas* szereti *Boldog Boglárkát*, *Boldog Boglárka* pedig szereti *Fekete Farkast* és *Madár Aladárt*:



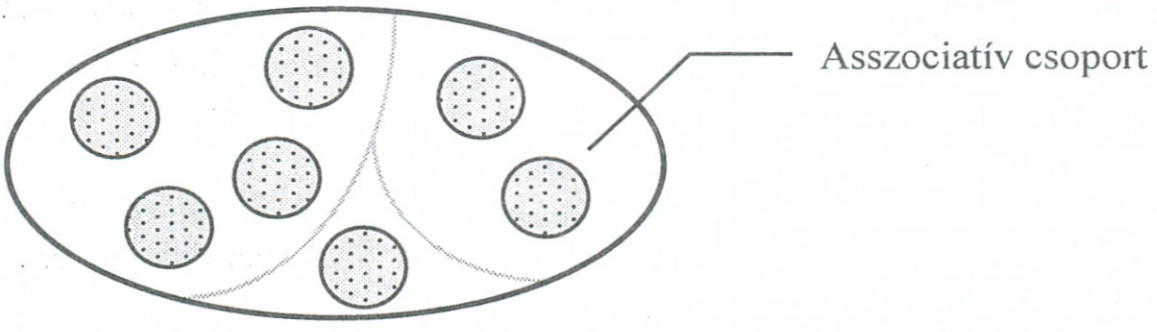
A modell működését leginkább az objektumok közötti relációk határozzák meg. A kapcsolatok alapján az adatszerkezetek az alábbi négy fő csoportba sorolhatók:

- Asszociatív adatszerkezetek
- Szekvenciális adatszerkezetek
- Hierarchikus adatszerkezetek
- Hálós adatszerkezetek

Az egyes csoportokat először röviden jellemezzük, és felsoroljuk az oda tartozó legfontosabb szerkezeteket. Az egyes adatszerkezeteket és azok számítógépes megvalósításait külön pontok tárgyalják. Az adatszerkezetek rendszerezése a 11.1. ábrán látható.

Asszociatív adatszerkezetek

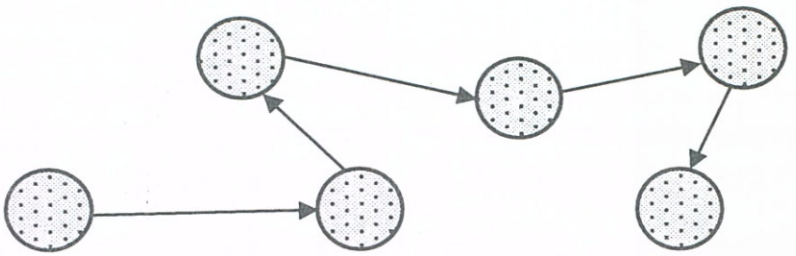
A struktúraelemek közötti kapcsolatokat az elemek azonos tulajdonságértékei létesítik. Az elemeket e tulajdonságok alapján csoportosítjuk. A kapcsolatok az asszociatív (csoportosítható) adatszerkezetekben a leglazábbak.



Asszociatív adatszerkezet memóriában a tömb, a ritka mátrix és a különböző táblák; külső tárolóeszközön pedig a direkt szervezésű állomány.

Szekvenciális adatszerkezetek

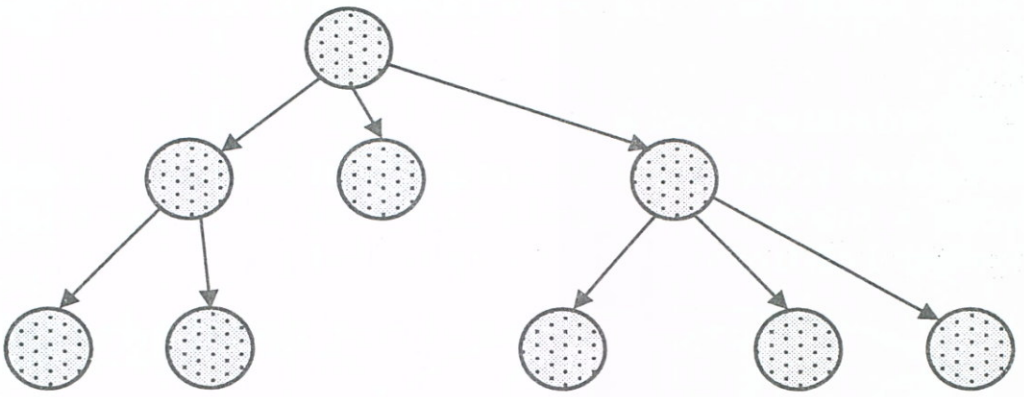
Itt az egyes struktúraelemek egymás után helyezkednek el. Mindig van egy kezdő elem, és minden elemet a struktúra egy jól meghatározott eleme követ. A kapcsolat egy-egy jellegű: minden elem csak egy helyről látható, és minden elem csak egy elemet lát. Ez egy rendezési reláció – az egyik elem a másik előtt, mögött, alatt, felett stb. helyezkedik el.



A szekvenciális adatszerkezet olyan problémák megoldására jó, ahol az elemeket valamilyen szempont szerint sorban kell feldolgozni – akár az összeset, akár azok egy folytonos részét. A memóriában megvalósítható szekvenciális adatszerkezetek a jelsorozat, a verem és a sor, külső tárolóeszközökön ilyenek a szekvenciális és láncolt adatállományok.

Hierarchikus adatszerkezetek

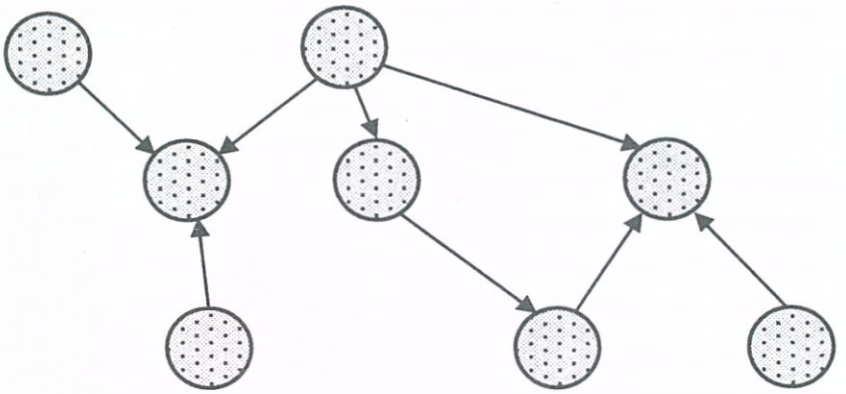
A struktúraelemek hierarchikusan egymás alá vannak rendelve. A kapcsolatok jellege egy-sok: minden csomópont csak egy helyről látható, egy csomópontból viszont sok csomópont látható:



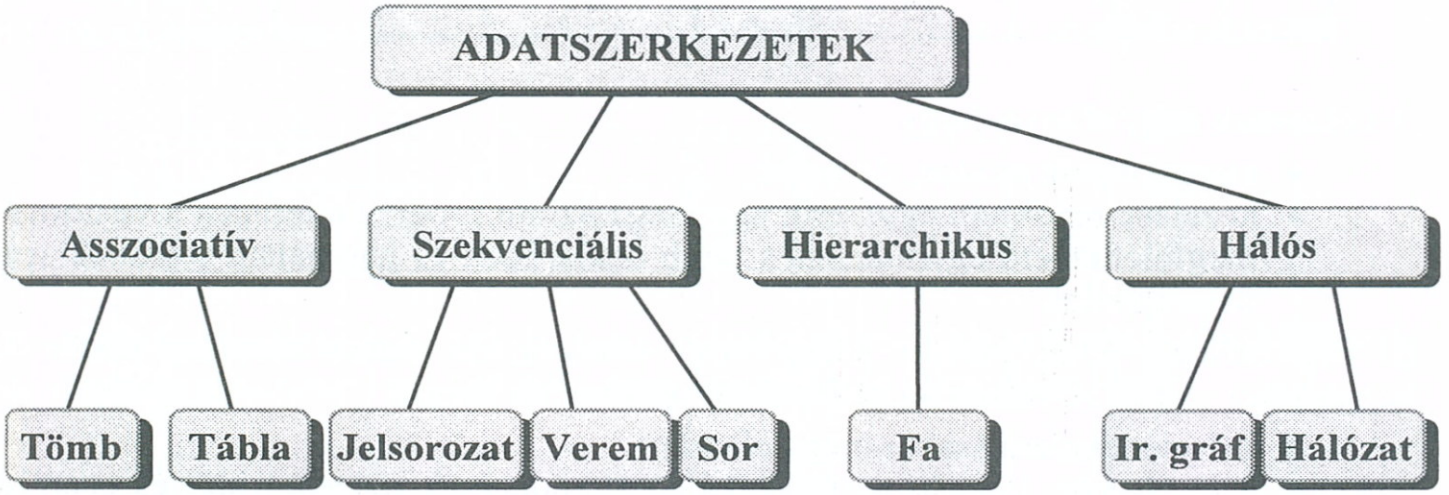
Hierarchikus adatszerkezet olyan problémák esetén alkalmazható, melyekre jellemző a tulajdonos viszony, illetve a lebontás. A hierarchikus adatszerkezeteket a belső tárban fának, a külső tárolókon hierarchiaállománynak nevezik.

Hálós adatszerkezetek

Hálós adatszerkezetek esetén bármelyik csomópont bármelyik csomóponttal kapcsolatban állhat egymással. A kapcsolatok sok-sok típusúak – ez a modell a kapcsolatkiépítés legbonyolultabb formája:



A hálós adatszerkezeteket a belső tárban irányított gráfnak, illetve hálózatnak, a külső tárolókon sémának nevezik. A hálózat abban különbözik az irányított gráftól, hogy a kapcsolatokhoz valamilyen mérőszám is tartozik.



11.1. ábra Adatszerkezetek rendszerezése

Az egyes adatszerkezetekre az elemek közti kapcsolatokon túl jellemzőek a rajtuk értelmezett műveletek halmaza. A műveleteknek lényegében két fő csoportját különböztetjük meg:

- *Konstrukciós műveletek:* Az adatszerkezetet létrehozó és továbbépítő mechanizmusok.
- *Szelekciós műveletek:* Az adatszerkezetet lebontó, megszüntető, valamint az adatelemek elérését biztosító mechanizmusok.

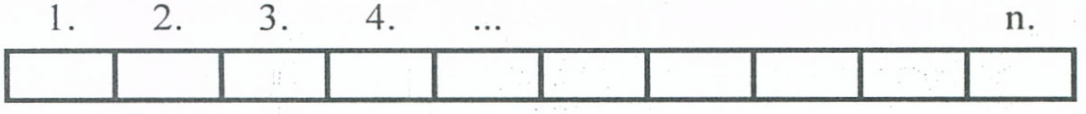
Minden adatszerkezethez a konstrukciós és szelekciós műveletek meghatározott készlete tartozik.

11.3 Absztrakt társzerkezetek

A matematikai adatszerkezetek elméleti adatszerkezetek – programozásukhoz le kell képeznünk azokat a számítógép valamely külső vagy belső tárolójára. A csomóponti adatokat el kell tudni tárolni, hivatkozni kell rájuk, és – ami a legnehezebb programozási feladat, – a kapcsolatokat is rögzíteni kell. Egy *(a,b)* reláció esetén *a*-nak ismernie kell *b*-t, illetve *a*-ból el kell tudni jutni *b*-be. Az egyes objektumok tárolása konkrét fizikai helyeken történik. Egyik objektumból egy másikra való hivatkozás csak úgy lehetséges, ha az objektumokat tartalmazó fizikai tároló címezhető. Egy nem címezhető tárolón csak szerkezet nélküli adathalmazokat tárolhatunk – ilyen esetben az egyetlen járható út a soros (fizikai sorrendben való) tárolás, illetve visszaolvasás. A memória és a lemezes egység is címezhető tárolók, melyeken két absztrakt tárolási lehetőség kínálkozik: a vektor és a lista.

Vektor

A vektor egy olyan fizikai tároló, amelyben a tárolt objektumok közvetlenül egymás után helyezkednek el, mégpedig pontosan ugyanolyan távolságra egymástól. A vektor elemei indexeléssel közvetlenül címezhetők:



A vektornak nagy előnye, hogy

- elemei direkt módon elérhetők;
- kezelése rendkívül egyszerű: egy magasszintű programnyelvben a vektornak megfeleltethető egydimenziós tömb deklarálása és használata senkinek sem okoz gondot.

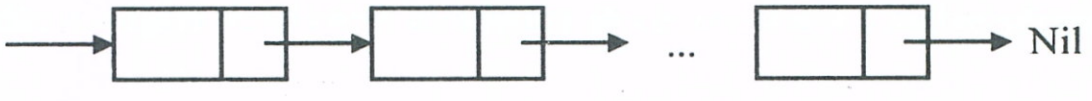
Elvileg bármely adatszerkezet leképezhető vektorra, de annak kezelése rendkívül bonyolulttá válhat. A vektor meglehetősen rugalmatlan tároló:

- méretét deklarálásakor meg kell adni;
- a tárolt adatok karbantartásával járó átrendezések rengeteg időt és energiát vesznek igénybe.

Minél inkább hasonlít a matematikai adatszerkezet az absztrakt tárolóra, annál egyszerűbb annak kezelése. A vektort, mint tárolót nagyon jól lehet alkalmazni olyan asszociatív és szekvenciális adatszerkezetek esetén, ahol az elemek mérete egyenlő, és nincs szükség túl sok átrendezésre.

Lista

A listáról már szó volt a „*Dinamikus lista*" című fejezetben. Láthattuk, hogy a lista elemei fizikailag teljesen véletlenszerűen helyezkedhetnek el a tárolóban, az elemek sorrendjét mutatókkal állítjuk fel:



A listának legnagyobb előnye abban áll, hogy átrendezés esetén nem kell óriási adatmennyiségeket mozgatni, azt néhány mutató átirányításával elintézhetjük. Létezik statikus lista is, ahol a lista elemeit mutatóikkal együtt egy vektorra képezzük le – ezzel azonban pont a tárolás rugalmasságát veszítjük el. Figyelembe kell venni a listának egy óriási korlátját is: elemei nem címezhetők közvetlenül, azok csak szekvenciálisan érhetők el. A lista alkalmazási körét ez a tény feltétlenül szűkíti.

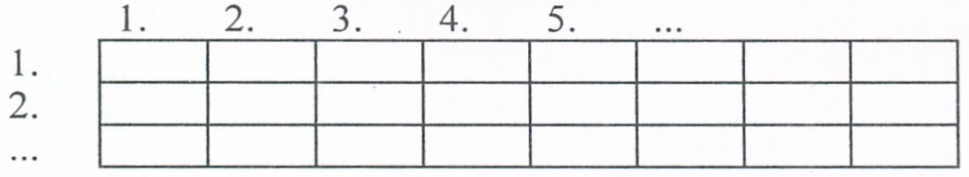
Lineáris listában elsősorban a dinamikus, szekvenciális adatszerkezeteket tárolhatjuk. A hierarchikus és hálós szerkezetek tárolására a multilisták alkalmasak. Ezekre a megfelelő adatszerkezetek tárgyalásánál térünk majd ki.

11.4 Tömb

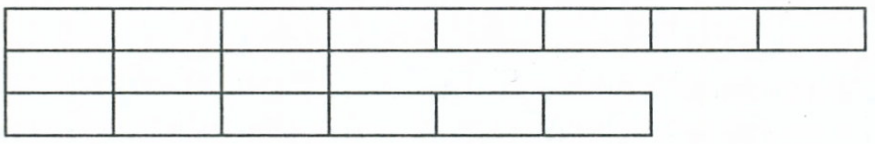
A tömb, mint adatszerkezet lehet *általános tömb* és *ritka mátrix*. Mindkettő lehet egy-, két-, illetve többdimenziós. A ritka mátrixokat azért kell külön említeni, mert azok tárolása egészen másképp történik, mint az általános tömböké.

Általános tömb

Az általános tömbökről részletesen szó volt az I. kötet *„Egydimenziós tömb"*, illetve *„Többdimenziós tömb"* fejezeteiben. A tömb lényege, hogy elemeit indexeken keresztül érjük el, mégpedig több dimenzió esetén az indexelést több „irányból" végezzük:



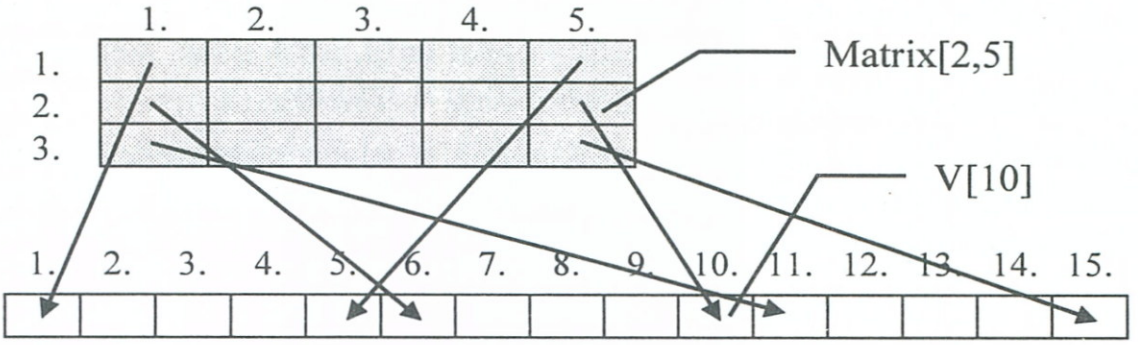
A tömbök használatának hátránya, hogy méretét előre meg kell adni, ezenkívül az indexhatárok dimenziónként kötöttek. Nem létezik tehát ilyen alakú tömb:



Az általános tömbök leképezése kivétel nélkül vektorra történik. Egydimenziós tömb esetén az adatszerkezet általában maga a tároló. Előfordul, hogy címszámítási algoritmust kell végeznünk, például akkor, ha az adatszerkezet indexhatárai a programozási nyelvben megengedett tartományhoz képest el vannak csúszva, vagy az indexek nem közvetlenül egymás mellettiek. A leképezés több dimenzió esetén is vektorra történik. Például a

```
Var
  Matrix : Array[1..3,1..5] Of Integer ;
```

esetén a leképezés egy 3*5=15 elemű V vektorra történik, melynek minden eleme *Integer* típusú. Könnyen kiszámítható, hogy Matrix[I,J] a V vektor (I-1)*5+J elemén kerül tárolásra:



A programozó saját leképezéseket is használhat tárolásra, ha úgy gondolja, nem érdemes az egész tömböt teljes egészében egy vektorban tárolni. A *szimmetrikus mátrix*nak például csak a felét használjuk – az egyre rövidülő tárolt sorok V vektorbeli indexeit egy *címszámítási algoritmus* adhatja:

	1.	2.	3.	4.	5.
1.	2	6	99	23	1
2.	0	7	344	3	29
3.	0	0	27	1	1
4.	0	0	0	9	2
5.	0	0	0	0	5

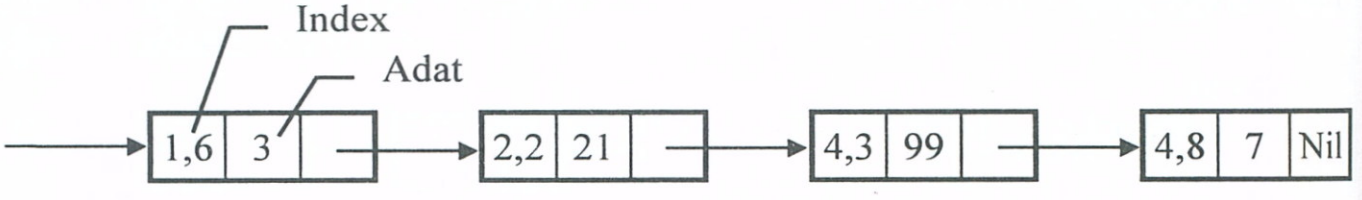
A programozó *indirekt indexelés*sel is elvégezheti a leképezést: felvehet egy indextömböt, mely elemei megadják a tárolandó tömb első sorainak kezdő indexeit a tárolt vektorban.

Ritka mátrix

Ritka mátrixnak nevezzük az olyan többdimenziós tömböt, ahol a tömb nagy része kihasználatlan:

	1.	2.	3.	4.	5.	6.	7.	8.	9.
1.	0	0	0	0	0	3	0	0	0
2.	0	21	0	0	0	0	0	0	0
3.	0	0	0	0	0	0	0	0	0
4.	0	0	99	0	0	0	0	7	0
5.	0	0	0	0	0	0	0	0	0

A ritka mátrix vektorra való leképezése rendkívül gazdaságtalan – ilyen esetekben jó tárolási eszköz a lista, melyre felfűzhetjük a „ritka adatokat”:



11.5 Jelsorozat

Egy jelekből összeállított sorozatot *jelsorozat*nak (füzér, string) nevezünk. A jelek, illetve szimbólumok bármik lehetnek, de azok a jelsorozat szempontjából oszthatatlanok. A lehetséges jelek halmazát jelhalmaznak vagy *ABC*-nek szokás nevezni. A jelsorozat bármelyik helyére bármelyik jelből választhatunk, és hossza nincs korlátozva. Jelhalmazok és azokból összeállított jelsorozatok például:

```
MagyarABC = {A,Á,B,C,D,E,É,F,G,...,T,U,Ú,Ü,Ű,V,W,X,Y,Z }
Magyarszó = BECÉZGET

Szavak = {itt, ott, is, így, úgy, persze, ne, se }
Mondat = ott is itt is persze ne is így ne is úgy
```

A jelsorozat egy tipikusan lineáris szerkezet, hiszen a jelek sorrendje jól meghatározott. Két jelsorozat akkor és csakis akkor azonos, ha azok pontosan ugyanolyan hosszúak, és jeleik rendre egyenlőek. Értelmezhető az üres jelsorozat is.

A jelsorozattal, mint adatszerkezettel kapcsolatos műveletek tipikusan a következők:

- Jelsorozat létrehozása
- Két jelsorozat összehasonlítása
- Jelsorozat hosszának meghatározása
- Két jelsorozat konkatenálása
- Részsorozat keresése egy adott jelsorozatban
- Részsorozat kimásolása egy jelsorozatból
- Részsorozat beszúrása egy adott jelsorozatba
- Részsorozat törlése

A Turbo Pascal-ban deklarált *String* típus egy speciális füzér, ahol az ABC elemei ASCII karakterek.

Jelsorozatok alkalmazásai

A jelsorozatok legnagyobb alkalmazási területe a *szövegszerkesztés*. Egy szöveg karakter jelekből áll, és olyan nagyobb, különböző struktúrájú egységekre tagolódhat, mint oldal, rész, fejezet, paragrafus, mondat stb. A szövegszerkesztőnek a jelsorozatok tárolását oly módon kell megoldania, hogy a felhasználó a lehető legrugalmasabb módon alkalmazhassa a különböző sorozatokra értelmezett műveleteket.

A jelsorozatoknak van egy másik alkalmazási területe, az ún. *formális nyelvek* elmélete. Ez a tudomány az ABC-k felett alkotható jelsorozatok halmazának részhalmazaival foglalkozik. Az ABC-t és a jelsorozatok képzésének szabályait együttesen *nyelvtan*nak nevezzük. A Pascal nyelv is egy formális nyelv.

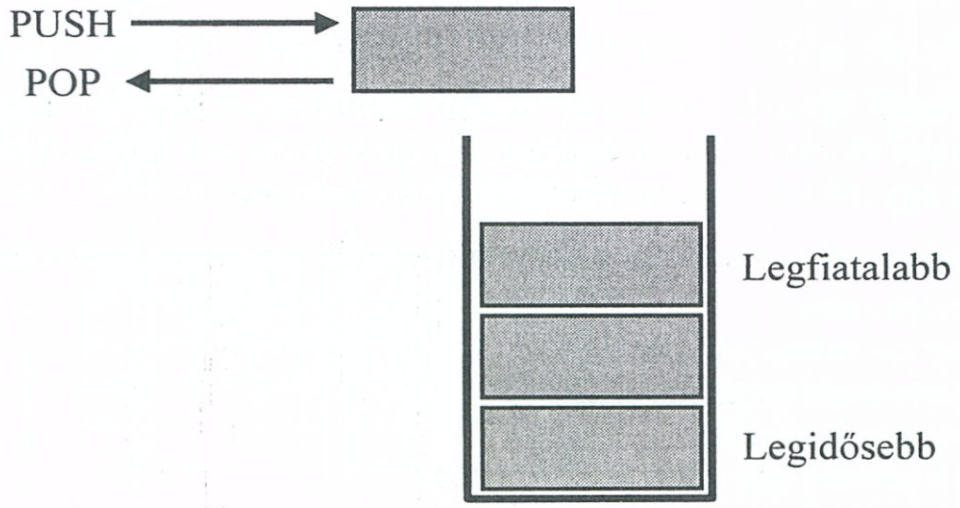
Jelsorozat tárolása

Mivel a jelsorozat egy tipikusan szekvenciális szerkezet, azt elvileg akár vektorban, akár listában tárolhatjuk. Nagyobb méretű jelsorozatok tárolására a vektor nyilvánvalóan nem alkalmas, hiszen egy részsorozat beszúrásakor az egész jeltömeget mozgatni kellene. A jelsorozatok tárolására az alkalmazási területektől függően nagyon komoly

elméletek és módszerek állnak rendelkezésre. Ennek részletezése nem e könyv feladata. A Turbo Pascal által támogatott *String* típus tárolása – mint tudjuk – vektorban történik.

11.6 Verem

A verem (stack) egy szekvenciális adatszerkezet, melynek mindig csak a legutoljára betett elemét lehet látni, illetve kivenni. A verem szokásos elnevezése még: LIFO (Last In First Out) szerkezet. A verem a következő ábrával szemléltethető:



Műveletek:

- PUSH: Elem betétele a verembe – mindig a tetejére
- POP: Elem kivétele a veremből – mindig a legfelsőt
- TOP: Legfelső elem lekérdezése – a verem változatlan marad

Verem tárolása

A megvalósítás legtöbb esetben vektor segítségével történik. Ekkor az elemszámot természetesen maximalizáljuk. Kell egy mutató, mely a mindenkori verem tetejére, illetve az első szabad helyre mutat. Egy veremről meg kell tudni állapítani, hogy az üres, vagy tele van, hiszen üres veremből nincs értelme kivenni, tele verembe pedig nincs értelme betenni elemet.

Nézzük a verem egy lehetséges megvalósításának Turbo Pascal kódját:

```
Program VeremPrg ;
Uses Crt ;
Const Max = 20 ;
Type
   TElem = Byte ;
   TVerem = Array[1..Max] Of TElem ;
```

```
Var
  Verem : TVerem ;
  Elem : TElem ;
  VM : Byte ; { Veremmutató az első szabad helyre }

Function Tele: Boolean ;
  Begin
    Tele := VM > Max ;
  End ;

Function Ures: Boolean ;
  Begin
    Ures := VM = 1 ;
  End ;

Procedure Push(X: TElem) ;
  Begin
    If Not Tele Then { Biztonsági intézkedés }
      Begin
        Verem[VM] := X ;
        Inc(VM) ;
      End ;
  End ;

Procedure Pop(Var X: TElem) ;
  Begin
    If Not Ures Then { Biztonsági intézkedés }
      Begin
        Dec(VM) ;
        X := Verem[VM] ;
      End ;
  End ;
```

A veremlista nem része a műveleteknek, csak a demonstráció kedvéért készítjük el:

```
Procedure VeremLista ;
  Var
    I : Byte ;
  Begin
    GotoXY(1,5) ;
    For I := 1 To VM-1 Do
      Write(Verem[I]:4) ;
    ClrEol ;
  End ;
```

A főprogram a verem működését demonstrálja. A verembe véletlenszerűen teszünk, illetve veszünk ki véletlen számokat. Az eljárás addig tart, amíg meg nem nyomnak egy billentyűt. Minden egyes veremművelet után a képernyőre írjuk a verem aktuális tartalmát:

```
Begin  {Főprogram, demonstráció }
  ClrScr ;
  Randomize ;
  VM := 1 ;
  Repeat
       Case Random(2) Of
         0: If Not Tele Then Push(Random(256)) ;
         1: If Not Ures Then Pop(Elem) ;
       End ;
       VeremLista ;
       Delay(500) ;
  Until KeyPressed ;
End.
```

A verem alkalmazásai

Az elemek sorrendjét legkönnyebben a verem alkalmazásával fordíthatjuk meg. Ugyanígy nagyon jól használható a verem különböző visszatérési utak megjegyzésére is. Odafelé utunkban – mielőtt tovább mennénk, – az aktuális állomás címét a veremre tesszük. Visszafelé nem kell mást tenni, mint a verem legfelső eleme szerint továbbmenni, és ezt az elemet eldobni. Menjünk le például Budapestről Pécsre egy akármilyen útvonalon:

```
Push(Budapest);
Push(Székesfehérvár);
Push(Fonyód);
Push(Kaposvár);
Push(Pécs);
```

Visszafelé a

```
Pop(Város) ;
```

ismételt végrehajtásával visszajönnek az érintett városok, és ugyanazon az útvonalon hazatalálunk. Ezt a módszert használja eljárások hívásakor programunk is a visszatérési címek meghatározásához. Egyetlen eljárás hívásakor „csak a szomszéd várost látogatjuk meg", de eljárások egymás utáni hívásaival vagy rekurzióval az „elkalandozás" hosszúra nyúlhat. Jól használható még a verem elmélete visszaléptetéses algoritmusok és veremautomaták alkalmazásánál is.

11.7 Sor

A sor (queue) egy szekvenciális adatszerkezet, melyből mindig a legelsőnek betett elemet lehet kivenni. Angol terminológiával ez egy FIFO (First In First Out) adatszerkezet:

Műveletek:

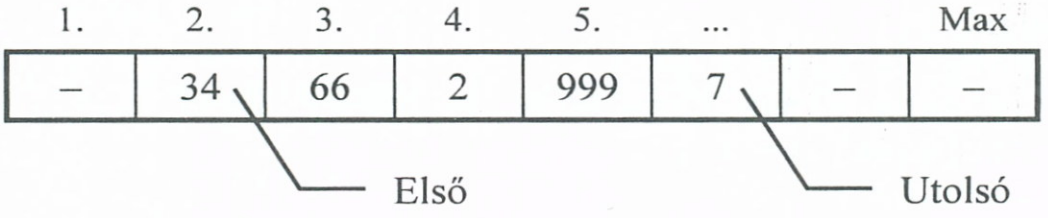
- PUT: Elem betétele a sorba – mindig a sor végére
- GET: Elem eltávolítása a sorból – mindig a sor elejéről
- FIRST: Első elem lekérdezése – a sor változatlan marad

A most definiált sor *egyszerű sor*. További definíciók:

- *Kettősvégű sor:* a sornak mindkét végéhez hozzá lehet tenni, és mindkét végéről ki lehet venni elemet.
- *Input-korlátozott sor*: a sornak csak az egyik végéhez lehet hozzátenni, de mindkét végéről ki lehet venni elemet.
- *Output-korlátozott sor*: a sornak mindkét végéhez hozzá lehet tenni, de csak az egyik végéről lehet elemet kivenni.

Sor tárolása

A sort természetesen vektorban és listában egyaránt tárolhatjuk. Nézzük a vektor esetét:



Tegyük fel, hogy a vektort elkezdjük elölről tölteni. A sor a tömb vége felé növekszik, elején pedig fogy. Ha az utolsó elem indexe elérte a maximumot, akkor kénytelenek vagyunk a tárolást a tömb elején folytatni (hacsak nem akarjuk a sor összes elemét állandóan lejjebb csúsztatni). Ezt a tárolást *ciklikus tárolás*nak nevezzük.

Nézzük most meg a sor egy lehetséges megvalósításának Turbo Pascal kódját. Az elemszámot maximalizálni kell, és valahogyan meg kell jegyezni a sor első és utolsó elemét. A sor ürességének, illetve telítettségének megállapítása okoz csak kisebb gondot: mi a helyzet akkor, ha az első és az utolsó mutató összeér? Legegyszerűbb, ha az utolsó elem mutatója helyett az elemek darabszámát jegyezzük meg. *Első* és *Darabszám* értékekből *Utolsó* számítható. A sor egy lehetséges megvalósítását a következő program mutatja:

```
Program SorPrg ;
Uses Crt ;
Const
  Max = 10 ;
Type
  TElem = Byte ;
  TSor = Array[1..Max] Of TElem ;

Var
  Sor : TSor ;
  Elem : TElem ;
  Db, Elso : Word ;

Function Tele: Boolean ;
  Begin
    Tele := Db = Max ;
  End ;

Function Ures: Boolean ;
  Begin
    Ures := Db = 0 ;
  End ;

Procedure Put(X: TElem) ;
  Var
    Hova : Word ;
  Begin
    If Not Tele Then
      Begin
        Hova := Elso + Db ;
        If Hova > Max Then Hova := Hova - Max;
        Sor[Hova] := X ;
        Inc(Db) ;
      End ;
  End ;

Procedure Get(Var X: TElem) ;
  Begin
    If Not Ures Then
      Begin
        X := Sor[Elso] ;
        Inc(Elso) ;
        If Elso > Max Then Elso := 1 ;
        Dec(Db) ;
      End ;
  End ;
```

```
Procedure SorLista ;
  Var
    M, I : Byte ;
  Begin
    GotoXY(1,5) ;
    M := Elso ;
    For I := 1 To Db Do
      Begin
        Write(Sor[M]:4) ;
        Inc(M) ;
        If M > Max Then M := 1 ;
      End ;
    ClrEol ;
  End ;

Begin  {Főprogram, demonstráció }
  ClrScr ;
  Randomize ;
  Db := 0 ;
  Elso := 1 ;

  While Not Tele Do
    Begin
      Put(Random(256)) ;
      Delay(1000) ;
      SorLista ;
    End ;

  While Not Ures Do
    Begin
      Get(Elem) ;
      Delay(1000) ;
      SorLista ;
    End ;
End.
```

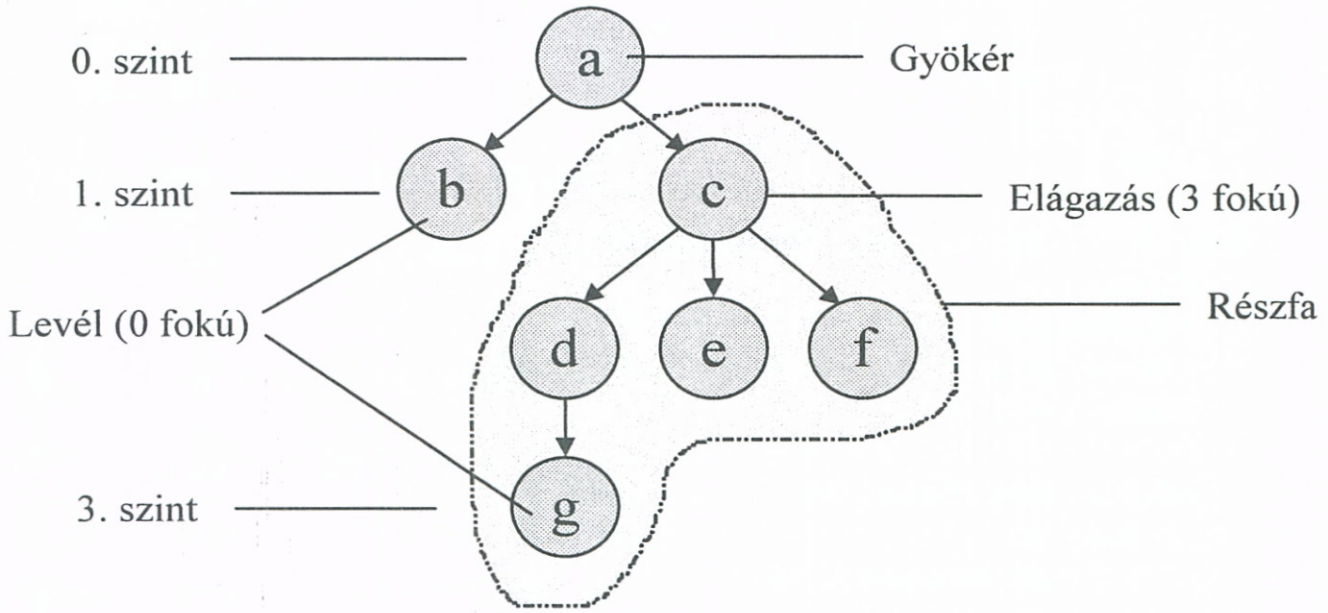
A sor alkalmazásai

Sorokkal tipikusan olyan feladatokat oldunk meg, melyekben az elemek feldolgozása érkezési sorrendben történik. Az adminisztrátor az érkező aktákat például olyan sorrendben kell, hogy elintézze, ahogyan azok beérkeztek. Sorként működnek a pufferek, mint például a billentyűzet- vagy a nyomtatópuffer. A nyomtatóra kiküldött karakterek nem állhatnak ki a sorból, először mindig az előbb kiküldött karakter kerül nyomtatásra.

11.8 Fa

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

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



Az előbbi definíció meghatározza a fát. A fának azonban van egy *rekurzív definíció*ja is. A fa vagy üres, vagy:

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

A fával kapcsolatos algoritmusok általában *rekurzív*ak.

További definíciók:

- *Csomópont foka:* a csomóponthoz kapcsolt részfák száma
- *Fa foka:* A fában található legnagyobb fokszám
- *Levél*: 0 fokú csomópont
- *Elágazás (átmenő csomópont):* > 0 fokú csomópont
- *Szülő (ős):* kapcsolat kezdőpontja (csak a levelek nem szülők)
- *Gyerek (leszármazott):* kapcsolat végpontja (csak a gyökér nem gyerek)
- *Szintszám:* gyökértől mért távolság. A gyökér szintszáma *0*. Ha egy csomópont szintszáma *n*, akkor a hozzá kapcsolódó csomópontok szintszámai: *n+1*.
- *Fa magassága:* a levelekhez vezető utak közül a leghosszabb, vagyis a maximális szintszám.

- *Rendezett fa:* ha az egy szülőhöz tartozó részfák sorrendje lényeges, vagyis azok rendezettek.
- *Kiegyensúlyozott fa:* olyan fa, melynek csomópontjai azonos fokúak, és minden ága egyforma hosszú. Csak a levelek szintjén lehetnek „hiányok". Megadott számú csomópont esetén az ilyen fának legkisebb a magassága.

Egy *f* fokú, *m* magasságú fában maximum $(f^{m+1}-1) / (f-1)$ csomópont helyezhető el. Az előző ábrán szereplő fa kiegyensúlyozatlan, magassága 3, foka 3. A 3 magasságú, 3 fokú fára 40 csomópontot lehetne „feltenni".

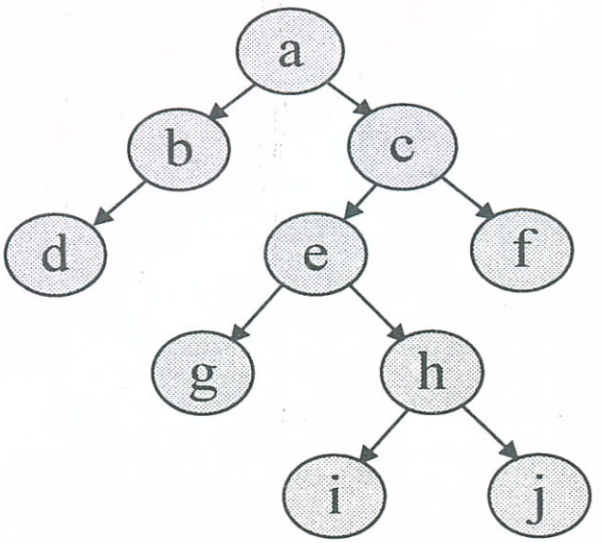
Fa bejárása

A fa csomópontjaiban rendszerint adatokat tárolunk. Világos, hogy ezeket az adatokat valamilyen szisztéma szerint el szeretnénk érni. Kereshetünk például egy adott tulajdonsággal rendelkező adatot, vagy a fa összes adatát ki szeretnénk listázni. Egy általános fa esetén a bejárási stratégiák lényegében a következők lehetnek:

- *Gyökérkezdő (preorder):* gyökér, majd a részfák bejárása sorban, például balról jobbra. Az előző oldalon lévő általános fa gyökérkezdő bejárása:
  `a b c d g e f`
- *Gyökérvégző (postorder):* részfák bejárása sorban, majd a gyökér. Az előző oldalon lévő általános fa gyökérvégző bejárása lépésenként kifejtve:
  ```
  b         c a
  b   d e f c a
  b g d e f c a
  ```

Bináris fa

A bináris fa gyökeréből legfeljebb két részfa ágazik: bal-, és jobboldali részfa.



Bináris fa bejárása

Bináris fa esetén a szülő egyértelműen a gyerekek között helyezkedik el. Ezért itt a gyökérközepű bejárásnak is van értelme. Az itt látható bináris fa bejárási stratégiái:

- *Gyökérkezdő (preorder):* gyökér, bal részfa, jobb részfa:
  `a b d c e g h i j f`
- *Gyökérközepű (inorder):* bal részfa, gyökér, jobb részfa:
  `d b a g e i h j c f`
- *Gyökérvégző (postorder):* bal részfa, jobb részfa, gyökér:
  `d b g i j h e f c a`

Rendezett bináris fa

Az adatok rendezésére, illetve keresésére használt fát *keresési fá*nak (search tree, kereső fa, rendező fa) szokás nevezni. Ilyen a rendezett bináris fa:

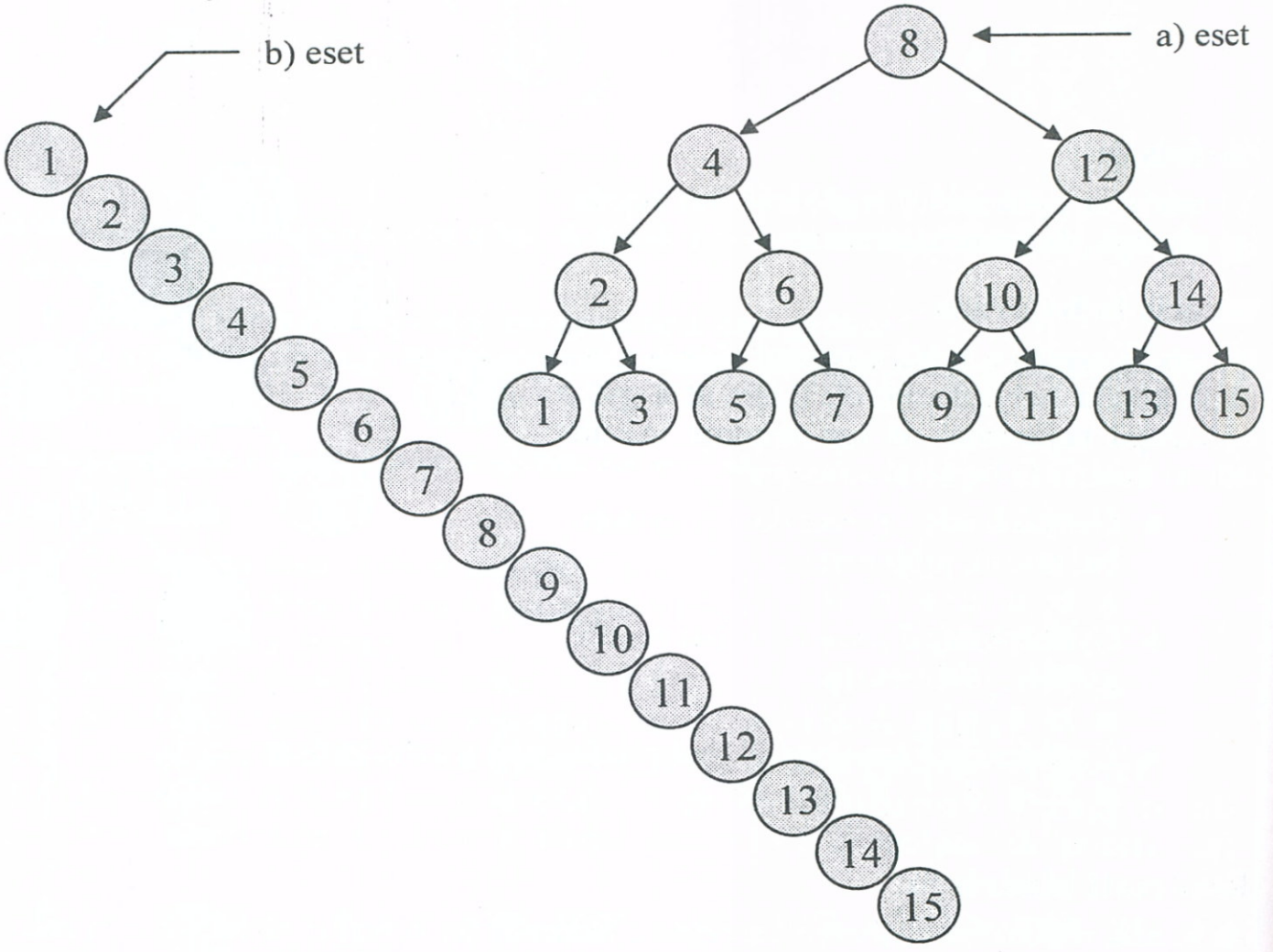
- a baloldali részfa összes eleme kisebb, mint a szülő;
- a jobboldali részfa összes eleme nagyobb, mint a szülő;
- egy n szintű fa elemeinek száma maximum $2^{n+1}-1$;
- egy elemet maximum $n+1$ lépésben megtalálunk.

Rendezett bináris fa esetén az *elemek beszúrásá*nak egyik szokásos algoritmusa a következő: Elemet csak levélként lehet beszúrni. Az első elem a fa gyökere. A többi elem esetén a gyökértől indulva megkeressük annak új helyét – ha az elem kisebb (esetleg egyenlő), mint a csomópont, akkor balra megyünk, egyébként jobbra. Ha a kijelölt út nem folytatódik tovább, akkor egy újabb élet kiépítve az elemet levélként feltesszük a fára. Elem keresése hasonlóképpen történik. A keresendő elemet a fa csomópontjaihoz hasonlítgatva egy egyértelmű útvonalon megyünk lefelé. Ha ezen az útvonalon nem találjuk meg az elemet, akkor az nincs a fán.

Példaként építsünk fel egy számokat tartalmazó bináris fát. A fára „akasztandó" számok legyenek az 1 és 15 közé eső számok. Az elemek beszúrási sorrendje legyen a következő:

```
a)  8, 4, 12, 10, 9, 14, 2, 3, 6, 1, 15, 5, 7, 13, 11
b)  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
```

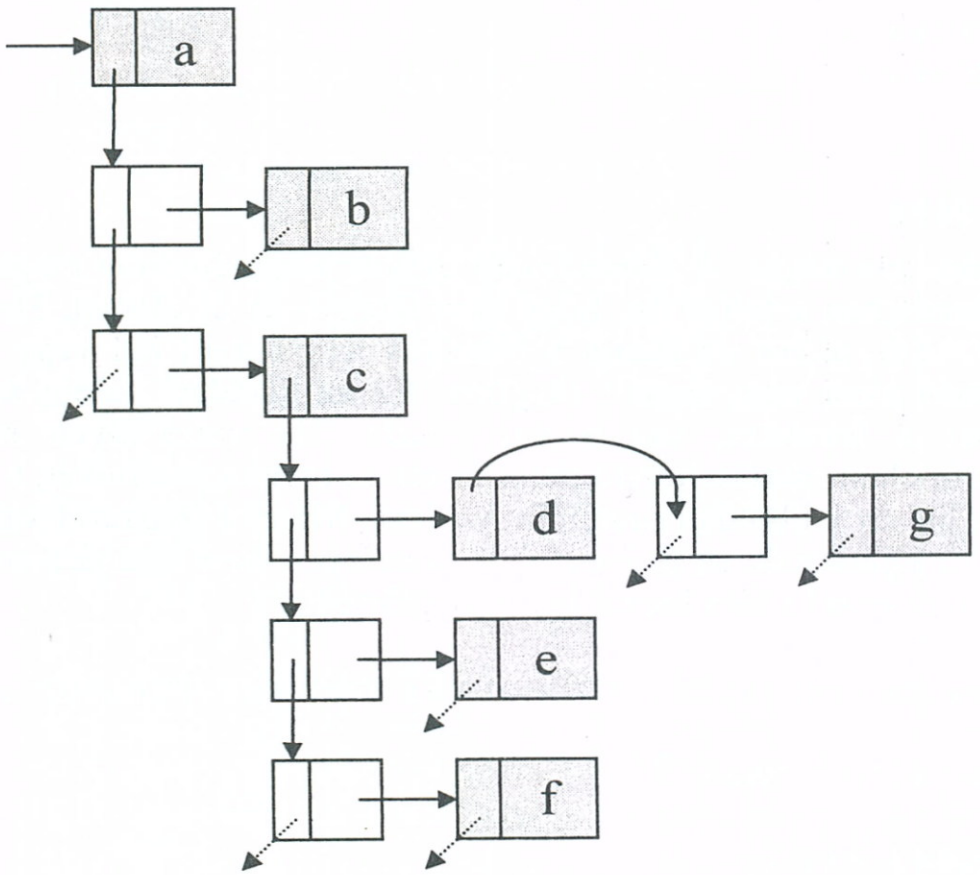
Íme az eredmények:



Talán észrevette az Olvasó, hogy az a) eset szép, kiegyensúlyozott fája nem egészen magától lett ilyen – a trükk az volt, hogy a beszúrandó számokat megfelelő sorrendbe állítottuk. b) esetben a számok teljesen rendezetten jöttek, ezért a fa igencsak torz lett, egészen lineárissá fajult. Az ilyen fát *degenerált fá*nak is nevezik. Látható tehát, hogy a fa alakja „szerencsén múlik": a fa magassága optimális esetben 3, elfajult esetben 14. A degenerált fában való keresés még rosszabb hatásfokú, mint egy lineáris listában, hiszen több hasonlítást végzünk, reménykedve az irányváltásban. Kiegyensúlyozott fa esetében viszont a keresett elem maximum 4 lépésben megvan, vagy kiderül, hogy nincs az elemek között. Általánosítva, ha N az elemek száma, akkor kiegyensúlyozott fa esetén a maximális lépések száma nagyságrendileg Log_2N. A szemléletesség kedvéért: ha az elemek száma 1 milliárd, kiegyensúlyozott fa esetében bármely elemet 30 lépésben mindenképpen megtalálunk (2^{30}=1 073 741 824), ellentétben a lineáris tárolással, ahol keresésnél 0.5 milliárd lépést fogunk átlagosan megtenni.

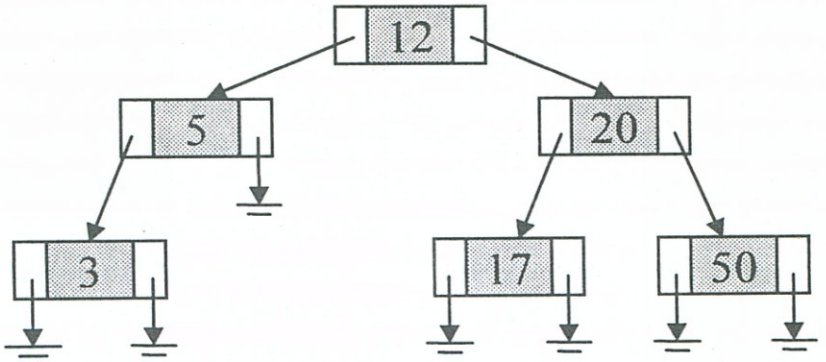
Fa tárolása

Egy általános fát legegyszerűbben multilistában tudunk tárolni. Minden csomóponthoz egy lineáris lista tartozik, melynek első eleme az adat, a többi pedig a kapcsolatok – annyi kapcsolati elem van, ahány fokú a csomópont. A kapcsolatok újabb csomópontokra, illetve lineáris listákra mutatnak. A 224. oldalon lévő fa multilistában való tárolása a következő:



A bináris fa ábrázolása ennek egy speciális esete: ott minden csomópontnak két mutatója van: az egyik a baloldali, a másik a jobboldali részfára mutat.

Kiegyensúlyozatlan bináris keresési fa leképezése dinamikus listára



Íme a megvalósítás Turbo Pascal kódja (részletek). Megfigyelhető, hogy a fával kapcsolatos eljárások mind rekurzívak – a csomópont minden műveletet bal és jobboldali részfájára ruház át. Ha nincs ilyen részfa, akkor az átruházás már nem sikerül, a műveletet meg kell csinálni:

```
Type
   TElem = Word ;
   PCsomo = ^TCsomo ;
   TCsomo = Record
      Bal : PCsomo ;
      Elem : TElem ;
      Jobb : PCsomo ;
   End ;
```

Új elem beszúrása a fába

Beszúrás csak a fa végére lehetséges. A *KeresBeszúr* eljárás paraméterként megkapja a fa gyökerét, vagyis kezdetben a teljes fát, hogy arra rátegye az új elemet. Ha van ilyen gyökér *(Gyökér<>Nil)*, akkor az átadja a beszúrás lehetőségét bal vagy jobb részfájának, a beszúrandó elem nagyságától függően. Ez az „átadás" addig megy, amíg van kinek átadni a feladatot, vagyis van részfa. Ha nincs ilyen részfa *(Gyökér=Nil)*, akkor az új elem részére helyfoglalás történik, megkapja értékét, s mert az új elem levél, annak bal és jobboldali mutatója *Nil* lesz. Mivel a gyökeret változó paraméterként adtuk át az eljárásnak, az ős megfelelő mutatója a *New* hatására erre az elemre fog most mutatni, és ezzel a fára kapcsolás is megtörtént.

```
Procedure KeresBeszur(Elem: TElem; Var Gyoker: PCsomo) ;
   Begin
      If Gyoker = Nil Then
         Begin
            New(Gyoker) ;
            Gyoker^.Elem := Elem ;
            Gyoker^.Bal := Nil ;
            Gyoker^.Jobb := Nil ;
         End
```

```
    Else
      Begin
        If Elem < Gyoker^.Elem Then
          KeresBeszur(Elem,Gyoker^.Bal)
        Else { Megengedjük az egyenlőt is }
          KeresBeszur(Elem,Gyoker^.Jobb) ;
      End ;
  End ;
```

Listázások a háromféle stratégia szerint

A rendezett bináris fából való rendezett kiírást mindig az *inorder* bejárás biztosítja:

```
Procedure InOrder(Gyoker:PCsomo) ;
  Begin
    If Gyoker <> Nil Then
      Begin
        InOrder(Gyoker^.Bal) ;
        Write(Gyoker^.Elem:4) ;
        InOrder(Gyoker^.Jobb) ;
      End ;
  End ;

Procedure PreOrder(Gyoker:PCsomo) ;
  Begin
    If Gyoker <> Nil Then
      Begin
        Write(Gyoker^.Elem:4) ;
        PreOrder(Gyoker^.Bal) ;
        PreOrder(Gyoker^.Jobb) ;
      End ;
  End ;

Procedure PostOrder(Gyoker:PCsomo) ;
  Begin
    If Gyoker <> Nil Then
      Begin
        PostOrder(Gyoker^.Bal) ;
        PostOrder(Gyoker^.Jobb) ;
        Write(Gyoker^.Elem:4) ;
      End ;
  End ;
```

Keresés a fában

```
Function Keres(Elem: TElem; Gyoker: PCsomo): PCsomo ;
  Begin
    If Gyoker = Nil Then
      Keres := Nil
    Else If Elem < Gyoker^.Elem Then
      Keres := Keres(Elem,Gyoker^.Bal)
    Else If Elem > Gyoker^.Elem Then
      Keres := Keres(Elem,Gyoker^.Jobb)
    Else
      Keres := Gyoker ;
  End ;
```

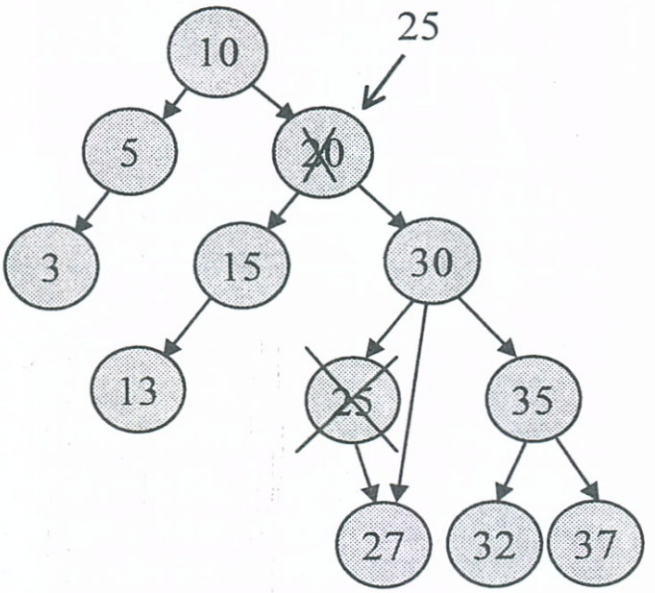
Főprogram – felvitel, listák, keresés

```
Var
  Fa : PCsomo ;
  Elem : TElem ;
  Csomo : PCsomo ;

Begin
  Fa := Nil ;
  Write('Elem: ') ;
  ReadLn(Elem) ;
  While Elem <> 0 Do
    Begin
      KeresBeszur(Elem,Fa) ;
      Write('Elem: ') ;
      ReadLn(Elem) ;
    End ;

  WriteLn('Inorder lista: '); InOrder(Fa); WriteLn ;
  WriteLn('Preorder lista: '); PreOrder(Fa); WriteLn ;
  WriteLn('Postorder lista: '); PostOrder(Fa); WriteLn ;

  WriteLn('Elem: ') ;
  ReadLn(Elem) ;
  Csomo := Keres(Elem,Fa) ;
  If Csomo = Nil Then
    WriteLn('Nincs')
  Else
    WriteLn('Van: ',Csomo^.Elem) ;
End.
```

Törlés a rendezett bináris fából

A törlendő elemet a fáról le kell kapcsolni, ami nem is olyan egyszerű. A kérdéses elemnek ugyanis gyerekei lehetnek, akik továbbra is a fán akarnak maradni. A gyerekeknek tehát új szülőt kell keresni, mégpedig úgy, hogy a rendezettség továbbra is fennálljon. Több esetet különböztetünk meg attól függően, hogy a törlendő elemnek vannak-e gyerekei, azaz van-e bal-, illetve jobboldali részfája:

- *A törlendő elem levél*, azaz nincs se baloldali, se jobboldali részfája: ez a legegyszerűbb eset, hiszen ekkor csupán szülőjének az eddig rá mutató mutatóját kell Nil-re állítani. Ha nincs szülő, vagyis az egyetlen gyökeret töröljük, akkor a fa üres lesz.
- *Csak baloldali vagy jobboldali részfa van:* A részfát átkapcsoljuk a törlendő elem szülőjére. A szülőnek azt a mutatóját, amelyik eddig a törlendő elemre mutatott, most átirányítjuk a megfelelő részfára. Ha nincs szülő (vagyis a gyökeret töröljük), a részfa lesz a gyökér.
- *Van baloldali és jobboldali részfa is*: ez a legbonyolultabb eset. Tegyük fel, hogy az ábrán a 20 értékű csomópontot szeretnénk kitörölni. Megkeressük a 20-at (inorder módon) követő legkisebb elemet – ez a jobboldali részfa legkisebb eleme, amit úgy találhatunk meg, hogy elindulunk jobbra, majd attól kezdve mindig balra. Ha nem tudunk tovább balra menni, akkor megvan az elem. Esetünkben ez a 25-ös. Könnyen belátható, hogy ennek az elemnek nem lehet baloldali részfája. Kapcsoljuk le a 25-ös elemet a fáról az előbbi két pont valamelyike szerint (a csomópontot ne veszítsük el). Ezután a törlendő 20-as csomópontot egyszerűen kicseréljük a 25-ösre.



Az előbb írt program egy nem kiegyensúlyozott fát produkált. Mi ennek a hátránya?

- A fa elvileg teljesen eldeformálódhat – ezzel hatékonysága oly mértékben leromlik, hogy a fa használhatatlan lesz.

Mi az előnye, és mi a hátránya a kiegyensúlyozott bináris fának?

- Előnye, hogy a keresés nagyon hatékony – hasonlítható a rendezett tömbben való bináris kereséshez, a „látogatott" elemek száma azonos.
- Hátránya, hogy a karbantartás bonyolult és időigényes.

Ha az elemek lényegében statikusak, vagyis nincs túl sok törlés, illetve beszúrás, akkor a legjobb megoldás a kiegyensúlyozott fa használata. Egyszer kell csak felépíteni egy ilyen fát, s a keresési idő verhetetlen. De mi a megoldás, ha a feladatból eredően sokat kell módosítanunk, és mégis optimális keresési időt szeretnénk elérni?

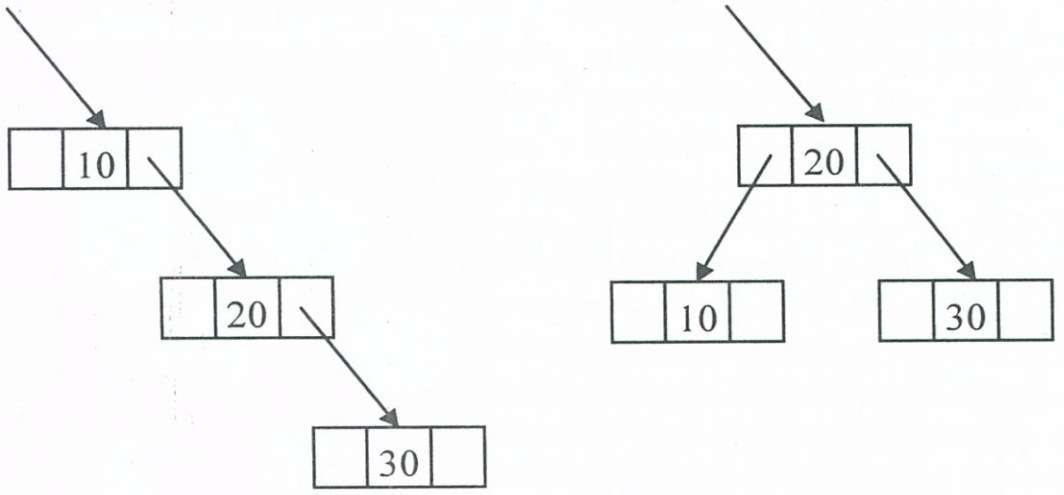
Magasság szerint kiegyensúlyozott bináris fa

A magasság szerint kiegyensúlyozott fa egy arany középút a kiegyensúlyozatlan és a teljesen kiegyensúlyozott fatípus között. Csak annyira van kiegyensúlyozva, hogy a karbantartás még elviselhető legyen, de annyira ki van egyensúlyozva, hogy a keresés elég gyors legyen. E fák elméletét Adelson-Velskii és Landis vezették be 1962-ben, és innen ezeket a fákat AVL fáknak nevezik. Definíciója a következő:

Egy bináris fa akkor kiegyensúlyozott magasság szerint, ha bármely csomópontra igaz: baloldali és jobboldali részfájának magassága közti különbség <= 1.

A szerzők azt is megmutatják, hogyan lehet egy ilyen fát egyensúlyban tartani. Ha például egy faág túlságosan lenyúlik, akkor annak elemeit rotálni kell:

Kiegyensúlyozás előtt: **Kiegyensúlyozás után:**



Adelson-Velskii és Landis bebizonyították, hogy egy ilyen fa legfeljebb 45%-kal magasabb egy tökéletesen kiegyensúlyozott fánál, vagyis ha a fa csomópontjainak száma N, akkor a keresés nagyságrendileg itt is Log_2N lépést vesz igénybe, s ugyanennyibe telik egy új beszúrás, illetve törlés.

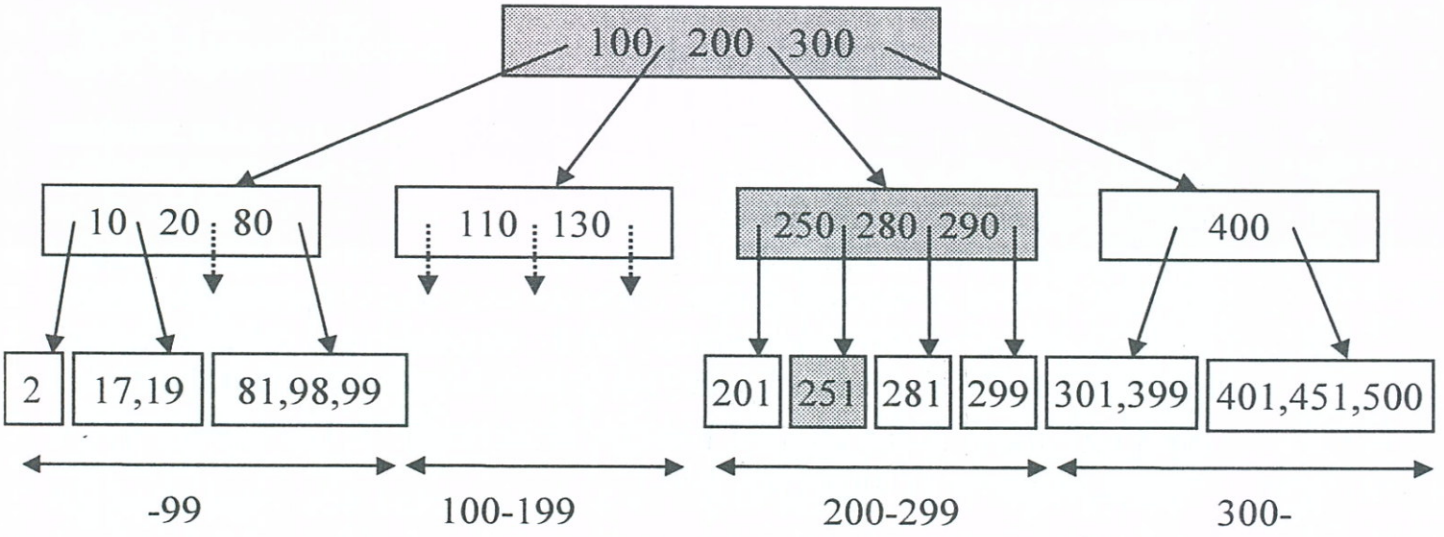
Az adatok gyors karbantartása és keresése érdekében általában az az ésszerű megoldás, ha az adatok kulcsait valamilyen fában tároljuk. A fák megvalósítása történhet akár memóriában, akár lemezen. A memória nem mindig jöhet szóba, hiszen lehet, hogy a kulcsok egyszerűen nem férnek be. De ha be is férnének, költséges megoldás lenne a fát minden feldolgozás előtt felépíteni. Ha a fa lemezen helyezkedik el, akkor a lista mutatói konkrét rekordszámok, és a keresés itt is nagyságrendileg Log_2N lépésbe fog kerülni. Ha nagyon sok a rekord, akkor ez sem jó megoldás, hiszen 1 millió rekord esetén például már minden keresés mintegy 23 lemezolvasást jelent, és ugyanennyi lemezművelet lesz minden egyes beszúrás és törlés is.

Az állományok rendezésénél már láttunk egyfajta köztes megoldást – az adatoknak egy részét lemezen tartjuk, más részét pedig behozzuk a memóriába. A lemezen faként kezeljük az adatokat úgy, hogy a fa csomópontjai több kulcsot tartalmaznak, csökkentve ezzel a lemezműveletek számát. A beolvasott csomópont kulcsait szekvenciáli-

san is szervezhetjük, hiszen a memóriában való keresés rendkívül gyors, főleg, ha a behozott kulcsok száma nem túl nagy.

m-utas keresési fa

Az *m-utas keresési fa* a bináris keresési fa egy *m*-ed fokú általánosítása. A csomópontok több kulcsot is tartalmaznak (ezek egyszerre kerülnek be a memóriába). Minden csomópontnak maximum *m* gyereke lehet, és egy csomópont legfeljebb *m–1* kulcsot tartalmazhat. Bármely csomópontnak eggyel több gyereke lehet, mint ahány kulcsa. A gyerekek kulcsaikkal együtt beékelődnek a szülő kulcsai közé. A következő ábra egy 4-utas keresési fát ábrázol:



A 251 keresése a következőképpen történik: A gyökeret beolvassuk a memóriába, és ott lineáris keresést hajtunk végre a rendezett kulcsok között. Ha ott van a keresett elem, készen is vagyunk. Ha nincs, akkor megállapítjuk, melyik két kulcs közé esik. Mivel a 251 a gyökér 200-as és 300-as alkulcsai közé esik, ezért ennek megfelelően a mutatott csomópontban folytatjuk a keresést (1. szinten a gyökér 3. csomópontja). Ott kiderül, hogy az elem 250 és 280 közötti, így ezután a 2. szinten a mutatott csomóban keresünk. Ott már csak egy kulcs van, mégpedig a keresett. Ajánlatos a gyökeret mindig a memóriában tartani, hiszen minden egyes keresés onnan indul.

B-fa

Belátható, hogy a többutas keresési fánál a lemezolvasások maximális száma a fa magasságával egyenlő. Nyilvánvaló, hogy minél kiegyensúlyozottabb a fa, annál jobban járunk. A fát tehát ki kell egyensúlyozni, és a karbantartási funkciókra is a lehető legjobb megoldást kell találni. Ennek megoldásában a pálmát R.Bayer és McCreight 1972-ben vitték el. Bayer neve után az általuk kidolgozott fát B-fának nevezték el.

A B-fa egy olyan m-utas keresési fa, melyben minden levél ugyanazon a szinten van (a fa kiegyensúlyozott), és minden csomópontnak legalább m/2 gyereke van, kivéve a gyökeret.

Nézzük meg egy speciális B-fa struktúráját és kezelési szabályait, amely egy törzsállomány rekordjainak kulcsait tartalmazza a törzsrekordra való hivatkozással együtt. A legtöbb indexállomány felépítése ezen az elven alapszik.

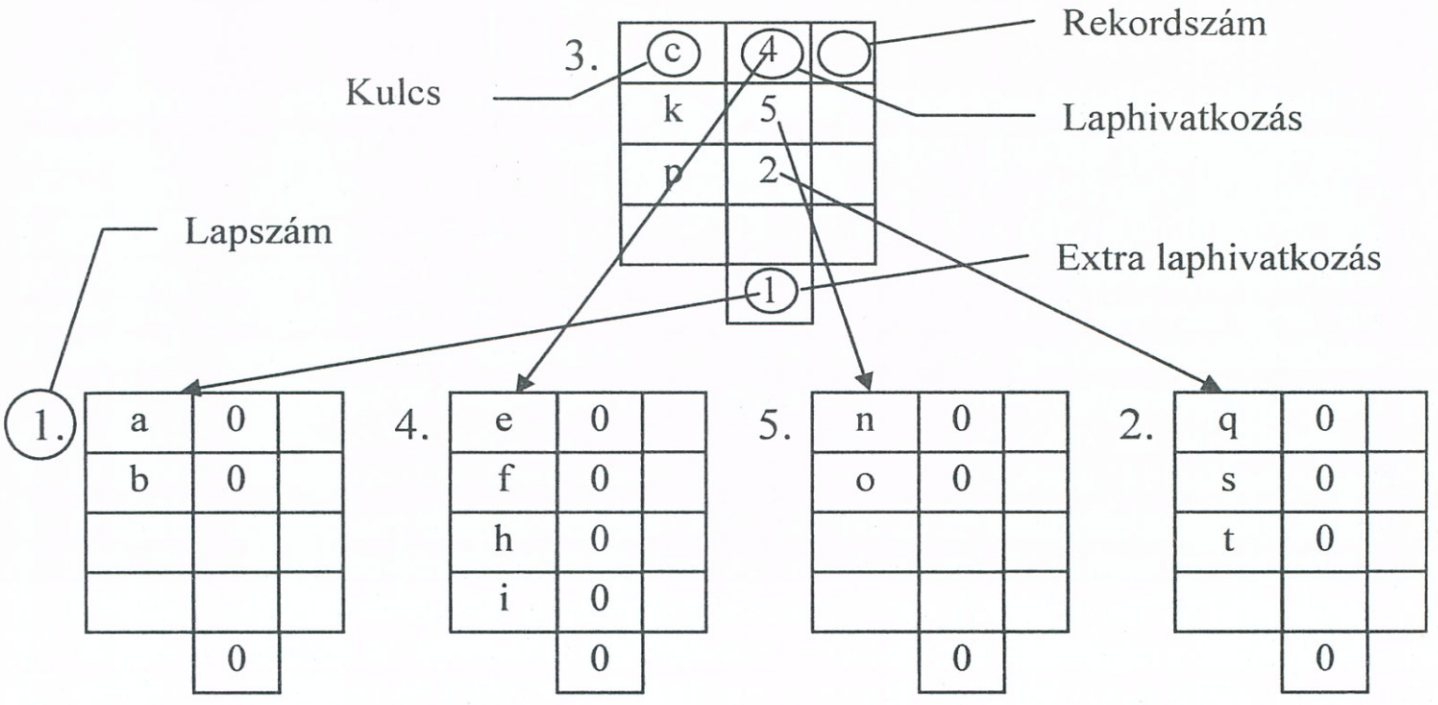
A csomópontokat *lap*oknak nevezzük, melyek *kulcsbejegyzés*eket tartalmaznak. Egy kulcsbejegyzés a következő információkat hordozza:

- *Kulcs*: a keresés tárgya
- *Laphivatkozás*: hivatkozás a fa egy lejjebbi szinten lévő csomópontjára. Ez általában a lemezen levő kulcsállomány egy rekordszáma.
- *Rekordszám*: a kulcshoz tartozó törzsrekord helye a törzsállományban.

Szabályok:

- A fa kiegyensúlyozott.
- A fa különböző szintjein lapok vannak (1 lap = 1 indexállomány rekord).
- A fa „tetején" a gyökérlap áll.
- Egy lap páros számú kulcsbejegyzést, és egy extra laphivatkozást tartalmaz.
- A lapon maximálisan elhelyezhető kulcsbejegyzések száma páros (2N).
- Minden lap legalább félig tele van, kivéve a gyökeret.
- A kulcsbejegyzés laphivatkozása olyan lapra mutat, ahol minden kulcs nagyobb nála, kivéve az utolsó szintet, ott nincsenek aktív laphivatkozások.
- Az extra lapszám olyan lapra mutat, ahol minden kulcs kisebb az ittlévő legkisebbnél. Kivétel az utolsó szint, mely nem hivatkozik semmire.

A következő ábra egy ilyen B-fát ábrázol konkrét értékekkel. A lap mérete 4, a kulcsok az egyszerűség kedvéért az angol abc betűi:

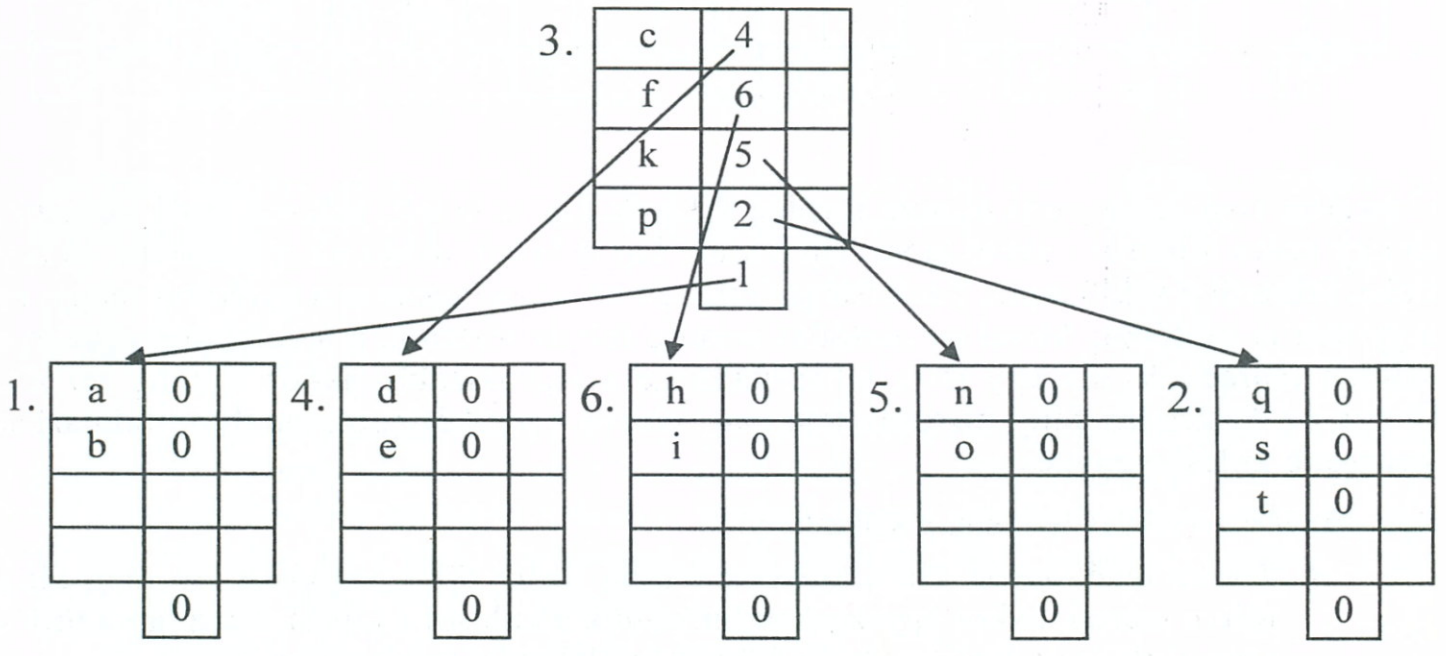


A lapok a lemezen 1, 2 stb. sorrendben helyezkednek el. A gyökérlap most a 3. lap, innen fog indulni mindenféle keresés. Ennek az extra lapszáma 1, ami azt jelenti, hogy az 1. lapon levő kulcsok az összes itt lévő kulcsnál kisebbek. Valóban, akár *a*, akár *b* kisebb, mint *c*, *k* vagy *p*. A *c* kulcsbejegyzés laphivatkozása 4, vagyis a 4. lapon talál-

hatók a *c* és *k* közötti elemek. *k* laphivatkozása 5, így az 5. lapon lévő kulcsértékek *k* és *p* közé esnek. Végül *p* laphivatkozása 2: a 2-es lapon lévő kulcsok mind nagyobbak *p*-nél. Látható, hogy az 1. szinten levő lapok fésűszerűen beékelődnek a szülő lap kulcsai közé. Rendezett végigolvasás esetén a lapok érintési sorrendje: 1,3,4,3,5,3,2. Mivel egy adott lapon található kulcsok alatt, között és felett is lehetnek értékek, ezért bármely lapnak eggyel több gyereke van, mint a lapon található kulcsok száma. A levéllapokon levő kulcsok összes laphivatkozása nyilvánvalóan 0, azaz nem mutatnak sehová sem. A lapok az indexállomány rekordjai, vagyis egy lemezolvasással egy egész lapot olvasunk be a memóriába. A lapon a kulcsok rendezetten helyezkednek el.

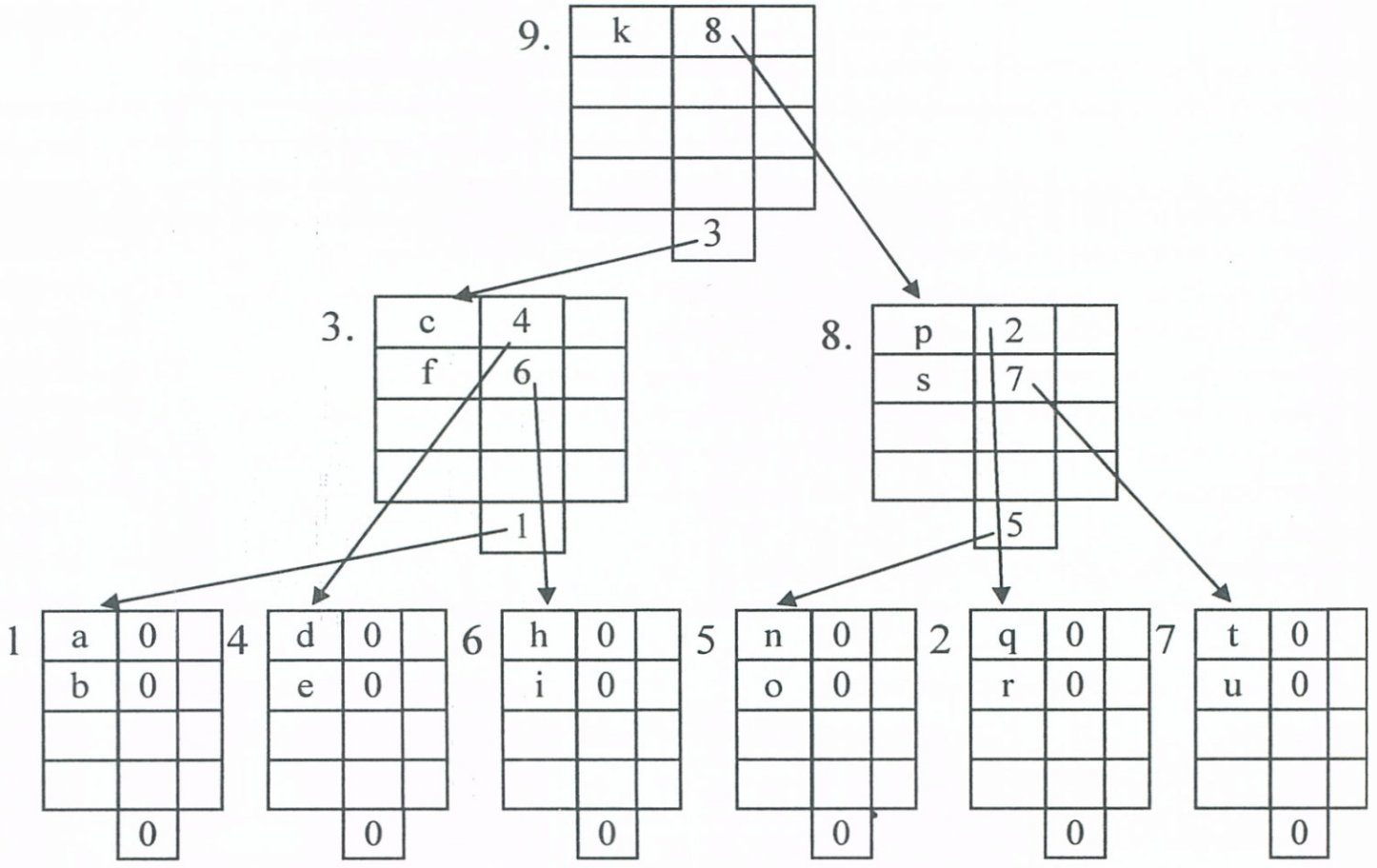
Egy elem keresésekor itt is egy egyértelmű úton haladunk lefelé, mint a bináris fa esetében. A különbség az, hogy most minden egyes csomópontnál több mint kétfelé ágazhatunk, és egy-egy csomópontban belső keresést is végzünk. Ha a fa magassága *h*, akkor maximum *h+1* lépésben az elemet megtaláljuk, vagy kiderül, hogy nincs a fán.

Nézzük most meg, hogyan történik egy újabb elem beszúrása a B-fába. Természetesen most is először meg kell keresnünk a kérdéses kulcsot. Ha van már ilyen, akkor a beszúrást nem végezzük el. Ha nincs, akkor az új elemet abba a levélbe kell tenni, ahol a keresést abbahagytuk. Ha például az *l* vagy *m* kulcsokat szeretnénk beszúrni, azok az 5. lapra kerülnének. Akkor van probléma, ha a kérdéses levélen nincs több hely. Próbáljuk például felvinni a *d* kulcsot! Ennek a 4. lapon lenne a helye, de ez a lap már előzőleg betelt (234. oldal). A levél ekkor osztódik: keletkezik egy újabb lap, melynek száma 6. A 4-es lapon most az új kulccsal együtt *2N+1* kulcsunk lenne *(d,e,f,h,i)*. Ebből az *N* legkisebb kulcs *(d,e)* a 4-es lapon marad, az *N* legnagyobb *(h,i)* átkerül az új 6-osba. A középső kulcs *(f)* a szülő lapra kerül (jelen esetben ez a 3-as gyökér), ahol a *c* és *k* kulcsok közé beszúródik az *f*. *k* és *p* lejjebb kerül, a laphivatkozásaikat viszik magukkal. *f* a 6. lapra fog mutatni.



Vigyük fel még az *r* és *u* kulcsokat! Mindkettő helye a 2. lapon lenne, de *u* már nem fér oda. Most tehát a 2. lap osztódik. Az szétosztandó kulcsok: *q,r,s,t* és *u*. A kicsik (*q*

és *r*) maradnak a 2-es lapon, a nagyok (*t* és *u)* a most létrehozott 7. lapra kerülnek, *s* pedig megy a gyökérbe. Igen ám, de a gyökér megtelt! Most tehát a gyökér osztódik a *c,f,k,p* és *s* kulcsokkal. Keletkezik egy 8-as lap, ahová az eddigi gyökér nagyjai kerülnek (*p,s*), a kicsik (*c,f*) maradnak a 3-as lapon. A középső (*k*) az új gyökérbe kerül, amely a 9-es lapszámot kapja. A gyökérlapon a *k* kulcs laphivatkozása 8 lesz, hiszen oda kerültek a nála nagyobb kulcsok. Az extra laphivatkozás pedig 3 lesz, hiszen ebben maradtak a *k*-nál kisebb kulcsok. A 8-as lap extra laphivatkozása 5 lesz, ez volt eddig a *k* kulcs hivatkozása:



Összefoglalva: a beszúrás mindig levélbe történik. Ha az betelt, akkor a levél osztódik, és egy kulcs felkerül a szülő lapra. Ha a szülő lap is betelt, akkor az is osztódik, és egy kulcs felkerül a szülő lapra. Szélsőséges esetben ez a lánc egészen a gyökérig megy, és a gyökér is osztódik. Mivel az osztódás mindig felfelé történik, a levelek ugyanazon a szinten maradnak. Ha egy B-fa kihasználtsága a legrosszabb, akkor is félig telített, hiszen az összes lap félig biztosan tele van, kivéve a gyökérlapot, amely tartalmazhat kevesebb kulcsbejegyzést is.

A *B-fából* való *törlés* lényege a következő:

- *Törlés levéllapról:* Ha a kulcsok száma N-nél több, vagyis a lap még félig töltve marad, akkor egyszerűen kitöröljük a kulcsot a lapról. Ha a lap a törlés után alultöltött lenne, akkor a következő lehetőségeink vannak:

a) Ha valamelyik szomszédos lapon N-nél több kulcs van, akkor a törölt kulcs helyébe valamelyikről úgy veszünk el egy kulcsot, hogy a rendezettség megmaradjon – az elemeket rotáljuk.
b) Ha a szomszédos lapokon sincsen elég kulcs, akkor két levelet egyesítünk, és a szülőből is az egyesített lapra teszünk egy kulcsot.

- *Ha a törlendő elem nem levéllapon van:* Belátható, hogy a B-fa bármelyik kulcsát közvetlenül megelőző és közvetlenül követő kulcs biztosan levéllapon helyezkedik el. A törlést úgy végezzük el, hogy a törlendő elem helyébe ezek közül valamelyiket áttesszük az előbbi szabályok figyelembe vételével.

Látható, hogy ugyan a beszúrás és főleg a törlés elmélete bonyolult, az algoritmussal járó lemezmozgás aránylag kevés, és így az adatok karbantartása és elérése kielégítő.

Kalkulációk:

Ha K a kulcsok várható száma, és $2N$ a lap mérete (maximális kulcsbejegyzések száma), akkor

- A B-fa várható magassága $Log(K) / Log(N) + 1$
- A lapok száma, vagyis az indexállomány mérete
  legrosszabb esetben: K/N, ha minden lapon csak N kulcs van,
  legjobb esetben: $K/(2N)$, ha minden lapon $2N$ kulcs van.
- Egy lap fizikai mérete a kulcs hosszából, a hivatkozások méreteiből, valamint a lapméretből számítható.

Például, ha a lap méretét 200-nak vesszük, akkor 10 milliárd kulcs esetén ($N=100$ és $K=10^{10}$):

- A várható magasság $Log(10^{10})/Log(10^2)+1=6$, vagyis bármely kulcsot maximum **6 lemezolvasással megtalálunk**, és ez a lényeg!
- A lapok száma a maximális kulcsszám esetén 50 000 000 és 100 000 000 között fog mozogni.
- Ha egy kulcsbejegyzés összesen 50 byte-ot foglal le, akkor az egyszerre beolvasandó laprekord mérete mintegy 200*50=10000 byte.

A B-fának vannak még ennél is jobb változatai, melyeket a szerzők fejlesztettek tovább *B*-fa* és *B^+-fa* neveken. Ezekkel azonban mi most nem foglalkozunk.

11.9 Tábla

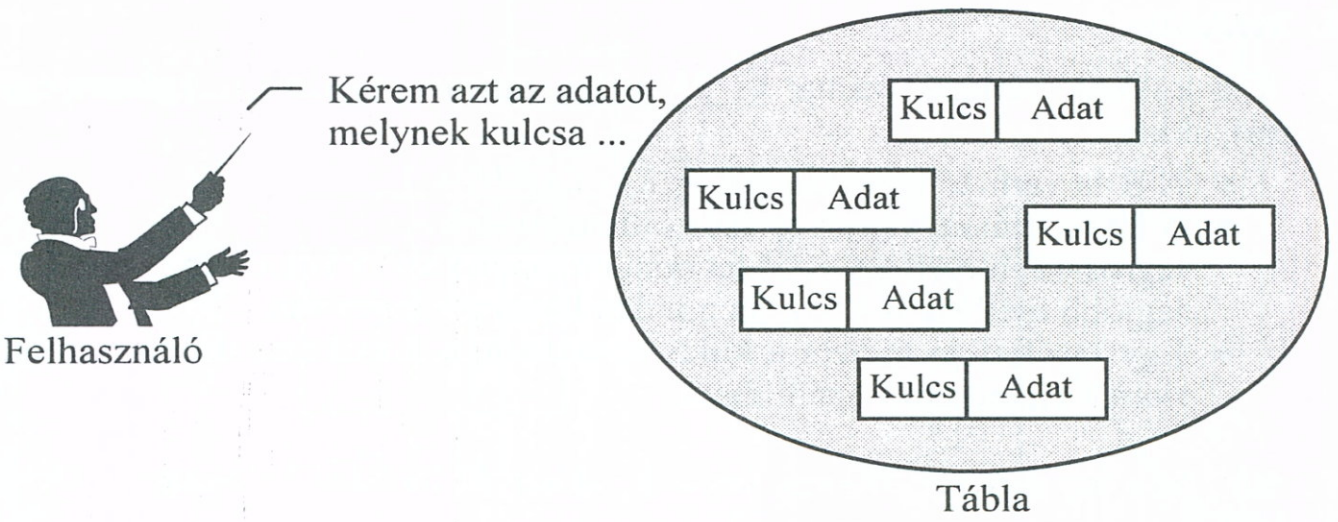
A tábla egy asszociatív adatszerkezet, mely elemei kulcs és adat párok, ahol a kulcsok egyediek, és bármely elem a kulcsán keresztül érhető el. A táblával kapcsolatos műveletek központi kérdése az adott kulcshoz tartozó adatok minél rövidebb idő alatt történő megkeresése, és a tábla karbantartása. A táblához, mint adatszerkezethez „intézett" kívánságok általában a következők:

- Jegyezd meg ezt a kulcsot és a hozzá tartozó adatot!
- Kérem az ehhez a kulcshoz tartozó adatokat!
- Töröld ki azt az adatot, melynek kulcsa ez!

- Módosítsd azt az adatot, melynek a kulcsa ez!
- Sorold fel az összes kulcsodat, hadd válasszak közülük!
- Sorold fel az összes, ilyen és ilyen tulajdonságokkal rendelkező adatodat!

A tábla nem fog minden kívánságot teljesíteni, hiszen vannak beépített szabályok, mely szerint cselekednie kell. Ezek a szabályok két csoportba oszthatók:

- *Logikai szabályok*, melyek minden táblára igazak fizikai megvalósítástól függetlenül. Ilyen például, hogy a tábla nem hajlandó olyan adatot megjegyezni, amelynek kulcsa már létezik a nyilvántartott kulcsok között.
- *Fizikai szabályok*, melyek az adott tábla fizikai megvalósításának korlátai. Ilyen például, hogy a tábla nem jegyez meg több adatot, ha tárolója betelt.



A tábla tehát egy olyan adatszerkezet, amelyen a következő műveletek vannak értelmezve:

- Keresés
- Beszúrás
- Törlés
- Szekvenciális elérés

A táblaműveletek hatékonysága a fizikai tárolástól, vagyis a tábla szervezésétől függ. Szinte minden művelet tartalmaz keresést, hiszen nem vihetünk be például olyan adatot, melynek kulcsa már szerepel a táblában. Ha a felhasználó erről nem is bizonyosodik meg, a tábla mindenképpen megteszi helyette: egy jól működő táblát nem lehet elrontani, legfeljebb értelmetlen adatokkal telepakolni.

A tábla az egydimenziós tömb általánosítása – ott a tárolt adatokat indexeken keresztül érjük el, vagyis a kulcsok maguk az indexek. A táblákkal kapcsolatos műveletekkel a könyv olvasása során már több ízben találkozhatott az Olvasó, akár az egydimenziós tömb, akár az állományok karbantartásánál. Ezért ebben a pontban a táblával kapcsolatos ismereteket összefoglalás jelleggel tárgyaljuk. Azt már tudjuk, hogy egy adattömeg „átlátása" és gyors karbantartása nem könnyű feladat. Ahhoz, hogy egy dinamikusan változó adathalmazban „pillanatokon belül" megtaláljunk egy adott kulcsot, ahhoz az adathalmazt komolyan meg kell szervezi, legyen az a memóriában, vagy egy

külső adathordozón. Alapvetően itt most a két „klasszikus" tároló kínálkozik – a vektor és a lista:

Vektor:

1.	Kulcs	Adat
2.	Kulcs	Adat
...		
m.	Kulcs	Adat

Lista:

Kulcs	Adat	Mutató
Kulcs	Adat	Mutató
...		
Kulcs	Adat	Mutató

Soros tábla

Ez a táblák legegyszerűbb formája. A soros táblában az adatokat az indexek növekvő sorrendjében tároljuk. Ha a tábla telítettsége N, a maximális elemszám pedig M, akkor az egyes műveletek a következők lesznek:

- *Keresés:* Egyszerű szekvenciális keresés rendezetlen tömbben:

```
I := 1
Ciklus amíg (I <= N) és (Kulcs <> Kulcsok[I])
   I := I+1
Ciklus vége

Ha I <= N akkor
   Kulcs indexe: I
egyébként
   Nincs ilyen kulcs
Elágazás vége
```

A keresés lényegesen gyorsítható, ha az utolsó elem után elhelyezzük a keresett elemet, mintegy ütközőként használva azt:

```
Kulcsok[N+1] := Kulcs
I := 1
Ciklus amíg Kulcs <> Kulcsok[I]
   I := I+1
Ciklus vége

Ha I <= N akkor
   Kulcs indexe I
egyébként
   Nincs ilyen kulcs
Elágazás vége
```

- *Beszúrás:* Ha van még hely a táblában és nincs még ilyen kulcs, akkor az új elemet betesszük az utolsó utáni helyre, vagyis az ütköző helyére. N értékét növeljük 1-gyel.
- *Törlés:* Ha van ilyen elem, akkor annak kulcs és adatrészét felülírjuk az utolsó elemmel. N értékét csökkentjük 1-gyel.

A soros táblák feltűnően jó hatásfokkal működnek kis elemszám esetén. Igaz ugyan, hogy a tábla rendezetlensége miatt nem alkalmazható a gyors bináris keresés, de olyan végtelenül egyszerű a beszúrás, illetve a törlés, hogy használata minden olyan esetben ajánlott, ahol nem túl nagy az elemszám, és nincs szükség rendezettségre.

Önátrendező tábla

Ha a kulcsok közül bizonyosakat sokkal többször keresünk, mint a többit, akkor összességében sokkal hatékonyabb a keresés, ha ezek a kulcsok a tábla elején vannak. Elég nehéz lenne azonban a kulcsok kereséseiről statisztikákat készíteni és ennek megfelelő módszert kidolgozni azok folyamatos átrendezésére. Óriási ötlet, miszerint a keresett kulcsot mindig a tábla elejére tesszük, hiszen így az a kulcs, amelyet sohasem keresünk, előbb vagy utóbb a tábla végére szorul. Vektoros megvalósítás esetén ugyan nem sokat nyerünk a sok adatmozgatás miatt, de listában való tárolásnál nem kell mást csinálnunk, mint a keresett adatot átkapcsolni a lista elejére.

A következő programrészlet egy önátrendező táblában való keresést valósít meg dinamikus listán. A *Van* függvény paraméterként megkapja a keresendő kulcsot, és visszaadja egyrészt a keresés sikerességét, másrészt találat esetén a keresett kulcs mutatóját, mely a keresés végrehajtása után minden esetben a lista első eleme lesz:

```
Type
   PElem = ^TElem ;
   TElem = Record
      Kov : PElem ;
      Kulcs : TKulcs ;
      Adat : TAdat ;
   End ;
```

```
Var
  Elso : PElem ;

Function Van(Kulcs: TKulcs; Var Akt: PElem): Boolean ;
  Var
    Elozo : PElem ;
  Begin
    Elozo := Nil ;
    Akt := Elso ;
    While (Akt <> Nil) And (Akt^.Kulcs <> Kulcs) Do
      Begin
        Elozo := Akt ;
        Akt := Akt^.Kov ;
      End ;
    If Akt = Nil Then
      Van := False
    Else
      Begin
        { Van ilyen kulcs, a lista elejére kapcsoljuk:}
        Van := True ;
        If Akt <> Elso Then
        Begin
          Elozo^.Kov := Akt^.Kov ;
          Akt^.Kov := Elso ;
          Elso := Akt ;
        End ;
      End ;
  End ;
```

Rendezett tábla

Egy rendezett táblában mindig gyorsabb a keresés, mint egy rendezetlenben. Szekvenciális keresés esetén például nem folytatjuk a keresést, ha a keresett elemet rendezettségben lehagytuk:

```
I := 1
Ciklus amíg (I <= N) és (Kulcs > Kulcsok[I])
  I := I + 1
Ciklus vége
Ha (I <= N) és (Kulcs = Kulcsok [I]) akkor
  Kulcs indexe: I
egyébként
  Nincs ilyen kulcs
Elágazás vége
```

Ütköző használatával a rendezett táblában való keresés így fest:

```
Kulcsok[N+1] := Ütköző (jó nagy)
I := 1
Ciklus amíg Kulcs > Kulcsok[I]
   I := I+1
Ciklus vége

Ha Kulcs = Kulcsok[I] akkor
   Kulcs indexe: I
Egyébként
   Nincs ilyen kulcs
Elágazás vége
```

Sikeres keresés esetén nincs javulás a szekvenciálishoz képest, de a sikertelen keresés átlagosan fele annyi időt vesz igénybe.

Ha a bináris keresést alkalmazzuk, akkor a keresési idő a lineáris keresési idő tört részére csökkenthető. A bináris keresés algoritmusa:

```
Elso := 1
Utolsó := N
Ciklus
   I := (Elso+Utolsó) Div 2
   Elágazás
      Kulcs > Kulcsok[I] esetén Elso := I+1
      Kulcs < Kulcsok[I] esetén Utolsó := I-1
   Elágazás vége
mígnem (Első > Utolsó) vagy (Kulcs = Kulcsok[I])
Ciklus vége

Ha Elso <= Utolsó akkor
   Kulcs indexe: I
egyébként
   Nincs ilyen kulcs
Elágazás vége
```

Rendező fa táblája

Ha az előző pontban tárgyalt fa adatszerkezetet arra használjuk, hogy egyedi kulcsokkal rendelkező adatokat tartunk karban (beszúrunk, törlünk, illetve módosítunk), valamint adatokat keresünk és adott esetben listázunk, akkor az egy tábla adatszerkezetként is felfogható. Mivel ott elég részletesen érintettük ezeket a kérdéseket, most a fa karbantartásának tárgyalásától eltekintünk.

Nézzük a rendezett bináris fában való keresés nem rekurzív algoritmusát. A fát egy bal és jobb mutatóval rendelkező listában tároljuk. Az *Adat(M)* az *M* mutató által mutatott *Adat*.

Bal	Kulcs	Adat	Jobb

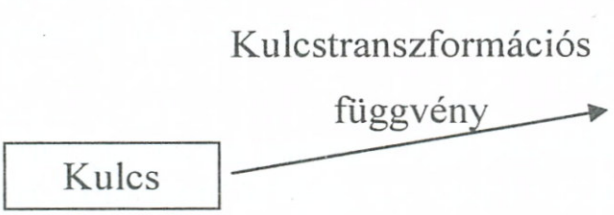
M egy mutató, és *Adat(M)* az *M* által mutatott *Adat*:

```
M := Gyökér
Ciklus amíg (M <> 0) és (Kulcs <> Kulcs(M))
   Ha Kulcs < Kulcs(M) akkor
      M := Bal(M)
   egyébként
      M := Jobb(M)
   Elágazás vége
Ciklus vége

Ha M <> 0 akkor
   Kulcs mutatója: M
egyébként
   Nincs ilyen kulcs
Elágazás vége
```

Kulcstranszformációs táblák

Minden táblához tartozik egy algoritmus, mely meghatározza az új elem beszúrásának helyét. Soros tábla esetén a beszúrás időpontja számít, rendezett táblánál a kulcs nagysága.

Kulcstranszformációs tábla esetében a kulcshoz tartozó transzformáció (függvény) adja meg az elem helyét. Az *„Állomány"* fejezetben már szó volt a direkt és a véletlen szervezésekről. Lényege, hogy a kulcs ismeretében egy algoritmus számítja ki a tárolt adat fizikai címét, és ezzel a hasonlítgatások számát minimálisra csökkentjük. Ha a kulcsok és a fizikai címek közt találunk egy egyértelmű leképezést, akkor természetes, hogy ez a legjobb táblaszervezési megoldás. A kulcsok értékei azonban csak nagyon ritkán olyan szabályosak, hogy ezt megtehessük.

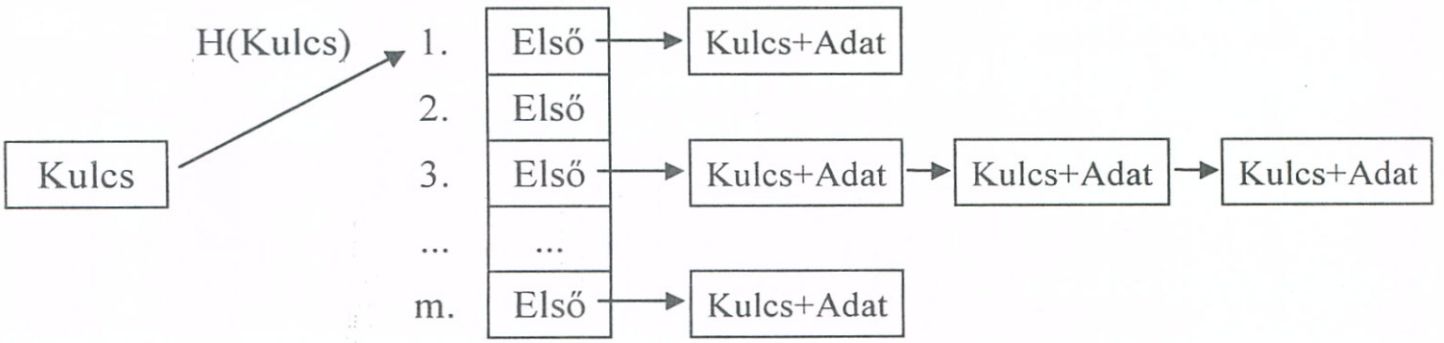


	Kulcs	Adat
1.	Kulcs	Adat
2.	Kulcs	Adat
3.	Kulcs	Adat
...	...	...
m.	Kulcs	Adat

A transzformációs függvényt *hasítófüggvény*nek is szokás nevezni. Vannak kulcstranszformációs függvények, melyek minden kulcshoz egyértelmű címet rendelnek – ilyen például a napi dátum leképezése az év napjainak sorszámára. Ez a legegyszerűbb

eset, és a karbantartás is rugalmas. Ha nincs egyértelmű transzformáció, akkor szokásos a kulcshoz rendelt véletlen érték is. Ezt a hozzárendelést randomizálásnak szokás nevezni. Randomizálás esetén természetesen előfordul, hogy a transzformációs függvény két kulcsot ugyanarra az indexre képez le. Az azonos helyre leképezett kulcsokat szinonimáknak nevezzük, és ezek elhelyezéséről külön kell gondoskodni. A szinonima-elhelyező algoritmust vagy függvényt külön meg kell határozni. Szokás a szinonimákat egy külön túlcsordulási területre elhelyezni, de fontos, hogy elhelyezés után azok egyértelműen felderíthetők legyenek. Sokszor jó megoldás például a következő:

Mivel minden kulcsot egy jól meghatározott címre képezünk le, ezen a címen tárolhatunk egy lista mutatót, mely az erre a címre leképzett adatok listájára mutat:



Ha a kulcsok egész számok, akkor randomizáló függvények például a következők:

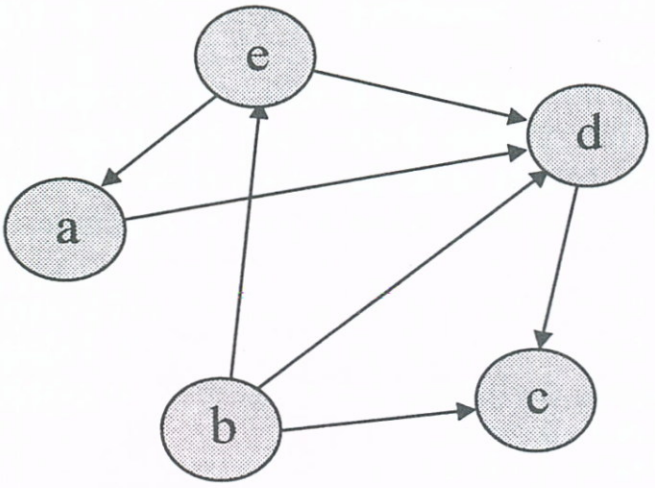
- *Maradékképzés*: Ha a kulcsok száma n, melyek aránylag egyenletesen oszlanak szét, és a fizikai címek 1 és m közé esnek, akkor a hasító függvény:
  H(Kulcs) := (Kulcs mod m) + 1
  Ez a függvény prímszám méretű tábla esetén a leggazdaságosabb.
- *Szorzás*: A kulcsot összeszorozzuk önmagával, és vesszük az eredmény középső jegyeit.
- Egy olyan *véletlenszám-generátor*, ami egy adott értékhez mindig ugyanazt a véletlen értéket rendeli (ez persze nem „igazi" véletlen szám).

11.10 Hálós adatszerkezetek

Egy hálós adatszerkezetben bármely két csomópont kapcsolatban állhat egymással. Ha csupán a kapcsolat léte és iránya lényeges, akkor *irányított gráf*ról beszélünk, ha pedig a kapcsolatokhoz valamilyen mérőszám is tartozik, akkor az adatszerkezet egy *hálózat*.

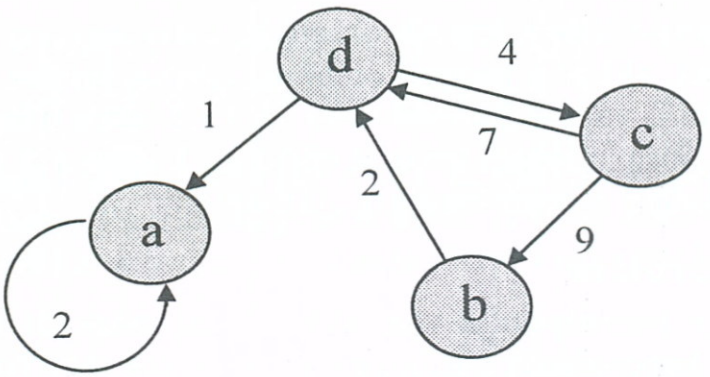
Egy irányított gráfot, illetve hálózatot nem szükségszerű „lerajzolni", azt a csomópontok és a kapcsolatok (plusz hálózat esetén a hozzá tartozó értékek) felsorolásával is megadhatjuk. A 11.2. ábrán egy irányított gráf grafikus és szöveges megadása látható. A kapcsolatokat szisztematikusan kell felsorolni – példánkban azok rendezettek kezdőpont és azon belül végpont szerint. A 11.3. ábrán a hálózat kapcsolataihoz egész

értékek tartoznak, melyeket a gráf éleire írtunk, illetve a kapcsolatok utolsó adataiként adtunk meg.



Csomópontok: (a,b,c,d,e)
Kapcsolatok: (a,d),(b,c),(b,d),(b,e),(d,c),(e,a),(e,d)

11.2. ábra. Irányított gráf



Csomópontok: (a,b,c,d)
Kapcsolatok: (a,a,2),(b,d,2),(c,b,9),(c,d,7),(d,a,1),(d,c,4)

11.3. ábra. Hálózat

Nézzünk néhány példát a hálós adatmodellekre:

- Tegyük fel, hogy adott *n* falu. Ismerjük, hogy melyik faluból közlekedik a másikba autóbusz. Az autóbuszjárat lehet egyirányú is. Kérdés, hogy egy adott faluból el lehet-e jutni autóbusszal a másikba? Az ilyen jellegű feladatoknál csak az számít, hogy van-e busz, vagy nincs. Hogy mennyit kell utaznunk ahhoz, hogy célba érjünk, az most nem érdekes. Ez az adatmodell egy *irányított gráf.*

- Mint az előző feladat, de most nem elégszünk meg akármilyen hosszú utazással. Arra vagyunk kíváncsiak, létezik-e egyik faluból a másikba egy adott útnál rövidebb út, illetve melyik út a legrövidebb a két falu között. Ebben az esetben már *hálózatról* beszélünk, hiszen a falvak közti úthosszakat is számításba kell venni.
- Van egy társaság, abban férfiak és nők. Az a feladat, hogy a társaság tagjait minél nagyobb létszámban össze kell házasítani. Mindenki megmondja, hogy kiket szeret. Magát senki sem szeretheti. Ha két egyén között mindkét irányban létezik kapcsolat, akkor a pár összeadható. Ez az adatmodell egy *irányított gráf.*
- Mint az előző feladat, de a társaság tagjai közül nem mindenki szeret egyformán. A szeretetet 0-tól 5-ig lehet osztályozni. Feladat a társaság tagjainak olyan összetársítása, melyben összességében a legtöbb szeretet van. A társaság tagjai *hálózatot* alkotnak.

Definíciók

- Egy hálós adatmodellben *út*nak nevezzük az $a_0, a_1, \ldots a_m$ csomópontok sorozatát, ha az
  $(a_0, a_1), (a_1, a_2), \ldots (a_{m-1}, a_m)$
  kapcsolatok fennállnak. Ekkor az út hossza m, kezdőpontja az a_0, végpontja pedig az a_m csomópont. A 11.2. ábrán a *b, e, d,* és *c* csomópontok egy 3 hosszúságú utat határoznak meg. Egy útban egy csomópont akár többször is szerepelhet.
  Ha a hálós adatmodellben létezik olyan út, mely egy csomópontot legalább kétszer tartalmaz, akkor a modell *ciklikus*, illetve *hurkot tartalmazó*, egyébként *ciklus nélküli*, vagyis *hurokmentes*. A 11.2. ábra irányított gráfja hurokmentes, a 11.3. ábra hálózata viszont ciklikus, hiszen a *d, c, b, d, c, b, d, c, b,* ... útvonalon például a végtelenségig „járhatnánk".

Hálós adatszerkezet tárolása

Ha az adatszerkezetnek n csomópontja van, akkor a csomóponti adatok megadhatók egy n elemű vektorral, a kapcsolatok pedig egy $n \times n$ elemű mátrixszal, ahol a csomópontok „keresztezésébe" irányított gráf esetén egy igen/nem adatot, hálózat esetén pedig egy mérőszámot teszünk.

Az irányított gráf kapcsolatainak mátrixát *szomszédossági mátrix*nak nevezik. Ez tehát Boole mátrix, mely $n \times n$ darab logikai értéket tartalmaz. Az (igen/nem) értékeket legtöbbször a 0 vagy 1 reprezentálja.

A hálózat kapcsolatait leíró mátrixot *súlymátrix*nak nevezik, ahol a mátrix elemei a kapcsolatokhoz tartozó értékek (súlyok).

Feladat
Öt falu (a, b, c, d és e) közötti buszjáratokat a 11.2. ábra szemlélteti. Írjunk programot, mely megadja, melyik faluk között létezik 1 hosszú út, 2 hosszú út stb., n hosszú út!

A faluk közti kapcsolatokat a következő szomszédossági mátrix írja le:

	a	*b*	*c*	*d*	*e*
a	0	0	0	1	0
b	0	0	1	1	1
c	0	0	0	0	0
d	0	0	1	0	0
e	1	0	0	1	0

Látható, hogy a mátrix főátlója csupa 0, hiszen egyik faluból sincs értelme buszjáratot indítani önmagába. A 11.2. ábra szerint az *a* faluból közvetlenül csak *d*-be lehet eljutni. Ezért az *a* sorában a *d* oszlopot kivéve minden érték 0. A *b* faluból *a*-ba és saját magába nincs közvetlen buszjárat, *c*, *d* és *e*-be viszont van, stb.

A szomszédossági mátrix tulajdonsága, hogy ha önmagával n-szer összeszorozzuk, akkor azokon a helyeken kapunk igen (1-es) értékeket, ahol a két csomópont között van pontosan n hosszú út.

Két mátrix szorzása:

Tegyük fel, hogy *M1* és *M2* szimmetrikus mátrixok (sorok és oszlopok száma egyenlő). Az *E* szorzatmátrix *[I1,I2]* elemét úgy kapjuk meg, hogy az *M1* mátrix *I1*. sorát megszorozzuk az *M2* mátrix *I2*. oszlopával:

```
E[I1,I2] := M1[I1,1]*M2[1,I2] + M1[I1,2]*M2[2,I2] + ...
            M1[I1,n]*M2[n,I2]
```

Boole mátrix szorzása esetén a szorzás AND, az összeadás pedig OR műveletet jelent. Nézzük a feladat Turbo Pascal megvalósítását. A szomszédossági mátrix áttekinthetőbb, ha *0* és *1* értékekkel adjuk meg, és nem *True*, *False* értékekkel.

```
Program IrGraf ;
Type
  TSzomszed = Array ['a'..'e','a'..'e'] Of Byte ;
  TBoole = Array ['a'..'e','a'..'e'] Of Boolean ;

Const
  Varos : Array ['a'..'e'] Of String[15] =
    ('AFalu','BFalu','CFalu','DFalu','EFalu') ;

  Szomszed : TSzomszed =
    ((0,0,0,1,0),
     (0,0,1,1,1),
     (0,0,0,0,0),
     (0,0,1,0,0),
     (1,0,0,1,0)) ;
```

A szomszédossági mátrix 0 és 1 értékeket tartalmaz. A Boole mátrix szorzásánál viszont akkor kapunk helyes eredményt, ha az AND, illetve az OR műveleteket használjuk. Ezért a mátrixra egy „igazi" Boole (Boolean elemeket tartalmazó) mátrixot definiálunk rá:

```
Procedure SzorozBoole(Matrix1, Matrix2: TSzomszed;
  Var Eredmeny: TSzomszed) ;
  Var
    M1 : TBoole Absolute Matrix1 ;
    M2 : TBoole Absolute Matrix2 ;
    E : TBoole Absolute Eredmeny ;
    C1,C2,C: Char ;
  Begin
    For C1 := 'a' To 'e' Do
      For C2 := 'a' To 'e' Do
      Begin
        E[C1,C2] := False ;
        For C := 'a' To 'e' Do
          E[C1,C2] := E[C1,C2] Or M1[C1,C] And M2[C,C2];
      End ;
  End ;
```

A mátrix hatványozása nem más, mint önmagával való szorzások sorozata.

Eredmény = $Matrix^N$:

```
Procedure HatvBoole(Matrix: TSzomszed; N: Word;
  Var Eredmeny: TSzomszed) ;
  Var
    I : Word ;
  Begin
    Eredmeny := Matrix ;
    For I := 1 To N-1 Do
      SzorozBoole(Eredmeny,Matrix,Eredmeny) ;
  End ;
```

A szomszédossági mátrix kiírása képernyőre:

```
Procedure Kiir(M: TSzomszed) ;
  Var
    C1, C2: Char ;
  Begin
    For C1 := 'a' To 'e' Do
      Begin
        For C2 := 'a' To 'e' Do
          Write(M[C1,C2]) ;
        WriteLn ;
      End ;
  End ;
```

A főprogram kiírja a konstansban megadott szomszédossági mátrixot, és 6 darab hatványt:

```
Var
   E : TSzomszed ;
   N : Byte ;

Begin { Főprogram }
   For N := 1 To 6 Do
      Begin
         WriteLn(N,'. hatvány') ;
         HatvBoole(Szomszed,N,E) ;
         Kiir(E) ;
      End ;
End.
```

A program által képzett hatványok a következők lesznek. Az *n.* hatvány mátrixában egy elem akkor *1*-es, ha a megfelelő falvak közt létezik pontosan *n* hosszúságú útvonal:

1. hatvány	**2. hatvány**	**3. hatvány**	**4. hatvány**	**5. hatvány**	**6. hatvány**
00010	00100	00000	00000	00000	00000
00111	10110	00110	00100	00000	00000
00000	00000	00000	00000	00000	00000
00100	00000	00000	00000	00000	00000
10010	00110	00100	00000	00000	00000

Pontosan 2 hosszú utak:

(a,c): a, d, c

(b,a): b, e, a

(b,c):b, d, c

(b,d):b, e, d

(e,c): e, d, c

(e,d): e, a, d

Pontosan 4 hosszú út egyetlen egy létezik:

(b,c): b, e, a, d, c

Könnyű belátni, hogy egy *n* csomóponti adatot tartalmazó hálós adatmodell akkor hurokmentes, ha a szomszédossági mátrixának *n.* hatványa már 0. Ha ez nem 0, az azt jelenti, hogy létezik egy *n* hosszúságú út, vagyis körbeértünk. A faluk közötti buszjáratokkal tehát nem tudunk „körbe körbe" buszozgatni, hiszen az 5. hatvány 0. Ebből már következik, hogy a 6. és az összes többi hatvány már 0.

Kérdések

1. Mit jelentenek a következő fogalmak?
   a) Adatmodell, eljárásmodell
   b) Gráf, irányított gráf
   c) Asszociatív, szekvenciális, hierarchikus, hálós adatszerkezet
   d) Csomópont, kapcsolat
   e) Absztrakt társzerkezet
   f) Vektor, lista
   g) Általános tömb, ritka mátrix
   h) Jelsorozat, szövegszerkesztés, nyelvtan
   i) Verem, LIFO, Push, Pop
   j) Sor, FIFO, Get, Put
   k) Fa, fa foka, magassága, bejárása
   l) Csomópont foka
   m) Szülő, gyerek, elágazás, levél, szintszám
   n) Rendezett fa, kiegyensúlyozott fa, kiegyensúlyozatlan fa, degenerált fa
   o) Keresőfa
   p) m-utas keresési fa
   q) B-fa
   r) Tábla, soros tábla, önátrendező tábla, rendezett tábla
   s) Rendező fa táblája, kulcstranszformációs tábla
   t) Hálós adatszerkezet, irányított gráf, hálózat
2. Próbáljon felsorolni olyan feladatokat, melyeket a következő adatszerkezetek segítségével lehetne legjobban megoldani:
   a) Tömb
   b) Ritka mátrix
   c) Jelsorozat
   d) Verem
   e) Sor
   f) Fa
   g) Tábla
   h) Hálózat
3. Hogyan lehet egy általános fát tárolni?
4. Hogyan lehet egy bináris fát tárolni?
5. Miért rekurzívak a fával kapcsolatos eljárások?
6. Hogyan lehet egy tábla adatszerkezetet tárolni?
7. Hogyan lehet egy hálós adatszerkezetet tárolni?
8. Hogyan lehet egyértelműen megállapítani egy hálós adatszerkezet hurokmentességét?

Feladatok

1. Olvasson be a billentyűzetről folyamatosan karaktereket! Az <ENTER> megnyomására jelenítse meg az utolsó tíz karaktert!
2. Olvasson be számokat, és írja ki azokat fordítva! A feladatot először verem adatszerkezettel, majd rekurzióval oldja meg!

3*. Írjon egy egységet, mely a vermet valósítja meg a heap-en. Használata minél egyszerűbb és kényelmesebb legyen!

4*. Egy ügyintéző asztalán elintézendő ügyiratok vannak egymásra helyezve. Minden ügyirathoz tartozik egy ügyiratszám, egy téma és egy név. Az ügyiratok sorban érkeznek, s azok mindig a halom tetejére kerülnek. Az ügyintézőnek időnként el kell intéznie egy adott névhez tartozó ügyet. Ehhez meg kell keresnie a megfelelő ügyiratot: felülről sorban leemeli az ettől különböző iratokat, és azokat egyenként egy másik halomra teszi. A megtalált ügyiratot végleg kiveszi a halomból, majd az imént félretetteket egyenként visszahelyezi oda. A végleg kivett aktákat egy külön halomra teszi.

   Szimulálja programmal a fent említett folyamatot! Kérésre adjon egy teljes listát a még el nem intézett, illetve az elintézett ügyiratokról!

5. Készítsen papíron egy bináris fát – tegye fel a fára a következő számokat: 34, 67, 12, 23, 44, 101, 29, 1, 55, 99, 4, 8, 9, 10, 11, 13.
   a) Játssza végig a következő elemek megkeresését, majd törlését: 67, 100, 23, 44, 34.
   b) Készítsen programot úgy, hogy a fát ábrázoló listát először vektorra, majd dinamikus listára képezze le! Ne engedje meg két egyforma szám felvitelét!
   c) Készítsen eljárást, mely egy adott számot megkeres, majd kitöröl a fából!
6. Olvasson be sorban, végjelig neveket és születési éveket! Tárolja az adatokat egy bináris fán születési év szerint csökkenő, s azon belül név szerint növekvő sorrendben!
   a) Írja ki a névsort születési év szerint csoportosítva!
   b) Listázza ki a fát fordított sorrendben!
   c) Törölje ki a fából az 1900 előtt születetteket!

Érdemes tanulmányozni

Mérey András: Adatszerkezetek című könyvét

MEGOLDÁSOK

1. Rekord

1/3. feladat

```
Program Fel3 ;
Type
  TTantargy = String[20] ;
  TNev = String[30] ;
Const
  TantargyakN = 10 ;
  Tantargyak : Array[1..TantargyakN] Of TTantargy =
    ('Angol','Biológia','Fizika','Földrajz','Kémia',
     'Magyar','Matematika','Német','Rajz','Történelem') ;

Function Tantargy(Kod: Byte) : TTantargy ;
  Begin
    If Kod In [1..TantargyakN] Then
      Tantargy := Tantargyak[Kod]
    Else
      Tantargy := '' ;
  End ;

Function TantargyKod(Tantargy: TTantargy) : Byte ;
  Var Kod : Byte ;
  Begin
    Kod := 1 ;
    While (Kod <= TantargyakN) And
      (Tantargy > Tantargyak[Kod]) Do
      Inc(Kod) ;
    If (Kod <= TantargyakN) And
      (Tantargy = Tantargyak[Kod]) Then
      TantargyKod := Kod
    Else
      TantargyKod := 0 ;
  End ;
```

```
Type
  TTanulo = Record
    Nev : TNev ;
    { Osztályzatok: }
    Oszt : Array[1..TantargyakN] Of Byte ;
    { Felvételi tantárgyak száma és a kódok: }
    FelvN : Byte ;
    FelvTant : Array[1..5] Of Byte ;
  End ;

Var
  Tanulok : Array[1..100] Of TTanulo ;
  TanulokN : Byte ;

Procedure Beolvas(Var Tanulo: TTanulo) ;
  Var
    I, Kod : Byte ;
    Nev, Felv : TNev ;
  Begin
    With Tanulo Do
      Begin
        WriteLn('Osztályzatai: ') ;
        For I := 1 To TantargyakN Do
          Begin
            Write(Tantargy(I),' tantárgyból: ') ;
            ReadLn(Oszt[I]) ;
          End ;

        WriteLn('Felvételi tárgyai - végjel = ''*'':') ;
        FelvN := 0 ;
        ReadLn(Felv) ;
        While (FelvN < 5) And (Felv <> '*') Do
          Begin
            Kod := TantargyKod(Felv) ;
            If Kod = 0 Then
              Write(#7)
            Else
              Begin
                Inc(FelvN) ;
                FelvTant[FelvN] := Kod ;
              End ;
            ReadLn(Felv) ;
          End ;
      End ;
  End ;
```

```
Var
   Nev : TNev ;
   I, J : Byte ;

Begin { Főprogram }
   TanulokN := 0 ;
   Write('Név: ') ; ReadLn(Nev) ;
   While Nev <> '' do
      Begin
         Inc(TanulokN) ;
         Tanulok[TanulokN].Nev := Nev ;
         Beolvas(Tanulok[TanulokN]) ;
         Write('Név: ') ; ReadLn(Nev) ;
      End ;
   WriteLn('Tanulók listája') ;
   For I := 1 To TanulokN do
      With Tanulok[I] Do
         Begin
            WriteLn('Név: ',Nev) ;
            WriteLn('Osztályzatok:') ;
            For J := 1 To TantargyakN Do
               Write(Oszt[J]:3) ;
            WriteLn ;
            WriteLn('Felvételi tárgyak:') ;
            For J := 1 To FelvN Do
               Write(Tantargy(FelvTant[J]),' ') ;
            WriteLn ; WriteLn ;
         End ;
End.
```

3. Típusos állomány

3/2. feladat

Az egyes megoldások a Feladat_A, Feladat_B, stb. eljárásban találhatók.

```
Program Fel2 ;
Type
   TSzemely = Record
      Nev : String[20] ;
      Irszam : String[4] ;
      Cim : String[30] ;
      Fizetes : LongInt ;
   End ;
```

```
Var
  Szemelyek : File Of TSzemely ;
  Szemely : TSzemely ;

Procedure Feladat_A ;
  Begin
    Assign(Szemelyek,'Szemely.Dat') ;
    {$I-} Reset(Szemelyek) ; {$I+}
    If IOResult = 0 Then
      Seek(Szemelyek,FileSize(Szemelyek))
    Else
      Rewrite(Szemelyek) ;

    With Szemely Do
      Begin
       Write('Név: ') ; ReadLn(Nev) ;
       While Nev <> '' Do
         Begin
           Write('Irszám : ') ; ReadLn(Irszam) ;
           Write('Cim   : ') ; ReadLn(Cim) ;
           Write('Fizetés: ') ; ReadLn(Fizetes) ;
           Write(Szemelyek,Szemely) ;
           Write('Név   : ') ; ReadLn(Nev) ;
         End ;
      End ;
    Close(Szemelyek) ;
  End ;

Procedure Feladat_B ;
  Begin
    Reset(Szemelyek) ;
    While Not Eof(Szemelyek) Do
      Begin
        Read(Szemelyek,Szemely) ;
        If (Szemely.Nev <> '') And
          (Szemely.Nev[1] = 'B') Then
          WriteLn(Szemely.Nev) ;
      End ;
    Close(Szemelyek) ;
  End ;


Procedure Feladat_C ;
  Var
    NagyFiz : File Of TSzemely ;
```

```
  Begin
    Reset(Szemelyek) ;
    Assign(Nagyfiz,'Nagyfiz.Dat') ;
    Rewrite(Nagyfiz) ;
    While Not Eof(Szemelyek) Do
      Begin
        Read(Szemelyek,Szemely) ;
        If Szemely.Fizetes > 100000 Then
          Write(Nagyfiz,Szemely) ;
      End ;
    Close(Nagyfiz) ;
    Close(Szemelyek) ;
  End ;

Procedure Feladat_D ;
  Var
    Temp : File Of TSzemely ;
  Begin
    Reset(Szemelyek) ;
    Assign(Temp,'Temp.Dat') ;
    Rewrite(Temp) ;
    While Not Eof(Szemelyek) Do
      Begin
        Read(Szemelyek,Szemely) ;
        If Szemely.Irszam[1] = '1' Then
          Write(Temp,Szemely) ;
      End ;
    Close(Szemelyek) ;
    Close(Temp) ;
    Erase(Szemelyek) ;
    Rename(Temp,'Szemely.Dat') ;
  End ;

Procedure Feladat_E ;
  Var
    Fizetes : Array[1..22] Of LongInt ;
    Darab : Array[1..22] Of LongInt ;
    Ker, I : Byte ;
    Kod : Integer ;

  Begin
    For I := 1 To 22 Do
      Begin
        Fizetes[I] := 0 ;
        Darab[I] := 0 ;
      End ;
```

```
    Assign(Szemelyek,'Szemely.Dat') ;
    Reset(Szemelyek) ;
    While Not Eof(Szemelyek) Do
      Begin
        Read(Szemelyek,Szemely) ;
        If Szemely.Irszam[1] = '1' Then
          Begin
            Val(Copy(Szemely.Irszam,2,2),Ker,Kod) ;
            If (Kod = 0) And (Ker In [1..22]) Then
              Begin
                Fizetes[Ker] :=
                  Fizetes[Ker] + Szemely.Fizetes ;
                Inc(Darab[Ker]) ;
              End ;
          End ;
      End ;
    Close(Szemelyek) ;

    For I := 1 To 22 Do
      If Darab[I] <> 0 Then
        WriteLn(I:2,'. ',Fizetes[I]/Darab[I]:12:2) ;
End ;
```

5. Karbantartás

5/3. feladat

A *Mozgas.Dat* itt egy olyan tranzakciós állomány, mely csak módosításokat tartalmaz, ezért tranzakciós kódra nincsen szükség. Mivel a törzsállomány is (indexen keresztül) és a tranzakciós állomány is rendezett számlaszám szerint, ezért ez egy speciális összeválogatásos feladat. Az eredmény állomány marad a törzsállomány, hiszen az se nem bővül, se nem csonkul. A módosításokat „helyben" elvégezzük. A törzsállományt úgy dolgozzuk fel rendezetten, hogy a Számlaindex állományt szekvenciálisan olvassuk, és a törzsből a számlához tartozó index alapján olvasunk. A feladat durva megoldása:

```
Mai dátum beolvasása
Ügyfelek, Tranz, Szamlaindex nyitása
Olvasás Ügyfelek-ből Számlaindex alapján
Olvasás Tranz-ból
Ciklus amíg van Ügyfél rekord és van Tranz rekord
  Elágazás
    Ügyfél.Számlaszám < Tranz.Számlaszám esetén
      Olvasás Ügyfelek-ből Számlaindex alapján
```

```
      Ügyfél.Számlaszám > Tranz.Számlaszám esetén
        Hiba, nincs a tranzakcióhoz ügyfél
        Olvasás Tranz-ból
      Egyéb esetben (Ügyfél.Számlaszám = Tranz.Számlaszám)
        Ha Tranz.Kód = 'B' akkor
          Ügyfél.Összeg = Ügyfél.Összeg + Tranz.Összeg
        egyébként
          Ügyfél.Összeg = Ügyfél.Összeg - Tranz.Összeg
        Elágazás vége
        Ügyfél.Dátum := Mai dátum
        Ügyfél visszaírása ugyanarra a helyre
    Elágazás vége
  Ciklus vége
  Állományok lezárása
```

6. Csoportváltás

6/1. feladat

```
Program Fel1 ;
Const
  N = 20 ;
  T : Array[1..N] Of Word =
  (1,1,1,3,4,4,4,4,4,4,4,4,6,6,6,9,9,9,9,9) ;

Var
  Szam : Word ;
  Ossz : Word ;
  I : Word ;

Begin
  I := 1 ;
  While I <= 20 Do
    Begin
      Szam := T[I] ;
      Ossz := 0 ;
      While (I <= 20) And (T[I] = Szam) Do
        Begin
          Inc(Ossz) ;
          Inc(I) ;
        End ;
      WriteLn(Szam:3,': ',Ossz:3) ;
    End ;
End.
```

6/2. feladat

```
Program Fel2 ;
Var
   T : Array[1..50] Of String[20] ;
   Nev, S : String[20] ;
   N, I, J : Word ;
   Betu : Char ;

Begin
   Write('Név: ') ; ReadLn(Nev) ;
   N := 0 ;
   While (N < 50) And (Nev <> '*') Do
      Begin
         Inc(N) ;
         T[N] := Nev ;
         Write('Név: ') ; ReadLn(Nev) ;
      End ;

   For I := 1 To N-1 Do
      For J := I+1 To N Do
         If T[J] < T[I] Then
            Begin
               S := T[I] ;
               T[I] := T[J] ;
               T[J] := S ;
            End ;

   I := 1 ;
   While I <= N Do
      Begin
         Betu := T[I][1] ;
         WriteLn(Betu) ;
         While (I <= N) And (T[I][1] = Betu) Do
            Begin
               WriteLn(T[I]) ;
               Inc(I) ;
            End ;
         WriteLn ;
      End ;
End.
```

7. Szöveges állományok

7/1. feladat

Ha az egyes mondatok hosszúságai nem haladnák meg a 255 karaktert, akkor a megoldás jóval egyszerűbb lenne, hiszen akkor a mondatokat egy-egy String típusú változóba olvasnánk, és ha értéke nem az üres lánc vagy a szóköz, felírnánk az új állományba. A mondatokat így kénytelenek vagyunk karakterenként kezelni:

```
Program Fel1 ;
Uses Crt ;
Var
   T : Text ;
   Kar : Char ;
Begin
   Assign(T,'Szoveg.Txt') ;
   {$I-} Append(T) ; {$I+}
   If IOResult <> 0 Then
      Rewrite(T) ;
   Kar := ReadKey ;
   While Kar <> #26 Do
      Begin
         While (Kar <> #13) And (Kar <> #26) Do
            Begin
               Write(T,Kar) ;
               Write(Kar) ;
               Kar := ReadKey ;
            End ;
         WriteLn(T,' Stop') ;
         WriteLn ;
         Kar := ReadKey ;
      End ;
   Close(T) ;
End.
```

7/2. feladat

Ugyanúgy, mint az előző feladatnál, a helyzetet itt is bonyolítják a hosszú mondatokra való felkészülés:

```
Program Fel2 ;

Var
   T, Uj : Text ;
   Kar : Char ;
```

```
Begin
  Assign(T,'Szoveg.Txt') ;
  Reset(T) ;
  Assign(Uj,'Uj.Txt') ;
  Rewrite(Uj) ;
  While Not Eof(T) Do
    Begin
      If EoLn(T) Then
        ReadLn(T)
      Else
        Begin
          Read(T,Kar) ;
          If Kar = ' ' Then
            ReadLn(T)
          Else
            Begin
              Write(Uj,Kar) ;
              While Not EoLn(T) And Not Eof(T) Do
                Begin
                  Read(T,Kar) ;
                  Write(Uj,Kar) ;
                End ;
              ReadLn(T) ;
              WriteLn(Uj) ;
            End ;
        End ;
    End ;
  Close(T) ;
  Close(Uj) ;
  Erase(T) ;
  Rename(Uj,'szoveg.Txt') ;
End.
```

7/5. feladat

```
Program Fel5 ;
Var
  T : Text ;
  Szam, Osszeg : Real ;
  Sorsz, Db : Word ;

Begin
  Assign(T,'Szamok.Txt') ;
  Reset(T) ;
  Sorsz := 0 ;
```

```
    WriteLn('Az átlagok:') ;
    While Not Eof(T) Do
      Begin
        Inc(Sorsz) ;
        Osszeg := 0 ;
        Db := 0 ;
        While Not EoLn(T) And Not Eof(T) Do
          Begin
            Read(T,Szam) ;
            Osszeg := Osszeg + Szam ;
            Inc(Db) ;
          End ;
        ReadLn(T) ;
        Write(Sorsz:3,': ') ;
        If Db = 0 Then
          WriteLn('üres sor')
        Else
          WriteLn(Osszeg/Db:10:2) ;
      End ;
    Close(T) ;
End.
```

8. Memóriakezelés

8/1. feladat

```
Program Fel1 ;
Var
  Elso : Char ;
  Ev : Word ;
  Nyelv : String[6] ;
  Ofszet : Word ;
  Ok : Boolean ;
  Kar : Char ;

Begin
  Elso := '1' ;
  Ev := 1995 ;
  Nyelv := 'Pascal' ;
  Ok := True ;

  For Ofszet := Ofs(Elso) To Ofs(Kar) Do
    Write(Ofszet:4) ;
  WriteLn ;
```

```
    For Ofszet := Ofs(Elso) To Ofs(Kar) Do
      Write(Mem[DSeg:Ofszet]:4) ;
    WriteLn ;

    For Ofszet := Ofs(Elso) To Ofs(Kar) Do
      Begin
        Kar := Char(Mem[DSeg:Ofszet]) ;
        If Kar In [#27..#255] Then
          Write(Kar:4)
        Else
          Write('.':4) ;
      End ;
    WriteLn ;
  End.
```

A fordítót utasítani lehet arra, hogy az 1 byte-nál nagyobb helyet elfoglaló változókat szóhatárra tegye (Word Align data, $A+). A program által produkált memória dump $A- illetve $A+ melletti fordítás után:

```
 82  83  84  85  86  87  88  89  90  91  92  93  94 (Ofszet címek)

 49 203   7   6  80  97 115  99  97 108  92   0   1   0
  1   -   .   .   P   a   s   c   a   l   \   .   .   .

 49   0 203   7   6  80  97 115  99  97 108   0  94   0   1   0
  1   .   -   .   .   P   a   s   c   a   l   .   ^   .   .   .
```

8/2. feladat

```
  Procedure Ir(X, Y: Byte; K: Char; Szin: Byte) ;
    Type
      Karhely = Record
        Kar: Char ;
        Attr: Byte ;
      End ;

    Var
      Kep : Array[1..25,1..80] Of Karhely Absolute $B800:0 ;

    Begin
      With Kep[Y,X] Do
        Begin
          Kar := K ;
          Attr := Attr And $F0 Or Szin ;
        End ;
    End ;
```

Az eljárás hívása, ha az 1. sor 5. oszlopába szeretnénk egy piros kukacot tenni:

```
Ir(5,1,'@',Red) ;
```

8/5. feladat

```
Procedure SorCsere(N,M: Byte) ;
  Type
    TSor = Array[1..80] Of Word ;

  Var
    Kep : Array[1..25] Of TSor Absolute $B800:0 ;

    Sor : TSor ;

  Begin
    Sor := Kep[N] ;
    Kep[N] := Kep[M] ;
    Kep[M] := Sor ;
  End ;

Procedure Kepcsere ;
  Var
    I : Byte ;
  Begin
    For I := 1 To 12 Do
      SorCsere(I,26-I) ;
  End ;
```

8/6. feladat

A képernyőt típusnélküli állományba[1] mentjük. Az állományt 4000 byte blokkmérettel nyitjuk meg, és összesen 1 blokknyi adatot írunk illetve olvasunk egyszerre. A puffert a képernyő elejére állítjuk

```
Program Fel6 ;

Procedure KepMent(KepNeve: String) ;
  Var
    F : File ;

  Begin
```

[1] A típusnélküli állományokat többek között az Angster-Kertész Feladatgyűjtemény I.-ben, illetve a referenciakönyvben tanulmányozhatja az Olvasó.

```
    Assign(F,KepNeve) ;
    Rewrite(F,4000) ;
    BlockWrite(F,Ptr($B800,0)^,1) ;
    Close(F) ;
  End ;

Procedure KepTolt(KepNeve: String) ;
  Var
    F : File ;
  Begin
    Assign(F,KepNeve) ;
    Reset(F,4000) ;
    BlockRead(F,Ptr($B800,0)^,1) ;
    Close(F) ;
  End ;

Begin
  KepMent('Eredeti.Kep') ;
  WriteLn('Mindjárt visszatöltöm az eredetit - ENTER!') ;
  ReadLn ;
  KepTolt('Eredeti.Kep') ;
  ReadLn ;
End.
```

8/11. feladat

```
Program Fel11 ;
Type
  TNevek = Array[1..1000] Of String[40] ;
Var
  Nevek : TNevek ;
  Ment : ^TNevek ;

Begin
  { Nevek tömb változtatása...}
  { Egy adott állapot elmentése: }
  New(Ment) ;
  Ment^ := Nevek ;
  { Nevek tömb változtatása...}
  {...}
  { Az elmentett állapot visszatöltése: }
  Nevek := Ment^ ;
  {...}
  Dispose(Ment) ;
End.
```

9. Dinamikus lista

9/3. feladat

```
Program Fel3 ;

Type
   PElem = ^TElem ;
   TElem = Record
      Kov : PElem ;
      Szam : LongInt ;
   End ;

Var
   Elso : PElem ;

Function Utolso: PElem ;
   Var
      Akt : PElem ;
   Begin
      If Elso = Nil Then
         Utolso := Nil
      Else
         Begin
            Akt := Elso ;
            While Akt^.Kov <> Nil Do
               Akt := Akt^.Kov ;
            Utolso := Akt ;
         End ;
   End ;

Function Elozo(Akt: PElem): PElem ;
   Var
      E, A : PElem ;
   Begin
      E := Nil ;
      A := Elso ;
      While A <> Akt Do
         Begin
            E := A ;
            A := A^.Kov ;
         End ;
      Elozo := E ;
   End ;
```

```
{ Listázás visszafelé: }
Procedure ListaVissza ;
  Var
    Akt : PElem ;
  Begin
    WriteLn('Visszafelé: ') ;
    Akt := Utolso ;
    While Akt <> Nil Do
      Begin
        Write(Akt^.Szam:5) ;
        Akt := Elozo(Akt) ;
      End ;
    WriteLn ;
  End ;

{ Lista megfordítása: }
Procedure Fordit ;
  Var
    Akt, Kov, Elozo: PElem ;
  Begin
    Elozo := Nil ;
    Akt := Elso ;
    While Akt <> Nil Do
      Begin
        Kov := Akt^.Kov ;
        Akt^.Kov := Elozo ;
        Elozo := Akt ;
        Akt := Kov ;
      End ;
    Elso := Elozo ;
  End ;

Var
  Uj : PElem ;
  Szam : LongInt ;

Begin
  { Elemek felfűzése: }
  Elso := Nil ;
  ReadLn(Szam) ;
  While Szam <> 0 Do
    Begin
      New(Uj) ;
      Uj^.Szam := Szam ;
```

```
        If Elso = Nil Then
          Elso := Uj
        Else
          Utolso^.Kov := Uj ;
        Uj^.Kov := Nil ;
        Utolso := Uj ;
        ReadLn(Szam) ;
      End ;

  ListaVissza ;
  Fordit ;
  ListaVissza ;
End.
```

9/5. feladat

```
Program Fel5 ;

Type
  TNev = String[127] ;
  PElem = ^TElem ;
  TElem = Record
    Kov : PElem ;
    Nev : TNev ;
  End ;

Var
  Fej : PElem ;

Procedure Lista ;
  Var
    Akt : PElem ;
  Begin
    WriteLn('Nevek:') ;
    Akt := Fej^.Kov ;
    While Akt <> Fej Do
      Begin
        WriteLn(Akt^.Nev) ;
        Akt := Akt^.Kov ;
      End ;
  End ;

Procedure Beszur(Nev: TNev) ;
  Var
    Uj, Elozo, Akt : PElem ;
```

```
  Begin
    New(Uj) ;
    Uj^.Nev := Nev ;
    Elozo := Fej ;
    Akt := Fej^.Kov ;
    While (Akt <> Fej) And (Nev > Akt^.Nev) Do
      Begin
        Elozo := Akt ;
        Akt := Akt^.Kov ;
      End ;
    Elozo^.Kov := Uj ;
    Uj^.Kov := Akt ;
  End ;

Procedure Betolt ;
  Var
    F : Text ;
    Nev : TNev ;
    Akt : PElem ;
  Begin
    New(Fej) ;
    Fej^.Kov := Fej ;
    Assign(F,'Nevek.Txt') ;
    {$I-} Reset(F) ; {$I+}
    If IOResult = 0 Then
      Begin
        While Not Eof(F) Do
          Begin
            ReadLn(F,Nev) ;
            Beszur(Nev) ;
          End ;
        Close(F) ;
      End ;
  End ;

Procedure Elment ;
  Var
    F : Text ;
    Nev : TNev ;
    Akt : PElem ;
  Begin
    Assign(F,'Nevek.Txt') ;
    Rewrite(F) ;
    Akt := Fej^.Kov ;
```

```
    While Akt <> Fej Do
      Begin
        WriteLn(F,Akt^.Nev) ;
        Akt := Akt^.Kov ;
      End ;
    Close(F) ;
  End ;

Procedure Hozzafuz ;
  Var
    Nev : TNev ;
  Begin
    Write('Név: ') ; ReadLn(Nev) ;
    While Nev <> '*' Do
      Begin
        Beszur(Nev) ;
        Write('Név: ') ; ReadLn(Nev) ;
      End ;
  End ;

Function Leghosszabb : TNev ;
  Var
    Akt : PElem ;
    Max : Byte ;
    Nev : TNev ;
  Begin
    Max := 0 ;
    Akt := Fej^.Kov ;
    While Akt <> Fej Do
      Begin
        If Length(Akt^.Nev) > Max Then
          Begin
            Nev := Akt^.Nev ;
            Max := Length(Akt^.Nev) ;
          End ;
        Akt := Akt^.Kov ;
      End ;
    Leghosszabb := Nev ;
  End ;

Procedure Torol ;
  Var
    Elozo, Akt, Torlendo : PElem ;
```

```
  Begin
    Elozo := Fej ;
    Akt := Fej^.Kov ;
    While Akt <> Fej Do
      Begin
        If Length(Akt^.Nev) < 10 Then
          Begin
            Torlendo := Akt ;
            Elozo^.Kov := Akt^.Kov ;
            Akt := Akt^.Kov ;
            Dispose(Torlendo) ;
          End
        Else
          Begin
            Elozo := Akt ;
            Akt := Akt^.Kov ;
          End ;
      End ;
  End ;

Begin
  Betolt ;
  Hozzafuz ;
  Lista ;
  WriteLn('A leghosszabb: ',Leghosszabb) ;
  Torol ;
  Lista ;
  Elment ;
End.
```

10. Programszegmentálás, kapcsolat az operációs rendszerrel

10/2. feladat

Pascal-ban nem lehet új műveletet definiálni, így a komplex számokkal való műveleteket függvényhívásokkal kell megoldanunk. Mivel a függvény csak egy értéket tud visszaadni, a komplex szám pedig egy rendezett (Valós, Képzetes) számpár, ezért egy trükköt kell kitalálni e két érték összeötvözésére. Legyenek a komplex számok *String[12]* típusúak! A karakterlánc első 6 karaktere lesz a valós rész, a második hat pedig a képzetes. Számolásnál erre a karakterláncra egyszerűen rádefiniálunk egy rekordot, mely segítségével a két rész valós számként is elérhető. engedélyezzük a koprocesszor használatát, mert a tudósoknak méréseknél fontos a gyorsaság:

```
{$N+,E+}
Unit UKomplex ;
Interface

Type
   TPart = Real ; {Adott esetben pl. Integer-re cserélhető}
   TComplex = String[12] ;

Function  Co(R,I:TPart): TComplex ; {Értékadás}
Function  Re(A:TComplex): TPart ; {Valós (real) rész}
Function Im(A:TComplex): TPart; {Képzetes (imaginárius) rész}
Function  Ca(A,B: TComplex): TComplex ; {Összeadás (add)}
Function  Cs(A,B: TComplex): TComplex ; {Kivonás (subtract)}
Function  Cm(A,B: TComplex): TComplex ; {Szorzás (multilpy)}
Function  Cd(A,B: TComplex): TComplex ; {Osztás (division)}
Function  Cabs(A: TComplex): TPart ; {Abszolut érték (abs)}
Procedure Cwrite(A: TComplex) ; {Komplex szám kiírása}

Implementation
Type
   TComp = Record
      B   : Byte ;
      R,I : TPart ;
   End ;

Const
   C : TComp = (B:12; R:0; I:0) ;
Var
   S : TComplex Absolute C ;

Function  Co(R,I: TPart): TComplex ;
   Begin
      C.R := R ;
      C.I := I ;
      Co := S ;
   End ;

Function  Re(A: TComplex): TPart ;
   Begin
      Re := TComp(A).R ;
   End ;

Function  Im(A: TComplex): TPart ;
   Begin
      Im := TComp(A).I ;
   End ;
```

```
Function Ca(A,B: TComplex): TComplex ;
  Begin
    C.R := TComp(A).R + TComp(B).R ;
    C.I := TComp(A).I + TComp(B).I ;
    Ca := S ;
  End ;

Function Cs(A,B: TComplex): TComplex ;
  Begin
    C.R := TComp(A).R - TComp(B).R ;
    C.I := TComp(A).I - TComp(B).I ;
    Cs := S ;
  End ;

Function Cm(A,B: TComplex): TComplex ;
  Begin
    C.R :=
      TComp(A).R * TComp(B).R - TComp(A).I * TComp(B).I ;
    C.I :=
      TComp(A).R * TComp(B).I + TComp(A).I * TComp(B).R ;
    Cm := S ;
  End ;

Function Cd(A,B: TComplex): TComplex ;
  Begin
    C.R :=
    (TComp(A).R * TComp(B).R + TComp(A).I * TComp(B).I) /
    (Sqr(TComp(B).R) + Sqr(TComp(B).I)) ;
    C.I :=
    (TComp(B).R * TComp(A).I - TComp(A).R * TComp(B).I) /
    (Sqr(TComp(B).R) + Sqr(TComp(B).I)) ;
    Cd := S ;
  End ;

Function Cabs(A: TComplex): TPart ;
  Begin
    Cabs := Sqrt(Sqr(TComp(A).R) + Sqr(TComp(A).I)) ;
  End ;

Procedure Cwrite(A: TComplex) ;
  Begin
    WriteLn(Re(A):10:3,Im(A):10:3) ;
  End ;

End.
```

A következő program az előbb definiált *UKomplex* egységet használja:

```
Program Fel2 ;
Uses UKomplex ;

Var
  A, B,
  Osszeg, Kulonbseg, Szorzat : TComplex ;

Begin
  A := Co(2,1) ;
  B := Co(4,1) ;
  Osszeg := Ca(A,B) ;
  Kulonbseg := Cs(A,B) ;
  Szorzat := Cm(A,B) ;
  Write('Osszeg = ') ; Cwrite(Osszeg) ;
  Write('Különbség = ') ; Cwrite(Kulonbseg) ;
  Write('Szorzat = ') ; Cwrite(szorzat) ;
  ReadLn ;
End.
```

10/7. feladat

```
{$N+,E+}
Program Fel7 ;
Uses Dos ;
Var
  Ora, Perc, Mp, Mp100 : Word ;
  Kezd, Veg, I : LongInt ;
  A : Extended ;

Function Ido : LongInt ;
  Begin
    GetTime(Ora,Perc,Mp,Mp100) ;
    Ido := Mp100+100*(Mp+60*(Perc+LongInt(60)*Ora)) ;
  End ;

Begin
  A := 1 ;
  Kezd := Ido ;
  For I := 1 To 1000000 Do A := A*1.001 ;
  Veg := Ido ;
  Write((Veg-Kezd) Div 100,' mp, ',
    (Veg-Kezd) Mod 100,' század mp') ;
End.
```

11. Adatszerkezetek

11/3. feladat

Az egységet használó programnak a vermet TVerem típusúnak kell definiálnia. A vermet használat előtt kötelező inicializálni, s ott megadni a verembe teendő elemek méretét. Használat után a vermet fel kell számolni.

```
Unit UVerem ;

Interface

Type
   PVeremElem = ^TVeremElem ;
   TVeremElem = Record
      Alatta : PVeremElem ;
      ElemP : Pointer ;
   End ;

   TVerem = Record
      Felso : PVeremElem ;
      Meret : Word ;
   End ;

{ Verem inicializálása, elem méretének megadása: }
Procedure Init(Var Verem: TVerem; Meret: Word) ;
{ True, ha a verembe nem fér több elem (nincs hely a
heap-ben): }
Function Tele(Verem: TVerem): Boolean ;
{ True, ha a veremben egy elem sincs: }
Function Ures(Verem: TVerem): Boolean ;
{ Egy elem betétele a verem tetejére: }
Procedure Push(Var Verem: TVerem; Var Elem) ;
{ egy elem kivétele a verem tetejéről: }
Procedure Pop(Var Verem: TVerem; Var Elem) ;
{ Verem felszámolása: }
Procedure Done(Var Verem: TVerem) ;

Implementation

Procedure Init(Var Verem: TVerem; Meret: Word) ;
   Begin
      Verem.Felso := Nil ;
      Verem.Meret := Meret ;
   End ;
```

```
Function Tele(Verem: TVerem): Boolean ;
  Begin
    Tele := MaxAvail < Verem.Meret ;
  End ;

Function Ures(Verem: TVerem): Boolean ;
  Begin
    Ures := Verem.Felso = Nil ;
  End ;

Procedure Push(Var Verem: TVerem; Var Elem) ;
  Var
    Uj : PVeremElem ;
  Begin
    If Not Tele(Verem) Then
      Begin
        New(Uj) ;
        GetMem(Uj^.ElemP,Verem.Meret) ;
        Move(Elem,Uj^.ElemP^,Verem.Meret) ;
        Uj^.Alatta := Verem.Felso ;
        Verem.Felso := Uj ;
      End ;
  End ;

Procedure Pop(Var Verem: TVerem; Var Elem) ;
  Var
    Torlendo : PVeremElem ;
  Begin
    If Not Ures(Verem) Then
      With Verem Do
        Begin
          Torlendo := Felso ;
          Felso := Felso^.Alatta ;
          Move(Torlendo^.ElemP^,Elem,Verem.Meret) ;
          FreeMem(Torlendo^.ElemP,Meret) ;
          Dispose(Torlendo) ;
        End ;
  End ;

Procedure Done(Var Verem: TVerem) ;
  Var
    Elem : Pointer ;
  Begin
    GetMem(Elem,Verem.Meret) ;
```

```
    While Not Ures(Verem) Do
      Pop(Verem,Elem) ;
    FreeMem(Elem,Verem.Meret) ;
  End ;

End.
```

11/4. feladat

Az előző pontban megalkotott UVerem egységet használjuk:

```
Program Fel4 ;

Uses UVerem ;

Type
  TAkta = Record
    Ugyiratszam : Word ;
    Nev : String[30] ;
    Tema : String[10] ;
  End ;

Var
  Elintezett,
  Elintezendo : TVerem ;

Procedure Megjelenit(Verem: TVerem) ;
  Var
    Id : TVerem;
    Akta : TAkta ;
  Begin
    Init(Id,SizeOf(TAkta)) ;
    While Not Ures(Verem) Do
      Begin
        Pop(Verem,Akta) ;
        WriteLn(Akta.Ugyiratszam) ;
        Push(Id,Akta) ;
      End ;
    While Not Ures(Id) Do
      Begin
        Pop(Id,Akta) ;
        Push(Verem,Akta) ;
      End ;
  End ;
```

```
Procedure Elintez ;
  Var
    Id : TVerem ;
    Akta : TAkta ;
    Ugyiratszam : Word ;
    Megvan : Boolean ;
  Begin
    Init(Id,SizeOf(TAkta)) ;
    Write('Mit intézzek? ') ;
    ReadLn(Ugyiratszam) ;
    Megvan := False ;
    While Not Megvan And Not Ures(Elintezendo) Do
      Begin
        Pop(Elintezendo,Akta) ;
        Megvan := Akta.Ugyiratszam = Ugyiratszam ;
        If Megvan Then
          Push(Elintezett,Akta)
        Else
          Push(Id,Akta) ;
      End ;

    While Not Ures(Id) Do
      Begin
        Pop(Id,Akta) ;
        Push(Elintezendo,Akta) ;
      End ;
  End ;

Var
  Ugyiratszam : Word ;
  Akta : TAkta ;

Begin { Főprogram }
  Init(Elintezett,SizeOf(TAkta)) ;
  Init(Elintezendo,SizeOf(TAkta)) ;
  Write('Ügyiratszám:') ;
  ReadLn(Ugyiratszam) ;
  While Ugyiratszam <> 0 Do
    Begin
      Akta.Ugyiratszam := Ugyiratszam ;
      Push(Elintezendo,Akta) ;
      Write('Ügyiratszám:') ;
      ReadLn(Ugyiratszam) ;
    End ;
```

```
  While Not Ures(Elintezendo) Do
    Begin
      WriteLn('Elintezendo') ;
      Megjelenit(Elintezendo) ;
      WriteLn('Elintezett') ;
      Megjelenit(Elintezett) ;
      Elintez ;
    End ;

  WriteLn('Elintezendo') ;
  Megjelenit(Elintezendo) ;
  WriteLn('Elintezett') ;
  Megjelenit(Elintezett) ;
End.
```

Ajánlott irodalom

(1) Angster Erzsébet, Kertész László: Turbo Pascal 6.0
Magánkiadás, 1993

(2) Angster Erzsébet, Kertész László: Turbo Pascal 6.0 'A'.. 'Z'
Magánkiadás, 1993

(3) Angster Erzsébet, Kertész László: Turbo Pascal Feladatgyűjtemény I., II.
Magánkiadás, 1993

(4) Bánné dr. Varga Gabriella: Programtervezési gyakorlatok Jackson módszer szerint
SZÁMALK, 1989

(5) Benkőné - Benkő - Tóth - Varga: Programozzunk Turbo Pascal nyelven
ComputerBooks, 1993

(6) Benkőné - Benkő - Meszéna - Gyenes: Programozási feladatok és algoritmusok
ComputerBooks, 1997

(7) Herneczki Katalin - Orbán Katalin: A Jackson-féle programtervezési módszer
Tudományszervezési és Informatikai Intézet, 1982

(8) Jodál Endre: Informatikai alapszókincs, Angol-magyar szótár
Czédrus, 1993

(9) Knuth: A számítógép-programozás művészete 1.-3.
Addison-Wesley – Műszaki könyvkiadó, 1987

(10) Kris Jamsa - Steven Nameroff: Turbo Pascal Programozói könyvtár
McGraw-Hill – Novotrade, 1987

(11) Lipschutz, Seymour: Adatszerkezetek
Panem – McGraw-Hill, 1993

(12) Mérey András: Adatszerkezetek
KSH SZÁMOK, 1979

(13) Mérey András: Programtervezés Jackson módszerrel
SZÁMALK, 1983

(14) Morvay János - dr. Sebők Ferenc: Az adatkezelés módszertani alapjai
SZÁMALK, 1982

(15) Morvay János - dr. Sebők Ferenc: Számítógépes adatkezelés
KSH SZÁMOK, 1981

(16) OXFORD Számítástechnikai értelmező szótár
Novotrade, 1989

(17) R. Baumgartner - S.Hansjakob - W.Praxl: Turbo Pascal Elmélet és gyakorlat
IWT – Novotrade, 1985

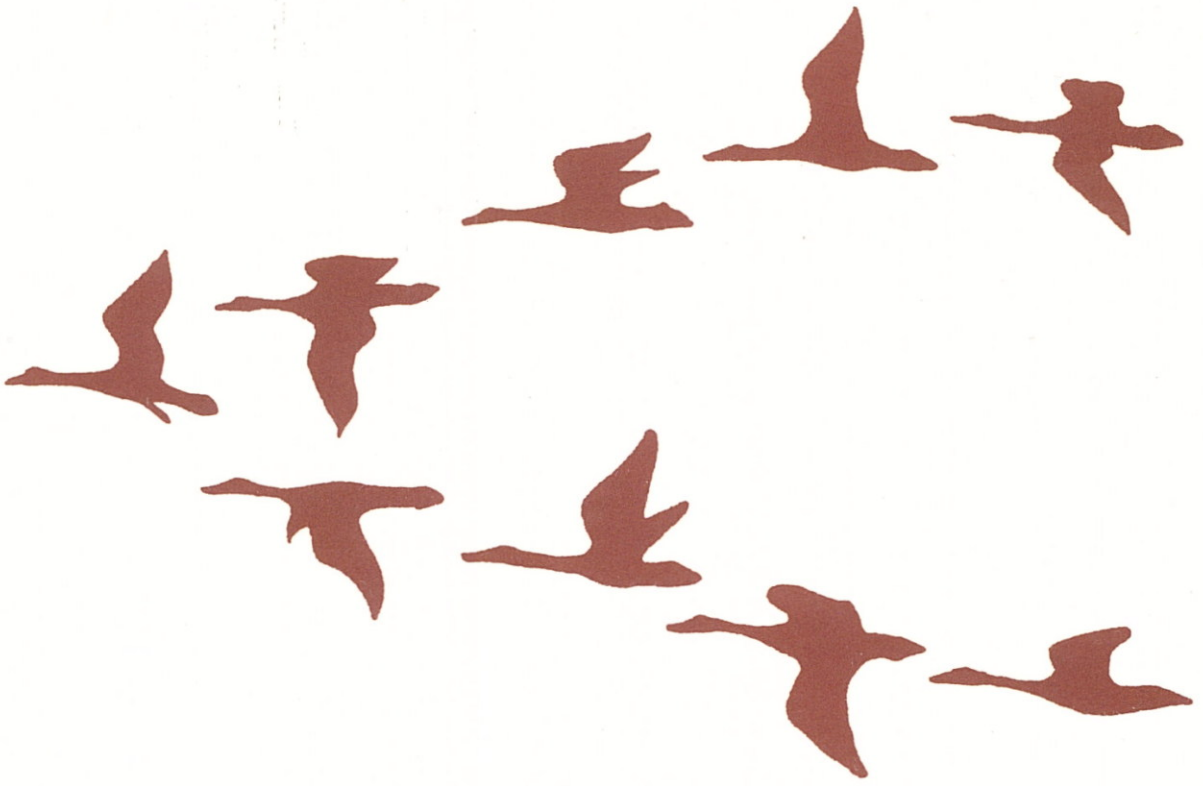
(18) Wirth, Niklaus: Algoritmusok+Adatstruktúrák = Programok
Műszaki könyvkiadó, 1982

Tárgymutató

A könyvben tárgyalt feladatok forráskódjai megtalálhatók az Interneten a **www.gdf.hu/Segedletek.htm** lapon, vagy a **www.borland.hu** címen a **könyvek** oldalon.

Figyelmébe ajánljuk a következő könyveket:

- Angster Erzsébet:
  Az objektumorientált tervezés és programozás alapjai
  (UML, Turbo Pascal, C++)
- Angster Erzsébet – Kertész László:
  Turbo Pascal 6.0 'A'.. 'Z'
  (referencia zsebkönyv)
- Angster Erzsébet – Kertész László:
  Turbo Pascal feladatgyűjtemény I., II.



ISBN 963-4509-57-6

9 789634 509578