Értékünk az ember
Humánerőforrás-fejlesztési Operatív Program

Pusztai Pál

ALGORITMUSOK ÉS ADATSTRUKTÚRÁK

SZÉCHENYI ISTVÁN EGYETEM
GYŐR

Magyarország célba ér
NEMZETI FEJLESZTÉSI TERV

Készült a HEFOP 3.3.1-P.-2004-09-0102/1.0 pályázat támogatásával.

Szerző: Pusztai Pál
egyetemi adjunktus

Lektor: Pukler Antal
egyetemi adjunktus

Varjasi Norbert
egyetemi adjunktus

Marton László emlékére

A dokumentum használata

Mozgás a dokumentumban

A dokumentumban való mozgáshoz a Windows és az Adobe Reader megszokott elemeit és módszereit használhatjuk.

Minden lap tetején és alján egy navigációs sor található, itt a megfelelő hivatkozásra kattintva ugorhatunk a használati útmutatóra, a tartalomjegyzékre, valamint a tárgymutatóra. A ◀ és a ▶ nyilakkal az előző és a következő oldalra léphetünk át, míg a Vissza mező az utoljára megnézett oldalra visz vissza bennünket.

Pozícionálás a könyvjelzőablak segítségével

A bal oldali könyvjelző ablakban tartalomjegyzékfa található, amelynek bejegyzéseire kattintva az adott fejezet/alfejezet első oldalára jutunk. Az aktuális pozíciónkat a tartalomjegyzékfában kiemelt bejegyzés mutatja.

A tartalomjegyzék és a tárgymutató használata

Ugrás megadott helyre a tartalomjegyzék segítségével

Kattintsunk a tartalomjegyzék megfelelő pontjára, ezzel az adott fejezet első oldalára jutunk.

Keresés a szövegben

A dokumentumban való kereséshez használjuk megszokott módon a Szerkesztés menü Keresés parancsát. Az Adobe Reader az adott pozíciótól kezdve keres a szövegben.

Tartalomjegyzék

1. Előszó

Az algoritmusokat az emberiség már jóval azelőtt ismerte és használta, mielőtt az őket végrehajtani tudó számítógépeket megépítette.

Noha az algoritmus szó eredete egy IX. századi perzsa matematikus (Al-Hvárizmi) nevéhez kötődik [Knu 87] [Sai 86], az algoritmusokat – mint egy feladat megoldásához vezető lépések sorozatát – már időszámításunk előtt is használták.

Sokáig csak papíron, „kézzel" volt lehetőség az algoritmusok „végrehajtására", ma már számítógépek végzik ezt helyettünk.

Az univerzális, magasszintű programozási nyelvek [Nyé 03] megjelenése óta olyan eszközrendszert használhatunk algoritmusaink leírására, a feladatunkat megoldó program megírására, amelyek közel állnak az emberi gondolkodás-, és jelölésmódhoz.

Természetesen ezeket a programokat közvetlenül nem értik meg a számítógépek, le kell őket „fordítani" az adott számítógép utasításkészletére, és csak ezután hajthatók végre, futtathatók le.

Könyvünkben bevezetett adatstruktúrák és algoritmus megadási módszerek nem egy konkrét programozási nyelvhez kötődnek, több nyelvből (C, Pascal, Basic) lettek „összegyúrva", kiemelve a közös, általános részeket, így lehetőségünk van az algoritmusok „nyelv-független" lényegére koncentrálni.

Algoritmusainkat strukturált módon készítjük, azaz csak a szekvencia, szelekció, iteráció vezérlőelemekből építkezünk, ugró utasítások nélkül, így megoldásaink áttekinthetők és könnyen programozhatók lesznek.

Amellett, hogy bemutatjuk és elemezzük a kiválasztott algoritmusokat, elsődleges célunk az algoritmikus feladatmegoldó készség kialakítása, fejlesztése, a logikus gondolkodásra való „nevelés".

Ha egy algoritmushoz olyan adatszerkezetet használunk, amelyet a programírásra szánt nyelv nem támogat (pl. a C nyelvben nincsenek halmazok), akkor kitérünk a megvalósítás lehetőségeire.

A megoldások, noha nem úgy terveztük őket, akár objektumorientáltan is programozhatók, kihasználva ezzel az egységbezárás előnyeit. Feladataink azonban többnyire kevés közös részt tartalmaznak, ezért az objektumorientált programozás igazi erejét adó öröklés, így annak előnyei nem érvényesíthetők.

Egy univerzális programozási nyelv ismeretében algoritmusaink programmá írhatók, kipróbálhatók, hiszen a „visszacsatolás”, a megoldás helyességének számítógéppel történő ellenőrzése a tanulási folyamat szerves része. Ezt a tantárgyunktól független, önálló programozást megkönnyítendő, a jegyzetben szereplő megoldásokat C és Pascal nyelven programoztuk, amelyekhez a Borland cég Turbo C és Turbo Pascal fejlesztőrendszerét használtuk. A forrásprogramok a függelékben megtalálhatók.

Ismerve az egyedül elvégzett, önálló munka hasznosságát és maradandóságát, fejezeteink végén egy csokor, az adott témához kapcsolódó feladatot tűzünk ki, amelyből kedvére válogathat a gyakorlásra vágyó, tudását lemérni kívánó hallgató, olvasó.

Jegyzetünk a BSc képzésű, műszaki informatika szakos hallgatók *Algoritmusok és adatstruktúrák* című tantárgyához készült, amelyet a második félévben, heti két órában tanulnak a nappali tagozaton. Példáinkat ennek megfelelően válogattuk, szem előtt tartva az összeállított tananyag 30 órában történő taníthatóságát.

Hallgatóink az első félévtől kezdődően három féléven keresztül (heti 3-5 órában) tanulják a *Programozás* című tantárgyat, amely két félév C és egy félév objektumorientált Java programozást tartalmaz.
Tantárgyunk átfed tehát a C programozás második félévével, így

- Könnyebb a bevezető fejezetek tanítása.
- Megoldásainkat a hallgatók C nyelven programozni tudják, így nemcsak kipróbálhatják, ellenőrizhetik azokat, hanem egyben „anyagot” is kapnak a C nyelvű programozás gyakorlásához.
- A nehezebb részeket (pl. mutatók, dinamikus adatstruktúrák) közel egy időben tárgyalja a két tantárgy, megkönnyítve ezzel a megértésüket.

Tudjuk, hogy a rendelkezésre álló időkeret nem engedi meg az összes feladat részletes tárgyalását, de bízunk benne, hogy a jegyzet anyaga tanári segédlet nélkül, önállóan is feldolgozható, megérthető és elsajátítható.

Győr, 2006. május A szerző

2. Bevezetés

Ha egy feladat megoldására számítógépes programot készítünk, akkor általában az alábbi lépéseket, tevékenységeket kell elvégeznünk:

1. A feladat megfogalmazása, pontosítás, általánosítás.
2. Matematikai (vagy egyéb) modell kiválasztása, megadása (ha szükséges, ill. lehetséges).
3. Az adatszerkezet definiálása, az input-output specifikálása.
4. A megoldást megvalósító algoritmus megtervezése, elkészítése.
5. Programírás, kódolás (az adatszerkezet és az algoritmus alapján).
6. Tesztelés, hibakeresés.
7. Dokumentálás (felhasználóknak, fejlesztőknek).

Természetesen az adott munka, ill. feladat jellegéből adódóan bizonyos lépések el is maradhatnak (pl. 2., 7.), ill. javítás, módosítás esetén szükség lehet egy korábbi szintre való visszalépésre is.

Jegyzetünkben elsősorban a megoldások érdemi részét jelentő 3. és 4. lépésekre fókuszálunk az

Algoritmusok + Adatstruktúrák = Programok

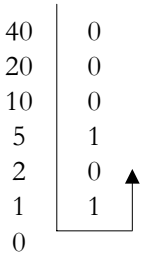
„képlet” alapján [Wir 82], de mint ahogy a puding próbája az evés, itt az 5., 6. lépések igazolják vissza megoldásunk helyességét, helytelenségét.

Az 1-5. megoldási lépéseket az alábbi feladat megoldásán keresztül szemléltetjük, de az alkalmazott adattípusokat, az adatszerkezeti táblázatot, az algoritmus megadási módszereket, azaz a megoldás eszközeit a későbbi fejezetekben részletesen tárgyaljuk.

Feladat: Alakítsunk át egy pozitív egész számot egy adott (2-16) számrendszerbe!

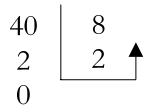
1. **Általánosítás**: A megoldást nem két konkrét adatra, hanem megadható, input adatokra készítjük el, így „tetszőlegesen” konvertálhatunk.
2. **Matematikai modell**: Ez maga az algoritmus, miszerint először a megadott számot, utána a keletkező hányadosokat mindaddig osszuk el az adott számrendszer alapszámával, amíg a hányados nulla nem lesz. Minden osztásnál jegyezzük fel az osztás maradékát. Ezekből, mint számjegyekből, a feljegyzésük fordított sorrendjében számot képezve, a megoldást kapjuk.

Pl: 40-et 2-s számrendszerbe



Eredmény: 101000

Pl: 40-et 16-s számrendszerbe



Eredmény: 28

3. **Adatszerkezet**: Tekintettel a hexadecimális számrendszer betűvel jelölt számjegyeire, ill. a nagy egész számok (4 bájt) esetén a kettes számrendszerbeli, esetlegesen 32 számjegyes eredményre, az eredmény típusa csak sztring lehet.

Funkció	Azonosító	Típus	Jelleg
Az átalakítandó szám	A	Egész	I
A számrendszer alapszáma	B	Egész	I
Az eredmény „szám”	ER	Sztring	M, O

Az ER változó munka jellegű is, mert értéke az algoritmus során alakul, változik, ahogyan a maradékokat egy sztringgé fűzzük.

4. **Algoritmus**: A lehetséges számjegyeket egy sztringkonstansban deklaráljuk:

Konstans
SZAMJEGYEK ”0123456789ABCDEF”

A megoldás pszeudokódja:

```
/* Számrendszer konverzió */
KONVERTAL(A,B)

ER ← ””
repeat
      ER ← SZAMJEGYEK[A MOD B+1]+ER
      A ← A DIV B
until A=0
return ER
```

5. **Programírás**: Az alábbiakban, az összehasonlítás végett három magasszintű és egy gépközeli nyelven is programmá írtuk a megoldásunkat.

C program

```
/* SZRKONV.C : Számrendszer konverzió */
#include <stdio.h>
#include <conio.h>
#define MaxHossz 32+1   /* +1: a végjelnek */
#define SzamJegyek "0123456789ABCDEF"

void Konvertal(int a, int b, char *er)
{ int i,j; char cs;
  /* Osztogatás */
  i=0;
  do {
    er[i++]=SzamJegyek[a%b];
    a=a/b;
  } while (a>0);
  /* Végjel */
  er[i]='\0';
  /* Megfordítás */
  for (j=0,i--; j<i; j++,i--) {
    cs=er[j]; er[j]=er[i]; er[i]=cs;
  }
}

void main()
{ int a,b;
  char c[MaxHossz];
  clrscr();
  printf("A konvertálandó pozitív egész szám:"); scanf("%d",&a);
  printf("A számrendszer alapszáma (2-16):"); scanf("%d",&b);
  Konvertal(a,b,c);
  printf("%s\n",c);
}
```

Megjegyzés

- A C nyelv nem elemi adattípusként kezeli a sztringeket, hanem adott végjelű (0-s kódú karakter) karakterláncok formájában, amelyeket egydimenziós karaktertömbök segítségével kezelhetünk. A fordított összefűzést megvalósítandó, az eredménysztring karaktereinek sorrendjét egyszerűen megfordítottuk.
- A tömböket a C nyelv 0-tól indexeli, így a *SzamJegyek* sztring egy adott számjegyének elérésére az osztás maradékát kell használnunk (nem eggyel nagyobbat).

Pascal program

```
{ Számrendszer konverzió }
program SZRKONV;
uses crt;
const SzamJegyek:string='0123456789ABCDEF';

function Konvertal(a,b:integer):string;
var er:string;
begin
  er:='';
  repeat
    er:=SzamJegyek[a mod b+1]+er;
    a:=a div b;
  until a=0;
  Konvertal:=er;
end;

var a,b:integer;
begin
  clrscr;
  write('A konvertálandó pozitív egész szám:'); readln(a);
  write('A számrendszer alapszáma (2-16):'); readln(b);
  writeln(Konvertal(a,b));
end.
```

Basic program

```
' Számrendszer konverzió
Option Explicit
Const SzamJegyek = "0123456789ABCDEF"

Function Konvertal(a As Integer, b As Integer) As String
  Dim er As String
  er = ""
  Do
    er = Mid(SzamJegyek, a Mod b + 1, 1) + er
    a = a / b
  Loop Until a = 0
  Konvertal = er
End Function

Sub Hivo()
  Dim a As Integer
  Dim b As Integer
  a = InputBox("A konvertálandó pozitív egész szám:")
  b = InputBox("A számrendszer alapszáma (2-16):")
  MsgBox (Konvertal(a, b))
End Sub
```

Megjegyzés: Ebben a Visual Basic megoldásban a *Hivo* szubrutin végzi el a C és Pascal megoldások főprogramjának szerepét, azaz az adatbekérést, a konvertálást elvégző *Konvertal* szubrutin meghívását, majd a kapott eredmény kiírását.

Assembly program

```
; Számrendszer konverzió
       .MODEL  SMALL
       .STACK  100h
       .DATA
SzJ    DB     "0123456789ABCDEF"
Er     DB     32 dup (?)
       .CODE
       mov    ax,@DATA            ;DS-be az adatszegmens
       mov    ds,ax               ;szegmenscímét
       mov    ax,40               ;A konvertálandó szám
       mov    bx,2                ;A cél szr. alapszáma
       mov    di,Offset Er        ;Az eredményterület
                                  ;offset címe
       call   Konv                ;CX-ben az Er sztring hossza

       mov    ah,2                ;Kiírás
       mov    si,0
Kiir:  mov    dl,Er[si]
       int    21h
       inc    si
       loop   Kiir

       mov    ah,4ch              ;Befejezés
       int    21h

; Input:      AX = a konvertálandó pozitív egész szám
;             BX = a cél számrendszer alapszáma
;             DI = az eredményterület offset címe
; Output:     DS:DI = az eredménysztring kezdőcíme
;             CX    = az eredménysztring hossza
; Elromló regiszterek: AX,DX,DI,SI

Konv   PROC
       mov    cx,0
Oszt:  mov    dx,0                ;Az előző maradék törlése
       div    bx                  ;Osztás
       inc    cx
       push   dx                  ;Maradékot a verembe
       cmp    ax,0
       jne    Oszt
       mov    dx,cx               ;CX megjegyzése
Cikl:  pop    si                  ;Az eredmény előállítása
       mov    al,SzJ[si]          ;AL-be a megfelelő karaktert
```

```
        mov     [di],al             ;A célterületre írjuk
        inc     di
        loop    Cikl
        mov     cx,dx               ;CX beállítása
        ret                         ;Visszatérés a hívóhoz
Konv    ENDP
        END
```

Megjegyzés: Ezen Turbo Assembly megoldás főprogramja fix adatokkal hívja meg a megoldást elvégző *Konv* szubrutint, így elkerültük az adatbekérés programozását.

3. Egyszerű adattípusok

Minden programfejlesztő rendszer meghatározza, definiálja az adatok azon körét, amelyet kezelni tud. Azt, hogy milyen fajta adatokat használhatunk, ezekkel milyen műveleteket végezhetünk, ezek hogyan tárolódnak, az *adattípusok* definiálják. Attól függően, hogy az adattípus egy, ill. több adat egyidejű használatát engedi meg, megkülönböztetünk *egyszerű* és *összetett* adattípusokat. Ebben a fejezetben az egyszerű adattípusokról lesz szó, amelyekből később, összetett adattípusokat „építünk" (lásd 9.).

3.1. Egész típus

Egész számok használatát megengedő adattípus.
Konstans: pl. −326
Műveletek:

- Előjel (+, −)
- Multiplikatív: Szorzás (*), Egész osztás hányadosa (/, DIV), Egész osztás maradéka (%, MOD)
- Additív: Összeadás (+), Kivonás (−)
- Hasonlítások: Egyenlő (=), Nem egyenlő (<>), Kisebb (<), Nagyobb (>), Kisebb vagy egyenlő (<=), Nagyobb vagy egyenlő (>=)

A hasonlítások eredményeként logikai, a többi művelet eredményeként egész értéket kapunk.
Pl. 5/3 → 1, 5 DIV 3 → 1, 5 MOD 3=5 % 3 → igaz

3.2. Karakter típus

Karakterek használatát megengedő adattípus.
Konstans: pl. 'A', #27
Műveletek:

- Összefűzés (+)
- Hasonlítások: mint az egészeknél, csak itt nem a számok értéke, hanem a karakterek ASCII kódja szerint történik a hasonlítás.

A hasonlítások eredményeként logikai értéket, az összefűzés eredményeként egy két karakterből álló sztringet (lásd 9.2.) kapunk.
Pl. 'A'<'B' → igaz, 'T'+'ó' → "Tó"

Megjegyzés
- Szimpla aposztrófot használunk, míg a sztringeknél (lásd 9.2.) duplát.
- Egy karakterkonstanst a karakter ASCII kódjának segítségével is megadhatunk (gondoljunk csak a nem begépelhető karakterekre), ekkor a karakter kódja elé a # karaktert tesszük (pl. #65 a 65-s kódú 'A' karaktert, a #27 az Esc billentyűhöz tartozó karaktert definiálja).

3.3. Logikai típus

Konstans: igaz, hamis
Műveletek:

- Tagadás (NOT), És (AND), Vagy (OR)
- Hasonlítások: mint az egészeknél.

Hogy valamennyi hasonlítás értelmezett legyen, a két logikai érték között is definiálunk sorrendiséget, a hamis megelőzi az igaz értéket.
A hasonlítások és a műveletek eredményeként logikai értéket kapunk.
Pl. hamis < igaz → igaz

A	B	NOT A	A AND B	A OR B
igaz	igaz	hamis	igaz	igaz
igaz	hamis	hamis	hamis	igaz
hamis	igaz	igaz	hamis	igaz
hamis	hamis	igaz	hamis	hamis

3.1. táblázat. A logikai műveletek.

3.4. Valós típus

Valós számok használatát megengedő adattípus.
Konstans: pl. 3.14
Műveletek: mint az egészeknél, csak itt egyetlen osztás megengedett (/), a többi (DIV, MOD, %) nem értelmezett.
A hasonlítások eredményeként logikai, a többi művelet eredményeként valós értéket kapunk.

Megjegyzés: A programfejlesztő rendszerek általában többféle egész, ill. valós típus használatát engedik meg, ezekkel külön nem foglalkozunk, az algoritmus programmá írásakor a megfelelő, az adatok pontos tárolásához már elegendő típust célszerű választani. Felesleges például 4 bájtos egész típust használni akkor, amikor az egész adataink elférnek 1 bájton is. Az egyszerűség kedvéért példaprogramjainkban egész típusként általában az *int,* ill. *integer,* valós típusként a *float,* ill. *real* típusokat használjuk.

4. Adatok tárolása

Ahhoz, hogy adatainkat a számítógép kezelni tudja, tárolnia kell. A tárolás miként je, konkrét megvalósítása egyrészt az adatok típusától, másrészt az alkalmazott fejlesztőkörnyezettől és az operációs rendszertől is függ, ezért ezt nem részletezzük (lásd [Nyé 03]). Az adatok memóriában történő egyszerű tárolásáról ebben a fejezetben, a dinamikus tárkezelésről a 13., míg a külső adathordozón, fájlokban történő adattárolásról a 14. fejezetben lesz szó.

4.1. Változó

Változón olyan azonosítóval ellátott memóriaterületet értünk, ahol a változó típusának megfelelő adatot tárolhatunk.

A változó értékén az éppen benne tárolt adatot értjük. Ez az érték az algoritmus, ill. a program végrehajtása során megváltozhat – innen ered az elnevezése –, ilyenkor a változóba kerülő új adat „felülírja" a régit.

4.2. Kifejezés

Kifejezésen olyan számítási műveletsort értünk, amellyel megmondjuk, hogy milyen adatokkal, milyen műveleteket, milyen sorrendben kívánunk elvégezni. A kifejezés kiértékelése, kiszámítása után egy új érték − a kifejezés értéke − keletkezik.
A kifejezésben szerepelhetnek:

- Konstansok, változók, függvényhívások
- Műveletek
- Zárójelek (csak kerek zárójelek használhatók)

A kifejezésben szereplő adatokat (konstansok, változók) operandusoknak, a műveleteket operátoroknak is szokták nevezni.
Pl. (–B+SQRT(B*B–4*A*C))/(2*A)
A 4 és a 2 konstansok, az A, B, C változók, az SQRT pedig a négyzetgyökvonást elvégző függvény neve.
A műveletek erősorrendje (prioritása) csökkenő erősorrendben:

- Egyoperandusú: Előjel (+,−), Tagadás (NOT)
- Multiplikatív (*, /, DIV, MOD, %, AND)
- Additív (+,−, OR)
- Hasonlítások (=, <>, <, >, <=, >=, IN)

Megjegyzés
- A hasonlításokat relációs műveleteknek is nevezik.
- A sztring- és halmazműveleteket (pl. a tartalmazás IN műveletét) a megfelelő fejezetekben tárgyaljuk.

A kifejezések kiértékelésének szabályai:

1. A zárójelbe tett kifejezések és függvényhívások operandus szintre emelkednek.
2. A magasabb prioritású műveletek végrehajtása megelőzi az alacsonyabb prioritású műveletek végrehajtását.
3. Az azonos prioritású műveleteknél a balról-jobbra szabály érvényes, miszerint a végrehajtás balról jobbra haladva történik.
4. A logikai kifejezések kiértékelése befejeződik, amint az eredmény értéke már nem változhat.
5. A numerikus műveleteknél az eredmény egész, ha az operandusok egészek és valós, ha valamelyik operandus valós.

A szabályok alapján egyértelműen meghatározható a kifejezésben szereplő műveletek végrehajtási sorrendje, így a kifejezés értéke mindig egyértelműen kiszámítható.

4.3. Függvények

Munkánk során, az algoritmusok készítése közben feltesszük, hogy bizonyos számolásokat, átalakításokat elvégezhetünk, azaz léteznek az alábbi tevékenységeket elvégző függvények.

4.3.1. Matematikai függvények

ABS(X)	X abszolút értéke.
EXP(X)	Az exponenciális függvény (e^x) értéke az X helyen.
LOG(X)	A természetes alapú logaritmus függvény értéke az X helyen.
SIN(X)	X szinusza (X radiánban adott).
COS(X)	X koszinusza (X radiánban adott).
SQR(X)	X négyzete.
SQRT(X)	X négyzetgyöke.
RANDOM(X)	Egy véletlen egész szám 0-tól X−1-ig (X egész érték).

4.3.2. Konverziós függvények

ASC(X)	Az X karakter ASCII kódja (pl. ASC('A') → 65).
CHR(X)	Az X ASCII kódú karakter (pl. CHR(65) → 'A').

Megjegyzés

- A C nyelvben nincs szükség ezekre a konverziókra, mert egy karakter és kódjának használata ekvivalens.
- A sztringkezelő függvényeket valamint a dinamikus tárkezeléshez, fájlkezeléshez kapcsolódó függvényeket a megfelelő fejezetekben tárgyaljuk.

4.4. Az értékadó utasítás

Egy vagy több azonos típusú változónak értéket adhatunk, bennük adatot tárolhatunk.
Jelölés:

változó ← kifejezés

vagy

változó ← … ← változó ← kifejezés

Végrehajtás:

- Kiértékelés: a kifejezés értékének kiszámítása.
- Tárolás: ha az érték tárolható, akkor tárolódik a változó(k)ban, különben hiba lép fel.

Egy érték akkor tárolható egy adott változóban, ha

- a típusuk megegyezik,
- valós típusú változóhoz egész értéket rendelünk,
- egész típusú változóhoz valós értéket rendelünk,
- sztring típusú változóhoz karaktert rendelünk.

Az esetleges túlcsordulástól (amikor egy érték a nagysága miatt nem tárolható), valamint a típuseltérések esetén szükséges (a tárolási mód különbözőségéből fakadó) konverzióktól eltekintünk, feltesszük, hogy a konverziók automatikusan végrehajtódnak. Valós értékek egész változókhoz rendelését megengedjük, ekkor a szám egész része konvertálódik, a törtrész figyelmen kívül marad.

Megjegyzés: A C nyelvben megengedett egészek és karakterek közötti automatikus oda-vissza konverziót nem engedjük meg, míg ott két karakter összege a kódok összegét adja, addig a mi jelölésmódunk szerint egy, kétkarakteres sztringet.

Pl. I←I+1 /* I értékéhez hozzáad 1-et és az eredményt I-be teszi. */
I←J←K←0 /* Az I, J, K változókba nullát tesz. */

Az első példa értékadása normális esetben megnöveli eggyel az I változó értékét, szélsőséges esetben (ha pl. I-ben, a típusának megfelelő legnagyobb szám van éppen) azonban nem. A számítógép véges memóriájában ugyanis nem tárolhatunk végtelen nagy számokat, a számítógép számábrázolása csak bizonyos korlátok között (értelmezési tartomány, pontosság) engedi meg a számok használatát.

4.5. Beolvasó utasítás

Egy vagy több változónak értéket adhat a felhasználó a standard input eszköz, a billentyűzet segítségével.

Jelölés:

Be: változólista

A változólista változók vesszővel elválasztott sorozata.

Pl. Be: A, B, C

Végrehajtás: a felhasználó által megadott adatok tárolódnak a felsorolt változókban.

Megjegyzés

- Adatok bekérésekor általában valamilyen információt (pl. tájékoztató üzenetet) kell adni arról, hogy milyen adatot várunk. Az egyszerűség kedvéért ettől eltekintünk, a megoldások programmá írásakor viszont célszerű ezeket megtenni.
- Feltesszük, hogy a felhasználó jó adatot ad meg, az adatok ellenőrzésével (ha csak a feladat jellege nem kifejezetten ilyen, lásd 11.4.) nem foglalkozunk.
- Csak az egész, karakter, valós és sztring típusú adatok beolvasása megengedett. Nem kérhetünk be tehát logikai értékeket, halmazokat, tömböket, rekordokat. Az összetett adatok (lásd 9.) bekérése elemenként, ill. mezőnként történhet, ha azok bekérése megengedett.

4.6. Kiíró utasítás

Egy vagy több kifejezés értéke kiíratható a standard output eszközre, a képernyőre.
Jelölés:

Ki: kifejezéslista

A kifejezéslista kifejezések vesszővel elválasztott sorozata.
Pl. Ki: ”A kör sugara:”, R, ”területe:”, R*R*3.14
Végrehajtás: a kifejezések értéke kiértékelődik, majd sorban kiíródik a képernyőre.

A példában szereplő első és harmadik kifejezés értékei maguk a (sztring) konstansok, a másodiké az R változó tartalma, a negyediké pedig, az R aktuális értékéből kiszámolt érték lesz, ezek íródnak ki.

Megjegyzés

- Az egyszerűség kedvéért a kiírás esetleges pozícionálásával, tagolással, soremeléssel itt nem foglalkozunk, ezeket a megoldások programmá írásakor kell megtennünk (a feladathoz illeszkedően, vagy ha ez nem kötött, akkor tetszés szerint).
- Csak az egész, karakter, valós és sztring típusú adatok kiírása megengedett. Nem írhatunk ki tehát logikai értékeket, halmazokat, tömböket, rekordokat. Az összetett adatok (lásd 9.) kiírása elemenként, ill. mezőnként történhet, ha azok kiírása megengedett.

Az értékadó, beolvasó és kiíró utasításokat *alaptevékenységeknek* nevezzük. Azt, hogy egy feladat megoldásához szükséges alaptevékenységeket milyen sorrendben kell végrehajtani, a *vezérlőszerkezetekkel* szabályozhatjuk, amelyeket a későbbiekben (lásd 7.) ismertetünk.

5. Adatszerkezeti táblázat

Egy feladat megoldásához szükséges adatokat és a tárolásukra használt változókat egy olyan táblázatban adjuk meg, amelyben egy-egy adatot az őt kezelő változó tulajdonságainak megadásával jellemzünk. Négy tulajdonságot tartunk fontosnak:

- **Funkció**: ez a tulajdonság mutatja meg, hogy a változóba milyen adat kerül, mire használjuk a változót.
- **Azonosító**: a változó azonosítója (neve), ezzel hivatkozunk az adatra.
- **Típus**: a változó, és ezen keresztül az adat típusa, tárolási módja. Itt adjuk meg az összetett változót jellemző információkat is (pl. tömbdimenziók).
- **Jelleg**: a változóban kiindulási (input), végeredmény (output), ill. részeredmény (munka) adatot tárolunk-e.

A változók azonosítóinak (csakúgy, mint a feladat megoldásához deklarált konstansok, típusok, szubrutinok azonosítóinak) egyedieknek kell lenniük. Célszerű rövid, de kifejező neveket választani. A megoldások programmá írását megkönnyítendő, az azonosítókban csak az angol betűket, a számjegyeket és az aláhúzás karaktert használjuk. Az olvashatóság és a könnyebb áttekinthetőség érdekében ezeket nagybetűvel írjuk, csakúgy, mint a műveletek és függvények neveit.

A munka jellegű változókban részeredményeket, ill. az algoritmus megadásához, vezérléséhez szükséges adatokat tároljuk.

Egy változó, a benne tárolt adattól függően többféle jelleggel is rendelkezhet. A jellegeket a kezdőbetűikkel rövidítjük.

Pl: Egy rendező algoritmushoz az alábbi adatszerkezeti táblázat definiálható:

Funkció	Azonosító	Típus	Jelleg
A rendezendő elemek	A	Egydimenziós, tetszőleges elemtípusú tömb	I, M, O
A rendezendő elemek száma	N	Egész	I
Két elem cseréjéhez	CS	Az A tömb elemeivel megegyező típusú	M
Segédváltozók	I, J, K	Egész	M

A tetszőleges elemtípusú jelen esetben azt jelenti, hogy az algoritmus működik az összes olyan adatsorra, amelynek elemeire értelmezettek a hasonlítás műveletek. A konkrét elemtípust a rendezendő adatsor elemeinek típusa határozza meg.

Az A tömbben kapjuk (input) a rendezendő elemeket, az algoritmus során itt rendezzük őket (munka), és a legvégén itt lesznek sorba rendezve is (output). A tömbök méretét (méreteit) is itt tüntetjük fel, ha ez a feladat szempontjából lényeges (lásd 9.1.).

Ha egy feladat megoldása során egy adott *konstans értéket* vagy egy új, általunk definiált *típust* több helyen is használni szeretnénk, akkor célszerű őket egyszer deklarálni, és utána csak az azonosítójukat használni. Az alábbi egyszerű formalizmust fogjuk használni:

Konstans
    Azonosító Adat
Típus
    Azonosító Típusleírás

Pl.

Konstans
    SORMAX 10
    OSZLMAX 20
Típus
    ELEM Egész
    MATRIX Kétdimenziós ELEM tömb[SORMAX, OSZLMAX]

A fenti példában két konstanst és két típust deklarálunk úgy, hogy a MATRIX tömbtípus deklarálásában már használjuk is az előtte deklarált ELEM típust és méretkonstansokat.

Megjegyzés: A feladatok kitűzésekor gyakran használunk szimbólumokat (azonosítókat) az adatok jelölésére (pl. *N*), amelyek tárolására felhasznált változóknak célszerűen, de nem szükségszerűen, ugyanazokat az azonosítókat, neveket adjuk (pl. N). Azért, hogy ez ne okozzon félreértést (azaz ne keverjük az adatokat az őket tároló változókkal), az adatok azonosítóját dőlt kiemeléssel írjuk, a változók azonosítóit (csakúgy, mint a feladat megoldásához deklarált konstansok, típusok, szubrutinok azonosítóit) nem.

6. Algoritmusok megadása

Az algoritmusok megadására, leírására többféle eszköz áll rendelkezésünkre.

Szöveges leírás
Az algoritmust szavakkal, mondatokkal írjuk le. Szükség esetén az egyes lépéseket sorszámokkal látjuk el, amelyek a megoldás menetét, a lépések végrehajtási sorrendjét fejezik ki. Minden megoldásnál használni fogjuk.

Folyamatábra
Az algoritmus folyamatát kifejező jelölésmód, amelyen jól követhető az algoritmus lépéseinek sorrendje, a megoldás menete. Csak ebben a fejezetben használjuk.

D-diagram
Olyan strukturált folyamatábra, amely csak a *szekvencia*, *szelekció*, *iteráció* vezérlőszerkezeteket engedi meg, csak ezekből építkezik. Csak ebben a fejezetben használjuk, ahol a mintafeladat megoldásában szereplő folyamatábra egyben D-diagram is.

Megjegyzés: A névben szereplő D betű a D-diagramot definiáló E. W. Dijkstra nevéből ered, aki így próbált „rendet tenni” a folyamatábra feltételes és feltétel nélküli vezérlésátadásaival okozható „kuszaságban”.

Struktúradiagram
Az algoritmusok felülről-lefelé történő tervezését támogató eszköz. Először csak a főbb lépéseket definiáljuk, majd azokat hasonló módon kifejtve, fokozatosan jutunk el a teljes megoldás leírásáig. Az algoritmusok tervezésénél ezt fogjuk használni.

Megjegyzés: Szerkezeti ábrának is nevezik [Mar 93].

Struktogram
Az algoritmusok vezérlőszerkezeteit „dobozokkal” kifejező módszer. Csak ebben a fejezetben használjuk.

Pszeudokód
Olyan „utasításszintű” leíró eszköz, amelynek vezérlőszerkezeteit a magasszintű programozási nyelvekben „megszokott” módon, vagy ahhoz

nagyon hasonlóan írjuk le. Mivel az alapul vett három programozási nyelv (C, Pascal, Basic) kicsit eltérő szintaktikával használja ezeket a vezérlőszerkezeteket, ezért egy általános, egyszerűsített, a közös lényeget kifejező jelölésmódot definiálunk. Az előbb felsorolt nyelvek ismeretében talán ez a leírás olvasható, ill. programozható a legkönnyebben, ezért valamennyi megoldásnál használni fogjuk.

Az egyszerűség és rövidség érdekében bizonyos feladatok megoldásánál (mint az alábbi példában is) feltesszük, hogy az input adatok már rendelkezésre állnak, ezeket megkapjuk, míg az output eredményeket visszaadjuk. Ilyenkor a megoldást egy *szubrutin* (lásd 10.) végzi, amelynek neve mögött zárójelben felsoroljuk a kommunikációt végző, input ill. output adatokat tartalmazó változók azonosítóit (mint a szubrutin paramétereit), jelezve ezzel azt, hogy a megoldás csak az érdemi részt tartalmazza, az esetleges adatbekérés és eredménykiírás, a szubrutint hívó programrész (pl. a főprogram) feladata.

Az algoritmus megadási módszerek hasonlóságát, különbözőségét az alábbi feladat megoldásán szeretnénk szemléltetni.

Feladat: Határozzuk meg két egész szám legnagyobb közös osztóját!

Megoldás: A feladatra Euklidesz (kb. 2300 évvel ezelőtt) egy, a maradékos osztáson alapuló megoldást adott, amelynek lényege a következő:

Az a és $b \neq 0$ számokra végezzük el a maradékos osztást, azaz

$$a = bq_1 + r_1 \qquad (0 \leq |r_1| < |b|).$$

Ezután b és r_1 között is,

$$b = r_1 q_2 + r_2 \qquad (0 \leq |r_2| < |r_1|)$$

és így tovább. Mivel a maradékok (r_1, r_2, …) abszolút értékben monoton csökkennek, így véges számú lépésben bekövetkezik az

$$r_{n-1} = r_n q_{n+1} + r_{n+1} \qquad (0 < |r_{n+1}| < |r_n|)$$
$$r_n = r_{n+1} q_{n+2}$$

állapot, azaz a maradék (r_{n+2}) nulla lesz. Az utolsó nem nulla maradék (r_{n+1}) az a és b számok abszolút értékben legnagyobb közös osztóját adja.

Az algoritmus lényege abban áll, hogy abszolút értékben egyre kisebb számokra (a és b helyett b és $a \bmod b$ értékekre) végzi el sorban ugyanazt (egy maradékos osztást), egészen addig, amíg az osztás maradéka nulla nem lesz.

Szöveges leírás

Hogy a nullával való osztást elkerüljük, a *b=0* feltételt vizsgáljuk meg először.

1. Ha *b* értéke 0, akkor *a* értéke a megoldás és készen vagyunk.
2. Ha *b* értéke nem 0, akkor osszuk el *a* értékét *b* értékével maradékosan és jelölje az osztás maradékát *r*.
3. Legyen *a* új értéke *b*, *b* új értéke *r* és folytassuk a végrehajtást az 1. lépéssel.

A végrehajtási lépéseket egyszerűen megsorszámoztuk.

Az algoritmusok „kézzel", papíron történő „lejátszása" nemcsak az algoritmus elkészítését, de a már kész algoritmusok megértését is segítheti.

Az alábbi táblázat az algoritmus egyes lépéseihez tartozó *a*, *b* és *r* értékeket tartalmazza, ami jól tükrözi az algoritmus „működését".

	a b r	a b r	a b r	a b r	a b r	a b r	a b r
1.	40 18 4	6 9 6	-5 3 -2	2 -5 2	-10 -8 -2	0 21 0	0 0
2.	18 4 2	9 6 3	3 -2 1	-5 2 -1	-8 -2 0	21 0	
3.	4 2 0	6 3 0	-2 1 0	2 -1 0	-2 0		
4.	2 0	3 0	1 0	-1 0			

6.1. táblázat. Értékek alakulása osztással.

Megjegyzés

- Az algoritmusnak csak abban az esetben van értelme, ha a *b* szám nem nulla (ekkor *a* akár 0 is lehet).
- Az algoritmus az *a=0* és *b=0* extrém esetben a 0 eredményt adja.

Ha a számok csak természetes számok lehetnek (1, 2,…), akkor a maradékos osztás helyett kivonáson alapuló algoritmus is adható, amely lényegében ugyanazt végzi, csak a maradékos osztás ismételt kivonásokkal „helyettesítődik".

1. Ha *a* értéke megegyezik *b* értékével, akkor ez az érték a megoldás és készen vagyunk.
2. Ha *a* értéke nem egyezik meg *b* értékével, akkor a nagyobb szám értékét csökkentsük a kisebb szám értékével és folytassuk a végrehajtást az 1. lépéssel.

	a b	*a b*
1.	40 18	6 9
2.	22 18	6 3
3.	4 18	3 3
4.	4 14	
5.	4 10	
6.	4 6	
7.	4 2	
8.	2 2	

6.2. táblázat. Értékek alakulása kivonással.

A további megoldások egzaktabb jelölésmódot használnak, ezért előtte megadjuk a feladathoz tartozó adatszerkezeti táblázatot is.

Adatszerkezeti táblázat

Funkció	Azonosító	Típus	Jelleg
Egyik szám	A	Egész	I, M, O
Másik szám	B	Egész	I, M
Az osztásos algoritmusban az osztás maradéka	R	Egész	M
A rekurzív függvények eredménye	ER	Egész	O

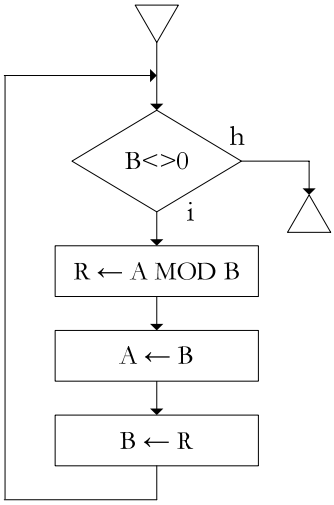
Az A és B változók munka jellegét az indokolja, hogy értékük megváltozik az algoritmus során, A output jellegét pedig az, hogy itt „keletkezik” az eredmény.

Mind a maradékos osztáson, mind a kivonáson alapuló algoritmust bemutatjuk az alábbiakban, de az R változó csak az osztáson alapuló algoritmushoz, míg az ER változó csak a rekurzív megoldásokhoz szükséges.

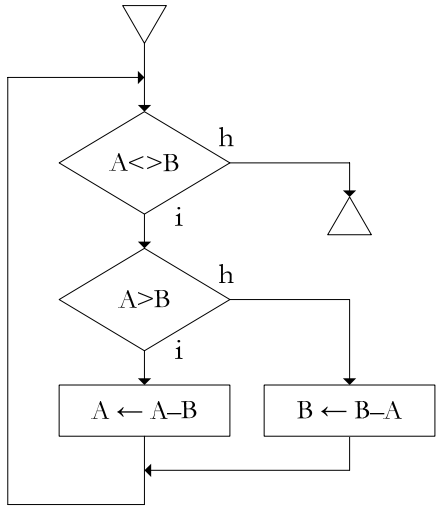
A rekurzív függvények ER változóját a megoldás strukturáltságának megőrzése érdekében használtuk (lásd 10.).

Folyamatábra

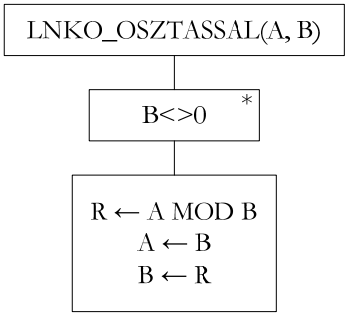
LNKO_OSZTASSAL(A, B)

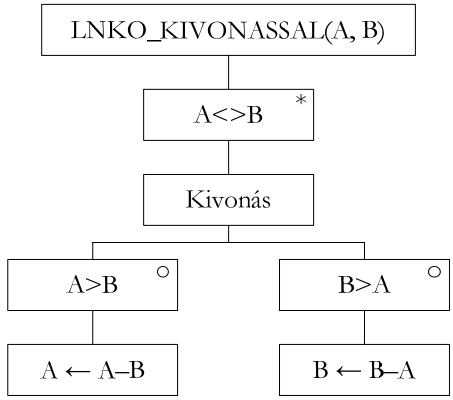


LNKO_KIVONASSAL(A, B)



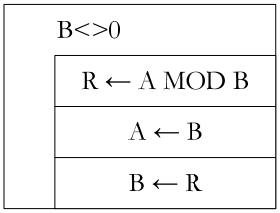
Struktúradiagram



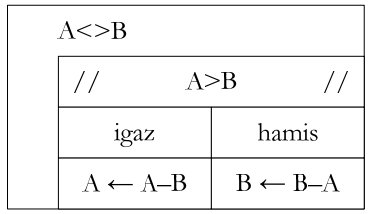


Struktogram

LNKO_OSZTASSAL(A, B)



LNKO_KIVONASSAL(A, B)



Pszeudokód

```
LNKO_OSZTASSAL(A,B)
while B<>0
        R ← A MOD B
        A ← B
        B ← R
```

```
LNKO_KIVONASSAL(A,B)
while A<>B
        if A>B
                A ← A-B
        else
                B ← B-A
```

A feladatra rekurzív megoldások is adhatók, ezek pszeudokódja a következő:

```
LNKO_OSZTASSAL(A,B)
if B=0
        ER ← A
else
        ER ← LNKO_OSZTASSAL(B,A MOD B)
return ER
```

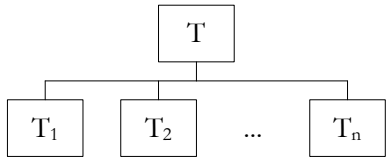
```
LNKO_KIVONASSAL(A,B)
if A=B
        ER ← A
else
        if A>B
                ER ← LNKO_KIVONASSAL(A-B,B)
        else
                ER ← LNKO_KIVONASSAL(A,B-A)
return ER
```

7. Strukturált algoritmusok tervezése

Egy feladat megoldása közben, az egyes tevékenységek végrehajtási sorrendjét vezérlőszerkezetekkel definiáljuk. Az egyes vezérlőszerkezeteket (szekvencia, szelekció, iteráció), ezek jelölését külön fejezetekben, különböző példákon szemléltetve tárgyaljuk.

7.1. Szekvencia

Olyan vezérlőszerkezet, amelynek során az egyes résztevékenységeket sorban, egymás után hajtjuk végre.
Struktúradiagram-jelölés



Jelentés: A T tevékenység végrehajtása a T_1, T_2, … , T_n résztevékenységek egymás utáni végrehajtásából áll.

Megjegyzés: Ha több, egymást követő résztevékenység alaptevékenység, akkor azok egyetlen téglalapba összevonva is megadhatók, elkerülve ezzel az ábra túl gyors szélesedését.

Pszeudokód-jelölés

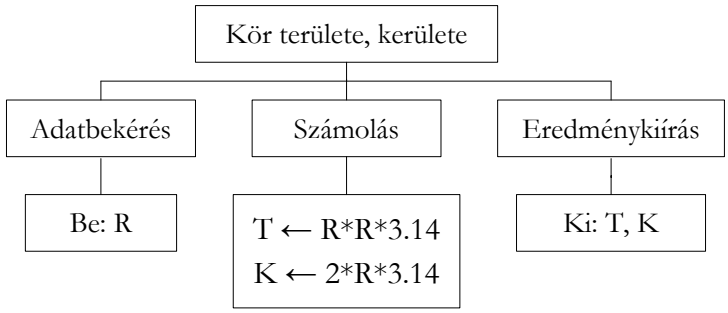
T_1
T_2
…
T_n

A T_1, T_2, … , T_n résztevékenységeket egyszerűen a T tevékenység helyére írjuk, egymás alá, egyvonalba.

Feladat: Határozzuk meg egy adott sugarú kör területét és kerületét!

Megoldás: Egy kör területét az $r^2\pi$, kerületét a $2r\pi$ matematikai képletek definiálják, ezeket kell kiszámolnunk.

Funkció	Azonosító	Típus	Jelleg
A kör sugara	R	Valós	I
A kör területe	T	Valós	O
A kör kerülete	K	Valós	O




```
/* Kör területe, kerülete */
Be: R
T ← R*R*3.14
K ← 2*R*3.14
Ki: T,K
```

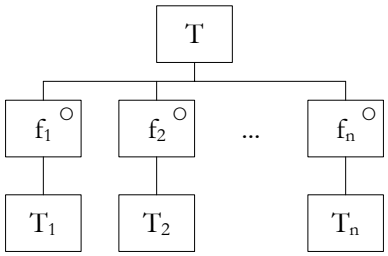
Megjegyzés

- A π értékére a 3.14 közelítő értéket használtuk.
- A pszeudokódba megjegyzéseket is tehetünk, ezeket új sorokba vagy a sorok végére tesszük, a /* és */ karakterek közé.

7.2. Szelekció

Olyan vezérlőszerkezet, amellyel kiválasztható a végrehajtandó résztevékenység.

Struktúradiagram-jelölés



f_i (i=1, 2, …, n): feltételek (logikai kifejezések), amelyek egymást páronként kizárják (azaz közülük legfeljebb egy lehet igaz).

T_i (i=1, 2, …, n): résztevékenységek.

Jelentés: A T tevékenység végrehajtása annak a T_i résztevékenységnek a végrehajtását jelenti, amelyhez tartozó f_i feltétel igaz. Ha egyik feltétel sem teljesül, akkor a T tevékenység üres tevékenység.

Pszeudokód-jelölés

```
if f1
        T1
else if f2
        T2
...
else if fn
        Tn
```

A fentieket a T tevékenység helyére írjuk úgy, hogy a T_1, T_2, … , T_n részfeladatok beljebb, egymás alá, egyvonalba kerüljenek.

Megjegyzés: Ha a szelekcióban csak egy feltétel szerepel (n=1), akkor nincs *else* rész.
Pl.

```
if A>0
        DB ← DB+1
```

Ha a szelekcióban lévő feltételek közül az egyik biztosan teljesül, akkor az utolsó feltétel (if f_n) elhagyható.
Pl.

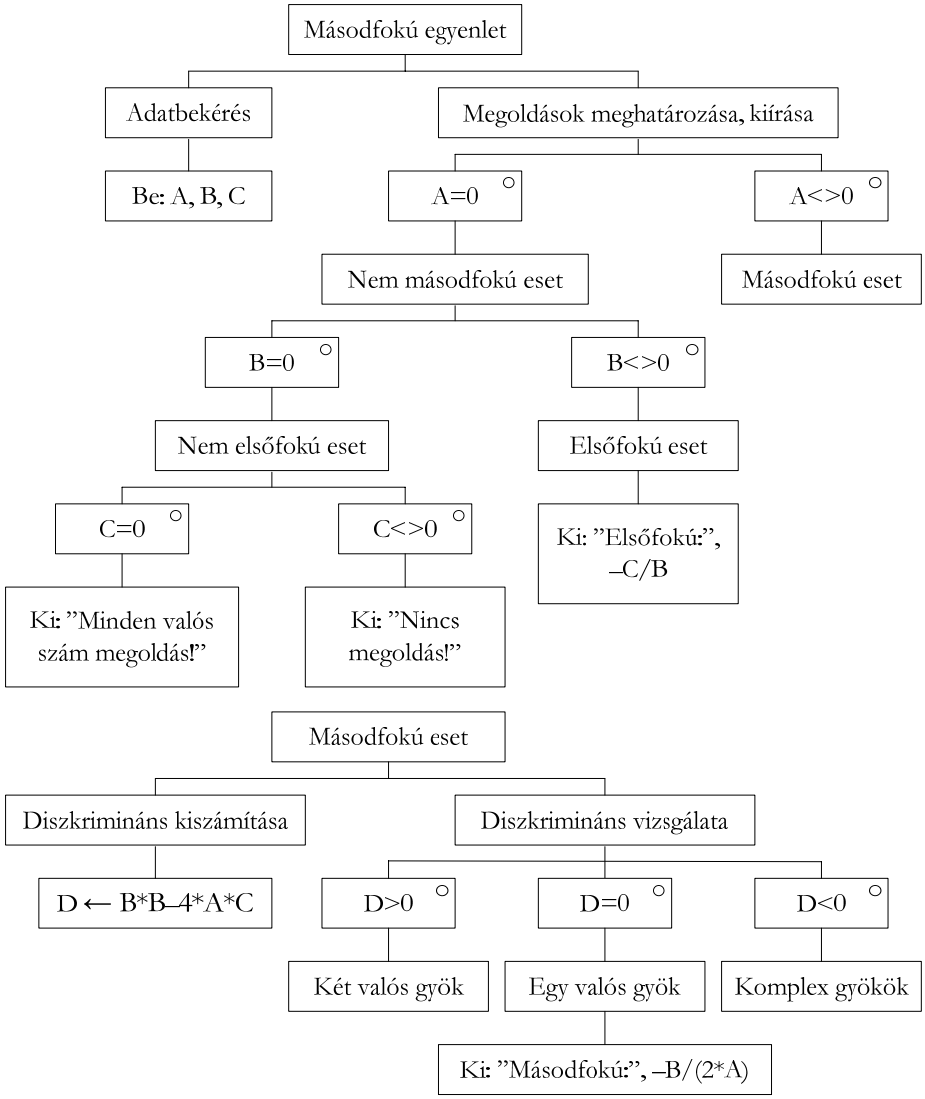
```
if A>B
        Ki: A
else
        Ki: B
```

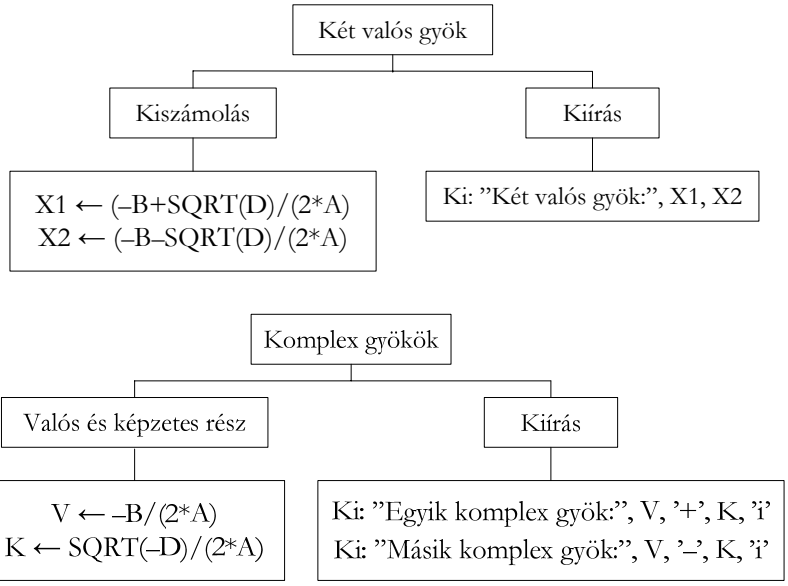
A szelekció ezen tagolása amellett, hogy az áttekinthetőséget növeli, elsősorban azért szükséges, mert így jelezzük az egyes ágakhoz tartozó résztevékenységek elejét és végét. Fokozottan ügyeljünk erre több, „egymásba ágyazott" szelekció esetén, mert csak a tagolás jelzi azt, hogy az egyes tevékenységek melyik feltételhez tartoznak.

Feladat: Oldjuk meg az $ax^2+bx+c=0$ másodfokú egyenletet, ahol az a, b, c együtthatók valós számok!

Megoldás: Minden esetet megvizsgálunk, a megoldásokat ezek alapján számoljuk és írjuk ki.

Funkció	Azonosító	Típus	Jelleg
Az együtthatók	A, B, C	Valós	I
Az egyik valós gyök	X1	Valós	O
A másik valós gyök	X2	Valós	O
A komplex gyökök valós része	V	Valós	O
A komplex gyökök képzetes része	K	Valós	O
A diszkrimináns értéke	D	Valós	M






```
/* Másodfokú egyenlet megoldása */
Be: A,B,C
if A=0
        /* Nem másodfokú eset */
        if B=0
                /*Nem elsőfokú eset */
                if C=0
                        Ki: ”Minden valós szám megoldás!”
                else
                        Ki: ”Nincs megoldás!”
        else
                Ki: ”Elsőfokú:”,-C/B
else
        /* Másodfokú eset */
        D ← B*B-4*A*C
        if D>0
                /* Két valós gyök */
                X1 ← (-B+SQRT(D))/(2*A)
                X2 ← (-B-SQRT(D))/(2*A)
                Ki: ”Két valós gyök:”,X1,X2
        else if D=0
                Ki: ”Másodfokú:”,-B/(2*A)
        else
                /* Komplex gyökök */
                V ← -B/(2*A)
                K ← ABS(SQRT(-D)/(2*A))
                Ki: ”Egyik komplex gyök:”,V,’+’,K,’i’
                Ki: ”Másik komplex gyök:”,V,’-’,K,’i’
```

Megjegyzés: A diszkrimináns kiszámításánál a négyzetre emelést egyszerű szorzással végeztük (az SQR függvény használata helyett).

7.3. Iteráció

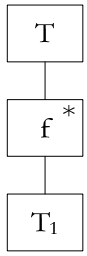
Olyan vezérlőszerkezet, amelyben egy tevékenység ismételten végrehajtható.

Azt a tevékenységet, amelyet az iteráció – vagy más néven ciklus – során ismételten végrehajtunk, ciklusmagnak nevezzük.

Három alapvető ciklusszervezés létezik, ezeket ismertetjük a továbbiakban.

7.3.1. Elöltesztelő iteráció

Struktúradiagram-jelölés



f : a ciklust vezérlő feltétel (logikai kifejezés)

T_1 : a ciklus magja (az ismételendő tevékenység)

Jelentés: A T tevékenység végrehajtása a T_1 tevékenység ismételt végrehajtását jelenti úgy, hogy amíg teljesül a ciklust vezérlő feltétel, addig végrehajtjuk a T_1 ciklusmagot. A feltétel értékét a ciklusmag végrehajtása előtt vizsgáljuk.

Pszeudokód-jelölés

```
while f
      T1
```

Az első sort a T helyére, míg a T_1 ciklusmag tevékenységeit beljebb írjuk.

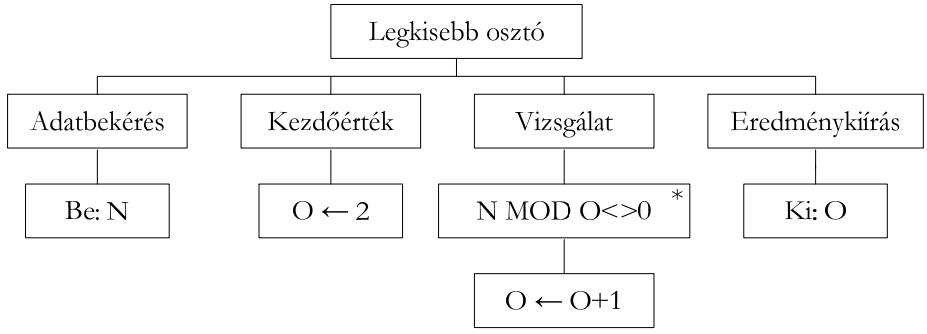
Megjegyzés

- A T_1 ciklusmagnak olyan tevékenységet is tartalmaznia kell, amely hatással van a ciklust vezérlő feltétel értékére (mert különben végtelen ismétlődés, végtelen ciklus keletkezhet – ha a feltétel értéke kezdetben igaz).
- Ha a feltétel értéke kezdetben hamis, akkor a ciklusmag egyszer sem kerül végrehajtásra, ekkor a T tevékenység üres tevékenység.

Feladat: Határozzuk meg egy 1-nél nagyobb egész szám 1-től különböző, legkisebb osztóját!

Megoldás: Mivel ilyen osztó biztosan létezik, ha más nem, a szám önmaga, ezért 2-től kezdve sorra vesszük az egész számokat és megnézzük, hogy osztja-e az adott számot vagy sem. Ha osztja, azaz megvan benne maradék nélkül, akkor ez a szám a megoldás, különben vesszük a következő számot.

Funkció	Azonosító	Típus	Jelleg
A vizsgált szám	N	Egész	I
Az aktuális osztó	O	Egész	M, O

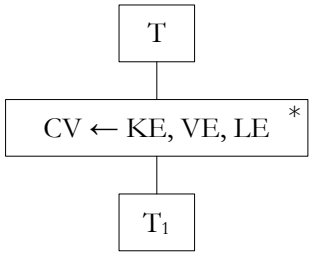



```
/* Legkisebb osztó */
Be: N
O ← 2
while N MOD O<>0
      O ← O+1
Ki: O
```

7.3.2. Növekményes iteráció

Gyakran a ciklusmagot adott számban, ill. adott értékeken végiglépdelve kell végrehajtani. Az ilyen feladatok megoldása egy speciális elöltesztelő ciklussal, a növekményes ciklussal történhet.

Struktúradiagram-jelölés



CV: ciklusváltozó

KE: kezdőérték

VE: végérték

LE: lépésköz

T_1 : a ciklus magja (az ismételendő tevékenység)

Jelentés: A T tevékenység végrehajtása a T_1 tevékenység ismételt végrehajtását jelenti úgy, hogy a ciklusváltozó a kezdőértéktől a végértékig lépdel, a megadott lépésközzel.

Pontosítva:

1. Kiszámolódnak a kezdőérték, végérték és lépésköz értékek, amelyek tetszőleges valós kifejezéssel megadhatók (de általában egész számok).
2. A ciklusváltozó felveszi a kezdőértéket.
3. A lépésköz előjelétől függően kiértékelődik a CV<=VE (pozitív lépésköz esetén), ill. a CV>=VE (negatív lépésköz esetén) feltétel.
4. Ha a feltétel igaz, akkor végrehajtódik a T_1 ciklusmag, és a ciklusváltozó értéke módosul a lépésköz értékével (CV ← CV+LE), majd a 3. ponttól folytatódik a végrehajtás.
5. Ha a feltétel hamis, kilépünk a ciklusból.

Pszeudokód-jelölés

```
for CV ← KE,VE,LE
        T1
```

Az első sort a T helyére, míg a T_1 ciklusmag tevékenységeit beljebb írjuk.

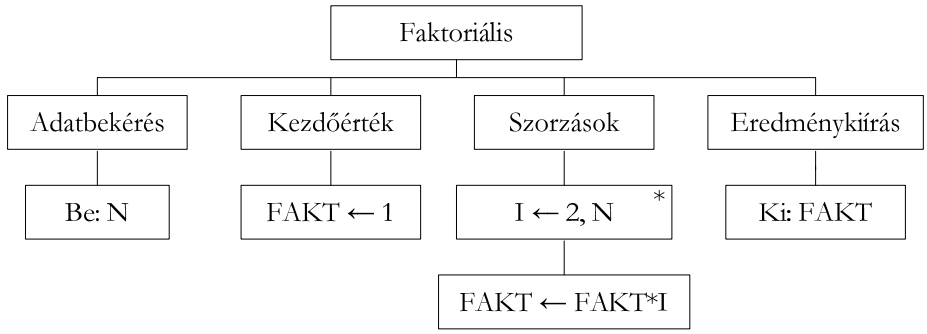
Megjegyzés

- Ha a lépésköz 1, akkor azt elhagyhatjuk, azaz ha nem írjuk ki, akkor 1 az alapértelmezés.
- A nulla lépésköz nem megengedett.
- Mivel a ciklus elöltesztelő, így lehet, hogy a ciklusmag egyszer sem kerül végrehajtásra (pl. pozitív lépésköz esetén, ha a kezdőérték nagyobb, mint a végérték). Ekkor a T tevékenység üres tevékenység.

Feladat: Határozzuk meg egy pozitív egész szám faktoriálisát!

Megoldás: Az *n!=1*2*3…*(n–1)*n* képlet nem használható közvetlenül, ezért a képletet egy ciklus segítségével kell kiszámoltatnunk. Ha a kezdetben 1-re állított eredményhez, sorba hozzászorozzuk a számokat (2-től *n*-ig), az eredményt mindig visszatéve az eredményt tároló változóba, akkor a végére éppen a kívánt szorzat áll elő ebben a változóban.

Funkció	Azonosító	Típus	Jelleg
N értéke	N	Egész	I
Az aktuális szorzat	FAKT	Egész	M, O
Az aktuális szorzó	I	Egész	M

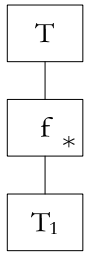



```
/* Faktoriális */
Be: N
FAKT ← 1
for I ← 2,N
       FAKT ← FAKT*I
Ki: FAKT
```

Megjegyzés: A faktoriális függvény egy, az exponenciális függvénynél is nagyobb mértékben növő függvény, így csak kis *n* értékekre számolható ki pontosan *n!* értéke. A megoldás programmá írásakor éppen ezért célszerű az eredményt tároló (FAKT) változót a „legnagyobb” egész típusúra vagy valós típusúra deklarálni.

7.3.3. Hátultesztelő iteráció

Struktúradiagram-jelölés



f : a ciklust vezérlő feltétel (logikai kifejezés)

T_1 : a ciklus magja (az ismételendő tevékenység)

Jelentés: A T tevékenység végrehajtása a T_1 tevékenység ismételt végrehajtását jelenti úgy, hogy amíg a ciklust vezérlő feltétel igazzá nem válik, addig végrehajtjuk a T_1 ciklusmagot. A feltétel értékét a ciklusmag végrehajtása után vizsgáljuk.

Pszeudokód-jelölés

```
repeat
        T1
until f
```

Az első sort a T helyére, a T_1 ciklusmag tevékenységeit beljebb, míg az utolsó sort az első sorral egyvonalba írjuk.

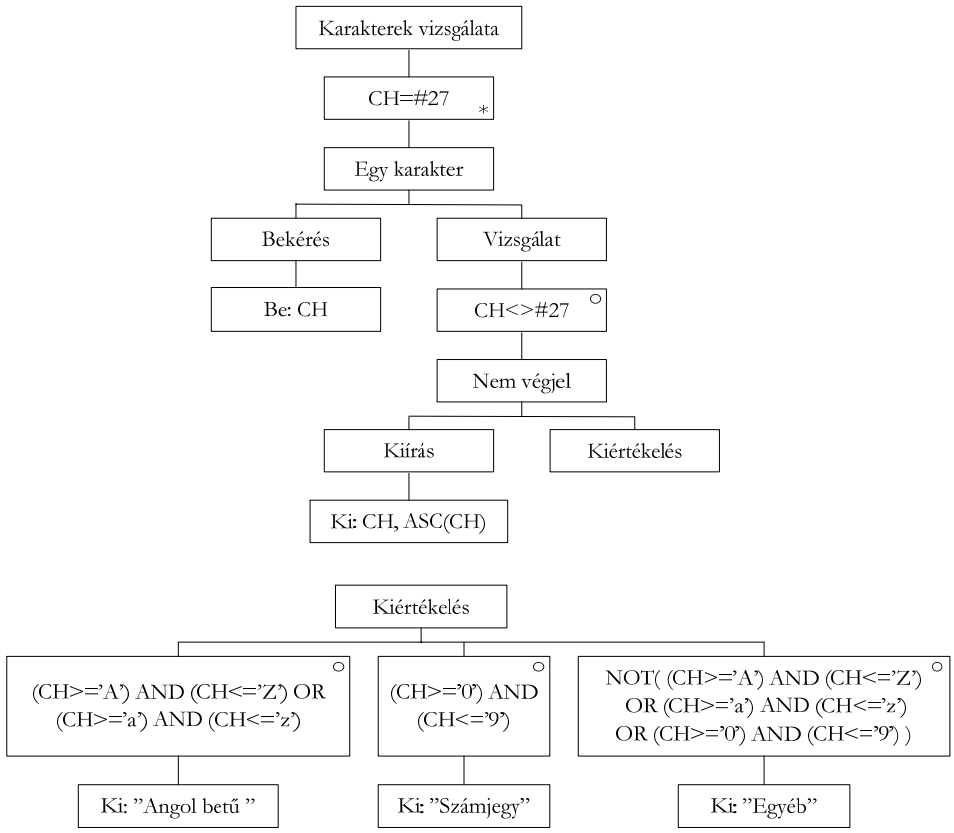
Megjegyzés

- A T_1 ciklusmagnak olyan tevékenységet is tartalmaznia kell, amely hatással van a ciklust vezérlő feltétel értékére (mert különben végtelen ismétlődés, végtelen ciklus keletkezhet – ha a feltétel értéke kezdetben hamis).
- A T_1 ciklusmag legalább egyszer végrehajtódik.

Feladat: Kérjünk be karaktereket az Esc billentyű (végjel) leütéséig és írjuk ki minden egyes karakterre a karakter ASCII kódját és azt, hogy angol betű, számjegy vagy egyéb karakter volt-e a megadott karakter!

Megoldás: Az ilyen jellegű feladatoknál, ahol az adatok száma nem ismert, ráadásul tetszőlegesen nagy lehet, az összes adat nem tárolható egy időben a memóriában (hiszen a memóriakapacitás is véges), következésképpen az adatok bekérése és feldolgozása nem különülhet el egymástól. Egy adatot, a bekérése után nyomban fel is dolgozzuk, így a következő adatot ugyanabba a változóba bekérhetjük.

Funkció	Azonosító	Típus	Jelleg
Az aktuális karakter	CH	Karakter	M




```
/* Karakterek vizsgálata */
Ki: "Karakterek vizsgálata (Kilépés:Esc)"
repeat
      /* Bekérés */
      Be: CH
      if CH<>#27
            /* Kiírás */
            Ki: CH,ASC(CH)
            /* Kiértékelés */
            if (CH>='A') AND (CH<='Z') OR (CH>='a') AND (CH<='z')
                  Ki: "Angol betű"
            else if (CH>='0') AND (CH<='9')
                  Ki: "Számjegy"
            else
                  Ki: "Egyéb"
until CH=#27
```

Megjegyzés

- Az aktuális karaktert tároló változó munka jellegű, annak ellenére, hogy értékét bekérjük és kiírjuk az algoritmus során, hiszen a feladat egésze szempontjából nem számít sem kiinduló (input), sem végeredmény (output) adatnak a benne tárolt adat.
- A feltételvizsgálatok halmazok és az IN művelet segítségével rövidebben is megadhatók (lásd 9.3.).
- A beolvasást célszerű olyan utasítással programozni, amelyik nem jeleníti meg a leütött karaktert a képernyőn (különben a kiírás miatt kétszer jelennek meg a karakterek).

8. Elemi feladatok

Ez a fejezet olyan feladatokat tartalmaz, amelyek megoldhatók egyszerű adattípusok használatával.

8.1. Prímfelbontás

Feladat: Adott egy *N* (>1) egész szám, mint input adat. Írjuk ki a szám prímtényezős felbontását!
Pl: 12 → 2*2*3

Megoldás: Felhasználva a legkisebb osztót megadó algoritmust (lásd 7.3.1.), a prímtényezőket a következőképpen kaphatjuk meg: megkeressük a szám legkisebb osztóját, kiírjuk, mint prímtényezőt, osztjuk vele a számot, majd újra keressük az új számra a legkisebb osztót, egészen addig, amíg az osztások eredményeként a szám eggyé nem válik. A legkisebb osztó keresést kezdetben 2-től kezdjük, majd mindig az előző osztó értékétől folytatjuk (hiszen a kisebb osztókat már megvizsgáltuk), és csak akkor lépünk a következő lehetséges osztóra, ha az adott osztó (már) nem osztja a számot. Hogy a példa szerinti eredményt írjuk ki, az utolsó osztó kivételével, az osztók után egy szorzásjelet is kiírunk. Azt, hogy egy osztó az utolsó volt-e vagy sem, a szám „elfogyása”, 1 értéke jelzi.

Funkció	Azonosító	Típus	Jelleg
A felbontandó szám	N	Egész	I, M
Az aktuális osztó	O	Egész	M, O

```
/* Prímfelbontás */
Be: N
O ← 2
while N>1
        if N MOD O=0
                N ← N DIV O
                if N=1
                        Ki: O
                else
                        Ki: O,'*'
        else
                O ← O+1
```

8.2. Monoton növő sorozat

Feladat: Adott N (>1) darab szám, mint input adat, ahol N is input adat. Kérjük be őket és mondjuk meg, hogy az adatmegadás sorrendjében monoton növők-e vagy sem!

Megoldás: Az ilyen jellegű feladatoknál, ahol azt kell eldöntenünk, hogy egy adott feltétel teljesül-e az összes adatra vagy sem, a következőképpen járhatunk el. Egy logikai változóban tároljuk az eredményt, azaz azt, hogy teljesül-e az adott feltétel az adatsorozat elemeire vagy sem. Kezdetben a változó értékét igazra állítjuk, majd sorban megvizsgáljuk az egyes elemeket. Az első olyan elemnél, amely megsérti az adott feltételt, az eredményt hamisra állítjuk, és a vizsgálatot befejezhetjük, hiszen az eredmény eldőlt. Ha a sorozat vizsgálata után igaz érték marad az eredményváltozóban, akkor a sorozat minden elemére teljesült a megadott feltétel.

Most, az adatbekérés miatt, nem állunk meg a vizsgálattal az első nem megfelelő elemnél, az összes adatot beolvassuk és feldolgozzuk, eredményt a teljes adatsor beolvasása után közlünk. A monoton növést olyan elem „sérti meg", amely kisebb, mint az őt megelőző elem. A vizsgálatot a második elemtől kezdve végezzük, hiszen az első elem értéke tetszőleges lehet.

Megjegyzés: Az adatsor elemei most ugyan valós számok, de a „monoton növés" értelmezhető minden olyan adatra, amelyekre értelmezettek a hasonlítások.

Funkció	Azonosító	Típus	Jelleg
Az adatok száma	N	Egész	I
Az aktuális adat	A	Valós	I
Az aktuális adatot megelőző adat	ELOZO	Valós	M
Monoton növő-e a sorozat	NOVO	Logikai	M, O
Ciklusváltozó	I	Egész	M

```
/* Monoton növő sorozat */
Be: N
/* Első adat */
Be: A
/* Kezdőértékek */
NOVO ← igaz
ELOZO ← A
```

```
/* Többi adat */
for I ← 2,N
        Be: A
        if A<ELOZO
                NOVO ← hamis
        ELOZO ← A
if NOVO
        Ki: "Monoton növők!"
else
        Ki: "Nem monoton növők!"
```

8.3. Pozitív adatok maximuma, átlaga

Feladat: Adott N db szám, mint input adat, ahol N is input adat. Kérjük be az adatokat és mondjuk meg a beérkezett pozitív adatok maximumát és átlagát!

Megoldás: Ha valamilyen szempont szerinti legjobb (pl. legkisebb, leghosszabb, stb.) adat megkeresése a feladat, akkor rendre meg kell vizsgálnunk valamennyi adatot. Ha az éppen vizsgálandó adat jobb, mint az addigi eredmény (azaz az addig megvizsgált elemek legjobbja), akkor ezt az adatot kell megjegyeznünk eredményként, ha nem, akkor nem változtatunk az eredményen. Az összes adat megvizsgálása után megkapjuk az eredményt.

A kérdés csupán az, hogy kezdetben mit jegyezzünk meg eredményként?

Ha az összes adat között keresünk, akkor egyszerűen jegyezzük meg az első adatot, és a vizsgálatot már csak a többi adaton végezzük el.

Ha az adatoknak csak egy részében keresünk (mint a mostani feladatban), akkor ez nem járható, hiszen lehet, hogy az első adat nem esik a megfelelő részbe, tehát abba, amelyek közül a legjobbat keressük. Ilyenkor vagy egy olyan kezdőértéket adunk az eredményt tároló változónak, amely módosul, amint beérkezik egy, a kívánt részbe eső adat (példánkban ilyen lehet egy negatív érték, hiszen az első pozitív szám nagyobb lesz nála), vagy egyszerűen számoljuk a megfelelő adatok darabszámát. Az első ilyen adatnál kezdőértéket állítunk, a többinél hasonlítunk.

Mivel az átlaghoz amúgy is meg kell határoznunk a pozitív adatok darabszámát (és persze összegét is), megoldásunkban az utóbbit választjuk. A darabszámot és az összeget gyűjtő változók kezdőértékét nullára állítjuk és minden egyes megfelelő adatnál (pozitív számnál), megnöveljük értéküket eggyel, ill. magával az adat értékével. Ha minden adatot bekértünk és feldolgoztunk, akkor az összeg és darabszám alapján meghatározható a

kívánt átlag. Természetesen, ha nem volt pozitív adat, akkor az átlaguk sem értelmezhető.

Funkció	Azonosító	Típus	Jelleg
Az adatok száma	N	Egész	I
Az aktuális adat	AKT	Valós	I
A pozitív adatok maximuma	MAX	Valós	M, O
A pozitív adatok átlaga	ATL	Valós	O
A pozitív adatok darabszáma	DB	Egész	M
A pozitív adatok összege	OSSZ	Valós	M
Ciklusváltozó	I	Egész	M

```
/* Pozitív adatok maximuma, átlaga */
Be: N
/* Kezdőértékek */
DB ← OSSZ ← 0
/* Adatbekérés, feldolgozás */
for I ← 1,N
       Be: AKT
       if AKT>=0
              DB ← DB+1
              OSSZ ← OSSZ+AKT
              if DB=1
                     MAX ← AKT
              else if AKT>MAX
                     MAX ← AKT
/* Eredménykiírás */
if DB=0
       Ki: "Nem volt pozitív adat!"
else
       ATL ← OSSZ/DB
       Ki: "A pozitív adatok maximuma:",MAX
       Ki: "A pozitív adatok átlaga:",ATL
```

8.4. e^x hatványsora

Feladat: Közelítsük e^x értékét a hatványsorának felhasználásával!

Megoldás: Mint ismeretes $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$. Ez az összeg természetesen nem számolható a végtelenségig, valahol abba kell hagyni, tekintettel időnk végességére és a számítógép tárolási pontosságára. Csak addig összegzünk,

amíg az általános tag egy megadott (nullánál nagyobb) ε pontosságnál abszolút értékben kisebb nem lesz. Ez előbb utóbb bekövetkezik, mivel az általános tag ($x^n/n!$) abszolút értéke határértékben a 0-hoz tart.

Az általános tag kiszámítása nem történhet úgy, hogy kiszámoljuk a számlálót is, meg a nevezőt is és elosszuk őket egymással, hiszen mindkét érték hamar kicsordulna a számítógép ábrázolási pontosságából (nem is beszélve az ismételt végrehajtásokról), ezért az általános tag értékét lépésenként, az előző tag értékéből számoljuk ki, egy egyszerű törttel (x/n) való szorzással. Az összegzést az első két tag összegével kezdjük, hogy az eredményben x is szerepeljen.

Megjegyzés
- Az, hogy az általános tag már elérte a megadott pontosságot, még nem feltétlenül jelenti azt, hogy e^x értékét is ilyen pontossággal közelítjük.
- Az exponenciális függvénnyel kiszámolt érték sem abszolút pontos, az is „csak” egy közelítő érték.

Funkció	Azonosító	Típus	Jelleg
x, ahol a függvényértéket közelítjük	X	Valós	I
A pontosság	EPSZ	Valós	I
A közelítés n lépésszáma	N	Egész	M, O
A közelítő érték	OSSZ	Valós	M, O
Az aktuális tag értéke	AKT	Valós	M

```
/* Az exponenciális függvény közelítése */
Be: X,EPSZ
/* Kezdőértékek */
N ← 1
AKT ← X
OSSZ ← 1+AKT
/* Közelítés */
while ABS(AKT)>=EPSZ
      N ← N+1
      AKT ← AKT*X/N
      OSSZ ← OSSZ+AKT
/* Eredménykiírás */
Ki: ”N értéke:”,N
Ki: ”A közelítő érték:”,OSSZ
Ki: ”A ’pontos’ érték:”,EXP(X)
```

8.5. Gyökkeresés intervallumfelezéssel

Feladat: Határozzuk meg egy *f(x)=0* egyenlet egy gyökét, egy adott *[a, b]* intervallumban, adott *ε* pontossággal!

Megoldás: Feltesszük, hogy az *f(x)* függvény folytonos az *[a, b]* intervallumban és az intervallum két végpontjában ellentétes előjelű. Ebben az esetben ugyanis biztosan van legalább egy gyök az *[a, b]* intervallumban és az intervallumfelezéses megoldási módszer alkalmazható. Felezzük meg az intervallumot! Ha a felezőpontban felvett függvényérték 0, akkor szerencsénk van, megkaptuk a keresett gyököt, ha nem, vizsgáljuk meg, hogy melyik végpontban felvett függvényértékkel azonos előjelű. Ha *f(a)*-val egyező előjelű, akkor hagyjuk el az intervallum első felét, ha *f(b)*-vel, akkor a másodikat, ha 0, akkor mindkét felét, a felezőpontot kivéve. A gyök biztosan a megmaradó intervallumban lesz. Az új intervallumra ismételjük meg a fenti tevékenységeket egészen addig, amíg az intervallum hossza rövidebb nem lesz a megadott (nullánál nagyobb) pontosságnál. Ebben az esetben a maradék intervallum bármelyik pontját elfogadhatjuk (pl. a felezőpontot) az egyenlet gyökeként.

Megjegyzés: A konkrét *f(x)* függvény a forrásprogramba beírandó, a mellékelt (lásd függelék) megoldásban ez az *f(x)=sin(x)* függvény.

Funkció	Azonosító	Típus	Jelleg
Az intervallum kezdőpontja	A	Valós	I
Az intervallum végpontja	B	Valós	I
A közelítés pontossága	EPSZ	Valós	I
A kiszámított közelítő érték	GYOK	Valós	O
A vizsgált intervallum kezdőpontja	XK	Valós	M
A vizsgált intervallum végpontja	XV	Valós	M
A vizsgált intervallum felezőpontja	XF	Valós	M
Függvényérték a kezdőpontban	YK	Valós	M
Függvényérték a végpontban	YV	Valós	M
Függvényérték a felezőpontban	YF	Valós	M

```
/* Gyökkeresés intervallumfelezéssel */
/* Adatbekérés */
Be: A,B,EPSZ
/* Kezdőértékek */
XK ← A
XV ← B
YK ← f(A)
YV ← f(B)
/* Közelítés */
while XV-XK>EPSZ
      /* Felezőpont */
      XF ← (XK+XV)/2
      YF ← f(XF)
      /* Csökkentés */
      if YK*YF<=0
            XV ← XF
            YV ← YF
      if YV*YF<=0
            XK ← XF
            YK ← YF
/* Eredmény */
GYOK ← (XK+XV)/2
/* Eredménykiírás */
Ki: "A gyök közelítő értéke:",GYOK
```

8.6. Integrálérték meghatározása közelítéssel

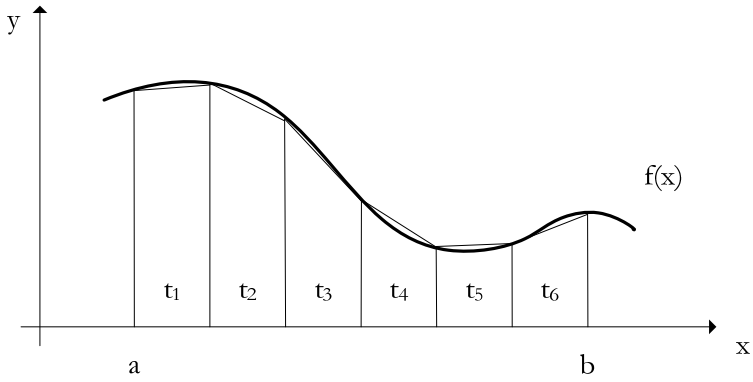
Feladat: Számítsuk ki egy *f(x)* függvény, adott *[a, b]* intervallumban vett határozott integrálját!

Megoldás: Osszuk be az *[a, b]* intervallumot *n* egyenlő részre, majd húzzunk az *y* tengellyel párhuzamos egyeneseket ezekben az osztáspontokban. Kössük össze a szomszédos párhuzamosok és a függvény metszéspontjának pontjait (lásd 8.1. ábra). Olyan trapézrendszert kapunk, amelynek területösszege a függvény határozott integrálját közelíti. A beosztást finomítva (pl. minden részintervallumot megfelezve), a trapézok területösszege a határozott integrál még jobb közelítését adja. A megoldás tehát úgy fogalmazható, hogy készítsük el azt a sorozatot, amely a finomodó beosztáshoz tartozó trapézok területösszegéből áll, ez a sorozat határértékben a keresett integrálértékhez tart, és ha két egymást követő területösszeg alig tér el egymástól (az eltérés abszolút értéke kisebb, mint ε), akkor a határozott integrál közelítéseként az utolsónak előállított összeget fogadjuk el.

Formalizálva, ha az intervallumot n egyenlő részre osztjuk, az i-edik osztáspontban felvett függvényértéket y_i -vel jelöljük, akkor a trapézok összege így írható fel:

$$T = h(\frac{y_0 + y_n}{2} + \sum_{i=1}^{n-1} y_i)$$

ahol y_0 az a-ban, y_n a b-ben felvett függvényérték és $h=(b-a)/n$.



8.1. ábra. A trapéz-módszer.

Megjegyzés: A konkrét $f(x)$ függvény a forrásprogramba beírandó, a mellékelt (lásd függelék) megoldásban ez az $f(x)=3x^2$ függvény.

Funkció	Azonosító	Típus	Jelleg
Az intervallum kezdőpontja	A	Valós	I
Az intervallum végpontja	B	Valós	I
A közelítés pontossága	EPSZ	Valós	I
A beosztás finomsága	N	Egész	M
A részintervallumok hossza	H	Valós	M
Az aktuális területösszeg	T	Valós	M, O
Az első ill. az előző területösszeg	E	Valós	M
Függvényérték az a helyen	Y0	Valós	M
Függvényérték a b helyen	YN	Valós	M
Ciklusváltozó	I	Egész	M

```
/* Határozott integrál közelítése trapéz-módszerrel */
/* Adatbekérés */
Be: A,B,EPSZ
/* Kezdőértékek */
Y0 ← f(A)
YN ← f(B)
T ← (B-A)*(Y0+YN)/2
/* Következő beosztás */
N ← 2
/* Közelítés */
repeat
       /* Előző területösszeg */
       E ← T
       /* Részintervallumok hossza */
       H ← (B-A)/N
       /* Trapézok területösszege */
       T ← (Y0+YN)/2
       for I ← 1,N-1
              T ← T+f(A+I*H)
       T ← T*H
       /* Következő beosztás */
       N ← 2*N
until ABS(T-E)<EPSZ
/* Eredménykiírás */
Ki: "Az integrál közelítő értéke:",T
```

8.7. Feladatok

- Számítsuk ki két adott, egy napon belüli időpont között eltelt időt! Az időpontokat és az eredményt is óra, perc, másodperc alakban adjuk meg.
- Adott n és k pozitív egész számokra határozzuk meg a binomiális együttható értékét!
- Határozzuk meg két pozitív egész szám legkisebb közös többszörösét!
- Írjuk ki egy inputként megadott, egynél nagyobb, pozitív egész szám prímtényezős felbontását úgy, hogy használjuk a '^' karaktert a hatványozás jelzésére!
  Pl. 12 → 2^2*3
- Döntsük el egy pozitív egész számról, hogy prímszám-e vagy sem!
- Az előző feladat segítségével írjuk ki egy adott *[a, b]* intervallumba eső prímszámokat!
- Döntsük el két pozitív egész számról, hogy relatív prímek-e vagy sem! Két szám relatív prím, ha 1-en kívül nincs más közös osztójuk.

- Az előző feladat segítségével írjuk ki egy adott *[a, b]* intervallumba eső összes relatív prím számpárokat!
- Adott *n* pozitív egész számra írjuk ki a *Fibonacci sorozat* első n elemét! A képzés módszere: az első két elem értéke 1, ezután minden következő az előző kettő összege (Pl. *n*=6: 1, 1, 2, 3, 5, 8).
- Oldjuk meg az $ax+by=c$ és $dx+ey=f$ egyenletekből álló, kétismeretlenes, lineáris egyenletrendszert, ahol *a, b, c, d, e, f* értékek valós számok!
- Adott *N* db szám, mint input adat, ahol *N* is input adat. Kérjük be az adatokat és mondjuk meg, hogy hány elemből áll a leghosszabb monoton növő elemsorozat!
  Pl. *N*=5 2, 1, 3, 2, 1 → 2
- Adott *N* db szám, mint input adat, ahol *N* is input adat. Kérjük be az adatokat és határozzuk meg az alábbiakat!
  - Az adatok hány százaléka pozitív.
  - Az adatok a beolvasás sorrendjében számtani sorozatot alkotnak-e vagy sem.
  - Az átlagukat úgy, hogy a maximális és minimális érték egy-egy előfordulását kihagyjuk az átlagszámításból.
- Adott valamennyi számadat, mint input adat. Az adatok számát nem ismerjük, az adatsor végét a 0 végjel (nem adat) jelzi. Kérjük be az adatokat és határozzuk meg az alábbiakat!
  - Négyzetes átlag (négyzetösszegből vont négyzetgyök, osztva az adatok számával).
  - Az adatok a beolvasás sorrendjében mértani sorozatot alkotnak-e vagy sem.
- Adott *N* db pont a síkon a koordinátáival, mint input adat, ahol *N* is input adat. Kérjük be az adatokat és határozzuk meg az alábbiakat!
  - Az adatok súlypontja (a pontok *X* koordinátáinak átlaga adja a súlypont *X* koordinátáját, az *Y* koordináták átlaga pedig a súlypont *Y* koordinátáját).
  - Az origótól legtávolabb lévő pont (feltesszük, hogy csak egy ilyen pont van a pontok között).
- Adott *N* db pont a síkon a koordinátáival, mint egy *N* csúcsú konvex sokszög csúcspontjai valamely körüljárási irány szerint, ahol *N* is input adat. Kérjük be az adatokat és határozzuk meg a sokszög kerületét!

- Készítsük el az *y=sinx* függvény értéktáblázatát adott *[a, b]* intervallumban, adott lépésközzel a számítógép monitorán úgy, hogy minden teleírt képernyő után várjunk billentyűleütésre!
- Adott *x* pozitív számhoz határozzuk meg azt a legkisebb *n* egész számot, amelyre igaz, hogy 1+1/2+1/3+…+1/*n*>=*x*. (A sor monoton növő és nem korlátos, tehát „elvileg" van megoldás.)
- Készítsünk algoritmust egy *f(x)* függvény, adott *[a, b]* intervallumon vett közelítő integráljának meghatározására a téglalap-módszerrel, az intervallum egyenletes beosztásának fokozatos finomításával, egészen addig, amíg az alsó és felső közelítő összeg eltérése kisebb nem lesz, mint egy adott $\varepsilon > 0$ szám. Feltételezhetjük, hogy a függvény az intervallumon monoton növő vagy csökkenő, így részintervallumonként egyértelműen (az intervallum végpontjaiban vett függvényértékekkel, mint magasságokkal) képezhetünk két, egy kisebb és egy nagyobb területű téglalapot. A kisebb téglalapok területösszege az alsó, a nagyobbaké a felső közelítő összeget adják.

9. Összetett adattípusok

Az eddig megismert adattípusokkal ellentétben, ahol egy változóban csak egy adatot tárolhatunk egy időben, az összetett adattípusok segítségével több adat együttes kezelésére is lehetőségünk nyílik. A következő fejezetekben ezekkel az adattípusokkal ismerkedünk meg.

9.1. Tömbök

A tömb egy általánosan és gyakran használt eszköz a szoftverfejlesztők, programozók körében, mivel a tömbökkel kényelmesen kezelhetünk több, azonos típusú adatot. Az adatok elérésére, a tömbök egy elemének kiválasztására sorszámokat, más néven indexeket használunk.

Ha ez elemeket egy sorszámmal azonosítjuk, akkor egydimenziós tömbről, ha két sorszámmal, akkor kétdimenziós tömbről és így tovább, ha *n* db sorszámmal azonosítjuk, akkor *n* dimenziós tömbről beszélünk. Egy tömb egy elemének kiválasztásához tehát pontosan annyi index szükséges, ahány dimenziós az adott tömb.

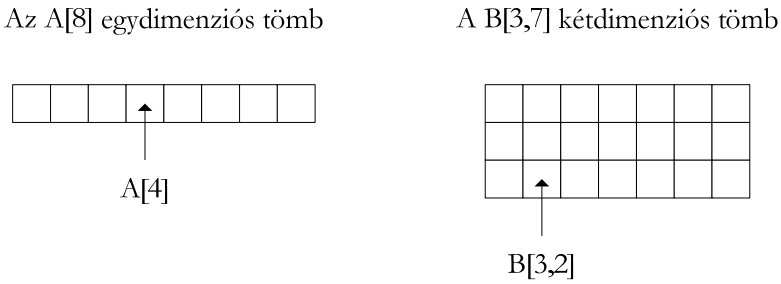
Ahogy az egyszerű adattípusok esetén is megkülönböztettük az adatot (pl. 3) az őt tároló változótól (pl. I) és annak típusától (pl. egész), itt is kitérünk ezek különbözőségére. A tömb elnevezés ugyanis, a rövidsége miatt, mind a három esetben használatos, így a tömb lehet:

- *Tömbadat*: amely egydimenziós tömb esetén egy adatsornak, vagy matematikai fogalommal egy vektornak, kétdimenziós tömb esetén egy adattáblázatnak vagy mátrixnak felel meg.
- *Tömb adattípus*: amely definiálja a tömb dimenzióit és a tömb elemeinek típusát.
- *Tömbváltozó*: amelyben az adatok tárolódnak. Egy elem kiválasztásával, azaz a tömbváltozó indexelésével egy, a tömb elemtípusával megegyező típusú változót hivatkozunk.

A tömbök indexelésére 1-től kezdődő sorszámokat használunk, azaz minden dimenzió legkisebb indexe 1. Az egyes dimenziók legnagyobb indexét (a tömb méretét ill. méreteit), ha ez a feladat szempontjából lényeges, deklaráláskor adjuk meg. Ha az egyszerűség kedvéért ezt elhagyjuk, akkor a megoldás programmá írásakor olyan tömböt kell használnunk, amelyben elférnek az adataink.

A programnyelvek általában nem korlátozzák a dimenziók számát, azaz használhatunk három, négy, … stb. dimenziós tömböket is, a gyakor-

latban (gondoljunk például a papíron történő „információcserére”) az egydimenziós (adatsor) és kétdimenziós (adattáblázat) tömbök a leggyakoribbak, mi is ezeket tárgyaljuk.



9.1. ábra. Tömbök és elemeik.

Deklarálás

A tömbváltozókat (csakúgy, mint az egyszerű változókat) használat előtt deklarálni kell, azaz definiálnunk kell a tömb dimenziót (az indexek felső határait) és a tömb elemeinek típusát.

A fordítóprogram ezek alapján lefoglalja a szükséges méretű (elemek száma*egy elem mérete) memóriaterületet a változó számára.

Az adatokat általában elölről kezdve, folyamatosan tároljuk a tömbben, az első adatot az első elemben, a másodikat a másodikban és így tovább.

Természetesen azt, hogy hány adatot tárolunk a tömbben, adminisztrálnunk kell.

Megjegyzés

- A Pascal nyelv nemcsak az egész számokkal történő indexelést engedi meg, hanem tetszőleges, ún. sorszámozott típusú adattal is indexelhetünk (pl. karakterrel). A karakterek indexként való felhasználását a C nyelv is megengedi, ekkor a karakter ASCII kódja adja a megfelelő indexet. Ezzel a lehetőséggel bizonyos feladatok könnyebben, egyszerűbben programozhatók, de megoldásainkban az egyszerűség kedvéért ezeket nem használjuk (nem engedjük meg).
- A C nyelv a tömböket 0-tól kezdődően indexeli, amit a megoldások programmá írásakor figyelembe kell vennünk.

- Vannak olyan feladatok (lásd 9.1.2.), ahol egy bizonyos adat csak a tömb egy bizonyos elemében tárolható, ekkor az elölről feltöltött, folyamatos adattárolás és darabszám adminisztrálás nem alkalmazható.
- A tömbök elemei összetett adattípusúak is lehetnek (pl. halmazok, rekordok, sztringek). Ha egy egydimenziós tömb elemeinek típusa szintén egy tömb, akkor ezt nem tömbökből álló tömbként, hanem egy kétdimenziós tömbként fogjuk használni.

Egy tömb deklarálható az adatszerkezeti táblázatban, de ha többször is szükség van ugyanolyan tömbökre, célszerű előtte, külön típusként deklarálni. A következő példában mindkét deklarálást bemutatjuk.

Konstans
SORMAX 10
OSZLMAX 20
Típus
MATRIX Kétdimenziós egész tömb[SORMAX, OSZLMAX]

Funkció	Azonosító	Típus	Jelleg
Egy adatsor	A	Egydimenziós valós tömb	I
Az adatsor adatainak száma	N	Egész	I
Egy ötöslottó szelvénye	L	Egydimenziós egész tömb[5]	M, O
Egy mátrix	B	MATRIX	I, M, O
A mátrix sorainak száma	S	Egész	I
A mátrix oszlopainak száma	O	Egész	I

Az A valós tömb méretét nem definiáltuk (ha a feladat szempontjából a méret nem fontos, akkor elhagyható), az L tömbben öt darab lottószámot, míg a B tömbben max. 10*20 db egész számot tárolhatunk.

A kétdimenziós tömbök első indexét a sorindexnek, a másodikat az oszlopindexnek szokás használni.

Hivatkozás

A tömbök elemeire a tömbváltozó azonosítója után, szögletes zárójelbe tett, egymástól vesszővel elválasztott indexekkel hivatkozunk. Az indexek tetszőleges egész kifejezések lehetnek.

Pl. A[1] az A tömb első eleme,
A[I+1] az A tömb I+1. eleme,
B[S, 2] a B tömb S. sorának 2. eleme.

Egy indexkifejezés eredményének „kicsordulása” a hozzá tartozó dimenzió tartományából természetesen hibának számít, hibás programot eredményez és programleállást is kiválthat.

Konstans
Tömbök esetén is megengedjük a konstansok használatát, amellyel több, azonos típusú konstanst deklarálhatunk.
Pl. SZINEK=(”Treff”, ”Káró”, ”Kőr”, ”Pikk”)
MTX=((1,2,3),(4,5,6))
A fenti példákban egy négyelemű, sztringekből álló, egydimenziós és egy 2×3-as, egész számokból álló, kétdimenziós tömbkonstanst deklaráltunk. A tömbelemekre való hivatkozás ugyanúgy történik, mint tömbváltozóknál, azaz pl. a SZINEK[1] hivatkozással a ”Treff” sztring, az MTX[2,3] hivatkozással a 6 érhető el.

Műveletek
A tömbökre nem értelmezünk műveleteket, míg az egyes tömbelemekkel minden olyan művelet megengedett, ami a tömb elemtípusára definiált.

Megjegyzés
- A C nyelvben egy többdimenziós tömb egy elemére való hivatkozáskor minden indexet külön szögletes zárójelbe kell tenni. Az általunk használt, egyszerűbb jelölés (az indexek vesszővel elválasztva egy szögletes zárójelben) a C nyelvben más jelentéssel bír.
- A megoldások programmá írására használt, DOS platformú fejlesztőrendszerek (a memóriacímzésből fakadóan) korlátozzák a változók, így a tömbváltozók maximális méretét is. Ez a korlát 64 Kbájt, így ezt figyelembe kell vennünk a tömbök használatánál.
- A példáinkban szereplő tömbök maximális mérete fordítási időben meghatározódik, azaz a program futásakor ezek már nem változtathatók. Az olyan tömböket, amelyek mérete futás közben állítható, *dinamikus tömböknek* nevezzük (lásd 13.1.). Az ilyen tömbök „mögött” valójában a dinamikus adatstruktúrák, ill. a dinamikus tárkezelés „rejtőzik”.

9.1.1. Átlagnál nagyobb elemek

Feladat: Adott *N* db szám, mint input adat. Kérjük be őket és írjuk ki az átlagnál nagyobb adatokat! Az adatok száma is input adat, értéke legfeljebb 100.

Megoldás: A feladat megoldásához kétszer kell az adatsoron „végigmenni", egyszer, amikor az átlaghoz összegezzük őket, majd amikor az átlagnál nagyobb elemeket kiírjuk. Az adatokat ezért egy tömbben tároljuk, így az adatok bekérése és feldolgozása elkülönülhet egymástól. Az adatok bekérésekor először az adatok számát kérjük be, majd egy növekményes ciklussal magukat az adatokat. Ezeket egy tömbben tároljuk, elölről kezdve folyamatosan, az *i.* adatot a tömb *i.* elemében. A tömb deklarált méretét az adatok maximális száma adja.

Konstans
NMAX 100

Funkció	Azonosító	Típus	Jelleg
Az adatok száma	N	Egész	I
Az adatok	A	Egydimenziós valós tömb[NMAX]	I
Az adatok összege	OSSZ	Valós	M
Az adatok átlaga	ATL	Valós	M
Ciklusváltozó	I	Egész	M

```
/* Átlagnál nagyobb elemek */
/* Adatbekérés */
Be: N
for I ← 1,N
      Be: A[I]
/* Átlagszámolás */
OSSZ ← 0
for I ← 1,N
      OSSZ ← OSSZ+A[I]
ATL ← OSSZ/N
/* Eredménykiírás */
for I ← 1,N
      if A[I]>ATL
            Ki: A[I]
```

9.1.2. Kockadobások gyakorisága

Feladat: Input adatként adott egy 6 oldalú, szabályos dobókockával dobott számsorozat. A dobások számát nem ismerjük, ez tetszőlegesen nagy lehet, a sorozat végét a 0 szám (mint végjel) jelzi. Kérjük be az adatokat és készítsünk gyakorisági táblázatot az egyes dobások előfordulási darabszámáról!

Megoldás: Mivel az adatok száma tetszőlegesen nagy lehet, magukat a dobott számokat nem tudjuk tömbben tárolni (hiszen mekkorára deklarálnánk), de erre nincs is szükség. Az eredményt fogjuk tömbben tárolni, mégpedig egy olyan 6 elemű tömbben, ahol az *i*. helyen az *i*, mint dobott szám darabszámát gyűjtjük. A darabszámokat kezdetben nullára állítjuk, majd egy dobás bekérésekor a megfelelő tömbelem értékét, azaz a hozzá tartozó darabszámot megnöveljük eggyel. A dobássorozat ill. adatmegadás végén (a 0 végjel beérkezése után) kiírjuk az eredményeket. A bekérést egy hátultesztelő ciklussal végezzük, hiszen legalább egy adatot be kell kérnünk. A végjelet nem dolgozzuk fel kockadobásként, az adatbekérést a dobás sorszámának kiírásával segítjük.

Konstans

N 6

Funkció	Azonosító	Típus	Jelleg
Az egyes dobások darabszáma	DB	Egydimenziós egész tömb[N]	M, O
Az aktuális dobás	A	Egész	I
Segédváltozó	I	Egész	M

```
/* Kockadobások gyakorisága */
Ki: "Kockadobások (1-",N,") gyakorisága (Kilépés:0)"
/* Kezdőértékek */
for I ← 1,N
      DB[I] ← 0
I ← 0
repeat
      /* Bekérés */
      I ← I+1
      Ki: I, ". dobás:"
      Be: A
      if A<>0
            DB[A] ← DB[A]+1
until A=0
Ki: "Gyakoriságok"
for I ← 1,N
      Ki: I,DB[I]
```

9.1.3. Érték törlése adatsorból

Feladat: Töröljünk ki egy *N* elemű adatsorból egy adott értéket, ahányszor csak előfordul!

Megoldás: Az adatsor tárolására tömböt használunk. A vizsgálatot a tömb első eleménél kezdjük, és az *N*-ediknél fejezzük be. Ha az aktuálisan vizsgált elem értéke megegyezik a törlendő értékkel, akkor azt egyszerűen „átugorjuk", vesszük a következő elemet, különben „előre másoljuk" az értéket a tömbben. Így a tömb „elejére" kerül majd a kívánt (az adott értéket már nem tartalmazó) adatsor, az adatok eredeti sorrendjében. Az előremásolt értékeket számoljuk (ezzel a végén az eredménytömb elemeinek számát kapjuk), így a soronkövetkező, előremásolandó elem új helyét is ismerni fogjuk, hiszen az éppen eggyel nagyobb.

Funkció	Azonosító	Típus	Jelleg
Az adatsor	A	Egydimenziós, tetszőleges elemtípusú tömb	I, M, O
Az adatok száma	N	Egész	I, O
A törlendő érték	X	Az A tömb elemeivel megegyező típusú	I
Az aktuálisan vizsgált tömbelem indexe	I	Egész	M
Az előremásolt elemek száma	J	Egész	M

```
/* Érték törlése adatsorból */
TOROL(A,N,X)

/* Kezdőérték */
J ← 0
/* Előremásolás */
for I ← 1,N
      if A[I]<>X
            J ← J+1
            A[J] ← A[I]
/* Darabszám */
N ← J
```

Megjegyzés: A tetszőleges elemtípusú jelen esetben azt jelenti, hogy az algoritmus működik az összes olyan adatsorra, amelynek elemeire értelmezettek a hasonlítás műveletek. A konkrét elemtípust a feldolgozandó adatsor elemeinek típusa határozza meg.

9.1.4. Eratosztenesz szitája

A most ismertetésre kerülő, prímszámok előállítására szolgáló algoritmust Eratosztenesz időszámításunk előtt kb. 200 évvel definiálta:

Írjuk fel az 1-nél nagyobb természetes számokat egy tetszőlegesen nagy *K* korlátig. A legkisebb szám 2, jelöljük meg és töröljük a 2-vel oszthatókat a 2 kivételével. A megmaradó, jelöletlen számok közül a legkisebb 3, ezzel végezzük el ugyanezt az eljárást: jelöljük meg és töröljük a 3-mal oszthatókat a 3 kivételével. A megmaradó, jelöletlen számok közül a legkisebbel mindig elvégezve az eljárást, előbb-utóbb „elfogynak" a számok, vagy töröljük őket, vagy megjelöljük. A megjelölt, a szitán mintegy „fennmaradó" számok éppen a *K*-nál nem nagyobb prímszámok.

Feladat: Készítsünk megoldást a fenti algoritmus megvalósítására tömb segítségével!

Megoldás: Egy olyan logikai tömböt fogunk használni a megoldás során, amelyben a tömb *i*. eleme azt fejezi ki, hogy az *i* mint szám még fent van-e a szitán, vagy már töröltük. A számok megjelölése nem szükséges, mert minden számot csak egyszer veszünk figyelembe (lásd a szitálás külső ciklusát). Az egyszerűség kedvéért a tömb K elemű lesz, tehát az első elemet (az 1-nek megfelelő értéket) nem használjuk.

Megjegyzés: A K korlát értéke természetesen most nem lehet tetszőlegesen nagy, értékét a tömb helyfoglalására vonatkozó (64Kb-os) korlát limitálja.

Konstans:

```
K  1000          /* Eddig határozzuk meg a prímszámokat */
```

Funkció	Azonosító	Típus	Jelleg
A felírt számokat reprezentáló tömb	A	Egydimenziós logikai tömb[K]	M, O
Az aktuálisan vizsgált szám	P	Egész	M
A többszörösök törléséhez	I	Egész	M

```
/* Eratosztenesz szitája tömbbel */
/* Számok felírása */
for I ← 2,K
      A[I] ← igaz
/* Szitálás */
for P ← 2,K
      /* A P szám prímszám? */
      if A[P]
            /* Igen, töröljük a többszöröseit */
            for I ← 2*P,K,P
                  A[I] ← hamis
/* Kiírás */
for I ← 2,K
      if A[I]
            Ki: I
```

9.1.5. Mátrixösszegek

Feladat: Adott egy *N*×*M*-es számmátrix. Határozzuk meg a mátrix sorainak, oszlopainak, valamint az összes elemének az összegét!

Megoldás: A mátrix elemein egyszer, sorfolytonosan (tehát először az első sorának elemein növekvő oszlopindex szerint, azután a második soron, és így tovább) végighaladva gyűjteni tudjuk mind a sorok, mind az oszlopok, mind a teljes mátrix összegét. Mivel *N* db sor van, ezért *N* db sorösszeg keletkezik, így azokat egy egydimenziós tömbben tároljuk. Az oszlopösszegekkel hasonló a helyzet, csak ott *M* db oszlopösszeg keletkezik. A teljes összeg gyűjtésére elegendő egy egyszerű változó. Itt nem magukat az elemeket fogjuk összegezni, hanem a sorösszegeket, (lehetne az oszlopösszegeket is) hiszen így csökkenthetjük az összeadások számát.

Megjegyzés: A mátrix maximális méreteire konstansokat definiálunk, megkönnyítve ezzel az eredménytömbök deklarálását.

Konstans

```
NMAX  10          /* Sorok maximális száma */
MMAX  10          /* Oszlopok maximális száma */
```

Funkció	Azonosító	Típus	Jelleg
A feldolgozandó mátrix	A	Kétdimenziós valós tömb[NMAX, MMAX]	I
A mátrix sorainak száma	N	Egész	I
A mátrix oszlopainak száma	M	Egész	I
A sorösszegek	SOR	Egydimenziós valós tömb[NMAX]	M, O
Az oszlopösszegek	OSZL	Egydimenziós valós tömb[MMAX]	M, O
A teljes mátrix összeg	OSSZ	Valós	M, O
Segédváltozók	I, J	Egész	M

```
/* Mátrixösszegek */
OSSZEGEK(A,N,M,SOR,OSZL,OSSZ)

/* Kezdőértékek */
for J ← 1,M
       OSZL[J] ← 0
OSSZ ← 0
/* Összegzés */
for I ← 1,N
       SOR[I] ← 0
       for J ← 1,M
              SOR[I] ← SOR[I]+A[I,J]
              OSZL[J] ← OSZL[J]+A[I,J]
       OSSZ ← OSSZ+SOR[I]
```

9.1.6. Oszlopok törlése mátrixból

Feladat: Adott egy *N*×*M*-es mátrix. Töröljük ki a mátrix azon oszlopait, amelyekben minden elem azonos!

Megoldás: Sorban megvizsgáljuk a mátrix oszlopait. Ha egy oszlop minden eleme egyforma, akkor kitöröljük a mátrixból. A törlés egyik lehetséges módja az, hogy az utolsó oszloppal felülírjuk a törlendő oszlopot, majd csökkentjük az oszlopok számát eggyel. Ez a törlés azonban „átrendezi" a mátrix oszlopait. Ahhoz, hogy a megmaradó oszlopok sorrendje ne változzon meg (amelyik előbb volt az eredeti mátrixban, az eredmény mátrixban is előrébb legyen), a törlést a törlendő oszlop mögötti oszlopok „előremásolásával" végezzük. Először a törlendő oszlop mögötti oszlopot másoljuk egy oszloppal előrébb, majd az azutánit, és így tovább, legvégül az utolsót. Ezután csökkentjük az oszlopok számát eggyel. Mindkét törlés-

fajta esetén, ha éppen az utolsó oszlop a törlendő, akkor természetesen elegendő az oszlopszám csökkentése (hiszen ekkor nem kell elemeket mozgatnunk a mátrixban). Annak eldöntése, hogy egy oszlop csupa egyforma elemeket tartalmaz-e vagy sem, szintén elvégezhető többféleképpen. Hasonlíthatjuk például a szomszédos elemeket, de hasonlíthatjuk az elemeket az oszlop egy adott (pl. legelső) eleméhez (ahogy azt a megoldásban tettük). Ha találunk eltérő elempárt, akkor az oszlopban nem lehet minden elem egyforma.

Funkció	Azonosító	Típus	Jelleg
A feldolgozandó mátrix	A	Kétdimenziós, tetszőleges elemtípusú tömb	I, O
A mátrix sorainak száma	N	Egész	I
A mátrix oszlopainak száma	M	Egész	I, O
Az aktuálisan vizsgált oszlop indexe	J	Egész	M
A vizsgált oszlop elemei egyformák-e	EGYF	Logikai	M
Az aktuálisan előremásolandó oszlop indexe	K	Egész	M
Segédváltozó	I	Egész	M

```
/* Egyforma elemű oszlopok törlése */
TOROL(A,N,M)

J ← 1
while J<=M
        /* A J. oszlop vizsgálata */
        EGYF ← igaz
        I ← 2
        while (I<=N) AND EGYF
                EGYF ← A[I,J]=A[1,J]
                I ← I+1
        if EGYF
                /* A J. oszlop törlése */
                for K ← J+1,M
                        for I ← 1,N
                                A[I,K-1] ← A[I,K]
                M ← M-1
        else
                J ← J+1
```

Megjegyzés

- A tetszőleges elemtípusú jelen esetben azt jelenti, hogy az algoritmus működik az összes olyan mátrixra, amelynek elemeire értelmezettek a hasonlítás műveletek. A konkrét elemtípust a feldolgozandó mátrix elemeinek típusa határozza meg.
- Vegyük észre, hogy az oszlopokon haladó külső ciklus nem növekményes, így mindig az aktuális oszlopszámhoz hasonlítunk, hiszen egy oszlop törlésével az oszlopok száma is megváltozik.
- Az adott oszlop (J) elemeinek egyezését vizsgáló ciklusban az EGYF változónak egy hasonlítás eredményét adtuk értékül.
- Az utolsó oszlop (M.) törlésekor nem történik elemmozgatás (a K ciklusváltozójú növekményes ciklus kezdőértéke ekkor nagyobb, mint a végértéke).
- Ha egy oszlopot kitörlünk, akkor nem „lépünk tovább” egy oszloppal, hiszen az előremásolással „helyére lépő” oszlopot is meg kell vizsgálnunk.

9.1.7. Feladatok

- Adott egy pozitív egész szám, mint egy forintban kifizetendő pénzösszeg. Fizessük ki a legkevesebb címlet felhasználásával!
  Pl. 204005 → 20000: 10db, 2000: 2db, 5: 1 db
- Adott *N* db pozitív egész szám, mint forintban kifizetendő pénzösszegek. Határozzuk meg, hogy a teljes kifizetésükhöz, az egyes címletekből hány darab kell, ha a legkevesebb címlettel akarunk fizetni!
- Adott egy *N* elemű adatsorozat. Számoljuk meg, hogy a sorozat egymást követő elemei hány darab monoton növekvő részsorozatot alkotnak! Részsorozat kezdődik az első elemnél és minden olyan elemnél, amely kisebb, mint az őt megelőző elem.
  Pl. 1, 2, 3, 2, 1, 2, 2, 3, 4 → 3 db
- Határozzuk meg egy adatsorozat leghosszabb monoton növő részsorozatát! Ha több ilyen van a sorozatban, akkor a legelsőt adjuk eredményül!
  Pl. 3, 2, 4, 5, 1, 3, 4 → 2, 4, 5
- Adott két, monoton növő adatsorozat. Válogassuk őket egy adatsorba úgy, hogy az összeválogatott elemek is monoton növők legyenek!
  Pl. 2, 4, 5 és 1, 3 → 1, 2, 3, 4, 5

- Végezzük el az előző összeválogatást úgy, hogy az eredmény csak a különböző elemeket tartalmazza, azaz a többször előforduló elemeket csak egyszer vegyük figyelembe!
  Pl. 2, 4, 5 és 1, 1, 2, 3 → 1, 2, 3, 4, 5
- Adott egy növekvően rendezett adatsor és egy adat. Soroljuk be a rendezett elemek közé ezt az adatot!
  Pl. 1, 2, 4, 5 sorozatba a 3-t → 1, 2, 3, 4, 5
- Adott egy számsorozat, amely egész számokat tartalmaz. Állapítsuk meg, hogy periodikus-e, ha igen, milyen hosszú a legrövidebb periódus!
  Pl. 8, 5, 2, 8, 5, 2, 8, 5, 2, 8, 5, 2 → igen, 3
  1, 1, 1, 1 → igen, 1
  1, 2, 3, 1, 2 → nem
- Adott N db pont a síkon a koordinátáival, mint egy N csúcsú konvex sokszög csúcspontjai valamely körüljárási irány szerint. Határozzuk meg a sokszög kerületét, területét, súlypontját, a leghosszabb oldalát és legrövidebb átlóját! A terület kiszámításához a súlypont segítségével bontsuk háromszögekre a sokszöget!
- Adott N db pont a síkon a koordinátáival, mint egy origó középpontú, céltáblára lőtt lövések. A céltábla egyes körei az egész sugarú körök (1-10-ig), azaz ha egy pont (lövés) távolsága az origótól a [0, 1] intervallumba esik, akkor 10 kört ér (10-es találat), ha a (1, 2] intervallumba esik, akkor 9 kört, …, ha a (9, 10] intervallumba esik, akkor 1 kört, egyébként 0 kört. Határozzuk meg a lövések össztalálatát, azaz, hogy hány kört lőttünk összesen!
- Adott egy legalább két pontot tartalmazó pontrendszer a síkon a koordinátáival. Határozzuk meg:
  - A pontrendszer két, egymástól legtávolabb ill. legközelebb lévő pontjának távolságát!
  - Hány, egymástól maximális ill. minimális távolságra lévő pontpár található a pontrendszerben!
  - A pontrendszer egy olyan pontját, amelynek a többi ponttól vett távolságainak összege minimális!
  - A pontrendszer két olyan pontját (ha van ilyen), amelyek által meghatározott egyenes két, azonos elemszámú részre osztja a pontrendszert (az egyenesre eső pontokat hagyjuk figyelmen kívül)!
  - Keressünk a pontrendszerben olyan legbővebb, „összefüggő részhalmazokat”, amelyekre igaz az, hogy legalább N db pont-

ból állnak, és bármely halmazbeli pontból el tudunk jutni a halmaz bármely pontjába úgy, hogy az úton csak halmazbeli pontokat érintünk, és sosem teszünk meg két szomszédos pont között K távolságnál nagyobbat.

- Polinomokat a fokszámmal (pozitív egész szám, max. 20) és csökkenő kitevő szerinti sorrendben adott együtthatókkal adunk meg. Oldjuk meg az alábbi feladatokat:
  - Határozzuk meg egy polinom deriváltját!
  - Számítsuk ki egy polinom határozott integrálját adott intervallumon!
  - Állítsuk elő két polinom összegét és különbségét!
- Adottak a számegyenesen lévő zárt intervallumok. Egy intervallumot a kezdő és végpontjával adunk meg, amelyek egész számok. Két intervallum „átfedő”, ha van közös pontjuk, különben „idegenek”. Oldjuk meg az alábbi feladatokat:
  - Ellenőrizendő, hogy az intervallumrendszer tagjai páronként idegenek-e vagy sem. Ha nem, akkor meghatározandó egy olyan intervallum, amely a legtöbb más intervallummal átfed!
  - Minimalizáljuk az intervallumrendszer intervallumainak számát az átfedő intervallumok összevonásával!
  - Határozzuk meg az intervallumrendszer által „lefedett” egész számok darabszámát! Egy szám lefedett, ha valamelyik intervallumba beleesik.
  - Ellenőrizendő, hogy az intervallumrendszer metszete üres-e vagy sem, ha nem, adjuk meg a metszetet intervallumként!
- Határozzuk meg egy négyzetes (sorok száma megegyezik az oszlopok számával) mátrixról, hogy szimmetrikus-e, azaz megegyezik-e minden $(A[i, j])$ elem a főátlóra vetített tükörképével $(A[j, i])$ vagy sem!
- Egy négyzetes mátrix elemei valós számok. Tegyük szimmetrikussá úgy, hogy a tükörkép elemek átlagát írjuk mindkét elem helyére!
- Döntsük el egy négyzetes, valós számokat tartalmazó mátrixról azt, hogy lehet-e „távolságmátrix” vagy sem! Ennek feltételei: a mátrix szimmetrikus, elemei nemnegatívak és a főátló elemei nullák.
- Egy négyzetes mátrix „bűvös négyzet”, ha a sorok, oszlopok és a két átló összege ugyanaz a szám. Ellenőrizzük, hogy egy adott mátrix bűvös négyzet-e!

- Keressünk egy mátrixban „nyeregpontokat”! Egy mátrixelem nyeregpont, ha sorában a minimális, oszlopában a maximális értékű. Végezzük el a keresést úgy, hogy a minimum és maximum vizsgálatnál megengedjük az egyenlőséget, és úgy is, hogy nem. (Ha megengedett az egyenlőség, akkor például egy csupa egyforma elemet tartalmazó mátrix minden eleme nyeregpont lesz, különben nem lesz benne nyeregpont.)
- Adott egy valós számokból álló mátrix. Oldjuk meg az alábbi feladatokat:
  - Töröljük a nulla összegű oszlopokat!
  - Töröljük a csak nullát tartalmazó sorokat!
  - Szúrjuk be első sorként az oszlopösszegeket tartalmazó sort!
  - Emeljük ki minden sorban a minimális értéket a sor elejére úgy, hogy ezekből egy új oszlop keletkezzen!
  - Minden sort osszunk le a sor maximális értékű elemével, ha ez lehetséges!
  - Cseréljük fel két, adott indexű oszlopát!
  - Töröljünk minden olyan sort, amely valamely másik sortól csak egy szorzófaktorral különbözik, vagyis a sor egy másikból egy konstanssal való szorzással előállítható!
- Szorozzunk össze két, valós számokat tartalmazó mátrixot! Egy A $N\times M$-es és egy B $M\times K$-s mátrix $A*B$ szorzatmátrixán egy olyan C $N\times K$-s mátrixot értünk, amelynek elemeire teljesül, hogy:
  $C[I, J]=A[I, 1]*B[1, J]+ A[I, 2]*B[2, J]+\ldots+ A[I, M]*B[M, J]$,
  ahol $I=1, 2, \ldots, N$ és $J=1, 2, \ldots, K$.
- Egy mátrixban az ötöslottó nyerőszámai vannak adott darabszámú hétre vonatkozóan. Egy sor egy heti húzásnak felel meg, tehát a mátrixnak 5 oszlopa van és annyi sora, ahány hét adataival dolgozunk. Az elemek értéke az [1, 90] intervallumba eső egész számok. Egy fogadó hetente adott számú és azonos tippekkel játszik, amelyeket az előzőhöz hasonló szerkezetű mátrixban adunk meg (5 oszlop, egy sor egy tipp). Határozzuk meg hetente és találatonként (1-5), hogy a fogadónak hány találata van!
- Generáljuk az 1, 2, …, N számok egy véletlenszerű sorrendjét! (Tipp: tegyük ezeket a számokat rendre egy egydimenziós tömbbe (első helyre az 1-t, másodikra a 2-t, …, N. helyre az N-t), majd generáljunk egy véletlen egész számot 1-től N-ig! Az ennyiedik számot cseréljük fel a tömb utolsó (N.) elemével, majd ismételjük meg ezt véletlenszám generálást és elemcserét a tömb egyre rövidülő (N–1, N–2, …., 2 elemű)

elején! Vegyük észre, hogy ez az algoritmus, ha a véletlenszám generálást és cserét pontosan *M*-szer végezzük el (ahol *M*<=*N*), alkalmas arra is, hogy az 1, 2, …, *N* számok közül *M* darabot véletlenszerűen kiválasszunk. A kiválasztott számok a tömb végére (*N*–*M*+1., …, *N*. helyre) kerülnek.)

- Egy kártyajátékot 52 lapos kártyával játszanak. Készítsünk véletlen leosztást *N* játékosnak úgy, hogy minden játékos *M* db lapot kap! A lapokat a sorszámukkal azonosítsuk, egy lapot csak egyszer osszunk ki, az eredmény egy *N*×*M*-es mátrix legyen!
- Jelenítsünk meg egy mátrixot mátrix alakban úgy, hogy egy képernyőre legfeljebb adott számú sort és oszlopot írunk ki, valamint (bizonyos billentyűkre) megengedjük a következő pozícionálásokat: elejére (a mátrix bal felső része), végére (a mátrix jobb alsó része), egy sorral előre, hátra, egy oszloppal jobbra, balra.

9.2. Sztringek

Szöveges adatok használatát megengedő adattípus.
Konstans: pl. ”Alma”
A sztringek elejének és végének jelzésére a dupla aposztrófot használjuk. Emlékezzünk, hogy a karaktereknél a szimpla aposztrófot használjuk, tehát az ”A” egy, egy karakterből álló sztringet jelent, míg az ’A’ egy karaktert.
Műveletek:

- Összefűzés (+)
- Hasonlítások: mint az egészeknél, csak itt nem a számok értéke, hanem a sztringek karakterei alapján történik a hasonlítás.

Egy sztring *hosszán* a sztring karaktereinek darabszámát értjük.

Az *üres sztring* az a sztring, amelynek egyetlen karaktere sincs, így a hossza nulla. Jelölés: ””

Összefűzés után az eredménysztring hossza, az összefűzendő sztringek hosszainak összege lesz.

Az üres sztring hozzáfűzése egy adott sztringhez, az adott sztringet adja eredményül.

Két sztring egyenlő, ha egyforma hosszúak (azaz ugyanannyi karakterből állnak) és karaktereik rendre megegyeznek.

Azokban a hasonlításokban, amelyekben a <, > relációjel szerepel, az első különböző karakter ASCII kódja alapján dől el a hasonlítás eredmé-

nye. Ha nincs ilyen karakter, azaz az egyik sztring éppen „kezdőszelete" a másiknak, akkor a rövidebb sztring a „kisebb".

Pl. "alma" + "fa" → "almafa"
"Alma" < "alma" → igaz (az 'A' kódja 65, az 'a' kódja 97)
"Kovács" < "Kovácsné" → igaz (a "Kovács" rövidebb)
"Kovacs" < "Kovács" → igaz (az 'a' kódja 97, az 'á' kódja 160)

Megjegyzés: Alapértelmezésben egy sztringet max. 255 karakter hosszúnak engedünk meg, de ha egy feladat megoldásához ennél rövidebb sztringek is elegendők (pl. egy név „elfér" 30 karakterben is, ami helyigényben kb. nyolcad akkora), akkor ezt szögletes zárójelben megadhatjuk deklaráláskor (pl. Sztring[30]).

Függvények:

LENGTH(X)	Az X sztring hossza. Pl. LENGTH("Maci") → 4
COPY(X, Y, Z)	Az X sztring Y-adik jelétől kezdődő Z db karaktert tartalmazó részsztring. Pl. COPY("Mazsola", 3, 2) → "zs" Ha nincs Y-adik karakter, akkor az eredmény az üres sztring. Pl. COPY("Hahó", 5, 1) → "" Ha az Y-adik karaktertől kezdve nincs Z db karakter, akkor az eredmény X végéig tart. Pl. COPY("valami", 5, 4) → "mi"
POS(X, Y)	Az X részsztring első előfordulásának kezdőpozíciója az Y sztringben ha X szerepel Y-ban, egyébként 0. Pl. POS("a","alma") → 1, POS("A","alma") → 0
VAL(X)	Egy számot jelentő X sztring numerikus értéke valós számként. Pl. VAL("3.14") → 3.14, VAL("–1") → –1
STR(X)	Az X szám sztringként. Pl. STR(3.14) → "3.14", STR(–1) → "–1"

9.2.1. Sztringek, mint egydimenziós karaktertömbök

A sztringek, mivel karaktereik egymás után tárolódnak, tekinthetők és használhatók olyan egydimenziós tömbként is, amelynek elemei karakterek. Az első elem a sztring első karaktere, a második elem a második karakter és

így tovább. Ha ST egy sztring típusú változó, akkor megengedett az ST[1] hivatkozás, ami a sztring, mint karaktertömb első elemét jelenti, egy karakteres változót, amely a sztring első karakterét tárolja. Ilyen módon nemcsak elérhetők a sztring egyes karakterei, de azok módosíthatók is.

Megjegyzés

- Egy sztring karaktertömbként történő módosítása nem változtatja meg a sztring hosszát. Ha a hosszt is be kell állítanunk, akkor azt nyelvtől függő módon (C-ben végjellel (lásd bevezető példa a 2. fejezetben), Pascal-ban a hosszbájttal) kell megtennünk.
- A C-ben nincs sztring adattípus, ezért csak így, karaktertömbként kezelhetők a sztringek és így (mint minden tömb) ez is 0-tól indexelődik, nem 1-től.

9.2.2. Sztring megfordítás

Feladat: Fordítsunk meg egy adott sztringet!
Pl. ”Réti pipitér” → ”rétipip itéR”

Megoldás: A feladatot nem a nehézsége miatt választottuk mintapéldának, hanem azért, hogy megmutassuk, hogy egy ilyen egyszerű feladatnak is létezhet többféle megoldása. Az első két megoldásban egy eredménysztringet állítunk elő a sztring karaktereinek összefűzésével, míg a harmadikban, helyben fordítjuk meg a sztring karaktereit.

Az első megoldásban, a kezdetben üres eredménysztringhez jobbról hozzáfűzzük a sztring karaktereit fordított sorrendben, azaz először a legutolsót, majd az utolsó előttit, legvégül az elsőt.

A második megoldásban, a kezdetben üres eredménysztringhez balról hozzáfűzzük a sztring karaktereit eredeti sorrendben, aminek eredményeként a sztring első karaktere kerül az eredménysztring legvégére, a második karaktere lesz az utolsó előtti, végül az utolsó karaktere kerül az eredménysztring legelejére.

A harmadik megoldásban a sztring karaktereit cserélgetjük fel oly módon, hogy az első karaktert az utolsóval cseréljük fel, a másodikat az utolsó előttivel és így tovább. Ezt a cserélgetést természetesen csak a sztring „feléig” kell megtennünk, hiszen ha a sztring „hátsó” karaktereire is elvégeznénk a megfelelő párral való cserét, akkor éppen az eredeti sorrendet állítanánk vissza. A megoldásban felhasználjuk, hogy a felcserélendő

karakterpárok indexeinek összege éppen a sztring hosszánál eggyel nagyobb (pl. az *i*-ik karakter párja a *h*+1–*i*-edik, ahol *h* a sztring hossza)

Megjegyzés: A C nyelv 0-tól indexel, így ott a két index összege *h*–1-et ad.

Funkció	Azonosító	Típus	Jelleg
A megfordítandó sztring	S	Sztring	I
Az eredménysztring	ER	Sztring	M, O
Ciklusváltozó	I	Egész	M

```
/* Sztring megfordítás 1 */
FORDIT1(S,ER)

ER ← ""
for I ← LENGTH(S),1,-1
       ER ← ER+S[I]

/* Sztring megfordítás 2 */
FORDIT2(S,ER)

ER ← ""
for I ← 1,LENGTH(S)
       ER ← S[I]+ER
```

Funkció	Azonosító	Típus	Jelleg
A megfordítandó sztring	S	Sztring	I, O
A sztring hossza	H	Egész	M
Két karakter cseréjéhez	CS	Karakter	M
Ciklusváltozó	I	Egész	M

```
/* Sztring megfordítás 3 */
FORDIT3(S)

H ← LENGTH(S)
for I ← 1,H DIV 2
       CS ← S[I]
       S[I] ← S[H+1-I]
       S[H+1-I] ← CS
```

9.2.3. Egyszerű kifejezés kiértékelése

Feladat: Adott egy sztring, amelyben egy egyszerű kifejezés található. Helyes a kifejezés, ha páratlan számú karakterből áll, a páratlan karakterpozíciókban számjegyek, a párosokban a '+' vagy '–' műveleti jelek szere-

pelnek. Döntsük el a kifejezésről, hogy helyes-e vagy sem és ha helyes, akkor számítsuk ki a kifejezés értékét!

Megoldás: Először a sztring helyességét vizsgáljuk meg, amelyet a hossz ellenőrzésével kezdünk. Ha ez megfelelő (páratlan), akkor sorban megvizsgáljuk a sztring egyes karaktereit. Az első nem megfelelő karakternél befejezzük az ellenőrzést, hiszen a kifejezés helytelen. Ha minden karakter helyes, akkor egy ciklussal kiszámítjuk a kifejezés értékét, azaz a számjegyek numerikus értékeit, az előttük álló előjeltől függően vagy hozzáadjuk, a kifejezés értékét gyűjtő változó értékéhez, vagy kivonjuk belőle. Az első számjegynek nincs előjele, tehát pozitív, így a gyűjtőváltozó értékét kezdetben az első számjegy numerikus értékére állítjuk. A számjegykarakterek számmá alakítását az ASC függvénnyel végezzük. Itt kihasználjuk azt, hogy a számjegykarakterek ('0'-tól '9'-ig) egymást követik az ASCII kódtáblában, így kódjaik is egyesével növekednek, azaz egy C számjegykarakter numerikus értékét éppen az ASC(C)–ASC('0') különbség adja.

Funkció	Azonosító	Típus	Jelleg
A feldolgozandó sztring	S	Sztring	I
A sztring helyessége	JO	Logikai	M, O
A kifejezés értéke	ER	Egész	M, O
A sztring hossza	H	Egész	M
Segédváltozó	I	Egész	M

```
/* Egyszerű kifejezés kiértékelése */
ERTEKEL(S,JO,ER)

/* Hosszellenőrzés */
H ← LENGTH(S)
JO ← H MOD 2=1
if JO
       /* Karakterek ellenőrzése */
       I ← 1
       while (I<=H) AND JO
              if (I MOD 2=1) AND ((S[I]<'0') OR (S[I]>'9')) OR
                 (I MOD 2=0) AND (S[I]<>'+') AND (S[I]<>'-')
                    JO ← hamis
              else
                    I ← I+1
       if JO
              /* A kifejezés értékének kiszámítása */
              ER ← ASC(S[1])-ASC('0')
              for I ← 3,H,2
```

```
if S[I-1]='+'
       ER ← ER+ASC(S[I])-ASC('0')
else
       ER ← ER-(ASC(S[I])-ASC('0'))
```

Megjegyzés: A karakterek vizsgálata halmazok és az IN művelet segítségével is elvégezhető (lásd 9.3.).

9.2.4. Feladatok

- Állítsunk elő adott karakterből álló, adott hosszúságú sztringet!
  Pl. '*', 3 → "***"
- Töltsünk fel egy adott sztringet, jobbról szóközökkel úgy, hogy az eredménysztring adott hosszúságú legyen!
  Pl. ""-t 3 hosszra → "_ _ _", "123"-t 2 hosszra → "123"
- Töröljük ki egy adott sztringből az egymás mellett lévő azonos karaktereket úgy, hogy csak egy maradjon meg belőlük!
  Pl. "abbbbbccD" → "abcD"
- Egy sztring összes angol kisbetűjét alakítsuk nagybetűsre!
  Pl. "aBCd1?" → "ABCD1?"
- Madárnyelv: Készítsünk egy adott sztringből olyan sztringet, amelyben a magánhangzókat egy 'v' karakterrel együtt megismételjük!
  Pl. "róka-koma" → "róvókava-kovomava"
- Hagyjuk ki egy adott sztringből a zárójeles részeket. Feltételezhetjük, hogy egy zárójelpáron belül nincs újabb zárójel. A zárójelpár a { } jelpár.
  Pl. "begin {előkészít} i:=0; {***} j:=1;" → "begin i:=0; j:=1;"
- Adott két sztring. Hagyjuk el mind a kettőből a leghosszabb közös kezdő részt!
  Pl. "fapapucs", "faló" → "papucs", "ló"
  Pl. "emberek", "gyerekek" → "emberek", "gyerekek"
- Adott két sztring. Hagyjuk el mind a kettőből a leghosszabb közös befejező részt!
  Pl. "fapapucs", "faló" → "fapapucs", "faló"
  Pl. "emberek", "gyerekek" → "ember", "gyerek"
- Egy sztringben az MS-DOS szabályai szerint teljes elérési úttal megadott fájl meghatározás van. Csak a fájlnevet hagyjuk meg!
  Pl. "C:\WORK\SZOVEG.TXT" → "SZOVEG.TXT"

- Soroljunk be egy adott sztringet az alábbi 4 osztály valamelyikébe! Egy sztring
  - növekvő, ha jelei (szigorúan) növekvő sorrendben vannak;
  - csökkenő, ha jelei (szigorúan) csökkenő sorrendben vannak;
  - konstans, ha jelei azonosak;
  - egyéb, minden más esetben.
- Ellenőrizzük egy sztringben adott formula zárójelezésének helyességét! Feltételezhetjük, hogy csak egyfajta (a kerek zárójelpár) zárójelezés fordul elő. A zárójelektől különböző jeleket nem kell ellenőrizni. A formula zárójelezése helyes, ha:
  - a kezdő- és végzárójelek száma egyenlő;
  - balról jobbra haladva a kezdőzárójelek száma sohasem kisebb a végzárójelek számánál.

  Pl. Helyes: ”i+1”, ”12*(3+(1))”
  Helytelen: ”3*(5+4”, ”2–)3+a(”,
- Oldjuk meg az előző feladatot kétféle zárójelpár esetére! A zárójelpárok a ( ) és a { } jelpárok. A kétféle zárójelezésnek önmagában, de egymás viszonylatában is helyesnek kell lennie: nem lehet átfedés.

  Pl. Helyes: ”2*{(a+b)/2–(3+1)}”
  ”2*({a+b}/2–{3+1})”
  Helytelen: ”2*{(a+b}/2–{3+1)}”
- Oldjuk meg az előző feladatot úgy, hogy a { } jelpár a „külső” zárójel, vagyis a ( ) zárójelen belül { } nem szerepelhet!
- Egy sztring egy 16-os számrendszerben írt számot tartalmaz. Állítsuk elő egy sztringben a szám 2-es számrendszerbeli alakját! Készítsük el a 2-es 16-os konverziót is!

  Pl. ”B18” → ”101100011000”
  ”101100011000” → ”B18”
- Egy sztringben egy olyan név szerepel, amely két részből áll (családnév, keresztnév), a két részt pontosan egy szóköz választja el. Cseréljük fel a név két részét!

  Pl. ”Makk Marci” → ”Marci Makk”
- Alakítsunk át egy sztringet úgy, hogy eltávolítjuk a kezdő és záró szóközöket, és a belsejében az egymást követő szóközök közül csak egyet hagyunk meg!

  Pl. ” Maci Laci ” → ”Maci Laci”

- Adott egy sztring amelyben csak angol betűk és szóközök vannak. Tegyük fel, hogy kezdő és záró szóközök nincsenek, és a sztring belsejében sincsenek egymást követő szóközök. „Szó” kezdődik a szöveg elején és minden szóköz után. Oldjuk meg a következő feladatokat:
  - Számoljuk ki az átlagos szóhosszt!
  - Minden szókezdő kisbetűt cseréljünk ki nagyra!
  - Minden, a szó belsejében lévő nagybetűt cseréljünk ki kicsire!
- Feltételezve, hogy a sztring az előző feladatnak megfelelő formátumú, fordítsuk meg a sztringben lévő szavak sorrendjét!
  Pl. ”Géza kék az ég” → ”ég az kék Géza”
- Kódoljuk át egy adott sztring minden karakterét egy kódtáblázat segítségével, majd készítsük el a kód visszafejtését is!
  Pl. Sztring: ”cica-mica”
  Kódtábla: (a-b, b-c, c-d)
  Eredmény: ”didb-midb”
- Kódoljunk át egy szöveget egy kulcsszóval (mindkettő sztring)! A kulcsszó jeleit (ha szükséges periodikusan ismételve) vonjuk le a szöveg jeleiből (az ASCII kódokat kivonva, ha különbség negatív lenne, akkor adjunk hozzá 256-ot)! Készítsük el a kód visszafejtését is!
  Pl. Kulcs: ”ABA”
  Sztring: ”szeptember”
  Eredmény: ”28$/2$,_$1”
- Kódoljunk egy szöveget egy kulcsszámmal úgy, hogy a kulcsszám hosszú részeit részenként megfordítjuk! A részeket a sztring elejétől kezdve képezzük, ha a végén egy nem teljes hosszú maradék lenne, az maradjon változatlan! Készítsük el a kód visszafejtését is!
  Pl. Kulcs: 3
  Sztring: ”Hé mama levele jött!”
  Eredmény: ” éHmaml aeve eltöjt!”
- Tömörítsünk egy adott sztringet úgy, hogy a benne egymást követő legalább 4 db azonos *X* karaktert a #*X*CHR(*DB*) karakterhármassal helyettesítjük, ahol *DB* az ismétlődő *X* karakterek számát, így CHR(*DB*) a *DB* ASCII kódú karaktert jelöli. Feltesszük, hogy az eredeti sztring nem tartalmaz ’#’ karaktert. Készítsük el a tömörített sztring kicsomagolását is!
  Pl. ”aaaaabcdddeeee” → ”#aCHR(5)bcddd#eCHR(4)”, ahol

CHR(5) ill. CHR(4) az 5-ös és 4-es ASCII kódú karaktert jelöli.

- Adott két sztring, *X* és *Y*, ahol *X* nem lehet hosszabb, mint *Y*. Keressük meg, hogy az *Y* mely részével egyezik legtöbb helyen az *X*! Az eredményt a legtöbb helyen egyező rész kezdőpozíciója és az egyező jelek száma jelenti. A vizsgálatot balról jobbra haladva végezzük!

  Pl. ”golyó”, ”szilvásgombóc”, → 8, 3

- Egy *X* és egy *Y* sztring „hasonló”, ha az alábbi esetek valamelyike fennáll:
  - egyforma hosszúak és karaktereik egy pozíció kivételével rendre megegyeznek;
  - az egyik egy jellel hosszabb, mint a másik, de a hosszabbnak van egy olyan jele, amelyet elhagyva éppen a rövidebbet kapjuk.

  Döntsük el két sztringről, hogy hasonlóak-e vagy sem!

- Egy sztringben egy olyan egyszerűsített kifejezés van, amelyben csak számjegyek, a ’+’, ’–’ és ’*’ műveleti jelek szerepelnek. A sztring számjeggyel kezdődik és végződik, valamint a számjegyek és műveleti jelek felváltva követik egymást. Számoljuk ki a kifejezés értékét a szorzás prioritásának figyelembevételével!

  Pl. ”1+2*3” → 7

- Oldjuk meg az előző feladatot úgy, hogy a számok nemcsak számjegyek lehetnek, hanem többjegyű számok is!

- Pl. ”10+2*300” → 610

- Előjel nélküli, tízes számrendszerű egész számokat (max. 255 jegyűek) sztringekben tárolunk. Egy jegy a sztring egy jele, a jegyek sorrendje a csökkenő helyérték szerinti sorrend. Oldjuk meg az alábbi feladatokat:
  - Számítsuk ki két szám összegét, ha ez lehetséges (ha az eredmény is tárolható így)!
  - Számítsuk ki két szám különbségét, ha ez lehetséges (ha a kivonandó nem nagyobb, mint a kisebbítendő)!
  - Számítsuk ki két szám szorzatát (ismételt összeadással), ha ez lehetséges (ha az eredmény is tárolható így)!
  - Számítsuk ki két számra az egész osztás hányadosát és maradékát (ismételt kivonással)!
  - Határozzuk meg két szám viszonyát (kisebb, nagyobb, egyenlő)!

  A műveletek elvégzésének megkönnyítésére célszerű a kisebb számot a nagyobb szám hosszában, balról nullákkal feltölteni.

- Adottak egy évre vonatkozóan sztring típusú, *HH.NN.OO* (hónap, nap, óra) formátumú időpontok, és hozzájuk tartozó egész értékű hőmérséklet adatok. Határozzuk meg naponta a napi, és havonta a havi átlaghőmérséklet adatokat! Jelezzük vissza azt is, ha valamely napra ill. hónapra nincs adat!
- Egy kártyajátékot francia kártyával (52 lap) játszanak. Készítsünk véletlen leosztást *N* játékosnak úgy, hogy minden játékos *M* db lapot kap! A lapokat a sorszámukkal azonosítsuk, egy lapot csak egyszer osszunk ki, az eredmény egy *N*×*M*-es mátrix legyen! Az osztás kiírásakor sorszámok helyett színt és figurát (pl. Pikk Ász) írjunk ki!
- Adott egy négyzetes karaktermátrix. Állítsuk elő azt a sztringet, amely a karakterek alábbi példa szerinti összeolvasásával adódik:

Pl. Kiindulás Eredmény

E	Z	L	A	S	
A	E	D	I	O	
F	A	M	L	E	EZAFELADATNEMISOLYANNEHÉZ
T	E	Y	N	H	
N	A	N	É	Z	

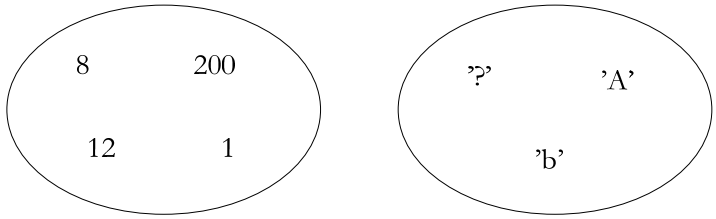
9.3. Halmazok

A halmazok fogalma jól ismert a matematikából, de számítógépes megvalósításuk, implementálásuk csak néhány nyelvben, ott is csak bizonyos korlátozásokkal történt meg.
Kétféle típusú halmazt használhatunk:

- *Karakterhalmaz*: olyan halmaz, amelynek elemei tetszőleges karakterek lehetnek.
- *Egészhalmaz*: olyan halmaz, amelynek elemei a [0, 255] intervallumba eső, egész számok lehetnek.

Konstans: Az elemek ill. elemintervallumok vesszővel elválasztott sorozata, szögletes zárójelben.

Pl.	[8, 1, 200, 12]	négy számot tartalmazó egészhalmaz,
	['b', 'A', '?']	három karaktert tartalmazó karakterhalmaz,
	['A'..'Z', 'a'..'z']	az angol betűket tartalmazó karakterhalmaz,
	[ ]	az üres halmaz, amelynek egyetlen eleme sincs.



9.2. ábra. Halmazok és elemeik.

Műveletek:

- Multiplikatív: Közös rész (más néven metszet) (*)
- Additív: Egyesítés (más néven unió) (+), Különbség (−)
- Hasonlítások: Egyenlő (=), Nem egyenlő (<>), Részhalmaz vizsgálat (<=, >=)
- Tartalmazás (IN)

Az első három művelet (+, *, –) azonos típusú halmazokon értelmezett, kétoperandusú művelet, a megfelelő halmazműveletek elvégzésére.

A hasonlítások két, azonos típusú halmaz között értelmezett, logikai eredményt adó műveletek, jelentésük értelemszerű.

A tartalmazás művelet egy elem és egy ugyanolyan típusú elemekből álló halmazra értelmezett, logikai értéket adó művelet, megmondja, hogy az adott elemet tartalmazza-e az adott halmaz vagy sem.

Pl. ['a', 'b'] * ['b', 'c'] → ['b']
[2, 3] + [3, 4] → [2, 3, 4]
[8] – [2, 8] → [ ]
[2, 3] >= [2] → igaz
'a' IN ['A', 'b'] → hamis
NOT ('a' IN ['A', 'b']) → igaz

Megjegyzés

- Az utolsó példa a „nem eleme" vizsgálatot végzi a NOT művelettel, amely erősebb prioritású (lásd 4.2.), mint a tartalmazás.
- Ezek a max. 256 elemű egész-, ill. karakterhalmazok egy-egy 256 elemű tömbbel is „leírhatók" (az *i*. elem 1, ha az *i* mint szám vagy az *i* kódú karakter bent van a halmazban, 0 különben), a halmazműveletek pedig programozhatók (pl. a metszetet ekkor az azonos indexű elemek szorzata adja).

- Láncolt listák (lásd 13.4.) segítségével, igaz több munka árán, a tömbös megvalósítás korlátai (elemtípus, elemszám) is feloldhatók, közelítve ezzel az általános matematikai halmazfogalomhoz.

A halmazok tehát jól használhatók akkor, ha egy feladat megoldásánál olyan jellegű kérdések vetődnek fel, hogy

- egy érték benne van-e egy adatcsoportban vagy sem,
- két adatcsoport viszonya milyen (egyik tartalmazza-e a másikat, van-e közös elemük, stb.),

de nincs szerepe az adatok sorrendjének, elemszámának (pl. egy lottószelvény esetén elsősorban az számít, hogy hány darab nyerőszám van rajta, míg az, hogy melyiket hányadiknak húzták ki, lényegtelen).

9.3.1. Lottószámok generálása

Feladat: Generáljunk véletlenszerűen lottószámokat!

Megoldás: A lottószámok maximális értékére és darabszámára egy-egy konstanst deklarálva oldjuk meg a feladatot, így a konstansok megváltoztatásával pl. a hatoslottó játékosai is generálhatnak maguknak számokat. A számok különbözőségét egy halmaz segítségével biztosítjuk, amelybe a generált számokat tesszük. Ha olyan számot generálunk, amely már szerepel a halmazban (azaz amelyet korábban már generáltunk), akkor helyette újat generálunk.

Megjegyzés

- Ha a lottószámok maximális értéke és darabszáma nem megfelelő (több számot szeretnénk generálni annál, mint ahány darab számunk van), akkor a generálás végtelen ciklusba esik.
- Ha közel annyi számot kell generálnunk, mint ahány számunk van, akkor a végefelé egyre nehezebben talál megfelelő (még nem generált) számot az algoritmus. Ennek kiküszöbölésére készíthető olyan (tömbös) megoldás is, amely legfeljebb annyi véletlenszámot generál, mint ahány számot generálnunk kell (lásd 9.1.7.).

Konstans

```
MAX  90        /* A maximális lottószám értéke */
DB   5         /* A generálandó lottószámok darabszáma */
```

Funkció	Azonosító	Típus	Jelleg
A generált lottószámok tömbje	A	Egydimenziós egész tömb[DB]	O
A generált lottószámok halmaza	H	Egészhalmaz	M
Az aktuálisan generált lottószám	X	Egész	M
Segédváltozók	I, J	Egész	M

```
/* Lottószámok generálása */
GENERAL(A)
```

```
/* Generálás */
H ← []
for I ← 1,DB
        repeat
                X ← RANDOM(MAX)+1
        until NOT (X IN H)
        H ← H+[X]
/* Az eredménytömb feltöltése */
J ← 0
for I ← 1,MAX
        if I IN H
                J ← J+1
                A[J] ← I
```

9.3.2. Eratosztenesz szitája

Feladat: Készítsünk megoldást az Eratosztenesz szitájának megvalósítására halmaz segítségével!

Megoldás: Az algoritmus részletes leírása és logikai tömbbel való megvalósítása a tömböket tárgyaló fejezetben (lásd 9.1.) található. Most egy olyan megoldást mutatunk be, amelyben a felírt számokat egy halmazban tároljuk. Ebből töröljük ki a prímszámok többszöröseit, így az algoritmus végén csak a prímszámok maradnak benne.

Megjegyzés: A *K* korlát értéke most maximum 255 lehet. A C nyelvű megoldás, ahol egy tömbbel valósítjuk meg a halmazt, éppen a tömbös megoldást adja, ezért csak a valódi halmazt használó Pascal nyelvű megvalósítás található meg a függelékben.

```
Konstans
        K  255          /* Eddig határozzuk meg a prímszámokat */
```

Funkció	Azonosító	Típus	Jelleg
A felírt számokat reprezentáló halmaz	H	Egészhalmaz	M, O
Az aktuálisan vizsgált szám	P	Egész	M
A többszörösök törléséhez	I	Egész	M

```
/* Eratosztenesz szitája halmazzal */
/* Számok felírása */
H ← [2..K]
/* Szitálás */
for P ← 2,K
      /* A P szám prímszám? */
      if P IN H
            /* Igen, töröljük a többszöröseit */
            for I ← 2*P,K,P
                  H ← H-[I]
/* Kiírás */
for I ← 2,K
      if I IN H
            Ki: I
```

9.3.3. Különböző karakterek száma

Feladat: Határozzuk meg egy sztring különböző karaktereinek számát!

Megoldás: A sztring karaktereinek különbözőségét egy halmaz segítségével vizsgáljuk. A kezdetben üres halmazt rendre bővítjük a sztring olyan karaktereivel, amelyek még nincsenek a halmazban. Ha egy karaktert hozzáveszünk a halmazhoz, akkor (a kezdetben nullára állított) különböző karakterek számát megnöveljük eggyel.

Funkció	Azonosító	Típus	Jelleg
A vizsgálandó sztring	S	Sztring	I
A különböző karakterek száma	DB	Egész	M, O
A különböző karakterek halmaza	H	Karakterhalmaz	M
Segédváltozó	I	Egész	M

```
/* Egy sztring különböző karaktereinek száma */
KULKARDB(S)
DB ← 0
H ← []
for I ← 1,LENGTH(S)
      if NOT (S[I] IN H)
            H ← H+[S[I]]
            DB ← DB+1
return DB
```

9.3.4. Karakterbekérés billentyűzetről

A *DOS* operációs rendszerben egyes billentyűk leütésekor egy bájt kerül a billentyűzet pufferébe, ezeket normálkódú karaktereknek nevezzük (ilyenek például a betűk, számok, az *Enter*, *Backspace*, *Esc* billentyűk, stb.), míg vannak olyan billentyűk, amelyek leütésekor két bájt. Az első bájt ilyenkor a 0-s kódú karakter (az a bájt, amelynek minden bitje 0), a második az adott billentyűt azonosító kód. Ezeket duplakódú karaktereknek vagy funkcióbillentyűknek nevezzük (pl. *F1*,…, *F12*, a kurzormozgató billentyűk, *Insert*, *Delete*, stb.). Szeretnénk egy olyan bekérő eljárást készíteni, amely csak az általunk előre megadott karaktereket fogadja el, a többit figyelmen kívül hagyja.

Feladat: Kérjünk be a billentyűzetről egy előre definiált karaktert az alábbiak figyelembevételével:

- Mind normál-, mind duplakódú karakterek bekérhetők legyenek.
- Csak a megadott karaktereket fogadjuk el.
- A duplakódú karakterek esetén a második jelet adjuk eredményül.
- Adjuk meg eredményül azt is, hogy normál-, vagy duplakódú karaktert fogadtunk-e el.

Megoldás: Bekérjük az éppen leütött billentyűnek megfelelő jelet. Ha ez a nullás kódú jel, akkor duplakódú billentyűt ütöttek le, ezért beolvassuk a második jelet is. Ha a megadott jel beleesik a megfelelő halmazba (normálkódú jel esetén a megengedett normálkódú karakterek halmazába, ha duplakódú, akkor a megengedett duplakódú karakterek halmazába), akkor megvan az eredmény is, különben folytatjuk a billentyűzet figyelését.

Megjegyzés: A beolvasást olyan utasítással kell programoznunk, amelyik nem jeleníti meg a leütött karaktert a képernyőn.

Funkció	Azonosító	Típus	Jelleg
Az elfogadható normálkódú jelek	NORMAL	Karakterhalmaz	I
Az elfogadható duplakódú billentyűk második jelei	DUPLA	Karakterhalmaz	I
Duplakódú-e a leütött billentyű	DUPLAE	Logikai	O
Az aktuálisan bekért jel	JEL	Karakter	M, O
Jó-e az aktuálisan bekért jel	JOJEL	Logikai	M

```
/* Billentyűzetről való bekérés */
BILLBE(NORMAL,DUPLA,DUPLAE)
repeat
       Be: JEL
       DUPLAE ← JEL=#0
       if DUPLAE
              Be: JEL
              JOJEL ← JEL IN DUPLA
       else
              JOJEL ← JEL IN NORMAL
until JOJEL
return JEL
```

9.3.5. Adatsor megjelenítése

Feladat: Jelenítsünk meg egy *N* elemű adatsort a képernyőn az alábbiak figyelembevételével:

- Egy képernyőre maximum, adott számú (*DB*) elemet írjunk ki.
- Minden elem külön sorba kerüljön a sorszámával együtt.
- Az elemek kiírását az első elemnél kezdjük.
- Tegyük lehetővé az előre- és visszalépdelést, a legelejére és legvégére való ugrást, valamint a kilépést is.

Megoldás: Az adatokat tömbben tároljuk és bevezetünk két változót (KEZD, BEF), amelyekkel az adatsor kiíratásra kerülő részét adminisztráljuk. KEZD tárolja majd az aktuálisan kiírandó első, BEF pedig az utolsó elem indexét. Tehát a BEF–KEZD+1 értéknek éppen az egy képernyőre írandó elemek számával kell megegyeznie. Ennél kevesebb csak abban az esetben lehet, ha az elemek száma kisebb, mint az egy képernyőre írandó elemek száma, hiszen hiába akarunk például 10 db értéket egy képernyőre íratni, ha összesen csak 5 db elemünk van.

Az egyes pozícionálásokat értelemszerűen a hozzájuk tartozó funkcióbillentyűkre fogjuk definiálni (pl. a *PgDn* billentyűvel lehet majd egy oldalnyit hátrafelé lapozni, a *Home* billentyűvel lehet majd a lista legelejére ugrani, és így tovább).

Minden egyes pozícionáláskor csak a KEZD értékét módosítjuk, a BEF értékét ehhez igazítjuk.

Az alábbi pozícionáló billentyűket engedjük meg, de csak akkor, ha az aktuális képernyőtől eltérő tömbrészre pozícionálnak:

Up (↑)	egy elemmel előre (az adatsor eleje felé),
Down (↓)	egy elemmel hátra (az adatsor vége felé),
PgUp	egy lappal (oldallal) előre,
PgDn	egy lappal hátra,
Home	az adatsor elejére,
End	az adatsor végére.

Megoldásunkban ugyanilyen nevű karakterkonstansokat fogunk használni, amelyek a hozzájuk tartozó billentyűket jelölik (pl. az ESC karakterkonstans az *Escape* billentyűnek felel majd meg). A karakterbekérést az előbbi példában szereplő BILLBE függvénnyel végezzük.
Az elemeket külön sorokba írjuk (a sorszámukkal együtt), egy fejléc sorral.

Funkció	Azonosító	Típus	Jelleg
A megjelenítendő adatsor	A	Egydimenziós, tetszőleges tömb	I
Az adatsor elemeinek száma	N	Egész	I
Egy képernyőre írandó adatok száma	DB	Egész	I
Az aktuálisan kiírandó elemek kezdő indexe	KEZD	Egész	M
Az aktuálisan kiírandó elemek befejező indexe	BEF	Egész	M
Az aktuálisan kiírandó elem indexe	I	Egész	M
Az aktuálisan elfogadható normálkódú billentyűk jelei	NORMAL	Karakterhalmaz	M
Az aktuálisan elfogadható duplakódú billentyűk második jelei	DUPLA	Karakterhalmaz	M
Az aktuálisan bekért jel	JEL	Karakter	M
Duplakódú-e a leütött billentyű	DUPLAE	Logikai	M

```
/* Adatsor megjelenítése */
KIIR(A,N,DB)
if N=0
       Ki: ”Nincsenek elemek!”
       Várakozás billentyű leütésre
else
       KEZD ← 1
       repeat
              /* A befejező index beállítása */
```

```
            BEF ← KEZD+DB-1
            if BEF>N
                  BEF ← N
            /* Adatkiírás */
            Képernyőtörlés
            Ki: "Ssz. Elem"
            for I ← KEZD,BEF
                  Ki: I,A[I]
            /* Az elfogadható jelek beállítása */
            NORMAL ← [ESC]
            DUPLA ← []
            if KEZD>1
                  DUPLA ← DUPLA+[HOME,UP,PGUP]
            if BEF<N
                  DUPLA ← DUPLA+[END,DOWN,PGDN]
            /* Billentyűzetről való bekérés */
            JEL ← BILLBE(NORMAL,DUPLA,DUPLAE)
            /* Pozícionálás */
            if JEL=HOME
                  KEZD ← 1
            else if JEL=END
                  KEZD ← N-DB+1
            else if JEL=PGDN
                  if BEF<N-DB+1
                        KEZD ← KEZD+DB
                  else
                        KEZD ← N-DB+1
            else if JEL=PGUP
                  if KEZD-DB>=1
                        KEZD ← KEZD-DB
                  else
                        KEZD ← 1
            else if JEL=DOWN
                  KEZD ← KEZD+1
            else if JEL=UP
                  KEZD ← KEZD-1
      until JEL=ESC
```

Megjegyzés: A tetszőleges elemtípusú jelen esetben azt jelenti, hogy az algoritmus működik az összes olyan adatsorra, amelynek elemei szerepelhetnek a kiíró utasításban (lásd 4.6.). A konkrét elemtípust a kiírandó adatsor elemeinek típusa határozza meg.

9.3.6. Feladatok

- Adott egy halmaz, amely 0 és 99 közötti egész értékeket tartalmaz. Határozzuk meg a halmaz számosságát (elemeinek darabszámát), valamint a minimális és maximális elemét!
- Adott két halmaz, amelyek karakter típusú értékeket tartalmaznak. Gyűjtsük ki egy tömbbe a két halmaz közös elemeit!
- Adott két tömb, amelyek 0 és 99 közötti egész értékeket tartalmaznak. Gyűjtsük ki egy halmazba azokat az értékeket, amelyek mind a két tömbben előfordulnak!
- Egy tömb páros számú adatot tartalmaz és tegyük fel, hogy az elemek különbözőek. Generáljunk egy teljes párosítást, azaz az elemek egy olyan párosítását, amelyben minden elem egy és csak egy párban szerepel! (Pl. Legyen a tömbben egy osztálynévsor. Az osztályteremben kétszemélyes padok vannak. Generáljunk egy ülésrendet!)
- Egy évfolyamon max. 200 hallgató van. Azonosításukra egyszerű sorszámokat használunk. Generáljunk egyenlő létszámú csoportokat! A csoportlétszám adott, min. 10 és max. 20 lehet. Ha a hallgatók száma nem osztható a kívánt csoportlétszámmal, akkor az utolsó csoportban legyen kevesebb hallgató!
- Generáljunk adott hosszú, angol nagybetűkből álló sztringet úgy, hogy a sztring karakterei különbözők legyenek!
- Egy kártyajátékot 52 lapos kártyával játszanak. Készítsünk véletlen leosztást N játékosnak úgy, hogy minden játékos M db lapot kap! A lapokat a sorszámukkal azonosítsuk, egy lapot csak egyszer osszunk ki, az eredmény N db halmaz legyen!
- Egy tervezett tárgyaláson résztvevő személyenként, egy kezdő- és végidőponttal (mindkettő egész óra a napon belül) adott a személyek szabadideje. A tárgyaláson mindenkinek jelen kell lennie. Határozzuk meg a tárgyalás maximális időtartamát, kezdő- és végidőpontját!
- Síkbeli pontokat a derékszögű koordinátáikkal adjuk meg. A koordináták 1 és 99 közé eső egész számok. A pontokból ponthalmazokat képezünk. Készítsük el a ponthalmazok implementációját (az egyes halmazműveletek megfelelőit) logikai mátrixokra alapozva: (i,j) pont eleme a halmaznak, ha a mátrix i. sorának j. oszlopában igaz érték van!
- Készítsünk „kukac-mozgató” játékot a következők szerint! A játéktér a karakteres képernyő. A kiíráshoz használható egy KIIR(O,S,K) szubrutin, amely a képernyő O oszlopába és S sorába kiírja a megadott K ka-

raktert. A kukac egyetlen karakter (pl. 'O'), amely a kurzormozgató (balra, jobbra, le, fel) billentyűkkel mozgatható (lásd BILLBE függvény). Ha a kukac visszafordul (éppen ellentétes irányba, mint amerre addig haladt), vagy eléri a játéktér szélét, akkor vége a játéknak. A kukac a játék kezdetén a játéktér közepén áll, majd egy iránybillentyű leütésére elindul a megadott irányba és mindaddig arra halad, amíg teheti ill. amíg irányt nem váltunk. A játék nehezíthető, hogy ha a kukac táplálékokra vadászik, amelyek a játéktéren véletlenszerűen elhelyezett karakterek (pl. '*'). Ha a kukac első karaktere rálép egy táplálékra, akkor azt egye meg, azaz ragadjon a kukac végéhez ez a megevett táplálék, megnövelve így a kukac hosszát eggyel. A hosszabb kukac mozgatásakor a kukac teste több helyen is megtörhet, aszerint, hogy milyen irányban kívánunk haladni. Ha a kukac saját testébe ütközne (ill. visszafordulna vagy elhagyná a játékteret), legyen vége a játéknak. Ha az összes táplálékot sikerült „megenni", akkor a játékos nyert, egyébként vesztett.

9.4. Rekordok

Az eddig megismert két összetett adattípus, a tömb és a halmaz is azonos típusú adatok egy összességének használatát biztosították, igaz, egymástól merőben eltérő módon. Ha azonban *különböző típusú* adatok egy összességét szeretnénk együtt kezelni, akkor nagy segítséget jelenthetnek a rekordok. A rekord adattípussal ugyanis több, tetszőlegeses számú és típusú adatot foglalhatunk egy logikai egységbe.

Deklarálás

Meg kell adnunk, hogy milyen típusú adatokat szeretnénk együtt használni, egy egységbe foglalni és ezeknek milyen azonosítói (nevei) legyenek, azaz definiálnunk kell a rekord felépítését, mezőit.
Az alábbi formalizmust fogjuk követni:

Típusnév Rekord
    $Mező_1$ $Típus_1$
    $Mező_2$ $Típus_2$
    …
    $Mező_n$ $Típus_n$

A Típusnév a definiált rekordtípus azonosítója (ezt használjuk majd az adatszerkezeti táblázatban, a rekordváltozók típusának megadására), a $Mező_1$, …, $Mező_n$ a rekord mezőinek azonosítói, míg a hozzájuk tartozó adattípusok rendre $Típus_1$, …, $Típus_n$.

Megjegyzés
- Ha több, egymást követő mező azonos típusú, akkor az azonosítójukat vesszővel elválasztva, a típust elég egyszer megadni.
- A rekordok egyes mezői tetszőleges egyszerű vagy összetett adattípusúak lehetnek, így a feladathoz leginkább illeszkedő adatstruktúra definiálható.
- A rekordtípus és a mezők azonosítóinál követjük az azonosítókra eddig használt írásmódot.

Hivatkozás

Rekordváltozó.Mezőazonosító

Ha egy adott típusú adatot tárolni szeretnénk, akkor egy ilyen típusú változót kell deklarálnunk az adatszerkezeti táblázatban. Ez a rekordokra is igaz. A rekordváltozó tárolja és hivatkozza a rekord összes adatát, míg az egyes mezők (a fenti módon hivatkozva, a típusuknak megfelelő változóként), a rekord egy-egy adatának kezelésére használatosak.

Műveletek

A rekordokra nem értelmezünk műveleteket, de két, azonos típusú rekordváltozó között megengedjük az értékadó utasítást. Az egyes mezőkkel, mint változókkal, minden olyan művelet megengedett, ami az adott mező típusára definiált.

Pl.

```
Konstans
   MAXTARGY  10      /* Félévenként felvehető tárgyak max. száma */
   MAXFELEV  12      /* A félévek max. száma */
   MAXNEVH  30       /* Egy név max. hossza */
Típus
   TANTARGY  Rekord
      NEV             Sztring[30]
      KREDIT, JEGY    Egész
   FELEV  Rekord
      TARGYAK         Egydimenziós TANTARGY tömb[MAXTARGY]
      TARGYDB         Egész
      ATLAG           Valós
      OSSZKREDIT      Egész
```

```
HALLGATO  Rekord
    NEPTUNKOD       Sztring[6]
    NEV             Sztring[MAXNEVH]
    FELEVEK         Egydimenziós FELEV tömb[MAXFELEV]
    FELEVDB         Egész
```

Ha H egy HALLGATO típusú változó, akkor

H.NEV	a hallgató nevét
H.FELEVEK[1].ATLAG	az első féléves átlagát
H.FELEVEK[2].TARGYAK[3].KREDIT	a 2. félévben felvett 3. tárgy kreditpontját

hivatkozza.

9.4.1. Üzletek tartozása

Az alábbi példában rekordokat és egydimenziós tömböket használunk, de a feladat megoldásához más, akár rekordok nélküli adatstruktúra is elképzelhető.

Feladat: Egy gyár *N* db terméket készít, és *M* db üzletnek szállítja ezeket. Tudjuk a termékek nevét és egységárát, valamint azt, hogy melyik üzlet, melyik termékből mennyit kapott. Kérjük be az adatokat, majd írjuk ki az üzletek tartozását! Az üzleteket egyszerűen a sorszámukkal azonosítjuk, és feltesszük, hogy legfeljebb 10 termékünk és 20 üzletünk van.

Megoldás: Az adatbekérésnél először a termékek adatait kérjük be, így az üzleteknek szállított mennyiségek bekérésénél már kiírhatjuk az egyes termékek nevét, megkönnyítve ezzel az adatok megadását. Egy termék adatait (név, egységár) egy rekordba fogjuk össze, így a termékek adatai egy rekordokból álló egydimenziós tömbben tárolhatók. Az üzletek adataiból (szállított mennyiségek, tartozás) szintén rekordot képezünk. Mivel mind az egységár, mind a szállított mennyiségek egészek, ezért a tartozásértékek is egészek lesznek, programíráskor azonban célszerű a legnagyobb egész típust használni, az esetleges túlcsordulások elkerülése végett. A tartozások kiszámítása egy egyszerű összegszámolás, amelyben a tagokat az egyes darabszámok és a hozzájuk tartozó egységárak szorzata adja.

```
Konstans
    NMAX  10        /* Termékek maximális száma */
    MMAX  20        /* Üzletek maximális száma */
    NEVMAXH  30     /* Terméknevek maximális hossza */
```

Típus

TERMEK	Rekord	/* Termék */
NEV	Sztring[NEVMAXH]	/* Név */
EAR	Egész	/* Egységár */
UZLET	Rekord	/* Üzlet */
DB	Egydimenziós egész tömb[NMAX]	/* Száll. mennyiségek */
TART	Egész	/* Tartozás*/

Funkció	Azonosító	Típus	Jelleg
A termékek száma	N	Egész	I
Az üzletek száma	M	Egész	I
A termékek adatai	T	Egydimenziós TERMEK tömb[NMAX]	I
Az üzletek adatai	U	Egydimenziós UZLET tömb[MMAX]	I, M, O
Segédváltozók	I, J	Egész	M

```
/* Adatbekérés */
Be: N,M
for I ← 1,N
      Be: T[I].NEV,T[I].EAR
for I ← 1,M
      Ki: I,". üzletnek szállított mennyiségek"
      for J ← 1,N
            Ki: T[J].NEV
            Be: U[I].DB[J]
/* Tartozások kiszámítása */
for I ← 1,M
      U[I].TART ← 0
      for J ← 1,N
            U[I].TART ← U[I].TART+U[I].DB[J]*T[J].EAR
/* Eredménykiírás */
Ki: "Tartozások"
for I ← 1,M
      Ki: I,". üzlet:",U[I].TART
```

9.4.2. Feladatok

- Adott *N* hallgató és *M* db tantárgy. Ismerjük a hallgatók nevét, az egyes tantárgyakból kapott érdemjegyeiket (minden hallgatónak minden tárgyból pontosan egy db érdemjegye van), valamint a tantárgyak neveit és kreditpontjait. Határozzuk meg a hallgatók összes teljesített

kreditpontját (a nem elégtelen érdemjegyű tárgyak kreditpontjainak összege) és tantárgyanként a jegyek százalékos megoszlását (azaz, hogy a hallgatók hány százaléka kapott abból a tárgyból 1-es, 2-es, … , 5-ös érdemjegyet)!

10. Szubrutinok

Szubrutinon egy olyan, jól meghatározott, önálló tevékenységsort megvalósító, formailag is elkülönülő programrészt értünk, amelynek tényleges végrehajtásához egy másik, a szubrutint aktivizáló, azt „meghívó” programrész is szükséges. Egy szubrutin többször is meghívható, így annak szolgáltatásai a program különböző helyein igénybe vehetők.

A szubrutin a programtervezés, ill. programozás alapszintű, de egyben alapvető fontosságú eszköze. Használatukkal érvényesíthető az egységbezárási elv, miszerint egy adott feladat megoldásához szükséges adatoknak és a „rajtuk dolgozó” algoritmusnak egy olyan egysége valósítható meg, amely csak a szükséges mértékben kommunikál a „külvilággal”, a szubrutint hívó programrészekkel, egyébként zárt, „dolgait” saját maga „intézi”, azokba „nem enged bepillantást”.

Szubrutinok alkalmazásával programunk logikailag és fizikailag is tagoltabb, áttekinthetőbb, következésképpen könnyebben karbantartható lesz.

Megjegyzés

- Az objektumorientált programozás objektumtípusai (osztályai) az adatoknak és a rajtuk dolgozó algoritmusoknak egy még általánosabb egységét valósítják meg azáltal, hogy az adatokhoz nemcsak egy, de tetszőleges számú algoritmus rendelhető. Az algoritmusokat megvalósító, szubrutinok (más terminológiával metódusok, tagfüggvények) egyrészt „látják” az objektum adatait, másrészt ők definiálják, írják le, modellezik az objektum viselkedését.
- A szubrutinokat szokás még alprogramoknak is nevezni [Nyé 03].

Megkülönböztetjük a szubrutin *deklarálását*, ahol definiáljuk a szubrutin által végrehajtandó tevékenységeket és a külvilággal való kapcsolattartás módját és szabályait, valamint a szubrutin *hívását*, amikor felhasználjuk a szubrutin szolgáltatásait, vagyis definiálva a kapcsolattartás eszközeit, kiváltjuk a tevékenységsor egy végrehajtását.

A kapcsolattartás legfőbb, de nem kizárólagos (hiszen az ún. globális, „mindenhonnan látható”, változókon keresztül is történhet a kommunikáció) eszközei a *paraméterek*.

A helyes programozási stílus a *teljes paraméterezésre* való törekvés. Ez azt jelenti, hogy egy szubrutin, a működéséhez szükséges adatokat a paramé-

terein keresztül kapja és az eredményeket (a függvényérték kivételével) ezeken keresztül adja vissza. Ezzel ugyanis elkerülhető egy esetleges módosítás olyan „mellékhatása”, amely a program egy „távoli részének” végrehajtásakor, előre nem látott, nem tervezett változást (s ezzel többnyire nehezen felderíthető hibát) okoz.
Kétféle szubrutint különböztetünk meg, az *eljárást* és a *függvényt*, amelyek deklarálása és hívása, ha kicsit is, de eltérő.

Deklarálás

Eljárás

```
Eljárásnév(Formális paraméterlista)
        /* Az eljárás utasításai */
```

Függvény

```
Függvénynév(Formális paraméterlista)
        /* A függvény utasításai */
return Függvényérték
```

A *Formális paraméterlista* változók vesszővel elválasztott sorozata.

A függvény eredménye (lásd *Függvényérték*) a *return* utasítás után adandó meg. Egyetlen *return* utasítást engedünk meg, a függvény utolsó utasításaként. Ezzel megőrizhető a megoldás strukturáltsága, a vezérlőszerkezetekhez hasonlóan ennek az „építőelemnek” is egy eleje és egy vége lesz.

Megjegyzés: A szubrutin nevét és formális paramétereit megadó részt a szubrutin fejének, vagy a kapcsolattartásra utaló kifejezéssel interfész résznek, míg a szubrutin utasításait tartalmazó részt a szubrutin törzsének nevezzük.

Hívás

Eljárás

```
Eljárásnév(Aktuális paraméterlista)
```

Az eljárás a nevével, „utasításszerűen” hívható, míg a függvény, mivel egy értéket ad eredményül (a függvény értékét), ezért kifejezésekben, a kifejezések részeként hívható, ugyanúgy, mint a fejlesztőrendszerben „készen kapott” függvények.

Függvény

Pl.

```
Változó ← Függvénynév(Aktuális paraméterlista)
```

Az *Aktuális paraméterlista* kifejezések vesszővel elválasztott sorozata.

A paraméterátadást tekintve megkülönböztetünk:

- *érték szerinti* és
- *cím szerinti* paraméterátadást.

Az érték szerinti paraméterátadásnál az aktuális paraméter értéke, míg cím szerinti átadásnál az aktuális paraméter memóriabeli címe adódik át (következésképpen ilyenkor az aktuális paraméter csak változó lehet, kifejezés nem).

A kétféle paraméterátadás közötti lényeges különbség az, hogy az érték szerint átadott paraméterben a szubrutin csak átvenni tud információt (input jelleg), míg a cím szerint átadott paraméterben átvenni és visszaadni is (input, output jelleg).

Ha egy feladat megoldását szubrutinnal programozzuk, akkor az adatszerkezeti táblázat input jellegű adatait érték szerint adjuk át, az input és output, valamint az output jellegű adatait cím szerint. A szubrutin formális paraméterlistáján csak ezen adatok (tehát csak a bemenő ill. kijövő adatok) változói szerepelnek, míg a munka jellegű adatokat a szubrutin lokális (csak a szubrutin működéséig „élő") munkaváltozóiban tároljuk. A felhasználóval való kommunikáció (pl. adatbekérés, eredménykiírás) ekkor a főprogramban, mint hívó környezetben programozható.

Megjegyzés

- A paramétereket a szubrutinok egy speciális memória, a verem (lásd 12.) segítségével, azon keresztül veszik át. Ide kerülnek egyébként a szubrutinok lokális változói is.
- A nagy helyigényű adatok (pl. tömbök) paraméterként való átadásakor célszerű cím szerinti paraméterátadást választani (az input jelleg ellenére is), hiszen így megtakarítható a tömb adatainak verembe másolása, ami ritka kivételektől eltekintve teljesen felesleges.
- A szubrutinok lehetnek rekurzívak is, azaz önmagukat is meghívhatják (lásd 11.5.).
- Az, hogy egy szubrutint eljárás vagy függvény formájában írunk meg, nem elvi kérdés, inkább programozói stílus kérdése. A két forma egymással ekvivalens módon mindig helyettesíthető. Megoldásainkban a feladathoz jobban illeszkedőt választjuk. Programmá íráskor nem mindig választhatunk, például a C nyelv csak függvényeket használ, itt az „eredmény nélküli" függvényeket tekinthetjük eljárásoknak is.

Egy-egy rövidke példával szeretnénk szemléltetni az eddigieket.

Feladat: Cseréljük fel két egész típusú változó tartalmát!

Megoldás: Ha az egyik változó tartalmát áttesszük a másik változóba, akkor annak eredeti tartalma elvész, pedig arra szükségünk van, ezért egy segédváltozót vezetünk be, ahol ideiglenesen megőrizzük ezt az értéket.

Funkció	Azonosító	Típus	Jelleg
Az egyik szám	A	Egész	I, O
A másik szám	B	Egész	I, O
Csereváltozó	CS	Egész	M

Deklarálás (megoldás)

```
CSEREL(A,B)

CS ← A
A ← B
B ← CS
```

Hívás

```
/* Az eljárás egy hívása */
Be: A,B
CSEREL(A,B)
Ki: A,B
```

Feladat: Határozzuk meg két egész szám közül a kisebbet!

Megoldás: A megoldásra készített függvényben egyszerűen a kisebbik értéket adjuk vissza.

Funkció	Azonosító	Típus	Jelleg
Az egyik szám	A	Egész	I
A másik szám	B	Egész	I
Az eredmény	ER	Egész	O

Deklarálás (megoldás)

```
MINIMUM(A,B)

if A<B
        ER ← A
else
        ER ← B
return ER
```

Hívás

```
/* A függvény egy hívása */
Ki:”5 és 3 közül a kisebb:”, MINIMUM(5,3)
```

Megjegyzés

- Az első feladat eljárásának hívásakor az aktuális paraméterek, output jellegük miatt tetszőleges egész változók lehetnek. Az egyszerűség kedvéért (hogy ne kelljen újabb adatszerkezeti táblázatot készíteni a hívó számára) ugyanazon változóneveket használtuk.
- A második feladat megoldásában az aktuális paraméterek, input jellegük miatt tetszőleges egész kifejezések lehetnek, most egyszerű konstansok.
- A hívó környezetet most csak a hívás szemléltetésére készítettük, a későbbiekben elhagyjuk. A függelékben található forrásprogramokban ezek természetesen megtekinthetők.

11. Algoritmusok

11.1. Algoritmusok hatékonysága

Ha egy feladat megoldására készített algoritmusokat össze szeretnénk hasonlítani, akkor szükségünk van egy olyan jelölésre, amellyel egyszerűen jellemezhetők, minősíthetők az egyes algoritmusok.

Tekintsük példaként a tömbrendező algoritmusokat. Amellett, hogy természetesen maguktól a bemenő adatoktól is függ az, hogy melyik módszer mennyi ideig fut, hány műveletet hajt végre, elsősorban a rendezendő elemek száma az, amely döntően meghatározza a végrehajtási lépések számát, azok nagyságrendjét.

Ha egy algoritmus pl. $2n^2+3n+1$ műveletet végez (ahol n jelöli az algoritmusban résztvevő elemek számát), akkor kellően nagy n-re a legnagyobb kitevős (jelen esetben a négyzetes) tag dominál. A konstans szorzóktól és a kisebb kitevős tagoktól eltekintve azt mondjuk, hogy ez az algoritmus n^2 nagyságrendű.

11.2. Elemi statisztikák

11.2.1. Minimumhelyek keresése

Feladat: Adott egy N elemű adatsorozat. Keressük meg a legkisebb elemét, állapítsuk meg hányszor és hol található meg ez az elem a sorozatban!

Megoldás: A feladatot két részre bonthatjuk. Az első rész a legkisebb elem megállapítása, a második megkeresni, hogy ez az érték hol, és hányszor fordul elő a sorozatban. Az első rész megoldására egy korábbi példában (lásd 8.3.) ismertetett módszert alkalmazhatjuk, azaz minimumként megjegyezzük az első elemet, és a többi elemet ehhez az aktuális minimumhoz hasonlítjuk. Ha kisebb elemet találunk, akkor azt jegyezzük meg minimumként. A sorozat elemeit tömbbel kezeljük, hiszen a második rész megoldásakor újra szükség lesz ezekre az értékekre. A minimumérték előfordulási számát és előfordulási helyét meghatározandó tegyük a következőket. A minimális elem előfordulásainak száma legyen nulla. Vizsgáljuk meg a sorozat összes elemét, és ha a minimummal megegyező elemet találunk, akkor növeljük meg ezt a darabszámot eggyel és jegyezzük fel az elem indexét.

Funkció	Azonosító	Típus	Jelleg
A sorozat elemei	A	Egydimenziós, tetszőleges elemtípusú tömb	I
A sorozat elemeinek száma	N	Egész	I
A legkisebb elem értéke	MIN	Az A tömb elemeivel megegyező típusú	M, O
A legkisebb elem darabszáma	DB	Egész	M, O
A legkisebb elem indexei	HELY	Egydimenziós egész tömb	M, O
Ciklusváltozó	I	Egész	M

```
/* Minimumhelyek keresése */
MINKERESES(A,N,MIN,DB,HELY)

/* Minimum meghatározása */
MIN ← A[1]
for I ← 2,N
       if A[I]<MIN
              MIN ← A[I]
/* Minimumhelyek */
DB ← 0
for I ← 1,N
       if A[I]=MIN
              DB ← DB+1
              HELY[DB] ← I
```

Megjegyzés: A tetszőleges elemtípusú jelen esetben azt jelenti, hogy az algoritmus működik az összes olyan adatsorra, amelynek elemeire értelmezettek a hasonlítás műveletek. A konkrét elemtípust a vizsgálandó adatsor elemeinek típusa határozza meg.

11.2.2. Átlag és szórás

Feladat: Adott egy *N* elemű számsorozat. Határozzuk meg az elemek átlagát és szórását!

Megoldás: A feladat ugyancsak két részre bontható. Az első rész az átlag meghatározása, amit egy korábbi példában (lásd 8.3.) leírtak szerint végezhetünk, azaz összegezzük az elemeket, majd elosztjuk az összeget az elemek számával. A sorozat elemeit az előző feladathoz hasonlóan tömbben tároljuk. A második rész megoldásához az

$$S = \sqrt{\frac{\sum_{i=1}^{n}(a_i - atl)^2}{n}}$$

képletet használjuk. Vegyük észre, hogy először itt is egy összeget kell kiszámolnunk, majd egy osztás és egy gyökvonás után megkapható a szórás.

Funkció	Azonosító	Típus	Jelleg
A sorozat elemei	A	Egydimenziós valós tömb	I
A sorozat elemeinek száma	N	Egész	I
Az elemek átlaga	ATL	Valós	O
Az elemek szórása	SZ	Valós	O
Az összegzésekhez	OSSZ	Valós	M
Ciklusváltozó	I	Egész	M

```
/* Átlag és szórás */
SZAMOL(A,N,ATL,SZ)
```

```
/* Átlag */
OSSZ ← 0
for I ← 1,N
      OSSZ ← OSSZ+A[I]
ATL ← OSSZ/N
/* Szórás */
OSSZ ← 0
for I ← 1,N
      OSSZ ← OSSZ+SQR(A[I]-ATL)
SZ ← SQRT(OSSZ/N)
```

11.2.3. Előfordulási statisztika

Feladat: Adott egy *N* elemű adatsorozat. Gyűjtsük ki a különböző elemeket és ezek darabszámát!

Megoldás: Egy olyan táblázatot készítünk, amelynek első oszlopában, a sorozatban előforduló különböző elemek lesznek, a második oszlopában ezek előfordulási száma. Kezdetben a táblázat üres, majd sorra megvizsgáljuk a sorozat elemeit. Minden elemet összehasonlítunk a táblázat első oszlopában lévő elemekkel, ha találunk egyezőt, akkor a második oszlop megfelelő elemének értékét eggyel növeljük, ha nem találtuk meg az elemet, akkor az első oszlop végére írjuk, a második oszlop végére egyet írunk, hiszen ennek az elemnek ez az első előfordulása. A megoldás ezen leírás alapján „szó sze-

rint" is elkészíthető (pl. a 11.3.2. fejezetbeli kereséssel), ezt az olvasóra bízzuk. A bemutatott megoldásunkban végeredményben ugyanezt fogjuk tenni, csak egy kicsit másképp. A keresést megkönnyítendő ugyanis, a sorozat elemeit mindig felvesszük a (kezdetben üres) táblázat első oszlopának legvégére 0 darabszámmal, így a táblázat elejétől induló keresés mindenképpen megtalálja majd a kérdéses elemet, ha máshol nem, akkor a táblázat végén. Ott, ahol megtaláljuk, a táblázat második oszlopában megnöveljük eggyel a darabszámot. Ha nem a táblázat legvégén találtuk meg a vizsgált elemet, akkor ő már szerepelt a táblázatban, ezért a táblázat elemeinek számát csökkentjük eggyel, törölve ezzel a táblázat végén lévő, 0 darabszámú, felesleges elemet.

Funkció	Azonosító	Típus	Jelleg
A sorozat elemei	A	Egydimenziós, tetszőleges elemtípusú tömb	I
A sorozat elemeinek száma	N	Egész	I
A táblázat első oszlopa	T	Az A tömbbel megegyező típusú	M, O
A táblázat második oszlopa	DB	Egydimenziós egész tömb	M, O
A különböző elemek száma	K	Egész	M, O
A vizsgált sorozatelem indexe	I	Egész	M
Segédváltozó a kereséshez	J	Egész	M

```
/* Előfordulási statisztika */
STATISZTIKA(A,N,T,DB,K)

/* Kezdőérték */
K ← 0
for I ← 1,N
       /* Felvétel */
       K ← K+1
       T[K] ← A[I]
       DB[K] ← 0
       /* Keresés */
       J ← 1
       while A[I]<>T[J]
              J ← J+1
       /* Darabszám növelés */
       DB[J] ← DB[J]+1
       /* Ha volt már ilyen, töröljük a táblázat végéről */
       if J<K
              K ← K-1
```

Megjegyzés: A tetszőleges elemtípusú jelen esetben azt jelenti, hogy az algoritmus működik az összes olyan adatsorra, amelynek elemeire értelmezettek a hasonlítás műveletek. A konkrét elemtípust a vizsgálandó adatsor elemeinek típusa határozza meg.

11.2.4. Jelstatisztika

Feladat: Adott egy sztring. Készítsünk statisztikát a benne előforduló jelekről!

Megoldás: A feladatot az előző probléma speciális eseteként is felfoghatjuk, tehát az előző megoldás is alkalmazható. Az ehhez szükséges módosításokat az olvasóra bízzuk.

A kockadobás gyakorisághoz (lásd 9.1.2.) hasonlóan egy olyan megoldást adunk, amelyben kihasználjuk a feladat specialitását, nevezetesen azt, hogy a sztring egy olyan jelsorozat, amelyben maximum 255 db különböző jel lehet (hiszen egy sztring max. 255 karakter hosszú lehet), és ezek kódja 0 és 255 közé esik.

Egy 0-tól 255-ig indexelt tömbben (DB) gyűjtjük majd az egyes jelek előfordulási gyakoriságát. A tömb *i*. eleme annak a karakternek a darabszámát tárolja, amelynek ASCII kódja *i*. Kezdetben a tömb minden elemét nullára állítjuk, majd a sztring jeleit sorbavéve mindig annak a tömbelemnek az értékét növeljük eggyel, amelynek indexe megegyezik a kérdéses jel ASCII kódjával.

Az így elkészült tömb segítségével azután előállítható a gyakorisági táblázat, hiszen a nemnulla elemek adják meg a megoldást. A sztringben előforduló jeleket a nemnulla elemek indexéhez, mint ASCII kódhoz tartozó karakterek adják, a darabszámokat maguk a nemnulla elemek. A kigyűjtést most egy olyan táblázatba végezzük, amelyet egy egydimenziós, rekordokból álló tömb reprezentál. A rekordoknak két mezője van, egyik az adott jelet, a másik a hozzá tartozó darabszámot tárolja. A tömböt elölről kezdve nézzük végig, ezért az eredményül kapott gyakorisági táblázat a jelek (kódjai) szerint rendezett lesz.

Típus
ELEM Rekord /* A gyakorisági táblázat egy eleme */
JEL Karakter
DARAB Egész

Funkció	Azonosító	Típus	Jelleg
A vizsgálandó jelsorozat	S	Sztring	I
A gyakorisági táblázat	T	Egydimenziós ELEM tömb[255]	O
A különböző jelek száma	K	Egész	O
A darabszámok tömbje	DB	Egydimenziós egész tömb[255]	M
A vizsgált jel kódja	KOD	Egész	M
Ciklusváltozó	I	Egész	M

```
/* Jelstatisztika */
STATISZTIKA(S,T,K)

/* Kezdőérték */
for I ← 0,255
       DB[I] ← 0
/* Darabszámok meghatározása */
for I ← 1,LENGTH(S)
       KOD ← ASC(S[I])
       DB[KOD] ← DB[KOD]+1
/* Eredménytáblázat elkészítése */
K ← 0
for I ← 0,255
       if DB[I]>0
              K ← K+1
              T[K].JEL ← CHR(I)
              T[K].DARAB ← DB[I]
```

Megjegyzés

- A DB tömböt most kivételesen 0-tól kezdődően indexeljük, hiszen a 0. elem a 0-s kódú karakter előfordulási darabszámát adminisztrálja.
- A kockadobások gyakoriságának meghatározásánál (lásd 9.1.2.) a darabszámokat gyűjtő tömbben a megfelelő elem kiválasztásához a tömböt magukkal az adatokkal (dobásokkal) indexeltük, itt most a jelek kódjaival. Megjegyezzük azonban, hogy a Pascal nyelvben karakterekkel is indexelhető egy tömb, a C nyelvben meg egy karakter és kódjának használata eleve ekvivalens, azaz az indexelés magával a karakterrel is történhet.

11.3. Rendezés és keresés

A rendezés egy nagyon sokszor alkalmazott adatkezelési eljárás. Szükségessége nyilvánvaló: az adatokat azért tároljuk, hogy lekérdezhessük, visszakereshessük őket, róluk különféle kimutatásokat, eredménylistákat készítsünk. Egy rendezett adatsorban nemcsak az ember keres gyorsabban (gondoljunk csak a telefonkönyvben való név ill. telefonszám megkeresésének különbözőségére), de a számítógép is.

A rendezési és keresési algoritmusokat egy egydimenziós tömb segítségével mutatjuk be. A rendezendő adatok, mint a tömböknél általában, elölről kezdve folyamatosan feltöltve találhatók a tömbben és a rendezett adatsor is itt keletkezik, azaz az algoritmusaink helyben rendeznek. A rendezendő adatok, így a tömb elemeinek típusa tetszőleges olyan típus lehet, amelyre értelmezettek a hasonlítás műveletek.

```
Típus
  ELEM  Egész                          /* A rendezendő elemek típusa */
  TOMB  Egydimenziós ELEM tömb         /* Az elemeket tároló tömb típusa */
```

A következőkben bemutatott három, egyszerű rendező algoritmusban a tömb két részre oszlik. A tömb első felében a már rendezett elemek találhatók, a tömb hátsó, másik felében a még rendezetlen elemek. Az algoritmusokban egy iteráció segítségével a rendezetlen rész egy eleme átkerül a rendezett részbe. Ezt az iterációt $n-1$-szer végrehajtva (ahol n a rendezendő elemek számát jelöli) $n-1$ darab elem kerül a helyére (amivel egyben az n. elem is), így rendezett lesz az adatsorunk.

11.3.1. Egyszerű rendezések

Feladat: Rendezzünk egy tömbben lévő adatsort növekvő (nem csökkenő) sorrendbe!

Buborékrendezés

Az algoritmus ismertségét és nevét a szemléletes alapgondolatának köszönheti, mivel a rendezés során az egyes elemek úgy kerülnek a helyükre, mint ahogy a folyadékban száll fel a buborék.

Képzeljük el a rendezendő elemeket egymás alatt, függőlegesen. Hasonlítsuk össze a sorozat összes szomszédos elemét alulról kezdve felfelé haladva, tehát először az utolsó és utolsó előtti elemet, majd az utolsó előtti és az azt megelőzőt,… legvégül a másodikat az elsővel. Ha a hasonlí-

tásoknál a kisebb elem van lejjebb, akkor cseréljük fel őket, azaz a kisebb elem (mint egy buborék) felfelé „száll”, a nagyobb lefelé „süllyed”.

Könnyen belátható, hogy egy ilyen menet során, esetleg néhány hasznos csere mellett (amikor is nem a legkisebb elem emelkedik), helyére kerül a legkisebb elem. Ezután, mivel a legkisebb elem már a helyén van (legelöl), ezért a következő, szintén alulról induló menetben elegendő a harmadik és második elemekig hasonlítani, hiszen a második menetben a második legkisebb elem száll felfelé a helyére, amely nem lehet kisebb a legkisebb elemnél, így a második helynél feljebb nem fog szállni. Hasonlóan az *n–1*. menetnek, amely az *n–1*. elemet viszi a helyére csupán egyetlen elempárt kell megvizsgálnia, az utolsó és utolsó előtti elemeket.

Az alábbi ábra oszlopai az egyes menetek után kialakult állapotot szemléltetik, egy 6 elemű adatsor rendezése során. K az adatsor kiinduló, kezdőállapotát jelenti.

K	**1.**	**2.**	**3.**	**4.**	**5.**
3	1	1	1	1	1
6	3	2	2	2	2
2	6	3	3	3	3
1	2	6	4	4	4
5	4	4	6	5	5
4	5	5	5	6	6

11.1. ábra. A buborékrendezés.

Vegyük észre, hogy:

- míg a legkisebb elem egy menetben a helyére kerül, addig a legnagyobb minden menetben csak egy helyet lép lefelé,
- ha egy menet során nem végeztünk elemcserét, akkor az elemek már rendezettek.

Ezek alapján javítható az algoritmus (lásd 11.7.).

Funkció	Azonosító	Típus	Jelleg
A rendezendő elemek	A	TOMB	I, M, O
Az elemek száma	N	Egész	I
Két elem cseréjéhez	CS	ELEM	M
A helyére kerülő elem indexe	I	Egész	M
A cserélgető ciklus változója	J	Egész	M

```
/* Buborékrendezés */
BUBREND(A,N)
for I ← 1,N-1
      /* I. elemet a helyére */
      for J ← N,I+1,-1
            if A[J]<A[J-1]
                  /* A J. és J-1. elemek cseréje */
                  CS ← A[J]
                  A[J] ← A[J-1]
                  A[J-1] ← CS
```

Rendezés kiválasztással

Keressük meg a tömb legkisebb értékű elemét és tegyük a tömb elejére, majd a második legkisebbet és tegyük a második helyre és így tovább, legvégül az *n–1*. legkisebbet az *n–1*. helyre. A rendezés során tehát a rendezetlen rész minimális értékű elemét felcseréljük a rész első elemével, így a rendezett részt növeljük, a rendezetlen részt csökkentjük egy elemmel. Kezdetben a rendezetlen rész a teljes tömb, azután az első elemet kivéve az összes elem és így tovább, legvégén már csak a két utolsó elem.

Megjegyzés: A maximális értékű elem kiválasztásával csökkenő (nem növekvő) sorrendet kapunk eredményül.

A következő ábra oszlopai az egyes menetek után kialakult állapotot szemléltetik, egy 6 elemű adatsor rendezése során. K az adatsor kiinduló, kezdőállapotát jelenti.

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

11.2. ábra. Rendezés kiválasztással.

Funkció	Azonosító	Típus	Jelleg
A rendezendő elemek	A	TOMB	I, M, O
Az elemek száma	N	Egész	I
Két elem cseréjéhez	CS	ELEM	M
A helyére kerülő elem indexe	I	Egész	M
Az aktuális minimum indexe	K	Egész	M
A minimumkereső ciklus változója	J	Egész	M

```
/* Rendezés kiválasztással */
KIVALREND(A,N)
for I ← 1,N-1
      /* I. elemet a helyére */
      K ← I
      for J ← I+1,N
            if A[J]<A[K]
                  K ← J
      if K>I
            /* Az I. és K. elemek cseréje */
            CS ← A[I]
            A[I] ← A[K]
            A[K] ← CS
```

Rendezés beszúrással

A rendezett rész kezdetben a tömb első eleme, a rendezetlen rész a többi elem. Ezután sorban kivesszük a rendezetlen rész első elemét és beszúrjuk a rendezett rész elemei közé, a rendezettséget megtartva, hasonlóan, mint ha kártyákat vagy egy halom dolgozatot szeretnénk sorba rendezni.

A beszúrandó elemnek úgy készítünk helyet, hogy a nála nagyobb elemeket rendre egy hellyel hátraléptetjük, végül a „megüresedett” helyre egyszerűen berakjuk a beszúrandó elemet. Az elemek hátraléptetését a

rendezett rész legutolsó elemétől, azaz a beszúrandó elemet megelőző elemtől kezdve, a tömb eleje felé haladva végezzük. Mivel az elsőként hátralépő elem éppen a beszúrandó elemet írja felül, ezért a beszúrandó elemet a hátraléptetés előtt megjegyezzük.

Az alábbi ábra oszlopai az egyes menetek után kialakult állapotot szemléltetik, egy 6 elemű adatsor rendezése során. K az adatsor kiinduló, kezdőállapotát jelenti.

K	**1.**	**2.**	**3.**	**4.**	**5.**
3	3	2	1	1	1
6	6	3	2	2	2
2	2	6	3	3	3
1	1	1	6	5	4
5	5	5	5	6	5
4	4	4	4	4	6

11.3. ábra. Rendezés beszúrással.

Funkció	Azonosító	Típus	Jelleg
A rendezendő elemek	A	TOMB	I, M, O
Az elemek száma	N	Egész	I
Két elem cseréjéhez	CS	ELEM	M
A helyére kerülő elem	X	ELEM	M
A helyére kerülő elem indexe	I	Egész	M
A helykészítő ciklus változója	J	Egész	M

```
/* Rendezés beszúrással */
BESZURREND(A,N)

for I ← 2,N
      /* I. elem beszúrása az előtte lévő rendezett részbe */
      X ← A[I]
      /* Helykészítés hátraléptetéssel */
      J ← I-1
      while (J>=1) AND (A[J]>X)
            A[J+1] ← A[J]
            J ← J-1
      /* I. elemet a helyére */
      A[J+1] ← X
```

11.3.2. Lineáris keresések

Feladat: Keressünk meg egy adott elemet egy rendezetlen adatsorban!

Megoldás: Ha az adatok, amelyek között keresünk rendezetlenek, akkor nincs mit tenni, sorban meg kell őket vizsgálni. Az első elemtől kezdve addig vizsgáljuk az adatsor elemeit, amíg meg nem találjuk a keresett adatot, vagy végig nem érünk az adatsoron. Ha tehát a keresett elem többször is előfordul az adatok között, akkor az első előfordulásánál megállunk, ezt az elemet találjuk meg, míg ha nem szerepel az adatok között, akkor a teljes adatsoron végig kell lépkednünk.

Funkció	Azonosító	Típus	Jelleg
Az elemek	A	TOMB	I
Az elemek száma	N	Egész	I
A keresendő elem	X	ELEM	I
A keresés eredménye	HOL	Egész	O
A keresés sikeressége	VAN	Logikai	O
A keresőciklus változója	I	Egész	M

```
/* Lineáris keresés rendezetlen adatok között */
LINKER(A,N,X,HOL)

I ← 1
while (I<=N) AND (A[I]<>X)
       I ← I+1
VAN ← I<=N
if VAN
       HOL ← I
return VAN
```

Feladat: Keressünk meg egy adott elemet egy rendezett adatsorban!

Megoldás: Ha az adataink rendezettek (pl. nem csökkenően), akkor előbb is megállhatunk, hiszen ha a keresett elemnél nagyobb elemre lépünk, akkor felesleges tovább keresnünk, mert ekkor a hátrébb lévő elemek is nagyobbak a keresett elemnél, következésképpen nem lehet közöttük a keresett elem. A keresés tehát vagy a keresendő elem első előfordulásán áll meg, vagy ott, ahol a keresett elem helye lenne a rendezettség szerint.

A lineáris (vagy más néven soros) keresések legrosszabb esetben tehát a teljes adatsort megvizsgálják, így az elemek számával arányos műveletet végeznek.

Funkció	Azonosító	Típus	Jelleg
Az elemek	A	TOMB	I
Az elemek száma	N	Egész	I
A keresendő elem	X	ELEM	I
A keresés eredménye	HOL	Egész	O
A keresés sikeressége	VAN	Logikai	O
A keresőciklus változója	I	Egész	M

```
/* Lineáris keresés rendezett adatok között */
LINKERREND(A,N,X,HOL)

I ← 1
while (I<=N) AND (A[I]<X)
       I ← I+1
VAN ← (I<=N) AND (A[I]=X)
HOL ← I
return VAN
```

Megjegyzés: A ciklusokat vezérlő feltételekben először az adatok „meglétét” vizsgáljuk, így elkerülhető a nemlétező adatra való hivatkozás. A kifejezések kiértékelésénél (lásd 4.2.) ugyanis feltettük, hogy a kiértékelés befejeződik, amint a kifejezés értéke már nem változhat. Példánkban most két relációt egy AND művelet kapcsol össze, így a második reláció csak akkor kerül kiértékelésre, ha az első reláció igaz.

11.3.3. Bináris keresés

Feladat: Keressünk meg egy adott elemet egy rendezett adatsorban!

Megoldás: Ha az adatok, amelyek között keresünk, rendezettek (pl. növekvően), akkor alkalmazható egy, nagyságrendjét tekintve $log_2 n$ műveletigényű keresési módszer (ahol n az adatsor elemeinek számát jelöli), az ún. bináris keresés.

A módszer lényege, hogy a vizsgálandó rész (ami kezdetben a teljes adatsor) középső eleméhez hasonlítjuk a keresett adatot. Ha megegyezik vele, akkor készen vagyunk, megtaláltuk a keresett elem egy előfordulását (de nem feltétlenül az elsőt), különben a keresést már csak egy fele akkora részben folytatjuk tovább. Ha ugyanis a keresett elem kisebb a középső elemnél, akkor csak előtte, azaz a keresési rész első felében, ha nagyobb, akkor csak mögötte, a keresési rész hátsó felében lehet.

Így minden egyes lépésben elhagyjuk az elemek felét, mindig feleződik az a rész, amelyben keresünk, így könnyen belátható, hogy legfeljebb $log_2 n$ felezési lépést kell végrehajtanunk.

A keresési részt úgy szűkítjük, hogy a középső elem már ne legyen benne (hiszen azzal nem egyezik a keresendő elem), így ha a keresett elem nem található az adatsorban, akkor a keresés mintegy rászűkül az elem leendő helyére és a keresési részt kijelölő *i*, *j* indexek átlépik egymást. Ilyenkor az elölről induló *i* index éppen azt mutatja, hogy hol lenne a keresett elem helye a rendezett adatsorban.

	2	2	3	4	4	6
	i		*k*			*j*
	2	2	3	4	4	6
	i	*j*				
	k					
	2	2	3	4	4	6
j	*i*					

11.4. ábra. A bináris keresés lépései az 1 keresésekor.

2	2	3	4	4	6
i		*k*			*j*
2	2	3	4	4	6
			i	*k*	*j*

11.5. ábra. A bináris keresés lépései a 4 keresésekor.

Funkció	Azonosító	Típus	Jelleg
Az elemek	A	TOMB	I
Az elemek száma	N	Egész	I
A keresendő elem	X	ELEM	I
A keresés eredménye	HOL	Egész	O
A keresés sikeressége	VAN	Logikai	O
A keresés helyének kezdőindexe	I	Egész	M
A keresés helyének végindexe	J	Egész	M
A keresés helyének középső indexe	K	Egész	M

```
/* Bináris keresés */
BINKER(A,N,X,HOL)

I ← 1
J ← N
VAN ← hamis
while (I<=J) AND NOT VAN
       K ← (I+J) DIV 2
       if A[K]=X
              VAN ← igaz
       else if X<A[K]
              J ← K-1
       else
              I ← K+1
if VAN
       HOL ← K
else
       HOL ← I
return VAN
```

11.3.4. Indextáblás rendezések, keresések

Az alkalmazások nagy részében az adatok *fizikai* rendezettsége gyakorlatilag nem biztosítható, rendkívül rossz hatékonysággal járna. Ennek oka lehet például az adatok nagy száma, vagy a többféle lekérdezési szempont (fizikailag rendezni csak egy szempont szerint lehet).

A vázolt probléma az informatika egyik központi problémája, a megoldások közös jellemzője, hogy magukat az alapadatokat egy rögzített, ill. relatíve nagyon keveset változó sorrendben tároljuk, és a rendezettséget, sorrendiséget megadó információkat kiegészítő adatokkal adjuk meg. Ezt hívjuk *logikai* rendezésnek, ill. rendezettségnek. A kiegészítő adatok akár nagyon bonyolult szerkezetűek (pl. gráfok) is lehetnek. Az alábbiakban a módszert a lehető legegyszerűbb formájával, az egydimenziós tömbök *indextáblás* rendezésével szemléltetjük.

Az adatstruktúra rendezési szempontonként egy tömbbel (indextábla) bővül, amelyek az egyes szempontokhoz tartozó logikai sorrendet tartalmazzák. Az indextábla indexeket tartalmaz, az alapadatok fizikai indexeit a rendezési szempont szerinti sorrendben, tehát az indextábla elemszáma megegyezik az adatok számával. Jelölje az alapadatok tömbjét A, az indextáblát IT. Ekkor a logikai sorrendben az I. helyen álló elemre az indextábla I. eleme mutat, ez az elem az alapadatok között az IT[I]-edik helyen található, értéke A[IT[I]].

Név szerinti indextábla	Ssz.	Adatok Név	Telefonszám	Tel.szám sz-i indextábla
5	1	Szabó	222222	3
3	2	Varga	777777	1
7	3	Kovács	111111	5
1	4	Takács	444444	4
4	5	Halász	333333	7
6	6	Vadász	666666	6
2	7	Madarász	555555	2

11.6. ábra. Indextáblák használata.

Típus

INDEXTABLA Egydimenziós egész tömb /* Az indextábla */

Az indextáblák használatát egy rendezési (kiválasztásos) és egy keresési (bináris) algoritmus átírásával, valamint egy komplett adatkarbantartási példával (lásd 14.3.2.) szemléltetjük.

Feladat: Rendezzünk egy tömbben lévő adatsort növekvő (nem csökkenő) sorrendbe a kiválasztásos rendezési módszerrel, indextábla használatával!

Megoldás: Tetszőleges rendezési algoritmus átírható indextáblássá az alábbiak szerint:

- Az indextáblát fel kell tölteni az eredeti sorrendet kifejező indexekkel.
- Az adatokra mindig közvetve, az indextáblán keresztül kell hivatkozni.
- Az adatok helyett az indexeik mozognak, tehát az elemek cseréjét átírjuk az indexeik cseréjére.

Funkció	Azonosító	Típus	Jelleg
A rendezendő elemek	A	TOMB	I
Az indextábla	IT	INDEXTABLA	M, O
Az elemek száma	N	Egész	I
Két index cseréjéhez	CS	Egész	M
A helyére kerülő elem indexe	I	Egész	M
Az aktuális minimum indexe	K	Egész	M
A minimumkereső ciklus változója	J	Egész	M

```
/* Rendezés kiválasztással, indextáblával */
KIVALRENDIT(A,IT,N)
/* Az indextábla feltöltése */
for I ← 1,N
       IT[I] ← I
for I ← 1,N-1
       /* I. elemet a helyére */
       K ← I
       for J ← I+1,N
              if A[IT[J]]<A[IT[K]]
                     K ← J
       if K>I
              /* Az I. és K. indexek cseréje */
              CS ← IT[I]
              IT[I] ← IT[K]
              IT[K] ← CS
```

Feladat: Keressünk meg egy adott elemet egy indextáblával rendezett adatsorban!

Megoldás: Kereséskor szintén az indextáblán keresztül hivatkozunk az elemekre, a keresés eredményeként előálló hely értéke (HOL) a logikai sorrend szerinti index lesz. Megoldásunkban a bináris keresés algoritmusát alkalmaztuk.

Funkció	Azonosító	Típus	Jelleg
Az elemek	A	TOMB	I
Az indextábla	IT	INDEXTABLA	I
Az elemek száma	N	Egész	I
A keresendő elem	X	ELEM	I
A keresés eredménye	HOL	Egész	O
A keresés sikeressége	VAN	Logikai	O
A keresés helyének kezdőindexe	I	Egész	M
A keresés helyének végindexe	J	Egész	M
A keresés helyének középső indexe	K	Egész	M

```
/* Bináris keresés indextáblával */
BINKERIT(A,IT,N,X,HOL)

I ← 1
J ← N
VAN ← hamis
while (I<=J) AND NOT VAN
       K ← (I+J) DIV 2
       if A[IT[K]]=X
              VAN ← igaz
       else if X<A[IT[K]]
              J ← K-1
       else
              I ← K+1
if VAN
       HOL ← K
else
       HOL ← I
return VAN
```

11.3.5. Hatékony rendezések

Az eddig bemutatott rendezések műveletigénye az elemek számának *(n)* négyzetével *(n^2)* arányos. Az ennél hatékonyabb (*$n*log_2 n$* műveletigényű) rendezésekről (gyorsrendezés rekurzívan, saját veremmel, ill. kupacrendezés) a megfelelő, a szükséges eszközöket ismertető fejezetekben lesz szó.

11.4. Ellenőrzött input

11.4.1. Alapfogalmak

Egy rendszer működésénél alapvető követelmény, hogy a „kívülről érkező", tehát még nem ellenőrzött (pl. adatbevitellel létrejövő, ekkor keletkező) adatok maximálisan ellenőrizve legyenek a rendszerben való tárolás, ill. felhasználás előtt. Tipikus adatbeviteli forma a billentyűzetről érkező jelsorozat, ebben a fejezetben ezzel foglalkozunk.

Ellenőrzött input alatt azt az adatbevitel-programozási technikát értjük, amellyel az adatbevitel formai, ill. tartalmi hibáit minél hamarabb, lehetőleg már a hiba keletkezésének pillanatában, optimális esetben már a hibát okozó jel beérkezésekor észreveszi és lekezeli a program (utólagos hibaüzenetek helyett), ezzel megakadályozva a hibás adat keletkezését.

A billentyűzet a számítógép egy perifériaegysége, külső eszköze, ezért a hozzáférés, kezelés módja erősen függ az operációs rendszertől. Ez a függőség érvényesül a programfejlesztő környezetekben is. Egy hagyományos, a szekvenciális programvégrehajtás alapelvére épülő (pl. *MS DOS*

alapú) fejlesztőrendszerben (a billentyűzet-kezeléshez adott eszközöket felhasználva) teljes mértékben „kezünkben tarthatjuk" a beolvasást, a billentyűzetről érkező jeleket egyenként elemezhetjük, akár normál adatjelek (pl. betűk), akár vezérlőjelek (pl. *Esc*, *Enter*, *Backspace*) ezek.

Más a helyzet egy alapvetően az eseménykezelés elvére épülő (pl. *Windows* alapú) fejlesztőrendszerben. Itt a program, némi egyszerűsítéssel, az egyes események (pl. egérkattintás) bekövetkezésekor végrehajtandó szubrutinokból áll. Az operációs rendszer egyrészt eleve lekezel bizonyos eseményeket, így az input folyamat egyes részletei (pl. egyes vezérlőjelek kezelése) rejtve marad a programozó elől, másrészt egyes szolgáltatásai (pl. egér, vágólap) is megnehezíti a jelenkénti ellenőrzést.

Az adatbevitelnél elsődlegesen keletkező adat (az általánosság miatt) formailag mindig egy sztring, így az ellenőrzési technikák is a sztringek és jelhalmazok kezelésére épülnek. Két, különböző, általánosan alkalmazható módszert ismertetünk (az egyes fejlesztőrendszerekhez igazodóan), amelylyel az adatbekérés, ill. ellenőrzés elvégezhető.

Az egyik a *halmazkonstrukciós* módszer, amelynek alapelve, hogy a megadott adat aktuális állapotának függvényében, az adatbevitel minden pillanatában meg tudjuk határozni a következő jelként elfogadható jelek halmazát és csak ilyen jeleket fogadunk el, dolgozunk fel, más jeleket figyelmen kívül hagyunk.

A másik az *utólagos ellenőrzés* módszere, amely nem tartalmaz bekérést, csak ellenőrzést, egyszerűen eldöntjük egy kapott adatról (sztringről), hogy helyes-e vagy sem. Egy ilyen, ellenőrző szubrutin segítségével azután a megfelelő helyeken (pl. az adat megváltozásakor, az adatmegadás befejeztével) elvégezhető az ellenőrzés, és lekezelhetők az esetleges hibák (pl. az adat előző, még helyes értékének visszaállítása, hibaüzenet, stb.). Ilyen ellenőrzést láttunk egy korábbi példában (lásd 9.2.3.), ezzel a módszerrel nem foglalkozunk részletesebben.

11.4.2. Halmazkonstrukció

A hibák kiszűrésére, kizárására jelhalmazokat építünk, az adatot egy sztring típusú változóban tároljuk. A könnyebb használhatóság érdekében az ellenőrzött adatbevitelt szubrutinok formájában definiáljuk. Lehetőséget biztosítunk új, még nem létező adat *bekérésére*, és egy már meglévő, helyes adat *módosítására*. Ha a paraméterként használatos sztringváltozóban üres sztringet kapunk, akkor adatbekérés, egyébként a kapott adat módosítása történik.

Az egyszerűség kedvéért csak 3 vezérlőjelet értelmezünk:

- Adatbevitel közben csak a *Backspace* billentyűvel javíthatunk.
- Az adatbevitel végét az Enter billentyűvel jelezzük (van új, helyes adat).
- Az adatbevitelt bármikor megszakíthatjuk az *Esc* billentyűvel (nincs új, helyes adat).

Egyéb jellegzetességek:

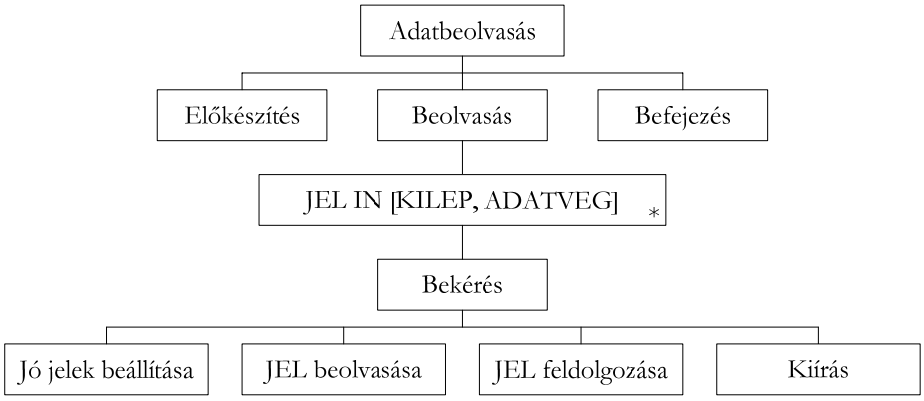
- Az adatbevitel helye a képernyőn paraméterezhető.
- Az adat helyét a lehetséges maximális hosszban, inverz csíkkal jelezzük.
- Formailag kizárt, hibás jel nem vihető be, leütése figyelmen kívül marad.

Az adatot tároló sztringváltozót karaktertömbként kezelve állítjuk be sztring aktuális értékét, amely a kijelzés kedvéért maximális hosszban, jobbról szóközökkel feltöltött. A mindenkori valódi adathosszt egy külön változóban tároljuk. A billentyűzetről úgy olvasunk, olyan standard függvénnyel, amely nem jeleníti meg a leütött karaktert, hiszen az lehet rossz is.
Az adatbevitelnél és a képernyőre írásnál az alábbi segédszubrutinokat használjuk:

- INVERZIR, NORMALIR: a karakter- és háttérszínt állítva beállítják a kiírás módját az inverz színkiemeléshez, ill. annak levételéhez.
- BILLBE: általános jelbekérő, amely mind a normál-, mind a funkcióbillentyűk (két jelet adó billentyűk) kezelésére alkalmas (lásd 9.3.4.).
- JELBE: a BILLBE egy alkalmazása az egy jelet adó (normál) billentyűk kezelésére.
- IR: a pozícionálást és a képernyőre írást vonja össze úgy, hogy a képernyő egy adott helyétől (oszlop, sor) kezdődően kiírja a paraméterként megadott sztringet.
- KIIRAS: az adatbekérő szubrutinok általános kiíró eljárása, amely az adattal, a hellyel (képernyő koordináták), valamint az adat logikai hosszával paraméterezendő. Az IR segítségével kiírja az adatot a megfelelő helyre és a kurzort (a hossz alapján) az adat végére állítja, így a kurzor nem az inverz csík végén, hanem a legutoljára bevitt jel után fog megjelenni.
- JOBBTOLT: egy adott sztringet, adott hosszra, jobbról szóközökkel tölt fel.

Az alábbi két esetet különböztetjük meg:

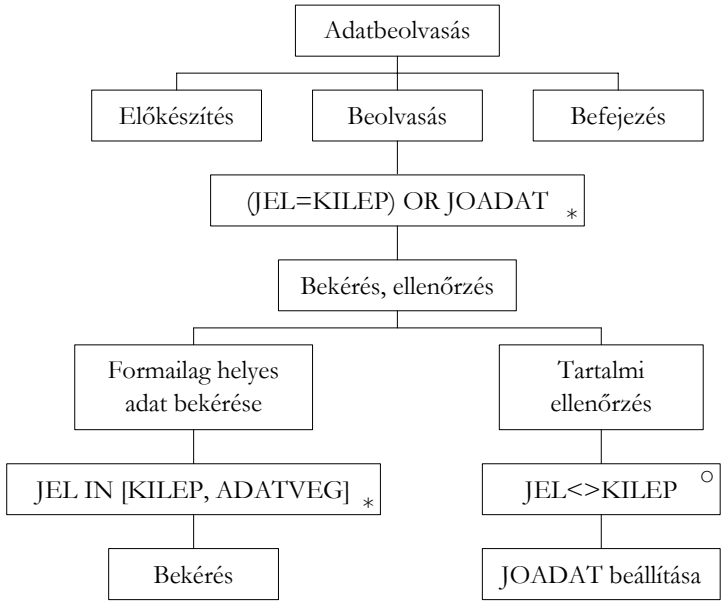
- Elegendő a formai ellenőrzés, ez már biztosítja az adat helyességét.
- Tartalmi ellenőrzés is szükséges, azaz a formai követelményeken kívül más feltételeket is szabunk az adat helyességére.



11.7. ábra. Megoldási séma csak formai ellenőrzéssel.

A megoldás váza tehát egy hátultesztelő ciklus, amely kilépésig, vagy az adat megadásának végéig tart. A cikluson belül az aktuális állapottól függően először beállítjuk az elfogadható jó jelek halmazát (halmazkonstrukció), ahol is érvényesítjük az adatra vonatkozó formai előírásokat, specifikációkat. Ezután a bekért jel – az előzetes konstrukció és szűrés következtében – csak olyan jel lehet, amely itt helyes. Ezt a beolvasás után feldolgozzuk, azaz beillesztjük az adatba (adatjel), vagy végrehajtjuk a megfelelő akciót (visszatörlés, kilépés). A kilépési jelet a ciklus végfeltétele dolgozza fel.

Ha a bekérendő ill. módosítandó adatra vonatkozóan tartalmi ellenőrzést is végeznünk kell, akkor azt, a formailag helyes adat megadása után végezhetjük.



11.8. ábra. Megoldási séma tartalmi ellenőrzéssel.

A megoldási sémákban az aktuálisan bekért jelet a JEL, karakter típusú változó tartalmazza, a kilépést a KILEP, az adatmegadás végét az ADATVEG karakterkonstansok jelölik, míg a tartalmi helyességet a JOADAT logikai változóban tároljuk. A visszatörlésre (a későbbiekben) a TOROL karakterkonstanst fogjuk használni. Mindkét sémát egy-egy konkrét példán szemléltetjük.

Rendszám bekérése

Feladat: Olvassunk be ellenőrzött inputtal egy, az alábbiak szerint specifikált gépjárműrendszámot:

- A rendszám pontosan 6 jegyű.
- Az első két jel angol nagybetű, a harmadik angol nagybetű vagy számjegy, a többi jel számjegy.
- A számrész nem lehet csupa 0.

Megoldás: Az adat fix hosszúságú, kötött formátumú, formai ellenőrzéssel biztosítható a helyessége. Az utolsó (6.) pozíción nem fogadjuk el a '0' karaktert, ha a megelőző számrész csupa 0.

Funkció	Azonosító	Típus	Jelleg
A rendszám	RENDSZAM	Sztring	I, M, O
Képernyő pozíció oszlop	OSZL	Egész	I
Képernyő pozíció sor	SOR	Egész	I
Az adat létezése, érvényessége	VANADAT	Logikai	O
Az aktuális jel	JEL	Karakter	M
Az aktuális jelhalmaz	JOJEL	Karakterhalmaz	M
Az aktuális valódi hossz	HOSSZ	Egész	M

```
/* Rendszám bekérés/módosítás */
RENDSZAMBE(RENDSZAM,OSZL,SOR,VANADAT)

/* Előkészítés */
HOSSZ ← LENGTH(RENDSZAM)
RENDSZAM ← JOBBTOLT(RENDSZAM,6)
INVERZIR
KIIRAS(RENDSZAM,OSZL,SOR,HOSSZ)
/* Beolvasás */
repeat
       /* Jó jelek beállítása */
       JOJEL ← [KILEP]
       if HOSSZ IN [0,1]
              JOJEL ← JOJEL+['A'..'Z']
       else if HOSSZ=2
              JOJEL ← JOJEL+['A'..'Z','0'..'9']
       else if HOSSZ IN [3..5]
              JOJEL ← JOJEL+['0'..'9']
       if HOSSZ>0
              JOJEL ← JOJEL+[TOROL]
       if (HOSSZ=5) AND (RENDSZAM[3] IN ['A'..'Z','0']) AND
          (COPY(RENDSZAM,4,2)="00")
              JOJEL ← JOJEL-['0']
       if HOSSZ=6
              JOJEL ← JOJEL+[ADATVEG]
       /* Jel beolvasása */
       JEL ← JELBE(JOJEL)
       /* Jel feldolgozása */
       if JEL IN ['A'..'Z','0'..'9']
              HOSSZ ← HOSSZ+1
              RENDSZAM[HOSSZ] ← JEL
       else if JEL=TOROL
              RENDSZAM[HOSSZ] ← ' '
              HOSSZ ← HOSSZ-1
       /* Kiírás */
       KIIRAS(RENDSZAM,OSZL,SOR,HOSSZ)
```

```
until JEL IN [KILEP,ADATVEG]
/* Befejezés */
NORMALIR
if JEL=KILEP
       KIIRAS(JOBBTOLT("",6),OSZL,SOR,6)
       RENDSZAM ← ""
       VANADAT ← hamis
else
       KIIRAS(RENDSZAM,OSZL,SOR,HOSSZ)
       VANADAT ← igaz
```

Egész szám bekérése

Feladat: Olvassunk be ellenőrzött inputtal egy adott, zárt intervallumba eső egész számot!

Megoldás: Ha az intervallum alsó határa negatív, akkor első jelként a '–' előjelet is meg kell engednünk. A beolvasandó adat maximális hosszát az intervallum határainak értékei közül a több jelből álló hosszára kell beállítanunk (pl. [–1, 1] intervallum esetén 2-re). Az adatmegadás végét jelző karaktert (ADATVEG) csak akkor engedjük meg, ha az adat utolsó jele számjegy. A tartalmi ellenőrzést a formailag helyes adat megadása után végezzük, ahogyan az a megoldási sémában látható (lásd 11.8. ábra). A számjegyeket a SZAMJEGYEK karakterhalmaz tartalmazza.

Funkció	Azonosító	Típus	Jelleg
Az egész szám sztringként	SZAMSZOV	Sztring	I, M, O
Képernyő pozíció oszlop	OSZL	Egész	I
Képernyő pozíció sor	SOR	Egész	I
Az elfogadható legkisebb érték	TOL	Egész	I
Az elfogadható legnagyobb érték	IG	Egész	I
Az adat létezése, érvényessége	VANADAT	Logikai	O
Az egész szám számként	SZAMERT	Egész	O
Az aktuális jel	JEL	Karakter	M
Az aktuális jelhalmaz	JOJEL	Karakterhalmaz	M
Az aktuális valódi hossz	HOSSZ	Egész	M
A maximális mezőszélesség	MAXH	Egész	M
Munkaváltozó a konverzióhoz	X	Valós	M

```
/* Egész szám bekérés/módosítás */
EGSZAMBE(SZAMSZOV,OSZL,SOR,TOL,IG,VANADAT,SZAMERT)
/* Előkészítés */
MAXH ← LENGTH(STR(TOL))
if LENGTH(STR(IG))>MAXH
       MAXH ← LENGTH(STR(IG))
HOSSZ ← LENGTH(SZAMSZOV)
SZAMSZOV ← JOBBTOLT(SZAMSZOV,MAXH)
INVERZIR
KIIRAS(SZAMSZOV,OSZL,SOR,HOSSZ)
/* Beolvasás */
repeat
       /* Formailag helyes adat bekérése */
       repeat
              /* Jó jelek beállítása */
              JOJEL ← [KILEP]
              if HOSSZ>0
                     JOJEL ← JOJEL+[TOROL]
              if (HOSSZ=0) AND (TOL<0)
                     JOJEL ← JOJEL+['-']
              if HOSSZ<MAXH
                     JOJEL ← JOJEL+SZAMJEGYEK
              if (HOSSZ>0) AND (SZAMSZOV[HOSSZ] IN SZAMJEGYEK)
                     JOJEL ← JOJEL+[ADATVEG]
              /* Jel beolvasása */
              JEL ← JELBE(JOJEL)
              /* Jel feldolgozása */
              if JEL IN SZAMJEGYEK+['-']
                     HOSSZ ← HOSSZ+1
                     SZAMSZOV[HOSSZ] ← JEL
              else if JEL=TOROL
                     SZAMSZOV[HOSSZ] ← ' '
                     HOSSZ ← HOSSZ-1
              /* Kiírás */
              KIIRAS(SZAMSZOV,OSZL,SOR,HOSSZ)
       until JEL IN [KILEP,ADATVEG]
       /* Tartalmi ellenőrzés */
       if JEL<>KILEP
              X ← VAL(COPY(SZAMSZOV,1,HOSSZ))
              JOADAT ← (X>=TOL) AND (X<=IG)
until (JEL=KILEP) OR JOADAT
/* Befejezés */
NORMALIR
if JEL=KILEP
       KIIRAS(JOBBTOLT("",MAXH),OSZL,SOR,MAXH)
       SZAMSZOV ← ""
       VANADAT ← hamis
```

```
else
        KIIRAS(SZAMSZOV,OSZL,SOR,HOSSZ)
        /* A SZAMSZOV hosszbeállítása HOSSZ-ra... */
        SZAMERT ← X
        VANADAT ← igaz
```

A befejezés részben található SZAMSZOV hosszbeállítása a C-ben a '\0' végjel hozzáfűzését, a Pascal-ban a hosszbájt (SZAMSZOV[0]) megfelelő (CHR(HOSSZ)-ra történő) beállítását jelenti.

11.5. Rekurzív algoritmusok

Azokat az algoritmusokat nevezzük rekurzívnak, amelyek kifejtésében, definiálásában maga a definiálandó algoritmus is szerepel.

Gyakorlatilag az ilyen algoritmusok csak szubrutinokkal (lásd 10.) definiálhatók, hiszen csak ők képesek saját magukat közvetlenül, vagy más szubrutinokon keresztül közvetve (lásd 11.5.3.) meghívni.

Egy rekurzív szubrutin működési módja az, hogy a kapott paraméterektől függően – esetleg bizonyos tevékenységek végrehajtása után – befejezi a működését, vagy új paramétereket állít elő, és ezekkel meghívja rekurzívan önmagát.

Az előbbi eset, a működés befejezése jelentheti a feladat megoldásának végét (a szubrutin kezdőhívásából lépünk ki), ill. egy olyan pontra való visszatérést (a rekurzív hívást követő pontra), ahonnan a megoldás során a szubrutint önmagából meghívtuk, azaz egy „előző szintre" való visszatérést, ahol a megoldás folytatódik.

Rekurzív híváskor a szubrutin paraméterei és lokális munkaváltozói által meghatározott adatcsoport egy újabb „példánya" kerül a verembe, míg a szubrutinból való kilépéskor – a veremmutató automatikus átállásával – egy ilyen példány „kerül ki" a veremből.

Noha bizonyítottan minden rekurzív algoritmus megvalósítható rekurzió nélkül is, ráadásul sokszor kevesebb memória és idő felhasználásával, a rekurzióval többnyire rövidebb, áttekinthetőbb, így a lényeget jobban kifejező megoldást adhatunk.

Megjegyzés: A rosszul alkalmazott, programozott rekurzív szubrutinok könnyen megtölthetik a vermet, amelynek „veremtúlcsordulási" futási hiba (és rendszerint programleállás) a következménye.

11.5.1. Faktoriális

Feladat: Határozzuk meg egy pozitív egész szám faktoriálisát!

Megoldás: Az *n!=1*2*3…*(n–1)*n* képlet szinte „magától adja” a rekurziót, ill. annak használatát, hiszen *n!=(n–1)!*n*. Ha tehát ki tudjuk számolni *(n–1)!* értékét, akkor abból *n!* értéke egy szorzással megkapható. A rekurzió végét, a matematikai definíció alapján, a 0 értékre definiáljuk, miszerint 0!=1.

Funkció	Azonosító	Típus	Jelleg
A szám, amelynek a faktoriálisát számoljuk	N	Egész	I
Az eredmény	ER	Egész	O

```
/* Faktoriális kiszámítása rekurzív függvénnyel */
FAKT(N)
if N=0
       ER ← 1
else
       ER ← FAKT(N-1)*N
return ER
```

Az egymást követő rekurzív hívások adatai a verembe kerülnek (amíg *n* értéke 0-ra nem csökken), majd amikor az egyes hívások visszaadják az eredményt, akkor történnek meg valójában a szorzások. A rekurzív hívás eredményének és a veremben lévő, az adott híváshoz tartozó paraméternek a szorzataként áll elő az adott hívás eredménye.

11.5.2. Gyorsrendezés

A gyorsrendezés egy, az „oszd meg és uralkodj” elven alapuló algoritmus. Ezek az algoritmusok a feldolgozandó adatokat több, kisebb részre osztják, majd ezeket feldolgozva (rajtuk uralkodva) állítják elő a feladat megoldását.

Az ilyen algoritmusokat többnyire rekurzív szubrutinokkal valósítják meg, amelyek a kisebb adatcsoportok feldolgozására saját magukat hívják meg rekurzívan.

A gyorsrendezés rekurzív algoritmusának paramétere a rendezendő adatokat tartalmazó egydimenziós tömb és a feldolgozandó rész kezdő- és végindexe.

Az eljárás lépései:

1. Ha a feldolgozandó rész nem tartalmaz legalább két elemet, akkor készen vagyunk, hiszen az adott rész már rendezett, különben folytassuk a 2. ponttal.
2. Elemcserékkel válasszuk szét az elemeket két részre úgy, hogy az első rész összes eleme legyen kisebb vagy egyenlő, mint a másik rész összes eleme.
3. Rendezzük az első részt, azaz hívjuk meg az algoritmust az első részre.
4. Rendezzük a második részt, azaz hívjuk meg az algoritmust a második részre.

A 2. pontban található szétválasztást az alábbiak szerint végezzük:

- Válasszunk egy középértéket (más néven strázsa elemet), legyen ez a fizikailag (index szerint) középső elem értéke.
- A tömbben elölről haladva keressük meg az első olyan elemet, amelyik nem kisebb, mint a középérték.
- A tömbben hátulról haladva keressük meg az első olyan elemet, amelyik nem nagyobb, mint a középérték.
- A két elemet cseréljük fel.
- A kereséseket és cseréket a cserélt elemektől az eredeti irányokban haladva mindaddig ismételjük, amíg a két oldal nem találkozik.

A találkozási pont két részre osztja a tömböt, elöl a középértéknél nem nagyobb, hátul a középértéknél nem kisebb értékű elemek találhatók.

7	3	8	5	1	6	4	2
2	3	8	5	1	6	4	7
2	3	4	5	1	6	8	7
2	3	4	1	5	6	8	7

11.9. ábra. A szétválasztás elemcseréi.

A fenti ábrán a középérték 5, a szétválasztás eredményeként most mindkét rész négy elemet tartalmaz ({2, 3, 4, 1} és {5, 6, 8, 7}). Ez sajnos nem mindig sikerül ilyen szerencsésen. Ha például a középérték éppen az adott rész legkisebb vagy legnagyobb eleme, akkor a szétválasztás után az egyik részbe csupán egy elem (a középérték) kerül.

A példából az is látható, hogy a gyorsrendezés során az egyes elemcserék után az elemek nem feltétlenül a végleges helyükre kerülnek (mint pl. a kiválasztásos rendezésnél). A buborékrendezésnél, ahol szomszédos elemeket cserélgetünk, lassan haladnak az elemek a végső helyük felé, míg itt, kezdetben nagy, majd egyre kisebb „ugrásokkal". Ez a „haladás" akkor optimális, ha szétválasztáskor feleződnek az egyes tömbrészek.

Ez a felezés nem garantált, a módszerre vonatkozó elméleti és gyakorlati statisztikai vizsgálatok szerint [Wir 82], ez az egyik legjobb tömbrendező eljárás. Noha az algoritmus hatékonysága legrosszabb esetben (rossz bemenet esetén) n^2 nagyságrendű (ahol n az elemek számát jelöli), átlagosan azonban $n\log_2 n$ nagyságrendű műveletigénnyel jellemezhető [Cor 97].

Típus

```
ELEM  Egész                        /* A rendezendő elemek típusa */
TOMB  Egydimenziós ELEM tömb       /* Ez elemeket tároló tömb típusa */
```

Funkció	Azonosító	Típus	Jelleg
A rendezendő elemek	A	TOMB	I, M, O
A rendezendő rész kezdete	K	Egész	I
A rendezendő rész vége	V	Egész	I
Az elölről induló pásztázó index	I	Egész	M
A hátulról induló pásztázó index	J	Egész	M
A strázsa elem	S	ELEM	M
Két elem cseréjéhez	CS	ELEM	M

```
/* Gyorsrendezés rekurzívan */
GYORSREND(A,K,V)

if K<V
       /* Van legalább két elem */
       I ← K
       J ← V
       S ← A[(I+J) DIV 2]
       /* Szétválogatás a strázsa (S) elemhez képest */
       while I<=J
              while A[I]<S
                     I ← I+1
              while A[J]>S
                     J ← J-1
              if I<=J
                     /* Az I. és J. elemek cseréje */
                     CS ← A[I]
                     A[I] ← A[J]
```

```
                A[J] ← CS
                I ← I+1
                J ← J-1
/* Az első rész rendezése rekurzív hívással */
GYORSREND(A,K,J)
/* A második rész rendezése rekurzív hívással */
GYORSREND(A,I,V)
```

A rendezést elvégző rekurzív szubrutin kezdőhívásakor a rendezendő elemeket tartalmazó tömb mellett az első és utolsó elem indexét kell paraméterként átadni.

Megjegyzés: Az I=J esetben történő felesleges elemcsere egy újabb feltétellel (I<J) kivédhető, de ekkor a feltétel kiértékelése kerül „plusz” időbe.

11.5.3. Kínai gyűrűk

A rekurzió nemcsak úgy érvényesülhet, hogy egy szubrutin rekurzívan meghívja önmagát, hanem úgy is, hogy több szubrutin hívja egymást, azaz a rekurzió egy másik szubrutinon keresztül, közvetve érvényesül. Erre mutatunk most példát.

A kínai gyűrűk játékban adott számú gyűrű van felfüggesztve egymás mellett elhelyezkedő rudakon. Minden gyűrűnek kétféle állapota van: vagy fent van a gyűrű, vagy lent. Kezdetben valamennyi gyűrű fent van, ezeket kell levenni (mindet), az alábbi szabályok betartásával:

- Egyszerre csak egy gyűrű mozgatható.
- Az 1. gyűrű szabadon mozgatható.
- Az *N*. gyűrű (*N*>1) mozgatásához:
  - az *N–1*. gyűrűnek fent kell lennie;
  - az *N–2*,…, 1. gyűrűknek lent kell lennie.

Feladat: Oldjuk meg a kínai gyűrűk játékot, azaz írjuk ki, a kezdetben fent lévő összes gyűrű levételéhez szükséges gyűrűmozgatásokat!

Megoldás: A szabályokból látszik, hogy egy gyűrű mozgatását a mögötte lévő gyűrűk helyzete nem befolyásolja, ezért először a leghátsó gyűrűt vesszük le, aztán az utolsó előttit, legvégül az elsőt. Természetesen egy adott gyűrű mozgatásához biztosítanunk kell a megelőző gyűrűk megfelelő állapotát. Három szubrutint készítünk, az egyik (FEL) feltesz, a másik

(LE) levesz egy adott gyűrűt, míg a harmadik (BALRALE) egy adott gyűrűtől kezdve őt is, meg az összes megelőző gyűrűt leveszi.

A gyűrűk állapotának tárolására egy egydimenziós egész tömböt használunk, amelyet olyan (ún. globális) változóban tárolunk, amelyet a szubrutinok elérhetnek, így a szubrutinok paramétereként csak az adott gyűrű sorszámát adjuk át. Az *i.* gyűrű helyzetét a tömb *i.* eleme fejezi ki: 1 esetén fent van 0 esetén lent.

A megteendő mozgatási lépések kiírását egy szubrutin segítségével végezzük (KIIR), amely egyszerűen megjeleníti a gyűrűk állapotát, amiből leolvasható az éppen elvégzett mozgatási lépés.

Megjegyzés

- A rekurziót megengedő nyelvek olyan eszközöket is biztosítanak, amellyel az egymást hívó szubrutinok deklarálása megoldható, most egyszerűen egymás után megadjuk a három szubrutint.
- A feladat megoldásához kezdetben az összes gyűrűt fel kell tennünk (a tömb elemeit 1-re állítjuk, majd meghívjuk a BALRALE szubrutint a gyűrűk számával, mint paraméterrel.

Feladat: Vegyünk le egy adott gyűrűt!

Megoldás: Ha a gyűrű fent van, akkor biztosítjuk a mozgatásához szükséges állapotot, majd „levesszük” az adott gyűrűt és megjelenítjük a gyűrűk aktuális állapotát (KIIR).

Funkció	Azonosító	Típus	Jelleg
A gyűrűk aktuális állapota	GYURUK	Egydimenziós egész tömb	I, O
A gyűrű sorszáma	N	Egész	I

```
/* Az N. gyűrű levétele */
LE(N)

if GYURUK[N]=1
       if N>1
              FEL(N-1)
              if N>2
                     BALRALE(N-2)
       GYURUK[N] ← 0
       KIIR
```

Feladat: Tegyünk fel egy adott gyűrűt!

Megoldás: Az előbbiek alapján, analóg módon elvégezhető.

Funkció	Azonosító	Típus	Jelleg
A gyűrűk aktuális állapota	GYURUK	Egydimenziós egész tömb	I, O
A gyűrű sorszáma	N	Egész	I

```
/* Az N. gyűrű feltétele */
FEL(N)

if GYURUK[N]=0
       if N>1
              FEL(N-1)
              if N>2
                     BALRALE(N-2)
       GYURUK[N] ← 1
       KIIR
```

Feladat: Vegyünk le egy adott gyűrűt, és az azt megelőző összes gyűrűt!

Megoldás: Egy ciklus (amely az adott gyűrűtől kezdve az első gyűrűig halad) és a LE eljárás segítségével könnyen megoldható a feladat.

Funkció	Azonosító	Típus	Jelleg
A gyűrűk aktuális állapota	GYURUK	Egydimenziós egész tömb	I, O
A gyűrű sorszáma	N	Egész	I
Az aktuális gyűrű	I	Egész	M

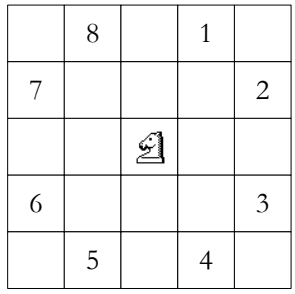
```
/* Az N. és a megelőző gyűrűk levétele */
BALRALE(N)

for I ← N,1,-1
       LE(I)
```

11.6. Visszalépéses algoritmusok

Vannak olyan feladatok, amelyek megoldásai nem állíthatók elő közvetlenül, direkt módon, ahol az egyes megoldások egy olyan „nagyszámosságú esethalmaz” egyes elemei, amelyben az esetek egyenkénti generálása és megvizsgálása gyakorlatilag még a leggyorsabb számítógépeken is lehetetlen lenne. A megoldásokat ilyenkor megpróbáljuk szisztematikusan, valamilyen módszer szerint úgy előállítani, hogy közben lehetőleg minél több eset (amelyek egyébként nem lehetnek megoldások) megvizsgálását elhagyjuk.

Tekintsük példaként azt a feladatot, amikor egy sakktáblát, egy adott kezdőmezőről indulva úgy kell bejárni egy huszárral, hogy minden mezőre pontosan egyszer lépünk.

Mivel egy mezőről elvileg 8 lehetséges irányba léphetünk (lásd 11.10 ábra), ezért az esethalmazunk 8^{63} esetet tartalmaz, hiszen a kezdőmező adott, azaz 63 lépést kell tennünk a sakktábla bejárásához. (Ez a szám önmagáért beszél, de megjegyezzük, hogy egy évben kevesebb, mint 10^8 másodperc van, ha ezred másodpercenként 1 milliárd esetet vizsgálnánk, ami azért nem lenne rossz teljesítmény, akkor 10^{20} esetet vizsgálnánk meg egy év alatt, azaz kb. 10^{30} évnyi munka után lennénk kész a 8^{63} db esettel.)



11.10. ábra. Huszár lehetséges lépései.

Az egyes esetek egy olyan 63 szintes fa (lásd 15.3.) leveleiként is elképzelhetők, amelyben minden pontnak 8 gyermeke van. A fa egyes szintjei az egyes lépésekhez tartoznak, a 0. szinten a kezdőállapot, az 1. szinten az onnan elérhető nyolc mező, a 2. szinten az 1. szint mezőiből elérhető mezők találhatók, és így tovább.

Ebben a fában kell utat keresni a legalsó szintig úgy, hogy az egyes szinteken megtett lépések mind szabályosak legyenek. Gondoljunk csak arra, hogy ha pl. a tábla sarkából indulunk (lásd 11.11. ábra), akkor csak két helyes lépésünk van, azaz 6 irány részfáit nem kell bejárnunk ill. megvizsgálnunk, hiszen nem vezethetnek megoldáshoz.

1	20	17	12	3
16	11	2	7	18
21	24	19	4	13
10	15	6	23	8
25	22	9	14	5

11.11. ábra. Huszár útja egy 5×5-ös táblán.

A módszeres próbálgatás most tehát azt jelenti, hogy a kezdőmezőről kiindulva, minden egyes lépésben megpróbáljuk folytatni az utunkat, mind a 8 lehetséges irányba. Ha egy adott irányba nem lehet lépni (mert lelépünk a tábláról, vagy ott már jártunk korábban), akkor (ezt a részfát kihagyva) vesszük a következő lehetséges irányt. Ha minden irányba léptünk már, akkor szabaddá téve ezt a mezőt, visszalépünk az előző lépéshez, és ott próbáljuk meglépni a soron következő lépést.

A visszalépés és a lépések hasonlósága miatt a megoldásra egy egyszerűsített rekurzív megoldási sémát adunk, ahol megkülönböztetjük azt a két esetet, amikor az összes megoldást elő kell állítanunk és azt, amikor elegendő egyetlen megoldást megtalálnunk.

Összes megoldás megkeresése:

```
Próbáljunk új választást
for Az összes választáson
      Válasszuk ki az adott választást
      if Megfelelő
            Jegyezzük fel
            if Megoldás nem teljes
                  /* Rekurzív hívás */
                  Próbáljunk új választást
            else
                  Megoldás kiírása
            A feljegyzés törlése
```

Ez a megoldás bejárja a fát, csak a „zsákutcákat” hagyja ki, amelyek nem vezetnek megoldáshoz (lásd a *Megfelelő* vizsgálatot). Ha megelégedünk egy megoldás megtalálásával, akkor biztosítanunk kell a rekurzióból való kilépést, mihelyt egy megoldást megtaláltunk (lásd *Van megoldás*). Az egyes rekurzív szinteken nem próbáljuk végig az összes lehetséges esetet (lásd az

előző megoldás *for* ciklusát), hanem egy hátultesztelő ciklussal vizsgáljuk az egyes választási lehetőségeket, és kilépünk a ciklusból, amint találtunk megoldást. A ciklus kezdőértékeinek beállítása történik meg *A választás előkészítése* részben. Ha a megoldás megtalálását jelző változót (*Van megoldás*) paraméterként kezeljük, akkor a *hamis* kezdőértékét is itt kell beállítanunk, egyébként, globális változó esetén az első hívás előtt.

Egy megoldás megkeresése:

```
Próbáljunk új választást
A választás előkészítése
repeat
        Válasszuk ki a következő választást
        if Megfelelő
                Jegyezzük fel
                if Megoldás nem teljes
                        /* Rekurzív hívás */
                        Próbáljunk új választást
                        if NOT (Van Megoldás)
                                A feljegyzés törlése
                else
                        Van megoldás ← igaz
until (Van megoldás) OR (Nincs több választás)
```

Mindkét módszert bemutatjuk egy-egy konkrét feladaton, az egy megoldás megkeresését a huszár útja példával, míg az összes megoldás megkeresését a klasszikusnak számító 8 királynő problémával szemléltetjük, amelyet Gauss vetett fel először 1850-ben.

11.6.1. Huszár útja a sakktáblán

Feladat: Járjunk be egy NxN-es méretű „sakktáblát”, egy adott kezdőmezőről indulva úgy, hogy minden mezőre pontosan egyszer lépünk!

Megoldás: A megoldásból készített program könnyebb és érdekesebb tesztelhetősége érdekében a tábla méretét input adatnak tekintjük (nem fixen 8-nak). A sakktáblát egy NxN-es mátrix segítségével kezeljük, ahol a megtett lépések sorszámait tároljuk (1-től N^2-ig), egy mező szabad voltát a 0 érték jelzi. A huszár aktuális helyzetét egy rekord segítségével írjuk le, amely a táblán elfoglalt sor és oszlop értékeket tartalmazza.

Típus
MEZO Rekord
        S, O Egész

A lehetséges 8 irányban történő lépések (lásd 11.10. ábra) könnyebb adminisztrálása végett, az egyes lépésekhez tartozó relatív elmozdulások értékeit (sor ill. oszlopszámban értve) az alábbi tömbkonstansok segítségével definiáljuk.

Konstans

LEPS=(–2, –1, 1, 2, 2, 1, –1, –2)

LEPO=(1, 2, 2, 1, –1, –2, –2, –1)

A megoldás teljességének vizsgálatához nyilvántartjuk az aktuális lépés sorszámát, így ha az eléri az N^2 értékét, akkor sikerült bejárnunk a táblát, találtunk egy megoldást.

Noha a tábla méretét inputként kezeljük, a maximális értékét korlátoznunk kell a TABLA deklarálása végett. Ha MAXMERET jelöli a maximális méretet, akkor a TABLA tömb MAXMERET*MAXMERET elemű.

Funkció	Azonosító	Típus	Jelleg
A sakktábla mérete	N	Egész	I
A sakktábla	TABLA	Kétdimenziós egész tömb	I, M, O
Az aktuális lépés sorszáma	LEPES	Egész	I
Az aktuális mező	AKT	MEZO	I
Van-e megoldás	VANMEGO	Logikai	M, O
Az aktuális lépés iránya	IRANY	Egész	M
A következő lépés mezője	KOV	MEZO	M

A rekurzív szubrutinok is programozhatók úgy, hogy a munkájukhoz szükséges input és output adataikat a paramétereiken keresztül kapják és adják. Mivel a szubrutin önmagát több mélységben is meghívhatja, előfordulhat, hogy ismétlődő adatok kerülnek a verembe, sőt, ha nem megfelelően adjuk át paramétereket (pl. nagy mennyiségű adatokat érték szerint adunk át és nem cím szerint), akkor a verem hamar meg is telhet.

Megoldásunkban most csak azokat az adatokat szerepeltetjük a szubrutin paraméterlistáján, amelyek az egyes hívásokhoz kötődnek (LEPES, AKT), míg azokat, amelyek a hívások során nem változnak (N) vagy amelyeket minden hívási szinten kezelni kell (TABLA, VANMEGO), azokat nem paraméterként, hanem globális változóként kezeljük. Ezzel áttekinthetőbb, egyszerűbb paraméterezést kapunk, ráadásul gyorsabb végrehajtást, hiszen a verembe nem kerülnek be ismétlődő adatok, de hangsúlyoz-

zuk, hogy ez a fajta információcsere hívó és hívott programrészek között általában – azaz nem rekurzív algoritmusok esetén – kerülendő.

```
/* A huszár következő lépése */
PROBAL(LEPES,AKT)

/* A választás előkészítése */
IRANY ← 1
repeat
       /* Válasszuk ki a következő választást */
       KOV.S ← AKT.S+LEPS[IRANY]
       KOV.O ← AKT.O+LEPO[IRANY]
       /* Megfelelő? */
       if (KOV.S>=1) AND (KOV.S<=N) AND
          (KOV.O>=1) AND (KOV.O<=N) AND
          (TABLA[KOV.S,KOV.O]=0)
              /* Jegyezzük fel */
              TABLA[KOV.S,KOV.O] ← LEPES
              /* A megoldás nem teljes? */
              if LEPES<N*N
                     /* Rekurzív hívás */
                     PROBAL(LEPES+1,KOV)
                     if NOT VANMEGO
                            /* A feljegyzés törlése */
                            TABLA[KOV.S,KOV.O] ← 0
              else
                     VANMEGO ← igaz
       IRANY ← IRANY+1
until VANMEGO OR (IRANY>8)
```

A szubrutin kezdőhívása előtt be kell állítanunk a kezdőmezőt, a táblába ide 1-t, a többi helyre 0-t kell tennünk, és a VANMEGO értékét *hamisra* kell állítanunk. A kezdőhívás 2 lépésszámmal végzendő, az eredményt a VANMEGO ill. a TABLA változók tartalmazzák.

11.6.2. Nyolc királynő

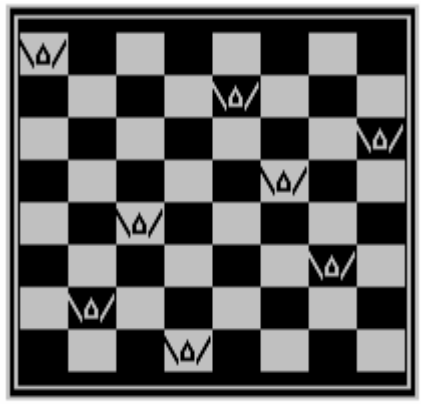
Feladat: Helyezzünk el egy sakktáblán 8 db királynőt úgy, hogy azok ne üssék egymást, azaz ne essenek egymás ütésvonalába!

Megoldás: Az összes megoldást megadó séma alapján (lásd 11.6.) haladunk soronként, azaz először az első sorba helyezünk el egy királynőt, aztán a másodikba és így tovább. Egy soron belül csak olyan mezőkbe (oszlopokba) tehetjük le a királynőt, ahol nem esik a táblán (a megelőző sorokban) lévő királynők ütésvonalába. Ehhez az oszlopot és a két átlót

kell megvizsgálnunk, a sort azért nem, mert minden sorba csak egy királynőt helyezünk el.

Vegyük észre, hogy ha a táblát mátrixként képzeljük el és az egyes mezőket (hasonlóan az előző feladathoz) a sorok és oszlopok sorszámával azonosítjuk, akkor a mátrix főátlójával párhuzamos átlókra (nevezzük őket most egyszerűen főátlóknak) az igaz, hogy a sor- és oszlopindexek különbsége azonos, míg a mellékátlóval párhuzamos átlók (nevezzük őket mellékátlóknak) esetén az indexek összege azonos. Egy 8*8-as sakktáblán 15-15 darab átló található, így ezek foglaltsága két 15 elemű, míg az oszlopok foglaltsága egy 8 elemű logikai tömbbel adminisztrálható.

A királynők helyzetét egy 8 elemű egész tömbben tároljuk, ahol az *i.* helyen az *i.* sorban lévő királynő oszlopának sorszáma található. Pl. az alábbi ábra megoldását a (1, 5, 8, 6, 3, 7, 2, 4) tömb írja le.



11.12. ábra. A 8 királynő probléma egy megoldása.

A tábla méretét, az előző feladathoz hasonlóan, most is inputként kezeljük, de maximális értékét korlátozzuk az adminisztráló tömbök deklarálása végett. Ha MAXMERET jelöli a maximális méretet, akkor a HOL és OSZL tömbök MAXMERET eleműek, az FATL, MATL tömbök 2*MAXMERET–1 eleműek.

Funkció	Azonosító	Típus	Jelleg
A sakktábla mérete	N	Egész	I
A királynők helye	HOL	Egydimenziós egész tömb	I, M, O
Az aktuális sor sorszáma	S	Egész	I
A főátlók foglaltsága	FATL	Egydimenziós logikai tömb	I, M, O
A mellékátlók foglaltsága	MATL	Egydimenziós logikai tömb	I, M, O
Az oszlopok foglaltsága	OSZL	Egydimenziós logikai tömb	I, M, O
Az aktuális oszlop	O	Egész	M

A rekurzív szubrutin paraméterezését hasonlóan végezzük, mint az előző feladatban, azaz csak az aktuális híváshoz kötődő sor értékét, azaz az éppen elhelyezni kívánt királynő sorszámát hagyjuk meg paraméterként, a tábla méretét, a királynők helyzetét és a foglaltságokat jelző tömböket globális változókkal kezeljük. A munkaváltozók most is (mint minden szubrutinnal megvalósított algoritmus esetén) lokálisak.

```
/* Egy királynő elhelyezése az S. sorba */
PROBAL(S)

/* Az összes választáson */
for O ← 1,N
      /* S. sorba az O. oszlopba téve az S. királynőt */
      /* Megfelelő? */
      if NOT OSZL[O] AND NOT FATL[S-O+N] AND NOT MATL[S+O-1]
            /* Jegyezzük fel */
            HOL[S] ← O
            OSZL[O] ← igaz
            FATL[S-O+N] ← igaz
            MATL[S+O-1] ← igaz
            /* A megoldás nem teljes? */
            if S<N
                  /* Rekurzív hívás */
                  PROBAL(S+1)
            else
                  MEGOKIIR
            /* A feljegyzés törlése */
            OSZL[O] ← hamis
            FATL[S-O+N] ← hamis
            MATL[S+O-1] ← hamis
```

A szubrutin kezdőhívása előtt a foglaltságokat jelző logikai tömbökbe (OSZL, FATL, MATL) *hamis* kezdőértékeket kell tennünk. A kezdőhívás 1-es paraméterrel végzendő, a megoldások kiírását a MEGOKIIR szubrutin végzi a HOL tömb alapján.

11.7. Feladatok

- Végezzünk futási idő összehasonlítást úgy, hogy a következő táblázat minden *f(n)* függvényére és *t* idejére határozzuk meg a probléma legnagyobb *n* méretét, amely még megoldható *t* idő alatt, feltételezve, hogy az algoritmus *f(n)* mikromásodperc alatt oldja meg a problémát.

	1 mp	1 perc	1 óra	1 nap	1 hónap	1 év	1 század
$\lg n$							
$\sqrt{n}$							
n							
$n\lg n$							
n^2							
n^3							
$2n$							
$n!$							

- Adottak bizonyos jelek és egy sztring. Készítsük el az adott jeleknek a sztringben való előfordulásra vonatkozó gyakorisági statisztikáját!
- Az előző feladat speciális eseteként készítsük el egy adott sztringben található számjegyek, magánhangzók, angol betűk gyakorisági statisztikáját!
- Adott egy sztring amelyben csak angol betűk és szóközök vannak. Tegyük fel, hogy kezdő és záró szóközök nincsenek, és a sztring belsejében sincsenek egymást követő szóközök. „Szó” kezdődik a szöveg elején és minden szóköz után. Készítsük el a szavak gyakorisági statisztikáját!
- Egy tömbben XX. századi évszámok vannak, készítsük el a gyakorisági statisztikát!
- Gyakorisági hisztogram: Adott egy számokat tartalmazó tömb és egy részintervallum darabszám *(r)*. A tömb értékeinek minimuma és maximuma által meghatározott intervallumot *r* egyenlő részre osztva határozzuk meg az egyes részintervallumokba eső elemek százalékos arányát a teljes elemszámhoz képest!
- Személyenként adottak életkor (egész év 0−100) és testsúly (0−250) adatok. Meghatározandók életkoronként és összesen az átlagsúlyok (egészre kerekítve)!

- Javítsuk a *buborékrendezés* algoritmusát úgy, hogy ha egy cseréket végző menet során nem történt csere, akkor fejezzük be a rendezést, hiszen már rendezett az adatsor!
- Módosítsuk a *buborékrendezés* algoritmusát úgy, hogy a cseréket végző menetek irányát váltogatjuk (pl. az első menetben hátravisszük a legnagyobb elemet, a másodikban előrehozzuk a legkisebbet, a harmadikban hátravisszük a második legnagyobbat és így tovább). A menetek a tömbnek csak azon részét vizsgálják meg, ahol még nem rendezettek az elemek. Ez a szakasz egyre szűkül, a menetek utolsó cseréinek helyei között található. A rendezés itt is befejezhető akkor, ha egy menet során már nem történt csere.
- Adottak sztringek. Egy sztring súlyán a benne található különböző karakterek darabszámát értjük (pl. ”alma” → 3). Határozzuk meg a sztringeknek a súly szerinti sorrendjét fizikai rendezéssel, ill. indextáblával!
- Egy tanulócsoportban ismerjük a neveket és a tanulmányi átlageredményeket. Állítsuk elő a név és a tanulmányi átlag szerinti rendezettséget leíró indextáblákat!
- Adott a síkon egy legalább két pontot tartalmazó pontrendszer a koordinátáival. Határozzuk meg:
  - A pontoknak az origótól vett távolság szerinti sorrendjét!
  - A pontoknak a pontrendszer súlypontjától vett távolsági sorrendjét!
  - A pontrendszer olyan sorrendjét, amelyben az első pont az origóhoz legközelebb lévő pont, a második az ehhez legközelebb eső (még szabad) pont, és így tovább!
- Adottak maximum négyjegyű, egész számok. Rendezzük a számokat növő sorrendbe, majd állítsuk elő a számok kettes, nyolcas és tizenhatos számrendszerbeli alakját! Oldjuk meg a feladatot úgy is, hogy a számok (számrendszerenként) azonos hosszúak legyenek, az előjel feltüntetésével, a leghosszabb szám hosszában előnullázva.
- Adott egy mátrix, amelynek elemei valós számok. Rendezzük át a mátrix sorait növekvő sorösszeg szerint!
- Adott egy mátrix, amelynek elemei angol betűk.
  - Rendezzük át a mátrix sorait úgy, hogy a soronkénti összeolvasással kapott sztringek ábécé szerint sorrendben legyenek!

    - Rendezzük át a mátrix oszlopait úgy, hogy egy adott sorban lévő karakterek ábécé szerint sorrendben legyenek!
- Készítsünk ellenőrző szubrutint az alábbi adatok helyességének ellenőrzésére, valamint egy halmazkonstrukciós szubrutint az adatok jelenként ellenőrzött inputtal való beolvasására, ill. módosítására:
    - Dátum *ÉÉÉÉ.HH.NN* (év, hónap, nap) alakban (hónap és nap előnullázva). (A Gergely-naptár szerint szökőév minden, néggyel osztható év, nem szökőév a kerek évszázad, de a 400-zal maradék nélkül oszthatók mégis azok. A február ekkor 29 napos, egyébként 28. Feltehető, hogy a dátum XX. vagy XXI századi.)
    - Időpont *OO.PP.SS* (óra, perc, másodperc) alakban (mindegyik előnullázva).
    - Szobaszám *ESXX* alakban, ahol
        - *E* (épületszárny): A, B, C, D, I lehet.
        - *S* (szint): az A és B szárnyban 1-6, a C és D szárnyban 1-7, az I szárnyban 1-5 lehet.
        - *XX* (sorszám): 1-12 (előnullázott).
    - Útazonosító (pl. M1, E75, 8265), amely:
        - Min. 1 és max. 6 jelből áll.
        - Számjegyeket tartalmaz, de az első jele lehet angol nagybetű is.
        - Nem kezdődhet nullával.
    - Fájlazonosítón értsünk egy *N.K* alakú sztringet, ahol *N* és *K* angol betűket és számjegyeket tartalmazhat, valamint
        - *N* névrész min. 1 és max. 8 jel hosszú.
        - *K* kiterjesztés max. 3 jel hosszú. A kiterjesztés a ponttal együtt el is maradhat, ha kiterjesztés nincs, pont sem lehet.
- Készítsünk egy halmazkonstrukciós, jelenként ellenőrző input szubrutint egy szöveg beolvasására, ill. módosítására az alábbiak figyelembevételével:
    - A szövegben megadható jelek (adatjelek) és az elválasztásra használható jelek (elválasztójelek) paraméterként megadhatók.
    - Az adat minimális és maximális hossza, valamint az elválasztójelek minimális és maximális száma paraméterként megadhatók.
    - Nem állhat elválasztójel a szöveg elején, végén, ill. nem állhat két elválasztójel egymás mellett.

- Készítsünk rekurzív függvényt a *Fibonacci sorozat n*-edik elemének meghatározására! A képzés módszere: az első két elem értéke 1, ezután minden következő az előző kettő összege.
- Készítsünk rekurzív függvényt egy szám hatványozására úgy, hogy a kitevő egy pozitív, egész szám!
  Pl. $2^8=2^4*2^4$, $2^4=2^2*2^2$, $2^2=2*2$, azaz most 3 db szorzással is kiszámolható az eredmény, a ciklussal megvalósított megoldás 7 db szorzása helyett.
- Készítsünk rekurzív szubrutint a *Hanoi tornyok* játék megoldására! A játékban *N* db, különböző méretű korong van egymásra rakva úgy, hogy legalul van a legnagyobb, rajta a második legnagyobb, és így tovább, legfelül a legkisebb. Át kell raknunk a korongokból álló tornyot egy másik helyre az alábbi szabályok betartásával:
  - Egyszerre csak egy korong mozgatható (egy torony tetejéről valamely más helyen lévő torony tetejére, vagy egy üres helyre).
  - Nem rakhatunk egy korongra nála nagyobbat.
  - A korongok rakosgatásához (a forrás- és a célhelyen kívül még) egy segédhelyet használhatunk.

  Pl. Ha a helyeket 1, 2, 3-mal jelöljük, és az 1. helyen áll egy 2 korongból álló torony, amelyet a 2. helyre szeretnénk átrakni, akkor a következő lépésekkel megoldanánk a feladatot: 1-3, 1-2, 3-1.
- Írjuk ki *N* db különböző elem összes permutációját, azaz az elemek összes lehetséges sorrendjét!
  Pl. Az 1, 2, 3 elemek összes lehetséges permutációja:
  1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1
- Készítsünk visszalépéses algoritmust *N* db bástya elhelyezésére egy *N***N*-es sakktáblán úgy, hogy azok ne üssék egymást! Állítsuk elő az összes lehetséges megoldást! Oldjuk meg a feladatot futókkal is!
- Készítsünk visszalépéses algoritmust a *stabil házasítás* feladatra. Adott *N* férfi és *N* nő, akiket egymás között ki kell házasítani. Minden személy rangsorolja az ellenkező nemű partnereket, azaz tudjuk, hogy milyen sorrendben szeretné őket házastársnak. Olyan párosítást nevezünk *stabilnak*, ahol nincs olyan férfi és nő, akik egymást kölcsönösen előrébb rangsorolták, mint a párosítás szerinti házastársukat. Egy párosítás minősíthető mindkét nem szempontjából egy számmal, amelyet a kapott partnereknek megfelelő, a rangsor szerinti sorszámok összegeként kaphatunk. Minél kisebbek ezek az összegek, annál jobb a pá-

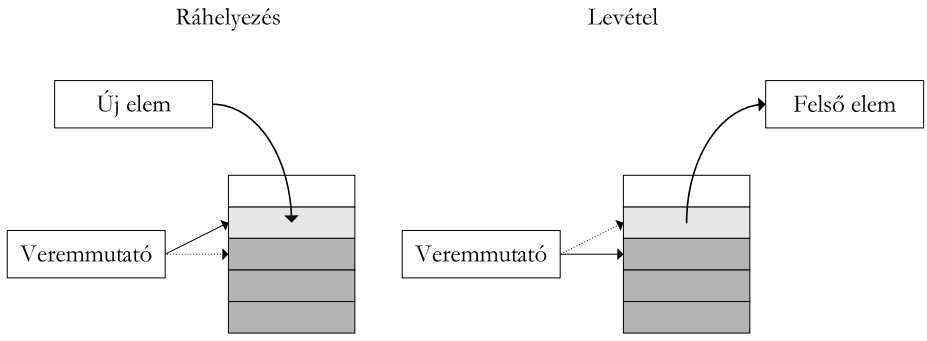
rosítás. Ha pl. mindkét érték N, akkor mindenki azt kapta, akit legjobban szeretett volna. Keressük meg az összes lehetséges megoldást, és írjuk ki a párosítást (nemek szerint) minősítő értékeket is! (Tipp: vegyük sorra a nőket vagy a férfiakat, majd a kiválasztott személy rangsora szerint haladva próbáljunk neki párt találni úgy, hogy a párosítás továbbra is stabil maradjon.)

12. Verem

12.1. Általános jellemzés

A verem adatstruktúra egy olyan összetett adatstruktúra, amelyet két művelet jellemez:

- Ráhelyezés, szakszóval *push* művelet: egy elemet teszünk a verembe, ez lesz a verem legfelső eleme.
- Levétel, szakszóval *pop* művelet: a verem legfelső elemét kivesszük, eltávolítjuk a veremből.



12.1. ábra. Veremműveletek.

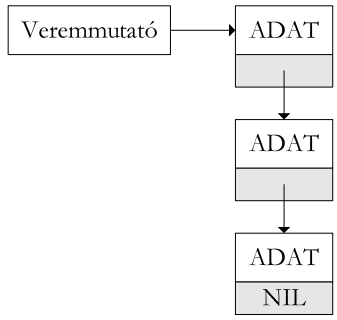
A verem és kezelése programozási nyelvtől függ. Az Assembly nyelvben pl. használhatók, de más nyelvekben többnyire implementálnunk, azaz a nyelv eszközeivel megvalósítanunk, programoznunk kell. Az adatok betételének és kivételének sorrendje fontos, az utoljára betett elemet vehetjük ki először *(last in first out),* ezért legegyszerűbb a vermet valamilyen egyszerű lineáris adatszerkezettel, például egydimenziós tömbbel, vagy egyirányban láncolt listával megvalósítani.

A tömbnél a verem tetején a tömb legnagyobb indexű (utolsó) eleme van, az aktuális elemszám, mint veremmutató (a verem tetejére mutató index) funkcionál.

- Ráhelyezésnél a tömb aktuális elemszáma eggyel nő, az új elem az utolsó (legfelső) elem lesz.
- Levételnél az utolsó elemet vesszük le, az aktuális elemszám eggyel csökken.

A listánál a verem tetején a lista kezdőeleme van, a kezdőmutató tölti be a veremmutató szerepét.

- Ráhelyezésnél az új elem a lista elejére kerül.
- Levételnél az első elemet vesszük le.



12.2. ábra. Listás verem.

12.2. Gyorsrendezés saját veremmel

Feladat: Rendezzünk egy tömbben lévő adatsort növekvő (nem csökkenő) sorrendbe a gyorsrendezés módszerével, saját veremkezeléssel!

Megoldás: A gyorsrendezés módszerét és rekurzív megvalósítását korábban (lásd 11.5.2.) részletesen tárgyaltuk, most erre építve egy, a rekurziót saját veremkezeléssel megvalósító, nem rekurzív megoldást mutatunk be.

Az ilyen jellegű feladatoknál tulajdonképpen azt kell tennünk, amit a rekurzív szubrutinok „láthatatlanul" megvalósítanak: biztosítanunk kell az egyes hívási szintek működésekor azok adatkörnyezetét, és vezérelnünk kell azok végrehajtását.

Az egyes hívásokhoz tartozó adatok tárolására egy saját vermet használunk. Ahhoz, hogy a vermet megfelelő méretűre deklaráljuk, tudnunk kell, hogy hány hívás adatai kerülhetnek be egy időben, „egymás fölé" a verembe.

A végrehajtás vezérlésére egy egyszerű iterációt használunk. Az iteráció mindaddig „dolgozik", amíg van a veremben adat, azaz van még „fel nem dolgozott" hívás. Az iteráció minden egyes lépésében kivesszük a verem tetején lévő (a legutolsó híváshoz tartozó) adatokat és feldolgozzuk őket. A feldolgozás lépéseiben követjük az eredeti, rekurzív szubrutin lépéseit. Ha rekurzív hívás történik, akkor ahhoz igazodóan új adatokat teszünk a

verembe. Mivel minden helyes rekurzióban van olyan eset is, amikor nem történik további rekurzív hívás, (azaz nem kerülnek be új adatok a verembe), ezért (hasonlóan az eredeti, rekurzív szubrutin „láthatatlan vermé-hez") az általunk használt verem is előbb-utóbb kiürül, így befejeződik a végrehajtás.

A megvalósítás „általános mikéntje" után kanyarodjunk vissza az eredeti feladat megoldásához.

A gyorsrendezés algoritmusa abban az esetben, ha a rendezendő rész már olyan „kicsi", hogy legfeljebb egy elemet tartalmaz, akkor valójában nem csinál semmit, hiszen az ilyen adatsor már rendezett. Ha viszont van legalább két elem a rendezendő részben, akkor egy strázsa elemet (a rendezendő rész középső elemét) kijelölve szétválasztja a rendezendő elemeket két részre (a strázsánál nem nagyobbakra és a strázsánál nem kisebbekre), majd két rekurzív hívással rendezi a szétválasztás során keletkező két részt. A két rekurzív hívás miatt tehát két hívás adatait kell a verembe tennünk. A megoldást vezérlő iteráció következő lépésében a verem tetején lévőt nyomban ki is vesszük és feldolgozzuk, ezért nem mindegy, hogy a két hívás adatait milyen sorrendben tesszük be a verembe. Ha ugyanis a „hosszabb" részt tesszük be előbb, akkor mindig a „rövidebb" részek feldolgozása történik meg előbb. A rövidebb részek minden esetben legfeljebb fele olyan „hosszúak", mint a szétválasztás előtti részek, így könynyen belátható, hogy legfeljebb $log_2 n$ (ahol n a rendezendő adatok számát jelöli) darab rész adatai kerülhetnek „egymás fölé" a verembe, következésképpen a vermet elegendő ekkorára deklarálni.

A rekurzív hívásokhoz tartozó adatkörnyezetet a hívási paraméterek, és a használt munkaváltozók aktuális értékeinek együttese jelenti, de csak azon változók értékeit kell a veremben tárolni, amelyeket a rekurzív hívást követően is használunk. A gyorsrendezés rekurzív hívásai után egyik változót sem használjuk, így csak a hívás paramétereit, a rendezendő elemeket tartalmazó tömbrész kezdő- és végindexét kell a veremben tárolnunk. A vermet egy egydimenziós tömbbel valósítjuk meg, a tömb egy eleme egy indexpár.

A veremkezelés alapműveleteit két eljárással (VEREMBE, VEREMBOL) végezzük.

```
Típus
   ELEM  Egész                          /* A rendezendő elemek típusa */
   TOMB  Egydimenziós ELEM tömb         /* Az elemeket tároló tömb típusa */
   VEREMELEM  Rekord
        K       Egész                   /* A rendezendő rész kezdete */
        V       Egész                   /* A rendezendő rész vége */
   TVEREM  Egydimenziós VEREMELEM tömb      /* A verem típusa */
```

A vermet reprezentáló TVEREM tömbtípus mérete a rendezendő elemeket leíró TOMB típus méretének kettesalapú logaritmusa.
Először a rendezéshez szükséges veremkezelő műveleteket valósítjuk meg.

Feladat: Tegyünk egy elemet a verembe!

Megoldás: A veremmutatót megnöveljük, a tárolandó értékeket a verem tetejére kerülő új elemben tároljuk.

Megjegyzés: A verem túlcsordulását nem kell ellenőriznünk, hiszen a rendezés során nem lépjük túl a verem méretét.

Funkció	Azonosító	Típus	Jelleg
A verem	VEREM	TVEREM	I, O
A veremmutató	VEREMMUT	Egész	I, O
A rendezendő rész kezdete	K	Egész	I
A rendezendő rész vége	V	Egész	I

```
/* Adat betétele a verembe */
VEREMBE(VEREM,VEREMMUT,K,V)

VEREMMUT ← VEREMMUT+1
VEREM[VEREMMUT].K ← K
VEREM[VEREMMUT].V ← V
```

Feladat: Vegyünk ki egy elemet a veremből!

Megoldás: A verem tetején lévő elemet (az abban tárolt értékeket) adjuk eredményül, majd a veremmutatót csökkentjük.

Megjegyzés: Az eljárást csak akkor hívjuk meg (ezt a hívó algoritmus, a rendezés biztosítja), ha a verem nem üres, azaz mindig elvégezhető a művelet.

Funkció	Azonosító	Típus	Jelleg
A verem	VEREM	TVEREM	I, O
A veremmutató	VEREMMUT	Egész	I, O
A rendezendő rész kezdete	K	Egész	O
A rendezendő rész vége	V	Egész	O

```
/* Adat kivétele a veremből */
VEREMBOL(VEREM,VEREMMUT,K,V)
```

```
K ← VEREM[VEREMMUT].K
V ← VEREM[VEREMMUT].V
VEREMMUT ← VEREMMUT-1
```

Feladat: Készítsük el a rendező eljárást!

Megoldás: A veremmutató inicializálása után a teljes rendezést leíró feladatot a verembe rakjuk, majd mindaddig amíg a verem ki nem ürül, a verem tetejéről levesszük a legfelső, éppen megoldandó részfeladatot és megoldjuk, azaz elvégezzük rajta a szétválasztást és az eredményként előálló két részt a verembe tesszük. Előbb a hosszabb részt tesszük a verembe, utána a rövidebbet, így a rövidebb részt vesszük ki előbb, azaz annak rendezésével folytatódik az algoritmus.

Funkció	Azonosító	Típus	Jelleg
A rendezendő elemek	A	TOMB	I, M, O
A rendezendő elemek száma	N	Egész	I
A verem	VEREM	TVEREM	M
A veremmutató	VEREMMUT	Egész	M
A rendezendő rész kezdete	K	Egész	M
A rendezendő rész vége	V	Egész	M
Az elölről induló pásztázó index	I	Egész	M
A hátulról induló pásztázó index	J	Egész	M
A strázsa elem	S	ELEM	M
Két elem cseréjéhez	CS	ELEM	M

```
/* Gyorsrendezés nem rekurzívan */
GYORSREND(A,N)
```

```
/* A verem inicializálása */
VEREMMUT ← 0
/* A teljes rendezendő részt a verembe */
VEREMBE(VEREM,VEREMMUT,1,N)
```

```
while VEREMMUT>0
      /* A rendezendő rész kivétele a veremből */
      VEREMBOL(VEREM,VEREMMUT,K,V)
      if K<V
            /* Van legalább két elem */
            I ← K
            J ← V
            S ← A[(I+J) DIV 2]
            /* Szétválogatás a strázsa (S) elemhez képest */
            while I<=J
                  while A[I]<S
                        I ← I+1
                  while A[J]>S
                        J ← J-1
                  if I<=J
                        /* Az I. és J. elemek cseréje */
                        CS ← A[I]
                        A[I] ← A[J]
                        A[J] ← CS
                        I ← I+1
                        J ← J-1
            /* A két rész felvétele a verembe */
            if J-K>V-I
                  VEREMBE(VEREM,VEREMMUT,K,J)
                  VEREMBE(VEREM,VEREMMUT,I,V)
            else
                  VEREMBE(VEREM,VEREMMUT,I,V)
                  VEREMBE(VEREM,VEREMMUT,K,J)
```

12.3. Feladatok

Oldjuk meg a rekurzív szubrutinnal megvalósított (lásd 11.5.), ill. megvalósítandó (lásd 11.7.) feladatokat (pl. faktoriális, permutáció, *Hanoi tornyok*, sakkbábú elhelyezés, stb.) rekurzió alkalmazása nélkül, saját veremkezeléssel!

13. Dinamikus adatstruktúrák

Az eddig használt változóink *statikusak* voltak abban az értelemben, hogy helyigényük, méretük már a forrásprogram fordításakor (fordítási időben) eldől, és ezt a végrehajtás során (futási időben) már nem lehet megváltoztatni.

Az olyan jellegű feladatoknál, ahol az adatok száma előre nem ismert, nehezen becsülhető és korlátozható, vagy ha az adatok száma nagymértékben ingadozik, akkor az adatok tárolására használt statikus változók

- méretének a legrosszabb „esethez" kell igazodnia, következésképpen helypazarló adattárolást valósítunk meg;
- lehet, hogy már nem is deklarálhatók (mert pl. túllépnénk a DOS memóriacímzéséből fakadó 64Kb-os korlátot).

Szükségünk van tehát olyan eszközökre, amelyekkel az adatok tárolása, így azok helyigénye, a végrehajtás során rugalmasan, *dinamikusan* kezelhető, szabályozható.

13.1. Dinamikus tömbök

Azokat a tömböket, amelyek méretét futásidőben igény szerint, szabadon állíthatjuk, dinamikus tömböknek nevezzük. Ha a tömb méretét növeljük, akkor újabb tömbelemek „keletkeznek", míg ha csökkentjük a tömb méretét, akkor tömbelemek, s ezzel esetleg adatok „vesznek el".

Megjegyzés: Sajnos nem minden fejlesztőkörnyezet támogatja ezt a hatékony eszközt, így a példáink demonstrálására használt Turbo C, Turbo Pascal sem, de pl. megtalálhatók a C#, Delphi, Visual Basic fejlesztőkörnyezetekben.

13.2. Mutatók és dinamikus változók

13.2.1. Mutatók

A mutató típusú értékek valójában memóriacímek, azaz a központi memória egy adott helyét címzik, arra mutatnak.

Aszerint, hogy egy mutató „mire” mutat, azon a címen „mi” található a memóriában, megkülönböztetünk

- adatra mutató és
- szubrutinra (programkód) mutató

mutatót.

A továbbiakban csak adatmutatókkal foglalkozunk, de megjegyezzük, hogy az utóbbiak segítségével lehetővé válik a szubrutinok paraméterként való kezelése (pl. egy függvényt listázó szubrutinnak magát a listázandó függvényt is átadhatjuk paraméterként), így még általánosabb, még jobban „idomítható” szubrutinokat készíthetünk.

13.2.2. Adatmutatók

Adatmutatókon (továbbiakban csak mutatók) olyan mutatókat értünk, amelyek egy adott, *Típus* típusú adatra mutatnak. A *Típus* tetszőleges egyszerű, ill. összetett adattípus lehet. Tulajdonképpen egy adatmutató által megcímzett memóriaterületet úgy használhatunk, mint egy, az adott típushoz tartozó változót.
Konstans: NIL
Jelentése: nem mutat sehová.

Megjegyzés
- Nem egyenértékű a definiálatlan mutatóval.
- Tetszőleges típusú adatra mutató mutatóváltozónak értékül adható, annak értékével összehasonlítható.

Műveletek:

- Hasonlítások: Egyenlő (=), Nem egyenlő (<>)

A megengedett két hasonlítás azonos típusú adatra mutató mutatók között értelmezett, eredményül logikai értéket kapunk. Ha a két mutató (memóriacím) megegyezik, azaz a memóriának ugyanazon helyére mutatnak, akkor igazat, egyébként hamisat kapunk.

13.2.3. Mutatóváltozók

A mutatókat, pontosabban a mutató típusú értékeket, mutató típusú változókban tároljuk. A mutatóváltozók deklarálásánál (az adatszerkezeti táblázatban) meg kell adnunk azt, hogy milyen típusú adatra fog mutatni, azaz milyen adat címét fogja tárolni. Ezzel valójában a mutatóváltozó típusát adjuk meg.

Pl.

Funkció	Azonosító	Típus	Jelleg
Egy egész szám	A	Egész	M
Egy egész szám címe	B	Egészre mutató	M

Bevezetünk két műveletet, amelyeket változókkal, ill. a mutatóváltozókkal kapcsolatosan használhatunk.

Ha A egy *Típus* típusú változó, akkor az &A jelentse a változó címét, ami a változóban tárolt, *Típus* típusú adatra mutató mutatóként használható.

Ha B egy *Típus* típusú adatra mutató változó, akkor a *B jelentse a B által mutatott címen lévő, *Típus* típusú adatot tároló változót.

Pl. Az előzőleg deklarált A és B változók felhasználásával

```
A ← 2          /* A-ba 2-t teszünk */
B ← &A         /* B mutasson A-ra */
Ki: *B         /* 2-t ír ki */
*B ← *B+1      /* B által mutatott változó használata */
Ki: A          /* 3-t ír ki */
```

Megjegyzés

- Az & műveletet cím operátornak, a * műveletet indirekció operátornak is nevezzük, prioritásuk az egyoperandusú műveletekkel azonos erősségű.
- A Turbo Pascal nyelvben a címoperátornak a @, az indirekció operátornak a ^ jelek felelnek meg.
- Rekordokra és tömbökre mutató mutatók esetén, a rekord egy mezőjének, ill. a tömb egy elemének beazonosításakor zárójelet alkalmazunk. Pl. (*RM).MEZO, (*TM)[I], ahol RM egy rekordra, TM egy egydimenziós tömbre mutató mutatóváltozó. A zárójelekkel kifejezzük azt, hogy a mutatókkal először az összetett adatot „érjük el” (indirekció), és csak utána vesszük a megfelelő mezőt, ill. tömbelemet. Zárójel nélküli hivatkozással (*RM.MEZO, *TM[I]) épp „fordítva” történne, előbb a megfelelő mezőt, ill. tömbelemet érnénk el, csak utána az általuk mutatott adatot. A hivatkozások helyességéhez ekkor az RM.MEZO és TM[I] változóknak kell mutatókat tartalmazniuk, nem az RM és TM változóknak.

13.2.4. Dinamikus változók

A mutatóváltozók segítségével olyan, ún. dinamikus változókat kezelhetünk, amelyeket futási időben hozhatunk létre, ill. szüntethetünk meg, ha már nincs rájuk szükségünk.

Az ilyen célra fenntartott szabad memória (heap) mérete és kezelése programnyelvenként eltérő, de „működési elvük" hasonló, ezért bevezetünk néhány általánosan használható szubrutint.

Ha P egy *Típus* típusú adatra mutató változó, akkor a

HELYFOGLAL(P)

eljárás foglaljon akkora helyet a szabad memóriából, amely egy *Típus* típusú adat tárolásához kell, és ennek memóriabeli címe kerüljön a P változóba. P ezután egy definiálatlan kezdőértékű, de már használható, *Típus* típusú, a dinamikus tárban lévő, dinamikus változóra mutat, annak címét tartalmazza.

A megfelelő hely meglétének ellenőrzése a

VANHELY(Méret)

függvénnyel végezhető, amely *igaz* értéket ad, ha van még *Méret* bájt, összefüggő szabad hely a dinamikus tárban, egyébként *hamisat*.

Egy változó ill. típus mérete, helyigénye bájtokban a

MERET(Azonosító)

függvénnyel kérdezhető le.

Ha P egy *Típus* típusú adatra mutató változó, akkor a

FELSZABADIT(P)

eljárás szabadítsa fel a P által mutatott címen lévő, *Típus* típusú változó által elfoglalt helyet, azaz szüntesse meg a címen lévő dinamikus változót. P értéke ezután definiálatlan lesz.

Pl. Legyen A egy egész típusú adatra mutató változó, akkor

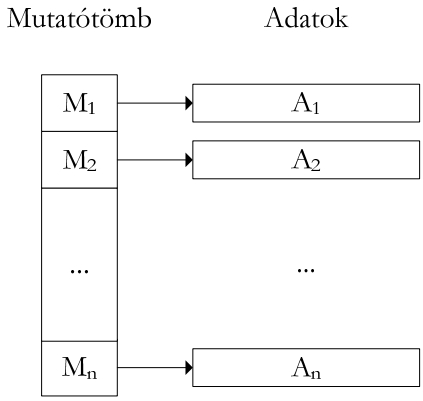
```
                    /*  *A még nem használható  */
HELYFOGLAL(A)       /*  Az A változó értéket kap  */
*A ← 2              /*  *A dinamikus változó használata  */
Ki: *A              /*  2-t ír ki  */
FELSZABADIT(A)      /*  *A dinamikus változó megszüntetése  */
                    /*  *A már nem használható  */
                    /*  Az A változó értéke definiálatlan  */
```

Megjegyzés

- A C nyelvben maga a helyfoglaló függvényhívás jelzi, hogy sikeres volt-e a helyfoglalás vagy sem, azaz mindig meghívható, míg a Pascal nyelvben, ha nincs elegendő hely, akkor futási hiba keletkezik helyfoglaláskor.
- Egy program befejezésekor, az általa elfoglalt memória (program és adatterület), így a lefoglalt dinamikus memória is felszabadul. Ha futás közben szabadítunk fel helyeket, akkor szabad részek, „lyukak" keletkezhetnek a dinamikus tárban (pl. ha egy korábban foglalt részt szabadítunk fel előbb), amelyeket a HELYFOGLAL szubrutin megpróbál újra „kitölteni".
- Ahogy más változóknak, úgy a mutatóváltozóknak sincs kezdőértékük, így a változók felhasználása az értékük definiálása előtt, nemcsak a programunk hibáját okozza, hanem esetleg „mást" is elronthat. Ha az előző példában az A változónak nem adunk értéket a HELYFOGLAL eljárással, akkor a *A ← 2 értékadás „valahová" beír a memóriába, ami ki tudja milyen „eredménnyel" jár. Éppen ezért a dinamikus tárkezelést tartalmazó programokat futtatásuk előtt ajánlatos menteni, mert a „normális befejezés", így az esetleges „mentés lehetősége" korántsem biztos és garantált.

13.3. Kollekciók

A *kollekció* a tömb olyan általánosítása, amelyben az adatok nem közvetlenül a tömbben találhatók, hanem a dinamikus tárban és az elérésüket biztosító mutatókat tároljuk a tömbben. Az adatokat a kollekció *tételeinek*, míg a rájuk mutató mutatókat tartalmazó tömböt *mutatótömbnek* nevezzük. Ezek együttese alkotja a kollekciót (lásd 13.1. ábra). Az egyes tételek – amelyek valójában dinamikus változók – a hozzájuk tartozó mutatók segítségével érhetők el. A mutatók, mivel tömbben tárolódnak, egyszerű indexeléssel hivatkozhatók.



13.1. ábra. Kollekció.

Az előző ábra egy egydimenziós mutatótömbbel megvalósított kollekciót szemléltet.

Megjegyzés: Noha használhatunk többdimenziós mutatótömböket is, a gyakorlatban ritkán van szükség rájuk, hiszen pl. ha egy egydimenziós mutatótömb esetén a tételeknek szintén egydimenziós tömböket választunk, akkor ezzel a kollekcióval valójában egy kétdimenziós tömböt valósítunk meg.

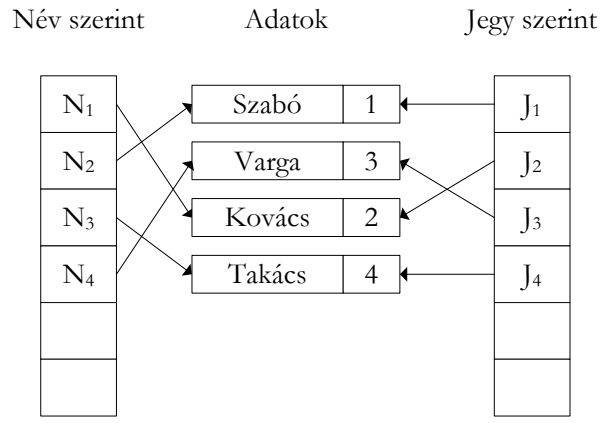
Mivel a mutatótömb egy tömb, ezért csak azonos típusú adatokat, azaz csak azonos típusú adatra mutató mutatókat tartalmazhat, következésképpen csak azonos típusú tételeket kezelhetünk.

Megjegyzés: Ha a fejlesztőkörnyezetünk megengedi az objektumok használatát, akkor a kollekció tételeinek szigorú típusegyezése kibővíthető. Egy adott objektumtípusból (osztályból) képzett kollekció esetén ugyanis a kollekció tételei nemcsak az adott osztály objektumai (példányai) lehetnek, hanem a belőle leszármazott osztályok objektumai is. Ekkor olyan eszközök is rendelkezésre állnak, amelyekkel az egyes objektumok típusa beazonosítható.

A kollekció adatstruktúra önmagában rendelkezik a logikai és fizikai sorrend elválasztásának, a minimális adatmozgatással megvalósítható sor-

rendcserének azzal a lehetőségével, amelyet a tömböknél az indextábla segítségével tudtunk megvalósítani. Két tétel sorrendjének felcseréléséhez nyilván elegendő csak a megfelelő mutatókat felcserélni.

Ugyanarra a tételsorra a mutatók különböző sorrendjével is hivatkozhatunk (mintha több indextáblát használnánk).



13.2. ábra. Azonos adatrészű kollekciók.

A kollekció-adatstruktúra alkalmas a dinamikus memória kihasználására és a nagy helyigényű, egyben már nem kezelhető adatok „szétdarabolására". Gondoljunk csak arra, hogy pl. egy 200*200-as, elemenként 4 bájtos (a valós számok tárolásához legalább ennyi bájt használatos) mátrix mérete meghaladja az egy változó méretét korlátozó 64Kbájtot, következésképpen a memóriában csak darabolva tárolható.

Alapvető karbantartási és lekérdezési műveletek algoritmusai a tömbökével megegyező, hiszen a kollekciót szerkezetileg a mutatótömb határozza meg, ami eltér, az egyrészt az elemekre való hivatkozás módja, másrészt a tételek, mint dinamikus változók helyfoglalásának és felszabadításának kezelése.

Mintaként, egy kicsit a fájlkezelést (lásd 14.) is bemutatandó, lottószelvényeket töltünk be és értékelünk ki. Az input és output adatokat a háttértáron szövegfájlokban, a memóriában egy kollekcióban tároljuk.

A kollekciót úgy deklaráljuk, hogy minél több tételt tudjon kezelni. Mivel a mutatók 4 bájtot foglalnak el (a DOS szegmens és offset címei 16 bitesek), így a maximális elemszám (a 64Kb-os korlát miatt) kb. 16 ezer.

Magát a mutatótömböt (az egyszerűség kedvéért) statikus változóban tároljuk, de a szabad memóriában, dinamikus változóként is kezelhető lenne, így a teljes kollekció statikus (adatszegmensbeli) helyfoglalása csupán 4 bájt lenne.

```
Konstans
   MAXSZELV   16000    /* A betölthető szelvények max. száma */
   DB         6        /* A lottószámok db száma egy szelvényen */
   MAX        45       /* Egy lottószám maximális értéke */
   MINTAL     3        /* A minimális találatszám a nyeréshez */

Típus
   SZELVENY  Egydimenziós egész tömb[DB]          /* Lottószelvény */
   TETEL  Rekord                                  /* A kollekció tétele */
      SZ        SZELVENY                          /* A számok */
      TAL       Egész                             /* Találatok száma a szelvényen */
   KOLLEKCIO  Egydimenziós TETEL-re mutató tömb[MAXSZELV]
                                                  /* A kollekció */
   STAT  Egydimenziós egész tömb                  /* Statisztika */
```

Egy statisztika (STAT típusú) tömbben gyűjtjük majd kiértékeléskor az egyes találatokhoz tartozó szelvények számát (az *i* találatos szelvények számát a tömb *i*. helyén tároljuk), így a tömböt kivételesen 0-tól indexeljük DB-ig.

Feladat: Töltsünk be adatokat egy szövegfájlból a memóriába kollekció segítségével! A fájl soronként egy szabályos lottószelvény számait tartalmazza, azaz ahány soros a fájl, annyi szelvény található a fájlban. Feltesszük, hogy a fájl tartalma helyes, de tartalmazhat annyi sort is, hogy teljes egészében nem fér be a dinamikus tárba.

Megoldás: Amíg a fájlban van adat, és van még hely a dinamikus tárban, ill. a kollekcióban, addig helyet foglalunk a szelvénynek, majd a szelvény számait beolvassuk a lefoglalt helyre. A művelet sikerességét a megoldásra készített függvény eredményeként adjuk vissza. Az adatokat tartalmazó fájl azonosítóját használjuk paraméterként, a fájlváltozó egy (a szubrutinban deklarálandó, lokális) munkaváltozó.

Funkció	Azonosító	Típus	Jelleg
A fájl azonosítója	FNEV	Sztring	I
Az eredmény kollekció	K	KOLLEKCIO	O
A kollekció elemszáma	KDB	Egész	O
Betöltöttük-e az összes adatot	OK	Logikai	O
A fájlváltozó	F	Szövegfájl	M
Segédváltozó	I	Egész	M

```
/* Az FNEV nevű szövegfájl szelvényeinek betöltése */
BETOLT(FNEV,K,KDB)

/* Adatbeolvasás */
NYIT(F,FNEV,"I")
KDB ← 0
while NOT FAJLVEGE(F) AND VANHELY(MERET(TETEL))
      AND (KDB<MAXSZELV)
       KDB ← KDB+1
       HELYFOGLAL(K[KDB])
       for I ← 1,DB
             Be F: (*K[KDB]).SZ[I]
OK ← FAJLVEGE(F)
ZAR(F)
return OK
```

Feladat: Értékeljünk ki egy, az előzőekben deklarált kollekcióban lévő szelvényeket, adott nyerőszámok alapján! A nyertes szelvényeket (amelyeken legalább MINTAL találat volt), írjuk ki egy szövegfájlba úgy, hogy először a telitalálatos szelvényeket, legvégül a MINTAL találatos szelvényeket listázzuk a darabszámuk feltüntetésével!

Megoldás: A könnyebb kiértékeléshez a nyerőszámokból egy halmazt készítünk, így egy szelvény egy számának kiértékelése (azaz annak eldöntése, hogy nyerőszám-e vagy sem) egyetlen feltétellel elvégezhető (kereső ciklus helyett), nevezetesen benne van-e a nyerőszámok halmazában vagy sem.

Az egyes találatokhoz (0, 1, …, DB) tartozó, ilyen találatos szelvények számát egy tömbben gyűjtjük, amelyet kezdetben lenullázunk és minden egyes szelvény kiértékelése után, valamelyik elemét eggyel megnöveljük.

Ha minden szelvényt kiértékeltünk, akkor elkészítjük az eredménylistát.

Megjegyzés

- Az egyes szelvények találatainak számát, magában a kollekcióban tároljuk (lásd TAL mező), így ezek a kiértékelés után felhasználhatók (lásd K output jellegét).
- A szövegfájlba írás soremeléseit az algoritmusban nem tüntettük fel, ezek a megfelelő helyeken, a megfelelő programnyelvi eszközökkel megtehetők (lásd függelék).

Funkció	Azonosító	Típus	Jelleg
Az eredmény fájl azonosítója	FNEV	Sztring	I
A kiértékelendő szelvények	K	KOLLEKCIO	I, M, O
A szelvények száma	KDB	Egész	I
A nyerőszámok tömbje	NY	SZELVENY	I
A nyerőszámok halmaza	H	Egészhalmaz	M
A találat statisztika	S	STAT	M
A fájlváltozó	F	Szövegfájl	M
Segédváltozók	I, J, L	Egész	M

```
/* Kiértékelés, a nyerőszelvények kiírása szövegfájlba */
ERTEKEL(FNEV,K,KDB,NY)

/* Nyerőszámok halmaza */
H ← []
for I ← 1,DB
       H ← H+[NY[I]]
/* Statisztika kezdőértéke */
for I ← 0,DB
       S[I] ← 0
/* A szelvények kiértékelése */
for I ← 1,KDB
       (*K[I]).TAL ← 0
       for J ← 1,DB
              if (*K[I]).SZ[J] IN H
                     (*K[I]).TAL ← (*K[I]).TAL+1
       /* Statisztika */
       S[(*K[I]).TAL] ← S[(*K[I]).TAL]+1
/* Adatkiírás */
NYIT(F,FNEV,"O")
Ki F:"A nyerőszámok"
for I ← 1,DB
       Ki F: NY[I]
/* A nyerőszelvények */
for L ← DB,MINTAL,-1
```

```
        Ki F: L," találatos szelvények száma:",S[L]
        for I ← 1,KDB
              if (*K[I]).TAL=L
                    for J ← 1,DB
                          Ki F: (*K[I]).SZ[J]
ZAR(F)
```

13.4. Láncolt listák

Az eddigiekben tárgyalt tömbszerű adatszerkezetek (statikus tömb, dinamikus tömb és kollekció) egydimenziós változatai egyben ún. *lineáris szerkezetek* is, ami azt jelenti, hogy van egyértelműen meghatározott *első* és *utolsó* elem, valamint (az utolsó elemet kivéve) minden elemhez van egyértelműen meghatározott *követő* elem, valamint (az első elemet kivéve) minden elemhez van egyértelműen meghatározott *előző* elem.

A láncolt listák is ilyen adatszerkezetek, csak itt a sorrendi kapcsolatot a lista elemeiben elhelyezett mutatók hozzák létre. A legegyszerűbb láncolt lista az *egyirányban láncolt* (vagy röviden egyirányú) *lista* (lásd 13.3. ábra). Egyszerűsége ellenére magában hordozza a láncolt listák meghatározó jellegzetességeit, alkalmas ezek bemutatására és megértésére.

13.3. ábra. Egyirányban láncolt lista.

Az elemek eléréséhez ismernünk kell az első elem mutatóját (az ábrán az ELSO adat). A lista végének jelzésére, a minden mutatótípussal kompatibilis NIL értéket használjuk. A listaelemek rekordok lesznek (mivel az ésszerűen elképzelhető esetekben legalább kétfajta adat van egy elemben, az egyik a mutató, a másik az adatsor egy eleme). A mutatók azonos típusú mutatók, az elemek azonos rekordtípusú *dinamikus változók*. A megfelelő deklarációs séma:

LANCELEM1 Rekord

ADAT	A tárolandó adat típusa
KOVETO	LANCELEM1 rekordra mutató

Bizonyos esetekben, pl. egy várakozó sor modellezésénél, ahol az érkezők a sor végére állnak, célszerű az első elem mellett az utolsó elem mutatóját is kezelnünk, mivel ilyenkor nem kell a listán „végiglépkedni” ahhoz, hogy

az utolsó rekordot elérjük, ez egy hivatkozással megtehető, így az új rekord könnyen a listához fűzhető.

Természetesen nem csak egy irányban láncolhatók az elemek. Ha egy feladatban az is szükséges vagy hasznos, hogy az elemeket visszafelé haladva is elérhessük, akkor még egy mutatómezőt veszünk fel a rekordba, ami mindig az előző rekordra mutat (a legelső elemnél NIL), így kapjuk a *kétirányban láncolt* (vagy röviden kétirányú) *listát*. A megfelelő deklarációs séma:

```
LANCELEM2 Rekord
    ADAT              A tárolandó adat típusa
    ELOZO, KOVETO     LANCELEM2 rekordra mutató
```

13.4. ábra. Kétirányban láncolt lista.

Az UTOLSO az utolsó listaelemre mutat, így a lista mindkét irányban elérhető.

Az üres, elemekkel nem rendelkező listát az ELSO (és UTOLSO) változók NIL értéke jelzi.

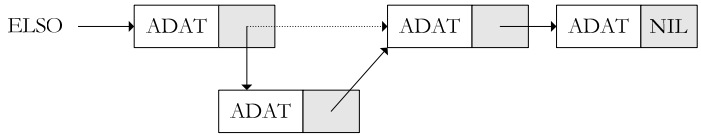
Egy lista *rendezett*, ha a lista elemei, a bennük tárolt adatok szerint rendezetten követik egymást. A későbbiekben feltesszük, hogy a rendezett listák növekvően (nem csökkenően) rendezettek.

A tömbszerű lineáris adatszerkezetekben a fizikai tárolási sorrend adja az egymásra következést, ezzel lehetővé téve az indexelést és az ezzel járó előnyöket (lekérdezés közvetlen hivatkozással, bináris keresés lehetősége stb.).

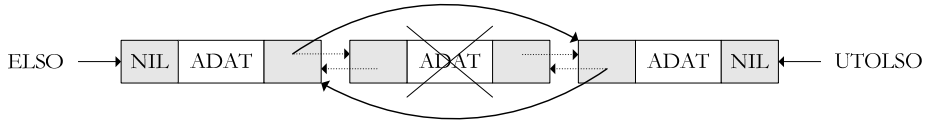
A listaelemekben lévő mutatók ilyen lehetőséget nem adnak, a láncolt listáknál nincs indexe az elemeknek, a hozzáférés, *lekérdezés* alapvetően lineáris, vagyis ha egy elemet el akarunk érni (pl. meg akarunk keresni), akkor ahhoz végig kell járnunk a megelőző elemeket is. Ebből következően tehát egy rendezett listánál sem tudjuk a leghatékonyabb módszert, a bináris keresést alkalmazni.

A láncolt listák igazi előnyei leginkább a módosító-karbantartó jellegű feladatoknál mutatkoznak meg. Míg a rendezett tömböknél egy elem beszúrása, ill. törlése az elem helyétől függően, bizonyos mennyiségű adatmozgatással jár (pl. az első elem törlésénél az összes elemet eggyel előrébb kell léptetni, „át kell pakolni”), addig a láncolt listáknál a beszúrás és törlés

az *elem helyétől függetlenül,* bármely elemre nézve is csak *néhány műveletet* igényel, hiszen egy-két mutató átírásával megvalósítható.



13.5. ábra. Elem beszúrása egy egyirányban láncolt listába.



13.6. ábra. Elem törlése egy kétirányban láncolt listából.

A láncolt listák *tárkihasználása* a dinamikus tömbökhöz hasonlóan nagyon jó, mivel itt is csak a ténylegesen létező adatok tárolódnak, csak nekik foglalódik le memória, míg a statikus tömbök és kollekciók esetén többnyire adathelyek, ill. mutatóhelyek vannak kihasználatlanul (kivéve persze azt a ritka esetet, amikor az aktuális elemszám éppen a maximális elemszámmal megegyező).
A láncolt listákat tehát akkor előnyös használni, ha

- az adatok száma nagyon változó, nehezen korlátozható;
- nincs szükség az elemek közvetlen elérésére;
- a módosító-karbantartó jellegű részek dominálnak a lekérdező jellegűekkel szemben.

Tipikusan ilyenek például a *sorban állást-kiszolgálást*, a várakozó sorokat leképező számítástechnikai modellek.

A láncolt listák is *dinamikus változókból* felépülő adatszerkezetek, ezért a módosítási műveleteket célszerű lépésekre tagolni, egyfajta elvi sémát adva ezek elvégzésére.
Egy új elem *felvétele,* a lista *bővítésének* lépései a következők:

- Az új listaelem (rekord) létrehozása
  - A szükséges hely meglétének (a bővítés lehetőségének) ellenőrzése.
  - A listaelem létrehozása, ha van hely.

- Ha van új elem, akkor
  - A listaelem feltöltése. Ez általában az adatrész végleges állapotának és a mutatók valamilyen célszerű kezdőállapotának (általában NIL) beállítását jelenti.
  - Az új elem beillesztése a struktúrába. Ennek során egyrészt az új elem mutatórészét, másrészt a környezetének (előző és követő elemek) mutatóit kell helyesen beállítani. A lista fajtájától, aktuális állapotától és az új elem helyétől függően szükség lehet még a lista kezdő- és végmutatóinak módosítására is. Szemléletes a beillesztést aszerint tagolni, hogy az új elem hová kerül: a lista *elejére, végére* vagy két, már meglévő elem *közé*.

Egy meglévő elem *törlése,* a lista *szűkítésének* lépései a következők:

- A törlendő elem kikapcsolása a struktúrából. Ennek során a törlendő elem láncbeli környezetének (előző és követő elemek) mutatóit kell helyesen beállítani. A lista fajtájától, aktuális állapotától és az új elem helyétől függően szükség lehet még a lista kezdő- és végmutatóinak módosítására is. Szemléletes a beillesztést aszerint tagolni, hogy az új elem honnan törlődik: a lista *elejéről, végéről* vagy két, már meglévő elem *közül*.
- A törlendő listaelem (rekord) megszüntetése, az általa lefoglalt hely felszabadítása.

Feladat: Keressünk meg egy adott értéket egy egyirányban láncolt listában!

Megoldás: A lista első elemétől elindulva, addig vizsgáljuk a lista elemeit, amíg meg nem találjuk a keresett értékű rekordot, ill. el nem érjük a lista végét. A keresés eredménye a keresett érték első előfordulásának mutatója lesz, ha van ilyen rekord, egyébként NIL. A kereséseknél szokásos elöltesztelő ciklust alkalmazunk, hiszen a lista lehet üres is. Az eredmény kihangsúlyozása érdekében, a megoldásokat függvény formájában adjuk meg. A listán tárolt adatok legyenek egész számok, így a használt rekord (a deklarációs séma szerint) a következő:

```
LANCELEM1 Rekord
    ADAT        Egész
    KOVETO      LANCELEM1 rekordra mutató
```

Funkció	Azonosító	Típus	Jelleg
A lista első eleme	ELSO	LANCELEM1-re mutató	I
A keresett érték	X	Egész	I
Az aktuálisan vizsgált elem	AKT	LANCELEM1-re mutató	M, O

```
/* Keresés egy egyirányban láncolt, rendezetlen listában */
KERESES1(ELSO,X)

AKT ← ELSO
while (AKT<>NIL) AND ((*AKT).ADAT<>X)
      AKT ← (*AKT).KOVETO
return AKT
```

Ha a lista rendezett, akkor csak addig „keresünk”, amíg van esély az elem megtalálására, azaz ha a keresett elem az aktuálisan vizsgált listaelem értékénél nagyobb, és megállunk amint egy nagyobb vagy egyenlő elemre lépünk. Ha a láncon végigértünk vagy a listaelem értéke, ahol megálltunk, nagyobb a keresett értéknél, akkor NIL az eredmény, egyébként annak az elemnek a mutatója, ahol a keresés megállt, hiszen annak értéke egyenlő a keresett értékkel.

Megjegyzés:

- A keresési algoritmusok tehát ugyanolyan elven „működnek”, mint a tömbökben való lineáris keresések (lásd 11.3.2.), csak itt nem tömbelemeken, hanem listaelemeken „lépkedünk”.
- A ciklusokat vezérlő feltételekben most is először az adatok „elfogyását” vizsgáljuk, hogy ne hivatkozzunk nemlétező adatra.

```
/* Keresés egy egyirányban láncolt, rendezett listában */
KERESESREND1(ELSO,X)

AKT ← ELSO
while (AKT<>NIL) AND ((*AKT).ADAT<X)
      AKT ← (*AKT).KOVETO
if (AKT<>NIL) AND ((*AKT).ADAT>X)
      AKT ← NIL
return AKT
```

Feladat: Bővítsünk egy egyirányban láncolt rendezett listát egy új listaelemmel!

Megoldás: Megkeressük az elem helyét a listában, azaz azt az elemet (MIUTAN), ami után be kell illesztenünk a listába az új elemet, majd a megfelelő mutatókat beállítjuk. A keresés az első olyan listaelemen megáll,

amelynek értéke, a beszúrandó elem értékénél nagyobb vagy egyenlő, hiszen ez elé (az ezt megelőző rekord után) kell beilleszteni az új elemet. Feltesszük, hogy a listaelem létezik és adatrésze definiált.

Funkció	Azonosító	Típus	Jelleg
A lista első eleme	ELSO	LANCELEM1-re mutató	I, O
A beszúrandó listaelem	UJ	LANCELEM1-re mutató	I
Az aktuálisan vizsgált elem	AKT	LANCELEM1-re mutató	M
A listaelem, ami után beillesztjük az új listaelemet	MIUTAN	LANCELEM1-re mutató	M

A lista első elemének output jellege a lista megváltozása mellett azt fejezi ki, hogy üres lista esetén az első elem mutatója közvetlenül is megváltozik.

```
/* Rendezett lista bővítése */
LISTARAREND1(ELSO,UJ)

/* Keresés */
AKT ← ELSO
MIUTAN ← NIL
while (AKT<>NIL) AND ((*AKT).ADAT<(*UJ).ADAT)
       MIUTAN ← AKT
       AKT ← (*AKT).KOVETO
/* Beillesztés */
if ELSO=NIL
       /* Üres listára */
       (*UJ).KOVETO ← NIL
       ELSO ← UJ
else if MIUTAN=NIL
       /* Nem üres lista elejére */
       (*UJ).KOVETO ← ELSO
       ELSO ← UJ
else if AKT=NIL
       /* A lista végére, a MIUTAN után */
       (*UJ).KOVETO ← NIL
       (*MIUTAN).KOVETO ← UJ
else
       /* A MIUTAN és az AKT közé */
       (*UJ).KOVETO ← (*MIUTAN).KOVETO
       (*MIUTAN).KOVETO ← UJ
```

Feladat: Vegyünk fel egy új listaelemet egy kétirányban láncolt lista legvégére!

Megoldás: Egyszerűen beállítjuk a megfelelő mutatókat aszerint, hogy üres volt-e a lista vagy sem. Feltesszük, hogy a listaelem létezik és adatrésze definiált. A listán tárolt adatok legyenek egész számok, így a használt rekord (a deklarációs séma szerint) a következő:

LANCELEM2 Rekord
ADAT Egész
ELOZO, KOVETO LANCELEM2 rekordra mutató

Funkció	Azonosító	Típus	Jelleg
A lista első eleme	ELSO	LANCELEM2-re mutató	I, O
A lista utolsó eleme	UTOLSO	LANCELEM2-re mutató	I, O
A felveendő listaelem	UJ	LANCELEM2-re mutató	I

A lista első és utolsó elemének output jellege a lista megváltozása mellett azt fejezi ki, hogy üres lista esetén ezek a mutatók közvetlenül is megváltoznak.

```
/* Beillesztés egy kétirányban láncolt lista végére */
LISTARA2(ELSO,UTOLSO,UJ)
if ELSO=NIL
       /* Üres listára */
       (*UJ).ELOZO ← NIL
       (*UJ).KOVETO ← NIL
       ELSO ← UJ
       UTOLSO ← UJ
else
       /* Az UTOLSO után */
       (*UJ).ELOZO ← UTOLSO
       (*UJ).KOVETO ← NIL
       (*UTOLSO).KOVETO ← UJ
       UTOLSO ← UJ
```

Feladat: Töröljünk egy adott mutatójú listaelemet egy kétirányban láncolt listáról!

Megoldás: A törlést az általános elvi séma szerint végezzük, a kapott adatokban minden információ megvan.

Funkció	Azonosító	Típus	Jelleg
A lista első eleme	ELSO	LANCELEM2-re mutató	I, O
A lista utolsó eleme	UTOLSO	LANCELEM2-re mutató	I, O
A törlendő listaelem	MIT	LANCELEM2-re mutató	I

A lista első és utolsó elemének output jellege a lista megváltozása mellett azt fejezi ki, hogy az egy elemű lista esetén ezek a mutatók közvetlenül is megváltoznak.

```
/* Elem törlése egy kétirányban láncolt listából */
LISTAROL2(ELSO,UTOLSO,MIT)

/* Kikapcsolás */
if (MIT=ELSO) AND (MIT=UTOLSO)
       /* Egyetlen elem */
       ELSO ← NIL
       UTOLSO ← NIL
else if MIT=ELSO
       /* Első, de nem egyetlen */
       (*(*MIT).KOVETO).ELOZO ← NIL
       ELSO ← (*MIT).KOVETO
else if MIT=UTOLSO
       /* Utolsó, de nem egyetlen */
       (*(*MIT).ELOZO).KOVETO ← NIL
       UTOLSO ← (*MIT).ELOZO
else
       /* Belső elem */
       (*(*MIT).ELOZO).KOVETO ← (*MIT).KOVETO
       (*(*MIT).KOVETO).ELOZO ← (*MIT).ELOZO
/* Megszüntetés */
FELSZABADIT(MIT)
```

13.5. Összetett listák

Egy láncolt listán lévő rekordokban bármilyen, eddig megismert típusú adatok tárolhatók, akár egy másik listára mutató mutatók is, így tetszőleges bonyolultságú, a feladathoz leginkább illeszkedő adatstruktúrát definiálhatunk. Az ilyen összetett listákra példaként nézzük meg a szakkönyvekben található *tárgymutatók* felépítését, amelyekben a szöveg egyes kiemelt szavai találhatók ábécé szerinti sorrendben, mindegyik mellett feltüntetve (növekvő sorrendben) azokat az oldalszámokat, ahol a szó előfordul.

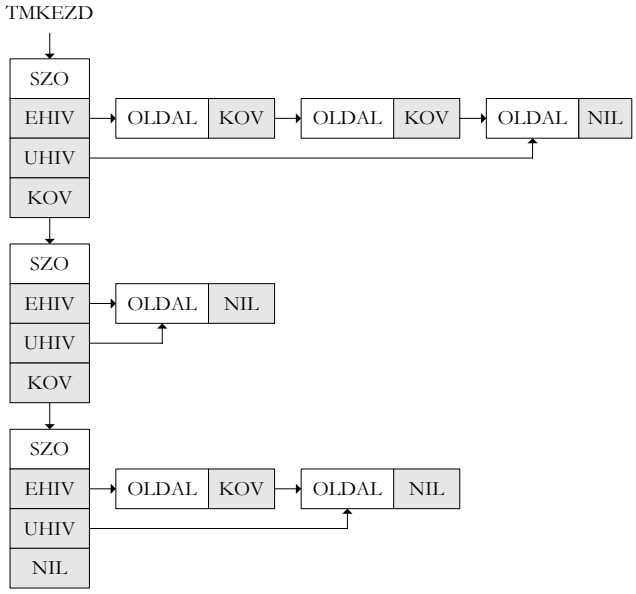
Feladat: Tervezzünk adatstruktúrát egy ilyen tárgymutató memóriában való tárolására!

Megoldás: A szavakat egy egyirányban láncolt, növekvően (tehát ábécé szerint) rendezett listára fűzzük, és minden szóhoz hozzákapcsoljuk a hivatkozások (szintén növekvő sorrendű) egyirányú listáját (lásd 13.7. ábra).

Feltesszük, hogy a tárgymutató bővítésekor egy szót és egy hivatkozást kapunk, ahol a hivatkozások értékei egy adott szó esetén növekvően érkeznek. (Ez a feltételezés eléggé természetes, gondoljunk arra például, ha egy könyvet dolgozunk fel, akkor az oldalszámok növekvő sorrendje szerint haladunk.)

A hivatkozásoknál így az egyediséget és a növekvő sorrendet nem kell ellenőrizni, ezért a szó egy újabb hivatkozását egyszerűen a hivatkozási lista végére kell beillesztenünk. Ezt a beillesztést megkönnyítendő, a szórekordban az első hivatkozása mellett az utolsó hivatkozására is rámutatunk.

```
Típus
HIVREK  Rekord
    OLDAL          Egész
    KOV            HIVREK rekordra mutató
SZOREK  Rekord
    SZO            Sztring
    EHIV, UHIV     HIVREK rekordra mutató
    KOV            SZOREK rekordra mutató
```



13.7. ábra. Tárgymutató felépítése.

Feladat: Vegyünk fel egy új hivatkozást a tárgymutatóba!

Megoldás: Ha az adott szó szerepel már a tárgymutatóban, akkor a hivatkozásait bővítjük az adott hivatkozással, egyébként egy új szórekordot készítünk az adott hivatkozással (ami egyben az első és utolsó hivatkozása is), és a megfelelő helyre (a szavak rendezettsége szerint, amelyet a keresés talált meg), beillesszük a szavak listájába. Feltesszük, hogy a szükséges memória (egy szó és egy hivatkozási rekord részére) rendelkezésre áll (pl. a funkció aktivizálása előtt ezt ellenőriztük.)

A könnyebb áttekinthetőség céljából az új rekord elkészítését tekintsük külön részfeladatnak.

Feladat: Egy adott szó, hivatkozásérték és következő szó mutatójának ismeretében készítsünk egy szórekordot a dinamikus tárban!

Megoldás: Helyet foglalunk a szó, ill. a hozzá tartozó hivatkozásrekordnak, majd beállítjuk a mezők értékeit. Eredményül az új szórekord mutatóját adjuk.

Funkció	Azonosító	Típus	Jelleg
Az új szó	SZO	Sztring	I
A hivatkozás oldalszáma	OLDAL	Egész	I
Az új szót követő szó	KOVSZO	SZOREK-re mutató	I
Az új szórekord	UJSZO	SZOREK-re mutató	M, O
Az új hivatkozási rekord	UJHIV	HIVREK-re mutató	M

```
/* Új szórekord készítése */
UJSZOREK(SZO,OLDAL,KOVSZO)

HELYFOGLAL(UJSZO)
(*UJSZO).SZO ← SZO
(*UJSZO).KOV ← KOVSZO
HELYFOGLAL(UJHIV)
(*UJSZO).EHIV ← UJHIV
(*UJSZO).UHIV ← UJHIV
(*UJHIV).OLDAL ← OLDAL
(*UJHIV).KOV ← NIL
return UJSZO
```

Bővítsük immár a tárgymutatónkat egy kapott szó- és hivatkozásérték alapján az UJSZOREK függvény segítségével!

Funkció	Azonosító	Típus	Jelleg
A tárgymutató kezdete	TMKEZD	SZOREK-re mutató	I, O
A felveendő hivatkozás szava	SZO	Sztring	I
A felveendő hivatkozás oldalszáma	OLDAL	Egész	I
Keresésnél az aktuális rekord	AKT	SZOREK-re mutató	M
Keresésnél az aktuális rekordot megelőző rekord	ELOZO	SZOREK-re mutató	M
Az új hivatkozási rekord	UJHIV	HIVREK-re mutató	M

```
/* Tárgymutató bővítése */
BOVIT(TMKEZD,SZO,OLDAL)

if TMKEZD=NIL
       /* Üres a tárgymutató */
       TMKEZD ← UJSZOREK(SZO,OLDAL,NIL)
else
       /* Keresés */
       AKT ← TMKEZD
       ELOZO ← NIL
       while ((*AKT).SZO<SZO) AND ((*AKT).KOV<>NIL)
              ELOZO ← AKT
              AKT ← (*AKT).KOV
       if (*AKT).SZO<SZO
              /* Végére új szó */
              (*AKT).KOV ← UJSZOREK(SZO,OLDAL,NIL)
       else if (*AKT).SZO>SZO
              /* Az AKT és az ELOZO közé új szó */
              if ELOZO=NIL
                     /* A lista elejére */
                     TMKEZD ← UJSZOREK(SZO,OLDAL,TMKEZD)
              else
                     /* A lista belsejébe */
                     (*ELOZO).KOV ← UJSZOREK(SZO,OLDAL,AKT)
       else
              /* Meglévő szó (AKT) új hivatkozása */
              HELYFOGLAL(UJHIV)
              (*UJHIV).OLDAL ← OLDAL
              (*UJHIV).KOV ← NIL
              (*(*AKT).UHIV).KOV ← UJHIV
              (*AKT).UHIV ← UJHIV
```

Megjegyzés: A keresési ciklust vezérlő feltétel most egy kicsit más, mint a korábbi keresésekben, mivel itt kihasználjuk, hogy a mutató biztosan adatra

mutat, hiszen van legalább egy szó a tárgymutatóban, és az utolsó szónál mindenképpen megállítjuk a keresést (lásd a kifejezés második relációját).

Feladat: Írjuk ki a tárgymutató adatait a képernyőre!

Megoldás: A tárgymutató szavainak listáján, ill. egy adott szó hivatkozásainak listáján „végiglépkedve" egyszerűen kiírjuk a megfelelő adatokat.

Funkció	Azonosító	Típus	Jelleg
A tárgymutató kezdete	TMKEZD	SZOREK-re mutató	I
Az aktuális szórekord	AKT	SZOREK-re mutató	M
Az aktuális hivatkozási rekord	AKTHIV	HIVREK-re mutató	M

```
/* Tárgymutató kiírása a képernyőre */
KIIR(TMKEZD)

AKT ← TMKEZD
while AKT<>NIL
       Ki: (*AKT).SZO
       AKTHIV ← (*AKT).EHIV
       while AKTHIV<>NIL
              Ki: (*AKTHIV).OLDAL
              AKTHIV ← (*AKTHIV).KOV
       /* Esetleges soremelés, billentyűleütésre várás */
       AKT ← (*AKT).KOV
```

13.6. Feladatok

- Egy szövegfájl max. 255 jel hosszú sorokat tartalmaz. A sorok maximális száma 1000. Kollekció segítségével rendezzük a fájl sorait ábécé szerint növekvően!
- Egy táblázat nemnegatív egész értékeket tartalmaz. A sorok és oszlopok száma maximum 500. Oldjuk meg az alábbi feladatokat:
  - Hozzuk létre a táblázatot adott méretekkel!
  - Töltsük fel a táblázatot adott méretekben, adott értékhatáron belüli, véletlenszerűen generált értékekkel!
  - Bővítsük a táblázatot egy újabb sorral, amely az oszlopösszegeket, valamint egy újabb oszloppal, ami a sorösszegeket tartalmazza!
  - Töröljük a táblázat egy adott indexű sorát!

    - Töröljük a táblázat egy adott indexű oszlopát!
    - Mentsük ki a táblázat adatait egy adott nevű szövegfájlba!
- Egy négyzetes táblázat valós számokat tartalmaz. A méret max. 200. Blokknak nevezzük a táblázat egy négyzet alakú részét, amelyet a kezdő sor- és oszlopindexszel, valamint a mérettel adunk meg. Oldjuk meg az alábbi feladatokat:
    - Blokk kiírása a képernyőre mátrix alakban (feltehető, hogy a blokk kifér egy képernyőre)!
    - Blokk szimmetrikusságának ellenőrzése: a blokkban a blokk főátlójára szimmetrikusan elhelyezkedő elemek értéke azonos-e?
    - Blokk szimmetrizálása: a blokkban a blokk főátlójára szimmetrikusan elhelyezkedő elemeket számtani átlagukkal helyettesítjük.
    - Blokkösszeg meghatározása.
- A Turbo Pascal nyelvben egy sztring max. 255 db jelet tartalmazhat. Lépjük túl ezt a korlátot úgy, hogy egy sztringet egy kollekcióban tárolunk, tételenként 255 jelet, kivéve az utolsó tételt, amely ennél kevesebbet is tartalmazhat. A „nagy sztring” a tételek növekvő index szerinti összeolvasásával kapott jelsor lesz. Készítsük el a normál sztringekre használható sztringkezelő függvények megfelelőit:
    - LENGTH;
    - COPY;
    - POS;
    - valamint készítsünk hasonlító függvényt két „nagy sztring” összehasonlítására!
- Egy szövegfájl max. 255 jel hosszú sorokat tartalmaz. Rendezzük a fájl sorait növekvően (láncolt lista segítségével)!
- „Tetszőlegesen nagy”, pozitív egész számokkal dolgozó aritmetikát valósítunk meg tízes számrendszerben úgy, hogy a számokat láncolt listák segítségével tároljuk. Egy számnak egy lista felel meg, ahol a lista elemeiben egy-egy számjegyet tároljuk. Oldjuk meg az alábbi feladatokat:
    - Adott számú számjegyet tartalmazó szám előállítása véletlenszerűen.
    - Szám kimentése szövegfájlba.
    - Szám betöltése szövegfájlból.
    - Szám hosszának (számjegyei darabszámának) a meghatározása.

  - Szám előnullázása adott hosszra.
  - Szám vezető nulláinak törlése.
  - Szám szorzása 10-zel.
  - Szám osztása 100-zal, két eredmény legyen, az egész hányados és a maradék.
  - Két szám összehasonlítása.
  - Számhoz 1 hozzáadása.
  - Két szám összeadása.
- Oldjuk meg az előző feladatot úgy, hogy a lista elemenként 8 számjegyet tartalmazzon. Megvalósítástól függően a lista első vagy utolsó eleme 8 számjegynél kevesebbet is tartalmazhat!
- Készítsünk megoldást az Eratosztenesz szitájának (lásd 9.1.) megvalósítására láncolt lista segítségével úgy, hogy a lista, elemenként egy számot, ill. úgy, hogy elemenként N db számot tartalmazzon (kivéve az utolsó elemet, amely kevesebbet is tartalmazhat)! N legyen konstans!
- A tárgymutató (lásd 13.5.) adatstruktúrához oldjuk meg az alábbi feladatokat:
  - Új hivatkozás felvétele úgy, hogy a hivatkozások tetszőleges sorrendben érkezhetnek (de rendezetten kell őket a láncban tárolni).
  - Hivatkozás törlése úgy, hogy nem maradhat hivatkozás nélküli szó a tárgymutatóban.
  - A tárgymutató kimentése szövegfájlba (soronként egy szó és hivatkozásai).
  - A tárgymutató betöltése (és egyben létrehozása) szövegfájlból (soronként egy szó és hivatkozásai).
- Egy fogadóirodában maximum 10 ablaknál bonyolódnak a fogadások, amik valójában pénz be- és kifizetések. Az ablakokat egyszerűen a sorszámukkal azonosítjuk. Oldjuk meg az alábbi feladatokat:
  - Érkezés: új kuncsaft érkezik és beáll a legrövidebb sor végére. A fizetés iránya (be- ill. kifizetés) és mennyisége (forintban) véletlenszerűen generálandó.
  - Távozás: egy véletlenszerűen kiválasztott, nem üres sor elején lévő kuncsaftot kiszolgáltak és távozik.

    - Leterheltség: meghatározandó ablakonként és összesen a sorban állók száma.
    - Egyenleg: meghatározandó ablakonként és összesen a napi egyenleg (összbefizetés–összkifizetés).
- Alapanyagokból keverékeket állítunk elő. Minden keveréknek van egy egyedi neve, amely max. 25 jelből áll, és betűvel kezdődik. Minden alapanyagnak van egy egyedi azonosító száma, amely egy ötjegyű pozitív egész szám. Egy keverék tetszőleges számú alapanyagból állhat, amit a benne szereplő alapanyagok, és azok mennyiségei határoznak meg. Az adatrendszert egy szövegfájlban tároljuk: keverékenként az első sor a név, utána soronként egy alapanyag a mennyiségével. A fájlban a keverékek ábécé szerint rendezettek, egy keveréken belül az alapanyagok azonosítói is növekvően rendezettek. A rendszer kezeléséhez oldjuk meg az alábbi feladatokat:
    - Adatok betöltése fájlból egy összetett listába.
    - A lista adatainak kimentése fájlba.
    - Keverék keresése, felvétele, törlése.
    - Egy keverék egy alapanyagának felvétele, törlése, mennyiségének módosítása.

14. Fájlok

Az eddigi fejezetekben megismert statikus és dinamikus változók az adatok *belső*, az operatív tárban a program futási ideje alatt történő tárolásának és kezelésének eszközei. Gyakorlati programokhoz a csak belső tárolás nem elegendő, hiszen a legtöbb esetben legalább az egyik, de általában mindkettő fellép az alábbiak közül:

- Az adatokat két feldolgozás, programfutás között is meg kell őrizni, a számítógép által közvetlenül hozzáférhető módon *újrafeldolgozás* és/vagy *továbbfeldolgozás* céljából.
- Az adatok mennyisége olyan nagy, hogy egyszerre nem fér be az operatív tárba, tehát feldolgozás közben is szükség van arra, hogy az adatok egy részét a számítógép által közvetlenül hozzáférhető módon, de az operatív táron kívül tartsuk.

Ezeket az igényeket kielégíteni tudó *külső* adattárolók a lemezes háttértárak (hajlékonylemez, merevlemez). A lemezen tárolt adatok definiálására és kezelésére szolgálnak a *fájl adattípusok* és a *fájlváltozók*.

A fájlok kezelését (valamilyen szinten) tehát minden programfejlesztő rendszernek támogatnia kell. Az operációs rendszer nyújtotta (fájlkezelő) szolgáltatásokat „direkt módon” elérhetővé teszik (pl. megszakításkezeléssel) és/vagy „becsomagolják” az adott nyelvi környezetbe (pl. könyvtári szubrutinok formájában), amely egy könnyebben, kényelmesebben használható „eszköztár” a programozók számára.

A fájlokat bájtok sorozataként képzelhetjük el, amelyek méretét csak az adott operációs rendszer korlátozza. Hogy egy fájlban tárolt bájtsorozatot hogyan értelmezünk és kezelünk, az a programunktól és magától az adatfájltól függ, ezeknek összhangban kell lenniük.
Kétféle fájltípust különböztetünk meg:

- a szövegfájlokat és
- a típusos fájlokat.

A *szövegfájlokat* olyan karaktersorozatnak tekintjük, amelyek a Carriage Return (kocsi vissza, a 13-as ASCII kódú karakter) és a Line Feed (soremelés, a 10-es ASCII kódú karakter) sorvégjelekkel sorokra tagolt.

A *típusos fájlokat* egy adott típusú (fix méretű) adatok, binárisan tárolt sorozatának tekintjük.

A fájlokhoz tartozik egy ún. *fájlmutató*, amely az aktuális írás ill. olvasás kezdőpozícióját adja meg a fájlban. Ez egy 0-tól kezdődő, pozitív egész szám, amely minden egyes írásnál ill. olvasásnál a kiírt ill. beolvasott adat méretével megnő.

A szövegfájlokat szekvenciális, soros elérésű fájloknak tekintjük, ahol az adatok elérése csak sorban, egymás után történhet, a fájlmutatót csak a kiíró ill. beolvasó utasítások állítják.

A típusos fájlokat véletlen elérésű fájloknak tekintjük, ahol az adatokat tetszőleges sorrendben elérhetjük, mivel itt megengedjük a fájlmutató pozícionálását is. Ha pl. 0-ra állítjuk a fájlmutatót, akkor utána a fájl első adatát tudjuk a fájlból kiolvasni, vagy a fájlba beírni.

Megjegyzés

- A típusos fájlok adatait komponenseknek, ill. a gyakori alkalmazás miatt rekordoknak is nevezik. Tetszőleges egyszerű, ill. összetett adattípust megengedünk.
- A C nyelvben használható, eszközhöz (pl. nyomtató) rendelt adatfolyam (stream) szekvenciális fájlnak, míg a lemezes fájlok, bájtokból álló típusos fájloknak felelnek meg.
- A fájlmutatónak (a nevével ellentétben) semmi köze sincs a korábban (lásd 13.2.) tárgyalt mutatókhoz, memóriacímekhez.

14.1. Fájlok kezelése

A fájlokat – a többi adathoz hasonlóan – változókkal fogjuk kezelni. Egy fájlt használat előtt meg kell nyitni, használat után le kell zárni.
Egy fájl megnyitását a

NYIT(fájlváltozó, fájlazonosító, mód)

eljárással végezzük, ahol a

- *fájlváltozó* az adatszerkezeti táblázatban megadott változó;
- *fájlazonosító* a fájl azonosítóját (esetleg meghajtóval, elérési úttal együtt) megadó sztringkifejezés;
- *mód* a fájl megnyitási módja, amelynek lehetséges értékei

  ”I” olvasásra (input),
  ”O” írásra (output),
  ”A”: hozzáfűzésre (append).

A fájlmegnyitás szabályai:
Olvasásra csak létező fájl nyitható meg, a fájlmutató a fájl elejére áll, a fájl tartalma csak olvasható.

Az írásra nyitott, esetlegesen létező fájl tartalma törlődik, a fájl üres lesz, a fájlmutató a fájl elejére áll, a fájlba csak írhatunk.

Ha a nyitás módját kiegészítjük a ”+” karakterrel (felújítás), akkor a fájl olvasható és írható is lesz. Ezek a nyitási módok (”I+”, ”O+”) csak típusos fájlra alkalmazhatók. Nyitáskor a fájlmutató a fájl elejére áll, amit mozgathatunk, beállíthatunk (lásd később), az olvasás ill. írás a fájlmutató által meghatározott pozíciótól kezdődően történik.

A hozzáfűzésre nyitott fájlt írásra nyitjuk úgy, hogy a fájlmutató a fájl végére áll, azaz a fájl tartalma bővíthető a fájlba történő írással. Nemlétező fájl esetén ekvivalens az írásra való nyitással.
Egy fájl lezárását a

ZAR(fájlváltozó)

eljárással végezzük.

Megjegyzés

- Az adatszerkezeti táblázatban szereplő fájlváltozó jellege, a nyitáskor hozzárendelt fájlban tárolt adatok jellegét fejezi ki, azaz ha azok kiinduló adatok, akkor input, ha végeredmények, akkor output, ha pedig részeredmények, akkor munka jellegű.
- Egy fájl meglétének ellenőrzése nyelvenként eltérő, nem részletezzük, az input fájlokról feltesszük, hogy léteznek, és a megfelelő adatokat tartalmazzák.
- Példáinkban a fájlok azonosítóit (az egyszerűség kedvéért) elérési út nélkül, konstansként adjuk meg.
- Az írás ill. olvasás fájlműveletek, a gyorsabb végrehajtás érdekében általában pufferezettek, azaz az adatok először csak egy, a memóriában található átmeneti tárolóba (a pufferbe) kerülnek, és csak akkor történik tényleges adatbeolvasás ill. adatkiírás, amikor a puffer kiürült ill. megtelt. Az írásra nyitott fájlok pufferében lévő (még ki nem írt) adatok fájlba írását ilyenkor a fájl lezárása végzi el.

Adatbeolvasás fájlból

Jelölés

Be fájlváltozó: változólista

Pl. Be F: A, B, C

Végrehajtás

A változókban tárolandó adatokat a fájlváltozóhoz rendelt fájlból olvassuk be.

Megjegyzés

- Típusos fájl esetén a változók típusának meg kell egyeznie a fájl adatainak típusával.
- Szöveges fájl esetén – a beolvasó utasításhoz hasonlóan – az egész, karakter, valós és sztring típusú változók használhatók, a logikai és összetett adattípusú változók nem.

Adatkiírás fájlba

Jelölés

Szövegfájl esetén

Ki fájlváltozó: kifejezéslista

Típusos fájl esetén

Ki fájlváltozó: változólista

Pl. Ki F: ”A kör sugara:”, R, ”területe:”, R*R*3.14

A példában szereplő F egy szövegfájl típusú fájlváltozó.

Végrehajtás: A kifejezések értéke kiértékelődik, majd sorban kiíródik a szövegfájlba, ill. típusos fájl esetén a változók tartalma rendre kiíródik a fájlba.

Megjegyzés: Típusos fájl esetén a változók típusának meg kell egyeznie a fájl adatainak típusával.

Egy fájl végének vizsgálatát a

FAJLVEGE(fájlváltozó)

függvénnyel vizsgáljuk, amely igaz értéket ad, ha a fájlváltozóhoz rendelt fájl fájlmutatója a fájl végén (az utolsó adat után) áll, hamisat különben.

Típusos fájlokra bevezetjük a

FAJLMERET(fájlváltozó)

és a

POZICIO(fájlváltozó)

függvényeket, amely a fájlváltozóhoz rendelt fájl adatainak darabszámát, ill. a fájlmutató aktuális értékét adják, valamint a

POZICIONAL(fájlváltozó, pozíció)

eljárást, amely a fájlmutatót mozgatja a *pozícióval* megadott sorszámú adatra.

Pl. Az F fájlváltozóhoz rendelt típusos fájlban az első adatra a 0, az utolsóra FAJLMERET(F)–1, míg a fájl végére a FAJLMERET(F) pozícióértékkel tudunk rápozícionálni.

Megjegyzés: A FAJLVEGE függvényünk C nyelvű megfelelője csak a fájl végén kiadott sikertelen olvasással áll igazra.

14.2. Szekvenciális fájlok

A szekvenciális elérésű fájlok (ahogy a nevük is kifejezi) csak szekvenciálisan kezelhetők, az adatok csak sorban, elölről kezdve folyamatosan haladva érhetők el, a fájlmutatót nem lehet pozícionálni, azt az olvasás ill. írás műveletek automatikusan végzik. Ezek a fájlok csak egyirányú adatátvitelre nyithatók (”I” vagy ”O”), nem nyithatók meg felújításra.

Megjegyzés: Ilyenek a C-ben az eszközhöz rendelt folyamok, Pascal-ban a szövegfájlok.

14.2.1. Szövegfájl képernyőre listázása

Feladat: Listázzuk ki a képernyőre egy adott szövegfájl tartalmát!

Megoldás: Megnyitjuk a fájlt olvasásra, majd mindaddig, amíg a fájlnak nincs vége, beolvasunk egy karaktert a fájlból és kiírjuk a képernyőre. A soremeléssel külön nem kell foglalkoznunk, hiszen a sorvégjelek kiírása a képernyőn soremelést eredményez.

Megjegyzés

- A C a szövegfájlok olvasásakor a CR-LF sorvégjelekből a CR karaktert elhagyja (transzláció), de a LF karakter kiírása a képernyőre egyben a következő (új sor) elejére is pozícionál. A Pascal meghagyja mindkét sorvégjelet, és eszerint is pozícionál, a CR a sor elejére, a LF pedig a következő sorba (ugyanabba az oszlopba) állítja a kiírás kurzorát.

- Ha a fájl sorai nem hosszabbak 255-nél, akkor a karakteres változót sztring típusúra is kicserélhetjük, így egy sort egy beolvasással ill. kiírással „elintézhetünk”.
- Ha a fájl tartalma hosszabb, mint egy képernyőoldal, akkor billentyűleütésre való várakozással (pl. minden karakter után, minden sor után, adott számú sor után) „lassítható” a kiírás. Megoldásunkban minden DB sor kiírása után (valamint a teljes listázás után, ha kell) várunk billentyűleütésre a VARAKOZAS „utasítással”. Egy sor végét az LF (CHR(10)) karakter beérkezése jelzi.

Konstans

```
DB  20          /* Max. ennyi sort írunk ki egy képernyőre */
```

Funkció	Azonosító	Típus	Jelleg
A listázandó szövegfájl	F	Szövegfájl	I
Az aktuális karakter	CH	Karakter	M
A kiírt sorok száma	S	Egész	M

```
NYIT(F,"SZOVEG.TXT","I")
S ← 0
while NOT FAJLVEGE(F)
       /* Egy karakter beolvasása a fájlból */
       Be F: CH
       /* Kiírás a képernyőre */
       Ki: CH
       if CH=CHR(10)
              S ← S+1
              if S MOD DB=0
                     /* Billentyűleütésre való várás */
                     VARAKOZAS
if S MOD DB<>0
       VARAKOZAS
ZAR(F)
```

14.3. Véletlen elérésű fájlok

Ezek a fájlok, mivel fix méretű, adott típusú adatokból állnak, lehetővé teszik a fájlmutató pozícionálását, így felújításra (olvasásra és írásra) is megnyithatók.

Megjegyzés: A C-ben ilyenek a háttértáron tárolt (lemezes) fájlok (1 bájtos „rekordokkal”), Pascal-ban a típusos és típusnélküli fájlok.

14.3.1. Összefésüléses fájlrendezés

Tegyük fel, hogy adott egy rekordokat tartalmazó típusos fájl, amelynek adatait a rekordok egy mezője szerint sorba akarjuk rendezni. Többféle megoldás is szóba jöhet attól függően, hogy a fájl hány rekordot tartalmaz és hogy a rekordok mérete mekkora. Mintapéldánkban egy olyan megoldást ismertetünk, amelynek memóriaigénye minimális, háttértárigénye is csak annyi, hogy a rendezendő adatfájl egy másolata is elférjen még a háttértáron.

Definíció: Láncnak nevezzük a rendezendő elemek egy olyan maximális hosszú sorozatát, amelyben az elemek rendezettek.

Pl. a {2, 4, 6, 3, 5, 5, 1, 2} sorozatban 3 db lánc található, a {2, 4, 6}, a {3, 5, 5} és az {1, 2}.

A rendezési elgondolás:

1. Válogassuk szét a rendezendő adatsor láncait két fájlba „egyet ide-egyet oda” alapon, azaz egy láncot az egyik fájlba, egy láncot a másik fájlba tegyünk át.
2. Fésüljük össze páronként a két fájl láncait az eredeti fájlba (a fájl felülírásával), majd az esetlegesen át nem másolt láncokat (pl. ha a két fájl nem egyforma számú láncot tartalmaz) másoljuk a fájl végére. Ha egy lánc keletkezett eredményül, akkor készen vagyunk, az adatsor rendezett, egyébként ismételjük meg a szétválogatás-összefésülés lépéseket, azaz folytassuk az 1. lépéssel!

Két lánc (rendezett adatsor) összefésüléséhez végezzük el a következőket: Vegyük a láncok első elemeit. A kisebbiket (egyenlő esetben az egyiket) tegyük be az eredménylláncba és ebben a láncban (amelyikből áttettük az elemet) vegyük a következő elemet, amelyet hasonlítsunk a másik lánc változatlan elemével. A hasonlítást és a kisebbik elem áttételét hasonlóan elvégezve, majd az eljárást ugyanígy folytatva előbb-utóbb valamelyik lánc elemei elfogynak. Ekkor a másik lánc megmaradt elemeit másoljuk át az eredménylláncba!

Pl. {3, 2, 5, 1, 4}→{3, 1, 4},{2, 5}→{2, 3, 5, 1, 4}→{2, 3, 5},{1, 4}→{1, 2, 3, 4, 5}

Konstans
  FNEV1 ”ADATOK1.DTA” /* A rendezendő adatfájl */
  FNEV2 ”ADATOK2.DTA” /* Segédfájl */
  FNEV3 ”ADATOK3.DTA” /* Segédfájl */
Típus
  TADAT Rekord
    A Egész /* Ezen mező szerint rendezünk */
  ADATFAJL TADAT rekordokból álló fájl

Az egyszerűség kedvéért az adatrekordok csak egy mezőt tartalmaznak (A), amely szerint rendezünk.

Feladat: Készítsünk függvényt annak eldöntésére, hogy egy adatfájlból legutoljára beolvasott rekord az őt tartalmazó lánc utolsó eleme-e vagy sem! A rekordot és a nyitott adatfájlt a szubrutin paraméterként kapja.

Megoldás: Ha a fájlmutató a fájl végén áll, akkor a láncnak is vége van, hiszen nincs több rekord a fájlban, egyébként kiolvassuk a fájl következő rekordját az összehasonlításhoz, majd visszapozícionáljuk a fájlmutatót egy rekorddal, hogy a fájlmutatót ne változtassuk meg. Ha a következő rekord megfelelő mezőjének értéke kisebb, mint a kapott rekord megfelelő mezőjének értéke, akkor a láncnak vége van, egyébként nem.

Funkció	Azonosító	Típus	Jelleg
Az adatfájl	MIBEN	ADATFAJL	I
Az utoljára beolvasott rekord	AKT	TADAT	I
A következő rekord	KOV	TADAT	M
Az eredmény	ER	Logikai	O

```
LANCVEGE(MIBEN,AKT)

if FAJLVEGE(MIBEN)
       ER ← igaz
else
       Be MIBEN: KOV
       POZICIONAL(MIBEN,POZICIO(MIBEN)-1)
       ER ← KOV.A<AKT.A
return ER
```

Feladat: Másoljunk egy láncot egy adatfájlból egy másikba!

Megoldás: Amíg a láncnak nincs vége, addig másoljuk a rekordokat. Mivel egy lánc legalább egy rekordot tartalmaz, ezért hátultesztelő ciklust alkalmazunk.

Funkció	Azonosító	Típus	Jelleg
A forrás adatfájl	BOL	ADATFAJL	I
A cél adatfájl	BA	ADATFAJL	O
Az aktuális rekord	AKT	TADAT	M

```
LANCMASOL(BOL,BA)
repeat
        Be BOL: AKT
        Ki BA: AKT
until LANCVEGE(BOL,AKT)
```

Feladat: Az előző két segédeljárás segítségével készítsük el a rendező eljárást!

Megoldás: A rendezési elgondolás alapján először szétválogatjuk a láncokat két fájlba, majd összefésüljük a láncokat páronként az eredeti fájlba.

Funkció	Azonosító	Típus	Jelleg
A rendezendő adatfájl	C	ADATFAJL	I, M, O
Segédfájlok	A, B	ADATFAJL	M
Láncok száma	LANCDB	Egész	M
Vége van-e az aktuális láncnak	LANCVEG	Logikai	M
Az egyik fájl aktuális eleme	AKT_A	TADAT	M
A másik fájl aktuális eleme	AKT_B	TADAT	M

```
repeat
        /* Szétosztás */
        NYIT(C,FNEV1,"I")
        NYIT(A,FNEV2,"O")
        NYIT(B,FNEV3,"O")
        while NOT FAJLVEGE(C)
                LANCMASOL(C,A)
                if NOT FAJLVEGE(C)
                        LANCMASOL(C,B)
        ZAR(A)
        ZAR(B)
        ZAR(C)
```

```
        /* Összefésülés */
        NYIT(C,FNEV1,"O")
        NYIT(A,FNEV2,"I")
        NYIT(B,FNEV3,"I")
        LANCDB ← 0
        while NOT FAJLVEGE(A) AND NOT FAJLVEGE(B)
                /* Egy lánc összefésülése */
                Be A: AKT_A
                Be B: AKT_B
                repeat
                        if AKT_A.A<=AKT_B.A
                                /* A-ból egy elemet */
                                Ki C: AKT_A
                                LANCVEG ← LANCVEGE(A,AKT_A)
                                if LANCVEG
                                        POZICIONAL(B,POZICIO(B)-1)
                                        LANCMASOL(B,C)
                                else
                                        Be A: AKT_A
                        else
                                /* B-ből egy elemet */
                                Ki C: AKT_B
                                LANCVEG ← LANCVEGE(B,AKT_B)
                                if LANCVEG
                                        POZICIONAL(A,POZICIO(A)-1)
                                        LANCMASOL(A,C)
                                else
                                        Be B: AKT_B
                until LANCVEG
                LANCDB ← LANCDB+1
        /* Maradék másolása */
        if NOT FAJLVEGE(A)
                while NOT FAJLVEGE(A)
                        Be A: AKT_A
                        Ki C: AKT_A
                LANCDB ← LANCDB+1
        if NOT FAJLVEGE(B)
                while NOT FAJLVEGE(B)
                        Be B: AKT_B
                        Ki C: AKT_B
                LANCDB ← LANCDB+1
        ZAR(A)
        ZAR(B)
        ZAR(C)
until LANCDB<=1
```

Megjegyzés: Ha a fájl üres, az összefésült láncok száma 0 lesz és kilépünk.

14.3.2. Indextáblás fájlkezelés

Rendezett adatok között könnyebb keresni. Gondoljunk csak a telefonkönyvre, ahol egy adott névhez tartozó telefonszámot viszonylag gyorsan megtalálhatunk, de egy adott számhoz tartozó név megkeresése már korántsem ilyen egyszerű. Természetesen ezzel nemcsak az ember van így, hanem, amint láttuk (lásd 11.3.), a számítógép is gyorsabban keres rendezett adatok között.

Tegyük fel, hogy adatainkat, egy rekordokat tartalmazó típusos fájlban tároljuk. Ha ezen adatokról rendezett listát szeretnénk készíteni, akkor magukat az adatokat is sorba rendezhetjük a fájlban (pl. az előző fejezet módszerével). Ez sajnos nem a legjobb módszer, hiszen egy adat felvétele ill. törlése esetén a rendezettséget megtartandó, az adatok egy részét mozgatnunk kell, nem is beszélve arról az esetről, ha az adatainkat több szempont szerint is sorba szeretnénk rendezni. Az adatok fizikai rendezése helyett indextáblákat használunk, minden egyes logikai sorrendhez egy indextáblát (lásd 14.1. ábra).

Név szerinti		Adatok		Tel.szám sz-i
indextábla	Ssz.	Név	Telefonszám	indextábla
4	0	Szabó	222222	2
2	1	Varga	777777	0
6	2	Kovács	111111	4
0	3	Takács	444444	3
3	4	Halász	333333	6
5	5	Vadász	666666	5
1	6	Madarász	555555	1

14.1. ábra. Indextáblák használata.

Az adatok az adatfájlban rendezetlenül tárolódnak, rendezettségüket az indextáblák fejezik ki, azaz a rendezettséghez a megfelelő indextáblán „keresztül” kell hivatkoznunk a rekordokat.
A rekordok méretétől és darabszámától függően az alábbi esetek lehetnek:

- Az adatfájl befér egy tömbbe. Ekkor az indextáblák az adattömbbeli indexeket tartalmazzák.

- Az adatfájl befér a dinamikus tárba. Ekkor az indextáblák mutatókat tartalmaznak, amelyek az adatrekordokat tartalmazó dinamikus változókra mutatnak.
- Az adatfájl nem fér be a memóriába. Ekkor az indextáblák az adatfájlbeli rekordsorszámokat tartalmazzák. A gyorsabb keresés végett, hogy ne kelljen minden egyes hivatkozáshoz (pl. hasonlításhoz) a háttértáron lévő adatfájlból olvasni, célszerű a rendezettséget definiáló információkat is az indextáblákban tárolni.
- Az indextáblák nem férnek be tömbökbe, de befférnek a dinamikus tárba. Ekkor láncolt listák segítségével kezelhetők az indextáblák.
- Az ennél nagyobb helyigényű esetekben valamilyen adatbázis-kezelőt kell használnunk, amelyek „elvégzik helyettünk” (igaz, hasonló elven működve) az indexelést.

Mintapéldánkban az egyszerűség kedvéért azzal az esettel foglalkozunk, amelyben az adatokat fájlban, a hozzá tartozó (egyetlen) indextáblát tömbben tartjuk. Ez az eset (amellett, hogy a gyakorlatban is jól használható), jól szemlélteti a típusos fájlok kezelését és az indextáblák használatát.

```
Konstans
   MAXDB  5                 /* Az adatok maximális száma */
   MAXAZHOSSZ  1            /* Az azonosító maximális hossza */
   MAXINFOHOSSZ  3          /* Az információ maximális hossza */
   AFNEV  "ADAT.DAT"        /* Az adatfájl neve */
   IFNEV  "ADAT.IND"        /* Az indextáblát tartalmazó fájl neve */
   URESAZ  " "              /* Üres azonosító */
Típus
   TAZ       Sztring[MAXAZHOSSZ]       /* Az azonosítóhoz */
   TINFO     Sztring[MAXINFOHOSSZ]     /* Az információhoz */
   TADAT  Rekord                       /* Az adatrekord */
      AZ     TAZ                       /* Azonosító */
      INFO   TINFO                     /* Információs rész */
   TITADAT  Rekord                     /* Az indextábla rekordja */
      AZ     TAZ                       /* Eszerint rendezünk */
      POZ    Egész                     /* Az adatfájlbeli rekordpozíció */
   TIT  Egydimenziós TITADAT rekordokat tartalmazó tömb[MAXDB]
   ADATFAJL  TADAT rekordokból álló fájl.
   INDEXFAJL  TIT tömböt tartalmazó fájl.
```

Mivel az indextáblát tömbben tároljuk, ezért a rendszerünkbe felvehető rekordok számát is korlátoznunk kell (MAXDB), hiszen csak annyi adatrekordra hivatkozhatunk, ahány elemű az indextábla. Az indextábla az érvényes adatrekordok adatfájlbeli pozícióját (rekordsorszámát), valamint a rendezettség alapjául szolgáló információt (azonosító) tárolja, elölről feltöltött és azonosítók szerint rendezett.

A rendszerben lévő, érvényes adatrekordok számát adminisztrálnunk kell. A könnyebb mentés és betöltés érdekében ezt a darabszámot magában az indextáblában (amelyet most 0-tól indexelünk), a 0. elemben (POZ) tároljuk.

A két alapvető, adatkarbantartó funkció, az új rekord felvétele és egy meglévő rekord törlése funkciók összefüggnek egymással, így először azt tisztázzuk, hogy mi legyen a törölt rekordokkal.

- Ha bent hagyjuk az adatfájlban, akkor a törölt rekordok szükség esetén innen még kinyerhetők, de így az adatfájl mérete egyre csak nő.
- Ha töröljük az adatfájlból (pl. az utolsó rekorddal történő felülírással, és az adatfájl méretének egy rekorddal való csökkentésével), akkor az adatfájl csak az aktuálisan létező adatokat fogja tartalmazni.

Példánkban egy „köztes” megoldást valósítunk meg kihasználva azt, hogy az indextáblát tömbben tároljuk. A törölt rekordok hivatkozásait minden esetben törölnünk kell az indextáblából (logikai törlés), de az adatfájlbeli (fizikai) törlést, most majd a rekordfelvétel segítségével végezzük el. Ezt megvalósítandó, az indextábla szabad helyeinek rekordsorszámait úgy kezeljük, hogy azok az adatfájl szabad helyeire mutassanak.

Kezdetben, amikor még nincsenek rekordok felvéve, ezek a rekordsorszámok a 0, 1, 2, … értékek, így a rekordok felvétele folyamatosan történik. Ha azonban törlünk egy rekordot, akkor annak rekordsorszámát az indextábla szabad helyeit mutató rész elejére tesszük (lásd 14.3. ábra). Így az adatfájl csak a nyilvántartandó rekordok maximális számáig (MAXDB) nőhet.

	Indextábla			Adatfájl	
Ind	Az-ó	Poz	Ssz	Az-ó	Inf-ó
0		3	0	b	bbb
1	a	2	1	d	ddd
2	b	0	2	a	aaa
3	d	1			
4		3			
5		4			

14.2. ábra. A *b*, *d*, *a* azonosítójú rekordok felvétele utáni állapot.

	Indextábla			Adatfájl	
Ind	Az-ó	Poz	Ssz	Az-ó	Inf-ó
0		2	0	b	bbb
1	a	2	1	d	ddd
2	d	1	2	a	aaa
3		0			
4		3			
5		4			

14.3. ábra. A *b* azonosítójú rekord törlése utáni állapot.

	Indextábla			Adatfájl	
Ind	Az-ó	Poz	Ssz	Az-ó	Inf-ó
0		3	0	c	ccc
1	a	2	1	d	ddd
2	c	0	2	a	aaa
3	d	1			
4		3			
5		4			

14.4. ábra. A *c* azonosítójú rekord felvétele utáni állapot.

Feladat: Keressünk meg egy azonosítót az indextáblában!

Megoldás: Mivel az indextábla azonosító szerint rendezett, így alkalmazható a bináris keresés. Ha a keresett elem nem található meg az elemek között, akkor a kezdőindex (I) értékét adjuk eredményül (HOL), amely ilyenkor azt mutatja, hogy hol lenne az elem helye a rendezettség szerint.

Funkció	Azonosító	Típus	Jelleg
A keresett azonosító	MIT	TAZ	I
Az indextábla	MIBEN	TTT	I
Az indextábla elemeinek száma	DB	Egész	I
A keresett elem helye	HOL	Egész	O
A keresett elem létezése	VAN	Logikai	O
Az aktuális kezdőindex	I	Egész	M
Az aktuális végindex	J	Egész	M
A középső elem indexe	K	Egész	M

```
BINKER(MIT,MIBEN,DB,HOL)

I ← 1
J ← DB
VAN ← hamis
while (I<=J) AND NOT VAN
       /* Felezés */
       K ← (I+J) DIV 2
       if MIT=MIBEN[K].AZ
              VAN ← igaz
       else if MIT<MIBEN[K].AZ
              J ← K-1
       else
              I ← K+1
if VAN
       HOL ← K
else
       HOL ← I
return VAN
```

Feladat: Szúrjunk be egy új elemet az indextábla egy adott helyére!

Megoldás: A rendezettséget megtartandó, a beszúrandó elemnek helyet készítünk az indextáblában az elemek hátraléptetésével, majd egyszerűen berakjuk az elemet az adott helyre.

Funkció	Azonosító	Típus	Jelleg
A beszúrandó elem	MIT	TITADAT	I
Az indextábla	MIBE	TIT	I, M, O
A beszúrás helye	HOVA	Egész	I
Az indextábla elemeinek száma	DB	Egész	I, O
Segédváltozó a helykészítéshez	I	Egész	M

```
BESZUR(MIT,MIBE,HOVA,DB)

/* Helykészítés */
for I ← DB,HOVA,-1
      MIBE[I+1] ← MIBE[I]
/* Beszúrás */
MIBE[HOVA] ← MIT
/* Darabszám növelés */
DB ← DB+1
```

Feladat: Töröljünk egy adott indexű elemet az indextáblából!

Megoldás: A törlendő elem rekordsorszáma (a hely, ahol a hozzá tartozó adatrekord megtalálható az adatfájlban) immár egy szabad helyet mutat az adatfájlban, ezért őt az indextábla első szabad helyére tesszük, míg magát az elemet, a mögötte lévő elemek előreléptetésével töröljük.

Funkció	Azonosító	Típus	Jelleg
Az indextábla	MIBOL	TIT	I, M, O
A törlendő elem indexe	HONNAN	Egész	I
Az indextábla elemeinek száma	DB	Egész	I, O
A törlendő elem tárolásához	S	TITADAT	M
Segédváltozó az előreléptetéshez	I	Egész	M

```
TOROL(MIBOL,HONNAN,DB)
/* A törlendő elem megjegyzése */
S ← MIBOL[HONNAN]
/* Törlés */
for I ← HONNAN+1,DB
      MIBOL[I-1] ← MIBOL[I]
/* Hogy az adatfájlba ide vegyünk fel legközelebb */
MIBOL[DB] ← S
/* Darabszám csökkentés */
DB ← DB-1
if DB=0
      /* Az elejéről töltsük fel az adatfájlt */
      for I ← 1,MAXDB
            MIBOL[I].POZ ← I-1
```

Megjegyzés: Ha az utolsó elemet töröljük az indextáblából, akkor az indextábla rekordmutatóit inicializáljuk, így az ezután felvett rekordok folyamatosan kerülnek majd az adatfájlba.

Feladat: Készítsük el az új rekord felvételét elvégző funkciót!

Megoldás: A szükséges adatbekérések és ellenőrzések után az adatrekordot a nyitott adatfájlba írjuk, míg a hozzá tartozó indextábla-rekordot az indextáblába illesztjük.

Funkció	Azonosító	Típus	Jelleg
Az indextábla	MIBE	TIT	I, O
Az indextábla elemeinek száma	DB	Egész	I, O
Az adatfájl	AF	ADATFAJL	O
Az új elem adatrekordja	ADAT	TADAT	I
Az új elem indextábla rekordja	ITADAT	TITADAT	M
Az új elem helye az indextáblában	HOL	Egész	M

```
FELVETEL(MIBE,DB,AF)
if DB=MAXDB
       Ki: "Nem vehető fel több elem!"
else
       Be: ADAT.AZ
       if BINKER(ADAT.AZ,MIBE,DB,HOL)
              Ki: "Van már ilyen azonosítójú rekord!"
       else
              Be: ADAT.INFO
              /* Beírás az adatfájlba */
              POZICIONAL(AF,MIBE[DB+1].POZ
              Ki AF: ADAT
              /* Felvétel az indextáblába */
              ITADAT.AZ ← ADAT.AZ
              ITADAT.POZ ← MIBE[DB+1].POZ
              BESZUR(ITADAT,MIBE,HOL,DB)
```

Feladat: Készítsük el a rekord törlését elvégző funkciót!

Megoldás: A szükséges adatbekérések és ellenőrzések után egyszerűen töröljük a rekordhoz tartozó indextábla-bejegyzést.

Funkció	Azonosító	Típus	Jelleg
Az indextábla	MIBOL	TIT	I, O
Az indextábla elemeinek száma	DB	Egész	I, O
A törlendő elem azonosítója	MIT	TAZ	I
A törlendő elem helye az indextáblában	HOL	Egész	M

```
TORLES(MIBOL,DB)
if DB=0
       Ki: "Nincs mit törölni!"
else
       Be: MIT
       if NOT BINKER(MIT,MIBOL,DB,HOL)
              Ki: "Nincs ilyen rekord!"
       else
              TOROL(MIBOL,HOL,DB)
```

Feladat: Írjuk ki a létező rekordok adatait a képernyőre!

Megoldás: Az indextábla alapján haladva behozzuk a megfelelő adatrekordot, majd kiírjuk a tartalmát. Felhasználjuk, hogy az érvényes rekordok számát az indextábla 0. elemében tároljuk (POZ).

Funkció	Azonosító	Típus	Jelleg
Az indextábla	IT	TIT	I
Az adatfájl	AF	ADATFAJL	I
Az aktuális rekord	ADAT	TADAT	M
Segédváltozó	I	Egész	M

```
KIIRAS(IT,AF)
for I ← 1,IT[0].POZ
       POZICIONAL(AF,IT[I].POZ)
       Be AF: ADAT
       Ki: ADAT.AZ,ADAT.INFO
```

Feladat: Befejezésképpen készítsük el az indextáblás adatkarbantartás funkcióit (felvétel, törlés, kiírás) aktivizáló keretet!

Megoldás: Ha a szükséges fájlok (adatfájl, indextáblát tartalmazó fájl) léteznek, akkor az adatfájlt megnyitjuk, az indextáblát betöltjük, ha nem léteznek, akkor az adatfájlt létrehozzuk, és az indextáblát inicializáljuk. Az adatkarbantartást az előzőekben megvalósított funkciókkal végezzük, kilépéskor az adatfájt lezárjuk és az indextáblát elmentjük. Az adatfájlt felúji-

tásra nyitjuk meg (”+”), mert olvasni és írni is szeretnénk. Egy fájl létezését a FAJLVAN függvénnyel végezzük.

Funkció	Azonosító	Típus	Jelleg
Az adatfájl	AF	ADATFAJL	I, O
Az indextáblát tartalmazó fájl	ITF	INDEXFAJL	I, O
Az indextábla	IT	TIT	M
A kiválasztott funkció	C	Karakter	M
Segédváltozó	I	Egész	M

```
if FAJLVAN(AFNEV) AND FAJLVAN(IFNEV)
       /* Az adatfájl megnyitása */
       NYIT(AF,AFNEV,”I+”)
       /* Az indextábla betöltése */
       NYIT(ITF,IFNEV,”I”)
       Be ITF: IT
       ZAR(ITF)
else
       /* Az adatfájl megnyitása */
       NYIT(AF,AFNEV,”O+”)
       /* Az indextábla inicializálása */
       for I ← 0,MAXDB
              IT[I].AZ ← URESAZ
              if I=0
                     /* A létező adatok száma */
                     IT[I].POZ ← 0
              else
                     /* A leendő rekordsorszámok */
                     IT[I].POZ ← I-1
repeat
       Ki: ”Felvétel:1 Törlés:2 Kiírás:3 Kilépés:0”
       Be: C
       if C=’1’
              FELVETEL(IT,IT[0].POZ,AF)
       else if C=’2’
              TORLES(IT,IT[0].POZ)
       else if C=’3’
              KIIRAS(IT,AF)
until C=’0’
/* Az adatfájl zárása */
ZAR(AF)
/* Az indextábla mentése */
NYIT(ITF,IFNEV,”O”)
Ki ITF: IT
ZAR(ITF)
```

14.4. Feladatok

- Válogassuk ki egy adott nevű, max. 255 jel hosszú sorokat tartalmazó szövegfájlból azokat a sorokat, amelyek tartalmaznak egy adott sztringet! A sorokat egy adott nevű szövegfájlba tegyük, az eredeti fájl formátumában és sorrendjében!
- Készítsünk megoldást az Eratosztenesz szitájának (lásd 9.1.) megvalósítására típusos fájl segítségével!
- Levélcímen értünk egy olyan sztringet, amely három részből áll: irányítószám (pontosan 4 jegyű, pozitív egész szám), helységnév (nem tartalmaz szóközt), többi adat. Az adatrészek elválasztására egy-egy szóközt használunk. A sztring hossza nem haladja meg a 80 jelet. Az adatok egy szövegfájlban vannak, soronként egy levélcím. A fájl tartalma nem rendezett. Oldjuk meg az alábbi feladatokat indextáblák segítségével:
    - Irányítószám szerint rendezett lista a képernyőre.
    - Helységnevek ábécé szerint rendezett listája adott nevű szövegfájlba.
    - Irányítószám szerinti keresés.
    - Helységnév szerinti közelítő keresés: az első olyan helység kiválasztása, amelyik egy megadott szóval kezdődik!
- Oldjuk meg az előző feladatot úgy is, hogy az indextáblákat listákban, az adatokat erre a célra elkészített típusos fájlban (egy rekordban egy levélcím) tároljuk!
- A dinamikus adatstruktúrát igénylő, fájlkezeléshez kapcsolódó feladatok a 13.6. fejezetben találhatók.

15. Gráfok

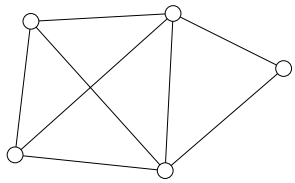
15.1. Alapfogalmak

A korábbi fejezetekben többnyire lineáris szerkezetű (tömb, lista), vagy ilyenekből felépített összetett adatszerkezetekkel foglalkoztunk. Ezekben az elemek egymáshoz való viszonya meglehetősen egyszerű, hiszen alapvetően sorrendiségről, egymásutániságról van szó. Ebben a fejezetben *gráfokkal* foglalkozunk, amelyek már összetettebb, bonyolultabb viszonyok és kapcsolatrendszerek modellezésére is alkalmasak. A gráf egyben matematikai fogalom is, több matematikai tudományág (pl. gráfelmélet) tárgya ill. eszköze, de matematikai definíciók helyett, leírásunkban a számítástechnikai modellezéshez, programtervezéshez elegendő pontosságú fogalmak kialakítására, a gráfok számítógépes kezelésének bemutatására törekszünk.

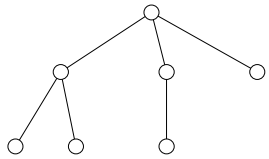
Gráfon bizonyos *elemek* és a köztük fennálló *közvetlen kapcsolatok* halmazát értjük. Nevezzük az elemeket *pontoknak*, a kapcsolatokat *éleknek*.

A gráfok grafikus ábrázolásakor a pontokat egyszerű alakzatokkal (pl. kör), az éleket, a pontokat összekötő vonalakkal, nyilakkal jelölhetjük.

Pl. A pontok emberek, és két pont között akkor van él, ha a két ember ismeri egymást. A megfelelő gráf lehet a 15.1. ábra szerinti is, ha viszont egy olyan szervezetről van szó, ahol mindenki csak a saját, közvetlen főnökét és saját, közvetlen beosztottjait ismeri, akkor a megfelelő gráf inkább a 15.2. ábrán látható szerkezetű lesz. A helyes értelmezéshez, ábrázoláshoz meg kell jelölni a „nagyfőnököt" is, akinek már nincs főnöke. Az ilyen, speciális struktúrájú gráfokat röviden *fáknak* nevezzük, a megjelölendő pont a fa gyökérpontja.



15.1. ábra. Általános gráf.



15.2. ábra. Fa.

Ha az élekhez irányítást rendelünk, kifejezve ezzel azt, hogy a kapcsolat nem feltétlenül szimmetrikus, akkor *irányított gráfról* beszélünk.

Pl. A pontok közlekedési csomópontok, az *a* pontból a *b* pontba akkor mutat él, ha az ilyen irányú közlekedés lehetséges (az egyirányú útszakaszok nem szimmetrikus kapcsolatok).

Egyszerű gráf az olyan gráf, amelyben bármely két pont között egy irányban legfeljebb egy él van és nincs *hurokél* (egy pont önmagával vett kapcsolata).

Pl. A pontok városok és a városok közötti elérhetőséget akarjuk az élekkel kifejezni, akkor több él is létezhet egyik városból egy másikba attól függően, hogy milyen közlekedési eszközöket (repülő, autó, vonat, busz, stb.) veszünk figyelembe.

A továbbiakban egyszerű gráfokkal foglalkozunk, példáink és algoritmusaink is ilyen gráfokra vonatkoznak.

Teljes gráf az olyan gráf, amelyben bármely két pont között van él (pl. egy baráti társaság, ahol mindenki ismeri egymást.).

Út a gráf egy olyan pontsorozata, amelyben a szomszédos pontok között van él.

Körmentes út egy olyan út, amelyben minden pont különböző, egyébként (ha van ismétlődő pont) az út *körös út*.

Egy irányítatlan gráf *összefüggő*, ha bármely két pontja között van út.

Megjegyzés: A gráfelméletben többféle „összefüggőség” fogalom is definiált.

Fa struktúrájú gráf, vagy röviden *fa* az olyan összefüggő, irányítatlan gráf, amelyben nem hozható létre körös út. Megjegyezzük, hogy az irányított él fogalmát tehát (olyan értelemben, mint az általános gráfoknál) itt nem vezetjük be, de speciális irányítás lehetséges (lásd 15.3. gyökeres fák).

Egy gráf *részgráfján* értjük a pontok és a köztük lévő élek egy részhalmazát.

Egy gráfot *szimmetrikusnak* nevezünk, ha irányítatlan, vagy minden irányított élnek van párja (azaz tetszőleges *a*, *b* pontok esetén, ha az *a*-*b* kapcsolat fennáll, akkor a *b*-*a* kapcsolat is).

A gráfok segítségével modellezendő gyakorlati problémák esetén a pontokhoz és az élekhez különféle adatok kapcsolódnak (pl. egy közlekedési hálózat esetén egy csomópont azonosítója, koordinátái, egy él hosszúsága, stb.), ezek együttese, az ún. *hálózati adatok* tárolandók, kezelendők.

15.2. Tárolás

A gráf, ill. a hálózat mint standard adattípus nem szerepel az univerzális programnyelvekben, de elég bonyolult ahhoz, hogy több értelmes és bizonyos (természetesen eltérő) szempontból optimális implementációja legyen.

A számítástechnikai modellekben elsődlegesen megoldandó részfeladat az azonosítás, tehát a hálózati pontokhoz és élekhez egyértelmű és a programmal kezelhető elnevezéseket, azonosítókat kell rendelnünk. Elsődleges a pontok azonosítása, az éleké ezekre már könnyen visszavezethető (hiszen egy élt a kezdő és végpontja „beazonosítja”).

A pontok „természetes”, vagyis a modellezendő feladatbeli azonosítója általában egy hosszabb, „beszélő” jellegű kód, mint pl. a személyi szám, vagy közlekedési hálózat esetén a kereszteződő utak nevei.

Az adatstruktúrák megválasztásánál az alábbi szempontok vehetők figyelembe:

- A *tárkihasználás*, a központi memóriában való tárolhatóság.
- A *karbantarthatóság*, a változások átvezetésének műveletigénye.
- A *lekérdezhetőség*, az információkinyerés műveletigénye.

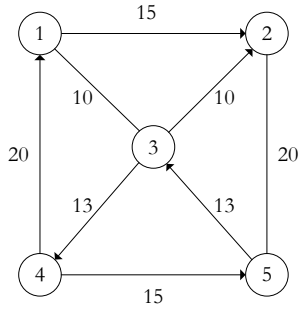
A számítástechnikában általában is igaz, és itt is érvényesül az, hogy memóriát nyerni csak végrehajtási idő rovására lehet és viszont, gyorsítani többnyire csak nagyobb memória felhasználásával tudunk. Ezeket mérlegelve választhatunk a lehetséges megoldások között.

A *pontok* adatait célszerű egy rekordokból álló, egydimenziós tömbben tárolni, ahol a rekord típusba az egy ponthoz tartozó adatok kerülnek.

Az *élek* tárolása már nem ilyen egyszerű és nyilvánvaló, hiszen ez erősen függ a tárolandó gráftól, az abban található élek számától.

Három lehetséges tárolási módszert mutatunk be, kitérve az egyes változatok jellemzőire tárkihasználás, karbantarthatóság és lekérdezhetőség szerint.

Tekintsük az alábbi példahálózatot.



15.3. ábra. Példahálózat.

Az ábrán az egyirányú éleket nyilak, a mindkét irányban létező éleket vonalakkal ábrázoltuk.

Egy egyszerűsített úthálózat modelljeként a gráfhoz három pontjellemzőt (az azonosítót és a két koordinátát) és egy éljellemzőt (hossz) veszünk fel (az ábrán az élek melletti számok), amelyekről feltesszük, hogy a kétirányú éleknél mindkét irányban azonosak.

A pontokat az egyszerűség kedvéért egész számokkal, a sorszámukkal azonosítottuk (a körökben lévő számok).

15.2.1. Pontok tárolása

A pontok adatait egy egydimenziós, rekordokból álló tömbben tárolhatjuk, amelyet célszerű a pontazonosítók szerint rendezetten tárolni, hogy a *pontazonosító* → *pontindex* konverzió bináris kereséssel (azaz a leggyorsabb módon) elvégezhető legyen. A belső adattárolásban ezután elegendő a pontok helyett az indexeikkel dolgoznunk, mert azok egyértelműen beazonosítják a pontokat és segítségükkel közvetlenül (egy indexeléssel) elérhetők az egyes pontok adatai.

Természetesen ennek a gyors elérésnek „ára van”, nevezetesen a kezdeti rendezés mellett, egy pont felvétele ill. törlése esetén, a rendezettséget megtartandó, mozgatni kell az adatok egy részét a tömbben.

Sorszám	1	2	3	4	5
Azonosító	1	2	3	4	5
X koordináta	0	15	8	0	15
Y koordináta	20	20	12	0	0

15.4. ábra. Pontok tárolása.

Újabb pontjellemző a pontrekord egy újabb adatmezőjét (vagy az adatokat pontjellemzőnként külön tömbökben tárolva egy újabb *pontszám* elemű egydimenziós tömb tárolását) jelenti.

15.2.2. Élek tárolása kétdimenziós tömbben

Minden éladathoz egy *pontszám*pontszám* méretű kétdimenziós tömb tartozik, amelyben minden elem egy él adatát tárolja úgy, hogy a sorok az egyes pontokból kiinduló, az oszlopok pedig az egyes pontokba érkező élek adatait tartalmazza.

Élhossz	1	2	3	4	5
1	-	15	10	-	-
2	-	-	-	-	20
3	10	10	-	13	-
4	20	-	-	-	15
5	-	20	13	-	-

15.5. ábra. Élhosszak tárolása kétdimenziós tömbben.

A *tárkihasználtság* teljes gráfoknál a legjobb (ekkor csak a főátló kihasználatlan), de „helypazarló” az ún. ritka gráfoknál, amelyekben a lehetséges éleknek csak egy töredéke definiált (pl. egy 1000 pontos közlekedési hálózatban a lehetséges 999000 irányított élből az élek száma nagy valószínűséggel 5000 alatt van (minden pontból átlagosan 5 db kimenőélt feltételezve), azaz a kihasználtság még az 1 százalékot sem éri el).

A *karbantarthatóság* kedvező, hiszen egy élt felvenni, törölni, él- vagy pontjellemzőt módosítani nagyon egyszerű, hiszen csak át kell írni egy tömbelemet.

Egy pont felvétele ill. törlése már műveletigényesebb, hiszen ez a sorok és oszlopok mozgatását jelenti.

A *lekérdezhetőség* optimális, a lehető leggyorsabb, hiszen az éljellemzők (az él két pontjának esetleges beazonosítása után) a pontindexek alapján egyszerű indexeléssel elérhetők.

Ha nem egy, hanem több éljellemzőt szeretnénk modellezni, akkor éljellemzőnként egy-egy (pontszám*pontszám elemű) kétdimenziós tömb szükséges, amely a nagy tárigény miatt korlátot szabhat a módszer gyakorlati alkalmazhatóságára.

15.2.3. Élek tárolása egydimenziós tömbben

A módszer lényege, hogy csak a meglévő kapcsolatok jellemzőit, a ténylegesen létező élek adatait tároljuk, bizonyos rendezettség mellett, ami a gyorsabb lekérdezést biztosítja.

Az élek adatait élenként kiegészítjük az él végpontjának, a pont tömbben elfoglalt helyét megadó indexével, majd ezeket élrekordba foglalva egy tömbben (vagy külön tárolva őket, adatonként egy-egy tömbben) tárolhatjuk.

A pontok adatait pontonként kiegészítjük egy mutatóértékkel, amely megadja az adott pontból, mint kezdőpontból kiinduló első él indexét.

Sorszám	1	2	3	4	5
Élmutató	1	3	4	7	9

15.6. ábra. Pontok kiegészítése élmutatókkal.

Sorszám	1	2	3	4	5	6	7	8	9	10
Végpont	2	3	5	1	2	4	1	5	2	3
Élhossz	15	10	20	10	10	13	20	15	20	13

15.7. ábra. Éladatok tárolása egydimenziós tömbben.

Az éleket tömören, a kezdőpont, azon belül a végpontok sorrendjében tároljuk.

A *tárkihasználás* nagyon jó, a memóriaigény az élek számával lineárisan nő, egy újabb éljellemző modellezéséhez élrekord esetén egy újabb adatmező (egyébként egy újabb (*élszám* méretű) egydimenziós tömb) szükséges.

A *karbantarthatóság* és a *lekérdezhetőség* az előző módszerhez képest roszszabb lesz.

Egy él felvétele ill. törlése a tömör és rendezett adattárolás miatt az éladatok (egy részének) mozgatásával, a tömbben (tömbökben) való lépte-

tésével, s emiatt az *Élmutató* tömb megfelelő igazításával, míg egy pont felvétele ill. törlése az élek *végpontjainak*, mint pontindexeknek az igazításával jár.

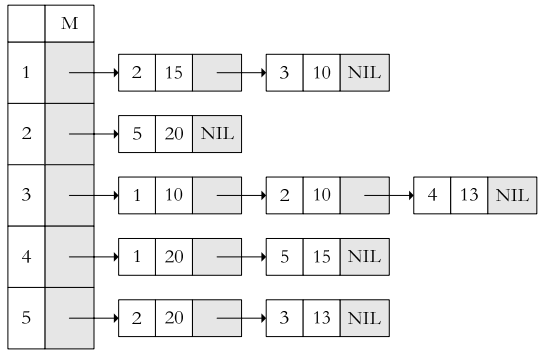
Az éljellemzők eléréséhez, a kezdőpont sorszámával az *Élmutató* tömböt indexelve az első él egy lépésben elérhető, de innen kiindulva már keresni kell a végponthoz tartozó élt. Ezt a keresést gyorsítja az azonos kezdőpontú élek végpont szerinti rendezettsége. Minél ritkább a hálózat, annál gyorsabb átlagosan a lekérdezés.

15.2.4. Élek tárolása dinamikusan

Az éladatok tömör, helytakarékos tárolása dinamikusan, láncolt listák segítségével is megvalósítható.

Az előző módszerhez hasonlóan az élek adatait kiegészítjük az élek végpontjának indexével, de most még egy, ilyen élrekordra mutató mezővel is ellátjuk, hogy az élrekordokat egy egyirányú listára fűzhessük.

A pontok adatait kiegészítő *Élmutató* tömb most is az adott pontból, mint kezdőpontból kiinduló első élt éri el, de most nem index, hanem egy élrekordra mutató mutatóval.



15.8. ábra. Élek tárolása dinamikusan.

A *tárkihasználás* jó, hiszen az élek adatain kívül csak a mutatókat kell tárolnunk, a memóriaigény az élek számával lineárisan nő, egy újabb éljellemző modellezéséhez az élrekordban egy újabb adatmező szükséges.

A *karbantarthatóság* az előző módszerhez képest jobb lesz, hiszen az élek felvétele ill. törlése esetén nem kell az élek adatait mozgatni, hiszen a rekordok helyfoglalása ill. felszabadítása mellett, a megfelelő mutatók beállításával ezek a karbantartások elvégezhetők.

Egy pont felvétele ill. törlése (hasonlóan az előző módszerhez), az élrekordban tárolt élvégpontok, mint pontindexek megfelelő igazításával jár.

A *lekérdezhetőség* az előző módszerhez képest kicsit rosszabb lesz. Noha az egy pontból kiinduló első él ugyanúgy, a kezdőpont sorszámával az *Élmutató* tömböt indexelve egy lépésben elérhető, de az innen kiinduló keresés (a lánc miatt) csak lineáris lehet. Természetesen az élek végpont szerinti rendezettsége itt is gyorsíthatja a keresést.

Végezetül megjegyezzük, hogy a pontadatok is tárolhatók tömb helyett láncolt listákkal, ekkor egy pont felvétele ill. törlése esetén nem kell a pontadatokat mozgatnunk, viszont a pontok keresése (az *azonosító* → *pontrekord* konverzió) csak lineáris lehet. Az éladatokban ekkor a végpont rekordra mutató mutatókat kell tárolnunk és adminisztrálnunk a végpontindexek helyett.

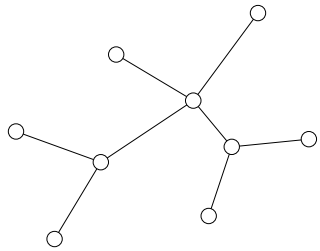
15.3. Fák

Ahogy azt már a bevezető részben definiáltuk, fa struktúrájú gráfon vagy röviden fán, egy olyan összefüggő, irányítatlan gráfot értünk, amelyben nem hozható létre körös út.
Néhány ekvivalens állítás a fákra vonatkozóan:

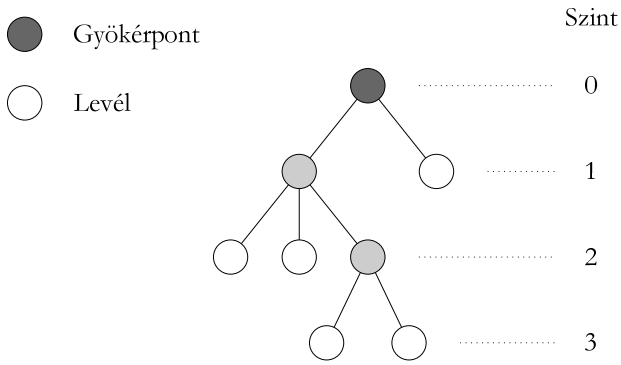
- Bármely két pont között egyértelműen létezik egy őket összekötő út.
- Éppen eggyel kevesebb él van, mint pont.
- Akár csak egy élt hozzávéve az élekhez, a kapott gráf már tartalmaz kört, azaz nem lesz fa.
- Akár csak egy élt is elhagyva az élek közül, a kapott gráf már nem lesz összefüggő, azaz már nem lesz fa.

Gyökeres fa az olyan fa, amelyben egy pontot kitüntetünk. Ezt a fa *gyökérpontjának* (root) nevezzük. A gyökérponthoz nem értelmezünk megelőző pontot, de minden más pontnak egyértelműen definiálható a *megelőzője* (más terminológiával szülő vagy ős). Egy pont megelőzője, a gyökérpontból hozzávezető (a fa struktúra következtében egyértelműen meghatározott, körmentes) úton, az őt megelőző pont. Egy pont *követője* (más terminológiával gyermek vagy leszármazott) az a pont, amelynek ő a megelőzője. A fa struktúrából következően egy pontnak több leszármazottja is lehet, de szülője csak egy, míg a gyökérpontnak nincs szülője. Az olyan pontot, amelynek nincs követője, *levélnek* hívjuk.

Megjegyzés: A fák rajzos ábrázolásánál a matematikai és informatikai szakirodalomban a „fordított állás” a szokásos, azaz a gyökérpont van legfelül és a fa lefelé ágazik el. A továbbiakban fán gyökeres fát értünk.



15.9. ábra. Fa.



15.10. ábra. Gyökeres fa.

Fákra értelmezhetjük a *szintek* és a *magasság* fogalmát is. A 0. szint a gyökérpont szintje, az 1. szinten a gyökérpont leszármazottai, a 2. szinten az 1. szinten lévő pontok leszármazottai találhatók, és így tovább. A fa magasságát a legnagyobb szintszám értéke adja.

Könnyen látható, hogy egy fa bármely pontja, ha őt gyökérpontnak tekintjük, szintén meghatároz egy fát, amely a pontot, mint gyökérpontot és a belőle „lefelé elérhető” pontokat tartalmazza. Ezt a fát az eredeti fa *részfájának* nevezzük.

Rendezett fa az olyan fa, amelyben a pontok leszármazottai között sorrendet értelmezünk, azaz beszélünk első, második, stb. leszármazottról,

vagy ha két leszármazott van, akkor például bal oldali és jobb oldali leszármazottról.

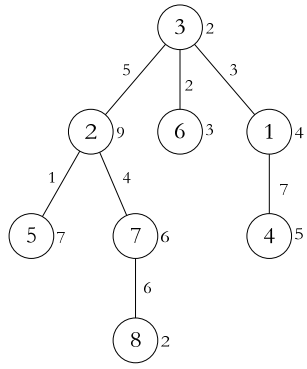
Megjegyzés: Ez a fogalom a leszármazottak beazonosításához szükséges és semmi köze a pontokhoz vagy élekhez rendelt értékekhez.

A pontokhoz és élekhez adatokat rendelve a fa egy széleskörűen alkalmazott, nagy hatékonyságú eszköz a rendezési és keresési feladatok megoldásánál. Ezeket legegyszerűbben bináris fákon tudjuk szemléltetni. Rendeljünk a pontokhoz olyan értékeket, amelyek egymással összehasonlíthatók (pl. számot, sztringet), nevezzük ezt pontértéknek.

15.3.1. Tárolás

A pont és éljellemzőkkel felszerelt fák is hálózatok, tehát elvben alkalmazhatnánk a korábban tárgyalt általános módszereket (lásd 15.2.), viszont a fatulajdonságok kihasználása lényegesen tömörebb és az algoritmusokkal könnyebben kezelhető implementációkat is lehetővé tesz.

Minden gyökeres fát ábrázolhatunk *címketömbbel*. Egy pont címkéjén az őt a fában megelőző pont azonosítóját értjük. A gyökérpontra ez nem értelmezett, de minden más pontnak van egyértelmű címkéje. Mivel a címke egyértelműen a pontokhoz rendeli az éleket is (a pontot a szülőjével összekötő élt a ponthoz), így mind az él-, mind a pontjellemzőket megadhatjuk ugyanebben a rendszerben.



15.11. ábra. Példa a címketömbös tároláshoz.

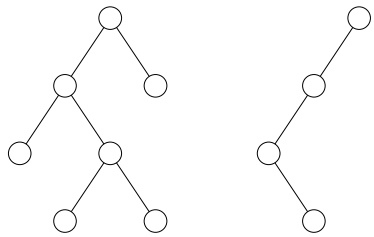
Pont	1	2	3	4	5	6	7	8
Címke	3	3	-	1	2	3	2	7
Pontjellemző	4	9	2	5	7	3	6	2
Éljellemző	3	5	-	7	1	2	4	6

15.12. ábra. Címketömb.

Egy pontjellemzőt (az 15.11. ábrán a pont melletti számok) és egy éljellemzőt (az élek melletti számok) vettünk fel, a pontokat egyszerűen a sorszámukkal azonosítva.

15.3.2. Bináris fák

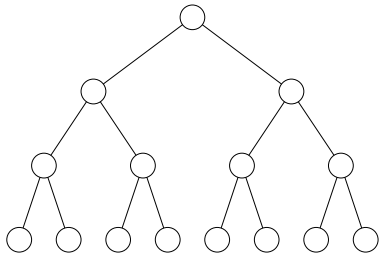
Bináris fa az olyan rendezett fa, amelyben egy pontnak legfeljebb két leszármazottja van. Az egyik leszármazottat *bal* leszármazottnak, az általa meghatározott részfát *bal részfának*, míg a másikat *jobb* leszármazottnak, az általa meghatározott részfát *jobb részfának* nevezzük. A definíció értelmében tehát egy-egy konkrét pontnál akár az egyik, akár a másik, akár mind a kettő hiányozhat.



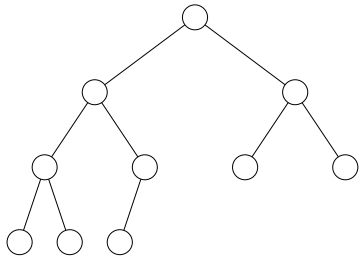
15.13. ábra. Bináris fák.

Teljes bináris fa az olyan bináris fa, amelyben az utolsó szinten lévő pontokat kivéve minden pontnak megvan mind a két leszármazottja, és az utolsó szinten csak levelek vannak.

Majdnem teljes bináris fa az olyan bináris fa, amely szintenként lefelé, egy adott szinten belül pedig balról-jobbra haladva „feltöltött” elemekkel. Tehát legfeljebb az utolsó szint jobb szélén lehet „üres”, ha nincs annyi elem, hogy a fa teljes legyen.

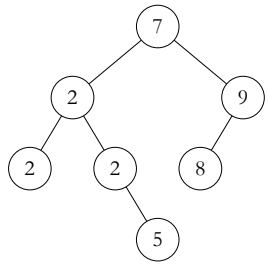


15.14. ábra. Teljes bináris fa.



15.15. ábra. Majdnem teljes bináris fa.

Bináris keresőfának nevezünk egy olyan bináris fát, amelyben minden pontra igaz, hogy a ponthoz, mint gyökérponthoz tartozó *bal részfa* pontértékei *nem nagyobbak,* a *jobb részfa* pontértékei *nem kisebbek* a pont értékénél.



15.16. ábra. Bináris keresőfa.

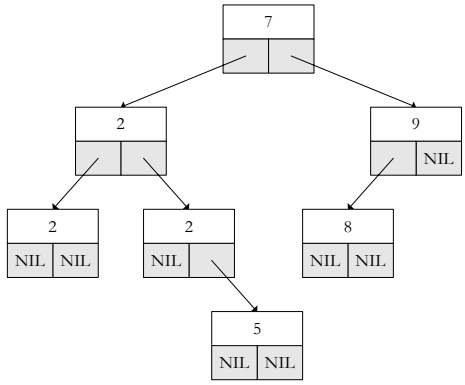
Bináris fák tárolására jól alkalmazhatók a dinamikus adatszerkezetek is. Pontonként egy rekordot veszünk fel, ebben tároljuk a pontjellemzőket, élenként az éljellemzőket, valamint a megfelelő követőkre (leszármazottakra) mutató mutatókat. A nemlétező követőket a mutatók NIL értéke jelzi.

A bináris fa deklarációs sémája (csak pontjellemzővel):

BINFAPONT Rekord

PONTJELL	Pontjellemző típus
BALAG, JOBBAG	BINFAPONT rekordra mutató

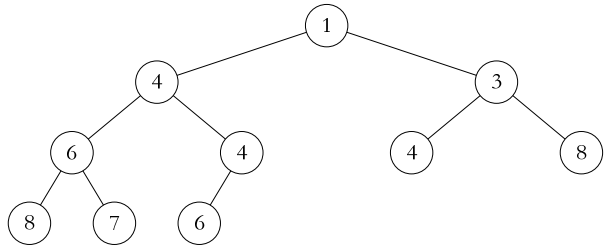
A deklarált BINFAPONT rekordban az egyszerűség kedvéért most egyetlen pontjellemzőt deklaráltunk, a pont esetlegesen létező további jellemzői ill. a bal és jobb él jellemzői egy-egy újabb adatmezőt jelentenének a rekordban.



15.17. ábra. Dinamikus adatszerkezettel megadott bináris fa.

Az előző példa a 15.16. ábra bináris keresőfáját mutatja dinamikus adatszerkezettel megvalósítva. Bár ez a szerkezet egyértelműen leírja a fát és biztosítja a gyökérpont felőli fabejárást, szükség esetén bővíthető még a szülőre mutató mutatóval is.

Bináris kupacnak vagy röviden *kupacnak* nevezünk egy olyan, majdnem teljes bináris fát, amelyben minden pontra igaz, hogy a ponthoz, mint gyökérponthoz tartozó *részfa* pontértékei *nem kisebbek* a pont értékénél.



15.18. ábra. Bináris kupac.

Megjegyzés

- Ez a definíció a növekvő rendezési iránynak felel meg, így a fa gyökérpontjának értéke minimális a fában. Hasonlóan definiálható a másik rendezési iránynak megfelelő kupac, ha azt kötjük ki, hogy a részfa pontértékei *nem nagyobbak* a pont értékénél.
- A bináris keresőfánál ilyen kettősség nincs, ugyanaz a fa az elemek csökkenő sorrendjét is „tartalmazza", hasonlóan a rendezett tömbökhöz és a kétirányban láncolt, rendezett listákhoz, csak az elemek elérését végző algoritmuson kell a megfelelő változtatásokat megtennünk.

A kupac adatstruktúra, mint speciális bináris fa, szabályosságánál fogva lehetővé tesz egy másfajta, bizonyos feladatoknál nagyon jól használható, tömbös implementációt is. Ennél csak pontjellemzőket kezelünk, azokat egy egydimenziós tömbben tárolva az alábbiak szerint:

- a gyökérpont indexe 1
- ha egy pont indexe *i*, akkor a bal gyermek a *2*i*, a jobb gyermek a *2*i+1* index alatt tárolódik.

Sorszám	1	2	3	4	5	6	7	8	9	10
Pontjellemző	1	4	3	6	4	4	8	8	7	6

15.19. ábra. Kupactömb.

Az egyszerűség kedvéért egyetlen pontjellemzőt deklaráltunk, ami egyben a pontérték is. A fenti ábra a 15.18. ábrabeli kupac, tömbben való tárolását szemlélteti. Több pontjellemző esetén vagy újabb egydimenziós tömböket használunk, vagy rekordba foglaljuk a pontjellemzőket, így egyetlen tömb is elegendő a kupac tárolására.

A definícióból és a tárolásból fakadóan a tömb első eleme mindig minimális pontértékű pont. Mind a beszúrás, mind a törlés „gyorsan" végrehajtható, hiszen ha a pontok száma *n*, akkor a fa magassága $log_2(n)+1$ egészrésze, így mind a két művelet ezzel arányos számú művelettel végrehajtható (lásd 15.3.2.4.).

Ez az adatstruktúra kifejezetten előnyös olyan feladatoknál, amelyeknél a kezelendő adathalmaz sűrűn változik, viszont a lekérdezés csak a minimális vagy maximális értékre vonatkozik. Ha összevetjük egy rendezett tömbbel vagy listával, láthatjuk, hogy a minimális vagy maximális érték ezeknél is közvetlenül rendelkezésre áll, de a beszúrás és a törlés művelet-

igénye magával az elemszámmal, nem pedig annak kettesalapú logaritmusával arányos. (A tömbben ugyan kereshetünk binárisan, de az elemeket mozgatni kell, a listában nem kell az elemeket mozgatni, viszont csak lineárisan kereshetünk.)

Az is észrevehető, hogy tetszőleges érték keresésénél már elveszik a kupac előnye a rendezett tömbbel szemben, hiszen bináris keresésre nem alkalmas, a belső elemeket a kupacban csak lineárisan lehet keresni.
Az előzőek alapján a bináris kupacok egy másfajta definíciója is adható:

- Az a_l, a_{l+1}, a_{l+2}, … a_r elemek sorozata rendelkezik a *kupactulajdonsággal*, ha minden a_i elemére teljesül, hogy:
  - $a_i <= a_{2*i}$, ha $2*i \in \{l, l+1, \ldots, r\}$ és
  - $a_i <= a_{2*i+1}$, ha $2*i+1 \in \{l, l+1, \ldots, r\}$.
- Ha $l=1$, azaz az első elem 1-es indexű, akkor az elemek bináris kupacot alkotnak, ahol a legkisebb elem éppen a_1.

A fák kezelését két feladatcsoporttal szemléltetjük. Az egyik a dinamikus adatszerkezettel megvalósított *bináris fára*, a másik *tömbben tárolt kupacra* vonatkozik.

Keresés keresőfában

Feladat: Keressünk meg egy értéket egy (ezen érték szerinti) bináris keresőfában.

Megoldás: A keresés módja a fa definíciójából közvetlenül adódik:

- Elindulunk a gyökérpontból.
- Minden érintett pontra elvégezzük az alábbiakat:
  - Ha a pontérték egyenlő a keresett értékkel, akkor készen vagyunk.
  - Ha a pontérték kisebb a keresett értéknél, akkor balra lépünk, a következő pont a bal leszármazott lesz.
  - Ha a pontérték nagyobb a keresett értéknél, akkor jobbra lépünk, a következő pont a jobb leszármazott lesz.

Az eljárás nagyon hasonlít a tömbben való bináris kereséshez, hiszen itt is minden lépésben kizárjuk az adathalmaz egyik részét (de nem biztosan a felét, mint a bináris keresésnél). Minden lépésben egy szinttel lejjebb kerülünk, tehát a keresés annál gyorsabb, minél kisebb a fa magassága. Szemlélet alapján is adódik, hogy a magasság akkor lesz minimális, ha a fa *kiegyensúlyozott*, vagyis minden pontjára, mint gyökérpontra teljesül az, hogy a bal részfa

és a jobb részfa pontjainak száma közötti eltérés minimális, azaz legfeljebb 1. Ez esetben a keresés ugyanolyan gyors, mint a bináris keresés. Ha a fa ehhez képest nagyon „torz”, akkor a keresés akár lineárissá is „fajulhat”.

Típus

BINFAPONT Rekord

PONTJELL	Pontjellemző típus
BALAG, JOBBAG	BINFAPONT rekordra mutató

Funkció	Azonosító	Típus	Jelleg
A fa gyökérpontja	GYOKER	BINFAPONT-ra mutató	I
A keresett érték	X	Pontjellemző típus	I
Az éppen vizsgált pont	AKT	BINFAPONT-ra mutató	M, O

```
/* Keresés bináris keresőfában */
KERES(GYOKER,X)

AKT ← GYOKER
while (AKT<>NIL) AND (X<>(*AKT).PONTJELL)
      if X<(*AKT).PONTJELL
            AKT ← (*AKT).BALAG
      else
            AKT ← (*AKT).JOBBAG
return AKT
```

Eredményül a keresett elem címét, vagy NIL-t kapunk, ha a keresett érték nem szerepel a fában.

Keresőfa bővítése

Feladat: Bővítsünk egy keresőfát egy új ponttal!

Megoldás: A keresőfában az új pontot a megfelelő helyre kell beilleszteni, ezt a helyet kell megkeresnünk a keresőfa definíciója alapján, tehát azt a pontot, mint szülőpontot, amelynek gyerekeként (és egyben levélként) be kell illesztenünk az új pontot.

Funkció	Azonosító	Típus	Jelleg
A fa gyökérpontja	GYOKER	BINFAPONT-ra mutató	I, O
Az új pont	X	BINFAPONT-ra mutató	I
Az éppen vizsgált pont	AKT	BINFAPONT-ra mutató	M
A szülő pont	SZULO	BINFAPONT-ra mutató	M

```
/* Bináris keresőfa bővítése */
BOVIT(GYOKER,X)

if GYOKER=NIL
       /* Üres fa */
       GYOKER ← X
else
       /* Helykeresés */
       AKT ← GYOKER
       SZULO ← NIL
       while AKT<>NIL
              SZULO ← AKT
              if (*X).PONTJELL<(*AKT).PONTJELL
                     AKT ← (*AKT).BALAG
              else
                     AKT ← (*AKT).JOBBAG
       /* Beillesztés */
       if (*X).PONTJELL<(*SZULO).PONTJELL
              (*SZULO).BALAG ← X
       else
              (*SZULO).JOBBAG ← X
```

Feltesszük, hogy az új ponthoz tartozó rekord már létezik, mezői definiáltak (PONTJELL a megfelelő értékkel, a BALAG és JOBBAG mezők NIL értékkel), és az erre a rekordra mutató mutatót kapjuk az X változóban.

A gyökérpont output jellege a fa módosulását fejezi ki, üres fa esetén maga a gyökérpont közvetlenül is módosul.

Fabejárás

Feladat: Írjuk ki egy bináris fa összes elemét a képernyőre!

Megoldás: Sok feladat csak úgy oldható meg, hogy a fa pontjait (az összes pontot, vagy valamilyen feltétel teljesüléséig az összes pontot) valamilyen rendszer szerint egyenként megvizsgálunk. Ezt másképpen úgy mondjuk, hogy a fát be kell járnunk. A *fabejárások* tipikusan *rekurzív* algoritmusok. Sokféle fabejárás lehetséges, néhány nevezetes alapmódszer:

- *Inorder* bejárás: a gyökérpontot a két részfa között érintjük.
- *Preorder* bejárás: a gyökérpontot a részfák előtt érintjük.
- *Posztorder* bejárás: a gyökérpontot a részfák után érintjük.

Most egy olyan rekurzív, inorder fabejárást adunk meg, amelyben balról-jobbra haladunk, azaz a bal részfát vesszük előre. Ha tehát a kiírandó fa éppen egy növő sorrendű keresőfa, akkor az elemeket növő (a jobb részfát előrevéve csökkenő) sorrendben kapjuk meg.

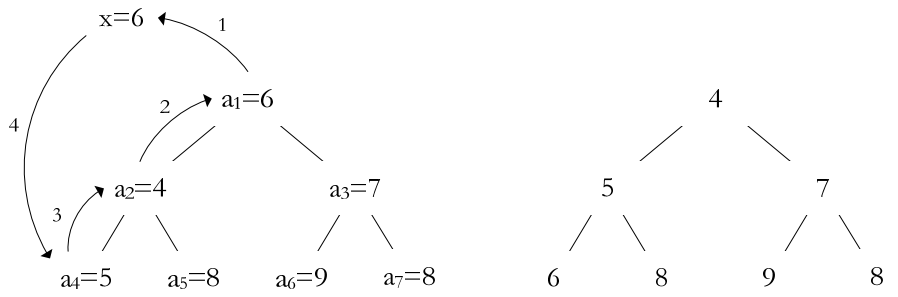
Funkció	Azonosító	Típus	Jelleg
A fa gyökérpontja	GYOKER	BINFAPONT-ra mutató	I

```
/* Bináris fa elemeinek kiírása rekurzív eljárással */
KIIR(GYOKER)

if GYOKER<>NIL
       /* Rekurzív hívás a bal ágra */
       KIIR((*GYOKER).BALAG)
       /* A gyökérponthoz tartozó érték kiírása */
       Ki: (*GYOKER).PONTJELL
       /* Rekurzív hívás a jobb ágra */
       KIIR((*GYOKER).JOBBAG)
```

Kupactulajdonságú elemek bővítése

Feladat: Bővítsük az a_{l+1}, a_{l+2}, …, a_r kupactulajdonsággal (lásd 15.3.2.) rendelkező elemeket egy új elemmel (a_l) úgy, hogy a kupactulajdonság megmaradjon!

Megoldás: A kupactulajdonság szerint minden elem (a_i) kisebb vagy egyenlő a gyerekeinél (a_{2*i}, a_{2*i+1}). Ezt megtartandó, az új elemet (a_l) „besüllyesztjük" a kupacba. Mindaddig, amíg az új elem nagyobb, mint a kisebbik gyereke (egy gyerek esetén a gyerekénél), addig felcseréljük őket. A „nagy" elem szemléletesen a „kicsi" elem helyére süllyed egészen addig, amíg a helyére nem kerül. Hogy az elemek cseréjéhez szükséges értékadások számát csökkentsük, nem felcserélni fogjuk az elemeket, hanem a gyerekek megfelelő „feljebb léptetésével" helyet készítünk a „süllyedő", új elem számára. Az 15.20. ábra szemlélteti a 6-os elem helyére süllyedését.



15.20. ábra. Elem besüllyesztése.

Típus
ELEM Egész /* Az elemek típusa */
TOMB Egydimenziós ELEM tömb /* Az elemeket tároló tömb típusa */

Az elemek típusaként egészeket használunk, de megjegyezzük, hogy tetszőleges olyan típust használhatunk, amelyre értelmezettek a hasonlítás műveletek.

Funkció	Azonosító	Típus	Jelleg
Az elemek	A	TOMB	I, M, O
A besüllyesztendő elem indexe	L	Egész	I
Az utolsó elem indexe	R	Egész	I
A besüllyesztendő elem tárolására	X	ELEM	M
A vizsgált elem indexe	I	Egész	M
A kisebbik gyerek indexe	J	Egész	M
Helyére süllyedt-e az elem	VEGE	Logikai	M

```
/* A[L] besüllyesztése az A[L+1],...,A[R] elemek közé */
SULLYESZT(A,L,R)

X ← A[L]
I ← L
J ← 2*I
VEGE ← hamis
/* Helykészítés */
while (J<=R) AND NOT VEGE
       /* A kisebb gyerekkel hasonlítsunk */
       if (J<R) AND (A[J+1]<A[J])
              J ← J+1
       if X<=A[J])
              /* Megvan a helye */
              VEGE ← igaz
       else
              /* A gyerek feljebb léptetése */
              A[I] ← A[J]
              I ← J
              J ← 2*I
/* Elemet a helyére */
A[I] ← X
```

Kupacrendezés

Feladat: Bináris kupac felhasználásával készítsünk rendező algoritmust!

Megoldás:

- Először képezzünk kupacot a rendezendő elemek a_1, a_2, …, a_n sorozatából. Legyen $l=n \text{ div } 2$ és $r=n$. Ekkor az a_{l+1}, a_{l+2}, …, a_r elemek rendelkeznek a kupactulajdonsággal, hiszen nincsenek gyerekeik. Az előző fejezetben kifejtett SULLYESZT eljárás segítségével bővíteni tudjuk az elemeket az a_l elemmel, majd l értékét rendre eggyel csökkentve a többi elemmel $(a_{l-1}, \dots, a_1)$ is.
- A kupactulajdonság miatt a legkisebb elem a_1. Tegyük ezt a legutolsó elem (a_n) helyére, azt pedig, mint a_1 süllyesszük be az a_2, …, a_{n-1} elemek közé. Ennek eredményeképpen újra a legkisebb elem kerül a legelső helyre, amely immár a rendezendő adatok második legkisebb eleme lesz. Ezeket a kicsi elemeket rendre hátratéve (az r. helyre), a kupac utolsó elemének indexét (r) csökkentve, összesen $n-1$ lépés után előáll a rendezettség.

Megjegyzés

- Az algoritmus csökkenő (nem növő) sorrendet készít (hiszen a legkisebb elem kerül leghátra), de a SULLYESZT eljárásban a megfelelő hasonlítás megfordításával (a kicsi elemek süllyesztésével) növő rendezést kapunk.
- Tekintettel a kupac definíciójára az algoritmus során minden elem $log_2 n$ lépésben a helyére kerül, így az algoritmus műveletigénye $n*log_2 n$ nagyságrenddel jellemezhető.

Funkció	Azonosító	Típus	Jelleg
A rendezendő elemek	A	TOMB	I, M, O
Az elemek száma	N	Egész	I
A kupac első elemének indexe	L	Egész	M
A kupac utolsó elemének indexe	R	Egész	M
Két elem cseréjéhez	X	ELEM	M

```
/* Kupacrendezés */
KUPACREND(A,N)
/* Kezdőkupac */
L ← N DIV 2+1
R ← N
while L>1
       L ← L-1
       SULLYESZT(A,L,R)
while R>1
       /* Legkisebbet hátra */
       X ← A[1]
       A[1] ← A[R]
       A[R] ← X
       /* Kupacot kisebbre */
       R ← R-1
       /* A hátul volt elem besüllyesztése */
       SULLYESZT(A,L,R)
```

Megjegyzés: A SULLYESZT eljárás meghívása előtt csökkentjük L értékét (ezért az N DIV 2+1 kezdőértékről indítjuk), mert így a kezdőkupac előállítása (első ciklus) után L értéke 1 lesz (és később már nem is változik meg), amit a SULLYESZT eljárás későbbi (második ciklusbeli) hívásai ki is használnak.

15.4. Útkeresés

A gyakorlati alkalmazásokban az egyik legfontosabb, legtöbbször használt hálózati algoritmus a bizonyos szempontból „legkedvezőbb útvonal" keresése. Természetesen ezt a fogalmat pontosítanunk kell ahhoz, hogy algoritmusokat adhassunk a meghatározására. E célból bevezetünk néhány új fogalmat.

Amint azt a bevezetőrészben (lásd 15.1.) már definiáltuk, a természetes szemlélettel megegyezően, *út* a gráf egy olyan pontsorozata, amelynek a felsorolás sorrendjében szomszédos pontjai között van a megelőző pontból a következő pontba mutató él. Nevezzük el az út első pontját *kezdőpontnak*, az utolsó pontját *végpontnak*. Egy kezdőpont-végpont párt *viszonylatnak* nevezünk. Egy gráfban általában több út is létezik egy adott viszonylatban (általában minél nagyobb a gráf, annál több a lehetséges utak száma).

Válasszuk ki az élek valamely jellemzőjét (pl. az él hosszát, vagy az él megtételéhez szükséges időt, stb.), ami szerint optimalizálni szeretnénk és

nevezzük ezt *élhossznak*. Tegyük fel, hogy az élhosszak pozitív (0-nál nagyobb) mérőszámok.

Egy *út hosszán*, az úton lévő, utat alkotó élek hosszainak összegét értjük. Ez alapján rangsorolhatók az egy adott viszonylathoz tartozó utak.

Minimális útnak nevezzük egy adott viszonylat útjai közül azt, amelyiknek a hossza minimális.

Megjegyzés
- Minimális hosszú útból egy adott viszonylatban több is létezhet.
- A pozitív élhosszak miatt az úthossz is pozitív (0-nál nagyobb), így a minimális utak szükségszerűen körmentesek (nincs az úton ismétlődő pont), hiszen egy kör (az ismétlődő pont közötti rész) levágása csökkenti a hosszt.

Hasonlóan a visszalépéses algoritmusokhoz (lásd 11.6.), a minimális út keresése is jól demonstrálja azt, hogy bizonyos keresési, optimalizálási problémák olyan „egyszerű megközelítése", hogy „vegyük az összes lehetséges esetet (utat), válasszuk ki ezek közül a legjobbat (a minimális hosszút)", gyakorlatilag használhatatlanok az esetek nagy száma miatt. (Egy adott viszonylathoz, rögzítve a kezdő- és végpontot, a többi pont összes lehetséges részhalmazát és az egyes részhalmazokba eső pontok összes lehetséges sorrendjét kéne képezni, majd eldönteni, hogy út-e, s ha igen mennyi a hossza. Ez egy 15 pontos hálózat esetén is több milliárd esetet jelentene.)

Bemutatunk két különböző, a témakörben klasszikusnak számító konkrét eljárást a minimális utak keresésére.

15.4.1. Mátrix módszer

Az alább ismertetetendő eljárás Floyd-Warshall algoritmus néven ismert [Cor 97]. Akkor alkalmazzuk, ha minden viszonylatra (vagy legalábbis a viszonylatok nagy részében) meg akarjuk határozni a minimális utat és van elegendő memóriánk az algoritmushoz szükséges két darab, *pontszám*pontszám* méretű mátrix tárolására.

A *T* távolságmátrix a a viszonylatok *aktuális távolságát*, azaz a viszonylathoz tartozó aktuális minimális út *hosszát* tartalmazza úgy, hogy *T[x,y]* az *x-y* viszonylathoz tartozó távolságot jelenti.

A *C* címkemátrix elemei pontok (vagy pontindexek), amelyek a minimális utak összerakásához, azaz a megfelelő pontsorozatok előállításához szükséges adatokat tartalmazzák úgy, hogy *C[x,y]* az a pont, amely az *x-y*

viszonylat aktuális minimális útján az *x* kezdőpont után jön (merre induljunk az *x* kezdőpontból az *y* végpont felé).

Az algoritmus a *T* és *C* mátrixok *kezdőállapotából* kiindulva, a mátrixokat *lépésenként javítva*, több lépés megtétele után jut el a végeredményt adó *végállapotba*. Mint látni fogjuk, pontosan annyi lépés kell, ahány pont van a hálózatban.

Kezdőállapot

T: *T[x,y]* legyen az *x*-*y* él hossza, ha létezik ilyen él, egyébként legyen „végtelen nagy”.

C: A viszonylatok címkéi legyenek a végpontok, azaz *C* első oszlopa tartalmazza az első pontot, második oszlopa a második pontot, és így tovább.

A *T* és *C* tartalma tehát kezdetben éppen az egy élű minimális utakat fejezik ki.

Javító lépések

A hálózat minden *w* pontjára (egyszer) és ezen belül minden *x*-*y* viszonylatra (egyszer) vizsgáljuk meg azt, hogy az *x*-*y* viszonylat aktuális minimális útja rövidíthető-e a *w* ponton való áthaladással.

Ha $T[x,w]+T[w,y] < T[x,y]$ teljesül (azaz *w*-n keresztül rövidítünk), akkor *T[x,y]* legyen $T[x,w]+T[w,y]$ és *C[x,y]* legyen *C[x,w]*. Azaz az *x*-*y* viszonylat minimális útját immár az *x*-*w* és *w*-*y* utak adják, tehát az *y*-hoz vezető minimális úton ugyanúgy kell indulni *x*-ből, mint a *w*-hez vezető minimális úton.

Végállapot

T végállapota megadja minden viszonylatra a minimális út hosszát. A „végtelen nagy” elemek azt fejezik ki, hogy az adott viszonylatokban nem létezik út.

C végállapota a megtalált minimális utakat tükrözi, definíciója alapján minden olyan *x*-*y* viszonylatra összerakható egy (az éppen megtalált) minimális út, amelyre *T[x,y]* nem „végtelen nagy”.

Az algoritmus számításigénye a pontszám köbével jellemezhető (lásd a javító lépések három ciklusa). A módszer kihasználja, hogy a *T* és *C* mátrixok elemei két index segítségével közvetlenül, egy lépésben elérhetők, így ezeket ténylegesen az operatív tárban, mátrixként kell tárolni, mert egyéb

megoldásoknál (pl. lemezes tárolás) az eljárás egyszerűségét és hatékonyságát veszti.

Megjegyzés

- A *T* távolságmátrix főátlójában található elemek a pontok „önmaguktól vett" távolságát fejezik ki. Ha ezen elemek kezdőértéke (kezdőállapot) „végtelen nagy", akkor az algoritmus végén (végállapot) az egyes pontokból kiinduló, önmagukba visszaérkező minimális utak távolságát kapnánk meg, ami a *C* címkemátrixból ugyanúgy összerakható, mint a többi út. Ha ezen elemek kezdőértéke 0 (mint a megoldásunkban), akkor ezen értékek már nem változnak meg az algoritmus során, kifejezve azt, hogy egy pont elérhető önmagából 0 távolsággal.
- A gráf éleinek hosszát mátrixként tároljuk (lásd 15.2.2.).

```
Konstans
    MAXPONTDB  5
    MAXELDB  MAXPONTDB*(MAXPONTDB–1)
    MAXELHOSSZ  99
    NINCSEL  –1
    MAXUTHOSSZ  MAXELDB*MAXELHOSSZ
    VEGTELEN  MAXUTHOSSZ+1
```

A gráfunk tehát most maximum 5 pontos lehet, ahol az egyes élek hosszai max. 99, pozitív egész értékek lehetnek, a nemlétező élt a –1 hosszúság fejezi ki. A „végtelen nagy" értéket tehát egy megfelelően nagy konstanssal kezeljük (az összes élhossz összegénél eggyel nagyobb szám már megfelelő, hiszen az útjaink körmentesek).

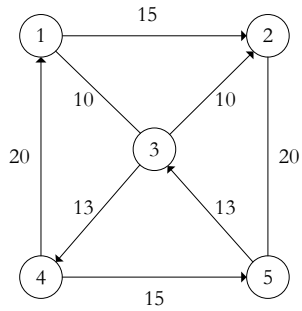
```
Típus
PONT  Rekord
    AZON        Karakter        /* A pont azonosítója */
    X, Y        Egész           /* Koordináták */
MATRIX  Kétdimenziós egész tömb [MAXPONTDB, MAXPONTDB]
GRAF  Rekord
    PONTDB      Egész           /* Pontok száma */
    PONTOK      Egydimenziós PONT tömb [MAXPONTDB]
    ELHOSSZ     MATRIX
```

Funkció	Azonosító	Típus	Jelleg
A hálózatot leíró gráf	G	GRAF	I
A *T* távolságmátrix	T	MATRIX	M, O
A *C* címkemátrix	C	MATRIX	M, O
Az aktuális viszonylat *x* kezdőpontja	X	Egész	M
Az aktuális viszonylat *y* végpontja	Y	Egész	M
A javító lépések *w* pontja	W	Egész	M

A *C* címkemátrix pontindexeket tartalmaz.

```
/* Mátrix módszer */
FLOYD_WARSHALL(G,T,C)

/* Kezdőállapot */
for X ← 1,G.PONTDB
      for Y ← 1,G.PONTDB
            if X=Y
                  T[X,Y] ← 0
            else
                  if G.ELHOSSZ[X,Y]=NINCSEL
                        T[X,Y] ← VEGTELEN
                  else
                        T[X,Y] ← G.ELHOSSZ[X,Y]
            C[X,Y] ← Y
/* Javító lépések */
for W ← 1,G.PONTDB
      for X ← 1,G.PONTDB
            for Y ← 1,G.PONTDB
                  if T[X,W]+T[W,Y]<T[X,Y]
                        T[X,Y] ← T[X,W]+T[W,Y]
                        C[X,Y] ← C[X,W]
```

Az algoritmus működése végigkövethető az alábbi példahálózaton.



15.21. ábra. Példahálózat.

T	1	2	3	4	5
1	0	15	10	-	-
2	-	0	-	-	20
3	10	10	0	13	-
4	20	-	-	0	15
5	-	20	13	-	0

C	1	2	3	4	5
1	1	2	3	4	5
2	1	2	3	4	5
3	1	2	3	4	5
4	1	2	3	4	5
5	1	2	3	4	5

15.22. ábra. Kezdőállapot.

T	1	2	3	4	5
1	0	15	10	-	-
2	-	0	-	-	20
3	10	10	0	13	-
4	20	35	30	0	15
5	-	20	13	-	0

C	1	2	3	4	5
1	1	2	3	4	5
2	1	2	3	4	5
3	1	2	3	4	5
4	1	1	1	4	5
5	1	2	3	4	5

15.23. ábra. Az 1. javító lépés ($w=1$) utáni állapot.

T	1	2	3	4	5
1	0	15	10	-	35
2	-	0	-	-	20
3	10	10	0	13	30
4	20	35	30	0	15
5	-	20	13	-	0

C	1	2	3	4	5
1	1	2	3	4	2
2	1	2	3	4	5
3	1	2	3	4	2
4	1	1	1	4	5
5	1	2	3	4	5

15.24. ábra. A 2. javító lépés ($w=2$) utáni állapot.

T	1	2	3	4	5
1	0	15	10	23	35
2	-	0	-	-	20
3	10	10	0	13	30
4	20	35	30	0	15
5	23	20	13	26	0

C	1	2	3	4	5
1	1	2	3	3	2
2	1	2	3	4	5
3	1	2	3	4	2
4	1	1	1	4	5
5	3	2	3	3	5

15.25. ábra. A 3. javító lépés ($w=3$) utáni állapot.

T	1	2	3	4	5
1	0	15	10	23	35
2	-	0	-	-	20
3	10	10	0	13	28
4	20	35	30	0	15
5	23	20	13	26	0

C	1	2	3	4	5
1	1	2	3	3	2
2	1	2	3	4	5
3	1	2	3	4	4
4	1	1	1	4	5
5	3	2	3	3	5

15.26. ábra. A 4. javító lépés (*w=4*) utáni állapot.

T	1	2	3	4	5
1	0	15	10	23	35
2	43	0	33	46	20
3	10	10	0	13	28
4	20	35	28	0	15
5	23	20	13	26	0

C	1	2	3	4	5
1	1	2	3	3	2
2	5	2	5	5	5
3	1	2	3	4	4
4	1	1	5	4	5
5	3	2	3	3	5

15.27. ábra. Az 5. javító lépés (*w=5*) utáni állapot, végállapot.

Az egyes lépésekben megváltozott értékeket kiemelés, a végtelen nagy értékeket a „-” jelzi. A végállapotból leolvasva, pl. a 2-4 viszonylat távolsága 46 és a hozzá tartozó minimális út: 2-5-3-4, hiszen C[2, 4]=5, C[5, 4]=3, C[3, 4]=4.

15.4.2. Faépítés

Ha a hálózatunk a tárkapacitásunkhoz képest nagy, vagy csak a viszonylatok kisebb részében akarunk minimális utakat keresni, akkor alkalmazhatjuk az ún. *faépítő* eljárásokat. Ezek nem az összes viszonylatra, hanem csak az *azonos kezdőpontú* viszonylatokra – tehát egy kezdőpontból az összes többi pontba, mint végpontba – határozzák meg a minimális utakat. Az ilyen utak egy fa struktúrájú részgráfot alkotnak a hálózatban, a kezdőponttal, mint gyökérponttal. Ezt az ún. *minimális fát* konstruálja meg, építi fel több lépésben az eljárás, innen ered az elnevezése.

Többféle faépítő módszer ismert. Itt egy olyat mutatunk be, amelynek működése könnyen érthető, és a módszeren alapuló konkrét algoritmusok a gyakorlatban is jól alkalmazhatók. Az alapeljárást a szakirodalom – első leírójáról – Dijkstra-féle eljárásnak nevezi.

Az eljárás során keletkező fákat egy *C címketömbbel* reprezentáljuk (lásd 15.3.1.). Az egyszerűbb kezelhetőség kedvéért a gyökérpontnak is címkét adunk, mégpedig saját magát. A címketömb az eljárás során felépülő minimális fát, annak aktuális állapotát írja le. Elemei *pontok* (vagy pontindexek), amelyekből a minimális utak összerakhatók. Egy *x* pont címkéje az *x* pont elődje a minimális fában, azaz az a pont ahonnan az *x* pontba jöttünk a minimális úton.

A címketömb mellé felveszünk egy hasonlóan indexelt (*pontszám* méretű) *T távolságtömböt* is, amelynek elemei megadják az egyes pontok aktuális távolságát a kezdőponttól, azaz az aktuális minimális fában hozzájuk vezető utak hosszát.

Vezessük be az alábbi jelöléseket:

- *K*: a kész pontok halmaza, elemei már végleges (nem rövidíthető) távolsággal és végleges címkével rendelkeznek.
- *A*: az aktív pontok halmaza, elemei már rendelkeznek távolsággal és címkével, de ezek még változhatnak.
- *H[x,y]*: az *x*-*y* él (azaz az *x* kezdő- és *y* végpontú él) hossza.

Az algoritmus *T* és *C* kezdőállapotából kiindulva, azok elemeit (tehát magát a fát) *lépésenként építve* és korrigálva, pontszám lépés megtétele után jut el a végeredményt adó *végállapotba*.

Kezdőállapot

T: A kezdőponthoz rendeljünk 0, a többi ponthoz „végtelen nagy” távolságértékeket.

C: A kezdőpont címkéje legyen önmaga.

K: Legyen üres.

A: Csak a kezdőpontot tartalmazza.

A kezdőállapot valójában az egy pontból, mint gyökérpontból álló fának felel meg.

Javító lépések

a) Válasszuk ki az *A* halmaz minimális távolságú elemét, jelölje ezt *x*. Töröljük *A*-ból és vegyük hozzá *K*-hoz (ez már kész, nem változik).

b) Az *x*-ből kiinduló élek minden *y* végpontjára vizsgáljuk meg azt, hogy *y* útja rövidíthető-e, az *x* ponton való áthaladással. Ha igen, akkor a pont

címkéjét x-re, távolságát rövidebbre állítjuk, és (ha még nem volt bent) hozzávesszük az A halmazhoz. Ha tehát y még nem volt a fában, akkor x elődjével bekerül, ha már bent volt, akkor elődjét x-re cseréljük. Képletekkel: ha $T[x]+H[x,y]<T[y]$ teljesül (azaz x-en keresztül rövidítünk), akkor $T[y]$ legyen $T[x]+H[x,y]$, $C[y]$ legyen x, és A legyen $A+[y]$.

c) Ha van még aktív elem, azaz ha A nem üres, akkor folytassuk a *b)* ponttól.

Végállapot

Ha már nincs aktív pont, akkor az eljárás véget ért, a minimális fa készen van, a T és C tömbök meghatározzák a végeredményt.

T végállapota minden pontra, mint végpontra, megadja a kezdőpontból hozzávezető minimális út hosszát. A „végtelen nagy" érték azt fejezi ki, hogy az adott pont nem érhető el a kezdőpontból.

C végállapota a megtalált minimális utakat, a minimális fát tükrözi, definíciója alapján minden olyan y végpontra összerakható (a végpontból visszafelé haladva) a minimális út, amelyre a $T[y]$ nem „végtelen nagy".

Megoldásunkban a gráf éleit egydimenziós tömbben tároljuk (lásd 15.2.3.). A szükséges típusok deklarálásánál felhasználjuk az előző fejezetben bevezetett konstansokat és a PONT rekordtípust. Az aktív pontok A halmazát nem halmazban, hanem egydimenziós tömbben tároljuk, így a pontok száma 256-nál nagyobb is lehet.

```
Típus
EL  Rekord
    VP          Egész                   /* Az él végpontjának indexe */
    ELHOSSZ     Egész                   /* Az él hossza */
GRAF  Rekord
    PONTDB      Egész                   /* Pontok száma */
    PONTOK      Egydimenziós PONT tömb [MAXPONTDB]
    ELMUT       Egydimenziós egész tömb [MAXPONTDB+1]
    ELDB        Egész                   /* Élek száma */
    ELEK        Egydimenziós EL tömb [MAXELDB]
```

Az élmutatók tömbjét (ELMUT) eggyel nagyobb méretűre deklaráltuk, mint a PONTOK tömböt, hogy megkönnyítsük a pontok éleinek kezelését, hiszen az ELMUT[PONTDB+1] értékét ELDB+1 értékre állítva az utolsó pont is ugyanúgy kezelhető, mint a többi. Nevezetesen az I. pontból, mint kezdőpontból kimenő élek, az ELEK tömb ELMUT[I]-edik

elemétől az ELMUT[I+1]–1-edik eleméig tart (ha ez a két érték azonos, akkor az I. pontnak nincs kimenő éle).

Funkció	Azonosító	Típus	Jelleg
A hálózatot leíró gráf	G	GRAF	I
A *T* távolságtömb	T	Egydimenziós egész tömb	M, O
A *C* címketömb	C	Egydimenziós egész tömb	M, O
A kezdőpont indexe	KP	Egész	I
Az aktív pontok indexei	A	Egydimenziós egész tömb	M
Az aktív pontok száma	ADB	Egész	M
Az aktivitásjelző	AKT	Egydimenziós logikai tömb	M
Segédváltozók	X, Y, I, K	Egész	M

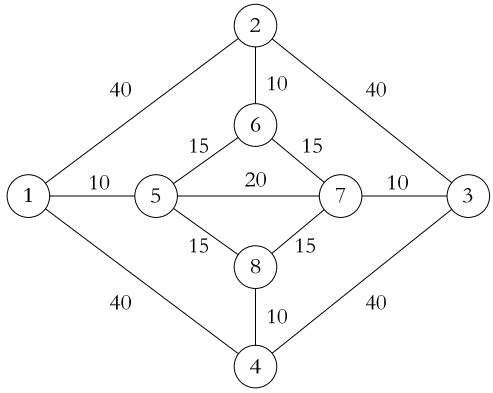
A T, C, A, AKT tömbváltozók egyaránt MAXPONTDB méretűek, az aktív pontokat tároló A és a címkéket tároló C tömbváltozók pontindexeket tartalmaznak.

```
/* Faépítés */
DIJKSTRA(G,T,C,KP)

/* Kezdőállapot */
for I ← 1,G.PONTDB
       T[I] ← VEGTELEN
       AKT[I] ← hamis
T[KP] ← 0
C[KP] ← KP
ADB ← 1
A[ADB] ← KP
AKT[KP] ← igaz
/* Javító lépések */
while ADB>0
       /* Az A minimális távolságú elemét X-be */
       K ← 1
       for I ← 2,ADB
              if T[A[I]]<T[A[K]]
                     K ← I
       X ← A[K]
       /* X törlése A-ból (az utolsó elemmel felülírjuk) */
       A[K] ← A[ADB]
       ADB ← ADB-1
       AKT[X] ← hamis
       /* Rövidítés X-en keresztül */
       for I ← G.ELMUT[X],G.ELMUT[X+1]-1
              Y ← G.ELEK[I].VP
```

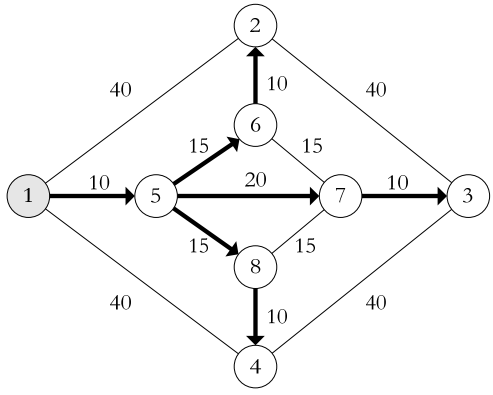
```
if T[X]+G.ELEK[I].ELHOSSZ<T[Y]
      T[Y] ← T[X]+G.ELEK[I].ELHOSSZ
      C[Y] ← X
      if NOT AKT[Y]
            /* Y nem aktív, hozzávesszük A-hoz */
            ADB ← ADB+1
            A[ADB] ← Y
            AKT[Y] ← igaz
```

Megjegyezzük, hogy *K* halmazt csak a jobb szemléltetés kedvéért használtuk, az algoritmusból el is hagyható (ahogy tettük ezt a megoldásunkban).

Az algoritmus működését az alábbi példahálózaton szemléltetjük, az 1. pontból, mint kezdőpontból kiindulva.



15.28. ábra. Példahálózat a faépítéshez.



15.29. ábra. Az 1 gyökérpontú minimális fa.

	1	2	3	4	5	6	7	8
T	0	35	40	35	10	25	30	25
C	1	6	7	8	1	5	5	5

15.30. ábra. A távolság- és címketömb.

A végállapotból (15.30. ábra) leolvasva, pl. az 1-4 viszonylat távolsága 35 és a hozzá tartozó minimális út: 1-5-8-4, hiszen C[4]=8, C[8]=5, C[5]=1.

Az első három javító lépés utáni állapotokat a 15.31. ábra, az aktuális minimális fát a 15.32. ábra mutatja. A fa kész pontjai már végleges távolsággal és címkével rendelkeznek, míg a többi pont távolsága és elődje, így a fa szerkezete még módosulhat. Összevetve a végállapottal (15.30 ábra) látható, hogy az eljárás hátralévő részében például módosul a 4. pont távolsága és elődje.

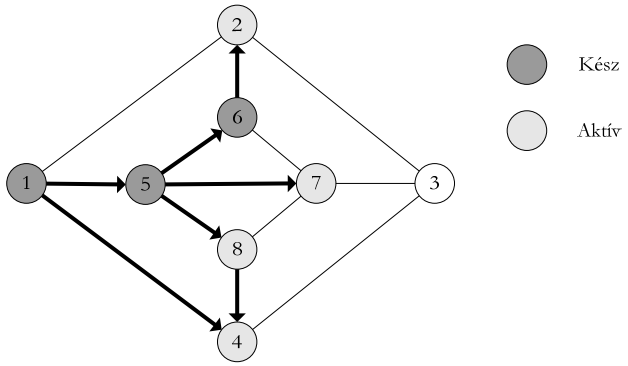
1	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
K	1							
A	2	4	5					
T	0	40	-	40	10	-	-	-
C	1	1		1	1			

2	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
K	1	5						
A	2	4	6	7	8			
T	0	40	-	40	10	25	30	25
C	1	1		1	1	5	5	5

3	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
K	1	5	6					
A	2	4	8	7				
T	0	35	-	40	10	25	30	25
C	1	6		1	1	5	5	5

15.31. ábra. Az első három javító lépés ($x=1,5,6$) utáni állapotok.

A javító lépések sorszámát kiemelés a végtelen nagy értékeket a „-” jelzi.



15.32. ábra. Az első három javító lépés utáni minimális fa.

Az eljárást *tárigény* szempontjából elemezve láthatjuk, hogy a végrehajtáshoz 3 db, egyenként *pontszám* elemű tömb szükséges (feltéve, hogy az aktív pontok halmazát is egy tömb tárolja).

Ez lehetővé teszi az alkalmazását nagy (pl. tízezer pontos) hálózatoknál is. A faépítés jól illeszkedik az élek tömör tárolási módjaihoz (lásd 15.2.3., 15.2.4.), hiszen az egy kezdőpontból kiinduló élek (amiket a javító lépések *b)* pontjában sorra kell vennünk) egymás után következnek, azaz gyorsan elérhetők.

Az eljárás *számításigénye* a pontszám négyzetével arányos, mivel minden lépésben (az *a)* pontban) átkerül egy pont a kész halmazba (külső ciklus), és ezen belül a *b)* pontban két, maximum pontszám lépésszámú ciklus található (egyik a minimumkiválasztás, a másik a pont éleit veszi sorra). Tehát ha az összes viszonylat minimális útját meg akarjuk határozni (azaz minden pontra, mint kezdőpontra felépítjük a minimális fát), akkor sem rosszabb nagyságrendben, mint a mátrixos módszer. Nagy elemszámú aktív halmazok esetén a minimumkiválasztás akár pontszám műveletigényű is lehet, ezt a kiválasztást meggyorsítandó, használhatunk kupac adatstruktúrát (lásd 15.3.2.) is, az aktív pontok tárolására. A minimumkiválasztás ekkor egy lépésben megtehető (hiszen ez a kupac legelső eleme), de ennek ára, a kupac-tulajdonság folyamatos megtartása, ami minden felvétel ill. törlés esetén az aktív pontok számának kettesalapú logaritmusával arányos műveletigényű.

Megjegyzés: A pontok koordinátái, mint pontjellemzők, csak informatív jelleggel szerepelnek a mellékelt példaprogramokban, az útkeresések szempontjából el is hagyhatók.

Gráfok számítógépes kezelésének további tanulmányozásához a [Mar 97] cikksorozatot, míg a fákkal kapcsolatos algoritmusok mélyebb és részletesebb megismerésére, vizsgálatára a szakirodalmat [Cor 97], [Wir 82], [Aho 82] ajánljuk.

15.5. Feladatok

- Oldjuk meg az alábbi, hálózatokra vonatkozó feladatokat az egyes éltárolási módszerek alkalmazásával:
  - A hálózat adatainak betöltése szövegfájlból.
  - A hálózat adatainak kimentése szövegfájlba.
  - A hálózati adatok (pontok, élek) karbantartása (keresés, felvétel, módosítás, törlés).
  - Ellenőrizzük, hogy egy éljellemzőre nézve szimmetrikus-e a hálózat vagy sem!
  - Szimmetrizáljuk a hálózatot egy éljellemzőre vonatkozóan (azaz legyen minden élnek párja, és a két éljellemző egyezzen meg)!
- Konvertáljuk egy hálózat éladatait az egyes tárolási reprezentációk között!
- Egy tömegközlekedés megállóit az 1, 2, …, *N* sorszámokkal jelöljük. A megállók között közlekedő, a sorszámukkal (1, 2, …, *M*) azonosított járatokat az érintett megállók felsorolásával definiáljuk (pl. egy járat megállói: 3, 5, 2). Feltesszük, hogy egyik járat sem tartalmaz kört (azaz ismétlődő megállót). Oldjuk meg az alábbi feladatokat:
  - Adatok betöltése alkalmasan megválasztott formátumú szövegfájlból.
  - Adott megállóból közvetlenül (átszállás nélkül) elérhető megállók meghatározása.
  - Minden kezdő- és végmegálló viszonylatra meghatározandó az elérhetőség ténye, azaz hogy el tudunk-e utazni a járatokon a kezdőmegállóból a végmegállóba vagy sem. Ha igen, mennyi az a minimális átszállás, amellyel ez megtehető és milyen járatokon kell ehhez utaznunk?

16. Irodalomjegyzék

[Aho 82] Aho, A. V. – Hopcroft, J. E. – Ullman, J. D.: *Számítógép-algoritmusok tervezése és analízise.* Műszaki könyvkiadó, 1982.

[Bau 04] Bauer Péter: *C programnyelv.* Elektronikus jegyzet, 2004.

[Cor 97] Cormen, T. H. – Leiserson, C. E. – Rivest, R. L.: *Algoritmusok.* Műszaki könyvkiadó, 1997.

[Ker 85] Kernighan, B. W. – Ritchie, D. M.: *A C programozási nyelv.* Műszaki könyvkiadó, 1985.

[Knu 87] Knuth, D. E.: *A számítógép-programozás művészete* Műszaki könyvkiadó, 1987.

[Nyé 03] Nyékyné Gaizler Judit: *Programozási nyelvek.* Kiskapu Kft., 2003.

[Mar 93] Marton László – Pukler Antal – Pusztai Pál: *Bevezetés a programozásba.* NOVADAT, 1993.

[Mar 97] Marton László – Pusztai Pál: *Gráfok és hálózatok kezelése számítógéppel I-VI.* Új Alaplap, 1997/4-9.

[Mar 98] Marton László: *Bevezetés a Pascal nyelvű programozásba.* NOVADAT, 1998.

[Mar 02] Marton László – Fehérvári Arnold: *Algoritmusok és adatstruktúrák.* NOVADAT, 2002.

[Sai 86] Sain Márton: *Nincs királyi út! Matematikatörténet.* Gondolat, 1986.

[Wir 82] Wirth, N.: *Algoritmusok + Adatstruktúrák = Programok.* Műszaki könyvkiadó, 1982.

17. Függelék

17.1. C programok

17.1.1. Legnagyobb közös osztó

```
/* LNKO.C : Két természetes szám legnagyobb közös osztójának
meghatározása */

#include <stdio.h>
#include <conio.h>

/* Nem rekurzív függvény osztással */
int lnko_o(int a, int b)
{ int r;
  while (b!=0) {
    r=a%b; a=b; b=r;
  }
  return (a);
}

/* Nem rekurzív függvény kivonással */
int lnko_k(int a, int b)
{ while (a!=b)
    if (a>b) a=a-b; else b=b-a;
  return (a);
}

/* Rekurzív függvény osztással */
int lnko_ro(int a, int b)
{ int er;
  if (b==0)
    er=a;
  else
    er=lnko_ro(b,a%b);
  return (er);
}

/* Rekurzív függvény kivonással */
int lnko_rk(int a, int b)
{ int er;
  if (a==b)
    er=a;
  else if (a>b)
    er=lnko_rk(a-b,b);
  else
    er=lnko_rk(a,b-a);
  return (er);
}

void main()
{ int a,b;
  clrscr();
```

```
  printf("Egyik természetes szám:"); scanf("%d",&a);
  printf("Másik természetes szám:"); scanf("%d",&b);
  printf("A legnagyobb közös osztó:\n");
  printf("%d %d %d %d\n",
         lnko_o(a,b),lnko_k(a,b),lnko_ro(a,b),lnko_rk(a,b));
}
```

17.1.2. Kör területe, kerülete

```
/* KOR.C : Kör területe, kerülete */

#include <conio.h>
#include <stdio.h>

void main(void)
{ float r,t,k;
  clrscr();
  printf("A kör sugara:"); scanf("%f",&r);
  t=r*r*3.14;
  k=2*r*3.14;
  printf("Terület:%0.2f\n",t);
  printf("Kerület:%0.2f\n",k);
}
```

17.1.3. Másodfokú egyenlet

```
/* MASODFOK.C : Másodfokú egyenlet megoldása */

#include <conio.h>
#include <stdio.h>
#include <math.h>

void main(void)
{ float a,b,c,d,x1,x2,k,v;
  clrscr();
  printf("Együtthatók (a b c):"); scanf("%f %f %f",&a,&b,&c);
  if (a==0)
    /* Nem másodfokú eset */
    if (b==0)
      /* Nem elsőfokú eset */
      if (c==0)
        printf("Minden valós szám megoldás!\n");
      else
        printf("Nincs megoldás!\n");
    else
      printf("Elsőfokú:%0.2f\n",-c/b);
  else {
    /* Másodfokú eset */
    d=b*b-4*a*c;
    if (d>0) {
      /* Két valós gyök */
      x1=(-b+sqrt(d))/(2*a);
      x2=(-b-sqrt(d))/(2*a);
```

```
      printf("Két valós gyök:%0.2f %0.2f\n",x1,x2);
    } else if (d==0)
      printf("Másodfokú:%0.2f\n",-b/(2*a));
    else {
      /* Komplex gyökök */
      v=-b/(2*a);
      k=abs(sqrt(-d)/(2*a));
      printf("Egyik komplex gyök:%0.2f+%0.2fi\n",v,k);
      printf("Másik komplex gyök:%0.2f-%0.2fi\n",v,k);
    }
  }
}
```

17.1.4. Legkisebb osztó

```
/* LEGKOSZT.C : Legkisebb osztó */

#include <conio.h>
#include <stdio.h>

void main(void)
{ int n,o;
  clrscr();
  printf("A vizsgált szám (>1):"); scanf("%d",&n);
  o=2;
  while (n%o!=0) o++;
  printf("A legkisebb, egynél nagyobb osztó:%d\n",o);
}
```

17.1.5. Faktoriális

```
/* FAKTOR.C : Faktoriális */

#include <conio.h>
#include <stdio.h>

void main(void)
{ int n,i; long double fakt;
  clrscr();
  printf("N értéke:"); scanf("%d",&n);
  fakt=1;
  for (i=2; i<=n; i++) fakt=fakt*i;
  printf("%d!=%0.0Lf\n",n,fakt);
}
```

17.1.6. Karakterek vizsgálata

```
/* KARVIZSG.C : Karakterek vizsgálata */

#include <stdio.h>
#include <conio.h>

void main(void)
```

```
{ int ch;
  clrscr();
  printf("Karakterek vizsgálata (Kilépés:Esc)\n");
  do {
    /* Bekérés */
    ch=getch();
    if (ch!=27) {
      /* Kiírás */
      printf("%c %d ",ch,ch);
      /* Kiértékelés */
      if (ch>='A' && ch<='Z' || ch>='a' && ch<='z')
        printf("Angol betű\n");
      else if (ch>='0' && ch<='9')
        printf("Számjegy\n");
      else printf("Egyéb\n");
    }
  } while (ch!=27);
}
```

17.1.7. Prímfelbontás

```
/* PRIMFELB.C : Prímfelbontás */

#include <conio.h>
#include <stdio.h>

void main(void)
{ int n,o;
  clrscr();
  printf("A felbontandó száma (>1):"); scanf("%d",&n);
  o=2;
  while (n>1) {
    if (n%o==0) {
      n=n/o;
      if (n==1) printf("%d\n",o);
      else printf("%d*",o);
    } else
      o++;
  }
}
```

17.1.8. Monoton növő sorozat

```
/* MONNOVO.C : Monoton növő sorozat */

#include <conio.h>
#include <stdio.h>

void main(void)
{ int n,i,Novo; float a,Elozo;
  clrscr();
  printf("Adatok száma (>1):"); scanf("%d",&n);
```

```
  /* Első adat */
  printf("1. adat:"); scanf("%f",&a);
  /* Kezdőértékek */
  Novo=1; Elozo=a;
  /* Többi adat */
  for (i=2; i<=n; i++) {
    printf("%d. adat:",i); scanf("%f",&a);
    if (a<Elozo) Novo=0;
    Elozo=a;
  }
  if (Novo) printf("Monoton növők!\n");
  else printf("Nem monoton növők!\n");
}
```

17.1.9. Pozitív adatok maximuma, átlaga

```
/* PMAXATL.C : Pozitív adatok maximuma, átlaga */

#include <stdio.h>
#include <conio.h>

void main(void)
{ float Akt,Max,Atl,Ossz; int n,Db,i;
  clrscr();
  printf("Adatok száma:"); scanf("%d",&n);
  /* Kezdőértékek */
  Db=Ossz=0;
  /* Adatbekérés, feldolgozás */
  for (i=0; i<n; i++) {
    printf("%d. adat:",i+1); scanf("%f",&Akt);
    if (Akt>=0) {
      Db++;
      Ossz+=Akt;
      if (Db==1) Max=Akt;
      else if (Akt>Max) Max=Akt;
    }
  }
  /* Eredménykiírás */
  if (Db==0)
    printf("Nem volt pozitív adat!\n");
  else {
    Atl=Ossz/Db;
    printf("A pozitív adatok maximuma:%0.2f\n",Max);
    printf("A pozitív adatok átlaga:%0.2f\n",Atl);
  }
}
```

17.1.10. e^x hatványsora

```
/* EXPXKOZ.C : Az exponenciális függvény közelítése */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>

void main(void)
{ double x,Epsz,Akt,Ossz; int n;
  clrscr();
  /* Adatbekérés */
  printf("X értéke:"); scanf("%lf",&x);
  printf("Pontosság:"); scanf("%lf",&Epsz);
  /* Kezdőértékek */
  n=1; Akt=x; Ossz=1+Akt;
  /* Közelítés */
  while (fabs(Akt)>=Epsz) {
    n++;
    Akt*=x/n;
    Ossz+=Akt;
  }
  /* Eredménykiírás */
  printf("N értéke:%d\n",n);
  printf("A közelítő érték:%0.10f\n",Ossz);
  printf("A 'pontos' érték:%0.10f\n",exp(x));
}
```

17.1.11. Gyökkeresés intervallumfelezéssel

```
/* GYOKKER.C : Gyökkeresés intervallumfelezéssel */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>

double f(double x)
{ return (sin(x));
}

void main(void)
{ double a,b,Epsz,Gyok,xk,xv,xf,yk,yv,yf;
  clrscr();
  /* Adatbekérés */
  printf("Az intervallum kezdőpontja:"); scanf("%lf",&a);
  printf("Az intervallum végpontja  :"); scanf("%lf",&b);
  printf("Pontosság:"); scanf("%lf",&Epsz);
  /* Kezdőértékek */
  xk=0; xv=b; yk=f(a); yv=f(b);
  /* Közelítés */
  while (xv-xk>Epsz) {
```

```
    /* Felezőpont */
    xf=(xk+xv)/2;
    yf=f(xf);
    /* Csökkentés */
    if (yk*yf<=0) {
      xv=xf; yv=yf;
    }
    if (yv*yf<=0) {
      xk=xf; yk=yf;
    }
  }
  /* Eredmény */
  Gyok=(xk+xv)/2;
  /* Eredménykiírás */
  printf("A gyök közelítő értéke:%0.10f\n",Gyok);
}
```

17.1.12. Integrálérték meghatározása közelítéssel

```
/* TRAPEZ.C : Határozott integrál közelítése trapéz-módszerrel
*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>

double f(double x)
{ return (3*x*x);
}

void main(void)
{ double a,b,Epsz,t,e,h,y0,yn; int n,i;
  clrscr();
  /* Adatbekérés */
  printf("Az intervallum kezdőpontja:"); scanf("%lf",&a);
  printf("Az intervallum végpontja  :"); scanf("%lf",&b);
  printf("Pontosság:"); scanf("%lf",&Epsz);
  /* Kezdőértékek */
  y0=f(a); yn=f(b);
  t=(b-a)*(y0+yn)/2;
  /* Következő beosztás */
  n=2;
  /* Közelítés */
  do {
    /* Előző területösszeg */
    e=t;
    /* Részintervallumok hossza */
    h=(b-a)/n;
    /* Trapézok területösszege */
    t=(y0+yn)/2;
    for (i=1; i<=n-1; i++)
```

```
      t+=f(a+i*h);
    t*=h;
    /* Következő beosztás */
    n=2*n;
  } while (fabs(t-e)>=Epsz);
  /* Eredménykiírás */
  printf("Az integrál közelítő értéke:%0.10f\n",t);
}
```

17.1.13. Átlagnál nagyobb elemek

```
/* ATLNAGY.C : Átlagnál nagyobb elemek */

#include <stdio.h>
#include <conio.h>

#define NMax 100    /* Az adatok maximális száma */

void main(void)
{ float a[NMax]; int n,i; float Ossz,Atl;
  clrscr();
  /* Adatbekérés */
  printf("Adatok száma (1-%d):",NMax); scanf("%d",&n);
  for (i=0; i<n; i++) {
    printf("%d. adat:",i+1); scanf("%f",&a[i]);
  }
  /* Átlagszámolás */
  Ossz=0;
  for (i=0; i<n; i++)
    Ossz+=a[i];
  Atl=Ossz/n;
  /* Eredménykiírás */
  printf("Az átlagnál nagyobb elemek\n");
  for (i=0; i<n; i++)
    if (a[i]>Atl)
      printf("%0.2f\n",a[i]);
}
```

17.1.14. Kockadobások gyakorisága

```
/* KOCKA.C : Kockadobások gyakorisága */

#include <stdio.h>
#include <conio.h>

#define n 6    /* A kocka oldalainak száma */

void main(void)
{ int db[n]; int a,i;
  clrscr();
  printf("Kockadobások (1-%d) gyakorisága (Kilépés:0)\n",n);
  /* Kezdőértékek */
```

```
  for (i=0; i<n; i++) db[i]=0;
  i=0;
  do {
    /* Bekérés */
    printf("%d. dobás:",++i); scanf("%d",&a);
    if (a!=0) db[a-1]++;
  } while (a!=0);
  /* Eredménykiírás */
  for (i=0; i<n; i++)
    printf("%d %d\n",i+1,db[i]);
}
```

17.1.15. Érték törlése adatsorból

```
/* ERTTORL.C : Érték törlése adatsorból */

#include <stdio.h>
#include <conio.h>
#include <math.h>

#define NMax 10                 /* Az adatok maximális száma */

typedef int Elem;               /* Az adatsor elemei */
typedef Elem Adatsor[NMax];     /* Az adatsor */

void Torol(Adatsor a, int *n, Elem x)
{ int i,j;
  /* Kezdőérték */
  j=0;
  /* Előremásolás */
  for (i=0; i<*n; i++)
    if (a[i]!=x) {
      a[j]=a[i];
      j++;
    }
  /* Darabszám */
  *n=j;
}

void main(void)
{ Adatsor a;
  int n,i; Elem x;
  clrscr();
  printf("Adatok száma (1-%d):",NMax); scanf("%d",&n);
  for (i=0; i<n; i++) {
    printf("%d. adat:",i+1); scanf("%d",&a[i]);
  }
  printf("A törlendő érték:"); scanf("%d",&x);
  Torol(a,&n,x);
  printf("Törlés után\n");
  for (i=0; i<n; i++) printf("%d ",a[i]); printf("\n");
}
```

17.1.16. Eratosztenesz szitája

```
/* SZITA_T.C : Eratosztenesz szitája tömbbel */

#include <conio.h>
#include <stdio.h>

#define K 1000          /* Eddig határozzuk meg a prímszámokat */

typedef char Tomb[K+1];       /* A számokat reprezentáló tömb */

void main(void)
{ Tomb a; int p,i;
  /* Számok felírása */
  for (i=2; i<=K; i++) a[i]=1;
  /* Szitálás */
  for (p=2; p<=K; p++)
    /* A p szám prímszám? */
    if (a[p])
      /* Igen, töröljük a többszöröseit */
      for (i=2*p; i<=K; i+=p) a[i]=0;
  /* Kiírás */
  clrscr();
  for (i=2; i<=K; i++)
    if (a[i]) printf("%8d",i);
}
```

17.1.17. Mátrixösszegek

```
/* MATOSSZ.C : Mátrixösszegek */

#include <stdio.h>
#include <conio.h>

#define NMax 10         /* Sorok maximális száma */
#define MMax 10         /* Oszlopok maximális száma */

typedef float Matrix[NMax][MMax];     /* A mátrix */
typedef float SorOsszeg[NMax];        /* A sorösszegek */
typedef float OszlOsszeg[NMax];       /* Az oszlopösszegek */

void Osszegek(Matrix a, int n, int m,
 SorOsszeg Sor, OszlOsszeg Oszl, float *Ossz)
{ int i,j;
  /* Kezdőértékek */
  for (j=0; j<m; j++) Oszl[j]=0;
  *Ossz=0;
  /* Összegzés */
  for (i=0; i<n; i++) {
    Sor[i]=0;
    for (j=0; j<m; j++) {
      Sor[i]+=a[i][j];
      Oszl[j]+=a[i][j];
```

```
    }
    *Ossz+=Sor[i];
  }
}

void main(void)
{ Matrix a; SorOsszeg s; OszlOsszeg o; float ossz;
  int n,m,i,j;
  clrscr();
  /* Adatbekérés */
  printf("Sorok száma (1-%d):",NMax); scanf("%d",&n);
  printf("Oszlopok száma (1-%d):",MMax); scanf("%d",&m);
  for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
      printf("%d. sor %d. elem:",i+1,j+1); scanf("%f",&a[i][j]);
    }
  /* A mátrix kiírása mátrix alakban */
  printf("A megadott mátrix\n");
  for (i=0; i<n; i++) {
    for (j=0; j<m; j++) printf("%8.1f",a[i][j]);
    printf("\n");
  }
  /* Összegzés */
  Osszegek(a,n,m,s,o,&ossz);
  /* Eredménykiírás */
  printf("A sorösszegek:\n");
  for (i=0; i<n; i++) printf("%8.1f",s[i]); printf("\n");
  printf("Az oszlopösszegek:\n");
  for (j=0; j<m; j++) printf("%8.1f",o[j]); printf("\n");
  printf("A teljes összeg:\n%8.1f\n",ossz);
}
```

17.1.18. Oszlopok törlése mátrixból

```
/* OSZLTORL.C : Oszlopok törlése mátrixból */

#include <stdio.h>
#include <conio.h>

#define NMax 10          /* Sorok maximális száma */
#define MMax 10          /* Oszlopok maximális száma */

typedef int Matrix[NMax][MMax];      /* A mátrix */

void Torol(Matrix a, int n, int *m)
{ int i,j,k,Egyf;
  j=0;
  while (j<*m) {
    /* A j. oszlop vizsgálata */
    Egyf=1;
    i=1;
    while (i<n && Egyf) {
```

```
      Egyf=a[i][j]==a[0][j];
      i++;
    }
    if (Egyf) {
      /* A j. oszlop törlése */
      for (k=j+1; k<*m; k++)
        for (i=0; i<n; i++)
          a[i][k-1]=a[i][k];
      (*m)--;
    } else
      j++;
  }
}

/* A mátrix kiírása mátrix alakban */
void Kiir(Matrix a, int n, int m)
{ int i,j;
  for (i=0; i<n; i++) {
    for (j=0; j<m; j++) printf("%6d",a[i][j]);
    printf("\n");
  }
}

void main(void)
{ Matrix a; int n,m,i,j;
  clrscr();
  /* Adatbekérés */
  printf("Sorok száma (1-%d):",NMax); scanf("%d",&n);
  printf("Oszlopok száma (1-%d):",MMax); scanf("%d",&m);
  for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
      printf("%d. sor %d. elem:",i+1,j+1); scanf("%d",&a[i][j]);
    }
  printf("A megadott mátrix\n"); Kiir(a,n,m);
  /* Törlés */
  Torol(a,n,&m);
  printf("Az eredmény mátrix\n"); Kiir(a,n,m);
}
```

17.1.19. Sztring megfordítás

```
/* STRFORD.C : Sztring megfordítása */

#include <stdio.h>
#include <conio.h>
#include <string.h>

#define MaxHossz 100           /* A sztring maximális hossza */

typedef char Sztring[MaxHossz+1];    /* A sztring, mint tömb */

void Fordit1(Sztring s, Sztring er)
```

```
{ int h,i,j;
  h=strlen(s);
  for (i=0, j=h-1; i<h; i++, j--) er[j]=s[i];
  /* Végjel */
  er[h]='\0';
}

void Fordit2(Sztring s, Sztring er)
{ int h,i,j;
  h=strlen(s);
  for (i=h-1, j=0; i>=0; i--, j++) er[j]=s[i];
  /* Végjel */
  er[h]='\0';
}

void Fordit3(Sztring s)
{ int h,i,cs;
  h=strlen(s);
  for (i=0; i<h/2; i++) {
    cs=s[i]; s[i]=s[h-1-i]; s[h-1-i]=cs;
  }
}

void main(void)
{ Sztring s,er1,er2;
  clrscr();
  printf("A megfordítandó sztring:"); scanf("%s",&s);
  Fordit1(s,er1); Fordit2(s,er2); Fordit3(s);
  printf("Megfordítva:\n");
  printf("%s\n",er1); printf("%s\n",er2); printf("%s\n",s);
}
```

17.1.20. Egyszerű kifejezés kiértékelése

```
/* STRKIF.C : Sztringben lévő egyszerű kifejezés kiértékelése */

#include <stdio.h>
#include <conio.h>
#include <string.h>

#define MaxHossz 100            /* A sztring maximális hossza */

typedef char Sztring[MaxHossz+1];     /* A sztring, mint tömb */

void Ertekel(Sztring s, int *jo, int *er)
{ int h,i;
  /* Hosszellenőrzés */
  h=strlen(s);
  *jo=h%2==1;
  if (jo) {
    /* Karakterek ellenőrzése */
    i=0;
```

```
    while (i<h && *jo)
      /* A páros/páratlan pozíció fordítva (0-tól indexelünk) */
      if (i%2==0 && (s[i]<'0' || s[i]>'9') ||
          (i%2==1 && s[i]!='+' && s[i]!='-'))
        *jo=0;
      else i++;
    if (*jo) {
      /* A kifejezés értékének kiszámítása */
      *er=s[0]-'0';
      i=2;
      while (i<h) {
        if (s[i-1]=='+')
          *er+=s[i]-'0';
        else
          *er-=s[i]-'0';
        i+=2;
      }
    }
  }
}

void main(void)
{ Sztring s; int jo,er;
  clrscr();
  printf("A kiértékelendő kifejezés:"); scanf("%s",&s);
  Ertekel(s,&jo,&er);
  if (jo)
    printf("Helyes kifejezés, értéke:%d\n",er);
  else
    printf("Hibás kifejezés!\n");
}
```

17.1.21. Lottószámok generálása

```
/* LOTTO.C : Lottószámok generálása */

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define Max 90          /* A maximális lottószám értéke */
#define Db 5            /* A generálandó lottószámok darabszáma */

typedef int Tomb[Db];         /* A lottószámok tömbje */
typedef int Halmaz[Max];      /* A lottószámok halmaza */

/* Lottószámok generálása */
void General(Tomb a)
{ Halmaz h; int x,i,j;
  /* A halmaz legyen üres */
  for (i=0; i<Max; i++) h[i]=0;
```

```
  /* Generálás */
  for (i=0; i<Db; i++) {
    do
      x=random(Max);
    while (h[x]==1);
    h[x]=1;
  }
  /* Az eredménytömb feltöltése */
  j=0;
  for (i=0; i<Max; i++)
    if (h[i]==1) {
      a[j]=i+1; j++;
    }
}

void main(void)
{ Tomb a; int i;
  clrscr(); randomize();
  /* Generálás */
  General(a);
  /* Kiírás */
  for (i=0; i<Db; i++) printf("%d ",a[i]);
}
```

17.1.22. Különböző karakterek száma

```
/* KULKAR.C : Egy sztring különböző karaktereinek száma */

#include <stdio.h>
#include <conio.h>
#include <string.h>

#define MaxHossz 100            /* A sztring maximális hossza */
#define MaxKar 256  /* A különböző karakterek maximális száma */

typedef char Sztring[MaxHossz+1];     /* A sztring, mint tömb */
typedef int Halmaz[MaxKar];           /* A halmaz, mint tömb */

int KulKarDb(Sztring s)
{ Halmaz h; int i,db;
  db=0;
  /* A halmaz legyen üres */
  for (i=0; i<MaxKar; i++) h[i]=0;
  for (i=0; i<strlen(s); i++)
    if (h[s[i]]==0) {
      h[s[i]]=1; db++;
    }
  return (db);
}

void main(void)
{ Sztring s;
```

```
  clrscr();
  printf("A sztring:"); scanf("%s",&s);
  printf("A különböző karakterek száma:%d\n",KulKarDb(s));
}
```

17.1.23. Adatsor megjelenítése

```
/* TOMBKIIR.C : Adatsor megjelenítése */

#include <conio.h>
#include <stdio.h>

#define Esc  27               /* A megengedett billentyűk */
#define Home 71
#define End  79
#define Down 80
#define Up   72
#define PgUp 73
#define PgDn 81

#define Max 256       /* Karakterhalmazok maximális elemszáma */
typedef int Jelek[Max];

#define NMax 10                  /* Az adatok maximális száma */
typedef int Elem;                /* Az adatsor elemei */
typedef Elem Adatsor[NMax];      /* Az adatsor */

int BillBe(int *Normal, int *Dupla, int *Duplae)
{ int Jel,JoJel;
  do {
    Jel=getch(); *Duplae=Jel==0;
    if (*Duplae) {
      Jel=getch();
      JoJel=Dupla[Jel];
    } else
      JoJel=Normal[Jel];
  } while (!JoJel);
  return (Jel);
}

void Ures(int *Jelek)
{ int i;
  for (i=0; i<Max; i++) Jelek[i]=0;
}

void Bovit1(int *Mit, int Mivel)
{ Mit[Mivel]=1;
}

void Kiir(Adatsor a, int n, int db)
{ int Kezd,Bef,i,Jel,Duplae; Jelek Normal,Dupla;
  if (n==0) {
```

```
    clrscr(); printf("Nincsenek elemek!\n"); getch();
  } else {
    Kezd=0;
    do {
      /* A befejező index beállítása */
      Bef=Kezd+db-1; if (Bef>n-1) Bef=n-1;
      /* Adatkiírás */
      clrscr(); printf("Ssz. Elem\n");
      for (i=Kezd; i<=Bef; i++) printf("%3d. %d\n",i+1,a[i]);
      /* Az elfogadható jelek beállítása */
      Ures(Normal); Bovit1(Normal,Esc); Ures(Dupla);
      if (Kezd>0) {
        Bovit1(Dupla,Home);Bovit1(Dupla,Up);Bovit1(Dupla,PgUp);
      }
      if (Bef<n-1) {
        Bovit1(Dupla,End);Bovit1(Dupla,Down);Bovit1(Dupla,PgDn);
      }
      /* Billentyűzetről való bekérés */
      Jel=BillBe(Normal,Dupla,&Duplae);
      /* Pozícionálás */
      if (Jel==Home) Kezd=0;
      else if (Jel==End) Kezd=n-db;
      else if (Jel==PgDn)
        if (Bef<n-db) Kezd+=db; else Kezd=n-db;
      else if (Jel==PgUp)
        if (Kezd-db>=0) Kezd-=db; else Kezd=0;
      else if (Jel==Down) Kezd++;
      else if (Jel==Up) Kezd--;
    } while (Jel!=Esc);
  }
}

void main(void)
{ Adatsor a;
  int n,db,i;
  clrscr();
  printf("Adatok száma (0-%d):",NMax); scanf("%d",&n);
  for (i=0; i<n; i++) {
    printf("%d. adat:",i+1); scanf("%d",&a[i]);
  }
  printf("Egy képernyőn lévő adatok száma (1-%d):",NMax);
  scanf("%d",&db);
  Kiir(a,n,db);
}
```

17.1.24. Üzletek tartozása

```
/* UZLETEK.C : Üzletek tartozása */

#include <conio.h>
#include <stdio.h>
```

```
#define NMax 10                 /* Termékek maximális száma */
#define MMax 20                 /* Üzletek maximális száma */
#define NevMaxH 30              /* Terméknevek maximális hossza */

typedef struct {                        /* Termék */
          char Nev[NevMaxH+1];          /* Terméknév */
          unsigned int Ear;             /* Egységár */
        } Termek;

typedef struct {                        /* Üzlet */
          unsigned int Db[NMax];        /* Szállított mennyiségek */
          long int Tart;                /* Tartozás */
        } Uzlet;

typedef Termek TermekTomb[NMax];        /* Termékek */
typedef Uzlet UzletTomb[MMax];          /* Üzletek */

void main(void)
{ int n,m,i,j;
  TermekTomb t; UzletTomb u;
  clrscr();
  /* Adatbekérés */
  printf("Termékek száma (1-%d):",NMax); scanf("%d",&n);
  printf("Üzletek száma (1-%d):",MMax); scanf("%d",&m);
  for (i=0; i<n; i++) {
    printf("%d. termék\n",i+1);
    printf("Neve:"); scanf("%s",t[i].Nev);
    printf("Egységára:"); scanf("%d",&t[i].Ear);
  }
  /* Szállított mennyiségek */
  for (i=0; i<m; i++) {
    printf("%d. üzletnek szállított mennyiségek\n",i+1);
    for (j=0; j<n; j++) {
      printf("%s:",t[j].Nev); scanf("%d",&u[i].Db[j]);
    }
  }
  /* Tartozások kiszámítása */
  for (i=0; i<m; i++) {
    u[i].Tart=0;
    for (j=0; j<n; j++)
      /* Típuskonverzióval, hogy ne legyen túlcsordulás */
      u[i].Tart+=(long int)u[i].Db[j]*t[j].Ear;
  }
  /* Eredménykiírás */
  printf("Tartozások\n");
  for (i=0; i<m; i++)
    printf("%d. üzlet:%ld\n",i+1,u[i].Tart);
}
```

17.1.25. Minimumhelyek keresése

```
/* MINHELY.C : Minimumhelyek keresése */

#include <stdio.h>
#include <conio.h>

#define NMax 10                    /* Az adatok maximális száma */

typedef int Elem;                          /* Az adatsor elemei */
typedef Elem Adatsor[NMax];                /* Az adatsor */
typedef int MinHelyek[NMax];               /* A minimumhelyek */

void MinKereses(const Adatsor a, int n, Elem *Min, int *db,
MinHelyek Hely)
{ int i;
  /* Minimum meghatározása */
  *Min=a[0];
  for (i=1; i<n; i++)
    if (a[i]<*Min)
      *Min=a[i];
  /* Minimumhelyek */
  *db=0;
  for (i=0; i<n; i++)
    if (a[i]==*Min) {
      Hely[*db]=i;
      (*db)++;
    }
}

void main(void)
{ Adatsor a; MinHelyek Hely;
  int n,i,db; Elem Min;
  clrscr();
  printf("Adatok száma (1-%d):",NMax); scanf("%d",&n);
  for (i=0; i<n; i++) {
    printf("%d. adat:",i+1); scanf("%d",&a[i]);
  }
  MinKereses(a,n,&Min,&db,Hely);
  printf("A legkisebb elem:%d\n",Min);
  printf("Előfordulási hely(ek):\n");
  for (i=0; i<db; i++)
    printf("%4d\n",Hely[i]+1);
}
```

17.1.26. Átlag és szórás

```
/* ATLSZOR.C : Átlag és szórás */

#include <stdio.h>
#include <conio.h>
#include <math.h>
```

```
#define NMax 10                    /* Az adatok maximális száma */

typedef int Elem;                        /* Az adatsor elemei */
typedef Elem Adatsor[NMax];              /* Az adatsor */

void Szamol(const Adatsor a, int n, float *Atl, float *Sz)
{ int i; float Ossz;
  /* Átlag */
  Ossz=0;
  for (i=0; i<n; i++)
    Ossz+=a[i];
  *Atl=Ossz/n;
  /* Szórás */
  Ossz=0;
  for (i=0; i<n; i++)
    Ossz+=pow(a[i]-*Atl,2);
  *Sz=pow(Ossz/n,0.5);
}

void main(void)
{ Adatsor a;
  int n,i; float Atl,Sz;
  clrscr();
  printf("Adatok száma (1-%d):",NMax); scanf("%d",&n);
  for (i=0; i<n; i++) {
    printf("%d. adat:",i+1); scanf("%d",&a[i]);
  }
  Szamol(a,n,&Atl,&Sz);
  printf("Átlag :%0.2f\n",Atl);
  printf("Szórás:%0.2f\n",Sz);
}
```

17.1.27. Előfordulási statisztika

```
/* ELOFSTAT.C : Előfordulási statisztika */

#include <stdio.h>
#include <conio.h>

#define NMax 10                    /* Az adatok maximális száma */

typedef int Elem;                        /* Az adatsor elemei */
typedef Elem Adatsor[NMax];              /* Az adatsor */
typedef int Dbsor[NMax];                 /* A darabszámok */

void Statisztika(Adatsor a, int n, Adatsor t, Dbsor db, int *k)
{ int i,j,van;
  /* Kezdőérték */
  *k=0;
  for (i=0; i<n; i++) {
    /* Felvétel */
    t[*k]=a[i];
```

```
    db[*k]=0;
    (*k)++;
    /* Keresés */
    j=0;
    while (a[i]!=t[j]) j++;
    /* Darabszám növelés */
    db[j]++;
    /* Ha volt már ilyen, töröljük a táblázat végéről */
    if (j<*k-1) (*k)--;
  }
}

void main(void)
{ Adatsor a,t; Dbsor db;
  int n,i,k;
  clrscr();
  printf("Adatok száma (1-%d):",NMax); scanf("%d",&n);
  for (i=0; i<n; i++) {
    printf("%d. adat:",i+1); scanf("%d",&a[i]);
  }
  Statisztika(a,n,t,db,&k);
  printf("A különböző adatok és darabszámuk:\n");
  for (i=0; i<k; i++)
    printf("%8d%4d db\n",t[i],db[i]);
}
```

17.1.28. Jelstatisztika

```
/* JELSTAT.C : Jelstatisztika */

#include <stdio.h>
#include <conio.h>
#include <string.h>

#define NMax 100                /* A sztring maximális hossza */

typedef char Sztring[NMax+1];      /* A sztring */
typedef struct {           /* A gyakorisági táblázat egy eleme */
          char Jel;
          unsigned char Darab;
        } Elem;
typedef Elem Tabla[255];           /* A gyakorisági táblázat */

void Statisztika(Sztring s, Tabla t, unsigned char *k)
{ int i; unsigned char kod, db[256];
  /* Kezdőértékek */
  for (i=0; i<=255; i++) db[i]=0;
  /* Darabszámok meghatározása */
  for (i=0; i<strlen(s); i++) {
    kod=s[i];
    db[kod]++;
  }
```

```
  /* Eredménytáblázat elkészítése */
  *k=0;
  for (i=0; i<=255; i++)
    if (db[i]>0) {
      t[*k].Jel=i;
      t[*k].Darab=db[i];
      (*k)++;
    }
}

void main(void)
{ Sztring s; Tabla t; unsigned char i,k;
  clrscr();
  printf("A sztring:"); scanf("%s",&s);
  Statisztika(s,t,&k);
  printf("A különböző jelek és darabszámuk:\n");
  for (i=0; i<k; i++)
    printf("%c%4d db\n",t[i].Jel,t[i].Darab);
}
```

17.1.29. Rendezés és keresés

```
/* RENDKER.C : Rendezések, keresések */

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MaxElemSzam 10    /* A rendezendő adatok max. száma */
#define MaxElem 20        /* A generálandó adatok max. értéke */

typedef int Elem;               /* A rendezendő elemek típusa */
typedef Elem Tomb[MaxElemSzam]; /* A rendezendő elemek tömbje */
typedef int IndexTabla[MaxElemSzam];  /* Az indextábla típusa */

/* Buborékrendezés */
void BubRend(Tomb a, int n)
{ Elem cs; int i,j;
  for (i=0; i<n-1; i++)
    /* i. elemet a helyére */
    for (j=n-1; j>=i+1; j--)
      if (a[j]<a[j-1]) {
        /* A j. és j-1. elemek cseréje */
        cs=a[j]; a[j]=a[j-1]; a[j-1]=cs;
      }
}

/* Rendezés kiválasztással */
void KivalRend(Tomb a, int n)
{ Elem cs; int i,j,k;
  for (i=0; i<n-1; i++) {
```

```
    /* i. elemet a helyére */
    k=i;
    for (j=i+1; j<n; j++)
      if (a[j]<a[k]) k=j;
    if (k>i) {
      /* Az i. és k. elemek cseréje */
      cs=a[i]; a[i]=a[k]; a[k]=cs;
    }
  }
}

/* Rendezés kiválasztással, indextáblával */
void KivalRendIt(Tomb a, IndexTabla it, int n)
{ int i,j,k,cs;
  /* Az indextábla feltöltése */
  for (i=0; i<n; i++) it[i]=i;
  /* Rendezés */
  for (i=0; i<n-1; i++) {
    /* i. elemet a helyére */
    k=i;
    for (j=i+1; j<n; j++)
      if (a[it[j]]<a[it[k]]) k=j;
    if (k>i) {
      /* Az i. és k. indexek cseréje */
      cs=it[i]; it[i]=it[k]; it[k]=cs;
    }
  }
}

/* Rendezés beszúrással */
void BeszurRend(Tomb a, int n)
{ Elem x; int i,j;
  for (i=1; i<n; i++) {
    /* Az i. elem beszúrása az előtte lévő rendezett részbe */
    x=a[i];
    /* Helykészítés hátraléptetéssel */
    j=i-1;
    while (j>=0 && a[j]>x) {
      a[j+1]=a[j]; j--;
    }
    /* i. elemet a helyére */
    a[j+1]=x;
  }
}

/* Lineáris keresés rendezetlen adatok között */
int LinKer(Tomb a, int n, Elem x, int *Hol)
{ int i,Van;
  i=0;
  while (i<n && a[i]!=x) i++;
  if (i<n) {
    Van=1;
```

```
    *Hol=i;
  } else Van=0;
  return (Van);
}

/* Lineáris keresés rendezett adatok között */
int LinKerRend(Tomb a, int n, Elem x, int *Hol)
{ int i,Van;
  i=0;
  while (i<n && a[i]<x) i++;
  Van=i<n && a[i]==x;
  *Hol=i;
  return (Van);
}

/* Bináris keresés */
int BinKer(Tomb a, int n, Elem x, int *Hol)
{ int i,j,k,Van;
  i=0; j=n-1; Van=0;
  while (i<=j && !Van) {
    k=(i+j)/2;
    if (a[k]==x) Van=1;
    else
      if (x<a[k]) j=k-1;
      else i=k+1;
  }
  if (Van) *Hol=k;
  else *Hol=i;
  return (Van);
}

/* Bináris keresés indextáblával */
int BinKerIt(Tomb a, IndexTabla it, int n, Elem x, int *Hol)
{ int i,j,k,Van;
  i=0; j=n-1; Van=0;
  while (i<=j && !Van) {
    k=(i+j)/2;
    if (a[it[k]]==x) Van=1;
    else
      if (x<a[it[k]]) j=k-1;
      else i=k+1;
  }
  if (Van) *Hol=k;
  else *Hol=i;
  return (Van);
}

/* Elemek generálása */
void General(Tomb a, int n)
{ int i;
  for (i=0; i<n; i++) a[i]=random(MaxElem+1);
}
```

```
/* Elemek áttétele Mibol Mibe */
void Attesz(Tomb Mibol, Tomb Mibe, int n)
{ int i;
  for (i=0; i<n; i++) Mibe[i]=Mibol[i];
}

void Kiiras(char *Szoveg, Tomb a, int n)
{ int i;
  printf("%s\n",Szoveg);
  for (i=0; i<n; i++) printf("%d ",a[i]);
  printf("\n");
}

void KiirasIt(char *Szoveg, Tomb a, IndexTabla it, int n)
{ int i;
  printf("%s\n",Szoveg);
  for (i=0; i<n; i++) printf("%d ",a[it[i]]);
  printf("\n");
}

void main(void)
{ Tomb a,b; IndexTabla it; Elem x; int Hol;
  randomize(); clrscr();

  General(b,MaxElemSzam);
  Kiiras("Rendezések előtt",b,MaxElemSzam);

  Attesz(b,a,MaxElemSzam);
  BubRend(a,MaxElemSzam);
  Kiiras("Buborék rendezés után",a,MaxElemSzam);

  Attesz(b,a,MaxElemSzam);
  KivalRend(a,MaxElemSzam);
  Kiiras("Kiválasztásos rendezés után",a,MaxElemSzam);

  Attesz(b,a,MaxElemSzam);
  BeszurRend(a,MaxElemSzam);
  Kiiras("Beszúrásos rendezés után",a,MaxElemSzam);

  KivalRendIt(b,it,MaxElemSzam);
  KiirasIt("Egy indextáblás rendezés után",b,it,MaxElemSzam);

  printf("Mit keressünk:"); scanf("%d",&x);
  if (LinKer(b,MaxElemSzam,x,&Hol))
    printf("A rendezetlen adatok közötti indexe:%d\n",Hol);
  else
    printf("Nincs a rendezetlen adatok között!\n");
  if (LinKerRend(a,MaxElemSzam,x,&Hol))
    printf("A rendezett adatok közötti indexe:%d\n",Hol);
  else
    printf("Nincs a rendezett adatok között, helye:%d\n",Hol);
  if (BinKer(a,MaxElemSzam,x,&Hol))
```

```
    printf("A rendezett adatok közötti indexe:%d\n",Hol);
  else
    printf("Nincs a rendezett adatok között, helye:%d\n",Hol);
  if (BinKerIt(b,it,MaxElemSzam,x,&Hol))
    printf("Indextáblabeli indexe:%d\n",Hol);
  else
    printf("Indextáblabeli helye:%d\n",Hol);
}
```

17.1.30. Ellenőrzött input

```
/* ELLINP.C : Ellenőrzött input */

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define Max 256       /* Karakterhalmazok maximális elemszáma */
#define Sotet 0       /* A színkiemeléshez */
#define Vilagos 7
#define Kilep 27      /* Esc */        /* Vezérlőbillentyűk */
#define AdatVeg 13    /* Enter */
#define Torol 8       /* Backspace */

#define MaxRszHossz 6       /* A rendszám maximális hossza */
#define MaxEgSzHossz 11     /* Az egész szám maximális hossza */
#define Elojel '-'          /* Előjel */

typedef int Jelek[Max];

void Ures(int *Jelek)
{ int i;
  for (i=0; i<Max; i++) Jelek[i]=0;
}

void Bovit(int *Mit, int Tol, int Ig)
{ int i;
  for (i=Tol; i<=Ig; i++) Mit[i]=1;
}

void Csokkent(int *Mit, int Tol, int Ig)
{ int i;
  for (i=Tol; i<=Ig; i++) Mit[i]=0;
}

void Bovit1(int *Mit, int Mivel)
{ Mit[Mivel]=1;
}

void Csokkent1(int *Mit, int Mivel)
{ Mit[Mivel]=0;
}
```

```
int Eleme(int Mi, int *Minek)
{ return (Minek[Mi]==1);
}

void Unio(int *Eredmeny, int *Mit, int *Mivel)
{ int i;
  for (i=0; i<=Max; i++)
    Eredmeny[i]=Eleme(i,Mit) || Eleme(i,Mivel);
}

void Metszet(int *Eredmeny, int *Mit, int *Mivel)
{ int i;
  for (i=0; i<=Max; i++)
    Eredmeny[i]=Eleme(i,Mit) && Eleme(i,Mivel);
}

int BillBe(int *Normal, int *Dupla, int *Duplae)
{ int Jel,JoJel;
  do {
    Jel=getch(); *Duplae=Jel==0;
    if (*Duplae) {
      Jel=getch();
      JoJel=Dupla[Jel];
    } else
      JoJel=Normal[Jel];
  } while (!JoJel);
  return (Jel);
}

int JelBe(int *JoJelek)
{ int Duplae; Jelek d;
  Ures(d);
  return (BillBe(JoJelek,d,&Duplae));
}

char *JobbTolt(char *Mit, int Hossz)
{ while (strlen(Mit)<Hossz)
    strcat(Mit," ");
  return (Mit);
}

void NormalIr(void)
{ textcolor(Vilagos); textbackground(Sotet);
}

void InverzIr(void)
{ textcolor(Sotet); textbackground(Vilagos);
}

void Ir(char *Mit, int Oszl, int Sor)
{ if (Oszl==0) Oszl=wherex();
  if (Sor==0) Sor=wherey();
```

```
  gotoxy(Oszl,Sor); cprintf(Mit);
}

void KiIras(char *Mit, int Oszl, int Sor, int Hossz)
{ Ir(Mit,Oszl,Sor); gotoxy(Oszl+Hossz,Sor);
}

void RendSzamBe(char *RendSzam,      /* i/o eredmény szöveg  */
                int Oszl, int Sor,   /* i   képernyőpozíció  */
                int *VanAdat         /* o   van-e új adat    */)
{ Jelek JoJel,AdatJel,SzamJegyek,Betuk;
  char Jel;
  int Hossz;
  char UresRsz[MaxRszHossz+1];

  /* Halmazok kezdőértékei */
  Ures(SzamJegyek); Bovit(SzamJegyek,'0','9');
  Ures(Betuk); Bovit(Betuk,'A','Z'); Bovit(Betuk,'a','z');
  Unio(AdatJel,Betuk,SzamJegyek);
  /* Előkészítés */
  /* Adat */
  Hossz=strlen(RendSzam); JobbTolt(RendSzam,MaxRszHossz);
  /* Képernyő */
  if (Oszl==0) Oszl=wherex();
  if (Sor==0) Sor=wherey();
  InverzIr();
  KiIras(RendSzam,Oszl,Sor,Hossz);
  /* Beolvasás */
  do {
    /* JoJel beállítás */
    Ures(JoJel);
    Bovit1(JoJel,Kilep);
    if (Hossz<=1)
      Unio(JoJel,JoJel,Betuk);
    else if (Hossz==2)
      Unio(JoJel,JoJel,AdatJel);
    else if (Hossz<=5)
      Unio(JoJel,JoJel,SzamJegyek);
    if (Hossz>0) Bovit1(JoJel,Torol);
    /* Utolsó számjegy */
    if (Hossz==5 && (Eleme(RendSzam[2],Betuk) ||
       RendSzam[2]=='0') && RendSzam[3]=='0' &&
       RendSzam[4]=='0')
      Csokkent1(JoJel,'0');
    if (Hossz==MaxRszHossz) Bovit1(JoJel,AdatVeg);
    /* Jel beolvasás */
    Jel=JelBe(JoJel);
    /* Jel feldolgozás */
    if (Eleme(Jel,AdatJel)) {
      /* Ha kisbetű, nagybetűssé alakítjuk */
      if (Jel>='a' && Jel<='z') Jel=Jel-32;
      RendSzam[Hossz]=Jel; Hossz++;
```

```
    } else if (Jel==Torol) {
      Hossz--; RendSzam[Hossz]=' ';
    }
    KiIras(RendSzam,Oszl,Sor,Hossz);
  } while (Jel!=Kilep && Jel!=AdatVeg);
  /* Befejezés */
  *VanAdat=Jel!=Kilep;
  /* Inverz kiemelés levétele */
  NormalIr();
  UresRsz[0]='\0'; JobbTolt(UresRsz,MaxRszHossz);
  KiIras(UresRsz,Oszl,Sor,MaxRszHossz);
  /* Hosszbeállítás */
  if (*VanAdat) RendSzam[Hossz]='\0';
  else RendSzam[0]='\0';
  KiIras(RendSzam,Oszl,Sor,Hossz);
}

void EgSzamBe(char *SzamSzov,        /* i/o eredmény szöveg  */
              int Oszl, int Sor,     /* i   képernyőpozíció  */
              long Tol, long Ig,     /* i   határok          */
              int *VanAdat,          /* o   van-e új adat    */
              long *SzamErt          /* o   az eredmény szám */)
{ Jelek JoJel,SzamJegyek;
  char Jel;
  int Hossz,JoAdat,MaxH;
  char UresEgSz[MaxEgSzHossz+1];
  char w[MaxEgSzHossz+1];
  double x;

  /* Halmazok kezdőértékei */
  Ures(SzamJegyek); Bovit(SzamJegyek,'0','9');
  /* Előkészítés */
  /* Adat */
  ltoa(Tol,w,10); MaxH=strlen(w); ltoa(Ig,w,10);
  if (MaxH<strlen(w)) MaxH=strlen(w);
  Hossz=strlen(SzamSzov); JobbTolt(SzamSzov,MaxH);
  /* Képernyő */
  if (Oszl==0) Oszl=wherex();
  if (Sor==0) Sor=wherey();
  InverzIr();
  KiIras(SzamSzov,Oszl,Sor,Hossz);
  /* Beolvasás */
  do {
    /* Tartalmi helyesség: Tol-Ig */
    do {
      /* Formai helyesség: csak szám */
      /* JoJel beállítás */
      Ures(JoJel);
      Bovit1(JoJel,Kilep);
      if (Hossz<MaxH) Unio(JoJel,JoJel,SzamJegyek);
      if (Hossz==0 && Tol<0) Bovit1(JoJel,Elojel);
      if (Hossz>0) Bovit1(JoJel,Torol);
```

```
      if (Hossz>0 && Eleme(SzamSzov[Hossz-1],SzamJegyek))
        Bovit1(JoJel,AdatVeg);
      /* Jel beolvasás */
      Jel=JelBe(JoJel);
      /* Jel feldolgozás */
      if (Eleme(Jel,SzamJegyek) || Jel==Elojel) {
        SzamSzov[Hossz]=Jel; Hossz++;
      } else if (Jel==Torol) {
        Hossz--; SzamSzov[Hossz]=' ';
      }
      KiIras(SzamSzov,Oszl,Sor,Hossz);
    } while (Jel!=Kilep && Jel!=AdatVeg);
    /* Tartomány ellenőrzés */
    if (Jel!=Kilep) {
      w[0]='\0'; strncat(w,SzamSzov,Hossz); x=atof(w);
      JoAdat=(x>=Tol && x<=Ig);
    };
  } while (Jel!=Kilep && !JoAdat);
  /* Befejezés */
  *VanAdat=Jel!=Kilep;
  /* Inverz kiemelés levétele */
  NormalIr();
  UresEgSz[0]='\0'; JobbTolt(UresEgSz,MaxH);
  KiIras(UresEgSz,Oszl,Sor,MaxH);
  /* Hosszbeállítás */
  if (*VanAdat) {
    SzamSzov[Hossz]='\0';
    *SzamErt=x;
  } else SzamSzov[0]='\0';
  KiIras(SzamSzov,Oszl,Sor,Hossz);
}

void main()
{ char st[20]; int van; long l;

  NormalIr(); clrscr();

  st[0]='\0';
  printf("Rendszám:");
  RendSzamBe(st,0,0,&van);
  printf("\n");
  if (van)
    printf("A megadott rendszám:%s\n",st);
  else
    printf("Nem adott meg adatot!\n");
  printf("\n");

  st[0]='\0';
  printf("Egész szám [-128,127]:");
  EgSzamBe(st,0,0,-128,127,&van,&l);
  printf("\n");
  if (van)  {
```

```
    printf("A megadott szám sztringként:%s\n",st);
    printf("A megadott szám számként:%ld\n",l);
  } else
    printf("Nem adott meg adatot!");
}
```

17.1.31. Faktoriális

```
/* FAKTOR_R.C : Faktoriális kiszámítása rekurzív függvénnyel */

#include <conio.h>
#include <stdio.h>

long double Fakt(int n)
{ long double er;
  if (n==0) er=1;
  else er=Fakt(n-1)*n;
  return (er);
}

void main(void)
{ int n;
  clrscr();
  printf("N értéke:"); scanf("%d",&n);
  printf("%d!=%0.0Lf\n",n,Fakt(n));
}
```

17.1.32. Gyorsrendezés

```
/* GYRENDR.C : Gyorsrendezés rekurzívan */

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MaxElemSzam 10   /* A rendezendő adatok max. száma */
#define MaxElem 20       /* A generálandó adatok max. értéke */

typedef int Elem;               /* A rendezendő elemek típusa */
typedef Elem Tomb[MaxElemSzam]; /* A rendezendő elemek tömbje */

/* A rekurzív eljárás amely rendezi az 'a' tömb 'k'-'v' indexű
elemeit */
void GyorsRend(Tomb a, int k, int v)
{ Elem s,cs; int i,j;
  if (k<v) {
    /* Van legalább két elem */
    i=k; j=v; s=a[(i+j)/2];
    /* Szétválogatás a strázsa (s) elemhez képest */
    while (i<=j) {
      while (a[i]<s) i++;
```

```
      while (a[j]>s) j--;
      if (i<=j) {
        /* Az i. és j. elemek cseréje */
        cs=a[i]; a[i]=a[j]; a[j]=cs;
        i++; j--;
      }
    }
    /* Az első rész rendezése rekurzív hívással */
    GyorsRend(a,k,j);
    /* A második rész rendezése rekurzív hívással */
    GyorsRend(a,i,v);
  }
}

/* Elemek generálása */
void General(Tomb a, int n)
{ int i;
  for (i=0; i<n; i++) a[i]=random(MaxElem+1);
}

void Kiiras(char *Szoveg, Tomb a, int n)
{ int i;
  printf("%s\n",Szoveg);
  for (i=0; i<n; i++) printf("%d ",a[i]);
  printf("\n");
}

void main(void)
{ Tomb a;
  randomize(); clrscr();

  General(a,MaxElemSzam);
  Kiiras("Rendezés előtt",a,MaxElemSzam);

  /* Sorbarendezés */
  GyorsRend(a,0,MaxElemSzam-1);

  Kiiras("Rendezés után",a,MaxElemSzam);
}
```

17.1.33. Kínai gyűrűk

```
/* KINAI.C : Kínai gyűrűk */

#include <conio.h>
#include <stdio.h>

#define Max 9           /* A gyűrűk maximális száma */

int Gyuruk[Max];        /* A gyűrűk állapota (1:fent, 0:lent) */
int n;                  /* A gyűrűk száma */
int l=0;                /* A mozgatási lépések száma */
```

```
/* A gyűrűk állapotának kiírása */
void Kiir(void)
{ int i;
  if (l==0) printf("Kezdőállapot\n");
  else printf("%d. lépés\n",l);
  for (i=0; i<n; i++)
    printf("%d",Gyuruk[i]);
  printf("\n");
  l++; getch();
}

/* Az n. és a megelőző gyűrűk levétele */
void BalraLe(int n)
{ void Le(int n);
  int i;
  for (i=n; i>=0; i--) Le(i);
}

/* Az n. gyűrű levétele */
void Le(int n)
{ void Fel(int n);
  if (Gyuruk[n]) {
    if (n>0) {
      Fel(n-1);
      if (n>1) BalraLe(n-2);
    }
    Gyuruk[n]=0;
    Kiir();
  }
}

/* Az n. gyűrű feltétele */
void Fel(int n)
{ if (!Gyuruk[n]) {
    if (n>0) {
      Fel(n-1);
      if (n>1) BalraLe(n-2);
    }
    Gyuruk[n]=1;
    Kiir();
  }
}

void main()
{ int i;
  clrscr();
  printf("A gyűrűk száma (1-%d):",Max);
  scanf("%d",&n);

  /* Kezdőállapot */
  for (i=0; i<n; i++) Gyuruk[i]=1;
  Kiir();
```

```
  /* A fő eljárás meghívása */
  BalraLe(n-1);
}
```

17.1.34. Huszár útja a sakktáblán

```
/* HUSZAR.C : Huszár útja a sakktáblán */

#include <stdio.h>
#include <conio.h>

#define MaxMeret 8       /* A tábla maximális mérete */
#define KellKiiras 0     /* Legyen-e kiírás a lépések után */

typedef struct {
  int S,O;
} Mezo;

/* A lehetséges 8 lépésirány relatív elmozdulásai */
const int LepS[8]={-2,-1,1,2,2,1,-1,-2};
const int LepO[8]={1,2,2,1,-1,-2,-2,-1};

/* Globális változók */
int N;                              /* A tábla mérete */
int Tabla[MaxMeret][MaxMeret];      /* A tábla */
int VanMego;                        /* Van-e már megoldás */

void Kiir(int Lepes)
{ int i,j;
  if (Lepes)
    printf("%d. lépés után\n",Lepes);
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++)
      printf("%4d",Tabla[i][j]);
    printf("\n");
  }; printf("\n");
  getch();
}

/* A huszár következő lépése */
void Probal(int Lepes, Mezo *Akt)
{ int Irany;
  Mezo Kov;
  /* A választás előkészítése */
  Irany=0;
  do {
    /* Válasszuk ki a következő választást */
    Kov.S=(*Akt).S+LepS[Irany];
    Kov.O=(*Akt).O+LepO[Irany];
    /* Megfelelő? */
    if (Kov.S>=0 && Kov.S<=N-1 && Kov.O>=0 && Kov.O<=N-1 &&
       Tabla[Kov.S][Kov.O]==0) {
```

```
      /* Jegyezzük fel */
      Tabla[Kov.S][Kov.O]=Lepes;
      if (KellKiiras)
        Kiir(Lepes);
      /* A megoldás nem teljes? */
      if (Lepes<N*N) {
        /* Rekurzív hívás */
        Probal(Lepes+1,&Kov);
        if (!VanMego)
          /* A feljegyzés törlése */
          Tabla[Kov.S][Kov.O]=0;
      } else
        VanMego=1;
    }
    Irany++;
  } while (!VanMego && Irany<8);
}

void main()
{ Mezo Km;
  int i,j;
  clrscr();
  printf("A sakktábla mérete (3-%d):",MaxMeret);
  scanf("%d",&N);
  printf("A kezdőmező sora (1-%d):",N);
  scanf("%d",&Km.S); Km.S--;
  printf("A kezdőmező oszlopa (1-%d):",N);
  scanf("%d",&Km.O); Km.O--;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      Tabla[i][j]=0;
  Tabla[Km.S][Km.O]=1;
  VanMego=0;

  printf("Dolgozom...\n");

  Probal(2,&Km);

  if (VanMego) {
    printf("\nA megtalált megoldás\n");
    Kiir(0);
  } else
    printf("Nincs megoldás!\n");
}
```

17.1.35. Nyolc királynő

```
/* KIRALYNO.C : A 8 királynő probléma */

#include <stdio.h>
#include <conio.h>
```

```
#define MaxMeret 8                /* A tábla maximális mérete */

/* Globális változók (A tömbök 0. elemeit nem használjuk) */
int n;                          /* A tábla mérete */
int Hol[MaxMeret+1];            /* A királynők helye */
int FAtl[2*MaxMeret];           /* A főátlók foglaltsága */
int MAtl[2*MaxMeret];           /* A mellékátlók foglaltsága */
int Oszl[2*MaxMeret];           /* Az oszlopok foglaltsága */
int db;                         /* Az összes megoldás darabszáma */

void SzinValt(int Szoveg, int Hatter)
{ textcolor(Szoveg);
  textbackground(Hatter);
}

void MegoKiir(int Kiiras)
{ int i,j,k;
  db++; printf("%d. megoldás\n",db);
  if (Kiiras) {
    /* Sima kiírás */
    for (i=1; i<=n; i++) printf("%4d",Hol[i]); printf("\n");
  } else {
    /* Táblás megjelenítés */
    printf("╔");
    for (k=1; k<=3*n; k++) printf("═");
    printf("╗\n");
    for (i=1; i<=n; i++) {
      printf("║");
      for (j=1; j<=n; j++) {
        if (i%2==0 && j%2==0 || i%2==1 && j%2==1) SzinValt(0,7);
        else SzinValt(7,0);
        cprintf("   ");
        if (Hol[i]==j) {
          gotoxy(wherex()-3,wherey()); printf("\\%c/",127);
        }
      }
      SzinValt(7,0); printf("║\n");
    }
    printf("╚");
    for (k=1; k<=3*n; k++) printf("═");
    printf("╝\n");
  }
  getch();
}

/* Egy királynő elhelyezése az S. sorba */
void Probal(int s)
{ int o;
  /* Az összes választáson */
  for (o=1; o<=n; o++) {
    /* s. sorba az o. oszlopba téve az s. királynőt */
    /* Megfelelő? */
```

```
    if (!Oszl[o] && !FAtl[s-o+n] && !MAtl[s+o-1]) {
      /* Jegyezzük fel */
      Hol[s]=o;
      Oszl[o]=1;
      FAtl[s-o+n]=1;
      MAtl[s+o-1]=1;
      /* A megoldás nem teljes? */
      if (s<n)
        /* Rekurzív hívás */
        Probal(s+1);
      else
        MegoKiir(0);
      /* A feljegyzés törlése */
      Oszl[o]=0;
      FAtl[s-o+n]=0;
      MAtl[s+o-1]=0;
    }
  }
}

void main()
{ int i;
  clrscr();
  printf("A sakktábla mérete (1-%d):",MaxMeret); scanf("%d",&n);
  for (i=1; i<=n; i++)
    Oszl[i]=0;
  for (i=1; i<=2*n-1; i++) {
    FAtl[i]=0; MAtl[i]=0;
  }
  db=0;
  Probal(1);
  if (db==0) printf("Nincs megoldás!");
  else printf("%d db megoldás létezett!",db);
}
```

17.1.36. Gyorsrendezés saját veremmel

```
/* GYRENDNR.C : Gyorsrendezés nem rekurzívan, saját veremmel */

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MaxElemSzam 10    /* A rendezendő adatok max. száma */
#define MaxElem 20        /* A generálandó adatok max. értéke */

typedef int Elem;               /* A rendezendő elemek típusa */
typedef Elem Tomb[MaxElemSzam]; /* A rendezendő elemek tömbje */

#define MaxVeremMeret 4  /* A verem mérete (log2(MaxElemSzam) */
typedef struct {          /* A veremben tárolt elemek típusa */
  int k,v;                /* A rendezendő elemek indexhatárai */
```

```
} VeremElem;
typedef VeremElem TVerem[MaxVeremMeret];    /* A verem típusa */

/* Adat betétele a verembe */
void Verembe(TVerem Verem, int *VeremMut, int k, int v)
{ (*VeremMut)++;
  Verem[*VeremMut].k=k;
  Verem[*VeremMut].v=v;
}

/* Adat kivétele a veremből */
void Verembol(TVerem Verem, int *VeremMut, int *k, int *v)
{ *k=Verem[*VeremMut].k;
  *v=Verem[*VeremMut].v;
  (*VeremMut)--;
}

/* A rendező eljárás */
void GyorsRend(Tomb a, int n)
{ TVerem Verem;
  Elem s,cs;
  int VeremMut,k,v,i,j;

  /* A verem inicializálása */
  VeremMut=-1;  /* Mert 0-tól indexelünk */
  /* A teljes rendezendő részt a verembe */
  Verembe(Verem,&VeremMut,0,n-1);
  while (VeremMut>-1) {
    /* A rendezendő rész kivétele a veremből */
    Verembol(Verem,&VeremMut,&k,&v);
    if (k<v) {
      /* Van legalább két elem */
      i=k; j=v; s=a[(i+j)/2];
      /* Szétválogatás a strázsa (s) elemhez képest */
      while (i<=j) {
        while (a[i]<s) i++;
        while (a[j]>s) j--;
        if (i<=j) {
          /* Az i. és j. elemek cseréje */
          cs=a[i]; a[i]=a[j]; a[j]=cs;
          i++; j--;
        }
      }
      /* A két rész felvétele a verembe (felülre a kisebbet) */
      if (j-k>v-i) {
        Verembe(Verem,&VeremMut,k,j);
        Verembe(Verem,&VeremMut,i,v);
      } else {
        Verembe(Verem,&VeremMut,i,v);
        Verembe(Verem,&VeremMut,k,j);
      }
    }
```

```
  }
}

/* Elemek generálása */
void General(Tomb a, int n)
{ int i;
  for (i=0; i<n; i++) a[i]=random(MaxElem+1);
}

void Kiiras(char *Szoveg, Tomb a, int n)
{ int i;
  printf("%s\n",Szoveg);
  for (i=0; i<n; i++) printf("%d ",a[i]);
  printf("\n");
}

void main(void)
{ Tomb a;
  randomize(); clrscr();

  General(a,MaxElemSzam);
  Kiiras("Rendezés előtt",a,MaxElemSzam);

  /* Sorbarendezés */
  GyorsRend(a,MaxElemSzam);

  Kiiras("Rendezés után",a,MaxElemSzam);
}
```

17.1.37. Kollekciók

```
/* KOLL.C : Kollekció lottószelvények kiértékelésére */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>

char *FNevI="KOLL_I.TXT";    /* A szelvények szövegfájlja */
char *FNevO="KOLL_O.TXT";    /* Az eredmény szövegfájl */

#define MaxSzelv 2000  /* A betölthető szelvények max. száma */
#define MaxGen 2200    /* A generálandó szelvények száma */
#define Db 6           /* A lottószámok száma egy szelvényen */
#define Max 45         /* Egy lottószám maximális értéke */
#define MinTal 3       /* A minimális találatszám a nyeréshez */

typedef int Szelveny[Db];     /* Lottószelvény */
typedef struct {              /* A kollekció tétele */
  Szelveny Sz;                /* A számok */
  int Tal;                    /* Találatok száma a szelvényen */
} Tetel;
```

```
typedef Tetel *Kollekcio[MaxSzelv];      /* A kollekció */
typedef int Stat[Db+1];                  /* Statisztika */
typedef int Halmaz[Max];                 /* A halmaz */

/* Egy db lottószelvény generálása */
void General(Szelveny Sz)
{ Halmaz h; int i,j,x;
  for (i=0; i<Max; i++) h[i]=0;
  for (i=0; i<Db; i++) {
    do {
      x=random(Max);
    } while (h[x]);
    h[x]=1;
  }
  i=0;
  for (j=0; j<Max; j++)
    if (h[j]) {
      Sz[i]=j+1; i++;
    }
}

/* N db lottószelvény generálása az FNev nevű szövegfájlba */
void FajlbaGeneral(char *FNev, int N)
{ FILE *f; Szelveny Sz;
  int i,j;
  f=fopen(FNev,"w");
  for (i=0; i<N; i++) {
    General(Sz);
    for (j=0; j<Db; j++) fprintf(f,"%3d",Sz[j]);
    fprintf(f,"\n");
  }
  fclose(f);
}

/* Az FNev nevű szövegfájl szelvényeinek betöltése */
int Betolt(char *FNev, Kollekcio K, int *KDb)
{ FILE *f; int i,j,Kilep,Ok; Szelveny Sz;
  /* Adatbeolvasás */
  f=fopen(FNev,"r");
  *KDb=0; Kilep=0;
  while (!Kilep) {
    if (*KDb>=MaxSzelv) Kilep=1;
    else {
      for (i=0; i<Db; i++)
        if (fscanf(f,"%d",&Sz[i])!=1) Kilep=1;
      if (!Kilep) {
        /* Van adat */
        if ((K[*KDb]=(Tetel*)malloc(sizeof(Tetel)))!=NULL) {
          /* Van hely */
          for (j=0; j<Db; j++)
            (*(K[*KDb])).Sz[j]=Sz[j];
          (*KDb)++;
```

```
        } else Kilep=1;
      }
    }
  }
  /* Nincs már több adat a fájlban? */
  Ok=(fscanf(f,"%d",&i)!=1);
  fclose(f);
  return (Ok);
}

/* Kiértékelés, a nyerőszelvények kiírása az FNev nevű szöveg-
fájlba */
void Ertekel(char *FNev, Kollekcio K, int KDb, Szelveny Ny)
{ FILE *f; Halmaz h;
  Stat S; int i,j,l;
  /* Nyerőszámok halmaza */
  for (i=0; i<Max; i++) h[i]=0;
  for (i=0; i<Db; i++) h[Ny[i]]=1;
  /* Statisztika kezdőértéke */
  for (i=0; i<=Db; i++) S[i]=0;
  /* A szelvények kiértékelése */
  for (i=0; i<KDb; i++) {
    (*K[i]).Tal=0;
    for (j=0; j<Db; j++)
      if (h[(*K[i]).Sz[j]]==1) (*K[i]).Tal++;
    /* Statisztika */
    S[(*K[i]).Tal]++;
  }
  /* Adatkiírás */
  f=fopen(FNev,"w");
  fprintf(f,"A nyerőszámok\n");
  for (i=0; i<Db; i++) fprintf(f,"%3d",Ny[i]); fprintf(f,"\n");
  /* A nyerőszelvények */
  for (l=Db; l>=MinTal; l--) {
    fprintf(f,"%d találatos szelvények száma:%d\n",l,S[l]);
    for (i=0; i<KDb; i++)
      if ((*K[i]).Tal==l) {
        for (j=0; j<Db; j++)
          fprintf(f,"%3d",(*K[i]).Sz[j]);
        fprintf(f,"\n");
      }
  }
  fclose(f);
}

void main()
{ Kollekcio K; int KDb; Szelveny Ny;
  clrscr(); randomize();
  /* Adatgenerálás */
  FajlbaGeneral(FNevI,MaxGen);
  printf("%d db szelvény generálva (%s)\n",MaxGen,FNevI);
  /* Adatbeolvasás */
```

```
  if (!Betolt(FNevI,K,&KDb))
    printf("Nem fért be minden szelvény!\n");
  printf("A betöltött szelvények száma:%d\n",KDb);
  /* Nyerőszámok generálása */
  General(Ny);
  /* Kiértékelés */
  Ertekel(FNevO,K,KDb,Ny);
  printf("A kiértékelés elkészült (%s)\n",FNevO);
}
```

17.1.38. Láncolt listák

```
/* LISTAK.C : Egy- és kétirányban láncolt listák */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef int Elem;

typedef struct LancElem1 {
  Elem Adat;
  struct LancElem1 *Koveto;
};

typedef struct LancElem1* LancElem1Mut;

struct LancElem2 {
  Elem Adat;
  struct LancElem2 *Elozo, *Koveto;
};
typedef struct LancElem2* LancElem2Mut;

/* Keresés egy egyirányban láncolt, rendezetlen listában */
LancElem1Mut Kereses1(LancElem1Mut Elso, Elem x)
{ LancElem1Mut Akt;
  Akt=Elso;
  while (Akt!=NULL && (*Akt).Adat!=x)
    Akt=(*Akt).Koveto;
  return (Akt);
}

/* Keresés egy egyirányban láncolt, rendezett listában */
LancElem1Mut KeresesRend1(LancElem1Mut Elso, Elem x)
{ LancElem1Mut Akt;
  Akt=Elso;
  while (Akt!=NULL && (*Akt).Adat<x)
    Akt=(*Akt).Koveto;
  if (Akt!=NULL && (*Akt).Adat>x)
    Akt=NULL;
  return (Akt);
}
```

```
/* Egy egyirányban láncolt, rendezett lista bővítése egy új lis-
taelemmel */
void ListaraRend1(LancElem1Mut *Elso, LancElem1Mut Uj)
{ LancElem1Mut Akt,Miutan;
  /* Keresés */
  Akt=*Elso; Miutan=NULL;
  while (Akt!=NULL && (*Akt).Adat<(*Uj).Adat) {
    Miutan=Akt;
    Akt=(*Akt).Koveto;
  }
  /* Beillesztés */
  if (*Elso==NULL) {
    /* Üres listára */
    (*Uj).Koveto=NULL;
    *Elso=Uj;
  } else if (Miutan==NULL) {
    /* Nem üres lista elejére */
    (*Uj).Koveto=*Elso;
    *Elso=Uj;
  } else if (Akt==NULL) {
    /* A lista végére, a Miutan után */
    (*Uj).Koveto=NULL;
    (*Miutan).Koveto=Uj;
  } else {
    /* A Miutan és az Akt közé */
    (*Uj).Koveto=(*Miutan).Koveto;
    (*Miutan).Koveto=Uj;
  }
}

/* Beillesztés egy kétirányban láncolt lista végére */
void Listara2(LancElem2Mut *Elso, LancElem2Mut *Utolso,
LancElem2Mut Uj)
{ if (*Elso==NULL) {
    /* Üres listára */
    (*Uj).Elozo=NULL;
    (*Uj).Koveto=NULL;
    *Elso=Uj;
    *Utolso=Uj;
  } else {
    /* Az Utolso után */
    (*Uj).Elozo=*Utolso;
    (*Uj).Koveto=NULL;
    (**Utolso).Koveto=Uj;
    *Utolso=Uj;
  }
}

/* Elem törlése egy kétirányban láncolt listából */
void Listarol2(LancElem2Mut *Elso, LancElem2Mut *Utolso,
LancElem2Mut Mit)
```

```
{ /* Kikapcsolás */
  if (Mit==*Elso && Mit==*Utolso) {
    /* Egyetlen elem */
    *Elso=NULL;
    *Utolso=NULL;
  } else if (Mit==*Elso) {
    /* Első, de nem egyetlen */
    (*(*Mit).Koveto).Elozo=NULL;
    *Elso=(*Mit).Koveto;
  } else if (Mit==*Utolso) {
    /* Utolsó, de nem egyetlen */
    (*(*Mit).Elozo).Koveto=NULL;
    *Utolso=(*Mit).Elozo;
  } else {
    /* Belső elem */
    (*(*Mit).Elozo).Koveto=(*Mit).Koveto;
    (*(*Mit).Koveto).Elozo=(*Mit).Elozo;
  }
  /* Megszüntetés */
  free(Mit);
}

void main()
{ LancElem1Mut Elso1,Akt1;
  LancElem2Mut Elso2,Utolso2,Akt2;
  Elem a; int i;
  clrscr();
  Elso1=NULL;
  printf("Adatmegadás vége:0\n"); i=1;
  do {
    printf("%d. elem:",i++); scanf("%d",&a);
    if (a!=0) {
      if ((Akt1=(LancElem1Mut)malloc(sizeof(
         struct LancElem1)))!=NULL) {
        /* Van hely, a rendezett láncba illesszük */
        (*Akt1).Adat=a;
        ListaraRend1(&Elso1,Akt1);
      }
    }
  } while (a!=0);

  printf("A megadott elemek rendezve:\n");
  for (Akt1=Elso1, i=1; Akt1!=NULL; Akt1=(*Akt1).Koveto, i++)
    printf("%d. elem:%d\n",i,(*Akt1).Adat);

  printf("A keresett elem:"); scanf("%d",&a);
  if (Kereses1(Elso1,a)==NULL)
    printf("Nincs ilyen!\n");
  else
    printf("Van ilyen!\n");
  if (KeresesRend1(Elso1,a)==NULL)
    printf("Nincs ilyen!\n");
```

```
  else
    printf("Van ilyen!\n");

  /* A foglalt hely felszabadítása */
  while (Elso1!=NULL) {
    Akt1=Elso1;
    Elso1=(*Elso1).Koveto;
    free(Akt1);
  }

  Elso2=NULL; Utolso2=NULL;
  printf("Adatmegadás vége:0\n"); i=1;
  do {
    printf("%d. elem:",i++); scanf("%d",&a);
    if (a!=0) {
      if ((Akt2=(LancElem2Mut)malloc(sizeof(
          struct LancElem2)))!=NULL) {
        /* Van hely, a lánc végére illesszük */
        (*Akt2).Adat=a;
        Listara2(&Elso2,&Utolso2,Akt2);
      }
    }
  } while (a!=0);

  printf("A megadott elemek eredeti sorrendben:\n");
  for (Akt2=Elso2, i=1; Akt2!=NULL; Akt2=(*Akt2).Koveto, i++)
    printf("%d. elem:%d\n",i,(*Akt2).Adat);

  printf("A megadott elemek fordított sorrendben:\n");
  for (Akt2=Utolso2, i=1; Akt2!=NULL; Akt2=(*Akt2).Elozo, i++)
    printf("%d. elem:%d\n",i,(*Akt2).Adat);

  /* A foglalt hely felszabadítása */
  /* (lehetne az utolsó törlésével is) */
  while (Elso2!=NULL)
    Listarol2(&Elso2,&Utolso2,Elso2);
}
```

17.1.39. Összetett listák

```
/* TARGYMUT.C : Tárgymutató */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>

#define MaxSzoHossz 20           /* A szavak maximális hossza */

typedef char TSzo[MaxSzoHossz+1];    /* A szavak típusa */
typedef unsigned int THiv;           /* A hivatkozások típusa */
```

```
struct HivRek {                          /* A hivatkozásrekord */
  THiv Oldal;
  struct HivRek *Kov;
};
typedef struct HivRek * HivRekMut;  /* A hiv. rekordra mutató */

struct SzoRek {                                  /* A szórekord */
  TSzo Szo;
  struct HivRek *EHiv,*UHiv;
  struct SzoRek *Kov;
};
typedef struct SzoRek * SzoRekMut;    /* A szórekordra mutató */

/* Új szórekord létrehozása */
SzoRekMut UjSzoRek(TSzo Szo, THiv Oldal, SzoRekMut KovSzo)
{ SzoRekMut UjSzo; HivRekMut UjHiv;
  UjSzo=(SzoRekMut)malloc(sizeof(struct SzoRek));
  strcpy((*UjSzo).Szo,Szo);
  (*UjSzo).Kov=KovSzo;
  UjHiv=(HivRekMut)malloc(sizeof(struct HivRek));
  (*UjSzo).EHiv=UjHiv;
  (*UjSzo).UHiv=UjHiv;
  (*UjHiv).Oldal=Oldal;
  (*UjHiv).Kov=NULL;
  return (UjSzo);
}

/* A tárgymutató bővítése */
void Bovit(SzoRekMut *TMKezd, TSzo Szo, THiv Oldal)
{ SzoRekMut Akt,Elozo; HivRekMut UjHiv;
  if (*TMKezd==NULL)
    /* Üres a tárgymutató */
    *TMKezd=UjSzoRek(Szo,Oldal,NULL);
  else {
    /* Keresés */
    Akt=*TMKezd;
    Elozo=NULL;
    while (strcmp((*Akt).Szo,Szo)<0 && (*Akt).Kov!=NULL) {
      Elozo=Akt;
      Akt=(*Akt).Kov;
    }
    if (strcmp((*Akt).Szo,Szo)<0)
      /* Végére új szó */
      (*Akt).Kov=UjSzoRek(Szo,Oldal,NULL);
    else if (strcmp((*Akt).Szo,Szo)>0)
      /* Az Akt és az Elozo közé új szó */
      if (Elozo==NULL)
        /* A lista elejére */
        *TMKezd=UjSzoRek(Szo,Oldal,*TMKezd);
      else
        /* A lista belsejébe */
        (*Elozo).Kov=UjSzoRek(Szo,Oldal,Akt);
```

```
    else {
      /* Meglévő szó (Akt) új hivatkozása */
      UjHiv=(HivRekMut)malloc(sizeof(struct HivRek));
      (*UjHiv).Oldal=Oldal;
      (*UjHiv).Kov=NULL;
      (*(*Akt).UHiv).Kov=UjHiv;
      (*Akt).UHiv=UjHiv;
    }
  }
}

/* A tárgymutató adatainak kiírása a képernyőre */
void Kiir(SzoRekMut TMKezd)
{ SzoRekMut Akt; HivRekMut AktHiv;
  Akt=TMKezd;
  while (Akt!=NULL) {
    printf("%-20s",(*Akt).Szo);
    AktHiv=(*Akt).EHiv;
    while (AktHiv!=NULL) {
      printf("%6d",(*AktHiv).Oldal);
      AktHiv=(*AktHiv).Kov;
    };
    printf("\n"); getch();
    Akt=(*Akt).Kov;
  }
}

void main()
{ SzoRekMut TMKezd; int VanHely,Vege;
  HivRekMut H; SzoRekMut Sz; TSzo Szo; THiv Hiv;
  clrscr();
  TMKezd=NULL;
  /* Feltöltés input adatokkal */
  printf("A tárgymutató adatai (Kilépés:'*' szó)\n");
  do {
    /* Lesz majd elég hely? */
    Sz=(SzoRekMut)malloc(sizeof(struct SzoRek));
    H=(HivRekMut)malloc(sizeof(struct HivRek));
    VanHely=(Sz!=NULL) && (H!=NULL);
    if (VanHely) {
      /* Igen */
      free(H); free(Sz);
      printf("Szó:"); scanf("%s",&Szo);
      Vege=strcmp(Szo,"*")==0;
      if (!Vege) {
        printf("Hivatkozás:"); scanf("%d",&Hiv);
        Bovit(&TMKezd,Szo,Hiv);
      }
    }
  } while (!Vege && VanHely);
  /* Kiírás */
  if (TMKezd==NULL)
```

```
    printf("Nem adott meg adatokat!\n");
  else {
    printf("A tárgymutató:\n");
    Kiir(TMKezd);
  }
}
```

17.1.40. Szövegfájl képernyőre listázása

```
/* SZFLIST.C : Szövegfájl kilistázása karakterenként */

#include <stdio.h>
#include <conio.h>

#define db 20      /* Max. ennyi sort írunk ki egy képernyőre */

void main()
{ FILE *fp;
  int ch,s;
  clrscr(); s=0;
  fp=fopen("szflist.c","r");
  while ((ch=fgetc(fp))!=EOF) {
    printf("%c",ch);
    if (ch==10) {
      s++;
      if (s%db==0) getch();
    }
  }
  if (s%db!=0) getch();
  fclose(fp);
}
```

17.1.41. Összefésüléses fájlrendezés

```
/* TIPFREND.C : Típusos fájl rekordjainak rendezése összefésü-
léssel */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>

char *fnev1="adatokc1.dta";    /* A rendezendő adatok */
char *fnev2="adatokc2.dta";    /* Segédfájl */
char *fnev3="adatokc3.dta";    /* Segédfájl */

#define MaxElemSzam 10     /* A generált adatok max. száma */
#define MaxElem 20         /* A generált adatok max. értéke */

typedef struct {           /* A rendezendő rekordok */
  int A;                   /* Ezen mező szerint rendezünk */
} TAdat;
```

```
/* A fájlmutatót egy rekorddal vissza */
void Vissza(FILE *fp)
{ fseek(fp,(ftell(fp)/sizeof(TAdat)-1)*sizeof(TAdat),SEEK_SET);
}

/* Fájlvégén állunk-e */
int FajlVege(FILE *fp)
{ long pos,meret;
  pos=ftell(fp);
  fseek(fp,0L,SEEK_END);
  meret=ftell(fp);
  fseek(fp,pos,SEEK_SET);
  return (pos==meret);
}

int LancVege(FILE *Miben, TAdat *Akt)
{ TAdat Kov; int Er; long pos;
  if (FajlVege(Miben))
    Er=1;
  else {
    fgetpos(Miben,&pos);
    fread(&Kov,sizeof(TAdat),1,Miben);
    fsetpos(Miben,&pos);
    Er=Kov.A<(*Akt).A;
  }
  return (Er);
}

void LancMasol(FILE *Bol, FILE *Ba)
{ TAdat Akt;
  do {
    fread(&Akt,sizeof(TAdat),1,Bol);
    fwrite(&Akt,sizeof(TAdat),1,Ba);
  } while (!LancVege(Bol,&Akt));
}

void Rendezes(void)
{ FILE *a,*b,*c;
  int LancDb,LancVeg;
  TAdat akt_a,akt_b;
  do {
    /* Szétosztás */
    c=fopen(fnev1,"rb");
    a=fopen(fnev2,"wb");
    b=fopen(fnev3,"wb");
    while (!FajlVege(c)) {
      LancMasol(c,a);
      if (!FajlVege(c))
        LancMasol(c,b);
    };
    fclose(a); fclose(b); fclose(c);
```

```
    /* Összefésülés */
    c=fopen(fnev1,"wb");
    a=fopen(fnev2,"rb");
    b=fopen(fnev3,"rb");
    LancDb=0;
    while (!FajlVege(a) && !FajlVege(b)) {
      /* Egy lánc összefésülése */
      fread(&akt_a,sizeof(TAdat),1,a);
      fread(&akt_b,sizeof(TAdat),1,b);
      do {
        if (akt_a.A<=akt_b.A) {
          /* a-ból egy elemet */
          fwrite(&akt_a,sizeof(TAdat),1,c);
          LancVeg=LancVege(a,&akt_a);
          if (LancVeg) {
            Vissza(b);
            LancMasol(b,c);
          } else
          fread(&akt_a,sizeof(TAdat),1,a);
        } else {
          /* b-ből egy elemet */
          fwrite(&akt_b,sizeof(TAdat),1,c);
          LancVeg=LancVege(b,&akt_b);
          if (LancVeg) {
            Vissza(a);
            LancMasol(a,c);
          } else
          fread(&akt_b,sizeof(TAdat),1,b);
        }
      } while (!LancVeg);
      LancDb++;
    };
    /* Maradékok másolása */
    if (!FajlVege(a)) {
      while (!FajlVege(a)) {
        fread(&akt_a,sizeof(TAdat),1,a);
        fwrite(&akt_a,sizeof(TAdat),1,c);
      };
      LancDb++;
    };
    if (!FajlVege(b)) {
      while (!FajlVege(b)) {
        fread(&akt_b,sizeof(TAdat),1,b);
        fwrite(&akt_b,sizeof(TAdat),1,c);
      };
      LancDb++;
    };
    fclose(a); fclose(b); fclose(c);
  } while (LancDb>1);
}
```

```
void main(void)
{ FILE *fp;
  TAdat Adat;
  int i;

  /* Véletlen elemgenerálás */
  randomize(); clrscr();
  printf("Rendezés előtt\n");
  fp=fopen(fnev1,"wb");
  for (i=0; i<MaxElemSzam; i++) {
    Adat.A=random(MaxElem+1);
    fwrite(&Adat,sizeof(TAdat),1,fp);
    printf("%d ",Adat.A);
  }; printf("\n");
  fclose(fp);

  /* Sorbarendezés */
  Rendezes();

  /* Eredménykiírás */
  printf("Rendezés után\n");
  fp=fopen(fnev1,"rb");
  while (!FajlVege(fp)) {
    fread(&Adat,sizeof(TAdat),1,fp);
    printf("%d ",Adat.A);
  }; printf("\n");
  fclose(fp);

  /* Munkafájlok törlése */
  remove(fnev2); remove(fnev3);
}
```

17.1.42. Indextáblás fájlkezelés

```
/* TIPFINDT.C : Típusos fájl kezelése indextáblával */

#include <stdio.h>
#include <conio.h>
#include <string.h>

/* Fájlnevek */
char *afnev="adatc.dat";          /* Adatfájl */
char *ifnev="adatc.ind";          /* Indextábla */

#define MaxDb 5             /* Az adatok maximális száma */
#define MaxAzHossz 1        /* Az azonosító maximális hossza */
#define MaxInfoHossz 3      /* Az információ maximális hossza */

char *UresAz=" ";           /* Az üres azonosító */

/* +1: a végjelnek */
typedef char TAz[MaxAzHossz+1];   /* A rekordazonosító típusa */
typedef char TInfo[MaxInfoHossz+1];   /* Az információ típusa */
```

```
typedef struct {            /* Az adatrekord típusa */
  TAz Az;
  TInfo Info;
} TAdat;

typedef struct {          /* Az indextábla egy elemének típusa */
  TAz Az;                 /* Ez alapján indexezünk */
  int Poz;                /* A rekord pozíciója az adatfájlban */
} TItAdat;

/* Az indextábla típusa */
/* A 0. rekord Poz értéke az adatfájl érvényes rekordjainak szá-
ma */
typedef TItAdat TIt[MaxDb+1];

/* Bináris keresés az indextáblában */
int BinKer(TAz Mit, TIt Miben, int Db, int *Hol)
{ int i,j,k,l,Van;
  i=1; j=Db; Van=0;
  while (i<=j && !Van) {
    k=(i+j)/2; l=strcmp(Mit,Miben[k].Az);
    if (l==0) Van=1;
    else
      if (l<0) j=k-1;
      else i=k+1;
  }
  if (Van) *Hol=k;
  else *Hol=i;
  return (Van);
}

/* Beszúrás az indextáblába */
void Beszur(TItAdat *Mit, TIt *Mibe, int Hova, int *Db)
{ int i;
  /* Helykészítés */
  for (i=*Db; i>=Hova; i--)
    (*Mibe)[i+1]=(*Mibe)[i];
  /* Beszúrás */
  (*Mibe)[Hova]=*Mit;
  /* Darabszám növelés */
  (*Db)++;
}

/* Törlés az indextáblából */
void Torol(TIt *Mibol, int Honnan, int *Db)
{ int i; TItAdat s;
  /* A törlendő elem megjegyzése */
  s=(*Mibol)[Honnan];
  /* Törlés */
  for (i=Honnan+1; i<=*Db; i++)
    (*Mibol)[i-1]=(*Mibol)[i];
  /* Hogy a fájlba majd ennek helyére vegyünk fel legközelebb */
```

```
  (*Mibol)[*Db]=s;
  /* Darabszám csökkentés */
  (*Db)--;
  if (Db==0)
    /* Az elejéröl töltsük fel az adatfájlt */
    for (i=1; i<=MaxDb; i++)
      ((*Mibol)[i]).Poz=i-1;
}

/* Elem felvétel */
void Felvetel(TIt *Mibe, int *Db, FILE *af)
{ int Hol;
  TAdat Adat; TItAdat ItAdat;
  if (*Db==MaxDb)
    printf("Nem vehetö fel több elem!");
  else {
    printf("Azonosító (max %d karakter):",MaxAzHossz);
    scanf("%s",&Adat.Az);
    if (BinKer(Adat.Az,*Mibe,*Db,&Hol)==1)
      printf("Van már ilyen azonosítójú rekord!");
    else {
      printf("Információ (max %d karakter):",MaxInfoHossz);
      scanf("%s",&Adat.Info);
      /* Beírás az adatfájlba */
      fseek(af,((*Mibe)[*Db+1]).Poz*sizeof(TAdat),SEEK_SET);
      fwrite(&Adat,sizeof(TAdat),1,af);
      /* Felvétel az indextáblába */
      strcpy(ItAdat.Az,Adat.Az);
      ItAdat.Poz=((*Mibe)[*Db+1]).Poz;
      Beszur(&ItAdat,Mibe,Hol,Db);
    }
  }
}

/* Elem törlés */
void Torles(TIt *Mibol, int *Db)
{ TAz Mit; int Hol;
  if (*Db==0)
    printf("Nincs mit törölni!");
  else {
    printf("Azonosító (max %d karakter):",MaxAzHossz);
    scanf("%s",&Mit);
    if (BinKer(Mit,*Mibol,*Db,&Hol)==0)
      printf("Nincs ilyen rekord!");
    else
      Torol(Mibol,Hol,Db);
  }
}

void Kiir(int x, int y, char *Mit)
{ gotoxy(x,y);
  printf("%s",Mit);
```

```
}

/* Adatok kiírása */
void Kiiras(TIt it, FILE *af)
{ int i; TAdat Adat;
  clrscr();
  /* Fejléc 123456789012345678901234567890 */
  Kiir(1,1,"Ssz  Indextábla  Adatfájl");
  Kiir(1,2,"     Az-ó Poz    Az-ó Info");
  /* Indextábla */
  for (i=0; i<=MaxDb; i++) {
    gotoxy(2,4+i); printf("%d.",i);
    if (i<=it[0].Poz)
      Kiir(6,4+i,it[i].Az);
    else
      Kiir(6,4+i,UresAz);
    gotoxy(11,4+i); printf("%d",it[i].Poz);
  }
  /* Adatfájl */
  fseek(af,0L,SEEK_SET); i=0;
  while (fread(&Adat,sizeof(TAdat),1,af)==1) {
    Kiir(18,4+i,Adat.Az);
    Kiir(23,4+i,Adat.Info);
    i++;
  }
  gotoxy(1,4+MaxDb+2); printf("Elemek száma:%d\n",it[0].Poz);
}

/* Fájl létezésének vizsgálata */
int FajlVan(const char *Nev)
{ FILE *fp; int Er;
  fp=fopen(Nev,"r");
  if (fp==NULL)
    Er=0;
  else {
    fclose(fp);
    Er=1;
  }
  return (Er);
}

void main()
{ FILE *af,*itf;
  TIt it;
  int i;
  clrscr();
  if (FajlVan(afnev)==1 && FajlVan(ifnev)==1) {
    af=fopen(afnev,"rb+");
    /* Az indextábla betöltése */
    itf=fopen(ifnev,"rb");
    fread(&it,sizeof(TItAdat),MaxDb+1,itf);
    fclose(itf);
```

```
  } else {
    af=fopen(afnev,"wb+");
    /* Indextábla inicializálás */
    for (i=0; i<=MaxDb; i++) {
      strcpy(it[i].Az,UresAz);
      if (i==0) it[i].Poz=0;      /* A létező adatok száma */
      else it[i].Poz=i-1;         /* A leendő rekordsorszámok */
    }
  }
  /* Kilépésig */
  do {
    printf("\nFelvétel:1 Törlés:2 Kiírás:3 Kilépés:0\n");
    do {
      i=getch();
    } while (i<'0' || i>'3');
    if (i=='1')
      Felvetel(&it,&it[0].Poz,af);
    else if (i=='2')
      Torles(&it,&it[0].Poz);
    else if (i=='3')
      Kiiras(it,af);
  } while (i!='0');
  fclose(af);
  /* Az indextábla mentése */
  itf=fopen(ifnev,"wb");
  fwrite(&it,sizeof(TItAdat),MaxDb+1,itf);
  fclose(itf);
}
```

17.1.43. Bináris fák

```
/* BINKERFA.C : Bináris keresőfa */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef int Elem;
struct BinFaPont {
  Elem PontJell;
  struct BinFaPont *BalAg,*JobbAg;
};
typedef struct BinFaPont* BinFaPontMut;

/* Keresés bináris keresőfában */
BinFaPontMut Keres(BinFaPontMut Gyoker, Elem x)
{ BinFaPontMut Akt;
  Akt=Gyoker;
  while (Akt!=NULL && x!=(*Akt).PontJell)
    if (x<(*Akt).PontJell)
      Akt=(*Akt).BalAg;
    else
```

```
      Akt=(*Akt).JobbAg;
  return (Akt);
}

/* Bináris keresőfa bővítése */
void Bovit(BinFaPontMut *Gyoker, BinFaPontMut x)
{ BinFaPontMut Akt,Szulo;
  if (*Gyoker==NULL)
    *Gyoker=x;
  else {
    /* Helykeresés */
    Akt=*Gyoker; Szulo=NULL;
    while (Akt!=NULL) {
      Szulo=Akt;
      if ((*x).PontJell<(*Akt).PontJell)
        Akt=(*Akt).BalAg;
      else
        Akt=(*Akt).JobbAg;
    }
    /* Beillesztés */
    if ((*x).PontJell<(*Szulo).PontJell)
      (*Szulo).BalAg=x;
    else
      (*Szulo).JobbAg=x;
  }
}

/* Bináris keresőfa elemeinek kiírása */
void Kiir(BinFaPontMut Gyoker)
{ if (Gyoker!=NULL) {
    /* Rekurzív hívás a bal ágra */
    Kiir((*Gyoker).BalAg);
    /* A gyökérponthoz tartozó érték kiírása */
    printf("%d ",(*Gyoker).PontJell);
    /* Rekurzív hívás a bal ágra */
    Kiir((*Gyoker).JobbAg);
  }
}

void main()
{ BinFaPontMut Gyoker,Akt; Elem a; int i;
  clrscr();
  Gyoker=NULL;
  printf("Adatmegadás vége:0\n");
  i=1;
  do {
    printf("%d. elem:",i++); scanf("%d",&a);
    if (a!=0) {
      if ((Akt=(BinFaPontMut)malloc(sizeof(
         struct BinFaPont)))!=NULL) {
        /* Van hely, a keresőfába illesszük */
        (*Akt).PontJell=a;
```

```
        (*Akt).BalAg=NULL; (*Akt).JobbAg=NULL;
        Bovit(&Gyoker,Akt);
      }
    }
  } while (a!=0);

  printf("A megadott elemek rendezve:\n");
  Kiir(Gyoker);
  printf("\n");

  printf("A keresett elem:"); scanf("%d",&a);
  if (Keres(Gyoker,a)==NULL)
    printf("Nincs ilyen elem!\n");
  else
    printf("Van ilyen elem!\n");
}
```

17.1.44. Kupacrendezés

```
/* KUPAC.C : Kupacrendezés */

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MaxElemSzam 10    /* A rendezendő adatok max. száma */
#define MaxElem 20        /* A generálandó adatok max. értéke */

typedef int Elem;         /* A rendezendő elemek típusa */
typedef Elem Tomb[MaxElemSzam+1];   /* Az elemek tömbje */

/* A kupacindexek miatt a tömb 0. elemét nem használjuk, 1-től
indexelünk */

/* Az a[l] besüllyesztése az a[l+1], a[l+2],..., a[r] elemek
közé*/
void Sullyeszt(Tomb a, int l, int r)
{ int i,j,vege; Elem x;
  x=a[l]; i=l; j=2*i; vege=0;
  while (j<=r && vege==0) {
    /* A kisebb gyerekkel hasonlítsunk */
    if (j<r && a[j+1]<a[j]) j++;
    if (x<=a[j])
      /* Megvan a helye */
      vege=1;
    else {
      /* A gyerek feljebb léptetése */
      a[i]=a[j]; i=j; j=2*i;
    }
  }
  /* Elemet a helyére */
```

```
  a[i]=x;
}

void KupacRend(Tomb a, int n)
{ int l,r; Elem x;
  /* Kezdőkupac */
  l=n/2+1; r=n;
  while (l>1) {
    l--;
    Sullyeszt(a,l,r);
  }
  while (r>1) {
    /* A legkisebbet hátra */
    x=a[1]; a[1]=a[r]; a[r]=x;
    /* A kupacot kisebbre */
    r--;
    /* A hátul volt elem besüllyesztése */
    Sullyeszt(a,l,r);
  }
}

/* Elemek generálása */
void General(Tomb a, int n)
{ int i;
  for (i=1; i<=n; i++) a[i]=random(MaxElem+1);
}

void Kiiras(char *Szoveg, Tomb a, int n)
{ int i;
  printf("%s\n",Szoveg);
  for (i=1; i<=n; i++) printf("%d ",a[i]);
  printf("\n");
}

void main(void)
{ Tomb a;
  randomize(); clrscr();

  General(a,MaxElemSzam);
  Kiiras("Rendezés előtt",a,MaxElemSzam);

  /* Sorbarendezés */
  KupacRend(a,MaxElemSzam);

  Kiiras("Rendezés után",a,MaxElemSzam);
}
```

17.1.45. Útkeresés

Mátrix módszer

```
/* FLO_WAR.C : Útkeresés Floyd-Warshall módszerrel */

#include <stdio.h>
#include <conio.h>

#define MaxPontDb 5                          /* Pontok max. száma */
#define MaxElDb MaxPontDb*(MaxPontDb-1)      /* Élek max. száma */
#define MaxElHossz 99                        /* A max. élhossz */
#define NincsEl -1                           /* A nemlétező él */
#define MaxUtHossz MaxElDb*MaxElHossz        /* A max. úthossz */
#define Vegtelen MaxUtHossz+1                /* A 'végtelen'-hez */
#define KellKiiras 1                         /* Legyenek kiírások */

typedef struct {                             /* A pontrekord */
  char Azon;                                 /* Pontazonosító */
  int x,y;                                   /* Koordináták */
} Pont;

typedef int Matrix[MaxPontDb][MaxPontDb];  /* A mátrixokhoz */

typedef struct {                             /* A gráf */
  int PontDb;                                /* Pontok száma */
  Pont Pontok[MaxPontDb];                    /* Pontok */
  Matrix ElHossz;                            /* Élhossz mátrix */
} Graf;

void MtxKiir(char *Szoveg, Matrix a, int n)
{ int i,j;
  printf("%s\n",Szoveg);
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
      if (a[i][j]==Vegtelen)
        printf("%6c",'-');
      else
        printf("%6d",a[i][j]);
    printf("\n");
  }
}

void Floyd_Warshall(Graf *G, Matrix T, Matrix C)
{ int x,y,w;
  /* Kezdőállapot */
  for (x=0; x<(*G).PontDb; x++)
    for (y=0; y<(*G).PontDb; y++) {
      if (x==y)
        T[x][y]=0;
      else
        if ((*G).ElHossz[x][y]==NincsEl)
          T[x][y]=Vegtelen;
```

```
        else
          T[x][y]=(*G).ElHossz[x][y];
      C[x][y]=y;
    }
  if (KellKiiras) {
    printf("A kezdőállapot\n");
    MtxKiir("T:",T,(*G).PontDb);
    MtxKiir("C:",C,(*G).PontDb);
    getch();
  }
  /* Javító lépések */
  for (w=0; w<(*G).PontDb; w++) {
    for (x=0; x<(*G).PontDb; x++)
      for (y=0; y<(*G).PontDb; y++)
        if (T[x][w]+T[w][y]<T[x][y]) {
          T[x][y]=T[x][w]+T[w][y];
          C[x][y]=C[x][w];
        }
    if (KellKiiras) {
      printf("%d. javító lépés után:\n",w+1);
      MtxKiir("T:",T,(*G).PontDb);
      MtxKiir("C:",C,(*G).PontDb);
      getch();
    }
  }
}

void GrafKiir(Graf *G)
{ int i,j,k;
  printf("Pontok száma:%d\n",(*G).PontDb);
  printf("Index Az-ó     X     Y\n");
  for (i=0; i<(*G).PontDb; i++)
    printf("%5d %c    %5d %5d\n",i+1,(*G).Pontok[i].Azon,
           (*G).Pontok[i].x,(*G).Pontok[i].y);
  getch();
  printf("A gráf élei:\n");
  for (i=0, k=1; i<(*G).PontDb; i++)
    for (j=0; j<(*G).PontDb; j++)
      if (i!=j && (*G).ElHossz[i][j]!=NincsEl) {
        printf("%c - %c : %d\n",(*G).Pontok[i].Azon,
               (*G).Pontok[j].Azon,(*G).ElHossz[i][j]);
        if (k++%20==0) getch();
      }
  getch();
}

void UtKiir(Graf *G, Matrix T, Matrix C, int Kp, int Vp)
{ int x;
  printf("%c - %c viszonylat\n",
         (*G).Pontok[Kp].Azon,(*G).Pontok[Vp].Azon);
  if (T[Kp][Vp]==Vegtelen)
    printf("Nem lehet eljutni!\n");
```

```
  else {
    printf("A távolság:%d\n",T[Kp][Vp]);
    printf("Az út:");
    x=Kp;
    printf("%c ",(*G).Pontok[x].Azon);
    do {
      x=C[x][Vp];
      printf("%c ",(*G).Pontok[x].Azon);
    } while (x!=Vp);
    printf("\n");
    getch();
  }
}

void main()
{ /* A példagráf mátrixos tárolásban */
  Graf G =
{5,{{'1',0,20},{'2',15,20},{'3',8,12},{'4',0,0},{'5',15,0}},
   {{0,15,10,NincsEl,NincsEl},{NincsEl,0,NincsEl,NincsEl,20},
   {10,10,0,13,NincsEl},{20,NincsEl,NincsEl,0,15},
   {NincsEl,20,13,NincsEl,0}}};
  Matrix T,C;
  clrscr();
  if (KellKiiras)
    GrafKiir(&G);

  Floyd_Warshall(&G,T,C);

  /* A 2. pontból a 4. pontba */
  UtKiir(&G,T,C,1,3);  /* 0-tól indexelünk */
}
```

Faépítés

```
/* DIJKSTRA.C : Útkeresés Dijkstra módszerrel */

#include <stdio.h>
#include <conio.h>

#define MaxPontDb 8                        /* Pontok max. száma */
#define MaxElDb MaxPontDb*(MaxPontDb-1)    /* Élek max. száma */
#define MaxElHossz 99                      /* A max. élhossz */
#define NincsEl -1                         /* A nemlétező él */
#define MaxUtHossz MaxElDb*MaxElHossz      /* A max. úthossz */
#define Vegtelen MaxUtHossz+1              /* A 'végtelen'-hez */
#define KellKiiras 1                       /* Legyenek kiírások */
#define NincsCimke -1                      /* A nemlétező címke */

typedef struct {                           /* A pontrekord */
  char Azon;                               /* Pontazonosító */
  int x,y;                                 /* Koordináták */
} Pont;
```

```
/* Az éleket egydimenziós tömbben tároljuk */

typedef struct {                          /* Az élrekord */
  int Vp;                                 /* Végpontindex */
  int ElHossz;                            /* Élhossz */
} El;

typedef struct {                          /* A gráf */
  int PontDb;                             /* Pontok száma */
  Pont Pontok[MaxPontDb];                 /* Pontok */
  int ElMut[MaxPontDb+1];                 /* Élmutatók */
  int ElDb;                               /* Élek száma */
  El Elek[MaxElDb];                       /* Élek */
} Graf;

typedef int Tomb[MaxPontDb];     /* Az egydimenziós tömbökhöz */

void GrafKiir(Graf *G)
{ int i,j,k;
  printf("Pontok száma:%d\n",(*G).PontDb);
  printf("Index Az-ó     X     Y\n");
  for (i=0; i<(*G).PontDb; i++)
    printf("%5d %c    %5d %5d\n",i+1,(*G).Pontok[i].Azon,
           (*G).Pontok[i].x,(*G).Pontok[i].y);
  getch();
  printf("A gráf élei:\n");
  for (i=0, k=1; i<(*G).PontDb; i++)
    for (j=(*G).ElMut[i]; j<(*G).ElMut[i+1]; j++, k++) {
      printf("%c - %c : %d\n",(*G).Pontok[i].Azon,
             (*G).Pontok[(*G).Elek[j].Vp].Azon,
             (*G).Elek[j].ElHossz);
      if (k%20==0) getch();
    }
  getch();
}

void AdatKiir(Graf *G, Tomb A, Tomb T, Tomb C, int ADb)
{ int i;
  printf("A:");
  for (i=0; i<ADb; i++)
    printf("%6c",(*G).Pontok[A[i]].Azon);
  printf("\nT:");
  for (i=0; i<(*G).PontDb; i++)
    if (T[i]==Vegtelen)
      printf("%6c",'-');
    else
      printf("%6d",T[i]);
  printf("\nC:");
  for (i=0; i<(*G).PontDb; i++)
    if (C[i]==NincsCimke)
      printf("%6c",' ');
    else
```

```
      printf("%6d",C[i]);
  printf("\n");
  getch();
}

void Dijkstra(Graf *G, Tomb T, Tomb C, int Kp)
{ Tomb A; int ADb;
  Tomb Akt;          /* A pontok aktívsága (1:aktív, 0:nem) */
  int x,y,i,j,k;
  /* Kezdőállapot */
  for (i=0; i<(*G).PontDb; i++) {
    T[i]=Vegtelen;
    Akt[i]=0;
    C[i]=NincsCimke;   /* Csak a kiíráshoz kell */
  }
  T[Kp]=0;
  C[Kp]=Kp;
  ADb=1;
  A[ADb-1]=Kp;
  Akt[Kp]=1;
  if (KellKiiras) {
    printf("A kezdőállapot\n");
    AdatKiir(G,A,T,C,ADb);
  }
  /* Javító lépések */
  if (KellKiiras) j=0;       /* A javító lépések száma */
  while (ADb>0) {
    /* Az A minimális távolságú elemét X-be */
    k=0;
    for (i=1; i<ADb; i++)
      if (T[A[i]]<T[A[k]]) k=i;
    x=A[k];
    /* X törlése A-ból (az utolsó elemmel felülírjuk) */
    A[k]=A[ADb-1];
    ADb=ADb-1;
    Akt[x]=0;
    /* Rövidítés X-n keresztül */
    for (i=(*G).ElMut[x]; i<(*G).ElMut[x+1]; i++) {
      y=(*G).Elek[i].Vp;
      if (T[x]+(*G).Elek[i].ElHossz<T[y]) {
        T[y]=T[x]+(*G).Elek[i].ElHossz;
        C[y]=x;
        if (Akt[y]==0) {
          /* Y még nem aktív, hozzávesszük A-hoz */
          ADb=ADb+1;
          A[ADb-1]=y;
          Akt[y]=1;
        }
      }
    }
    if (KellKiiras) {
      printf("%d. javító lépés (x=%c) után\n",++j,
```

```
              (*G).Pontok[x].Azon);
      AdatKiir(G,A,T,C,ADb);
    }
  }
}

void UtKiir(Graf *G, Tomb T, Tomb C, int Kp, int Vp)
{ Tomb Ut;       /* A Kp-Vp út pontindexei */
  int UtDb;      /* Az út pontjainak száma */
  int x,cs;
  printf("%c - %c viszonylat\n",
         (*G).Pontok[Kp].Azon,(*G).Pontok[Vp].Azon);
  if (T[Vp]==Vegtelen)
    printf("Nem lehet eljutni!\n");
  else {
    /* Az út összerakása (visszafelé haladva) */
    UtDb=1;
    Ut[UtDb-1]=Vp;
    x=Vp;
    do {
      x=C[x];
      UtDb=UtDb+1;
      Ut[UtDb-1]=x;
    } while (x!=Kp);
    /* Az út megfordítása */
    for (x=0; x<UtDb/2; x++) {
      cs=Ut[x]; Ut[x]=Ut[UtDb-x-1]; Ut[UtDb-x-1]=cs;
    }
    /* Az út kiírása */
    printf("A távolság:%d\n",T[Vp]);
    printf("Az út:");
    for (x=0; x<UtDb; x++)
      printf("%c ",(*G).Pontok[Ut[x]].Azon);
    printf("\n");
    getch();
  }
}

void main()
{ /* A példagráf az éleket egydimenziós tömbben tárolva */
  Graf G =
{8,{{'1',0,20},{'2',20,40},{'3',40,20},{'4',20,0},{'5',10,20},
    {'6',20,30},{'7',30,20},{'8',20,10}},
   {0,3,6,9,12,16,19,23,26},
 26,{{1,40},{3,40},{4,10},{0,40},{2,40},{5,10},{1,40},
     {3,40},{6,10},{0,40},{2,40},{7,10},{0,10},{5,15},
     {6,20},{7,15},{1,10},{4,15},{6,15},{2,10},{4,20},
     {5,15},{7,15},{3,10},{4,15},{6,15}}};
  Tomb T,C;
  clrscr();
  if (KellKiiras)
    GrafKiir(&G);
```

```
  /* A 1. pontból kiinduló utakat */
  Dijkstra(&G,T,C,0);     /* 0-tól indexelünk */

  /* A 1. pontból a 4. pontba */
  UtKiir(&G,T,C,0,3);
}
```

17.2. Pascal programok

17.2.1. Legnagyobb közös osztó

```
{ Két természetes szám legnagyobb közös osztójának meghatározása
}
program LNKO;
uses crt;

{ Nem rekurzív függvény osztással }
function lnko_o(a,b:integer):integer;
var r:integer;
begin
  while b<>0 do begin
    r:=a mod b; a:=b; b:=r;
  end;
  lnko_o:=a;
end;

{ Nem rekurzív függvény kivonással }
function lnko_k(a,b:integer):integer;
begin
  while a<>b do
    if a>b then a:=a-b else b:=b-a;
  lnko_k:=a;
end;

{ Rekurzív függvény osztással }
function lnko_ro(a,b:integer):integer;
var er:integer;
begin
  if b=0 then
    er:=a
  else
    er:=lnko_ro(b,a mod b);
  lnko_ro:=er;
end;

{ Rekurzív függvény kivonással }
function lnko_rk(a,b:integer):integer;
var er:integer;
begin
  if a=b then
    er:=a
  else
```

```
    if a>b then
      er:=lnko_rk(a-b,b)
    else
      er:=lnko_rk(a,b-a);
  lnko_rk:=er;
end;

var a,b:integer;
begin
  clrscr;
  write('Egyik természetes szám:'); readln(a);
  write('Másik természetes szám:'); readln(b);
  writeln('A legnagyobb közös osztó:');
  writeln(lnko_o(a,b),' ',lnko_k(a,b),' ',
          lnko_ro(a,b),' ',lnko_rk(a,b));
end.
```

17.2.2. Kör területe, kerülete

```
{ Kör területe, kerülete }
program KOR;
uses crt;
var r,t,k:real;
begin
  clrscr;
  write('A kör sugara:'); readln(r);
  t:=r*r*pi;
  k:=2*r*pi;
  writeln('Terület:',t:0:2);
  writeln('Kerület:',k:0:2);
end.
```

17.2.3. Másodfokú egyenlet

```
{ Másodfokú egyenlet megoldása }
program MASODFOK;
uses crt;
var a,b,c,d,x1,x2,k,v:real;
begin
  clrscr;
  write('Együtthatók (a b c) : '); readln(a,b,c);
  if a=0 then
    { Nem másodfokú eset }
    if b=0 then
      { Nem elsőfokú eset }
      if c=0 then
        writeln('Minden valós szám megoldás!')
      else
        writeln('Nincs megoldás!')
    else
      writeln('Elsőfokú:',-c/b:0:2)
  else begin
```

```
    { Másodfokú eset }
    d:=sqr(b)-4*a*c;
    if d>0 then begin
      { Két valós gyök }
      x1:=(-b+sqrt(d))/(2*a);
      x2:=(-b-sqrt(d))/(2*a);
      writeln('Két valós gyök:',x1:0:2,' ',x2:0:2);
    end else
      if d=0 then
        writeln('Másodfokú:',-b/(2*a):0:2)
      else begin
        { Komplex gyökök }
        v:=-b/(2*a);
        k:=abs(sqrt(-d)/(2*a));
        writeln('Egyik komplex gyök:',v:0:2,'+',k:0:2,'i');
        writeln('Másik komplex gyök:',v:0:2,'-',k:0:2,'i');
      end;
  end;
end.
```

17.2.4. Legkisebb osztó

```
{ Legkisebb osztó }
program LEGKOSZT;
uses crt;
var n,o:integer;
begin
  clrscr;
  write('A vizsgált szám (>1):'); readln(n);
  o:=2;
  while n mod o<>0 do inc(o);
  writeln('A legkisebb, egynél nagyobb osztó:',o);
end.
```

17.2.5. Faktoriális

```
{ Faktoriális }
program FAKTOR;
uses crt;
var n,i:byte; fakt:real;
begin
  clrscr;
  write('N értéke:'); readln(n);
  fakt:=1;
  for i:=2 to n do fakt:=fakt*i;
  writeln(n,'!=',fakt:0:0);
end.
```

17.2.6. Karakterek vizsgálata

```
{ Karakterek vizsgálata }
program KARVIZSG;
uses crt;
var ch:char;
begin
  clrscr;
  writeln('Karakterek vizsgálata (Kilépés:Esc)');
  repeat
    { Bekérés }
    ch:=readkey;
    if ch<>#27 then begin
      { Kiírás }
      write(ch,' ',ord(ch),' ');
      { Kiértékelés }
      case ch of
        'A'..'Z','a'..'z':writeln('Angol betű');
        '0'..'9':writeln('Számjegy');
        else writeln('Egyéb');
      end;
    end;
  until ch=#27;
end.
```

17.2.7. Prímfelbontás

```
{ Prímfelbontás }
program PRIMFELB;
uses crt;
var n,o:integer;
begin
  clrscr;
  write('A felbontandó szám (>1):'); readln(n);
  o:=2;
  while n>1 do begin
    if n mod o=0 then begin
      n:=n div o;
      if n=1 then writeln(o)
      else write(o,'*');
    end else
      inc(o);
  end;
end.
```

17.2.8. Monoton növő sorozat

```
{ Monoton növő sorozat }
program MONNOVO;
uses crt;
var n,i:integer; a,Elozo:real; Novo:boolean;
begin
  clrscr;
```

```
  write('Adatok száma (>1):'); readln(n);
  { Első adat }
  write('1. adat:'); readln(a);
  { Kezdőértékek }
  Novo:=true; Elozo:=a;
  { Többi adat }
  for i:=2 to n do begin
    write(i,'. adat:'); readln(a);
    if a<Elozo then Novo:=false;
    Elozo:=a;
  end;
  if Novo then writeln('Monoton növők!')
  else writeln('Nem monoton növők!');
end.
```

17.2.9. Pozitív adatok maximuma, átlaga

```
{ Pozitív adatok maximuma, átlaga }
program PMAXATL;
uses crt;
var Akt,Max,Atl,Ossz:real; n,Db,i:integer;
begin
  clrscr;
  write('Adatok száma:'); readln(n);
  { Kezdőértékek }
  Db:=0; Ossz:=0;
  { Adatbekérés, feldolgozás }
  for i:=1 to n do begin
    write(i,'. adat:'); readln(Akt);
    if Akt>=0 then begin
      inc(Db);
      Ossz:=Ossz+Akt;
      if Db=1 then Max:=Akt
      else if Akt>Max then Max:=Akt;
    end;
  end;
  { Eredménykiírás }
  if Db=0 then
    writeln('Nem volt pozitív adat!')
  else begin
    Atl:=Ossz/Db;
    writeln('A pozitív adatok maximuma:',Max:0:2);
    writeln('A pozitív adatok átlaga:',Atl:0:2);
  end;
end.
```

17.2.10. e^x hatványsora

```
{ Az exponenciális függvény közelítése }
program EXPXKOZ;
uses crt;
var x,Epsz,Akt,Ossz:real; n:integer;
```

```
begin
  clrscr;
  { Adatbekérés }
  write('X értéke:'); readln(x);
  write('Pontosság:'); readln(Epsz);
  { Kezdőértékek }
  n:=1; Akt:=x; Ossz:=1+Akt;
  { Közelítés }
  while abs(Akt)>=Epsz do begin
    inc(n);
    Akt:=Akt*x/n;
    Ossz:=Ossz+Akt;
  end;
  { Eredménykiírás }
  writeln('N értéke:',n);
  writeln('A közelítő érték:',Ossz:0:10);
  writeln('A "pontos" érték:',exp(x):0:10);
end.
```

17.2.11. Gyökkeresés intervallumfelezéssel

```
{ Gyökkeresés intervallumfelezéssel }
program GYOKKER;
uses crt;

function f(x:real):real;
begin
  f:=sin(x);
end;

var a,b,Epsz,Gyok,xk,xv,xf,yk,yv,yf:real;
begin
  clrscr;
  { Adatbekérés }
  write('Az intervallum kezdőpontja:'); readln(a);
  write('Az intervallum végpontja  :'); readln(b);
  write('Pontosság:'); readln(Epsz);
  { Kezdőértékek }
  xk:=a; xv:=b; yk:=f(a); yv:=f(b);
  { Közelítés }
  while xv-xk>Epsz do begin
    { Felezőpont }
    xf:=(xk+xv)/2;
    yf:=f(xf);
    { Csökkentés }
    if yk*yf<=0 then begin
      xv:=xf; yv:=yf;
    end;
    if yv*yf<=0 then begin
      xk:=xf; yk:=yf;
    end;
  end;
```

```
  { Eredmény }
  Gyok:=(xk+xv)/2;
  { Eredménykiírás }
  writeln('A gyök közelítő értéke:',Gyok:0:10);
end.
```

17.2.12. Integrálérték meghatározása közelítéssel

```
{ Határozott integrál közelítése trapéz-módszerrel }
program TRAPEZ;
uses crt;

function f(x:real):real;
begin
  f:=3*x*x;
end;

var a,b,Epsz,t,e,h,y0,yn:real; n,i:integer;
begin
  clrscr;
  { Adatbekérés }
  write('Az intervallum kezdőpontja:'); readln(a);
  write('Az intervallum végpontja  :'); readln(b);
  write('Pontosság:'); readln(Epsz);
  { Kezdőértékek }
  y0:=f(a); yn:=f(b);
  t:=(b-a)*(y0+yn)/2;
  { Következő beosztás }
  n:=2;
  { Közelítés }
  repeat
    { Előző területösszeg }
    e:=t;
    { Részintervallumok hossza }
    h:=(b-a)/n;
    { Trapézok területösszege }
    t:=(y0+yn)/2;
    for i:=1 to n-1 do
      t:=t+f(a+i*h);
    t:=t*h;
    { Következő beosztás }
    n:=2*n;
  until abs(t-e)<Epsz;
  { Eredménykiírás }
  writeln('Az integrál közelítő értéke:',t:0:10);
end.
```

17.2.13. Átlagnál nagyobb elemek

```
{ Átlagnál nagyobb elemek }
program ATLNAGY;
uses crt;
```

```
const NMax=100;   { Az adatok maximális száma }
var a:array[1..NMax] of real;
    n,i:integer; Ossz,Atl:real;
begin
  clrscr;
  { Adatbekérés }
  write('Adatok száma (1-',NMax,'):'); readln(n);
  for i:=1 to n do begin
    write(i,'. adat:'); readln(a[i]);
  end;
  { Átlagszámolás }
  Ossz:=0;
  for i:=1 to n do
    Ossz:=Ossz+a[i];
  Atl:=Ossz/n;
  { Eredménykiírás }
  writeln('Az átlagnál nagyobb elemek');
  for i:=1 to n do
    if a[i]>Atl then
      writeln(a[i]:0:2);
end.
```

17.2.14. Kockadobások gyakorisága

```
{ Kockadobások gyakorisága }
program KOCKA;
uses crt;
const n=6;   {A kocka oldalainak száma}
var db:array[1..n] of integer;
    a,i:integer;
begin
  clrscr;
  writeln('Kockadobások (1-',n,') gyakorisága (Kilépés:0)');
  { Kezdőértékek }
  for i:=1 to n do db[i]:=0;
  i:=0;
  repeat
    { Bekérés }
    inc(i); write(i,'. dobás:'); readln(a);
    if a<>0 then inc(db[a])
  until a=0;
  { Eredménykiírás }
  writeln('Gyakoriságok');
  for i:=1 to n do
    writeln(i,' ',db[i]);
end.
```

17.2.15. Érték törlése adatsorból

```
{ Érték törlése adatsorból }
program ERTTORL;
uses crt;
```

```
const
  NMax=10;                          { Az adatok maximális száma }
type
  Elem=integer;                     { Az adatsor elemei }
  Adatsor=array[1..NMax] of Elem; { Az adatsor }

procedure Torol(var a:Adatsor; var n:integer; x:Elem);
var i,j:integer;
begin
  { Kezdőérték }
  j:=0;
  { Előremásolás }
  for i:=1 to n do
    if a[i]<>x then begin
      inc(j);
      a[j]:=a[i];
    end;
  { Darabszám }
  n:=j;
end;

var a:Adatsor;
    n,i:integer; x:Elem;
begin
  clrscr;
  write('Adatok száma (1-',NMax,'):'); readln(n);
  for i:=1 to n do begin
    write(i,'. adat:'); readln(a[i]);
  end;
  write('A törlendő érték:'); readln(x);
  Torol(a,n,x);
  writeln('Törlés után');
  for i:=1 to n do write(a[i],' '); writeln;
end.
```

17.2.16. Eratosztenesz szitája

```
{ Eratosztenesz szitája tömbbel }
program SZITA_T;
uses crt;
const
  K=1000;                { Eddig határozzuk meg a prímszámokat }
type
  Tomb=array[2..K] of boolean;  { A számokat reprezentáló tömb }

var a:Tomb; p,i:integer;
begin
  { Számok felírása }
  for i:=2 to K do a[i]:=true;
  { Szitálás }
  for p:=2 to K do
    { A p szám prímszám? }
```

```
    if a[p] then begin
      { Igen, töröljük a többszöröseit }
      i:=2*p;
      while i<=K do begin
        a[i]:=false;
        i:=i+p;
      end;
    end;
  { Kiírás }
  clrscr;
  for i:=2 to K do
    if a[i] then write(i:8);
end.
```

17.2.17. Mátrixösszegek

```
{ Mátrixösszegek }
program MATOSSZ;
uses crt;

const
  NMax=10;    { Sorok maximális száma }
  MMax=10;    { Oszlopok maximális száma }

type
  Matrix=array[1..NMax,1..MMax] of real;    { A mátrix }
  SorOsszeg=array[1..NMax] of real;         { A sorösszegek }
  OszlOsszeg=array[1..MMax] of real;        { Az oszlopösszegek }

procedure Osszegek(const a:Matrix; n,m:integer;
 var Sor:SorOsszeg; var Oszl:OszlOsszeg; var Ossz:real);
var i,j:integer;
begin
  { Kezdőértékek }
  for j:=1 to m do Oszl[j]:=0;
  Ossz:=0;
  { Összegzés }
  for i:=1 to n do begin
    Sor[i]:=0;
    for j:=1 to m do begin
      Sor[i]:=Sor[i]+a[i,j];
      Oszl[j]:=Oszl[j]+a[i,j];
    end;
    Ossz:=Ossz+Sor[i];
  end;
end;

var a:Matrix; s:SorOsszeg; o:OszlOsszeg; ossz:real;
    n,m,i,j:integer;
begin
  clrscr;
  { Adatbekérés }
```

```
  write('Sorok száma (1-',NMax,'):'); readln(n);
  write('Oszlopok száma (1-',MMax,'):'); readln(m);
  for i:=1 to n do
    for j:=1 to m do begin
      write(i,'. sor ',j,'. elem:'); readln(a[i,j]);
    end;
  { A mátrix kiírása mátrix alakban }
  writeln('A megadott mátrix');
  for i:=1 to n do begin
    for j:=1 to m do write(a[i,j]:8:1);
    writeln;
  end;
  { Összegzés }
  Osszegek(a,n,m,s,o,ossz);
  { Eredménykiírás }
  writeln('A sorösszegek:');
  for i:=1 to n do write(s[i]:8:1); writeln;
  writeln('Az oszlopösszegek:');
  for j:=1 to m do write(o[j]:8:1); writeln;
  writeln('A teljes összeg:'); writeln(ossz:8:1);
end.
```

17.2.18. Oszlopok törlése mátrixból

```
{ Oszlopok törlése mátrixból }
program OSZLTORL;
uses crt;

const
  NMax=10;    { Sorok maximális száma }
  MMax=10;    { Oszlopok maximális száma }

type
  Matrix=array[1..NMax,1..MMax] of integer;      { A mátrix }

procedure Torol(var a:Matrix; n:integer; var m:integer);
var i,j,k:integer; Egyf:boolean;
begin
  j:=1;
  while j<=m do begin
    { A j. oszlop vizsgálata }
    Egyf:=true;
    i:=2;
    while (i<=n) and Egyf do begin
      Egyf:=a[i,j]=a[1,j];
      inc(i);
    end;
    if Egyf then begin
      { A j. oszlop törlése }
      for k:=j+1 to m do
        for i:=1 to n do
          a[i,k-1]:=a[i,k];
```

```
      dec(m);
    end else
      inc(j);
  end;
end;

{ A mátrix kiírása mátrix alakban }
procedure Kiir(const a:Matrix; n,m:integer);
var i,j:integer;
begin
  for i:=1 to n do begin
    for j:=1 to m do write(a[i][j]:6);
    writeln;
  end;
end;

var a:Matrix; n,m,i,j:integer;
begin
  clrscr;
  { Adatbekérés }
  write('Sorok száma (1-',NMax,'):'); readln(n);
  write('Oszlopok száma (1-',MMax,'):'); readln(m);
  for i:=1 to n do
    for j:=1 to m do begin
      write(i,'. sor ',j,'. elem:'); readln(a[i,j]);
    end;
  writeln('A megadott mátrix'); Kiir(a,n,m);
  { Törlés }
  Torol(a,n,m);
  writeln('Az eredmény mátrix'); Kiir(a,n,m);
end.
```

17.2.19. Sztring megfordítás

```
{ Sztring megfordítása }
program STRFORD;
uses crt;

procedure Fordit1(s:string; var er:string);
var i:byte;
begin
  er:='';
  for i:=length(s) downto 1 do er:=er+s[i];
end;

procedure Fordit2(s:string; var er:string);
var i:byte;
begin
  er:='';
  for i:=1 to length(s) do er:=s[i]+er;
end;
```

```
procedure Fordit3(var s:string);
var h,i:byte; cs:char;
begin
  h:=length(s);
  for i:=1 to h div 2 do begin
    cs:=s[i]; s[i]:=s[h+1-i]; s[h+1-i]:=cs;
  end;
end;

var s,er1,er2:string;
begin
  clrscr;
  write('A megfordítandó sztring:'); readln(s);
  Fordit1(s,er1); Fordit2(s,er2); Fordit3(s);
  writeln('Megfordítva:');
  writeln(er1); writeln(er2); writeln(s);
end.
```

17.2.20. Egyszerű kifejezés kiértékelése

```
{ Sztringben lévő egyszerű kifejezés kiértékelése }
program STRKIF;
uses crt;

procedure Ertekel(s:string; var jo:boolean; var er:integer);
var h,i:byte;
begin
  { Hosszellenőrzés }
  h:=length(s);
  jo:=odd(h);
  if jo then begin
    { Karakterek ellenőrzése }
    i:=1;
    while (i<=h) and jo do
      if odd(i) and ((s[i]<'0') or (s[i]>'9')) or
         not odd(i) and (s[i]<>'+') and (s[i]<>'-') then
        jo:=false
      else inc(i);
    if jo then begin
      { A kifejezés értékének kiszámítása }
      er:=ord(s[1])-ord('0');
      i:=3;
      while i<=h do begin
        if s[i-1]='+' then
          er:=er+ord(s[i])-ord('0')
        else
          er:=er-(ord(s[i])-ord('0'));
        inc(i,2);
      end;
    end;
  end;
end;
```

```
var s:string; jo:boolean; er:integer;
begin
  clrscr;
  write('A kiértékelendő kifejezés:'); readln(s);
  Ertekel(s,jo,er);
  if jo then
    writeln('Helyes kifejezés, értéke:',er)
  else
    writeln('Hibás kifejezés!');
end.
```

17.2.21. Lottószámok generálása

```
{ Lottószámok generálása }
program LOTTO;
uses crt;
const
  Max=90;               { A maximális lottószám értéke }
  Db=5;                 { A generálandó lottószámok darabszáma }
type
  Szam=1..Max;                  { A lottószám típus }
  Tomb=array[1..Db] of Szam;    { A lottószámok tömbje }
  Halmaz=set of Szam;           { A lottószámok halmaza }

{ Lottószámok generálása }
procedure General(var a:Tomb);
var h:halmaz; x:Szam; i,j:integer;
begin
  { Generálás }
  h:=[];
  for i:=1 to Db do begin
    repeat
      x:=random(Max)+1;
    until not (x in h);
    h:=h+[x];
  end;
  { Az eredménytömb feltöltése }
  j:=0;
  for i:=1 to Max do
    if i in h then begin
      inc(j); a[j]:=i;
    end;
end;

var a:Tomb; i:integer;
begin
  clrscr; randomize;
  { Generálás }
  General(a);
  { Kiírás }
  for i:=1 to Db do write(a[i],' ');
end.
```

17.2.22. Eratosztenesz szitája

```
{ Eratosztenesz szitája halmazzal }
program SZITA_H;
uses crt;
const
  K=255;               { Eddig határozzuk meg a prímszámokat }
type
  Halmaz=set of 2..K;  { A felírt számokat reprezentáló halmaz }

var h:Halmaz; p,i:integer;
begin
  { Számok felírása }
  h:=[2..K];
  { Szitálás }
  for p:=2 to K do
    { A p szám prímszám? }
    if p in h then begin
      { Igen, töröljük a többszöröseit }
      i:=2*p;
      while i<=K do begin
        h:=h-[i];
        i:=i+p;
      end;
    end;
  { Kiírás }
  clrscr;
  for i:=2 to K do
    if i in h then write(i:8);
end.
```

17.2.23. Különböző karakterek száma

```
{ Egy sztring különböző karaktereinek száma }
program KULKAR;
uses crt;

function KulKarDb(s:string):byte;
var h:set of char; i,db:byte;
begin
  db:=0; h:=[];
  for i:=1 to length(s) do
    if not (s[i] in h) then begin
      h:=h+[s[i]]; inc(db);
    end;
  KulKarDb:=db;
end;

var s:string;
begin
  clrscr;
  write('A sztring:'); readln(s);
```

```
  writeln('A különböző karakterek száma:',KulKarDb(s));
end.
```

17.2.24. Adatsor megjelenítése

```
{ Adatsor megjelenítése }
program TOMBKIIR;
uses crt;
const                              { A megengedett billentyűk }
  Esc=#27;
  Home=#71; End_=#79;
  Down=#80; Up=#72;
  PgUp=#73; PgDn=#81;
const
  NMax=10;                         { Az adatok maximális száma }
type
  Elem=integer;                              { Az adatsor elemei }
  Adatsor=array[1..NMax] of Elem;            { Az adatsor }
  Jelek=set of char;

function BillBe(Normal,Dupla:Jelek; var Duplae:Boolean):char;
var Jel:char; JoJel:boolean;
begin
  repeat
    Jel:=readkey; Duplae:=Jel=#0;
    if Duplae then begin
      Jel:=readkey;
      JoJel:=Jel in Dupla;
    end else
      JoJel:=Jel in Normal;
  until JoJel;
  BillBe:=Jel;
end;

procedure Kiir(const a:Adatsor; n,db:integer);
var Kezd,Bef,i:integer;
    Jel:char; Normal,Dupla:Jelek; Duplae:boolean;
begin
  if n=0 then begin
    clrscr; writeln('Nincsenek elemek!'); readkey;
  end else begin
    Kezd:=1;
    repeat
      { A befejező index beállítása }
      Bef:=Kezd+db-1; if Bef>n then Bef:=n;
      { Adatkiírás }
      clrscr; writeln('Ssz. Elem');
      for i:=Kezd to Bef do writeln(i:3,'. ',a[i]);
      { Az elfogadható jelek beállítása }
      Normal:=[Esc]; Dupla:=[];
      if Kezd>1 then Dupla:=Dupla+[Home,Up,PgUp];
      if Bef<n then Dupla:=Dupla+[End_,Down,PgDn];
```

```
      { Billentyűzetről való bekérés }
      Jel:=BillBe(Normal,Dupla,Duplae);
      { Pozícionálás }
      case Jel of
        Home:Kezd:=1;
        End_:Kezd:=n-db+1;
        PgDn:if Bef<n-db+1 then inc(Kezd,db) else Kezd:=n-db+1;
        PgUp:if Kezd-db>=1 then dec(Kezd,db) else Kezd:=1;
        Down:inc(Kezd);
        Up  :dec(Kezd);
      end;
    until Jel=Esc;
  end;
end;

var a:Adatsor;
    n,db,i:integer;
begin
  clrscr;
  write('Adatok száma (0-',NMax,'):'); readln(n);
  for i:=1 to n do begin
    write(i,'. adat:'); readln(a[i]);
  end;
  write('Egy képernyőn lévő adatok száma (1-',NMax,'):');
  readln(db);
  Kiir(a,n,db);
end.
```

17.2.25. Üzletek tartozása

```
{ Üzletek tartozása }
program UZLETEK;
uses crt;
const
  NMax=10;                        { Termékek maximális száma }
  MMax=20;                        { Üzletek maximális száma }
  NevMaxH=30;                     { Terméknevek maximális hossza
}
type
  Termek=record                   { Termék }
    Nev:string[NevMaxH];          { Név }
    Ear:word;                     { Egységár }
  end;
  Uzlet=record                    { Üzlet }
    Db:array[1..NMax] of word;    { Szállított mennyiségek }
    Tart:longint;                 { Tartozás }
  end;
  TermekTomb=array[1..NMax] of Termek;    { Termékek }
  UzletTomb=array[1..MMax] of Uzlet;      { Üzletek }
var
  n,m,i,j:integer;
  t:TermekTomb; u:UzletTomb;
```

```
begin
  clrscr;
  { Adatbekérés }
  write('Termékek száma (1-',NMax,'):'); readln(n);
  write('Üzletek száma (1-',MMax,'):'); readln(m);
  for i:=1 to n do begin
    writeln(i,'. termék');
    write('Neve:'); readln(t[i].Nev);
    write('Egységára:'); readln(t[i].Ear);
  end;
  { Szállított mennyiségek }
  for i:=1 to m do begin
    writeln(i,'. üzletnek szállított mennyiségek');
    for j:=1 to n do begin
      write(t[j].Nev,':'); readln(u[i].Db[j]);
    end;
  end;
  { Tartozások kiszámítása }
  for i:=1 to m do begin
    u[i].Tart:=0;
    for j:=1 to n do
      { Típuskonverzióval, hogy ne legyen túlcsordulás }
      inc(u[i].Tart,Longint(u[i].Db[j])*t[j].Ear);
  end;
  { Eredménykiírás }
  writeln('Tartozások');
  for i:=1 to m do
    writeln(i,'. üzlet:',u[i].Tart);
end.
```

17.2.26. Minimumhelyek keresése

```
{ Minimumhelyek keresése }
program MINHELY;
uses crt;

const
  NMax=10;                          { Az adatok maximális száma }
type
  Elem=integer;                             { Az adatsor elemei }
  Index=1..NMax;                            { Az indexek típusa }
  Adatsor=array[Index] of Elem;             { Az adatsor }
  MinHelyek=array[Index] of Index;          { A minimumhelyek }

procedure MinKereses(const a:Adatsor; n:integer; var Min:Elem;
 var db:integer; var Hely:MinHelyek);
var i:integer;
begin
  { Minimum meghatározása }
  Min:=a[1];
  for i:=2 to n do
    if a[i]<Min then
```

```
      Min:=a[i];
  { Minimumhelyek }
  db:=0;
  for i:=1 to n do
    if a[i]=Min then begin
      inc(db);
      Hely[db]:=i;
    end;
end;

var a:Adatsor; Hely:MinHelyek;
    n,i,db:integer; Min:Elem;
begin
  clrscr;
  write('Adatok száma (1-',NMax,'):'); readln(n);
  for i:=1 to n do begin
    write(i,'. adat:'); readln(a[i]);
  end;
  MinKereses(a,n,Min,db,Hely);
  writeln('A legkisebb elem:',Min);
  writeln('Előfordulási hely(ek):');
  for i:=1 to db do
    writeln(Hely[i]:4);
end.
```

17.2.27. Átlag és szórás

```
{ Átlag és szórás }
program ATLSZOR;
uses crt;

const
  NMax=10;                          { Az adatok maximális száma }
type
  Elem=integer;                             { Az adatsor elemei }
  Adatsor=array[1..NMax] of Elem;           { Az adatsor }

procedure Szamol(const a:Adatsor; n:integer; var Atl,Sz:real);
var i:integer; Ossz:real;
begin
  { Átlag }
  Ossz:=0;
  for i:=1 to n do
    Ossz:=Ossz+a[i];
  Atl:=Ossz/n;
  { Szórás }
  Ossz:=0;
  for i:=1 to n do
    Ossz:=Ossz+sqr(a[i]-Atl);
  Sz:=sqrt(Ossz/n);
end;
```

```
var a:Adatsor;
    n,i:integer; Atl,Sz:real;
begin
  clrscr;
  write('Adatok száma (1-',NMax,'):'); readln(n);
  for i:=1 to n do begin
    write(i,'. adat:'); readln(a[i]);
  end;
  Szamol(a,n,Atl,Sz);
  writeln('Átlag :',Atl:0:2);
  writeln('Szórás:',Sz:0:2);
end.
```

17.2.28. Előfordulási statisztika

```
{ Előfordulási statisztika }
program ELOFSTAT;
uses crt;

const
  NMax=10;                          { Az adatok maximális száma }
type
  Elem=integer;                              { Az adatsor elemei }
  Adatsor=array[1..NMAX] of Elem;            { Az adatsor }
  Dbsor=array[1..NMax] of integer;           { A darabszámok }

procedure Statisztika(const a:Adatsor; n:integer; var t:Adatsor;
 var db:Dbsor; var k:integer);
var i,j:integer; van:boolean;
begin
  { Kezdőérték }
  k:=0;
  for i:=1 to n do begin
    { Felvétel }
    inc(k);
    t[k]:=a[i];
    db[k]:=0;
    { Keresés }
    j:=1;
    while a[i]<>t[j] do inc(j);
    { Darabszám növelés }
    inc(db[j]);
    { Ha volt már ilyen, töröljük a táblázat végéről }
    if j<k then dec(k);
  end;
end;

var a,t:Adatsor; db:Dbsor;
    n,i,k:integer;
begin
  clrscr;
  write('Adatok száma (1-',NMax,'):'); readln(n);
```

```
  for i:=1 to n do begin
    write(i,'. adat:'); readln(a[i]);
  end;
  Statisztika(a,n,t,db,k);
  writeln('A különböző adatok és darabszámuk:');
  for i:=1 to k do
    writeln(t[i]:8,db[i]:4,' db');
end.
```

17.2.29. Jelstatisztika

```
{ Jelstatisztika }
program JELSTAT;
uses crt;

type
  Elem=record               { A gyakorisági táblázat egy eleme }
    Jel:char;
    Darab:byte;
  end;
  Tabla=array[1..255] of Elem;        { A gyakorisági táblázat }

procedure Statisztika(s:string; var t:Tabla; var k:byte);
var i,kod:byte; db:array[byte] of byte;
begin
  { Kezdőértékek }
  for i:=0 to 255 do db[i]:=0;
  { Darabszámok meghatározása }
  for i:=1 to length(s) do begin
    kod:=ord(s[i]);
    inc(db[kod]);
  end;
  { Eredménytáblázat elkészítése }
  k:=0;
  for i:=0 to 255 do
    if db[i]>0 then begin
      inc(k);
      t[k].Jel:=chr(i);
      t[k].Darab:=db[i];
    end;
end;

var s:string; t:Tabla; i,k:byte;
begin
  clrscr;
  write('A sztring:'); readln(s);
  Statisztika(s,t,k);
  writeln('A különböző jelek és darabszámuk:');
  for i:=1 to k do
    writeln(t[i].Jel,t[i].Darab:4,' db');
end.
```

17.2.30. Rendezés és keresés

```
{ Rendezések, keresések }
program RENDKER;
uses crt;
const
  MaxElemSzam=10;       { A rendezendő adatok maximális száma }
  MaxElem=20;           { A generálandó adatok maximális értéke }
type
  Elem=integer;                    { A rendezendő elemek típusa }
  Index=1..MaxElemSzam;                     { Az indexek típusa }
  Tomb=array[Index] of Elem;       { A rendezendő elemek tömbje }
  IndexTabla=array[Index] of Index;     { Az indextábla típusa }

{ Buborékrendezés }
procedure BubRend(var a:Tomb; n:integer);
var cs:Elem; i,j:integer;
begin
  for i:=1 to n-1 do
    { i. elemet a helyére }
    for j:=n downto i+1 do
      if a[j]<a[j-1] then begin
        { A j. és j-1. elemek cseréje }
        cs:=a[j]; a[j]:=a[j-1]; a[j-1]:=cs;
      end;
end;

{ Rendezés kiválasztással }
procedure KivalRend(var a:Tomb; n:integer);
var cs:Elem; i,j,k:integer;
begin
  for i:=1 to n-1 do begin
    { i. elemet a helyére }
    k:=i;
    for j:=i+1 to n do
      if a[j]<a[k] then k:=j;
    if k>i then begin
      { Az i. és k. elemek cseréje }
      cs:=a[i]; a[i]:=a[k]; a[k]:=cs;
    end;
  end;
end;

{ Rendezés kiválasztással, indextáblával }
procedure KivalRendIt(var a:Tomb; var it:IndexTabla; n:integer);
var i,j,k,cs:integer;
begin
  { Az indextábla feltöltése }
  for i:=1 to n do it[i]:=i;
  { Rendezés }
  for i:=1 to n-1 do begin
    { i. elemet a helyére }
    k:=i;
```

```
    for j:=i+1 to n do
      if a[it[j]]<a[it[k]] then k:=j;
    if k>i then begin
      { Az i. és k. indexek cseréje }
      cs:=it[i]; it[i]:=it[k]; it[k]:=cs;
    end;
  end;
end;

{ Rendezés beszúrással }
procedure BeszurRend(var a:Tomb; n:integer);
var x:Elem; i,j:integer;
begin
  for i:=2 to n do begin
    { Az i. elem beszúrása az előtte lévő rendezett részbe }
    x:=a[i];
    { Helykészítés hátraléptetéssel }
    j:=i-1;
    while (j>=1) and (a[j]>x) do begin
      a[j+1]:=a[j]; dec(j);
    end;
    { i. elemet a helyére }
    a[j+1]:=x;
  end;
end;

{ Lineáris keresés rendezetlen adatok között }
function LinKer(const a:Tomb; n:integer; x:Elem; var
Hol:integer):boolean;
var i:integer; Van:boolean;
begin
  i:=1;
  while (i<=n) and (a[i]<>x) do inc(i);
  Van:=i<=n;
  if Van then Hol:=i;
  LinKer:=Van;
end;

{ Lineáris keresés rendezett adatok között }
function LinKerRend(const a:Tomb; n:integer; x:Elem; var
Hol:integer):boolean;
var i:integer; Van:boolean;
begin
  i:=1;
  while (i<=n) and (a[i]<x) do inc(i);
  Van:=(i<=n) and (a[i]=x);
  Hol:=i;
  LinKerRend:=Van;
end;
```

```
{ Bináris keresés }
function BinKer(const a:Tomb; n:integer; x:Elem; var
Hol:integer):boolean;
var i,j,k:integer; Van:boolean;
begin
  i:=1; j:=n; Van:=false;
  while (i<=j) and not Van do begin
    k:=(i+j) div 2;
    if a[k]=x then Van:=true
    else
      if x<a[k] then j:=k-1
      else i:=k+1;
  end;
  if Van then Hol:=k
  else Hol:=i;
  BinKer:=Van;
end;

{ Bináris keresés indextáblával }
function BinKerIt(const a:Tomb; const it:IndexTabla; n:integer;
 x:Elem; var Hol:integer):boolean;
var i,j,k:integer; Van:boolean;
begin
  i:=1; j:=n; Van:=false;
  while (i<=j) and not Van do begin
    k:=(i+j) div 2;
    if a[it[k]]=x then Van:=true
    else
      if x<a[it[k]] then j:=k-1
      else i:=k+1;
  end;
  if Van then Hol:=k
  else Hol:=i;
  BinKerIt:=Van;
end;

{ Elemek generálása }
procedure General(var a:Tomb; n:integer);
var i:integer;
begin
  for i:=1 to n do
    a[i]:=random(MaxElem+1);
end;

procedure Kiiras(Szoveg:string; const a:Tomb; n:integer);
var i:integer;
begin
  writeln(Szoveg);
  for i:=1 to n do write(a[i],' ');
  writeln;
end;
```

```
procedure KiirasIt(Szoveg:string; const a:Tomb;
 const it:IndexTabla; n:integer);
var i:integer;
begin
  writeln(Szoveg);
  for i:=1 to n do write(a[it[i]],' ');
  writeln;
end;

var a,b:Tomb; it:IndexTabla; x:Elem; Hol:integer;
begin
  randomize; clrscr;

  General(b,MaxElemSzam);
  Kiiras('Rendezések előtt',b,MaxElemSzam);

  a:=b;
  BubRend(a,MaxElemSzam);
  Kiiras('Buborék rendezés után',a,MaxElemSzam);

  a:=b;
  KivalRend(a,MaxElemSzam);
  Kiiras('Kiválasztásos rendezés után',a,MaxElemSzam);

  a:=b;
  BeszurRend(a,MaxElemSzam);
  Kiiras('Beszúrásos rendezés után',a,MaxElemSzam);

  KivalRendIt(b,it,MaxElemSzam);
  KiirasIt('Egy indextáblás rendezés után',b,it,MaxElemSzam);

  write('Mit keressünk:'); readln(x);
  if LinKer(b,MaxElemSzam,x,Hol) then
    writeln('A rendezetlen adatok közötti indexe:',Hol)
  else
    writeln('Nincs a rendezetlen adatok között!');
  if LinKerRend(a,MaxElemSzam,x,Hol) then
    writeln('A rendezett adatok közötti indexe:',Hol)
  else
    writeln('Nincs a rendezett adatok között, helye:',Hol);
  if BinKer(a,MaxElemSzam,x,Hol) then
    writeln('A rendezett adatok közötti indexe:',Hol)
  else
    writeln('Nincs a rendezett adatok között, helye:',Hol);
  if BinKerIt(b,it,MaxElemSzam,x,Hol) then
    writeln('Indextáblabeli indexe:',Hol)
  else
    writeln('Indextáblabeli helye:',Hol);
end.
```

17.2.31. Ellenőrzött input

```
{ Ellenőrzött input }
program ELLINP;
uses crt;
const
  Sotet=0;                  { A színkiemeléshez }
  Vilagos=7;
  Kilep=#27;                { Esc }         { Vezérlőbillentyűk }
  AdatVeg=#13;              { Enter }
  Torol=#8;                 { Backspace }

  MaxRszHossz=6;            { A rendszám maximális hossza }
  MaxEgSzHossz=11;          { Az egész szám maximális hossza }
  Elojel='-';               { Előjel }

  SzamJegyek=['0'..'9'];    { A számjegyek halmaza }

type
  Jelek=set of char;

function BillBe(Normal,Dupla:Jelek; var Duplae:Boolean):char;
var Jel:char; JoJel:boolean;
begin
  repeat
    Jel:=readkey; Duplae:=Jel=#0;
    if Duplae then begin
      Jel:=readkey;
      JoJel:=Jel in Dupla;
    end else
      JoJel:=Jel in Normal;
  until JoJel;
  BillBe:=Jel;
end;

function JelBe(JoJelek:Jelek):char;
var Duplae:boolean;
begin
  JelBe:=BillBe(JoJelek,[],Duplae);
end;

function JobbTolt(Mit:string; Hossz:byte):string;
begin
  while length(Mit)<Hossz do Mit:=Mit+' ';
  JobbTolt:=Mit;
end;

procedure NormalIr;
begin
  textcolor(Vilagos); textbackground(Sotet);
end;
```

```
procedure InverzIr;
begin
  textcolor(Sotet); textbackground(Vilagos);
end;

procedure Ir(Mit:string; Oszl,Sor:byte);
begin
  if Oszl=0 then Oszl:=wherex;
  if Sor=0 then Sor:=wherey;
  gotoxy(Oszl,Sor); write(Mit);
end;

procedure KiIras(Mit:string; Oszl,Sor,Hossz:byte);
begin
  Ir(Mit,Oszl,Sor); gotoxy(Oszl+Hossz,Sor);
end;

procedure RendSzamBe(var RendSzam:string; {i/o eredmény szöveg}
                         Oszl,Sor:byte;   {i   képernyőpozíció}
                     var VanAdat:boolean  {o   van-e új adat});
const
  Betuk=['A'..'Z','a'..'z']; AdatJel=Betuk+SzamJegyek;
var
  Jel:char; JoJel:Jelek; Hossz:byte;
begin
  {Előkészítés}
  {Adat}
  Hossz:=length(RendSzam);
  RendSzam:=JobbTolt(RendSzam,MaxRszHossz);
  {Képernyö}
  if Oszl=0 then Oszl:=wherex;
  if Sor=0 then Sor:=wherey;
  InverzIr;
  KiIras(RendSzam,Oszl,Sor,Hossz);
  {Beolvasás}
  repeat
    {JoJel beállítás}
    JoJel:=[KiLep];
    case Hossz of
      0..1: JoJel:=JoJel+Betuk;
         2: JoJel:=JoJel+AdatJel;
      3..5: JoJel:=JoJel+SzamJegyek;
    end;
    if Hossz>0 then JoJel:=JoJel+[Torol];
    {Utolsó számjegy}
    if (Hossz=5) and (RendSzam[3] in Betuk+['0']) and
       (copy(RendSzam,4,2)='00') then JoJel:=JoJel-['0'];
    if Hossz=MaxRszHossz then JoJel:=JoJel+[AdatVeg];
    {Jel beolvasás}
    Jel:=JelBe(JoJel);
    {Jel feldolgozás}
    if Jel in AdatJel then begin
```

```
      {Nagybetűssé alakítjuk}
      inc(Hossz); RendSzam[Hossz]:=upcase(Jel)
    end else
    if Jel=Torol then begin
      RendSzam[Hossz]:=' '; dec(Hossz);
    end;
    KiIras(RendSzam,Oszl,Sor,Hossz);
  until (Jel in [KiLep,AdatVeg]);
  {Befejezés}
  VanAdat:=Jel<>KiLep;
  {Inverz kiemelés levétele}
  NormalIr;
  KiIras(JobbTolt('',MaxRszHossz),Oszl,Sor,MaxRszHossz);
  {Hosszbeállítás}
  if VanAdat then RendSzam[0]:=chr(Hossz)
  else RendSzam:='';
  KiIras(RendSzam,Oszl,Sor,Hossz);
 end;

procedure EgSzamBe(var SzamSzov:string; {i/o eredmény szöveg}
                       Oszl,Sor:byte;   {i képernyőpozíció}
                       Tol,Ig:longint;  {i határok        }
                   var VanAdat:boolean; {o van-e új adat  }
                   var SzamErt:longint  {o az eredmény szám });
var
  Jel:char; JoJel:Jelek; Hossz,MaxH:byte; JoAdat:boolean;
  w:string; x:real; i:integer;
begin
  {Előkészítés}
  {Adat}
  str(Tol,w); MaxH:=length(w); str(Ig,w);
  if MaxH<length(w) then MaxH:=length(w);
  Hossz:=length(SzamSzov); SzamSzov:=JobbTolt(SzamSzov,MaxH);
  {Képernyő}
  if Oszl=0 then Oszl:=wherex;
  if Sor=0 then Sor:=wherey;
  InverzIr;
  KiIras(SzamSzov,Oszl,Sor,Hossz);
  {Beolvasás}
  repeat
    {Tartalmi helyesség: Tol-Ig }
    repeat
      {Formai helyesség: csak szám}
      {Jojel beállítás}
      JoJel:=[KiLep];
      if Hossz<MaxH then JoJel:=JoJel+SzamJegyek;
      if (Hossz=0) and (Tol<0) then JoJel:=JoJel+[Elojel];
      if Hossz>0 then JoJel:=JoJel+[Torol];
      if (Hossz>0) and (SzamSzov[Hossz] in SzamJegyek) then
        JoJel:=JoJel+[AdatVeg];
      {Jel beolvasás}
      Jel:=JelBe(JoJel);
```

```
      {Jel feldolgozás}
      if (Jel in SzamJegyek) or (Jel=Elojel) then begin
        Hossz:=Hossz+1; SzamSzov[Hossz]:=Jel
      end else
      if Jel=Torol then begin
        SzamSzov[Hossz]:=' '; Hossz:=Hossz-1;
      end;
      KiIras(SzamSzov,Oszl,Sor,Hossz);
    until Jel in [KiLep,AdatVeg];
    {Tartomány ellenőrzés}
    if Jel<>KiLep then begin
      val(copy(SzamSzov,1,Hossz),x,i) ;
      JoAdat:=(x>=Tol) and (x<=Ig);
    end;
  until (Jel=KiLep) or JoAdat;
  {Befejezés}
  VanAdat:=Jel<>KiLep;
  {Inverz kiemelés levétele}
  NormalIr;
  KiIras(JobbTolt('',MaxH),Oszl,Sor,MaxH);
  {Hosszbeállítás}
  if VanAdat then begin
    SzamSzov[0]:=chr(Hossz);
    SzamErt:=round(x);
  end
  else SzamSzov:='';
  KiIras(SzamSzov,Oszl,Sor,Hossz);
end;

var st:string; van:boolean; l:longint;
begin
  NormalIr; clrscr;

  st:='';
  write('Rendszám:');
  RendszamBe(st,0,0,van);
  writeln;
  if van then
    writeln('A megadott rendszám:',st)
  else
    writeln('Nem adott meg adatot!');
  writeln;

  st:='';
  write('Egész szám [-128,127]:');
  EgSzamBe(st,0,0,-128,127,van,l);
  writeln;
  if van then begin
    writeln('A megadott szám sztringként:',st);
    writeln('A megadott szám számként:',l);
```

```
  end else
    writeln('Nem adott meg adatot!');
end.
```

17.2.32. Faktoriális

```
{ Faktoriális kiszámítása rekurzív függvénnyel }
program FAKTOR_R;
uses crt;

function Fakt(n:byte):real;
var er:real;
begin
  if n=0 then er:=1
         else er:=Fakt(n-1)*n;
  Fakt:=er;
end;

var n:byte;
begin
  clrscr;
  write('N értéke:'); readln(n);
  writeln(n,'!=',Fakt(n):0:0);
end.
```

17.2.33. Gyorsrendezés

```
{ Gyorsrendezés rekurzívan }
program GYRENDR;
uses crt;
const
  MaxElemSzam=10;       { A rendezendő adatok maximális száma }
  MaxElem=20;           { A generálandó adatok maximális értéke }
type
  Elem=integer;                  { A rendezendő elemek típusa }
  Tomb=array[1..MaxElemSzam] of Elem;  { A rendezendő elemek }

{ A rekurzív eljárás amely rendezi az 'a' tömb 'k'-'v' indexű
elemeit }
procedure GyorsRend(var a:Tomb; k,v:integer);
var s,cs:Elem; i,j:integer;
begin
  if k<v then begin
    { Van legalább két elem }
    i:=k; j:=v; s:=a[(i+j) div 2];
    { Szétválogatás a strázsa (s) elemhez képest }
    while i<=j do begin
      while a[i]<s do inc(i);
      while a[j]>s do dec(j);
      if i<=j then begin
        { Az i. és j. elemek cseréje }
        cs:=a[i]; a[i]:=a[j]; a[j]:=cs;
```

```
        inc(i); dec(j);
      end;
    end;
    { Az első rész rendezése rekurzív hívással }
    GyorsRend(a,k,j);
    { A második rész rendezése rekurzív hívással }
    GyorsRend(a,i,v);
  end;
end;

{ Elemek generálása }
procedure General(var a:Tomb; n:integer);
var i:integer;
begin
  for i:=1 to n do
    a[i]:=random(MaxElem+1);
end;

procedure Kiiras(Szoveg:string; const a:Tomb; n:integer);
var i:integer;
begin
  writeln(Szoveg);
  for i:=1 to n do write(a[i],' ');
  writeln;
end;

var a:Tomb;
begin
  randomize; clrscr;

  General(a,MaxElemSzam);
  Kiiras('Rendezés előtt',a,MaxElemSzam);

  { Sorbarendezés }
  GyorsRend(a,1,MaxElemSzam);

  Kiiras('Rendezés után',a,MaxElemSzam);
end.
```

17.2.34. Kínai gyűrűk

```
{ Kínai gyűrűk }
program KINAI;
uses crt;
const
  Max=9;                          { A gyűrűk maximális száma }
Var
  Gyuruk:array[1..Max] of integer;   { A gyűrűk állapota }
  n:integer;                         { A gyűrűk száma }
const
  l:integer=0;                    { A mozgatási lépések száma }
```

```
{ A gyűrűk állapotának kiírása }
procedure Kiir;
var i:integer;
begin
  if l=0 then writeln('Kezdőállapot')
  else writeln(l,'. lépés');
  for i:=1 to n do
    write(Gyuruk[i]);
  writeln;
  inc(l); readkey;
end;

procedure Le(n:integer); forward;

{ Az n. és a megelőző gyűrűk levétele }
procedure BalraLe(n:integer);
var i:integer;
begin
  for i:=n downto 1 do Le(i);
end;

procedure Fel(n:integer); forward;

{ Az n. gyűrű levétele }
procedure Le(n:integer);
begin
  if Gyuruk[n]=1 then begin
    if n>1 then begin
      Fel(n-1);
      if n>2 then BalraLe(n-2);
    end;
    Gyuruk[n]:=0;
    Kiir;
  end;
end;

{ Az n. gyűrű feltétele }
procedure Fel(n:integer);
begin
  if Gyuruk[n]=0 then begin
    if n>1 then begin
      Fel(n-1);
      if n>2 then BalraLe(n-2);
    end;
    Gyuruk[n]:=1;
    Kiir;
  end;
end;

var i:integer;
begin
  clrscr;
```

```
  write('A gyűrűk száma (1-',Max,'):');
  readln(n);
  { Kezdőállapot }
  for i:=1 to n do Gyuruk[i]:=1;
  Kiir;
  { A fő eljárás meghívása }
  BalraLe(n);
end.
```

17.2.35. Huszár útja a sakktáblán

```
{ Huszár útja a sakktáblán }
program HUSZAR;
uses crt;

const
  MaxMeret=8;            { A tábla maximális mérete }
  KellKiiras=false;      { Legyen-e kiírás a lépések után }

type
  Mezo=record
    S,O:integer;
  end;

  Lepesek=array[1..8] of integer;

{ A lehetséges 8 lépésirány relatív elmozdulásai }
const
  LepS:Lepesek=(-2,-1,1,2,2,1,-1,-2);
  LepO:Lepesek=(1,2,2,1,-1,-2,-2,-1);

{ Globális változók }
var
  N:integer;                                  { A tábla mérete }
  Tabla:array[1..MaxMeret,1..MaxMeret] of integer;   { A tábla }
  VanMego:boolean;                          { Van-e már megoldás }

procedure Kiir(Lepes:integer);
var i,j:integer;
begin
  if Lepes>0 then
    writeln(Lepes,'. lépés után');
  for i:=1 to N do begin
    for j:=1 to N do
      write(Tabla[i,j]:4);
    writeln;
  end; writeln;
  readkey;
end;

{ A huszár következő lépése }
procedure Probal(Lepes:integer; Akt:Mezo);
```

```
var Irany:integer; Kov:Mezo;
begin
  { A választás előkészítése }
  Irany:=1;
  repeat
    { Válasszuk ki a következő választást }
    Kov.S:=Akt.S+LepS[Irany];
    Kov.O:=Akt.O+LepO[Irany];
    { Megfelelő? }
    if (Kov.S>=1) and (Kov.S<=N) and (Kov.O>=1) and (Kov.O<=N)
       and (Tabla[Kov.S,Kov.O]=0) then begin
      { Jegyezzük fel }
      Tabla[Kov.S,Kov.O]:=Lepes;
      if KellKiiras then
        Kiir(Lepes);
      { A megoldás nem teljes? }
      if Lepes<N*N then begin
        { Rekurzív hívás }
        Probal(Lepes+1,Kov);
        if not VanMego then
          { A feljegyzés törlése }
          Tabla[Kov.S,Kov.O]:=0;
      end else
        VanMego:=true;
    end;
    inc(Irany);
  until VanMego or (Irany>8);
end;

var Km:Mezo; i,j:integer;
begin
  clrscr;
  write('A sakktábla mérete (3-',MaxMeret,'):'); readln(N);
  write('A kezdőmező sora (1-',N,'):'); readln(Km.S);
  write('A kezdőmező oszlopa (1-',N,'):'); readln(Km.O);
  for i:=1 to N do
    for j:=1 to N do
      Tabla[i,j]:=0;
  Tabla[Km.S,Km.O]:=1;
  VanMego:=false;

  writeln('Dolgozom...');

  Probal(2,Km);

  if VanMego then begin
    writeln; writeln('A megtalált megoldás');
    Kiir(0);
  end else
    writeln('Nincs megoldás!');
end.
```

17.2.36. Nyolc királynő

```
{ A 8 királynő probléma }
program KIRALYNO;
uses crt;

const
  MaxMeret=8;                        { A tábla maximális mérete }

{ Globális változók }
var
  n:integer;                                  { A tábla mérete }
  Hol:array[1..MaxMeret+1] of integer;        { A királynők helye }
  FAtl:array[1..2*MaxMeret] of boolean;       { A főátlók }
  MAtl:array[1..2*MaxMeret] of boolean;       { A mellékátlók }
  Oszl:array[1..2*MaxMeret] of boolean;       { Az oszlopok }
  db:integer;                                 { Az összes megoldás }

procedure SzinValt(Szoveg,Hatter:integer);
begin
  textcolor(Szoveg);
  textbackground(Hatter);
end;

procedure MegoKiir(Kiiras:boolean);
var i,j,k:integer;
begin
  inc(db); writeln(db,'. megoldás:');
  if Kiiras then begin
    { Sima kiírás }
    for i:=1 to n do write(Hol[i]:4); writeln;
  end else begin
    { Táblás megjelenítés }
    write('╔'); for k:=1 to n*3 do write('═'); writeln('╗');
    for i:=1 to n do begin
      write('║');
      for j:=1 to n do begin
        if (i mod 2=0) and (j mod 2=0) or
           (i mod 2=1) and (j mod 2=1) then SzinValt(0,7)
        else SzinValt(7,0);
        write('   ');
        if Hol[i]=j then begin
          gotoxy(wherex-3,wherey); write('\'#127'/');
        end;
      end;
      SzinValt(7,0); writeln('║');
    end;
    write('╚'); for k:=1 to n*3 do write('═'); writeln('╝');
  end;
  readkey;
end;
```

```
{ Egy királynő elhelyezése az S. sorba }
procedure Probal(s:integer);
var o:integer;
begin
  { Az összes választáson }
  for o:=1 to n do begin
    { s. sorba az o. oszlopba téve az s. királynőt }
    { Megfelelő? }
    if not Oszl[o] and not FAtl[s-o+n] and
       not MAtl[s+o-1] then begin
      { Jegyezzük fel }
      Hol[s]:=o;
      Oszl[o]:=true;
      FAtl[s-o+n]:=true;
      MAtl[s+o-1]:=true;
      { A megoldás nem teljes? }
      if s<n then
        { Rekurzív hívás }
        Probal(s+1)
      else
        MegoKiir(false);
      { A feljegyzés törlése }
      Oszl[o]:=false;
      FAtl[s-o+n]:=false;
      MAtl[s+o-1]:=false;
    end;
  end;
end;

var i:integer;
begin
  clrscr;
  write('A sakktábla mérete (1..',MaxMeret,'):'); readln(n);
  for i:=1 to n do
    Oszl[i]:=false;
  for i:=1 to 2*n-1 do begin
    FAtl[i]:=false; MAtl[i]:=false;
  end;
  db:=0;
  Probal(1);
  if db=0 then writeln('Nincs megoldás!')
  else writeln(db,' db megoldás létezett!');
end.
```

17.2.37. Gyorsrendezés saját veremmel

```
{ Gyorsrendezés nem rekurzívan, saját veremkezeléssel }
program GYRENDNR;
uses crt;
const
  MaxElemSzam=10;           { A rendezendő adatok max. száma }
  MaxElem=20;               { A generálandó adatok max. értéke }
```

```
type
  Elem=integer;                    { A rendezendő elemek típusa }
  Tomb=array[1..MaxElemSzam] of Elem;       { Az elemek tömbje }
const
  MaxVeremMeret=4;    { A verem max. mérete (log2(MaxElemSzam) }
type
  VeremElem=record          { A veremben tárolt elemek típusa }
    k,v:integer;            { A rendezendő elemek indexhatárai }
  end;
  TVerem=array[1..MaxVeremMeret] of VeremElem;  {A verem típusa}

{ Adat betétele a verembe }
procedure Verembe(var Verem:TVerem; var VeremMut:integer;
k,v:integer);
begin
  inc(VeremMut);
  Verem[VeremMut].k:=k;
  Verem[VeremMut].v:=v;
end;

{ Adat kivétele a veremből }
procedure Verembol(var Verem:TVerem; var VeremMut,k,v:integer);
begin
  k:=Verem[VeremMut].k;
  v:=Verem[VeremMut].v;
  dec(VeremMut);
end;

{ A rendező eljárás }
procedure GyorsRend(var a:Tomb; n:integer);
var Verem:TVerem;
    s,cs:Elem;
    VeremMut,k,v,i,j:integer;
begin
  { A verem inicializálása }
  VeremMut:=0;
  { A teljes rendezendő részt a verembe }
  Verembe(Verem,VeremMut,1,n);
  while VeremMut>0 do begin
    { A rendezendő rész kivétele a veremből }
    Verembol(Verem,VeremMut,k,v);
    if k<v then begin
      { Van legalább két elem }
      i:=k; j:=v; s:=a[(i+j) div 2];
      { Szétválogatás a strázsa (s) elemhez képest }
      while i<=j do begin
        while a[i]<s do inc(i);
        while a[j]>s do dec(j);
        if i<=j then begin
          { Az i. és j. elemek cseréje }
          cs:=a[i]; a[i]:=a[j]; a[j]:=cs;
          inc(i); dec(j);
```

```
        end;
      end;
      { A két rész felvétele a verembe (felülre a kisebbet) }
      if j-k>v-i then begin
        Verembe(Verem,VeremMut,k,j);
        Verembe(Verem,VeremMut,i,v);
      end else begin
        Verembe(Verem,VeremMut,i,v);
        Verembe(Verem,VeremMut,k,j);
      end;
    end;
  end;
end;

{ Elemek generálása }
procedure General(var a:Tomb; n:integer);
var i:integer;
begin
  for i:=1 to n do
    a[i]:=random(MaxElem+1);
end;

procedure Kiiras(Szoveg:string; const a:Tomb; n:integer);
var i:integer;
begin
  writeln(Szoveg);
  for i:=1 to n do write(a[i],' ');
  writeln;
end;

var a:Tomb;
begin
  randomize; clrscr;

  General(a,MaxElemSzam);
  Kiiras('Rendezés előtt',a,MaxElemSzam);

  { Sorbarendezés }
  GyorsRend(a,MaxElemSzam);

  Kiiras('Rendezés után',a,MaxElemSzam);
end.
```

17.2.38. Kollekciók

```
{ Kollekció használata lottószelvények kiértékelésére }
program KOLL;
uses crt;
const
  FNevI='KOLL_I.TXT';    { A szelvényeket tartalmazó szövegfájl }
  FNevO='KOLL_O.TXT';    { A kiértékelt szelvények szövegfájlja }
  MaxSzelv=16000;        { A betölthető szelvények max. száma }
```

```
  MaxGen=20000;          { A generálandó szelvények száma }
  Db=6;                  { A lottószámok száma egy szelvényen }
  Max=45;                { Egy lottószám maximális értéke }
  MinTal=3;              { A minimális találatszám a nyeréshez }
type
  Szelveny=array[1..Db] of byte;          { Lottószelvény }
  Tetel=record                            { A kollekció tétele }
    Sz:Szelveny;                          { A számok }
    Tal:byte;                    { Találatok száma a szelvényen }
  end;
  TetelMut=^Tetel;
  Kollekcio=array[1..MaxSzelv] of TetelMut;     { A kollekció }
  Stat=array[0..Db] of word;                    { Statisztika }

{ Egy db lottószelvény generálása }
procedure General(var Sz:Szelveny);
var h:set of byte; i,j,x:byte;
begin
  h:=[];
  for i:=1 to Db do begin
    repeat
      x:=random(Max)+1;
    until not (x in h);
    h:=h+[x];
  end;
  i:=0;
  for j:=1 to Max do
    if j in h then begin
      inc(i); Sz[i]:=j;
    end;
end;

{ N db lottószelvény generálása az FNev nevű szövegfájlba }
procedure FajlbaGeneral(FNev:string; N:word);
var
  f:text; Sz:Szelveny;
  i,j:word;
begin
  assign(f,FNev);
  rewrite(f);
  for i:=1 to N do begin
    General(Sz);
    for j:=1 to Db do write(f,Sz[j]:3);
    writeln(f);
  end;
  close(f);
end;

{ Az FNev nevű szövegfájl szelvényeinek betöltése }
function Betolt(FNev:string; var K:Kollekcio; var
KDb:word):boolean;
var f:text; i:byte; Ok:boolean;
```

```
begin
  { Adatbeolvasás }
  assign(f,FNev);
  reset(f);
  KDb:=0;
  while not eof(f) and (maxavail>=sizeof(Tetel))
        and (KDb<MaxSzelv) do begin
    inc(KDb);
    new(K[KDb]);
    for i:=1 to Db do read(f,K[KDb]^.Sz[i]);
    readln(f);
  end;
  Ok:=eof(f);
  close(f);
  Betolt:=Ok;
end;

{ Kiértékelés, a nyerőszelvények kiírása az FNev nevű szöveg-
fájlba }
procedure Ertekel(FNev:string; var K:Kollekcio; KDb:word; const
Ny:Szelveny);
var f:text; h:set of byte;
    S:Stat; i,j,l:word;
begin
  { Nyerőszámok halmaza }
  h:=[];
  for i:=1 to Db do h:=h+[Ny[i]];
  { Statisztika kezdőértéke }
  for i:=0 to Db do S[i]:=0;
  { A szelvények kiértékelése }
  for i:=1 to KDb do begin
    K[i]^.Tal:=0;
    for j:=1 to Db do
      if K[i]^.Sz[j] in h then inc(K[i]^.Tal);
    { Statisztika }
    inc(S[K[i]^.Tal]);
  end;
  { Adatkiírás }
  assign(f,FNev);
  rewrite(f);
  writeln(f,'A nyerőszámok');
  for i:=1 to Db do write(f,Ny[i]:3); writeln(f);
  { A nyerőszelvények }
  for l:=Db downto MinTal do begin
    writeln(f,l,' találatos szelvények száma:',S[l]);
    for i:=1 to KDb do
      if K[i]^.Tal=l then begin
        for j:=1 to Db do
          write(f,K[i]^.Sz[j]:3);
        writeln(f);
      end;
  end;
```

```
  close(f);
end;

var
  K:Kollekcio; KDb:Word; Ny:Szelveny;
begin
  clrscr; randomize;
  { Adatgenerálás }
  FajlbaGeneral(FNevI,MaxGen);
  writeln(MaxGen,' db szelvény generálva (',FNevI,')');
  { Adatbeolvasás }
  if not Betolt(FNevI,K,KDb) then
    writeln('Nem fért be minden szelvény!');
  writeln('A betöltött szelvények száma:',KDb);
  { Nyerőszámok generálása }
  General(Ny);
  { Kiértékelés }
  Ertekel(FNevO,K,KDb,Ny);
  writeln('A kiértékelés elkészült (',FNevO,')');
end.
```

17.2.39. Láncolt listák

```
{ Egy- és kétirányban láncolt listák }
program LISTAK;
uses crt;
type
  Elem=integer;
  LancElem1Mut=^LancElem1;
  LancElem1=record
    Adat:Elem;
    Koveto:LancElem1Mut;
  end;
  LancElem2Mut=^LancElem2;
  LancElem2=record
    Adat:Elem;
    Elozo,Koveto:LancElem2Mut;
  end;

{ Keresés egy egyirányban láncolt, rendezetlen listában }
function Kereses1(Elso:LancElem1Mut; x:Elem):LancElem1Mut;
var Akt:LancElem1Mut;
begin
  Akt:=Elso;
  while (Akt<>nil) and (Akt^.Adat<>x) do
    Akt:=Akt^.Koveto;
  Kereses1:=Akt;
end;

{ Keresés egy egyirányban láncolt, rendezett listában }
function KeresesRend1(Elso:LancElem1Mut; x:Elem):LancElem1Mut;
var Akt:LancElem1Mut;
```

```
begin
  Akt:=Elso;
  while (Akt<>nil) and (Akt^.Adat<x) do
    Akt:=Akt^.Koveto;
  if (Akt<>nil) and (Akt^.Adat>x) then
    Akt:=nil;
  KeresesRend1:=Akt;
end;

{ Egy egyirányban láncolt, rendezett lista bővítése egy új lis-
taelemmel }
procedure ListaraRend1(var Elso:LancElem1Mut; Uj:LancElem1Mut);
var Akt,Miutan:LancElem1Mut;
begin
  { Keresés }
  Akt:=Elso; Miutan:=nil;
  while (Akt<>nil) and (Akt^.Adat<Uj^.Adat) do begin
    Miutan:=Akt;
    Akt:=Akt^.Koveto;
  end;
  { Beillesztés }
  if Elso=nil then begin
    { Üres listára }
    Uj^.Koveto:=nil;
    Elso:=Uj;
  end else if Miutan=nil then begin
    { Nem üres lista elejére }
    Uj^.Koveto:=Elso;
    Elso:=Uj;
  end else if Akt=nil then begin
    { A lista végére, a Miutan után }
    Uj^.Koveto:=nil;
    Miutan^.Koveto:=Uj;
  end else begin
    { A Miutan és az Akt közé }
    Uj^.Koveto:=Miutan^.Koveto;
    Miutan^.Koveto:=Uj;
  end;
end;

{ Beillesztés egy kétirányban láncolt lista végére }
procedure Listara2(var Elso,Utolso:LancElem2Mut;
Uj:LancElem2Mut);
begin
  if Elso=nil then begin
    { Üres listára }
    Uj^.Elozo:=nil;
    Uj^.Koveto:=nil;
    Elso:=Uj;
    Utolso:=Uj;
  end else begin
    { Az Utolso után }
```

```
    Uj^.Elozo:=Utolso;
    Uj^.Koveto:=nil;
    Utolso^.Koveto:=Uj;
    Utolso:=Uj;
  end;
end;

{ Elem törlése egy kétirányban láncolt listából }
procedure Listarol2(var Elso,Utolso:LancElem2Mut;
Mit:LancElem2Mut);
begin
  { Kikapcsolás }
  if (Mit=Elso) and (Mit=Utolso) then begin
    { Egyetlen elem }
    Elso:=nil;
    Utolso:=nil;
  end else if Mit=Elso then begin
    { Első, de nem egyetlen }
    Mit^.Koveto^.Elozo:=nil;
    Elso:=Mit^.Koveto;
  end else if Mit=Utolso then begin
    { Utolsó, de nem egyetlen }
    Mit^.Elozo^.Koveto:=nil;
    Utolso:=Mit^.Elozo;
  end else begin
    { Belső elem }
    Mit^.Elozo^.Koveto:=Mit^.Koveto;
    Mit^.Koveto^.Elozo:=Mit^.Elozo;
  end;
  { Megszüntetés }
  dispose(Mit);
end;

var Elso1,Akt1:LancElem1Mut;
    Elso2,Utolso2,Akt2:LancElem2Mut;
    a:Elem; i:integer;
begin
  clrscr;
  Elso1:=nil;
  writeln('Adatmegadás vége:0');
  i:=1;
  repeat
    write(i,'. elem:'); readln(a); inc(i);
    if a<>0 then begin
      if sizeof(LancElem1)<=maxavail then begin
        { Van hely a rendezett láncba illesszük }
        new(Akt1);
        Akt1^.Adat:=a;
        ListaraRend1(Elso1,Akt1);
      end;
    end;
  until a=0;
```

```
writeln('A megadott elemek rendezve:');
Akt1:=Elso1; i:=1;
while Akt1<>nil do begin
  writeln(i,'. elem:',Akt1^.Adat);
  Akt1:=Akt1^.Koveto;
  inc(i);
end;

write('A keresett elem:'); readln(a);
if Kereses1(Elso1,a)=nil then
  writeln('Nincs ilyen!')
else
  writeln('Van ilyen!');
if KeresesRend1(Elso1,a)=nil then
  writeln('Nincs ilyen!')
else
  writeln('Van ilyen!');

{ A foglalt hely felszabadítása }
while Elso1<>nil do begin
  Akt1:=Elso1;
  Elso1:=Elso1^.Koveto;
  dispose(Akt1);
end;

Elso2:=nil; Utolso2:=nil;
writeln('Adatmegadás vége:0');
i:=1;
repeat
  write(i,'. elem:'); readln(a); inc(i);
  if a<>0 then begin
    if sizeof(LancElem2)<=maxavail then begin
      { Van hely, a lánc végére illesszük }
      new(Akt2);
      Akt2^.Adat:=a;
      Listara2(Elso2,Utolso2,Akt2);
    end;
  end;
until a=0;

writeln('A megadott elemek eredeti sorrendben:');
Akt2:=Elso2; i:=1;
while Akt2<>nil do begin
  writeln(i,'. elem:',Akt2^.Adat);
  Akt2:=Akt2^.Koveto;
  inc(i);
end;

writeln('A megadott elemek fordított sorrendben:');
Akt2:=Utolso2; i:=1;
while Akt2<>nil do begin
  writeln(i,'. elem:',Akt2^.Adat);
```

```
    Akt2:=Akt2^.Elozo;
    inc(i);
  end;

  { A foglalt hely felszabadítása }
  { (lehetne az utolsó törlésével is) }
  while Elso2<>nil do
    Listarol2(Elso2,Utolso2,Elso2);
end.
```

17.2.40. Összetett listák

```
{ Tárgymutató }
program TARGYMUT;
uses crt;
const
  MaxSzoHossz=20;                    { A szavak maximális hossza }
type
  TSzo=string[MaxSzoHossz];              { A szavak típusa }
  THiv=word;                             { A hivatkozások típusa }

  HivRekMut=^HivRek;               { A hivatkozásrekordra mutató }
  HivRek=record                            { A hivatkozásrekord }
    Oldal:THiv;
    Kov:HivRekMut;
  end;

  SzoRekMut=^SzoRek;                    { A szórekordra mutató }
  SzoRek=record                                { A szórekord }
    Szo:TSzo;
    EHiv,UHiv:HivRekMut;
    Kov:SzoRekMut;
  end;

{ Új szórekord létrehozása }
function UjSzoRek(Szo:TSzo; Oldal:THiv;
KovSzo:SzoRekMut):SzoRekMut;
var UjSzo:SzoRekMut; UjHiv:HivRekMut;
begin
  new(UjSzo);
  UjSzo^.Szo:=Szo;
  UjSzo^.Kov:=KovSzo;
  new(UjHiv);
  UjSzo^.EHiv:=UjHiv;
  UjSzo^.UHiv:=UjHiv;
  UjHiv^.Oldal:=Oldal;
  UjHiv^.Kov:=nil;
  UjSzoRek:=UjSzo;
end;

{ A tárgymutató bővítése }
procedure Bovit(var TMKezd:SzoRekMut; Szo:TSzo; Oldal:THiv);
```

```
var Akt,Elozo:SzoRekMut; UjHiv:HivRekMut;
begin
  if TMKezd=nil then
    { Üres a tárgymutató }
    TMKezd:=UjSzoRek(Szo,Oldal,nil)
  else begin
    { Keresés }
    Akt:=TMKezd;
    Elozo:=nil;
    while (Akt^.Szo<Szo) and (Akt^.Kov<>nil) do begin
      Elozo:=Akt;
      Akt:=Akt^.Kov;
    end;
    if Akt^.Szo<Szo then
      { Végére új szó }
      Akt^.Kov:=UjSzoRek(Szo,Oldal,nil)
    else if Akt^.Szo>Szo then
      { Az Akt és az Elozo közé új szó }
      if Elozo=nil then
        { A lista elejére }
        TMKezd:=UjSzoRek(Szo,Oldal,TMKezd)
      else
        { A lista belsejébe }
        Elozo^.Kov:=UjSzoRek(Szo,Oldal,Akt)
    else begin
      { Meglévő szó (Akt) új hivatkozása }
      new(UjHiv);
      UjHiv^.Oldal:=Oldal;
      UjHiv^.Kov:=nil;
      Akt^.UHiv^.Kov:=UjHiv;
      Akt^.UHiv:=UjHiv;
    end;
  end;
end;

{ A tárgymutató adatainak kiírása a képernyőre }
procedure Kiir(TMKezd:SzoRekMut);
var Akt:SzoRekMut; AktHiv:HivRekMut;
begin
  Akt:=TMKezd;
  while Akt<>nil do begin
    write(Akt^.Szo,'':MaxSzoHossz-length(Akt^.Szo));
    AktHiv:=Akt^.EHiv;
    while AktHiv<>nil do begin
      write(AktHiv^.Oldal:6);
      AktHiv:=AktHiv^.Kov;
    end;
    writeln; readkey;
    Akt:=Akt^.Kov;
  end;
end;
```

```
var TMKezd:SzoRekMut; VanHely,Vege:boolean;
    Szo:TSzo; Hiv:THiv;
begin
  clrscr;
  TMKezd:=nil;
  { Feltöltés input adatokkal }
  writeln('A tárgymutató adatai (Kilépés:"*" szó)\n');
  repeat
    { Lesz majd elég hely? }
    VanHely:=sizeof(SzoRek)+sizeof(HivRek)<=memavail;
    if VanHely then begin
      { Igen }
      write('Szó:'); readln(Szo);
      Vege:=Szo='*';
      if not Vege then begin
        write('Hivatkozás:'); readln(Hiv);
        Bovit(TMKezd,Szo,Hiv);
      end;
    end;
  until Vege or not VanHely;
  { Kiírás }
  if TMKezd=nil then
    writeln('Nem adott meg adatokat!')
  else begin
    writeln('A tárgymutató:');
    Kiir(TMKezd);
  end;
end.
```

17.2.41. Szövegfájl képernyőre listázása

```
{ Szövegfájl kilistázása a képernyöre karakterenként }
program SZFLIST;
uses crt;
const db=20;          { Max. ennyi sort írunk ki egy képernyőre }
var f:text; ch:char; s:integer;
begin
  clrscr; s:=0;
  assign(f,'szflist.pas');
  reset(f);
  while not eof(f) do begin
    read(f,ch);
    write(ch);
    if ch=chr(10) then begin
      inc(s);
      if s mod db=0 then readkey;
    end;
  end;
  if s mod db<>0 then readkey;
  close(f);
end.
```

17.2.42. Összefésüléses fájlrendezés

```
{ Típusos fájl rekordjainak rendezése összefésüléssel }
program TIPFREND;
uses crt;
const
  fnev1='adatokp1.dta';      { A rendezendő adatok }
  fnev2='adatokp2.dta';      { Segédfájl }
  fnev3='adatokp3.dta';      { Segédfájl }

  MaxElemSzam=10;          { A generált adatok maximális száma }
  MaxElem=20;              { A generált adatok maximális értéke }
type
  TAdat=record               { A rendezendő rekordok }
    A:integer;               { Ezen mező szerint rendezünk }
  end;
  AdatFajl=file of TAdat;    { Az adatfájl }

{ A fájlmutatót egy rekorddal vissza }
procedure Vissza(var f:AdatFajl);
begin
  seek(f,filepos(f)-1);
end;

{ Fájlvégén állunk-e }
function FajlVege(var f:AdatFajl):boolean;
begin
  FajlVege:=eof(f);
end;

function LancVege(var Miben:AdatFajl; const Akt:TAdat):boolean;
var Kov:TAdat; Er:boolean;
begin
  if FajlVege(Miben) then Er:=true
  else begin
    read(Miben,Kov);
    Vissza(Miben);
    Er:=Kov.A<Akt.A;
  end;
  LancVege:=Er;
end;

procedure LancMasol(var Bol,Ba:AdatFajl);
var Akt:TAdat;
begin
  repeat
    read(Bol,Akt); write(Ba,Akt);
  until LancVege(Bol,Akt);
end;

procedure Rendezes;
var a,b,c:AdatFajl;
    LancDb:integer; LancVeg:boolean;
```

```
    akt_a,akt_b:TAdat;
begin
  assign(c,fnev1); assign(a,fnev2); assign(b,fnev3);
  repeat
    { Szétosztás }
    reset(c); rewrite(a); rewrite(b);
    while not FajlVege(c) do begin
      LancMasol(c,a);
      if not FajlVege(c) then
        LancMasol(c,b);
    end;
    close(a); close(b); close(c);
    { Összefésülés }
    rewrite(c); reset(a); reset(b);
    LancDb:=0;
    while not FajlVege(a) and not FajlVege(b) do begin
      { Egy lánc összefésülése }
      read(a,akt_a); read(b,akt_b);
      repeat
        if akt_a.A<=akt_b.A then begin
          { a-ból egy elemet }
          write(c,akt_a);
          LancVeg:=LancVege(a,akt_a);
          if LancVeg then begin
            Vissza(b);
            LancMasol(b,c);
          end else
            read(a,akt_a);
        end else begin
          { b-ből egy elemet }
          write(c,akt_b);
          LancVeg:=LancVege(b,akt_b);
          if LancVeg then begin
            Vissza(a);
            LancMasol(a,c);
          end else
            read(b,akt_b);
        end;
      until LancVeg;
      inc(LancDb);
    end;
    { Maradékok másolása }
    if not FajlVege(a) then begin
      while not FajlVege(a) do begin
        read(a,akt_a);
        write(c,akt_a);
      end;
      inc(LancDb);
    end;
    if not FajlVege(b) then begin
      while not FajlVege(b) do begin
        read(b,akt_b);
```

```
        write(c,akt_b);
      end;
      inc(LancDb);
    end;
    close(a); close(b); close(c);
  until LancDb<=1;
end;

var f:AdatFajl; Adat:TAdat;
    i,db:integer;
begin
  { Véletlen elemgenerálás }
  randomize; clrscr;
  writeln('Rendezés előtt');
  assign(f,fnev1);
  rewrite(f);
  for i:=1 to MaxElemSzam do begin
    Adat.A:=random(MaxElem+1);
    write(f,Adat);
    write(Adat.A,' ');
  end; writeln;
  close(f);

  { Sorbarendezés }
  Rendezes;

  { Eredménykiírás }
  writeln('Rendezés után');
  reset(f);
  while not FajlVege(f) do begin
    read(f,Adat);
    write(Adat.A,' ');
  end; writeln;
  close(f);

  { Munkafájlok törlése }
  assign(f,fnev2); erase(f);
  assign(f,fnev3); erase(f);
end.
```

17.2.43. Indextáblás fájlkezelés

```
{ Típusos fájl kezelése indextáblával }
program TIPFINDT;
uses crt;
const
  { Fájlnevek }
  afnev='adatp.dat';          { Adatfájl }
  ifnev='adatp.ind';          { Indextábla }

  MaxDb=5;                    { Az adatok maximális száma }
  MaxAzHossz=1;               { Az azonosító maximális hossza }
```

```
  MaxInfoHossz=3;             { Az információ maximális hossza }
  UresAz=' ';                 { Az üres azonosító }
type
  TAz=string[MaxAzHossz];     { Az adatrekord azonosító típusa }
  TInfo=string[MaxInfoHossz];{ Az adatrekord információ típusa }

  TAdat=record                { Az adatrekord típusa }
    Az:TAz;
    Info:TInfo;
  end;

  TItAdat=record           { Az indextábla egy elemének típusa }
    Az:TAz;                { Ez alapján indexezünk }
    Poz:integer;           { A rekord pozíciója az adatfájlban }
  end;

  { Az indextábla típusa }
  { A 0. rekord Poz értéke az adatfájl érvényes rekordjainak
száma }
  TIt=array[0..MaxDb] of TItAdat;
  { Az adatfájl tipusa }
  AdatFajl=file of TAdat;
  { Az indexfájl tipusa }
  IndexFajl=file of TIt;

{ Bináris keresés az indextáblában }
function BinKer(Mit:TAz; const Miben:TIt; Db:integer;
 var Hol:integer):boolean;
var i,j,k:integer; Van:boolean;
begin
  i:=1; j:=Db; Van:=false;
  while (i<=j) and not Van do begin
    k:=(i+j) div 2;
    if Mit=Miben[k].Az then Van:=true
    else
      if Mit<Miben[k].Az then j:=k-1
      else i:=k+1;
   end;
  if Van then Hol:=k
  else Hol:=i;
  BinKer:=Van;
end;

{ Beszúrás az indextáblába }
procedure Beszur(const Mit:TItAdat; var Mibe:TIt;
 Hova:integer; var Db:integer);
var i:integer;
begin
  { Helykészítés }
  for i:=Db downto Hova do
    Mibe[i+1]:=Mibe[i];
  { Beszúrás }
```

```
  Mibe[Hova]:=Mit;
  { Darabszám növelés }
  inc(Db);
end;

{ Törlés az indextáblából }
procedure Torol(var Mibol:TIt; Honnan:integer; var Db:integer);
var i:integer; s:TItAdat;
begin
  { A törlendő elem megjegyzése }
  s:=Mibol[Honnan];
  { Törlés }
  for i:=Honnan+1 to Db do
    Mibol[i-1]:=Mibol[i];
  { Hogy a fájlba majd ennek helyére vegyünk fel legközelebb }
  Mibol[Db]:=s;
  { Darabszám csökkentés }
  dec(Db);
  if Db=0 then
    { Az elejéröl töltsük fel az adatfájlt }
    for i:=1 to MaxDb do
      Mibol[i].Poz:=i-1;
end;

{ Elem felvétel }
procedure Felvetel(var Mibe:TIt; var Db:integer; var
af:AdatFajl);
var Hol:integer;
    Adat:TAdat; ItAdat:TItAdat;
begin
  if Db=MaxDb then
    writeln('Nem vehetö fel több elem!')
  else with Adat do begin
    write('Azonosító (max ',MaxAzHossz,' karakter):');
    readln(Az);
    if BinKer(Az,Mibe,Db,Hol) then
      writeln('Van már ilyen azonosítójú rekord!')
    else begin
      write('Információ (max ',MaxInfoHossz,' karakter):');
      readln(Info);
      { Beírás az adatfájlba }
      seek(af,Mibe[Db+1].Poz);
      write(af,Adat);
      { Felvétel az indextáblába }
      ItAdat.Az:=Adat.Az; ItAdat.Poz:=Mibe[Db+1].Poz;
      Beszur(ItAdat,Mibe,Hol,Db);
    end;
  end;
end;

{ Elem törlés }
procedure Torles(var Mibol:TIt; var Db:integer);
```

```
var Mit:TAz; Hol:integer;
begin
  if db=0 then
    writeln('Nincs mit törölni!')
  else begin
    write('Azonosító (max ',MaxAzHossz,' karakter):');
    readln(Mit);
    if not BinKer(Mit,Mibol,Db,Hol) then
      writeln('Nincs ilyen rekord!')
    else
      Torol(Mibol,Hol,Db);
  end;
end;

procedure Kiir(x,y:integer; Mit:string);
begin
  gotoxy(x,y); write(Mit);
end;

{ Adatok kiírása }
procedure Kiiras(const it:TIt; var af:AdatFajl);
var i:integer; Adat:TAdat;
begin
  clrscr;
  { Fejléc  12345678901234567890123456 }
  Kiir(1,1,'Ssz  Indextábla  Adatfájl');
  Kiir(1,2,'     Az-ó Poz    Az-ó Info');
  { Indextábla }
  for i:=0 to MaxDb do begin
    gotoxy(2,4+i); write(i,'.');
    if i<=it[0].Poz then
      Kiir(6,4+i,it[i].Az)
    else
      Kiir(6,4+i,UresAz);
    gotoxy(11,4+i); write(it[i].Poz);
  end;
  { Adatfájl }
  seek(af,0); i:=0;
  while not eof(af) do begin
    read(af,Adat);
    Kiir(18,4+i,Adat.Az);
    Kiir(23,4+i,Adat.Info);
    inc(i);
  end;
  gotoxy(1,4+MaxDb+2); writeln('Elemek száma:',it[0].Poz);
end;

{ Fájl létezésének vizsgálata }
function FajlVan(Nev:string):boolean;
var f:file; Er:boolean;
begin
  if Nev='' then Er:=false
```

```
  else begin
    assign(f,Nev);
    {$i-} reset(f); {$i+}
    if ioresult=0 then begin
      Er:=true; close(f);
    end else Er:=false;
  end;
  FajlVan:=Er;
end;

var af:AdatFajl; itf:IndexFajl;
    it:TIt; i:integer; c:char;
begin
  clrscr;
  assign(af,afnev); assign(itf,ifnev);
  if FajlVan(afnev) and FajlVan(ifnev) then begin
    reset(af);
    { Az indextábla betöltése }
    reset(itf); read(itf,it); close(itf);
  end else begin
    rewrite(af);
    { Indextábla inicializálás }
    for i:=0 to MaxDb do begin
      it[i].Az:=UresAz;
      if i=0 then it[i].Poz:=0     { A létező adatok száma }
      else it[i].Poz:=i-1;         { A leendő rekordsorszámok }
    end;
  end;
  { Kilépésig }
  repeat
    writeln; writeln('Felvétel:1 Törlés:2 Kiírás:3 Kilépés:0');
    repeat c:=readkey; until c in ['0'..'3'];
    case c of
      '1':Felvetel(it,it[0].Poz,af);
      '2':Torles(it,it[0].Poz);
      '3':Kiiras(it,af);
    end;
  until c='0';
  close(af);
  { Az indextábla mentése }
  rewrite(itf); write(itf,it); close(itf);
end.
```

17.2.44. Bináris fák

```
{ Bináris keresőfa }
program BINKERFA;
uses crt;
type
  Elem=integer;
  BinFaPontMut=^BinFaPont;
  BinFaPont=record
    PontJell:Elem;
```

```
    BalAg,JobbAg:BinFaPontMut;
  end;

{ Keresés bináris keresőfában }
function Keres(Gyoker:BinFaPontMut; x:Elem):BinFaPontMut;
var Akt:BinFaPontMut;
begin
  Akt:=Gyoker;
  while (Akt<>NIL) and (x<>Akt^.PontJell) do
    if x<Akt^.PontJell then
      Akt:=Akt^.BalAg
    else
      Akt:=Akt^.JobbAg;
  Keres:=Akt;
end;

{ Bináris keresőfa bővítése }
procedure Bovit(var Gyoker:BinFaPontMut; x:BinFaPontMut);
var Akt,Szulo:BinFaPontMut;
begin
  if Gyoker=NIL then
    Gyoker:=x
  else begin
    { Helykeresés }
    Akt:=Gyoker; Szulo:=NIL;
    while Akt<>NIL do begin
      Szulo:=Akt;
      if x^.PontJell<Akt^.PontJell then
        Akt:=Akt^.BalAg
      else
        Akt:=Akt^.JobbAg;
    end;
    { Beillesztés }
    if x^.PontJell<Szulo^.PontJell then
      Szulo^.BalAg:=x
    else
      Szulo^.JobbAg:=x;
  end;
end;

{ Bináris keresőfa elemeinek kiírása }
procedure Kiir(Gyoker:BinFaPontMut);
begin
  if Gyoker<>NIL then begin
    { Rekurzív hívás a bal ágra }
    Kiir(Gyoker^.BalAg);
    { A gyökérponthoz tartozó érték kiírása }
    write(Gyoker^.PontJell,' ');
    { Rekurzív hívás a bal ágra }
    Kiir(Gyoker^.JobbAg);
  end;
end;
```

```
var Gyoker,Akt:BinFaPontMut; a:Elem; i:integer;
begin
  clrscr;
  Gyoker:=NIL;
  writeln('Adatmegadás vége:0');
  i:=1;
  repeat
    write(i,'. elem:'); inc(i); readln(a);
    if a<>0 then begin
      if sizeof(BinFaPont)<=maxavail then begin
        { Van hely, a keresőfába illesszük }
        new(Akt);
        with Akt^ do begin
          PontJell:=a; BalAg:=NIL; JobbAg:=NIL;
        end;
        Bovit(Gyoker,Akt);
      end;
    end;
  until a=0;

  writeln('A megadott elemek rendezve:');
  Kiir(Gyoker);
  writeln;

  write('A keresett elem:'); readln(a);
  if Keres(Gyoker,a)=NIL then
    writeln('Nincs ilyen elem!')
  else
    writeln('Van ilyen elem!');
end.
```

17.2.45. Kupacrendezés

```
{ Kupacrendezés }
program KUPAC;
uses crt;
const
  MaxElemSzam=10;           { A rendezendő adatok max. száma }
  MaxElem=20;               { A generálandó adatok max. értéke }
type
  Elem=integer;                  { A rendezendő elemek típusa }
  Tomb=array[1..MaxElemSzam] of Elem;       { Az elemek tömbje }

{ Az a[l] besüllyesztése az a[l+1], a[l+2],..., a[r] elemek közé
}
procedure Sullyeszt(var a:Tomb; l,r:integer);
var i,j:integer; vege:boolean; x:Elem;
begin
  x:=a[l]; i:=l; j:=2*i; vege:=false;
  while (j<=r) and not vege do begin
    { A kisebb gyerekkel hasonlítsunk }
    if (j<r) and (a[j+1]<a[j]) then inc(j);
```

```
    if x<=a[j] then
      { Megvan a helye }
      vege:=true
    else begin
      { A gyerek feljebb léptetése }
      a[i]:=a[j]; i:=j; j:=2*i;
    end;
  end;
  { Elemet a helyére }
  a[i]:=x;
end;

procedure KupacRend(var a:Tomb; n:integer);
var l,r:integer; x:Elem;
begin
  { Kezdőkupac }
  l:=n div 2+1; r:=n;
  while l>1 do begin
    dec(l);
    Sullyeszt(a,l,r);
  end;
  while r>1 do begin
    { A legkisebbet hátra }
    x:=a[1]; a[1]:=a[r]; a[r]:=x;
    { A kupacot kisebbre }
    dec(r);
    { A hátul volt elem besüllyesztése }
    Sullyeszt(a,l,r);
  end;
end;

{ Elemek generálása }
procedure General(var a:Tomb; n:integer);
var i:integer;
begin
  for i:=1 to n do
    a[i]:=random(MaxElem+1);
end;

procedure Kiiras(Szoveg:string; const a:Tomb; n:integer);
var i:integer;
begin
  writeln(Szoveg);
  for i:=1 to n do write(a[i],' ');
  writeln;
end;

var a:Tomb;
begin
  randomize; clrscr;

  General(a,MaxElemSzam);
```

```
  Kiiras('Rendezés előtt',a,MaxElemSzam);

  { Sorbarendezés }
  KupacRend(a,MaxElemSzam);

  Kiiras('Rendezés után',a,MaxElemSzam);
end.
```

17.2.46. Útkeresés

Mátrix módszer

```
{ Útkeresés Floyd-Warshall módszerrel }
program FLO_WAR;
uses crt;
const
  MaxPontDb=5;                      { A pontok maximális száma }
  MaxElDb=MaxPontDb*(MaxPontDb-1);  { Az élek maximális száma }
  MaxElHossz=99;                    { A maximális élhossz }
  NincsEl=-1;                       { A nemlétező él jelzésére }
  MaxUtHossz=MaxElDb*MaxElHossz;    { A maximális úthossz }
  Vegtelen=MaxUtHossz+1;            { A 'végtelen' kezeléséhez }
  KellKiiras=true;                  { Legyenek-e kiírások }
type
  Pont=record                       { A pontrekord }
    Azon:char;                      { Pontazonosító }
    x,y:integer;                    { Koordináták }
  end;
  { A mátrixos tároláshoz }
  Matrix=array[1..MaxPontDb,1..MaxPontDb] of integer;

  Graf=record                                { A gráf }
    PontDb:integer;                          { Pontok száma }
    Pontok:array[1..MaxPontDb] of Pont;      { Pontok }
    ElHossz:Matrix;                          { Élhossz mátrix }
  end;

procedure MtxKiir(Szoveg:string; const a:Matrix; n:integer);
var i,j:integer;
begin
  writeln(Szoveg);
  for i:=1 to n do begin
    for j:=1 to n do
      if a[i,j]=Vegtelen then
        write('-':6)
      else
        write(a[i,j]:6);
    writeln;
  end;
end;

procedure Floyd_Warshall(const G:Graf; var T,C:Matrix);
var x,y,w:integer;
```

```
begin
  { Kezdőállapot }
  for x:=1 to G.PontDb do
    for y:=1 to G.PontDb do begin
      if x=y then
        T[x,y]:=0
      else
        if G.ElHossz[x,y]=NincsEl then
          T[x,y]:=Vegtelen
        else
          T[x,y]:=G.ElHossz[x,y];
      C[x,y]:=y;
    end;
  if KellKiiras then begin
    writeln('A kezdőállapot');
    MtxKiir('T:',T,G.PontDb);
    MtxKiir('C:',C,G.PontDb);
    readkey;
  end;
  { Javító lépések }
  for w:=1 to G.PontDb do begin
    for x:=1 to G.PontDb do
      for y:=1 to G.PontDb do
        if T[x,w]+T[w,y]<T[x,y] then begin
          T[x,y]:=T[x,w]+T[w,y];
          C[x,y]:=C[x,w];
        end;
    if KellKiiras then begin
      writeln(w,'. javító lépés után:');
      MtxKiir('T:',T,G.PontDb);
      MtxKiir('C:',C,G.PontDb);
      readkey;
    end;
  end;
end;

procedure GrafKiir(const G:Graf);
var i,j,k:integer;
begin
  writeln('Pontok száma:',G.PontDb);
  writeln('Index Az-ó     X     Y');
  for i:=1 to G.PontDb do
    writeln(i+1:5,' ',G.Pontok[i].Azon:4,
            G.Pontok[i].x:6,G.Pontok[i].y:6);
  readkey;
  writeln('A gráf élei:');
  k:=1;
  for i:=1 to G.PontDb do
    for j:=1 to G.PontDb do
      if (i<>j) and (G.ElHossz[i,j]<>NincsEl) then begin
        writeln(G.Pontok[i].Azon,' - ',G.Pontok[j].Azon,' : ',
                G.ElHossz[i,j]);
```

```
        inc(k);
        if k mod 20=0 then readkey;
      end;
  readkey;
end;

procedure UtKiir(const G:Graf; const T,C:Matrix; Kp,Vp:integer);
var x:integer;
begin
  writeln(G.Pontok[Kp].Azon,' - ',G.Pontok[Vp].Azon,
          ' viszonylat');
  if T[Kp,Vp]=Vegtelen then
    writeln('Nem lehet eljutni!')
  else begin
    writeln('A távolság:',T[Kp,Vp]);
    write('Az út:');
    x:=Kp;
    write(G.Pontok[x].Azon,' ');
    repeat
      x:=C[x,Vp];
      write(G.Pontok[x].Azon,' ');
    until x=Vp;
    writeln;
    readkey;
  end;
end;

const
  { A példagráf mátrixos tárolásban }
  G:Graf=(PontDb:5;
          Pontok:((Azon:'1';x:0;y:20),(Azon:'2';x:15;y:20),
                  (Azon:'3';x:8;y:12),(Azon:'4';x:0;y:0),
                  (Azon:'5';x:15;y:0));
          ElHossz:((0,15,10,NincsEl,NincsEl),
                   (NincsEl,0,NincsEl,NincsEl,20),
                   (10,10,0,13,NincsEl),
                   (20,NincsEl,NincsEl,0,15),
                   (NincsEl,20,13,NincsEl,0)));
var T,C:Matrix;
begin
  clrscr;
  if KellKiiras then
    GrafKiir(G);

  Floyd_Warshall(G,T,C);

  { A 2. pontból a 4. pontba }
  UtKiir(G,T,C,2,4);
end.
```

Faépítés

```
{ Útkeresés Dijkstra módszerrel }
program DIJKSTRAP;
uses crt;
const
  MaxPontDb=8;                        { A pontok maximális száma }
  MaxElDb=MaxPontDb*(MaxPontDb-1);    { Az élek maximális száma }
  MaxElHossz=99;                      { A maximális élhossz }
  NincsEl=-1;                         { A nemlétező él jelzésére }
  MaxUtHossz=MaxElDb*MaxElHossz;      { A maximális úthossz }
  Vegtelen=MaxUtHossz+1;              { A 'végtelen' kezeléséhez }
  KellKiiras=true;                    { Legyenek-e kiírások }
  NincsCimke=-1;                      { A nemlétező címke }
type
  Pont=record                         { A pontrekord }
    Azon:char;                        { Pontazonosító }
    x,y:integer;                      { Koordináták }
  end;
  { Az éleket egydimenziós tömbben tároljuk }
  El=record                          { Az élrekord }
    Vp:integer;                      { Végpontindex }
    ElHossz:integer;                 { Élhossz }
  end;
  Graf=record                                  { A gráf }
    PontDb:integer;                            { Pontok száma }
    Pontok:array[1..MaxPontDb] of Pont;        { Pontok }
    ElMut:array[1..MaxPontDb+1] of integer;    { Élmutatók }
    ElDb:integer;                              { Élek száma }
    Elek:array[1..MaxElDb] of El;              { Élek }
  end;
  { Az egydimenziós tömbökhöz (A,T,C) }
  Tomb=array[1..MaxPontDb] of integer;

procedure GrafKiir(const G:Graf);
var i,j,k:integer;
begin
  writeln('Pontok száma:',G.PontDb);
  writeln('Index Az-ó     X     Y');
  for i:=1 to G.PontDb do
    writeln(i:5,' ',
            G.Pontok[i].Azon:4,G.Pontok[i].x:6,G.Pontok[i].y:6);
  readkey;
  writeln('A gráf élei:');
  k:=1;
  for i:=1 to G.PontDb do
    for j:=G.ElMut[i] to G.ElMut[i+1]-1 do begin
      writeln(G.Pontok[i].Azon,' - ',
              G.Pontok[G.Elek[j].Vp].Azon,' : ',
              G.Elek[j].ElHossz);
      inc(k);
      if k mod 20=0 then readkey;
    end;
```

```
  readkey;
end;

procedure AdatKiir(const G:Graf; const A,T,C:Tomb; ADb:integer);
var i:integer;
begin
  write('A:');
  for i:=1 to ADb do
    write(G.Pontok[A[i]].Azon:6);
  writeln; write('T:');
  for i:=1 to G.PontDb do
    if T[i]=Vegtelen then
      write('-':6)
    else
      write(T[i]:6);
  writeln; write('C:');
  for i:=1 to G.PontDb do
    if C[i]=NincsCimke then
      write(' ':6)
    else
      write(C[i]:6);
  writeln;
  readkey;
end;

procedure Dijkstra(const G:Graf; var T,C:Tomb; Kp:integer);
var A:Tomb; ADb:integer;
    Akt:array[1..MaxPontDb] of boolean;  { A pontok aktívsága }
    x,y,i,j,k:integer;
begin
  { Kezdőállapot }
  for i:=1 to G.PontDb do begin
    T[i]:=Vegtelen;
    Akt[i]:=false;
    C[i]:=NincsCimke;   { Csak a kiíráshoz kell }
  end;
  T[Kp]:=0;
  C[Kp]:=Kp;
  ADb:=1;
  A[ADb]:=Kp;
  Akt[Kp]:=true;
  if KellKiiras then begin
    writeln('A kezdőállapot');
    AdatKiir(G,A,T,C,ADb);
  end;
  { Javító lépések }
  if KellKiiras then j:=0;       { A javító lépések száma }
  while ADb>0 do begin
    { Az A minimális távolságú elemét X-be }
    k:=1;
    for i:=2 to ADb do
      if T[A[i]]<T[A[k]] then k:=i;
```

```
    x:=A[k];
    { X törlése A-ból (az utolsó elemmel felülírjuk) }
    A[k]:=A[ADb];
    ADb:=ADb-1;
    Akt[x]:=false;
    { Rövidítés X-n keresztül }
    for i:=G.ElMut[x] to G.ElMut[x+1]-1 do begin
      y:=G.Elek[i].Vp;
      if T[x]+G.Elek[i].ElHossz<T[y] then begin
        T[y]:=T[x]+G.Elek[i].ElHossz;
        C[y]:=x;
        if not Akt[y] then begin
          { Y még nem aktív, hozzávesszük A-hoz }
          ADb:=ADb+1;
          A[ADb]:=y;
          Akt[y]:=true;
        end;
      end;
    end;
    if KellKiiras then begin
      inc(j);
      writeln(j,'. javító lépés (x=',G.Pontok[x].Azon,') után');
      AdatKiir(G,A,T,C,ADb);
    end;
  end;
end;

procedure UtKiir(const G:Graf; const T,C:Tomb; Kp,Vp:integer);
var Ut:Tomb;       { A Kp-Vp út pontindexei }
    UtDb:integer;  { Az út pontjainak száma }
    x,cs:integer;
begin
  writeln(G.Pontok[Kp].Azon,' - ',
          G.Pontok[Vp].Azon,' viszonylat');
  if T[Vp]=Vegtelen then
    writeln('Nem lehet eljutni!')
  else begin
    { Az út összerakása (visszafelé haladva) }
    UtDb:=1;
    Ut[UtDb]:=Vp;
    x:=Vp;
    repeat
      x:=C[x];
      UtDb:=UtDb+1;
      Ut[UtDb]:=x;
    until x=Kp;
    { Az út megfordítása }
    for x:=1 to UtDb div 2 do begin
      cs:=Ut[x]; Ut[x]:=Ut[UtDb-x+1]; Ut[UtDb-x+1]:=cs;
    end;
    { Az út kiírása }
    writeln('A távolság:',T[Vp]);
```

```
    write('Az út:');
    for x:=1 to UtDb do
      write(G.Pontok[Ut[x]].Azon,' ');
    writeln;
    readkey;
  end;
end;

const
  { A példagráf az éleket egydimenziós tömbben tárolva }
  G:Graf=
  (PontDb:8;
   Pontok:((Azon:'1';x:0;y:20),(Azon:'2';x:20;y:40),
           (Azon:'3';x:40;y:20),(Azon:'4';x:20;y:0),
           (Azon:'5';x:10;y:20),(Azon:'6';x:20;y:30),
           (Azon:'7';x:30;y:20),(Azon:'8';x:20;y:10));
   ElMut:(1,4,7,10,13,17,20,24,27);
   ElDb:26;
   Elek:((Vp:2;ElHossz:40),(Vp:4;ElHossz:40),(Vp:5;ElHossz:10),
         (Vp:1;ElHossz:40),(Vp:3;ElHossz:40),(Vp:6;ElHossz:10),
         (Vp:2;ElHossz:40),(Vp:4;ElHossz:40),(Vp:7;ElHossz:10),
         (Vp:1;ElHossz:40),(Vp:3;ElHossz:40),(Vp:8;ElHossz:10),
         (Vp:1;ElHossz:10),(Vp:6;ElHossz:15),(Vp:7;ElHossz:20),
         (Vp:8;ElHossz:15),(Vp:2;ElHossz:10),(Vp:5;ElHossz:15),
         (Vp:7;ElHossz:15),(Vp:3;ElHossz:10),(Vp:5;ElHossz:20),
         (Vp:6;ElHossz:15),(Vp:8;ElHossz:15),(Vp:4;ElHossz:10),
         (Vp:5;ElHossz:15),(Vp:7;ElHossz:15),(Vp:0;ElHossz:0),
         (Vp:0;ElHossz:0),(Vp:0;ElHossz:0),(Vp:0;ElHossz:0),
         (Vp:0;ElHossz:0),(Vp:0;ElHossz:0),(Vp:0;ElHossz:0),
         (Vp:0;ElHossz:0),(Vp:0;ElHossz:0),(Vp:0;ElHossz:0),
         (Vp:0;ElHossz:0),(Vp:0;ElHossz:0),(Vp:0;ElHossz:0),
         (Vp:0;ElHossz:0),(Vp:0;ElHossz:0),(Vp:0;ElHossz:0),
         (Vp:0;ElHossz:0),(Vp:0;ElHossz:0),(Vp:0;ElHossz:0),
         (Vp:0;ElHossz:0),(Vp:0;ElHossz:0),(Vp:0;ElHossz:0),
         (Vp:0;ElHossz:0),(Vp:0;ElHossz:0),(Vp:0;ElHossz:0),
         (Vp:0;ElHossz:0),(Vp:0;ElHossz:0),(Vp:0;ElHossz:0),
         (Vp:0;ElHossz:0),(Vp:0;ElHossz:0)));
var T,C:Tomb;
begin
  clrscr;
  if KellKiiras then
    GrafKiir(G);

  { A 1. pontból kiinduló utakat }
  Dijkstra(G,T,C,1);

  { A 1. pontból a 4. pontba }
  UtKiir(G,T,C,1,4);
end.
```