Adatszerkezetek és algoritmusok - gyakorlati jegyzet

Kovács György

Debreceni Egyetem

Tartalomjegyzék

[1] A szerző email elérhetőségei: e-mail: gyuriofkovacs@gmail.com, url: http://www.inf.unideb.hu/~gykovacs

1. Előszó

A gyakorlataimon leadott anyagok kézirata, NEM LEKTORÁLT, HIBÁKAT TARTALMAZ, HASZNÁLATÁT SENKINEK SEM JAVASLOM!

2. A gyakorlat követelményei

A gyakorlat teljesítéséhez a félév során kettő zh legalább 61%-os teljesítése szükséges. A félév végén a két zh egyike javítható.

3. Irodalom

Az félév során tárgyalt algoritmusok zömének pszeudokódja megtalálható a Műszaki Könyvkiadónál megjelent Algoritmusok [1] című könyvben.

4. A C programozási nyelv

A C programozási nyelv elsajátítása nem része az adatszerkezetek és algoritmusok tárgy tematikájának. A C nyelv irodalma széles. A Magasszinű programozási nyelvek 1 linken található gyakorlati jegyzetben számos utalás és technikai részlet található a különböző adatszerkezetek C-ben történő megvalósítására vonatkozóan. A félév első felében statikus memóriakezeléssel ábrázolható adatszerkezetekkel foglalkozunk, így a C nyelv alapelemein túl a *kétirányú elágaztató utasítás*, az *előírt lépésszámú ciklus*, a *függvények* és *tömbök* használatának ismerete elegendő az adatszerkezetek és algoritmusok anyag elsajátításához.

5. A tömb

A tömb definíciója: homogén, statikus adatszerkezet.

- A homogenitás arra utal, hogy azonos típusú adatelemekből épül fel,
- a statikusság pedig azt jelenti, hogy minden tömb mérete rögzített, nem változtatható.

A tömb elemeinek elérése közvetlen, minden elem hivatkozható az indexével. A C programozási nyelvben `N` méretű, `tipus` típusú, `azonosito` nevű tömböt az alábbi deklarációs utasítással hozhatunk létre.

```
tipus azonosito[N];
```

A tömb i. elemét szintén a szögletes zárójelek segítségével hivatkozhatjuk:

```
azonosito[i];
```

Egy tömbnek kezdőértéket kapcsoszárójelek között felsorolt kezdőérték sorozattal adhatunk:

```
tipus azonosito[N]= {konst_kif1, konst_kif2, ..., konst_kifN};
```

Lássunk egy egyszerű példát: az alábbi kódban létrehozunk egy egészeket (`int`) tartalmazó 5 elemű tömböt (4,6,8,10,12) kezdőértékkel, és kiírom a konzolra a tömbben tárolt értékeket:

```
#include <stdio.h>

int main(int argc, char** argv)
{
  int t[5]= {4, 6, 8, 10, 12};
  int i;

  for ( i= 0; i < 5; ++i )
    printf("%d ", t[i]);

  return 0;
}
```

Fontos megjegyeznünk, hogy a tömbök indexelése 0-val kezdődik! FONTOS

Tömbökkel kapcsolatban a két legfontosabb algoritmuscsalád a kereső és rendező algoritmusok családja. A továbbiakban ezek néhány elemét vesszük sorra.

5.1. Teljes keresés

Legyen adott t tömb és egy, a tömbben tárolt értékek típusával megegyező típusú k érték. A keresés célja annak eldöntése, hogy a k érték szerepel-e a tömbben. Azon kívül, hogy szerepel-e, gyakran hasznos információ még az is, hogy hol szerepel a keresett elem a tömbben, vagyis mi az elem indexe. Mivel az eljárásorientált programozás lényege, hogy programunk viszonylag rövid, egyszerű függvényekből és eljárásokból épüljön fel, a továbbiakban az egyes adatszerkezeteket kezelő algoritmusokat függvények formájában írjuk meg. Ennek fényében tehát egy kereső algoritmus megvalósításához olyan függvényt kell írnunk, amely paraméterként kap egy tömböt, valamint egy értéket, és visszaadja ezen érték indexét ha az szerepel a tömbben, és valamilyen speciális értéket, ha nem szerepel a tömbben. Ez a speciális érték többnyire `-1`, mivel a `-1`-et nem használjuk tömb indexelésére, ha a függvény visszatérése `-1`, abból lehet tudni a hívási környezetben, hogy a keresett érték nincs benne a tömbben.

Teljes keresés esetén nincs semmilyen feltételezésünk a tömbről, a teljes keresés tehát általánosan használható, a tömb minden elemét sorra vesszük és megvizsgáljuk, hogy megegyezik-e a keresett értékkel. Ha igen, visszaadjuk annak indexét, ha nem, a következő elemet vizsgáljuk, egészen addig, amíg el nem értük a tömb végét. Ha elértük a tömb végét, biztos, hogy a keresett elem nincs benne a tömbben, ezért a `-1` értéket adjuk vissza. Az alábbiakban egy egészeket tartalmazó tömbben történő teljes keresést implementálunk.

```
int teljesKereses(int t[], int n, int k)
{
  int i;
  for ( i= 0; i < n; ++i )
    if ( t[i] == k )
       return i;
  return -1;
}
```

A fenti függvényben a `t` paraméter a tömb, amelyben keresünk, `n` a tömb mérete, ugyanis egy tömb méretét mindig át kell adnunk paraméterként és `k` a keresett egész típusú érték. A fenti függvény nagyon könnyen átírható `float` típusú tömbre:

```
int teljesKereses(float t[], int n, float k)
{
  int i;
  for ( i= 0; i < n; ++i )
```

```
    if ( t[i] == k )
      return i;
  return -1;
}
```

Könnyen látható, hogy csak a paraméterként kapott tömb és a keresett érték típusát kell átírnunk a paraméterlistán. A visszatérési érték továbbra is `int` típusú, hiszen vagy egy indexet adunk vissza, vagy a -1 értéket!

A teljes keresés csak egy koncepció, a konkrét megvalósítás problémától függ. A fentiekben egészeket, majd lebegőpontos értékeket tartalmazó tömbökre implementáltuk a teljes keresés algoritmusát. A teljes keresés egy speciális esete, amikor nem egy konkrét értéket keresünk, hanem a tömbben szereplő legkisebb vagy legnagyobb értéket vagy annak helyét. Ebben az esetben természetesen nincs szükség a keresett k értékre, hiszen nem ismerjük.

```
int legkisebbErtek(int t[], int n)
{
  int min= t[0];
  int i;
  for ( i= 1; i < n; ++i )
    if ( min > t[i] )
      min= t[i];
  return min;
}
```

A `legkisebbErtek` függvény törzsében a `min` változóban tároljuk a feltételezett legkisebb értéket. Feltéve, hogy a tömb mérete legalább 1, kezdeti feltételezésünk, hogy a legkisebb elem éppen a tömb első eleme. Ezt követően a tömb többi elemét sorra összehasonlítjuk a feltételezett legkisebb értékkel, és ha kiderül, hogy a tömb valamely eleme kisebb, mint eddigi feltételezésünk (azaz teljesül az `if` feltétele), módosítjuk a feltételezésünket az újonnan talált, feltételezett legkisebb értékre. Vegyük észre, hogy a ciklus lefutása után a `min` változó biztosan a tömbben szereplő legkisebb értéket fogja tárolni! Nagyon fontos, hogy az inicializálást mindig a tömb valamely elemével kell végeznünk, hiszen ha a `min` változó kezdőértéke például 10 lenne és a tömb csak 100-tól nagyobb értékeket tárolna, akkor a visszatérési érték is 10 lenne (helytelenül), hiszen a tömb egyik elemére sem teljesülne, hogy kisebb, mint a `min` változó értéke.

Módosítsuk a fenti függvényt úgy, hogy a legkisebb érték helyét adja vissza!

```
int legkisebbErtekHelye(int t[], int n)
{
  int min= 0;
  int i;
  for ( i= 1; i < n; ++i )
    if ( t[min] > t[i] )
      min= i;
  return min;
}
```

Könnyű látni, hogy a különbség csupán az, hogy most a `min` változóban a legkisebb elem indexét tároljuk, az algoritmus koncepciója ugyanaz, kezdeti feltételezésünk, hogy a legkisebb elem a tömb első eleme (`int min= 0`). Mivel a `min` most egy értéket tárol, értelemszerűen módosítanunk kell azon utasításokat, amelyekben felhasználjuk a `min` értékét.

5.1. Feladat: Írjon függvényeket, amelyek visszaadják a paraméterként kapott egész típusú tömbben található legnagyobb értéket és annak helyét!

5.1. Megoldás: A fentiek alapján:

```
int legnagyobbErtek(int t[], int n)
{
  int max= t[0];
  int i;
  for ( i= 1; i < n; ++i )
    if ( max < t[i] )
      max= t[i];
  return max;
}


int legnagyobbErtekHelye(int t[], int n)
{
  int max= 0;
  int i;
  for ( i= 1; i < n; ++i )
    if ( t[max] < t[i] )
      max= i;
  return max;
}
```

5.2. Feladat: Írjon főprogramot az előző függvények felhasználásával, amely a konzolról egészekkel feltölt egy 10 elemű tömböt, majd kiírja a tömbben található legkisebb és legnagyobb elemeket, azok helyét. Ezt követően beolvas egy egész számot a konzolról és ha szerepel a tömbben, kiírja annak indexét, ha nem szerepel, a tömbben, akkor `-1`-et!

5.2. Megoldás: A teljesség kedvéért az alábbiakban még leírjuk az egész főprogramot, amely fordítható és futtatható, a későbbiekben hasonló esetben csak magát a `main` függvényt fogjuk megírni:

5.1.1. forráskód: teljeskereses.c

```
#include <stdio.h>

int teljesKereses(int t[], int n, int k)
{
  int i;
  for ( i= 0; i < n; ++i )
    if ( t[i] == k )
       return i;
  return -1;
}

int legkisebbErtek(int t[], int n)
{
  int min= t[0];
  int i;
  for ( i= 1; i < n; ++i )
    if ( min > t[i] )
      min= t[i];
  return min;
}

int legkisebbErtekHelye(int t[], int n)
{
  int min= 0;
  int i;
  for ( i= 1; i < n; ++i )
    if ( t[min] > t[i] )
      min= i;
  return min;
}
```

```
int legnagyobbErtek(int t[], int n)
{
  int max= t[0];
  int i;
  for ( i= 1; i < n; ++i )
    if ( max < t[i] )
      max= t[i];
  return max;
}

int legnagyobbErtekHelye(int t[], int n)
{
  int max= 0;
  int i;
  for ( i= 1; i < n; ++i )
    if ( t[max] < t[i] )
      max= i;
  return max;
}

int main(int argc, char** argv)
{
  int t[10], i, k;

  printf("adjon meg 10 egesz szamot:\n");
  for ( i= 0; i < 10; ++i )
    scanf("%d", &(t[i]));

  printf("legkisebb ertek: %d\n", legkisebbErtek(t, 10));
  printf("legkisebb ertek helye: %d\n", legkisebbErtekHelye(t, 10));
  printf("legnagyobb ertek: %d\n", legnagyobbErtek(t, 10));
  printf("legnagyobb ertek helye: %d\n", legnagyobbErtekHelye(t, 10));

  printf("adjon meg egy keresendo erteket:\n");
  scanf("%d", &k);
  printf("%d\n", teljesKereses(t, 10, k));

  return 0;
}
```

A fenti program kimenete a (3,2,5,4,7,0,9,-1,6,6) értékek begépelése, valamint a keresendő 6 érték esetén:

```
adjon meg 10 egesz szamot:
3 2 5 4 7 0 9 -1 6 6
legkisebb ertek: -1
legkisebb ertek helye: 7
legnagyobb ertek: 9
legnagyobb ertek helye: 6
adjon meg egy keresendo erteket:
6
8
```

5.2. *Lineáris keresés*

FONTOS

A lineáris keresés funkcióját tekintve megegyezik a teljes kereséssel, azaz a keresett elem indexét adja vissza ha az szerepel a tömbben, és -1-et, ha nem szerepel benne. A különbség az a **feltételezés, hogy a tömb rendezett**, azaz egész számok esetén azokat növekvő sorrendben tartalmazza. Ekkor ugyanis ha például a 10 értéket keressük a növekvő sorrendbe rendezett tömbben és az elejéről, azaz a legkisebb értéktől indulva elérünk egy olyan elemig, amely nagyobb, mint 10, abbahagyhatjuk a keresést, hiszen a rendezettség miatt biztos, hogy a tömb hátra lévő részében már csak 10-től nagyobb értékek szerepelnek.

```
int linearisKereses(int t[], int n, int k)
```

```
{
  int i;

  for ( i= 0; i < n; ++i )
    if ( t[i] >= k )
      break;

  if ( i == n )
    return -1;
  if ( t[i] == k )
    return i;

  return -1;
}
```

A fenti formájában a lineáris keresés az algoritmus szó szerinti megvalósítása. Elindulunk egy ciklussal és addig megyünk, amíg el nem éri a ciklusváltozó (`i`) a tömb végét vagy a keresett értéktől nagyobb vagy azzal egyenlő számot találunk a tömbben, ekkor ugyanis a `break` utasítással megszakítjuk a futást. Ezt követően `if` utasításokkal döntjük el, hogy pontosan melyik szituáció állt elő:

- ha az `i` ciklusváltozó értéke `n`, akkor a tömb végére értünk anélkül, hogy teljesült volna az `if` feltétele, így biztos, hogy nincs a keresett érték a tömbben, tehát a visszatérési érték -1;
- ha a vezérlés ezen az elágaztató utasításon túl lép, akkor azt ellenőrizzük, hogy a keresett értéken álltunk-e meg, azaz a `for` ciklus magjában található >= feltételből pontosan az egyenlőség teljesült-e. Ha igen, visszaadjuk az i értékét;
- ha nem értük el a tömb végét és az egyenlőség sem teljesült a ciklus magjában, akkor csak a > reláció teljesülhetett, azaz nagyobb értéket találtunk a keresettnél, de azt nem, így a visszatérési érték -1. Itt nincs szükség újabb `if`-re, hiszen ha a harmadik `return` utasításhoz ér a vezérlés, csak ez a szituáció állhat fenn.

Fontos, hogy annak vizsgálata, hogy az `i` változó elérte-e a tömb végét megelőzze annak vizsgálatát, hogy a tömb `i`. eleme megegyezik-e a keresett elemmel. **Ha a két feltétel vizsgálatot fordított sorrendben írnánk, futási hibához jutnánk** abban az esetben, amikor az `i` eléri a tömb végét, ugyanis a ciklus lefutása után az `i` változó értéke `n` lenne, azonban egy `n` méretű tömböt csak a 0,...,n-1 értékekkel lehet indexelni, így a `t[i] == k` feltétel vizsgálatában i=n esetén túlindexelés áll elő. Lássuk, hogyan írható a fenti algoritmus elegánsabb, de nem hatékonyabb formában:

FONTOS

```
int linearisKereses(int t[], int n, int k)
{
  int i;

  for ( i= 0; i < n && t[i] <= k; ++i )
    if ( t[i] == k )
      return i;
  return -1;
}
```

Az utóbbi kódban összevontuk a ciklus kilépési feltételeit. Vigyázni kell azonban, hogy az `i < n` vizsgálat a feltételben megelőzze a `t[i] <= k` vizsgálatot. Az indok az korábbihoz hasonlóan a túlindexelés elkerülése. Kihasználva hogy az `&&` operátor rövidzár operátor, i=n esetén a bal oldalán álló kifejezés hamis, így a jobb oldalán álló kifejezés nem kerül kiértékelésre, vagyis a túlindexelés nem áll elő.

Az algoritmus utóbbi formája elegánsabb ugyan, mert rövidebb, de nem hatékonyabb, ugyanis a legtöbb ciklusléptésben három feltétel ellenőrzését végzi, míg a korábbi implementáció csak két feltételt ellenőriz ciklusléptésenként.

5.3. Feladat: Módosítsa az egészek tömbjén lineáris keresést végző függvényt úgy, hogy kiírja a kimenetre, hányadik lépésben (azaz hányadik hasonlítás után) képes döntést hozni!

5.3. Megoldás: A lépések száma a ciklusváltozó értékéből könnyen kiszámítható, ugyanis a ciklusváltozó értkétől éppen egyel több hasonlítást végzünk (a +1 hasonlítás a 0-val kezdődő indexelésből adódik).

```
int linearisKereses(int t[], int n, int k)
{
  int i;

  for ( i= 0; i < n; && t[i] <= k; ++i )
    if ( t[i] == k )
    {
      printf("%d. lepesben megtalaltam\n", i + 1);
      return i;
    }
  printf("%d lepes utan biztosan nincs benne\n", i + 1);
  return -1;
}
```

5.3. Bináris keresés

FONTOS Bináris keresés esetén szintén azzal a **feltételezéssel élünk, hogy a tömb rendezett**. A bináris keresés egy iteratív keresés, amelyben mindig egy indextartomány középső elemét vizsgáljuk. A kiindulási tartomány a tömb teljes indextartománya. Ezt követően ha a tömb középső eleme a keresett elem, akkor visszaadjuk azt, ha nem az, akkor kihasználva hogy rendezett a tömb:

- ha a keresett elem nagyobb, mint a középső eleme a vizsgált tartományak, akkor a tartomány alsó határát a középső elem utáni elemre állítjuk, hiszen ekkor a keresett elem a tartomány felső felében kell, hogy szerepeljen,
- ha a keresett elem kisebb, mint a középső elem, akkor a vizsgált tartomány felső határát a középső elem előtti elemre állítjuk, hiszen a keresett elem a tartomány alsó felében kell, hogy szerepeljen.

A tartomány módosítása után újra vizsgáljuk a középső elemet. Mindezt iteratívan addig folytatjuk, amíg meg nem találjuk a keresett elemet, vagy az alsó határ túl nem halad a felső határon, azaz a keresési tartomány 0 eleműre nem szűkül.

Könnyű látni, hogy minden iteratív lépésben a vizsgált tartomány hossza feleződik, ami azt jelenti, hogy n elemű tömb esetén $log_2 n$ lépésben 1 elemű tartományhoz, azaz egy elemhez jutunk, amelyről rögtön eldönthető, hogy a keresett elem-e vagy sem. A bináris keresés a leggyorsabb keresés, ami rendezett tömbökön végrehajtható.

A bináris keresés C kódja:

```
int binarisKereses(int t[], int n, int k)
{
  int alsoh= 0, felsoh= n - 1, kozep;
  for ( kozep= (alsoh + felsoh) / 2; alsoh <= felsoh; kozep= (alsoh + felsoh) / 2 )
  {
    if ( t[kozep] == k )
       return kozep;
    if ( t[kozep] < k )
       alsoh= kozep + 1;
    else
       felsoh= kozep - 1;
```

```
  }

  return -1;
}
```

5.4. Feladat: Módosítsa az egészek tömbjén bináris keresést végző függvényt úgy, hogy kiírja a kimenetre, hányadik lépésben (azaz hányadik hasonlítás után) képes döntést hozni!

5.4. Megoldás: Mivel itt nem a szokványosnak mondható módon használjuk a ciklusváltozót, be kell vezetnünk egy új változót, amely a ciklus lépéseit számolja.

```
int binarisKereses(int t[], int n, int k)
{
  int alsoh= 0, felsoh= n - 1, kozep, i= 0;
  for ( kozep= (alsoh + felsoh) / 2; alsoh <= felsoh; kozep= (alsoh + felsoh) / 2 )
  {
    ++i;
    if ( t[kozep] == k )
    {
      printf("%d. lepesben megtalaltam\n", i );
      return kozep;
    }
    if ( t[kozep] < k )
      alsoh= kozep + 1;
    else
      felsoh= kozep - 1;
  }

  printf("%d lepes utan biztosan nincs benne\n", i );
  return -1;
}
```

5.5. Feladat: Módosítsuk a fenti kereső függvényeket úgy, hogy azoknak egy `kezdet` és `veg` nevű paraméterrel meg lehessen adni azt az indextartományt, amelyben keresniük kell! (A `veg` paraméter azt az utolsó indexet kapja értékül, amelyet még ellenőrizni kell.)

5.5. Megoldás:

```
int teljesKereses(int t[], int k, int kezdet, int veg)
{
  int i;
  for ( i= kezdet; i <= veg; ++i )
    if ( t[i] == k )
       return i;
  return k;
}

int linearisKereses(int t[], int k, int kezdet, int veg)
{
  int i;

  for ( i= kezdet; i <= veg && t[i] <= k; ++i )
    if ( t[i] == k )
      return i;
  return -1;
}

int binarisKereses(int t[], int k, int kezdet, int veg)
{
  int alsoh= kezdet, felsoh= veg, kozep;
```

```
  for ( kozep= (alsoh + felsoh) / 2; alsoh <= felsoh; kozep= (alsoh + felsoh) / 2 )
  {
    if ( t[kozep] == k )
       return kozep;
    if ( t[kozep] < k )
       alsoh= kozep + 1;
    else
       felsoh= kozep - 1;
  }

  return -1;
}
```

5.6. Feladat: Adott a rendre (1, 2, 2, 5, 6, 8, 10, 12, 12) értékeket tartalmazó tömb. Állapítsa meg, hogy a bináris keresés hányadik lépésben találja meg a

- 2 elemet,
- 5 elemet,
- 1 elemet,
- hányadik lépésben áll meg, ha a 0 elemet kell megkeresnie?

5.6. Megoldás:

- 2: 2. lépésben,
- 5: 4. lépésben,
- 1: 3. lépésben,
- 0: 4. lépésben áll meg.

5.4. Szélsőérték kiválasztásos rendezés

Az lineáris keresés és bináris keresés algoritmusokban azzal a feltételezéssel éltünk, hogy a tömb, amelyben keresünk, rendezett. Most azt nézzük meg, hogy egy rendezetlen tömböt hogyan tudunk gyorsan rendezni. Az egyik legegyszerűbb keresés a szélsőérték kiválasztásos rendezés. Ennek koncepciója a következő:

- tegyük fel, hogy a tömb elemeit növekvő sorrendbe szeretnénk rendezni.
- Ekkor ha kiválasztjuk a tömb legkisebb elemét, majd megcseréljük azt az első elemmel, a tömb első eleme biztosan jó helyen lesz, ugyanis növekvő sorrendbe rendezett tömb esetén az első elem a legkisebb elem.
- Ezt követően tekintsük az eredeti tömbnek azon résztömbjét, amelyik az eredeti tömb második (vagyis 1 indexű) elemével kezdődik és a tömb végéig tart.
- Ha erre megismételjük az előző lépést, azaz megkeressük a legkisebb elemét (ami a teljes tömb második legkisebb eleme) és megcseréljük azt a résztömb első elemével (ami a teljes tömbnek a második eleme), akkor biztosak lehetünk benne, hogy a teljes tömb első két eleme már jó helyen van, hiszen az első elem az egész tömb legkisebb eleme, a második eleme pedig az teljes tömb második legkisebb eleme.
- A fenti lépéseket ismételve a tömb végéig, a tömb rendezetté válik.

A minimum kiválasztásos növekvő rendezés C kódja:

```
void minimumKivalasztasosNovekvoRendezes(int t[], int n)
{
  int min, i, j, tmp;

  for ( i= 0; i < n; ++i )
  {
```

```
    min= i;

    for ( j = i; j < n; ++j )
      if ( t[j] < t[min] )
        min= j;

    tmp= t[min];
    t[min]= t[i];
    t[i]= tmp;
  }
}
```

A függvény `void` visszatérési típusa azt jelenti, hogy nincs visszatérési értéke, azaz lényegében eljárást írunk. Ekkor nincs szükség a `return` utasításra sem, azaz nincs visszatérési érték. Ennek azaz oka, hogy a rendezés minden esetben kivitelezhető, nincs információ, amelyet vissza szeretnénk juttatni a hívási környezetbe. Négy változót használunk, a `min` változót a legkisebb elem keresésénél használjuk fel, az `i` és `j` ciklusváltozók, míg a `tmp` változót két változó értékének cseréjére használjuk. A minimumkiválasztásos rendezés koncepcionális leírásánál olvasható, hogy a legkisebb értéket meg kell cserélnünk a vizsgált tömb első elemével és ez a legegyszerűbben egy segédváltozó segítségével oldható meg.

A belső `for`-ciklus az azt megelőző `min= i` értékadással egy tömb legkisebb eleme helyének (indexének) meghatározása. (Hasonlítsuk össze ezt a kódot a korábban megírt legkisebb helyet meghatározó függvény kódjával!) Figyeljük meg, hogy itt a minimum hely keresése nem a tömb elejéről indul, hanem a tömb `i`. elemétől, amely egy külső ciklusban folyamatosan nő.

Az külső ciklus első lépésében tehát `i= 0` esetén a belső ciklus a teljes tömb legkisebb elemének helyét határozza meg, melynek indexe a `min` változóba kerül. Ezt követően három utasítással megcseréljük a tömb első, tehát `i=0` indexű, és legkisebb, azaz `min` indexű elemét. A második lépésben a legkisebb érték keresése a tömbben már `i=1`-től indul, azaz az eredeti tömb második elemétől kezdődően keressük a tömb legkisebb elemét, amely így az egész tömb második legkisebb eleme lesz. Ezen második legkisebb elemet aztán már az `i=1` indexű elemmel, azaz a teljes tömb második elemével cseréljük meg. Ekkor a tömb első két eleme már rendezett. Könnyű látni, hogy a külső ciklus folytatásával az egész tömb rendezetté válik.

Az alábbi példában egy 5 elemű tömb rendezése során a tömbnek a külső ciklus lépései közötti állapota látható (félkövérrel szedtük a már biztosan jó helyen lévő elemeket):

elem indexe	0.	1.	2.	3.	4.
rendezetlen tömb:	3	5	0	6	3
1. lépés után	**0**	5	3	6	3
2. lépés után	**0**	**3**	5	6	3
3. lépés után	**0**	**3**	**3**	6	5
4. lépés után	**0**	**3**	**3**	**5**	**6**

Lássuk, hogyan tehetjük absztraktabbá az algoritmusunkat, a legkisebb érték helyének keresését függvényként kiemelve:

```
int legkisebbErtekHelyeKezdettel(int t[], int n, int kezdet)
{
  int i, min= kezdet;

  for ( i= kezdet + 1; i < n; ++i )
    if ( t[i] < t[min] )
      min= i;
```

```
  return min;
}

void minimumKivalasztasosNovekvoRendezes2(int t[], int n)
{
  int min, i, j, tmp;

  for ( i= 0; i < n; ++i )
  {
    min= legkisebbErtekHelye(t, n, i);

    tmp= t[min];
    t[min]= t[i];
    t[i]= tmp;
  }
}
```

A fenti `legkisebbErtekHelye` függvény első paramétere továbbra is a tömb, amelyben keresünk, második paramétere a tömb mérete, harmadik paramétere pedig azon index, amelytől a keresést kezdeni kell.

5.7. Feladat: Módosítsa úgy a minimum kiválasztásos növekvően rendező algoritmust, hogy az maximum kiválasztással csökkenő sorrendbe rendezze a paraméterként kapott tömböt!

5.7. Megoldás:

```
void maximumKivalasztasosCsokkenoRendezes(int t[], int n)
{
  int max, i, j, tmp;

  for ( i= 0; i < n; ++i )
  {
    max= i;

    for ( j = i; j < n; ++j )
      if ( t[j] > t[max] )
        max= j;

    tmp= t[max];
    t[max]= t[i];
    t[i]= tmp;
  }
}
```

5.8. Feladat: Módosítsa úgy a minimum kiválasztásos növekvően rendező algoritmust, hogy az minimum kiválasztással csökkenő sorrendbe rendezze a paraméterként kapott tömböt!

5.8. Megoldás:

```
void minimumKivalasztasosCsokkenoRendezes(int t[], int n)
{
  int min, i, j, tmp;

  for ( i= n-1; i > 0; ++i )
  {
    min= i;

    for ( j = 0; j < i; ++j )
      if ( t[j] < t[min] )
        min= j;
```

```
    tmp= t[min];
    t[min]= t[i];
    t[i]= tmp;
  }
}
```

5.9. Feladat: Módosítsa úgy a minimum kiválasztásos növekvően rendező algoritmust, hogy az maximum kiválasztással növekvő sorrendbe rendezze a paraméterként kapott tömböt!

5.9. Megoldás:

```
void maximumKivalasztasosNovekvoRendezes(int t[], int n)
{
  int max, i, j, tmp;

  for ( i= n-1; i > 0; ++i )
  {
    max= i;

    for ( j = 0; j < i; ++j )
      if ( t[j] > t[max] )
        max= j;

    tmp= t[max];
    t[max]= t[i];
    t[i]= tmp;
  }
}
```

5.5. Buborék rendezés

A buborék rendezés célja ugyanaz, mint a szélsőérték kiválasztásos rendezésé, koncepciójában különbözik csak:

- Tegyük fel, hogy növekvő sorrendben szeretnénk rendezni a tömböt.
- Induljunk a tömb végéről és hasonlítsuk össze a tömb utolsó elemét az azt megelőzővel, ha az utolsó elem (n.) kisebb, mint az azt megelőző (n-1.), akkor a rendezett tömbben biztos, hogy az (n.) elemnek valahol az (n-1.) elem előtt kell szerepelnie, ezért cseréljük meg őket, hogy egymáshoz képes legalább jó sorrendben legyenek. Ismételjük meg ezt a lépést az utolsó előtti elemre és az azt megelőzőre, és így tovább, egészen a második elemig és az azt megelőző első elemig.
- Az előző műveletsor eredményeként a tömb legkisebb eleme biztosan a tömb első helyén fog szerepelni.
- Ismételjük meg a fenti műveletsort, ismét a tömb n. elemétől kezdve. Könnyű látni, hogy ez a tömb második legkisebb elemét a mádosik helyre fogja juttatni.
- Ismételjük meg a fenti műveletsort még n-3 alkalommal. Mivel minden ismétlés egy újabb elemet tesz a helyére, n-1 ismétlés után az egész tömb rendezett lesz.

A gyakorlatban a második pontban leírt műveletsort minden alkalommal a tömb utolsó eleménél kezdjük, azonban nem kell mindig a tömb első eleméig elmenni vele, hiszen a tömb elején lévő már rendezett elemek sorrendje már nem változhat, nem teljesülhet az "i. kisebb, mint az i-1." feltétel.

A buborék rendezés C kódja (ügyeljünk arra, hogy a tömb első elemének indexe 0, a tömb utolsó elemének indexe n-1):

```
void novekvoBuborekRendezes(int t[], int n)
{
  int i, j, tmp;

  for ( i= 0; i < n; ++i )
    for ( j = n - 1; j > i; --j )
      if ( t[j] < t[j-1] )
      {
        tmp= t[j-1];
        t[j-1]= t[j];
        t[j]= tmp;
      }
}
```

A külső ciklus ciklusváltozója (i) vezérli azt, hogy a páronkénti cserét a tömb végétől indulva hányadik (i.) elemig kell végrehajtani. A belső ciklus pedig a tömb végétől indulva az i. elemig a szomszédos elemek relatív sorrendjét a növekvő rendezésnek megfelelő módon állítja. Az alábbiakban a külső ciklus lépései után a tömb állapotát mutatjuk be (félkövérrel szedtük az egyes lépésekben már biztosan jó helyen lévő elemeket):

elem indexe	0.	1.	2.	3.	4.
rendezetlen tömb:	3	5	0	6	3
1. lépés után	**0**	3	5	3	6
2. lépés után	**0**	**3**	3	5	6
3. lépés után	**0**	**3**	**3**	5	6
4. lépés után	**0**	**3**	**3**	**5**	**6**

A fenti példában látható, hogy a tömb már a 2. lépésben teljesen rendezetté vált. Az algoritmus hatékonyabbá tehető, ha figyeljük, hogy változik-e a tömbünk, és ha nem, akkor abbahagyjuk a rendező algoritmust, hiszen már rendezett. Ehhez létrehozunk egy segédváltozót (`f`), melynek értékét minden külső ciklus lépés elején 0-ra állítjuk, ha érték csere történik, akkor pedig 1-re. Ha a belső ciklus lefutása után az `f` értéke még mindig 0, azaz nem történt érték csere, az azt jelenti, hogy a tömb rendezett, függetlenül attól, hogy hányadik lépésben járunk az `i` változóval, azaz abbahagyhatjuk a rendezést és befejezhetjük az eljárás futását a paraméter nélküli `return` utasítással:

```
void novekvoBuborekRendezes2(int t[], int n)
{
  int i, j, tmp, f;

  for ( i= 0; i < n; ++i )
  {
    f= 0;
    for ( j = n - 1; j > i; --j )
      if ( t[j] < t[j-1] )
      {
        f= 1;
        tmp= t[j-1];
        t[j-1]= t[j];
        t[j]= tmp;
      }
    if ( f == 0 )
      return;
  }
}
```

5.10. Feladat: Módosítsa a növekvő buborék rendezést csökkenő buborék rendezésre!

5.10. Megoldás:

```
void csokkenoBuborekRendezes(int t[], int n)
{
  int i, j, tmp;

  for ( i= 0; i < n; ++i )
    for ( j = n - 1; j > i; --j )
      if ( t[j] > t[j-1] )
      {
        tmp= t[j-1];
        t[j-1]= t[j];
        t[j]= tmp;
      }
}
```

5.11. Feladat: Módosítsa a növekvő buborék rendezést oly módon, hogy a belső ciklus a tömb elejéről induljon és a feltételben az i. elemet az i+1. elemhez hasonlítsa!

5.11. Megoldás:

```
void novekvoBuborekRendezes(int t[], int n)
{
  int i, j, tmp;

  for ( i= n-1; i >= 0; --i )
    for ( j = 1; j <= i; ++j )
      if ( t[j] < t[j-1] )
      {
        tmp= t[j-1];
        t[j-1]= t[j];
        t[j]= tmp;
      }
}
```

5.12. Feladat: Írja meg a csökkenő buborék rendezést oly módon, hogy a belső ciklus a tömb elejéről induljon és a feltételben az i. elemet az i+1. elemhez hasonlítsa!

5.12. Megoldás:

```
void csokkenoBuborekRendezes(int t[], int n)
{
  int i, j, tmp;

  for ( i= n-1; i >= 0; --i )
    for ( j = 1; j <= i; ++j )
      if ( t[j] > t[j-1] )
      {
        tmp= t[j-1];
        t[j-1]= t[j];
        t[j]= tmp;
      }
}
```

5.6. Gyorsrendezés

A gyorsrendezés koncepciója hasonló a bináris keresés koncepciójához, azaz a rendezendő elemeket megfelezi, majd a rendezést a két fél tartományon hajtja végre:

- Első lépésként jelöljünk ki egy elemet (x), amelyet a megfelelő helyre szeretnénk rakni.

- Tegyük ezt az elemet a tömb végére (cseréljük meg az utolsó elemmel) és inicializáljuk az $aktualis$ változó értékét 0-ra.
- Induljunk el a tömb elejéről és a tömb utolsó előtti eleméig ha valamely elemre (y) teljesül, hogy kisebb, mint a korábban kijelölt x elem, akkor ezen y elemet cseréljük meg a tömb $aktualis$ sorszámú értékét az y értékkel, majd növeljük az $aktualis$ valtozó értékét 1-gyel.
- Könnyű látni, hogy a fenti műveletsor elvégzése után az $aktualis$ változó értéke éppen egyel több lesz, mint ahány tömb elem kisebb volt a kijelölt x elemnél.
- Ekkor ha a tömb utolsó elemét (x) megcseréljük az $aktualis$ változóban tárolt indexű elemmel, akkor éppen teljesülni fog, hogy az x elemtől balra lévő elemek minde kisebbek, mint x, a tőle jobbra lévő elemek mind nagyobbak, mint x.
- Ezt követően végezzük el az eddigi lépéseket (beleértve ezt is) az x-et megelőző résztömbre és az x-et követő résztömbre, ha az több, mint 1 elemből áll.
- Könnyű látni, hogy a fenti algoritmus végrehajtása után a tömb rendezetté válik.

A gyorsrendezést C-ben két függvényben valósítjuk meg, ez ugyanis elegánsabb és rövidebb, mintha egy függvényt használnák:

```
int particionalas(int t[], int kezdet, int veg, int kijelolt)
{
  int aktualis, tmp, i;
  int x= t[kijelolt];

  t[kijelolt]= t[veg-1];
  t[veg-1]= x;

  aktualis= kezdet;

  for ( i= kezdet; i < veg-1; ++i )
    if ( t[i] < x )
    {
      tmp= t[i];
      t[i]= t[aktualis];
      t[aktualis]= tmp;
      ++aktualis;
    }

  t[veg-1]= t[aktualis];
  t[aktualis]= x;

  return aktualis;
}

void gyorsRendezesKezdetEsVeg(int t[], int kezdet, int veg)
{
  int aktualis;
  if ( veg > kezdet )
  {
    aktualis= particionalas(t, kezdet, veg, (kezdet + veg)/2);
    gyorsRendezesKezdetEsVeg(t, kezdet, aktualis);
    gyorsRendezesKezdetEsVeg(t, aktualis + 1, veg);
  }
}
```

A `particionalas` függvény végzi az tömb elemeinek két csoportra bontását. A függvény végén a `kijelolt` változóban tárolt indexű elemtől balra a tőle kisebb elemek, jobbra a tőle nagyobb elemek szerpelnek majd a tömbben. Visszatérési értéke `kijelolt` indexű elem új helyének indexe a tömbben. A `gyorsRendezesKezdetEsVeg` függvényt *rekurzív*nak nevezzük, hiszen önmagát hívja különböző paraméterekkel. Abban az esetben, ha a rendezendő résztömb utolsó elemének indexe nagyobb, mint az első elemének indexe, azaz a résztömb legalább egy elemű (a `veg` változó mindig a résztömb (utolsó elemének indexe + 1) értéket tartalmaz), akkor végrehajtjuk a particionálást, melynek

visszatérési értéke, azaz az `aktualis` változó értéke azon elemnek az indexe, amely biztosan a helyén van. Mivel tudjuk, hogy tőle balra csak tőle kisebb elemek, tőle jobbra csak tőle nagyobb elemek vannak, meghívjuk újra a `gyorsRendezesKezdetEsVeg` függvényt, úgy paraméterezve, hogy az a tömb kezdetétől a jó helyen lévő elemig, vagy a jó helyen lévő elemet követő elemtől a tömb végéig fusson le. Könnyű látni, hogy a rekurzív hívások során a tömb rendezetté válik.

```
int t[10]= {4, 2, 6, 1, 7, 3, 8, 9, 0, 2};
gyorsRendezesKezdetEsVeg(t, 0, 10);
```

kódrészlet a függvény alkalmazását szemlélteti Ahhoz, hogy könnyebbé tegyük a használatát, írhatunk egy egyszerű burkoló (ún. *wrapper*) függvényt, amely elfedi előlünk a rekurzív függvény 0 paraméterének megadását:

```
void gyorsRendezes(int t[], int n)
{
  gyorsRendezesKezdetEsVeg(t, 0, n);
}
```

Az alábbi példában a korábban vizsgált példán keresztül mutatom be az algoritmus lépéseit az egyes particionálási lépések után (félkövérrel szedtük az egyes lépésekben már biztosan jó helyen lévő elemeket):

elem indexe	0.	1.	2.	3.	4.	
rendezetlen tömb:	3	5	0	6	3	(kijelölt elem: 0)
1. lépés után	**0**	5	3	6	3	(a 0 van a helyén, kijelölt elem: 6)
2. lépés után	**0**	5	3	3	**6**	(a 6 a helyére került, kijelölt elem: 3 (az 1. 3-as)
3. lépés után	**0**	3	**3**	5	**6**	(a második 3-as került a helyére, a kijelölt elemek: 3, 5)
4. lépés után	**0**	**3**	**3**	**5**	**6**	(mivel az utoljára kijelölt résztömbök 1 eleműek, a rendezés véget ért)

5.13. Feladat: Írjon programot, amely a bemenetről beolvasva 10 egész számot, feltölt egy egészekből álló 10 elemű tömböt, rendezi azt növekvőleg minimumkiválasztásos rendezéssel, majd további számokat olvas be addig, amíg a -1 értéket nem olvassa be, és minden beolvasott szám után kiírja, hogy a szám benne van-e a tömbben, vagy nincs. A kereséshez használjon bináris keresést!

5.13. Megoldás:

5.6.2. forráskód: rendezkeres.c

```
#include <stdio.h>

void minimumKivalasztasosNovekvoRendezes(int t[], int n)
{
  int min, i, j, tmp;

  for ( i= 0; i < n; ++i )
  {
    min= i;

    for ( j = i; j < n; ++j )
      if ( t[j] < t[min] )
        min= j;

    tmp= t[min];
    t[min]= t[i];
    t[i]= tmp;
  }
}
```

```
int binarisKereses(int t[], int n, int k)
{
  int alsoh= 0, felsoh= n - 1, kozep;
  for ( kozep= (alsoh + felsoh) / 2; alsoh <= felsoh; kozep= (alsoh + felsoh) / 2 )
  {
    if ( t[kozep] == k )
       return kozep;
    if ( t[kozep] < k )
       alsoh= kozep + 1;
    else
       felsoh= kozep - 1;
  }

  return -1;
}

int main(int argc, char** argv)
{
  int t[10];
  int i;

  printf("adjon meg 10 szamot:\n");
  for ( i= 0; i < 10; ++i )
    scanf("%d",&(t[i]));

  minimumKivalasztasosNovekvoRendezes(t, 10);

  printf("adjon meg egy szamot:\n");
  scanf("%d", &i);
  while ( i != -1 )
  {
    if ( binarisKereses(t, 10, i) != -1 )
      printf("benne van\n");
    else
      printf("nincs benne\n");

    printf("adjon meg egy szamot:\n");
    scanf("%d", &i);
  }

  return 0;
}
```

A főprogramban (`main` függvényben) az `i` változót előbb ciklusváltozóként használjuk, majd az `i` változóba olvassuk be azokat az értékeket, amelyeket aztán a tömbben keresünk. A program kimenete a begépelt bemenet esetén:

```
adjon meg 10 szamot:
3 2 5 6 4 7 5 6 3 9
adjon meg egy szamot:
10
nincs benne
adjon meg egy szamot:
7
benne van
adjon meg egy szamot:
-1
```

6. Halmazok

A halmaz adatszerkezet a matematikai halmaznak megfelelő struktúra, amely tetszőleges típusú elemeket tartalmazhat. Legfontosabb tulajdonságga, hogy minden elemből csak egyet tartalmazhat. Ellentétben a tömbbel, a C nyelv nem tartalmaz beépített nyelvi eszközt halmazok létrehozására, ezért a meglévő eszközeinket kell felhasználnunk arra, hogy halmaz adatszerkezetet hozzunk létre.

A halmazt magát egy tömbbel fogjuk reprezentálni, a halmazhoz kapcsolódó műveleteket pedig függvényekkel. Egy halmazt a matematikában nagyon gyakran annak indikátorfüggvényével reprezentálunk. Az indikátorfüggvény olyan függvény, amely 1 értéket vesz fel azokon a pontokon, amelyek elemei egy halmaznak, és 0 értéket azokon a pontokon, amelyek nem elemei a halmaznak. A

$$x = \sum_{i \in \mathbb{Z}} i \mathbb{I}_{\{i : i \geq 0 \wedge i < 100 \wedge i | 3 = 0\}} \tag{1}$$

kifejezésben az $\mathbb{I}$ annak a halmaznak az indikátorfüggvényét jelöli, amely a 0-tól nagyobb és 10-tól kisebb, 3-mal osztható számokat tartalmazza. A fenti kifejezés szerint így x értéke ezen halmaz elemeinek összege. Hasonlóan folytonos esetben a

$$y = \int_{-\infty}^{\infty} x^2 \mathbb{I}_{\{i \geq 0 \wedge i < 1\}} \tag{2}$$

kifejezés szerint y értéke az `x^2` függvény integrálja a [0,1[halmazon.

Termszetesen csak a diszkrét esettel foglalkozunk, tehát véges számosságú halmazokkal. Vegyük észre, hogy a korábbi diszkrét kifejezésben az indikátor függvény feltétele két részből áll. Az első része egy intervallumot definiál, míg a második része egy oszthatósági feltételt fogalmaz meg. Az intervallum kiemelhető a feltételből:

$$x = \sum_{i \in \mathbb{Z}} i \mathbb{I}_{\{i : i \geq 0 \wedge i < 100 \wedge i | 3 = 0\}} = \sum_{i=0}^{99} i \mathbb{I}_{\{i : i | 3 = 0\}}, \tag{3}$$

hiszen amelyik egész szám nem eleme a $[0, 99]$ intervallumnak, arra az indikátor függvény értéke biztosan 0. Azt az intervallumot, amelyen egy indikátorfüggvény 1 értéket is felvehet, a továbbiakban **alaphalmaz**nak nevezzük. Tekintsük az alábbi kifejezést, **FONTOS**

$$x = \sum_{i=0}^{99} i \mathbb{I}_{\{i : i | 10 = 0\}}, \tag{4}$$

amely a [0,99] intervallumban 10-zel osztható egész számok összegét jelenti. A fenti kifejezésben a szumma után szereplő i változó önmagát jelenti. egy pici módosítással azonban a

$$x = \sum_{i=0}^{9} 10 i \mathbb{I}_{i : |true|}, \tag{5}$$

kifejezés ugyanazt jelenti, mint a korábbi, azonban az indikátor függvény már csak a [0,9] halmazon vesz fel 1 értéket. A két kifejezés eredménye megegyezik, azaz a `10i` függvénnyel a szumma után leképeztük az [0,99] intervallumbeli 10-zel osztható számok összegét adja. Ebből a példából jól látható, hogy egy egyszerű transzformációval, a halmaz tetszőleges eleméhez egy új érték rendelhető, így például a [0,9] halmazzal tetszőleges 10 elemű halmaz reprezentálható, ha találunk egy megfelelő transzformációt (függvényt), amely az $i \in [0, 9]$ értékhez meg tudja határozni a reprezentált halmaz valamely elemét.

A fentiek miatt (tehát mivel az egész számok egy N elemű halmazával tetszőleges azonos elemszámú, nem csak számhalmaz reprezentálható), a továbbiakban csak egészeket tartalmazó halmazokkal fogunk dolgozni.

FONTOS **Egy halmazhoz mindenekelőtt rögzítenünk kell az alaphalmazt, azaz azt a maximális elemszámot, amelyet kezelni fogunk. A továbbiakban egy halmazt az N elemű alaphalmaznak megfelelő N elemű tömb fog reprezentálni. Egy N elemű tömbbel a legkönnyebben a [0,N-1] egészeket tartalmazó alaphalmaz reprezentálható, ugyanis ha az alaphalmaz egy olyan A részhalmazát szeretnénk reprezentálni, amely az $i \in [0, N-1]$ értéket tartalmazza, akkor a tömb i. elemét, mint az alaphalmazon értelmezett indikátorfüggvény értékét az i helyen, 1-re állítjuk.**

Halmaz létrehozása. Legyen tehát az alaphalmaz az $N = 10$ elemű, egészeket tartalmazó $[0, N-1]$ intervallum. Ekkor egy üres halmazt (amely tehát az alaphalmaznak kell, hogy részhalmaza legyen), az alábbi módon hozhatunk létre.

```
int A[10]= {0};
```

A fenti kifejezés eredményeként létrejön egy 100 elemű tömb, amely minden eleme 0. Ez tehát azt jelenti, hogy az `A` halmaz indikátorfüggvénye az alaphalmaz minden elemén 0-t vesz fel, tehát a $[0, N-1]$ intervallum egyik eleme sincs benne az `A` halmazban. Nem üres halmazt értelemszerű kezdőértékadással hozhatunk létre:

```
int B[10]= {0, 0, 1, 0, 1, 0, 1, 0, 1, 0};
```

A `B` halmaz indikátor függvénye, azaz a `B` tömb 2., 4., 6. és 8. indexű eleme `1`, tehát a `B` halmaz szemléletesen a 2, 4, 6 és 8 számokat tartalmazza.

Tartalmazás vizsgálat. A kérdés, hogy egy adott halmaz esetén az k érték eleme-e a halmaznak? Ennek megválaszolására egy logikai függvényt írunk:

```
int eleme(int A[], int n, int k)
{
  return k < n && A[k];
}
```

A fenti függvény első paramétere egy tömb, tehát egy halmaz, második paramétere a tömb mérete, azaz az alaphalmaz számossága. A harmadik paraméter a vizsgált elem, amelyről el kell döntenünk, benne van-e a tömbben. A visszatérési érték logikai. Ha a `k < n` nem teljesül, akkor a keresett k elem nincs benne az alaphalmazban, így a visszatérési érték `0` lesz. Ha benne van, akkor kiértékelésre kerül az `A[k]` részkifejezés, amely éppen akkor `1`, tehát igaz, ha a `k` benne van a halmazban.

Halmazok metszete. Halmazok metszetét nagyon egyszerű eljárással számíthatjuk ki. Az eljárásnak paramétere lesz a két halmaz, amelyiknek a metszetét szeretnénk venni, azaz két egész típusú tömb, egy harmadik, az előzőekkel megegyező méretű halmaz (tömb), amelybe az eredményt kiszámítjuk, valamint az alaphalmaz (vagyis a tömbök) mérete.

```
void metszet(int A[], int B[], int C[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    C[i]= A[i] && B[i];
}
```

Könnyű látni, hogy a `C` eredmény tömbben éppen akkor lesz egy eleme értéke `1`, ha az mind az `A`, mind a `B` tömbben `1`, azaz mindkét tömbben benne van.

Halmazok uniója. Halmazok unióját az előzőhöz hasonló paraméterezésű egyszerű eljárással számíthatjuk ki:

```
void unio(int A[], int B[], int C[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    C[i]= A[i] || B[i];
}
```

A fenti megoldásban felhasználtuk, hogy ha az `A[i]` értéke vagy `B[i]` értéke nem nulla, azaz `1`, akkor a logikai "vagy" művelet értéke is `1`. Mindez azt jelenti, hogy ha az `A` vagy `B` tömb i. eleme `1`, akkor a `C` tömb i. eleme is `1` lesz.

Halmazok különbsége. Az alábbiakban az $C = A/B$ különbséget valósítjuk meg az előzőekhez hasonló paraméterezésű eljárással, azaz azon elemek lesznek benne a C halmazban (azoknak lesz `1` értéke a `C` tömbben), amelyek az A halmazban benne vannak, de a B halmazban nincsenek benne.

```
void kulonbseg(int A[], int B[], int C[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    C[i]= A[i] && !B[i];
}
```

Könnyű látni, hogy az `A[i] && !B[i]` kifejezés éppen akkor lesz igaz, azaz `1` értékű, ha az `A` tömb i. eleme `1`, és a `B` tömb i. eleme `0`.

Halmaz komplementere. A halmaz komplementerét számító eljárás csak három paramétert vár: a halmazt, amelynek komplementerét ki szeretnénk számítani, a halmazt, amelybe a komplementert ki szeretnénk számítani, valamint az alaphalmaz méretét.

```
void komplementer(int A[], int B[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    B[i]= !A[i];
}
```

A ciklus magjában egy negáció szerepel, melynek eredményeként a `B` tömb i. eleme éppen akkor lesz `1`, ha az `A` tömb i. eleme `0`, és fordítva.

Halmaz számossága. A halmaz számosságát megadó függvény még a fentiektől is egyszerűbb, paraméterként csak egy halmazt, azaz egy halmaz indikátorfüggvényét (tömb), valamint az alaphalmaz számosságát (tömb mérete) várja:

```
int szamossag(int A[], int n)
{
  int i, sum= 0;
  for ( i= 0; i < n; ++i )
    sum+= A[i];
  return sum;
}
```

Egyszerűen kiszámítjuk a tömbben lévő `1` értékek összegét majd visszaadjuk azt visszatérési értékként.

6.1. Feladat: Módosítsa a metszetet, uniót, különbséget és komplementert számító eljárásokat úgy, hogy a ciklusok magjában egy matematikai művelet szerepeljen a logikai kifejezés helyett, de az eljárások működése ne változzon!

6.1. Megoldás:

- Metszet: `A[i]*B[i]`,
- unió: `1 - (1 - A[i])*(1 - B[i])`,
- különbség: `A[i]*(A[i] - B[i])`,
- komplementer: `1 - A[i]`.

6.2. Feladat: Módosítsa a metszetet, uniót és különbséget számító eljárásokat úgy, hogy feltesszük, a tömbök mérete, tehát a halmazok alaphalmaza különböző, és az első paraméterként kapott tömb mérete a nagyobb, valamint az eredmény halmaz alaphalmaza (a harmadik tömb paraméter mérete) megegyezik az első paraméterként kapott tömb méretével.

6.2. Megoldás:

```
void metszet(int A[], int n, int B[], int m, int C[])
{
  int i;
  for ( i= 0; i < m; ++i )
    C[i]= A[i] && B[i];
  for ( ; i < n; ++i )
    C[i]= 0;
}

void unio(int A[], int n, int B[], int m, int C[])
{
  int i;
  for ( i= 0; i < m; ++i )
    C[i]= A[i] || B[i];
  for ( ; i < n; ++i )
    C[i]= A[i];
}

void kulonbseg(int A[], int n, int B[], int m, int C[])
{
  int i;
  for ( i= 0; i < m; ++i )
    C[i]= A[i] && !B[i];
  for ( ; i < n; ++i )
    C[i]= A[i];
}
```

A fenti kódokat kiegészítettük tehát egy újabb méret paraméterrel. A függvények törzsében található első ciklus minden esetben megegyezik a korábbi esetekkel, az egyetlen különbség, hogy a ciklusváltozó felső határa a kisebbik tömb mérete, ugyanis az alaphalmazoknak csak a közös részén értelmezhetőek a műveletek. A második ciklus minden esetben az "alapértelmezéssel" tölti fel az eredménytömböt, azaz metszet esetén természetesen az eredmény halmazba nem kerülnek be azok az elemek, amelyek csak az egyik halmazban vannak benne, unió esetén ha benne vannak a nagyobb halmazban, akkor bekerülnek az eredményhalmazba is, különbség esetén hasonlóan.

6.3. Feladat: Írjon függvényt, amely kiszámítja a paramterként kapott két darab, azonos méretű halmaz szimmetrikus differenciáját!

6.3. Megoldás: A szimmetrikus differencia egy szám, amely két halmaz különbözőségét jellemzi. Definíciója

$$A \ominus B = |A \backslash B| + |B \backslash A|, \tag{6}$$

tehát egy elem akkor számítandó bele az A és B halmaz szimmetrikus differenciájába, ha benne van az A halmazban, de nincs benne a B-ben, vagy benne van a B halmazban, de nincs benne az A-ban. Ezek alapján

```
int szimmetrikusDifferencia(int A[], int B[], int n)
{
  int i, sum= 0;
  for ( i= 0; i < n; ++i )
    sum+= (A[i] && !B[i]) || (B[i] && !A[i]);
  return sum;
}
```

Könnyű látni, hogy a logikai kifejezés éppen azt fejezi ki, hogy az i érték vagy benne van A-ban, de nincs benne B-ben, vagy benne van B-ben, de nincs benne A-ban. Ha a két részkifejezés valamelyike igaz, akkor a logikai "vagy"-nak köszönhetően az egész kifejezés értéke "igaz", azaz `1` lesz, amit hozzáadunk a `sum` változóhoz.

6.4. Feladat: Írja át a szimmetrikus differenciát számító függvényben szereplő kifejezést matematikai műveletre!

6.4. Megoldás: A fenti egyszerűbb példák alapján:

```
1 - (1 - A[i]*(A[i] - B[i]))*(1 - B[i]*(B[i] - A[i]));
```

6.5. Feladat: Írjon főprogramot, amely a konzolról feltölt két 5 elemű halmazt, azaz beolvassa az indikátorfüggvényüket, mint egész számokat, majd a kiszámítja és a kimenetre írja a metszetük és uniójuk számosságát!

6.5. Megoldás:

6.0.1. forráskód: halmaz.c

```
#include<stdio.h>

void metszet(int A[], int B[], int C[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    C[i]= A[i] && B[i];
}

void unio(int A[], int B[], int C[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    C[i]= A[i] || B[i];
}

int szamossag(int A[], int n)
{
  int i, sum= 0;
  for ( i= 0; i < n; ++i )
    sum+= A[i];
  return sum;
}
```

```
int main(int argc, char** argv)
{
  int a[5], b[5], c[5];
  int i;

  printf("kerem az elso halmaz indikator fuggvenyet:\n");
  for ( i= 0; i < 5; ++i )
    scanf("%d", &(a[i]));

  printf("kerem az masodik halmaz indikator fuggvenyet:\n");
  for ( i= 0; i < 5; ++i )
    scanf("%d", &(b[i]));

  metszet(a, b, c, 5);
  printf("a halmazok metszetenek szamossaga: %d\n", szamossag(c,5));
  unio(a, b, c, 5);
  printf("a halmazok uniojanak szamossaga: %d\n", szamossag(c,5));

  return 0;
}
```

A fenti alkalmazás lefuttatásakor valamint a példa bemenet begépelésekor az alábbi kimenetet kapjuk:

```
kerem az elso halmaz indikator fuggvenyet:
1 0 1 1 0
kerem az masodik halmaz indikator fuggvenyet:
1 1 1 0 0
a halmazok metszetenek szamossaga:
2
a halmazok uniojanak szamossaga:
4
```

7. Multihalmazok

FONTOS

A *multihalmaz* adatszerkezet a halmaz adatszerkezet (és matematikai fogalom) általánosítása. A koncepcionális különbség: **míg egy halmazban egy elemből csak egy szerepelhetett, a multihalmazban egy elemből tetszőleges sok szerepelhet.** Multihalmazok ábrázolása hasonlóan történik a halmazokéhoz:

- kijelölünk egy alaphalmazt, példáinkban a természetes számok halmazát egy bizonyos N számig,
- a halmazokhoz hasonlóan egy N elemű alaphalmazon értelmezett multihalmazt egy N elemű egész számokból álló tömbbel reprezentálunk,
- a tömb egyes elemei azonban nem logikai értékek lesznek, hanem az i. indexű elem értéke azt adja meg, hányszor szerepel az i szám a multihalmazban.

Lássuk a korábban vizsgált algoritmusok hogyan alakulnak.

Létrehozás. A multihalmazokat a halmazokhoz hasonlóan tehát tömbökkel reprezentáljuk, a különbség azonban, hogy ezen tömbök tetszőleges nem-negatív számot tartalmazhatnak. Az alábbi példában azon $[0, 4]$ alaphalmazon értelmezett multihalmazt hozzuk létre, amely 3 darab 1-es elemet tartalmaz:

```
int multihalmaz[5]= {0, 3, 0, 0, 0};
```

A halmazhoz egy i elemet a

```
++multihalmaz[i];
```

művelettel adhatunk hozzá, és a j számot a

```
if ( multihalmaz[j] > 0 )
  --multihalmaz[j];
```

utasítással vehetjük ki a halmazból.

Tartalmazás vizsgálat. Olyan eljárást szeretnénk írni, amely logikailag igaz értéket ad vissza, ha a paraméterként kapott érték benne van a szintén paraméterként kapott multihalmazban és hamis értéket ad vissza, ha nincs benne:

```
int eleme(int A[], int n, int k)
{
  return A[k] > 0;
}
```

Megjegyezzük, hogy a fenti eljárás esetén feltettük, hogy a vizsgált `k` érték benne van a $[0, N-1]$ tartományban, ezért az indexhatárok vizsgálata nem szerepel az eljárásban. A fenti eljárás nem csak multihalmaz, hanem halmaz esetén is működik. Vegyük észre, hogy a `return` utasítás után szereplő logikai kifejezés teljesen ekvivalens azzal, ha csak `A[k]`-t írunk a `return` utasítás után.

Unió. Természetesen az unió képzésnél egyszerűen össze kell adnunk a megfelelő elemeket. Az alábbi eljárásban az `A` és `B` azon multihalmazokat reprezentáló tömbök, amelyek unióját szeretnénk képezni, a függvény lefutása után `C` tartalmazza majd az uniót, `n` pedig a multihalmazok közös mérete:

```
void unio(int A[], int B[], int C[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    C[i]= A[i] + B[i];
}
```

Metszet. Az alábbi függvény paraméterezése az `unio` függvényéhez hasonló. A metszetképzés értelemszerűen történik: a metszet halmazban az i elem annyiszor fog előfordulni, ahányszor A-ban és B-ben is előfordul, azaz az `A[i]` és `B[i]` közül a kisebb érték kell hogy kerüljön `C[i]`-be.

```
void metszet(int A[], int B[], int C[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    C[i]= A[i] < B[i] ? A[i] : B[i];
}
```

A megoldásban a háromoperandusú operátort használtuk, természetesen a háromoperandusú operátor helyett egy egyszerű `if` utasítás is használható a ciklus magjában.

Különbség. A különbségképzéshez rendre ki kell vonni egyik halmaz elemszámaiból a másik halmaz megfelelő elemszámait, ügyelnünk kell azonban arra, hogy az eredményként előálló multihalmazban nem szerepelhet negatív elemszám, ha az A halmazban valamelyik elemből kevesebb van, mint a B halmazban, akkor a különbséghalmazba 0 kerül az adott elemből.

```
void kulonbseg(int A[], int B[], int C[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    C[i]= A[i] < B[i] ? 0 : A[i] - B[i];
}
```

A fenti eljárásban szintén a háromoperandusú operátort használtuk a kifejezés rövidítése végett, természetesen `if` utasítással is megvalósítható.

Számosság. Egy multihalmaz számosságát a halmaz számosságához hasonlóan mérhetjük:

```
int szamossag(int A[], int n)
{
  int i, sum= 0;
  for ( i= 0; i < n; ++i )
    sum+= A[i];
  return sum;
}
```

Vegyük észre, hogy a multihalmazokra megírt `eleme` és `szamossag` függvények ugyanúgy működnek és használhatóak halmazokra is.

7.1. Feladat: Módosítsa az `unio`, `metszet` és `kulonbseg` eljárásokat úgy, hogy különböző méretű alaphalmazokkal rendelkező multihalmazok esetén is működjenek. Feltehető, hogy ha az A és B halmaz alaphalmazának mérete különböző, akkor az A, tehát első paraméteré lesz nagyobb, és a C paraméterként átvett multihalmaz mérete megegyezik az A méretével!

7.1. Megoldás:

```
void unio(int A[], int B[], int C[], int sizeAC, int sizeB)
{
  int i;
  for ( i= 0; i < sizeB; ++i )
    C[i]= A[i] + B[i];
  for ( ; i < sizeAC; ++i )
    C[i]= A[i];
}

void metszet(int A[], int B[], int C[], int sizeAC, int sizeB)
{
  int i;
  for ( i= 0; i < sizeB; ++i )
    C[i]= A[i] < B[i] ? A[i] : B[i];
  for ( ; i < sizeAC; ++i )
    C[i]= 0;
}

void kulonbseg(int A[], int B[], int C[], int sizeAC, int sizeB)
{
  int i;
  for ( i= 0; i < sizeB; ++i )
    C[i]= A[i] < B[i] ? 0 : A[i] - B[i];
  for ( ; i < sizeAC; ++i )
    C[i]= A[i];
}
```

A fenti függvének paraméterezése azonos, a `sizeAC` paraméter tartalmazza az `A` és `C` multihalmazok azonos alaphalmazának méretét, míg a `sizeB` a `B` multihalmaz alaphalmazának számosságát tartalmazza.

7.2. Feladat: Módosítsa a multihalmazokra megírt `eleme` függvényt úgy, hogy az ellenőrizze azt is, hogy a vizsgált k érték (melyről el szeretnénk dönteni, hogy benne van-e a multihalmazban vagy sem), benne van-e az alaphalmazban, azaz a paraméterként kapott tömbünk indexelésekor nem követünk-e el alul- vagy túlindexelést!

7.2. Megoldás:

```
int eleme(int A[], int n, int k)
{
```

```
  return 0 <= k && k < n && A[k];
}
```

A fenti példában a `return` utáni logikai kifejezések sorrendje viszonylag kötött, ugyanis kihasználjuk, hogy az `&&` operátor rövidzár operátor. Ha az `A[k]` kifejezést a logikai kifejezés elejére írnánk, i < 0 vagy i >= n esetben túlindexelés állhatna elő. FONTOS

7.3. Feladat: Írjon főprogramot, amely a konzolról feltölt két 5 elemű multihalmazt, azaz beolvassa rendre elemeik számát, mint egész számokat, majd a kiszámítja és a kimenetre írja a metszetük és uniójuk számosságát!

7.3. Megoldás:

7.0.1. forráskód: multihalmaz.c

```
#include<stdio.h>

void metszet(int A[], int B[], int C[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    C[i]= A[i] + B[i];
}

void unio(int A[], int B[], int C[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    C[i]= A[i] < B[i] ? A[i] : A[i] - B[i];
}

int szamossag(int A[], int n)
{
  int i, sum= 0;
  for ( i= 0; i < n; ++i )
    sum+= A[i];
  return sum;
}

int main(int argc, char** argv)
{
  int a[5], b[5], c[5];
  int i;

  printf("kerem az elso multihalmaz elemeit:\n");
  for ( i= 0; i < 5; ++i )
    scanf("%d", &(a[i]));

  printf("kerem az masodik multihalmaz elemeit:\n");
  for ( i= 0; i < 5; ++i )
    scanf("%d", &(b[i]));

  metszet(a, b, c, 5);
  printf("a multihalmazok metszetenek szamossaga: %d\n", szamossag(c,5));
  unio(a, b, c, 5);
  printf("a multihalmazok uniojanak szamossaga: %d\n", szamossag(c,5));

  return 0;
}
```

A fenti alkalmazás lefuttatásakor valamint a példa bemenet begépelésekor az alábbi kimenetet kapjuk:

```
kerem az elso multihalmaz elemeit fuggvenyet:
2 0 3 4 0
```

```
kerem az masodik multihalmaz elemeit fuggvenyet:
4 4 3 0 0
a multihalmazok metszetenek szamossaga:
5
a multihalmazok uniojanak szamossaga:
20
```

8. Két- és többdimenziós tömb

Korábban megismertük a tömb adatszerkezetet, ami könnyen látható, hogy éppen a matematikai vektor koncepció megfelelője. Most megnézzük, hogy tömb segítségével hogyan hozhatunk létre olyan adatszerkezetet, amely a kétdimenziós mátrix vagy bármilyen többdimenziós adatszerkezet megfelelője lehet.

Két vagy többdimenziós tömbök tárolási módjának magyarázatát és paraméterként történő átadásának lehetőségeit lásd a *Magasszintű programozási nyelvek 1 - gyakorlati jegyzet* megfelelő szakaszaiban. Az alábbiakban rögzített méretű két- és többdimenziós tömböket vizsgálunk.

Rögzített $N \times M$ méretű, `tipus` típusú és `azonosito` nevű kétdimenziós tömböt az alábbi módon hozhatunk létre:

```
tipus azonosito[N][M];
```

A kétdimenziós tömb szemléletesen egydimenziós tömbök tömbje, így kezdőértéket az egydimenziós tömbök esetéhez hasonlóan adhatunk neki, azonban ügyelnünk kell rá, hogy a tömb elemei szintén tömbök legyenek:

```
tipus azonosito[N][M]= {{konstKif1, konstKif2, konstKifM}, ..., {konstKif1, konstKif2,
    konstKifM}};
```

Fontos megjegyeznünk, hogy a fenti felírásban az `azonosito` nevű tömb M elemű egészeket tartalmazó tömbök N elemű tömbje. Két dimenziós tömb (mátrix) esetén annak i, j indexű elemét `azonosito[i][j]` módon érhetjük el. Lássunk egy konkrét példát, amelyben létrehozok egy tömböt, majd a kimenetre írom az értékeit, úgy, hogy minden sorban a második paraméternek (M) megfelelő méretű résztömbök elemei kerüljenek:

```
int t[3][2]= {{1,2},{2,3},{3,4}};
int i,j;
for ( i= 0; i < 3; ++i )
{
  for ( j= 0; j < 2; ++j )
    printf("%d ",t[i][j]);
  printf("\n");
}
```

A fenti kódrészlet lefutásának eredménye:

```
1 2
2 3
3 4
```

Vegyük észre, hogy az `i,j` indexváltozók felső határa annak a dimenziónak a mérete, amely helyen alkalmazzuk őket. Többdimenziós tömbök esetén hasonlóan járunk el, csak tovább növeljük a dimenziók számát. Az alábbiakban egy három dimenziós esetet vizsgálunk:

```
int t[2][3][4]= {{{1,2,3,4},{2,3,4,5},{3,4,5,6}},{{4,5,6,7},{5,6,7,8},{6,7,8,9}}};
int i, j, k;
for ( i= 0; i < 2; ++i )
```

```
{
  for ( j= 0; j < 3; ++j )
  {
    for ( k= 0; k < 4; ++k )
      printf("%d ", t[i][j][k]);
    printf("\n");
  }
  printf("\n");
}
```

A fenti kódrészlet kimenete:

```
1 2 3 4
2 3 4 5
3 4 5 6

4 5 6 7
5 6 7 8
6 7 8 9
```

A két- illetve többdimenziós adatszerkezetek funkciójukat tekitve vagy egy matematikai konstrukció megvalósulásai (pl. mátrix), vagy egyszerűen tároló (container) adatszerkezetek, azaz egyetlen céljuk, hogy két-, három, stb. indextől függő adatokat tudjunk tárolni bennük. Mivel utóbbi esetben az adatok erősen függenek az indexektől, így kevés olyan algoritmus van, amely ténylegesen ilyen többdimenziós adatszerkezeteken hajtana végre műveleteket, hiszen bármilyen algoritmus, ami megváltoztatja az elemek helyét, stb., elrontja az indexektől való függőséget. Az egyetlen algoritmus, amit itt megvizsgálunk, a sor-oszlop kompozíciós mátrixszorzás. Az alábbiakban rögzített méretű kétdimenziós tömbökre készítjük el a mátrix szorzó eljárást. Feltettük, hogy `X`, `Y` és `Z` globális konstansok:

```
const int X= 3;
const int Y= 4;
const int Z= 5;
void matrixSzorzas(float a[X][Y], float b[Y][Z], float c[X][Z])
{
  int i, j, k;
  float sum;
  for ( i= 0; i < X; ++i )
  {
    for ( j= 0; j < Z; ++j )
    {
      sum= 0;
      for ( k= 0; k < Y; ++k )
        sum+= a[i][k]*b[k][j];
      c[i][j]= sum;
    }
  }
}
```

A fenti példában az `a` és `b` mátrixok szorzatát számítjuk ki a `c` mátrixba.

FONTOS

Két és többdimenziós tömbök létrehozásának fenti módja azzal az előnnyel jár, hogy nem kell a többdimenziós tömbök méreteit átadni paraméterként (természetesen a fenti megközelítés egydimenziós tömbök esetén is használható). Hátránya azonban, hogy a fenti függvény csak a rögzített `X=3`, `Y=4` és `Z=5` méretű tömbök esetén fog működni, azaz csak akkor használhatjuk különböző méretű mátrixok szorzására a fenti függvényt, ha a mátrixok szorzása előtt megváltoztatjuk az `X`, `Y` és `Z` változó értékét a kódban, majd újra lefordítjuk azt. Ez azonban a legtöbb esetben ellent mond az eljárásorientált programozás szemléletének, azaz annak, hogy egy függvény minél általánosabban valósítson meg egy funkciót.

Ahhoz, hogy az eljárásunk tetszőleges méretű mátrixok esetén is működjön, a kétdimenziós tömböket

egydimenziós tömbként kell ábrázolnunk. Az egydimenziós tömbként történő ábrázolás történhet sor- vagy oszlopfolytonosan. A sorfolytonos ábrázolás azt jelenti, hogy az $M \times N$ méretű két dimenziós tömb elemeit egy $M * N$ méretű egydimenziós tömbben tároljuk, úgy, hogy a sorai egymást követik az egydimenziós tömbben. Oszlopfolytonos tárolás esetén az egydimenziós tömb mérete szintén $M * N$, azonban az eredeti kétdimenziós tömb oszlopai követik egymást az egydimenziós tömbben.

A gyakorlatban általában a sorfolytonos ábrázolást alkalmazzák.

Adott egy $S \times O$ méretű kétdimenziós tömb (S - sorok száma, O - oszlopok száma) sorfolytonosan ábrázolva egy $S * O$ méretű egydimenziós tömbben. Az a kérdés, hogy hogyan érhetjük el a kétdimenziós tömb s, o indexű elemét (s - sorindex, o - oszlopindex) az egydimenziós tömbben, azaz sorfolytonos ábrázolásnál mi lesz a kétdimenziós tömb s, o indexű elemének sorfolytonos indexe? A választ egy nagyon egyszerű formulában kapjuk meg:

```
egydimenzios_index= s * O + o
```

A fenti nagyon egyszerű gondolat és formula segítségével már átadhatunk paraméterként tetszőleges méretű kétdimenziós tömböt egydimenziós tömbként és a sorfolytonos ábrázolásban az s, o indexű elemet a fenti formula segítségével érhetjük el. A korábbi példában létrehozott kétdimenziós tömböt az alábbi módon valósíthatjuk meg sorfolytonos ábrázolással, kezdőértékadással:

```
int t[6]= {1, 2, 2, 3, 3, 4};
```

Valósítsuk meg a fenti mátrix szorzást megvalósító eljárást, tetszőleges méretű mátrixokra:

```
void matrixSzorzas(float a[], int sa, int oa, float b[], int sb, int ob, float c[])
{
  int i, j, k;
  float sum;
  for ( i= 0; i < sa; ++i )
  {
    for ( j= 0; j < ob; ++j )
    {
      sum= 0;
      for ( k= 0; k < oa; ++k )
        sum+= a[oa * i + k] * b[k * ob + j];
      c[ob * i + j]= sum;
    }
  }
}
```

Könnyű látni, hogy az eljárás működése teljesen megegyezik a korábbi `matrixSzorzas` eljárás működésével, a különbség csak a paraméterezésben valamint az elemek elérésében jelenik meg.

8.1. Feladat: Írjon eljárást, amely paraméterként kap egy egészeket tartalmazó, sorfolytonosan ábrázolt kétdimeniós tömböt valamint annak méreteit és kiírja az elemeit a képernyőre mátrixos alakban!

8.1. Megoldás:

```
void matrixAlak(int t[], int sorok, int oszlopok)
{
  int i, j;
  for ( i= 0; i < sorok; ++i )
  {
    for ( j= 0; j < oszlopok; ++j )
```

```
      printf("%d ", t[i * oszlopok + j]);
    printf("\n");
  }
}
```

8.2. Feladat: Írjon főprogramot, amely beolvassa egy 2×3 (a) valamint 3×2 (b) méretű, lebegőpontos értékeket tartalmazó mátrix elemeit soronként, kiírja őket mátrixos alakban, összeszorozza őket ab valamint ba alakban, majd kiírja az eredményeket is mátrixos alakban! A megoldáshoz használja fel a korábban létrehozott eljárásokat!

8.2. Megoldás: A megoldás során létre kell hoznunk két 6 méretű, egydimenziós tömböt a 2×3 valamint 3×2 méretű mátrixok tárolására, valamint mivel az eredmény mátrixok 2×2 valamint 3×3 méretűek, létrehozunk még egy 4 és egy 9 méretű tömböt az eredmények tárolására.

8.0.1. forráskód: matrix.c

```
#include <stdio.h>

void matrixSzorzas(float a[], int sa, int oa, float b[], int sb, int ob, float c[])
{
  int i, j, k;
  float sum;
  for ( i= 0; i < sa; ++i )
  {
    for ( j= 0; j < ob; ++j )
    {
      sum= 0;
      for ( k= 0; k < oa; ++k )
        sum+= a[oa * i + k] * b[k * ob + j];
      c[ob * i + j]= sum;
    }
  }
}

void matrixAlak(float t[], int sorok, int oszlopok)
{
  int i, j;
  for ( i= 0; i < sorok; ++i )
  {
    for ( j= 0; j < oszlopok; ++j )
      printf("%.1f ", t[i * oszlopok + j]);
    printf("\n");
  }
}

int main(int argc, char** argv)
{
  float a[6];
  float b[6];
  float c[9];
  float d[4];
  int i, j;

  for ( i= 0; i < 2; ++i )
  {
    printf("Kerem az 'a' matrix 3 elemu %d. sorat:\n", i);
    scanf("%f%f%f", &(a[i*3 + 0]), &(a[i*3 + 1]), &(a[i*3 + 2]));
  }
  for ( i= 0; i < 3; ++i )
  {
    printf("Kerem a 'b' matrix 2 elemu %d. sorat:\n", i);
    scanf("%f%f", &(b[i*2 + 0]), &(b[i*2 + 1]));
  }
```

```
    printf("a beolvasott 'a' matrix:\n");
    matrixAlak(a, 2, 3);
    printf("a beolvasott 'b' matrix:\n");
    matrixAlak(b, 3, 2);

    matrixSzorzas(a, 2, 3, b, 3, 2, c);
    printf("a*b:\n");
    matrixAlak(c, 2, 2);

    matrixSzorzas(b, 3, 2, a, 2, 3, d);
    printf("b*a:\n");
    matrixAlak(d, 3, 3);

    return 0;
}
```

A program futásának egy lehetséges kimenete (a látható begépelt számok esetén):

```
Kerem az 'a' matrix 3 elemu 0. sorat:
1 2 3
Kerem az 'a' matrix 3 elemu 1. sorat:
2 3 4
Kerem a 'b' matrix 2 elemu 0. sorat:
1 2
Kerem a 'b' matrix 2 elemu 1. sorat:
2 3
Kerem a 'b' matrix 2 elemu 2. sorat:
3 4
a beolvasott 'a' matrix:
1.0 2.0 3.0
2.0 3.0 4.0
a beolvasott 'b' matrix:
1.0 2.0
2.0 3.0
3.0 4.0
a*b:
14.0 20.0
20.0 29.0
b*a:
5.0 8.0 11.0
8.0 13.0 18.0
11.0 18.0 25.0
```

9. Ritkamátrix

A gyakorlatban mérési eredményekből származó adatokat nagyon gyakran nagyméretű mátrixok formájában kapjuk meg. Ezen mátrixok azonban ritkák, abban az értelemben, hogy valamely értékből nagyon sok van a mátrixban, a többi értékből azonban kevés. Azt az elemet, amelyikből nagyon sok van **gyakori elem**nek nevezzük. A gyakori elem általában a 0. Könnyű belátni, hogy ilyen esetben felesleges az egész mátrixot tárolni, elegendő csak azokat az elemeket és koordinátáikat tárolnunk, amelyek értéke nem egyenlő a gyakori elemmel. Ennek megfelelően a fenti értelemben ritka mátrixoknak több különböző hatékony és helytakarékos reprezentációját is kidolgozták, ezek közül a három soros és négy soros reprezentációt vizsgáljuk meg.

FONTOS

9.1. Három soros reprezentáció

Legyen egy a reprezentálandó mátrix mérete $M \times N$, a gyakori elem előfordulásainak száma k. Ekkor a három soros reprezentáció egy olyan három, $M*N-k$ elemű sorból álló szerkezet, amelyben az első sor, második sor és harmadik sor i elemei rendre az i. nem gyakori elem sorindexét, oszlopindexét és

értékét tartalmazzák. Tekintsük az alábbi mátrixot:

(7)
$$\begin{pmatrix} 0 & 0 & 0 & 3 & 1 \\ 0 & 2 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

A fenti mátrixban a gyakori elem szemmel láthatóan a 0. A mátrix mérete 4×5, a gyakori elem előfordulásainak száma 14, azaz a háromsoros reprezentáció egy három sorból és soronként $4 * 5 - 14 = 6$ elemből álló struktúra lesz:

sor	0	0	1	1	2	2
oszlop	3	4	1	4	1	3
érték	3	1	2	1	1	2

Könnyű látni, hogy az eredeti mátrix ábrázolásához 20 értéket kellett tárolni, ezen utóbbi háromsoros reprezentációhoz azonban elegendő 18 érték tárolása, tehát helyfelhasználás szempontjából hatékonyabb. Természetesen ha még "ritkább" lenne a kiindulási mátrix, az ábrázolás még hatékonyabb lenne. Kérdés, hogy a fenti három soros reprezentációt hogyan valósíthatjuk meg egy programban? A válasz ismét az, hogy egydimenziós tömbök segítségével, azaz a ritka mátrixok reprezentációjához használt három sort három darab egydimenziós tömbbel ábrázolva. Ennek megfelelően, az alábbi függvény paraméterként kap egy sorfolytonosan reprezentált egészeket tartalmazó kétdimenziós tömböt, sorainak és oszlopainak számát, a gyakori elemet, valamint három egydimenziós tömböt, amelyek mérete egy olyan szám, amelytől biztosan kevesebb a nem-gyakori elemek száma a kétdimenziós tömbben. A függvény visszatérési értéke a nem-gyakori elemek pontos száma.

```
int felepit3sor(int t[], int sorok, int oszlopok, int gyakoriElem, int sor3[], int
    oszlop3[], int ertek3[])
{
  int i, j, k= 0;
  for ( i= 0; i < sorok; ++i )
    for ( j= 0; j < oszlopok; ++j )
      if ( t[i*oszlopok + j] != gyakoriElem )
      {
        sor3[k]= i;
        oszlop3[k]= j;
        ertek3[k]= t[i*oszlopok + j];
        ++k;
      }
  return k;
}
```

A fenti függvény létrehozza tehát a paraméterként kapott mátrix három soros reprezentációját a szintén paraméterként kapott egydimenziós tömbökben. Megjegyezzük, feltettük, hogy az egydimenziós tömbök mérete nagyobb vagy egyenlő a gyakori elemtől különböző mátrix elemek számával. **Három soros reprezentáció esetén tehát egy mátrixot három egydimenziós tömb valamint a gyakori elem reprezentál!**. Írjuk most meg azt a függvényt, amely paraméterként kapja egy mátrix három soros reprezentációját (három tömb és azok pontos mérete), a gyakori elemmel nem megegyező elemek számát (azaz az a háromsoros reprezentáció tömbjeinek méretét), a gyakori elemet, valamint az s, o koordinátákat és visszatérési értékként visszaadja az s, o koordinátájú elemet!

FONTOS

```
int elem(int sor3[], int oszlop3[], int ertek3[], int meret, int gyakoriElem, int s, int
    o)
{
  int i;
  for ( i= 0; i < meret; ++i )
```

```
    if ( sor3[i] == s && oszlop3[i] == o )
      return ertek3[i];
  return gyakoriElem;
}
```

Könnyű látni, hogy az utolsó ciklusban tulajdonképpen egy teljes keresést valósítunk meg, azaz megvizsgáljuk a három soros reprezentációhoz használt `sor3` és `oszlop3` tömböket, hogy tartalmazzák-e azon elem indexét (s,o), amelyet la akarunk kérdezni, ha ugyanis tartalmazzák, akkor visszaadjuk az `ertek3` tömb megfelelő elemét, ha nem tartalmazzák, akkor visszaadjuk a gyakori elemet.

9.2. Négy soros reprezentáció

A négysoros reprezentáció a háromsoros reprezentáció kibővítése egy újabb sorral, amely a háromsoros reprezentációba bekerült minden elemhez a háromsoros reprezentáció azon indexét tartalmazza, amely az elem oszlopában a következő nem-gyakori elem. Ha nincs ilyen, akkor a speciális -1 értéket tartalmazza. Tehát a korábban is használt

(8)
$$\begin{pmatrix} 0 & 0 & 0 & 3 & 1 \\ 0 & 2 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

mátrix esetén:

sor	0	0	1	1	2	2
oszlop	3	4	1	4	1	3
érték	3	1	2	1	1	2
mutató	5	-1	4	-1	-1	-1

Könnyű látni tehát, hogy az első nem-gyakori elem a 3, amelynek oszlopában a következő az utolsó sorban található 2. Ez a 2 a háromsoros ábrázolás utolsó, azaz 5 indexű eleme, ezért a 3 elemhez tartozó mutató az 5, ami így tehát a megfelelő oszlop következő nem-gyakori elemére mutató a háromsoros ábrázolásban. A négysoros reprezentációt felépítő függvény pedig az alábbi:

```
int felepit4sor(int t[], int sorok, int oszlopok, int gyakoriElem, int sor3[], int
    oszlop3[], int ertek3[], int mutato3[])
{
  int i, j, k= 0;
  for ( i= 0; i < sorok; ++i )
    for ( j= 0; j < oszlopok; ++j )
      if ( t[i*oszlopok + j] != gyakoriElem )
      {
        sor3[k]= i;
        oszlop3[k]= j;
        ertek3[k]= t[i*oszlopok + j];
        ++k;
      }
  for ( i= 0; i < k; ++i )
  {
    for ( j= i+1; j < k; ++j )
      if ( oszlop3[i] == oszlop3[j] )
        break;
    if ( j == k )
      mutato3[i]= -1;
    else
      mutato3[i]= j;
  }

  return k;
}
```

A négysoros reprezentációt felépítő függvény első része, azaz a `sor3`, `oszlop3` és `ertek3` tömbök kitöltése megegyezik a háromsoros reprezentációt felépítő eljárással. Ezt követi egy for-ciklus, amely rendre végig megy a már kitöltött háromsoros reprezentáción, és minden elemhez olyan elemet keres, amelyiknek az oszlop indexe megegyezik az éppen vizsgált elemével. Ha talál ilyet, akkor berakja a `mutato3` tömb megfelelő helyére az indexét, ha nem talál, akkor `-1`-et rak a mutató tömbbe. Könnyű látni, hogy ezen utóbbi ciklus belseje éppen egy teljes keresés, ahol a visszatérési érték a `mutato3` tömb `i` indexű helyére kerül.

A háromsoros reprezentációnál megírt `elem` függvény négysoros reprezentációnál is használható.

9.1. Feladat: Készítse el az alábbi mátrix három és négysoros reprezentációját!

$$\begin{pmatrix} 1 & 1 & 2 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \tag{9}$$

9.1. Megoldás: Könnyű látni, hogy a gyakori elem az 1. Ez alapján tehát a háromsoros reprezentáció:

sor	0	1
oszlop	2	2
érték	2	0

A négysoros reprezentáció pedig:

sor	0	1
oszlop	2	2
érték	2	0
mutató	1	-1

9.2. Feladat: Készítse el az alábbi négysoros reprezentációhoz tartozó mátrix-ot, ha a mátrix négy sorból és három oszlopból áll, a gyakori elem pedig -1!

sor	0	2
oszlop	1	2
érték	1	2
mutató	-1	-1

9.2. Megoldás:

$$\begin{pmatrix} -1 & 1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & 2 & -1 \end{pmatrix} \tag{10}$$

9.3. Feladat: Írjon eljárást, amely paraméterként kapja egy kétdimenziós tömb háromsoros reprezentációját (3 tömb, azok pontos mérete), a gyakori elemet valamint a mátrix méretét (sorok, oszlopok száma) és a kimenetre írja a mátrixot mátrix alakban! (Feltehetjük, hogy a háromsoros reprezentációba a sorfolytonos ábrázolásnak megfelelő sorrendben kerültek be az elemek!)

9.3. Megoldás:

```
void kiir(int sor3[], int oszlop3[], int ertek3[], int meret, int gyakoriElem, int sorok
    , int oszlopok)
{
  int i, j;
  for ( i= 0; i < sorok; ++i )
  {
    for ( j= 0; j < oszlopok; ++j )
      printf("%d ", elem(sor3, oszlop3, ertek3, meret, gyakoriElem, i, j));
    printf("\n");
  }
}
```

A megoldáshoz felhasználtuk a korábban megírt `elem` függvényt.

9.4. Feladat: Írjon főprogramot, amely beolvas egy 5 sorból és 4 oszlopból álló kétdimenziós tömböt soronként, amelyről tudjuk, hogy a gyakori elem a 0, és a nem-gyakori elemek száma legfeljebb 10. A program elkészíti a mátrix háromsoros reprezentációját a `felepit` függvénnyel, majd a kimenetre írja azt, és ezt követően a háromsoros reprezentáció alapján a `kiir` függvénnyel a kimenetre írja az eredeti mátrixot is!

9.4. Megoldás:

9.2.1. forráskód: ritkamatrix.c

```
#include <stdio.h>

int felepit3sor(int t[], int sorok, int oszlopok, int gyakoriElem, int sor3[], int
    oszlop3[], int ertek3[])
{
  int i, j, k= 0;
  for ( i= 0; i < sorok; ++i )
    for ( j= 0; j < oszlopok; ++j )
      if ( t[i*oszlopok + j] != gyakoriElem )
      {
        sor3[k]= i;
        oszlop3[k]= j;
        ertek3[k]= t[i*oszlopok + j];
        ++k;
      }
  return k;
}

int elem(int sor3[], int oszlop3[], int ertek3[], int meret, int gyakoriElem, int s, int
     o)
{
  int i;
  for ( i= 0; i < meret; ++i )
    if ( sor3[i] == s && oszlop3[i] == o )
      return ertek3[i];
  return gyakoriElem;
}

void kiir(int sor3[], int oszlop3[], int ertek3[], int meret, int gyakoriElem, int sorok
    , int oszlopok)
{
  int i, j;
  for ( i= 0; i < sorok; ++i )
  {
    for ( j= 0; j < oszlopok; ++j )
      printf("%d ", elem(sor3, oszlop3, ertek3, meret, gyakoriElem, i, j));
    printf("\n");
  }
}
```

```
int main(int argc, char** argv)
{
  int t[20];
  int sor3[10];
  int oszlop3[10];
  int ertek3[10];
  int meret;
  int i;

  for ( i= 0; i < 5; ++i )
  {
    printf("Kerem az 't' matrix 4 elemu %d. sorat:\n", i);
    scanf("%d%d%d%d", &(t[i*4 + 0]), &(t[i*4 + 1]), &(t[i*4 + 2]), &(t[i*4 + 3]));
  }

  meret= felepit3sor(t, 5, 4, 0, sor3, oszlop3, ertek3);

  printf("sor\t");
  for ( i= 0; i < meret; ++i )
    printf("%d ", sor3[i]);
  printf("\noszlop\t");
  for ( i= 0; i < meret; ++i )
    printf("%d ", oszlop3[i]);
  printf("\nertek\t");
  for ( i= 0; i < meret; ++i )
    printf("%d ", ertek3[i]);
  printf("\n");

  kiir(sor3, oszlop3, ertek3, meret, 0, 5, 4);

  return 0;
}
```

A program kimenete a begépelt sorok esetén:

```
Kerem az 't' matrix 4 elemu 0. sorat:
0 0 0 1
Kerem az 't' matrix 4 elemu 1. sorat:
2 0 0 0
Kerem az 't' matrix 4 elemu 2. sorat:
0 0 -1 0
Kerem az 't' matrix 4 elemu 3. sorat:
0 0 0 0
Kerem az 't' matrix 4 elemu 4. sorat:
0 0 1 0
sor     0 1 2 4
oszlop  3 0 2 2
ertek   1 2 -1 1
0 0 0 1
2 0 0 0
0 0 -1 0
0 0 0 0
0 0 1 0
```

9.5. Feladat: A ritka mátrixok három- és négysoros tárolásának előnye tehát, hogy kevesebb helyen lehet tárolni őket, mint magát a teljes mátrixot. Számolja ki, hogy egy $N \times M$ méretű, egészeket tartalmazó mátrix esetén legfeljebb hány nem-ritka elem fordulhat elő a mátrixban, hogy a három, valamint négysoros reprezentáció kevebb helyet foglaljon a memóriában, mint az eredeti mátrix!

9.5. Megoldás: A háromsoros reprezentáció esetén minden egyes nem-gyakori elem előfordulást három adattal rögzítünk, a kérdés tehát az, hogy melyik az az n érték, amelyre

(11) $$3n < MN$$

teljesül? A válasz természetesen az, hogy legfeljebb

$$n < \frac{MN}{3} \tag{12}$$

olyan elem lehet a mátrixban, amely nem egyezik meg a gyakori elemmel. Négysoros reprezentáció esetén hasonlóan adódik az

$$n < \frac{MN}{4} \tag{13}$$

FONTOS megoldás. Megjegyezzük, hogy a fenti számításoknál kihasználtuk, hogy mind az indexek, mind a mátrix értékei `int` egész típusúak, így méretük megegyezik. Más típusú adatoknál, vagy más típussal indexelt mátrix esetén más értéket kaphatunk a hatékonyság felső határára!

10. Sor

FONTOS A sor adatszerkezet az egyik leggyakrabban használt adatszerkezet, amely az alábbi műveletekkel rendelkezik:

- üres sor létrehozása,
- sor ürességének vizsgálata,
- elem hozzáadása a sorhoz (PUT),
- első elem elérése (TOP),
- elem eltávolítása (GET),
- (a GET művelet egyben TOP művelet is lehet, azaz visszaadhatja az éppen eltávolított elemet).

A sor a műveleteiből is láthatóan dinamikus adatszerkezet, azaz a benne tárolt elemek mérete változhat. Viselkedését tekintve FIFO adatszerkezet, ami az angol **F**irst **I**n, **F**irst **O**out kifejezésből ered, ami arra utal, hogy amelyik elem először bekerül a sorba, az is kerül ki belőle először. A sornak tehát nem érhető el minden eleme közvetlenül, úgy, mint a tömbnek, mindig csak egy elemhez férhetünk hozzá.

Szemléletesen a sort úgy képzelhetjük el, mint egy sort egy bevásárlóközpont pénztáránál, aki előbb beáll a sorba, az is áll ki először a sorból, és senki más nem áll ki a sorból, csak amikor sorrakerül, tehát a beérkezés sorrendjében történik a távozás is.

A sorhoz kapcsolódóan a fenti műveleteket fogjuk megvalósítani. A megvalósítás során megint csak tömböt használunk, így bár elméletben egy sor tetszőleges sok elemet tartalmazhat, a gyakorlatban a sor maximális elemszámát korlátozni fogja a megvalósításához használt tömb mérete.

Természetesen a sorokat kezelő algoritmusokat is nagyon sok féle módon meg lehet valósítani, mi most az egyszerűség kedvéért olyan megvalósítást nézünk meg, amely globális változókkal dolgozik. A programunk tehát szemben az eddigiekkel egyetlen adatszerkezettel fog dolgozni, amely min-

FONTOS den függvényből és eljárásból elérhető lesz. **A globális változó olyan változó, amelyet a kód szövegében függvénydefiníciókon kívül helyezhetünk el, és a változó tetszőleges őt követő függvényben használható, paraméterátadás nélkül.**

A következő három változata a sornak mind ugyanazt az adatszerkezetet valósítja meg, az egyes megvalósítások csak abban különböznek, hogy melyik hogyan kezeli azokat az eseteket, amikor az elemek elérik a tömb végét, stb.

10.1. Rögzített sor

A rögzített sor

- az elemeket tömbben tárolja, a tömb mérete megadja a maximális elemszámot,
- a sor elemének indexét a tömbben egy `veg` változóban tartjuk nyilván,
- a sor első eleme mindig a tömb első eleme,
- ha eltávolítunk egy elemet a sor elejéről, azaz "kivesszük" a sorból az első elemet, a sor többi elemét egyel előrébb csúsztatjuk a sorban.

A továbbiakban egy egészeket tartalmazó sort és annak műveleteit vizsgáljuk.

Létrehozás. Sor létrehozása példánkban egy tetszőleges méretű globális tömb változó létrehozását és egy szintén globális `veg` változó létrehozását, valamint a `-1`-re állítását jelenti. A `veg` változó értéke tehát mindig a sor aktuális utolsó elemének indexe a tömbben, a `-1` azt jelzi, hogy üres a sor. Tehát a

```
const int N= 20;
int sor[20];
int veg= -1;
```

kódrészlettel létrehoztunk egy üres sort. Esetünkben az `N` konstans tárolja a tömb méretét, azaz a sor maximális elemszámát.

Üres sor vizsgálata. Azt, hogy üres-e a sor egyszerűen egy olyan függvénnyel vizsgálhatjuk meg, amely logikai igaz értéket ad vissza ha a sorban nincs egyetlen elem sem (a `veg` változó értéke `-1`).

```
int uresSor()
{
  return veg == -1;
}
```

Első elem elérése. (TOP) A sor első eleme mindig a tömb első eleme lesz, így a sor első elemét a tömb `0` indexű elemének elérésével kérdezhetjük le. Az alábbi függvény visszaadja a sor első elemének értékét:

```
int top()
{
  return sor[0];
}
```

Elem hozzáadása. (PUT) Elem hozzáadását a `veg` változó felhasználásával végezzük egy nagyon egyszerű eljárásban, amely paraméterként kapja azt az elemet (esetünkben egész számot), amelyet be szeretnénk tenni a sor végére.

```
void put(int x)
{
  if ( veg == N - 1 )
    printf("A sor tele van!\n");
  else
  {
    ++veg;
    sor[veg]= x;
  }
}
```

Elem eltávolítása. (GET) Elem eltávolítása mindig az első elem, tehát a `0` indexű tömb elem eltávolítását jelenti. Célunk, hogy az eltávolítás után az eddigi második elem legyen a 0 indexű, tehát a tömb elemeit a tömb elejére kell csúsztatni, azaz

```
int get()
{
  int i;
  int aktualis= -1;
  if ( veg == -1 )
    printf("A sor üres!\n");
  else
  {
    aktualis= sor[0];
    for ( i= 0; i < veg; ++i )
      sor[i]= sor[i+1];
    --veg;
  }
  return aktualis;
}
```

Ha üres a sor, semmi sem történik. Ha a sor azonban nem üres, akkor az első elem eltávolítását és a többi elem előre csúsztatását egy lépésben végezzük, a tömb `i+1`. elemét a tömb `i`. elemére változtatva, majd csökkentjük a `veg` változó értékét, hiszen a tömb így már kevesebb elemet tartalmaz.

10.2. Vándorló sor

A vándorló sor a sor adatszerkezetnek a rögzített sortól egy hatékonyabb megvalósítása. A hatékonyság abban nyilvánul meg, hogy míg rögzített sornál az elemeket minden egyes *get* műveletnél mozgatni kell, a vándorló sor esetén az első elem indexét egy újabb `int` típusú változóban tartjuk nyilván, így eltávolításnál, amikor a második elem válik a sor új első elemévé, elegendő a sor kezdetét jelző indexváltozó értékét növelni egyel. Elem mozgatásra csak akkor van szükség, ha a sor vége elért a tömb végét, ekkor az összes elemet a sor elejére csúsztatjuk.

Létrehozás. Hasonló a rögzített sorhoz, a különbség, hogy egy újabb, `kezdet` változót is létrehozunk:

```
const int N= 20;
int sor[20];
int veg= -1;
int kezdet= 0;
```

Most a `sor`, a `veg` és a `kezdet` változók reprezentálják számunkra a sor adatszerkezetet.

Üres sor vizsgálata. Mivel ebben az esetben a sor kezdete is vándorol, a sor utolsó elemének indexéből nem tudjuk megmondani, hogy üres-e a sor. Ha azonban a sor utolsó elemének indexe kisebb, mint a sor első elemének indexe, akkor értelemszerűen az indextartomány egy üres intervallumot definiál a tömbben, azaz a tömbben ábrázolt sor üres:

```
int uresSor()
{
  return kezdet > veg;
}
```

Első elem elérése. (TOP) Az első elem elérés a sor kezdetét jelző `kezdet` változóban tárolt index alapján történik:

```
int top()
{
  return sor[kezdet];
}
```

Új elem hozzáadása. (PUT) Új elemet a sor végére rakhatunk be, a korábbihoz hasonló módon:

```
void put(int x)
{
  int i;
  if ( kezdet == 0 && veg == N - 1 )
  {
    printf("A sor megtelt!\n");
    return;
  }
  else if ( veg == N - 1 )
  {
    for ( i= kezdet; i < N; ++i )
      sor[i - kezdet]= sor[i];
    veg-= kezdet;
    kezdet= 0;
  }
  ++veg;
  sor[veg]= x;
}
```

A sor végére új elemet berakó függvény egy kicsit komplexebb, mint a rögzített sor esetén. Az első `if` feltételben azt vizsgáljuk meg, hogy tele van-e a tömb, azaz a `kezdet` változó értéke `0`, a `veg` változó értéke `N-1`-e. Ha igen, akkor nem tudunk új elemet hozzáadni a sorhoz, kiírunk egy hibaüzenetet, majd befejezzük az eljárást. Az `else if` ágban található feltétel akkor fog teljesülni, ha a `veg` változó elérte a tömb végét, azonban a `kezdet` változó értéke nem `0`, azaz a sor kezdete elvándorolt a tömb kezdetéről, így ha előrecsúsztatjuk az elemeket úgy, hogy a `kezdet` változó a 0 index-re mutasson, akkor a tömb végén újra lesz hely új elemeknek. A `for` ciklus ezt az előrecsúsztatást valósítja meg. Ezt követően mivel éppen a `kezdet` változó értékének megfelelő számú lépést tett meg minden elem a tömb eleje felé, a `veg` változóból kivonjuk a `kezdet` értékét, hogy megkapjuk a sor aktuális utolsó elemének indexét, míg a `kezdet` változó értékét `0`-ra állítjuk. Ezt követően, ha a vezérlés eljut a `++veg` utasításig, az azt jelenti, hogy tudunk berakni új elemet a sor végére, így megnöveljük az utolsó elem indexét egyel (`veg`), és betesszük az új elemet a sor végére.

Elem eltávolítása. (GET) Az első elem eltávolítása nagyon egyszerű, a `kezdet` értékét kell csak növelnünk egyel.

```
int get()
{
  if ( kezdet > veg )
  {
    printf("A sor üres!\n");
    return -1;
  }
  else
    ++kezdet;

  return sor[kezdet-1];
}
```

10.3. Ciklikus sor

A ciklikus a sor a sorok tömbbel történő megvalósításának leghatékonyabb módja, ciklikus megvalósítás esetén sohasem fordul elő elem mozgatás, minden esetben csak a sor kezdetét és végét jelző változókat manipuláljuk. Ha a `veg` változó elérte a tömb végét, akkor az elején fog újra megjelenni, tehát a sor mintegy körbeérhet a tömbön. Az indexműveletek során az indexváltozók "körbejárását" a maradékos osztás művelettel valósítjuk meg azaz az indexváltozók növelése után mindig maradékosan osztjuk azokat a tömb méretével. Abból kifolyólag, hogy a `veg` változóban tárolt index a maradékos osztás miatt a `kezdet` változó elé kerülhet, előállhat az a kellemetlen helyzet, hogy

FONTOS a `veg` változó értéke éppen egyel kevesebb, mint a `kezdet` értéke. **Ekkor nem tudjuk eldönteni, hogy a sor üres-e, vagy tele van-e.** Hogy megoldást találjunk erre, bevezetünk egy újabb változót, amelyben azt fogjuk számolni, hogy hány elem van a sorban. Ez a számlás egyszerűen azt jelenti, hogy minden `get` műveletnél kivonunk egyet a változó érétkéből és minden `put` művelet során hozzáadunk egyet a változó értékéhez. Ezt követően már ezen változó alapján el tudjuk dönteni, hogy üres-e a sor.

Létrehozás.

```
const int N= 20;
int sor[20];
int veg= -1;
int kezdet= 0;
int elemSzam= 0;
```

Az előzőek alapján tehát van egy új változónk, az `elemSzam`.

Üres sor vizsgálata. Akkor lesz üres a sor, ha a `veg` változó értéke az inicializálásnak megfelelő `-1` érték, vagy ha a `kezdet` és `veg` változó értéke megegyezik, azaz

```
int uresSor()
{
  return elemSzam == 0;
}
```

Első elem elérése. (TOP) A top művelet szintén nagyon egyszerűen valósítható meg, az eddigiekhez hasonló módon a `kezdet` változó által indexelt tömb elem.

```
int top()
{
  return sor[kezdet];
}
```

Új elem hozzáadása. (PUT) Ha a sorban lévő elemek száma kisebb, mint a maximális elemszám - 1, az azt jelenti, hogy van még hely a sorban, tehát felvehetjük az `x` paraméter értékét a sor végére:

```
void put(int x)
{
  if ( elemSzam < N - 1 )
  {
    veg= (veg + 1) % N;
    sor[veg]= x;
    ++elemSzam;
  }
  else
  {
```

```
    printf("A sor tele van!\n");
  }
}
```

Új elem eltávolítása. (GET) Új elem eltávolítását nagyon egyszerűen tehetjük meg: ha nem üres a sor, akkor egyszerűen növeljük a `kezdet` változó értékét:

```
int get()
{
  if ( elemSzam > 0 )
  {
    kezdet= (kezdet + 1) % N;
    --elemSzam;
  }
  else
  {
    printf("A sor üres!\n");
    return -1;
  }
  return sor[(kezdet-1)%N];
}
```

10.4. Példa

Lássunk most egy egyszerű példát, üres sorból kiindulva a táblázatban foglalom össze, hogy egy `5` elemű tömbben tárolt sor tartalma hogyan változik az egyes megvalósítások esetén. A példában pirossal jelzem a sor első elemét (amely tehát minden esetben a `top` függvény visszatérési értéke), és félkövérrel szedem az utolsó elemet. Az alulvonás jelek az `5` elemű tömb egyes helyeit jelzik.

Művelet	Rögzített sor	Vándorló sor	Ciklikus sor
put(3)	3 _ _ _ _	3 _ _ _ _	3 _ _ _ _
put(5)	3 **5** _ _ _	3 **5** _ _ _	3 **5** _ _ _
put(2 + 2)	3 5 **4** _ _	3 5 **4** _ _	3 5 **4** _ _
get()	5 **4** _ _ _	_ 5 **4** _ _	_ 5 **4** _ _
get()	4 _ _ _ _	_ _ 4 _ _	_ _ 4 _ _
get()	_ _ _ _ _	_ _ _ _ _	_ _ _ _ _
put(1)	1 _ _ _ _	_ _ _ 1 _	_ _ _ 1 _
put(5)	1 **5** _ _ _	_ _ _ 1 **5**	_ _ _ 1 **5**
put(7)	1 5 **7** _ _	1 5 **7** _	**7** _ _ 1 5
put(6)	1 5 7 **6** _	1 5 7 **6** _	7 **6** _ 1 5
get()	5 7 **6** _ _	_ 5 7 **6** _	7 **6** _ _ 5
get()	7 **6** _ _ _	_ _ 7 **6** _	7 **6** _ _ _

A következő példaprogramban a rögzített sorokon keresztül szemléltetjük a sorok használatának menetét. Egész számokat olvasok be a bemenetről egészen a `-1` értékig, és berakom őket egy sorba, egyedül a `put` műveletet használva. Ezt követően egy ciklusban a `top`, `get` és `uresSor` függvények segítségével kiírom a sor tartalmát és ki is ürítem azt.

10.4.1. forráskód: sor.c

```
#include <stdio.h>

const int N= 20;

int sor[20];
int veg= -1;
```

```
int uresSor()
{
  return veg == -1;
}

int top()
{
  return sor[0];
}

void put(int x)
{
  if ( veg == N - 1 )
    printf("A sor tele van!\n");
  else
  {
    ++veg;
    sor[veg]= x;
  }
}

int get()
{
  int i;
  int aktualis= -1;
  if ( veg == -1 )
    printf("A sor Ăźres!\n");
  else
  {
    aktualis= sor[0];
    for ( i= 0; i < veg; ++i )
      sor[i]= sor[i+1];
    --veg;
  }

  return aktualis;
}

int main(int argc, char** argv)
{
  int szam;

  do
  {
    printf("kerek egy szamot:\n");
    scanf("%d", &szam);

    if ( szam == -1 )
      break;

    put(szam);
  }while ( 1 );

  printf("a sor elemei:\n");
  while ( ! uresSor() )
  {
    printf("%d ", top());
    get();
  }

  return 0;
}
```

A példa bemenet és kimenet:

```
kerek egy szamot:
2
kerek egy szamot:
4
kerek egy szamot:
3
kerek egy szamot:
6
kerek egy szamot:
-1
a sor elemei:
2 4 3 6
```

10.1. Feladat: Három elemű tömbben reprezentálva a rögzített, vándorló és ciklikus sorokat, mi lesz a sorok tartalma és mi lesz a `kezdet`, `veg`, `elemSzam` változók értéke az alábbi műveletek egymás után történő végrehajtását követően?

- put(3);
- put(2);
- put(2);
- get();

10.1. Megoldás: A megoldásban pirossal jelöltem a sor első, félkövérrel a sor utolsó elemét.

- Rögzített sor: 2 **2** 2 (`veg=1`).
- Vándorló sor: 3 2 **2** (`veg=2`, `kezdet=1`).
- Ciklikus sor: 3 2 **2** (`veg=2`, `kezdet=1`, `elemSzam=2`).

A megoldásban zölddel jelölt értékek nem részei a sornak, azonban a végrehajtott műveletek után az adott helyeken megtalálhatóak a tömbökben.

10.2. Feladat: Négy elemű tömbben reprezentálva a rögzített, vándorló és ciklikus sorokat, mi lesz a sorok tartalma és mi lesz a `kezdet`, `veg`, `elemSzam` változók értéke az alábbi kódrészlet végrehajtása után?

```
int a= 2;
put(3);
put(a*a);
put(a + 2);
put(a % 2);
a= get();
get();
put(a);
```

10.2. Megoldás: A megoldásban pirossal jelöljük a sor első, félkövérrel a sor utolsó elemét.

- Rögzített sor: 4 0 **3** 0 (`veg=2`).
- Vándorló sor: 4 0 **3** 0 (`kezdet=0`, `veg=2`),
- Ciklikus sor: **3**, 4 4 0 (`kezdet=2`, `veg=0`, `elemSzam=3`).

Megjegyezzük, hogy az előző példához hasonlóan a zölddel szedett értékek nem részei a soroknak, a műveletek során azonban bekerültek a tömbökbe és felülírásig ottmaradnak.

10.3. Feladat: Írjon programot, amely végrehajtja az előző feladatban leírt műveletsort ciklikus soron, majd a kimenetre írja a sor reprezentációjára használt tömb tartalmát. A sor első elemét a tömbben egy azt megelőző . karakterrel jelezze, a sor utolsó elemét pedig egy azt megelőző _ karakterrel.

10.3. Megoldás: A megoldáshoz egyszerűen egy programba szervezzük a ciklikus sort kezelő `put` eljárást és `get` függvényt, majd miután a főprogramba (`main`-függvénybe) bemásoltuk az előző feladat műveletsorát, egy `for`-ciklusban kiírjuk a `sor` tömb tartalmát. A `for`-ciklusban figyeljük, hogy az

`i` ciklusváltozó értéke épp `kezdet` vagy `veg`-e, és ha igen, akkor kiírjuk a sorban tárolt szám előtt kiírjuk a megfelelő szimbólumot.

10.4.2. forráskód: ciklikussor.c

```
#include <stdio.h>

const int N= 4;
int sor[4];
int veg= -1;
int kezdet= 0;
int elemSzam= 0;

void put(int x)
{
  if ( elemSzam <= N - 1 )
  {
    veg= (veg + 1) % N;
    sor[veg]= x;
    ++elemSzam;
  }
  else
  {
    printf("A sor tele van!\n");
  }
}

int get()
{
  if ( elemSzam > 0 )
  {
    kezdet= (kezdet + 1) % N;
    --elemSzam;
  }
  else
  {
    printf("A sor Ăźres!\n");
    return -1;
  }
  return sor[(kezdet-1)%N];
}

int main( int argc, char** argv)
{
  int i;
  int a= 2;
  put(3);
  put(a*a);
  put(a + 2);
  put(a % 2);
  a= get();
  get();
  put(a);

  for ( i= 0; i < N; ++i )
  {
    if ( i == kezdet )
      printf(".");
    if ( i == veg )
      printf("_");
    printf("%d ", sor[i]);
  }
  return 0;
}
```

A program kimenete futtatás után:

```
_3 4 .4 0
```

11. Verem

A verem a sorhoz hasonló nagyon egyszerű adatszerkezet. Legfontosabb műveletei: **FONTOS**

- verem ürességének ellenőrzése,
- verem felső elemének elérése (TOP),
- új elem hozzáadása a veremhez (PUSH),
- felső elem eltávolítása (POP),
- (a POP művelet magában foglalhatja a TOP műveletet is, azaz visszaadhatja a verem legfelső elemét).

A verem a sorhoz hasonlóan dinamikus adatszerkezet, tehát elméletben tetszőleges sok elemet tartalmazhat, az egydimenziós tömbökkel történő megvalósításban azonban a tömb mérete korlátozni fogja a verem méretét. A verem LIFO adatszerkezet, azaz **L**ast **I**n **L**ast **O**ut: tehát az utoljára berakott elem kerül ki a veremből először.

Szemléletesen a verem egy tényleges veremként, azaz gödörként képzelhető el, tárgyakat teszünk a verembe, egymás tetejére, és amikor ki akarunk venni egy elemet, mindig csak a legfelsőt tudjuk kivenni, tehát azt, amit utoljára tettünk bele.

A vermet is globális változón keresztül valósítjuk meg, a megvalósításhoz használt tömbön kívül egyetlen változóra van csak szükségünk, amely a verem tetejének indexét, azaz az utoljára berakott elem indexét fogja tartalmazni. A vermet mindig a tömb vége felé bővítjük.

Létrehozás.

```
const int N= 20;
int verem[20];
int teteje= -1;
```

Egy üres verem tetejének indexe tehát `-1`.

Üresség ellenőrzése.

```
int uresVerem()
{
  return teteje == -1;
}
```

A verem csak akkor lehet üres, ha nincs benne egyetlen elem sem, tehát a tetejének indexe `-1`.

Felső elem elérése. (TOP)

```
int top()
{
  return verem[teteje];
}
```

Látható, hogy a top művelet egy egyszerű tömb elem elérés, ahogy a sorok esetén is.

Új elem hozzáadása. (PUSH) Az alábbi `push` eljárás a paraméterként kapott `x` értéket fogja berakni a verem tetejére, ha van még hely a tömbben.

```
void push(int x)
{
  if ( teteje != N - 1 )
  {
    ++teteje;
    verem[teteje]= x;
  }
  else
    printf("A verem tele van!\n");
}
```

Felső elem eltávolítása. (POP) A `pop` műveletben egyszerűen csak csökkentjük a teteje változó értékét, abban az esetben, ha az nem `-1`, tehát nem üres a verem.

```
int pop()
{
  if ( teteje != -1 )
    --teteje;
  return verem[teteje+1];
}
```

11.1. Példa

A következő táblázatban néhány egyszerű művelet során vizsgáljuk, hogyan változik a verem tartalma és a `teteje` változó, ha a verem egy 4 elemű tömbbel kerül reprezentálásra.

Művelet	Verem tartalma	`teteje`
put(3)	3 _ _ _	0
get()	_ _ _ _	-1
put(2)	2 _ _ _	0
put(3)	2 3 _ _	1
put(2)	2 3 2 _	2
put(3)	2 3 2 3	3
get()	2 3 2 _	2
get()	2 3 _ _	1

A következő egyszerű példaprogramban a verem és a sor fő felhasználását szemléltetem. A sort leggyakrabban egyfajta bufferként használják, azaz ha valamilyen beérkező adathalmazt a beérkezés sorrendjében kell feldolgozni, akkor átmenetileg egy sor adatszerkezetben lehet tárolni őket. Ha az egyesével érkező adatokat a beérkezés sorrendjével ellenkező sorrendben kell feldolgozni, akkor viszont a verem adatszerkezetet alkalmazzák. Az alábbi példaprogramban a felhasználótól egész számokat kérünk a `-1` értékig, és minden egész szám megadása után a beolvasott értéket berakjuk egy verembe. A verem maximális mérete `20` lesz. A beolvasást követően kiürítjük az adatszerkezetet úgy, hogy a megfelelő függvényeit használjuk az elemek elérésére és eltávolítására.

11.1.1. forráskód: verem.c

```
#include <stdio.h>

const int N= 20;

int verem[20];
int teteje= -1;
```

```
int uresVerem()
{
  return teteje == -1;
}

int topVerem()
{
  return verem[teteje];
}

void push(int x)
{
  if ( teteje != N - 1 )
  {
    ++teteje;
    verem[teteje]= x;
  }
  else
    printf("A verem tele van!\n");
}

int pop()
{
  if ( teteje != -1 )
    --teteje;
  return verem[teteje+1];
}

int main(int argc, char** argv)
{
  int szam;

  do
  {
    printf("kerek egy szamot:\n");
    scanf("%d", &szam);

    if ( szam == -1 )
      break;

    push(szam);
  }while ( 1 );

  printf("a verem elemei:\n");
  while ( ! uresVerem() )
  {
    printf("%d ", topVerem());
    pop();
  }

  return 0;
}
```

A példa bemenet és kimenet:

```
kerek egy szamot:
2
kerek egy szamot:
4
kerek egy szamot:
3
kerek egy szamot:
6
kerek egy szamot:
-1
a verem elemei:
```

```
6 3 4 2
```

11.1. Feladat: Mi lesz a 3 elemű tömbbel reprezentált verem tartalma és a `teteje` változó értéke az alábbi műveletek végrehajtása után?

- push(3);
- push(2);
- push(2);
- pop();

11.1. Megoldás: A verem tartalma `3, 2` lesz, a `teteje` változó értéke pedig `1` lesz.

11.2. Feladat: Mi lesz a 4 elemű tömbbel reprezentált verem tartalma és a `teteje` változó értéke az alábbi műveletek végrehajtása után?

```
int a= 2;
push(3*a);
push(a);
push(0);
a= pop();
push(pop());
push(a + pop());
```

11.2. Megoldás: A harmadik `push` utasítás után a verem tartalma (6, 2, 0; `teteje`= 2). Ezt követően arra kell ügyelnünk, hogy a paraméterlistán szereplő kifejezés mindig, minden körülmények között kiértékelődik a paraméterátadás előtt, tehát a paraméterlistán lévő művelet hamarabb kerül végrehajtásra, mint az a függvény/eljárás, amelyiknek a paraméterlistáján szerepel. Ennek megfelelően a `a=pop();` utasítás egyszerűen eltávolítja a felső elemet (0), és berakja annak értékét az `a` változóba. A következő, `push(pop());` utasítás kiveszi a felső elemet (2), majd vissza is rakja azt a verembe a `push` művelettel, tehát a verem lényegében nem változik. Az utolsó utasítás ismét kiveszi a felső elemet (2), majd visszarakja annak és az `a` változónak az összegét. Mivel az `a` értéke megint csak `0`, ebben a speciális esetben a verem ismét nem változik, azaz a verem tartalma (6, 2; `teteje`= 1).

12. Rekord

Definíció szerint a rekord heterogén, statikus adatszerkezet. Heterogenitása arra utal, hogy a legkülönbözőbb típusú adatokat tartalmazhatja, szemben a tömbbel, amely mindig csak egy bizonyos típusú adatot tartalmazhatott. Statikussága a szokásos tulajdonság: egy rekord mérete nem változtatható. A rekordot a gyakorlatban számtalan helyen alkalmazzuk, valahányszor össze szeretnénk fogni összetartozó adatokat egy egységgé, egy változóvá. Egy rekord definiálása minden esetbe egy új, saját adattípus létrehozását jelenti, tehát valahányszor rekordot hozunk létre, azaz definiálunk, mindig létrejön egy új típus, amelyet mindenhol használhatunk a programunkban, ahol eddig atomi típus szerepelt. Rekord definíciójának szintaktikája az alábbi:

```
struct azonosito
{
  tipus azonosito;
  [tipus azonosito;]...
};
```

A `struct` alapszó után szereplő azonosító lesz az új, rekord típus neve. A blokkban szereplő utasítások deklarációs utasítások, a blokkban szereplő azonosítókkal létrehozott változókat a rekord típus mezőinek nevezzük. A deklarációs utasítások összevonhatóak, azaz ha azonos típusú mezőket szeretnénk létrehozni, akkor felsorolhatjuk a mezőneveket a megfelelő típus egyszeri kiírása után.

Nagyon fontos, hogy a `;`-t ne hagyjuk le a struktúra definíció végéről! Miután definiáltunk egy ilyen struktúrát, ami a C nyelv beépített rekord típusa, tetszőleges helyen használhatjuk a `struct azonosito` kifejezést, ahol eddig atomi típust használtunk. Miután deklaráltunk egy ilyen struktúra típusú változót, az lényegében egy változóban fogja össze a mezőit, mint egyfajta "alárendelt" vagy al-változókat, és azokat a `.` minősítő operátorral érhetjük el. Lássunk egy példát arra, hogyan hozhatunk létre egy `pont2D` nevű struktúrát, amely az Euklideszi sík egy egész koordinátájú pontját hivatott reprezentálni.

```
struct pont2D
{
  int x;
  int y;
}
```

Ezt követően a kódunkban bárhol létrehozhatunk `struct pont2D` típusú változót, például az alábbi deklarációs utasítással:

```
struct pont2D p;
```

amelyben a `p` változót hoztuk létre. A változónak adhatunk kezdőértéket, úgy, ahogy tömbök esetén is, azaz kapcsoszárójelben felsoroljuk a mezők értékét, a mezők definícióbeli sorrendjének megfelelően:

```
struct pont2D p= {1, 2};
```

A fenti utasítás hatására a `p` változó `x` mezőjének értéke `1` lesz, míg a `y` mező értéke `2`. A `.` operátorral érhetjük el egy `struct pont2D` típusú változó mezőit, amelyet aztán úgy használhatunk, mint bármely más változót, tehát kifejezésekben, átadhatjuk paraméterként, és értékadás bal oldalán is állhat. Az alábbi kódrészletben megváltoztatom a `p` változó `x` és `y` mezőit `0`-ra, majd kiírom a tartalmukat a konzolra, ezt követően beolvasok két egész számot a bemenetről a `scanf` függvénnyel, egyenesen a `p` `x` és `y` mezőibe, majd kiírom ismét a mezők tartalmát.

```
p.x= 0;
p.y= 0;
printf("%d %d\n", p.x, p.y);
scanf("%d %d", &(p.x), &(p.y));
printf("%d %d\n", p.x, p.y);
```

12.1. Feladat: Hozzon létre rekord típust, amely az Euklideszi-tér három dimenziós (lebegőpontos koordinátákat is felvehető) pontjaink ábrázolására használható!

12.1. Megoldás:

```
struct pont3D
{
  float x, y, z;
};
```

12.2. Feladat: Hozzon létre rekord típust, amely hallgatók információit tartalmazhatja, 6 karakteres sztringben a neptunkódját és egész típusú változóban a születési évét!

12.2. Megoldás:

```
struct hallgato
{
  char neptunkod[7];
  int szul_ev;
};
```

Ne felejtsük el, hogy 6 karakteres sztring tárolásához 7 elemű karaktertömböt kell létrehoznunk, a sztringet záró `'\0'` karakter miatt.

12.3. Feladat: Hozzon létre egy `struct hallgato` típusú változót és adjon neki tetszőleges kezdőértéket!

12.3. Megoldás:

```
struct hallgato h= {"ABC123", 1985};
```

12.4. Feladat: Írjon függvényt, amely paraméterként kapja egy három dimenziós objektum pontjait egy `pont3D` típusú elemeket tartalmazó tömbként és visszatérési értékként egy `pont3D` típusú értéket ad vissza, amely az objektum súlypontját tartalmazza. (Egy ponthalmaz súlypontjának koordinátáit úgy kaphatjuk meg, hogy a halmaz pontjainak koordinátáit koordinátánként átlagoljuk.)

12.4. Megoldás:

```
struct pont3D sulypont(struct pont3D t[], int n)
{
  int i;
  struct pont3D sp= {0, 0, 0};

  for ( i= 0; i < n; ++i )
  {
    sp.x+= t[i].x;
    sp.y+= t[i].y;
    sp.z+= t[i].z;
  }

  sp.x/= n;
  sp.y/= n;
  sp.z/= n;

  return sp;
}
```

Az eddig bemutatott algoritmusok értelmét sokan megkérdőjelezhetik, hiszen mire használható az a gyakorlatban, hogy egész számokat tartalmazó tömbökben keresünk, vagy egész számokat tartalmazó tömböket rendezünk, egész számokat rakunk sor és verem adatszerkezetbe, stb. A korábbi algoritmusok demonstrálására az egészeket tartalmazó tömbök voltak a legkézenfekvőbbek. A továbbiakban megnézzük, hogy hogyan írhatunk ténylegesen használható programokat, illetve hogyan implementálhatjuk az eddigi algoritmusainkat elegánsabban a rekordok felhasználásával.

12.1. Keresés

12.5. Feladat: Írjon függvényt, amely paraméterként kapja `struct hallgato` típusú objektumok tömbjét valamint egy neptun kódot (sztringként), és teljes keresést megvalósítva visszaadja azon tömb elem indexét, amely a paraméterként kapott neptun-kódú hallgatóhoz tartozik, ha nincs benne a tömbben, -1-et ad vissza.

12.5. Megoldás:

```
int teljesKereses(struct hallgato t[], int n, char nk[])
{
  int i;
  for ( i= 0; i < n; ++i )
    if ( strcmp(t[i].neptunkod, nk) == 0 )
      return i;
  return -1;
}
```

Hasonlítsuk össze a fenti kódot az egész számok tömbjében teljes keresést végző függvénnyel. Jól látható, hogy maga az algoritmus ugyanaz, a különbség csak a paraméterezésben (hiszen most `struct hallgato` típusú adatok között keresünk), és a hasonlításban van, amelyhez mivel sztringekkel dolgozunk, a `strcmp` függvényt kell használnunk.

12.6. Feladat: Írjon programot, amely létrehoz egy 5 elemű, `struct hallgato` típusú változókat tartalmazó tömböt, feltölti azt a konzolról a `scanf` függvénnyel, majd neptun kódokat olvas be egészen addig és kiírja a megfelelő hallgató születési évét egészen addig, amíg a felhasználó a "vege" sztringet nem gépeli be neptun kód helyett!

12.6. Megoldás:

12.1.1. forráskód: hallgatokereses.c

```
#include <stdio.h>

struct hallgato
{
  char neptunkod[7];
  int szul_ev;
};

int teljesKereses(struct hallgato t[], int n, char nk[])
{
  int i;
  for ( i= 0; i < n; ++i )
    if ( strcmp(t[i].neptunkod, nk) == 0 )
      return i;
  return -1;
}

int main(int argc, char** argv)
{
  struct hallgato h[5];
  char tmp[7];
  int i, j;

  for ( i= 0; i < 5; ++i )
    scanf("%s %d", h[i].neptunkod, &(h[i].szul_ev));

  while ( 1 )
  {
    printf("kerek egy neptunkodot:\n");
    scanf("%s", tmp);
    if ( strcmp(tmp, "vege") == 0 )
      break;

    j= teljesKereses(h, 5, tmp);
    if ( j == -1 )
      printf("ismeretlen neptun-kod\n");
    else
      printf("%d\n", h[j].szul_ev);
```

```
  }

  return 0;
}
```

12.7. Feladat: Az eljárásorientált szemlélet gyakorlásaként emeljük ki a `h` tömb feltöltését és a kiírást egy-egy eljárásba!

12.7. Megoldás:

12.1.2. forráskód: hallgatokereses2.c

```
#include <stdio.h>

struct hallgato
{
  char neptunkod[7];
  int szul_ev;
};

int teljesKereses(struct hallgato t[], int n, char nk[])
{
  int i;
  for ( i= 0; i < n; ++i )
    if ( strcmp(t[i].neptunkod, nk) == 0 )
      return i;
  return -1;
}

void feltolt(struct hallgato t[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    scanf("%s %d", t[i].neptunkod, &(t[i].szul_ev));
}

void kiir(struct hallgato t[], int n, char nk[])
{
  int x;
  x= teljesKereses(t, n, nk);
  if ( x == -1 )
    printf("ismeretlen hallgato\n");
  else
    printf("%d\n", t[x].szul_ev);
}

int main(int argc, char** argv)
{
  struct hallgato h[5];
  char tmp[7];

  feltolt(h, 5);

  while ( 1 )
  {
    printf("kerek egy neptunkodot:\n");
    scanf("%s", tmp);
    if ( strcmp(tmp, "vege") == 0 )
      break;

    kiir(h, 5, tmp);
  }

  return 0;
```

```
}
```

12.8. Feladat: Alakítsuk át most a kódot a paraméterátadásokat globális változókon keresztül egyszerűsítve!

12.8. Megoldás:

12.1.3. forráskód: hallgatokereses3.c

```
#include <stdio.h>

struct hallgato
{
  char neptunkod[7];
  int szul_ev;
};

struct hallgato h[5];
char nk[7];
int n= 5;

int teljesKereses()
{
  int i;
  for ( i= 0; i < n; ++i )
    if ( strcmp(h[i].neptunkod, nk) == 0 )
      return i;
  return -1;
}

void feltolt()
{
  int i;
  for ( i= 0; i < n; ++i )
    scanf("%s %d", h[i].neptunkod, &(h[i].szul_ev));
}

void kiir()
{
  int x;
  x= teljesKereses();
  if ( x == -1 )
    printf("ismeretlen hallgato\n");
  else
    printf("%d\n", h[x].szul_ev);
}

int main(int argc, char** argv)
{
  char tmp[5];

  feltolt();

  while ( 1 )
  {
    printf("kerek egy neptunkodot:\n");
    scanf("%s", tmp);
    if ( strcmp(tmp, "vege") == 0 )
      break;

    kiir();
  }

  return 0;
```

```
}
```

Látható, hogy a főprogram az első megoldáshoz képest meglehetősen leegyszerűsödött, a beszédes függvény és eljárásnevekből könnyen kitalálható ránézésre a program működése, a függvények és eljárások szintén egyszerűek, egy-egy néhány lépéses feladatot valósítanak meg. Megjegyezzük, hogy a második megoldás általánosabb a paraméterezhető függvények és eljárások miatt.

12.2. Rendezés

12.9. Feladat: Írjon eljárást, amely egy `struct hallgato` típusú elemekből álló tömböt születési év szerint rendez. A rendezés a minimum kiválasztásos növekvő rendezés legyen!

12.9. Megoldás:

```
void rendezSzulEv(struct hallgato t[], int n)
{
  struct hallgato tmp;
  int i, j, min;

  for ( i= 0; i < n; ++i )
  {
    min= i;
    for ( j= i+1; j < n; ++j )
      if ( t[j].szul_ev < t[min].szul_ev )
        min= j;
    tmp= t[min];
    t[min]= t[i];
    t[i]= tmp;
  }
}
```

Összehasonlítva az egészeket rendező kóddal, könnyű látni, hogy különbség csak a paraméterezésben, a `tmp` segédváltozó típusában és a hasonlításban van.

12.10. Feladat: Írjon eljárást, amely egy `struct hallgato` típusú elemekből álló tömböt neptun-kód szerint rendez. A rendezés a minimum kiválasztásos növekvő rendezés legyen!

12.10. Megoldás: Mivel a neptun-kód mező sztring típusú, a hasonlításhoz a `strcmp` függvényt kell használnunk!

```
void rendezNK(struct hallgato t[], int n)
{
  struct hallgato tmp;
  int i, j, min;

  for ( i= 0; i < n; ++i )
  {
    min= i;
    for ( j= i+1; j < n; ++j )
      if ( strcmp(t[j].neptunkod, t[min].neptunkod) < 0 )
        min= j;
    tmp= t[min];
    t[min]= t[i];
    t[i]= tmp;
  }
}
```

Jól látható, hogy a ez a megoldás a születési év szerint rendező kódtól egyedül a rendezési feltételben tér el, ahol felhasználtuk azt, hogy a `strcmp` függvény negatív értéket ad vissza, ha az első paramétere alfabetikus rendezés szerint kisebb, mint a második.

12.11. Feladat: Módosítsa a születési év szerint rendező eljárást úgy, hogy az azonos születési évhez tartozó elemek neptun kód szerinti növekvő sorrendbe legyenek rendezve!

12.11. Megoldás: A megoldáshoz egyedül a rendezési feltételt kell bővítenünk. Értelemszerűen most akkor lesz kisebb egy elem a másiknál, ha kisebb a születési éve, vagy megegyezik a születési év, de kisebb a neptun kód értéke alfabetikusan:

```
void rendezSzulEvNK(struct hallgato t[], int n)
{
  struct hallgato tmp;
  int i, j, min;

  for ( i= 0; i < n; ++i )
  {
    min= i;
    for ( j= i+1; j < n; ++j )
      if ( t[j].szul_ev < t[min].szul_ev || (t[j].szul_ev == t[min].szul_ev && strcmp(t[
          j].neptunkod, t[min].neptunkod) == 0 ))
        min= j;
    tmp= t[min];
    t[min]= t[i];
    t[i]= tmp;
  }
}
```

12.12. Feladat: Írjon eljárást, amely paraméterként kap egy hallgatók adatait tartalmazó `struct hallgato` típusú tömböt, valamint annak méretét, és kiírja a tömb elemeit azok tárolási sorrendjében a konzolra!

12.12. Megoldás:

```
void kiir(struct hallgato t[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    printf("%s %d\n", t[i].neptunkod, t[i].szul_ev);
}
```

12.13. Feladat: Írjon programot, amely feltölt egy 5 elemű tömböt hallgatók adataival, majd rendezi a tömböt előbb születési év, majd neptun-kód, majd születési év és neptun kód szerint, és kiírja mindhárom rendezés után a tömb tartalmát a konzolra!

12.13. Megoldás:

12.2.4. forráskód: hallgatorendez.c

```
#include <stdio.h>

struct hallgato
{
  char neptunkod[7];
  int szul_ev;
};

void rendezSzulEv(struct hallgato t[], int n)
```

```
{
  struct hallgato tmp;
  int i, j, min;

  for ( i= 0; i < n; ++i )
  {
    min= i;
    for ( j= i+1; j < n; ++j )
      if ( t[j].szul_ev < t[min].szul_ev )
        min= j;
    tmp= t[min];
    t[min]= t[i];
    t[i]= tmp;
  }
}

void rendezNK(struct hallgato t[], int n)
{
  struct hallgato tmp;
  int i, j, min;

  for ( i= 0; i < n; ++i )
  {
    min= i;
    for ( j= i+1; j < n; ++j )
      if ( strcmp(t[j].neptunkod, t[min].neptunkod) < 0 )
        min= j;
    tmp= t[min];
    t[min]= t[i];
    t[i]= tmp;
  }
}

void rendezSzulEvNK(struct hallgato t[], int n)
{
  struct hallgato tmp;
  int i, j, min;

  for ( i= 0; i < n; ++i )
  {
    min= i;
    for ( j= i+1; j < n; ++j )
      if ( t[j].szul_ev < t[min].szul_ev || (t[j].szul_ev == t[min].szul_ev && strcmp(t[
           j].neptunkod, t[min].neptunkod) == 0 ))
        min= j;
    tmp= t[min];
    t[min]= t[i];
    t[i]= tmp;
  }
}

void kiir(struct hallgato t[], int n)
{
  int i;
  for ( i= 0; i < n; ++i )
    printf("%s %d\n", t[i].neptunkod, t[i].szul_ev);
}

int main(int argc, char** argv)
{
  struct hallgato t[5];
  int i;

  for ( i= 0; i < 5; ++i )
  {
```

```
    printf("adjon meg egy neptun-kodot es szuletesi evet:\n");
    scanf("%s %d", t[i].neptunkod, &(t[i].szul_ev));
  }

  rendezSzulEv(t, 5);
  printf("szuletesi ev szerint rendezve:\n");
  kiir(t, 5);
  rendezNK(t, 5);
  printf("neptun-kod szerint rendezve:\n");
  kiir(t, 5);
  rendezSzulEvNK(t, 5);
  printf("szuletesi ev es neptun-kod szerint rendezve:\n");
  kiir(t, 5);

  return 0;
}
```

A program futásának egy lehetséges ki és bemenete:

```
djon meg egy neptun-kodot es szuletesi evet:
asdfas 1991
adjon meg egy neptun-kodot es szuletesi evet:
ababab 1991
adjon meg egy neptun-kodot es szuletesi evet:
aaabbb 1991
adjon meg egy neptun-kodot es szuletesi evet:
cddefg 1990
adjon meg egy neptun-kodot es szuletesi evet:
alaman 1991
szuletesi ev szerint rendezve:
cddefg 1990
ababab 1991
aaabbb 1991
asdfas 1991
alaman 1991
neptun-kod szerint rendezve:
aaabbb 1991
ababab 1991
alaman 1991
asdfas 1991
cddefg 1990
szuletesi ev es neptun-kod szerint rendezve:
cddefg 1990
ababab 1991
alaman 1991
asdfas 1991
aaabbb 1991
```

12.3. Ritka mátrix

A ritka mátrix ábrázolásának három és négy soros megvalósítása meglehetősen kényelmetlen volt, ugyanis a három és négy sort egy-egy tömbben tartottuk nyilván. Rekordok alkalmazásával sokkal elegánsabb megoldást kaphatunk, ugyanis a három, illetve négy sor összetartozó (egy oszlopban lévő) adatait összefoghatjuk egy saját rekord típusba, és ezt követően a három soros reprezentáció már egy tömbbel megadható.

12.14. Feladat: Hozzon létre rekord típust, amely a háromsoros reprezentáció összetartozó adatainak összefogására alkalmas!

12.14. Megoldás:

```
struct sor3
{
  int sor, oszlop, ertek;
};
```

12.15. Feladat: Írja át a három soros reprezentációt felépítő függvényt és az három soros reprezentáció egy elemét elérő függvényt úgy, hogy azok `struct sor3` típusú tömböket várjanak paraméterként!

12.15. Megoldás:

```
int felepit3sor(int t[], int sorok, int oszlopok, int gyakoriElem, int sor3[])
{
  int i, j, k= 0;
  for ( i= 0; i < sorok; ++i )
    for ( j= 0; j < oszlopok; ++j )
      if ( t[i*oszlopok + j] != gyakoriElem )
      {
        sor3[k].sor= i;
        sor3[k].oszlop= j;
        sor3[k].ertek= t[i*oszlopok + j];
        ++k;
      }
  return k;
}

int elem(int sor3[], int meret, int gyakoriElem, int s, int o)
{
  int i;
  for ( i= 0; i < meret; ++i )
    if ( sor3[i].sor == s && sor3[i].oszlop == o )
      return sor3[i].ertek;
  return gyakoriElem;
}
```

12.4. Sor

A sor adatszerkezet megvalósítása sem a lehető legelegánsabb, ugyanis globális változókat használunk, így függvényeink csak egyetlen sor kezelésére képesek a programban.

12.16. Feladat: Hozzunk létre egy saját adattípust, amely alkalmas arra, hogy egy 10 elemű tömbben tárolt sort reprezentáljon (legyen benne egy 10 elemű tömb, valamint egy `vege` mező), majd alakítsuk át a rögzített sort kezelő függvényeket és eljárásokat úgy, hogy a sort, amelyen dolgozniuk kell, mindig megkapják paraméterként!

12.16. Megoldás: A sort reprezentáló struktúra:

```
struct sor
{
  int t[10];
  int vege;
};
```

Az üres sort létrehozó függvény:

```
struct sor ujSor()
{
  struct sor t;
  t.vege= -1;
  return t;
}
```

A sor ürességét ellenőrző függvény:

```
int uresSor(struct sor s)
{
  return s.vege == -1;
}
```

A sorba történő beszúrást megvalósító függvény:

```
struct sor put(struct sor s, int x)
{
  if ( s.vege == 9 )
    printf("a sor tele van\n");
  else
  {
    ++s.vege;
    s.t[s.vege]= x;
  }
  return s;
}
```

A sor első elemének elérését megvalósító függvény:

```
int top(struct sor s)
{
  return s.t[0];
}
```

A sor első elemét eltávolító függvény:

```
struct sor get(struct sor s)
{
  if ( s.vege == -1 )
    printf("a sor üres\n");
  else
    --s.vege;
  return s;
}
```

A sor tartalmát kiíró eljárás:

```
void kiir(struct sor s)
{
  int i;
  for ( i= 0; i <= s.vege; ++i )
    printf("%d ", s.t[i]);
  printf("\n");
}
```

A fenti függvényekben a `get` eljárás abban különbözik, a korábban haszálttól, hogy nem adja vissza visszatérési értékként a sor első elemét, ezzel szemben magát a sor reprezentáló adatszerkezetet adja vissza. Ennek a megközelítésnek azaz oka, hogy valahányszor megváltozik a sor (`put` és `get` műveletek), az érték szerinti paraméterátadással átvett, sort reprezentáló struktúrát vissza kell juttatni a hívási környezetbe és értékül kell adni annak a paraméternek, amellyet a paraméterlistáján átadtunk neki, ahhoz hogy a módosítás ott is érvénybe lépjen. A paraméterként átvett sor ebben az esetben NEM az az objektum, amelyre meghívtuk, hanem annak másolata!

12.17. Feladat: Írjon programot, amely a fenti függvények felhasználásával két sort kezel: `a`-t és `b`-t. A felhasználó parancsokat ad meg, és a program elvégzi a megfelelő soron a megfelelő műveletet! A lehetséges parancsok a `put` és a `get`, a parancsot minden esetben egy karakter követi, `'a'` az `a` sorra utal, `'b'` a `b` sorra, `put` esetén további paraméter a szám, amelyet be szeretnénk szúrni. Például a

`put a 2` parancs határása bekerül az `a` sorba a 2 érték. A `get b` parancs eltávolítja a `b` sor legelső elemét. Amikor a felhasználó a "vege" sztringet gépeli be, a program kiírja a sorok tartalmát és befejezi futását.

12.17. Megoldás:

12.4.5. forráskód: sorvezerlovel.c

```
#include <stdio.h>

struct sor
{
  int t[10];
  int vege;
};

struct sor ujSor()
{
  struct sor t;
  t.vege= -1;
  return t;
}

int uresSor(struct sor s)
{
  return s.vege == -1;
}

struct sor put(struct sor s, int x)
{
  if ( s.vege == 9 )
    printf("a sor tele van\n");
  else
  {
    ++s.vege;
    s.t[s.vege]= x;
  }
  return s;
}

int top(struct sor s)
{
  return s.t[0];
}

struct sor get(struct sor s)
{
  if ( uresSor(s) )
    printf("a sor Ăźres\n");
  else
    --s.vege;
  return s;
}

void kiir(struct sor s)
{
  int i;
  for ( i= 0; i <= s.vege; ++i )
    printf("%d ", s.t[i]);
  printf("\n");
}

int main(int argc, char** argv)
{
  struct sor a, b;
```

```
  char tmp[5];
  char c;
  int x, par;

  a= ujSor();
  b= ujSor();

  while ( 1 )
  {
    par= scanf("%s %c %d", tmp, &c, &par);

    if ( strcmp(tmp, "vege") == 0 )
      break;
    if ( strcmp(tmp, "put") == 0 )
    {
      if ( c == 'a' )
        a= put(a, x);
      else if ( c == 'b' )
        b= put(b, x);
      else
        printf("ismeretlen sor azonosito\n");
    }
    else if ( strcmp(tmp, "get") == 0 )
    {
      if ( c == 'a' )
        a= get(a);
      else if ( c == 'b' )
        b= get(b);
      else
        printf("ismeretlen sor azonosito\n");
    }
    else
      printf("ismeretlen utasitas\n");
  }

  printf("az 'a' sor tartalma:\n");
  kiir(a);
  printf("a 'b' sor tartalma:\n");
  kiir(b);

  return 0;
}
```

A program kimenete a megfelelő bemenet esetén:

```
put a 2
put a 3
put a 4
get a
put b 1
put b 2
get a
vege
az 'a' sor tartalma:
2
a 'b' sor tartalma:
1 2
```

Természetesen a struktúrák felhasználásával sorokban is tárolhatunk tetszőleges összetettségű saját típusokat, tehát az előző példákhoz hasonlóan akár hallgatók adatait is.

12.18. Feladat: Készítse el a vándorló sor, a ciklikus sor és a verem adatszerkezeteket a rögzített sorhoz hasonló, az adatszerkezetet struktúrával reprezentáló megközelítéssel!

13. Dinamikus memóriakezelés

Eddigi programjainknak és adatszerkezeteinknek közös hátránya, hogy megvalósításukhoz tömböt használtunk, tömbből indultunk ki, a tömbök pedig definíció szerint statikus adatszerkezetek ami azt jelenti, hogy méretük rögzített, meg nem változtatható, így ez a méret felső határt szab minden eddigi adatszerkezet maximális elemszámának.

Vegyük észre tovább, hogy a következő problémát eddigi eszközeinkkel nem tudjuk megoldani: Írjon programot, amely a bemenetről beolvas egy egész számot (n), majd n darab egész számot és fordított sorrendben kiírja azokat.

Azért nem tudjuk ezt megoldani eddigi eszközeinkkel, mert ahhoz hogy fordított sorrendben ki tudjuk írni az elemeket, tárolnunk kell azokat, MINDET, azonban úgy, hogy nem tudjuk, az n maximális méretét, nem mondhatjuk azt, hogy egy N méretű tömb elég lesz, mert $n = N + 1$ elemet már rögtön nem tudunk tárolni. Az lenne a jó, ha a program futása közben tudnánk meghatározni azt, hogy pontosan mekkora memóriaterületekkel, tömbökkel szeretnénk dolgozni. Amikor a memóriában lévő adatszerkezetek méretét dinamikus kezeljük, dinamikus memóriakezelésnek nevezzük. A dinamikus memóriakezelés első lépéseként a mutatók használatával/jelentőségével/lehetőségeivel kell tisztában lennünk.

FONTOS A *mutató* olyan változó, amely memóriacímet tartalmazhat. Jegyezzük meg, hogy egy memóriacímmel önmagában semmit sem tudunk kezdeni. Ahogy azt programozás gyakorlaton tanulta mindenki, ahhoz, hogy egy, a memóriában lévő adattal dolgozni tudjunk, a címén kívül tudnunk kell azt is, hogy pontosan hány bájton van ábrázolva az adat és azt is, hogy pontosan milyen ábrázolással. Azonban az ábrázolás és a tárolásra használt bájtok száma a *típus*okhoz van rendelve, így tehát mutatók használata esetén tudnunk kell annak az adatnak a típusát, amelyet ábrázolni szeretnénk. Mindezt összefoglalva, mutatókat az alábbi deklarációs utasítással hozhatunk létre:

```
tipus * azonosito;
```

ahol a `*` jelzi azt, hogy az azonosító nem egy `tipus` típusú értéket, hanem egy olyan memóriacímet tartalmaz, amelyen egy `tipus` ábrázolású érték található. Példák mutatók létrehozására:

```
char* character_mutato;
float* float_mutato;
struct hallgato* hallgato_mutato;
```

Jegyezzük meg, hogy mivel a mutatók minden esetben egy memóriacímet tartalmaznak, azok mérete minden esetben a gépi szó hossz, azaz például 32 bites architektúrán a fenti három mutató mérete egyaránt 4-4 bájt. 64 bites architektúrán a fenti mutatók mérete 8-8 bájt. Mutatók használatához két kérdést kell még tisztáznunk:

- Milyen értékeket adhatunk mutatóknak?
- Hogyan kezelhetjük a mutatók által címzett adatokat?

Az első kérdés kapcsán jegyezzük meg, hogy egy mutatónak sohasem adhatunk explicit módon értéket, azaz sohasem tehetjük meg azt, hogy `character_mutato= 2342`. Ennek az az oka, hogy a programok írásakor nincs róla tudomásunk, hogy a program a memóriának pontosan melyik részén fog elhelyezkedni, ezért explicit módon nem dolgozhatunk címekkel! Egyetlen speciális érték van, amelyet egy mutatóhoz hozzárendelhetünk, ez pedig a `NULL` mutató, ami azt jelenti, hogy a mutató sehová sem mutat. Mutatók létrehozásakor, azaz deklarálásakor azok értéke határozatlan, ezért a memória egy véletlenszerű részét címzik, amelyet nem használhatunk, hiszen ha egy memória területet használni szeretnénk, azt jeleznünk kell előbb az operációs rendszernek. A mutató változókat érdemes tehát a `NULL` értékkel inicializálni, jelezve azt, hogy sehová sem mutatnak. Ha explicit módon nem adhatunk értéket a mutatóknak, akkor hogyan tehetjük meg? Jegyezzük meg, hogy az operációs rendszer minden változó számára fenntart egy memóriaterületet, ezért például ha egy változó címkomponensét, azaz címét le tudnánk kérdezni, akkor ezt a címet már értékül adhatnánk

egy megfelelő mutató típusú változónak. Egy címkomponenssel rendelkező programozási eszköz címét az `&` ('címe') operátorral kérdezhetjük le. Egy mutató által címzett objektumot a mutató elé írt `*` ('indirekciós') operátorral érhetünk el. Az indirekciós operátort használva például egy `int* p;` mutatón, a `*p` kifejezés a kód bármely pontján egy `int` típusú értéket jelent, amelynek a címe éppen a `p`-ben tárolt cím.

```
int a= 5;
int *p;
p= &a;
*p= 2;
printf("%d", a);
```

A fenti kódrészlet hatására a képernyőn az 'elvárt' 5-ös értékkel szemben a 2 érték fog megjelenni. Ennek az az oka, hogy a `p` mutató változóba beletettük az `a` változó címét, így a `*p=2` értékadás éppen a `p`-ben található címen lévő adat értékét fogja 2-re állítani. A `p`-ben lévő címen azonban éppen az `a` változó található, hiszen az `a` címkomponensét tettük bele a `p`-be, így az értékadás az `a` változót írja át 2-re. Ezek alapján mutatóknak értékül adhatjuk címkomponenssel rendelkező változók címét. Ez azonban még mindig nem oldja meg azt a problémát, hogy hogyan tudunk futás közben, változó méretű memóriaterületeket lefoglalni. Erre ad megoldást egy beépített, a `stdlib.h` header-ben található standard könyvtári függvény, a `malloc`. A `malloc` függvény specifikációja a következő:

```
void* malloc(int n);
```

A `malloc` függvény működése a következő: paraméterként kapja a lefoglalandó memóriaterület méretét bájtokban, és visszatérési értékként visszaadja egy n méretű memóriaterület címét, ha sikerül lefoglalnia, vagy a `NULL` mutatót, ha nem sikerül a memóriafoglalás. Lássuk, a gyakorlatban hogyan használható a függvény például 10 darab egész (`int`) érték tárolására alkalmas memóriaterületet:

```
int *p;
p= (int*)malloc(sizeof(int)*10);
```

Elemezzük a fenti kifejezést, bentről kifelé haladva: a `sizeof` operátor azt adja vissza, hogy az operandusaként kapott típus hány bájton ábrázolható. Ezt megszorozva 10-zel éppen 10 darab egész szám ábrázolásához szükséges bájtok számát kapjuk. A `malloc` tehát lefoglalja ezt követően a 10 darab egész ábrázolásához szükséges memóriaterületet, majd visszaadja annak címét `void*` típusként. Ez teljes értékű cím, de nincs hozzá típus rendelve, azaz a speciális üres (`void`) típus van hozzárendelve. Mielőtt ezt értékül adnánk a `p` mutatónak, explicit konverzióval `int*` típusúvá konvertáljuk. A `p` mutató ezt követően egy olyan memóriaterületre mutat, amely 10 darab egész szám tárolására alkalmas. Jegyezzük meg a fenti konstrukciót, amely a `malloc` függvény használatának általános módját demonstrálja. Megjegyezzük, hogy a 10 szám helyett tetszőleges változó is használható. Egy kérdés maradt már csak: ha lefoglaltunk 10 egész tárolására alkalmas memóriaterületet, és tudjuk annak címét, hogyan érhetjük el az `i`-edik ($0 \leq i \leq 9$) egész ábrázolására alkalmas memóriaterületet. A válasz a mutatóaritmetika. Mutatókhoz hozzáadhatunk konstans értékeket, ha a mutató típusa $tipus$, akkor a konstans i értéket hozzáadva a p cím nem i-vel fog nőni, tehát nem i bájttal fog jobbra mutatni, hanem $i * sizeof(tipus)$ bájttal, ami azt jelenti, hogy ha a mutatóhoz hozzáadunk i-t, akkor éppen az i-edik adat tárolására alkalmas terület címét kaphatjuk meg a lefoglalt memóriaterületen belül.

```
int *p;
p= (int*)malloc(sizeof(int)*10);
*(p+1)= 2;
printf("%d", *(p+1));
```

A fenti kód tehát lefoglal egy 10 egész tárolására alkalmas memóriaterületet, majd a második helyre berakja a 2 értéket, s ezt követően ki is írja azt. Figyeljük meg, hogy a mutatók konstanssal történő

növelése után ismét az indirekciós `*` operátort használtuk, hiszen azt tetszőleges cím előtt használhatjuk, hogy az általa címzett területet elérjük.

Ha most összehasonlítjuk a p mutatónak történő értékadás után létrejött szerkezetet a memóriában, azt vehetjük észre, hogy az éppen megfelel az egy dimenziós tömbök szerkezetének (lásd a tömbökről szóló fejezetet). Ezek alapján aztán könnyű látni, hogy a mutatók és a tömbök lényegében azonos szerkezetek a memóriában. Ennek megfelelően a C nyelv lehetőséget is biztosít arra, hogy a mutatókat tömbként kezeljük, azaz a fenti `*(p+1)` kifejezés mindkét helyen ekvivalens módon használható a `p[1]` kifejezéssel. Összegezve mindezt tehát, egy mutató amely egy n adat tárolására alkalmas lefoglalt memóriaterületre mutat, tekinthető egy n méretű tömbnek, és az `*(p+i)` kifejezés ekvivalens a `p[i]` kifejezéssel.

A dinamikusan (`malloc` függvény használatával lefoglalat) memóriaterületeket azonban miután nincs rá szükségünk, minden esetben fel kell szabadítanunk. A felszabadításra a `free` függvény használható. A `free` függvény egyetlen paramétere az a mutató, mely által címzett memóriaterületet fel szeretnénk szabadítani, azaz visszaadni az operációs rendszernek tetszőleges használatra. Az előző kódrészletben lefoglalt memóriaterületet tehát az alábbi függvényhívással szabadíthatjuk fel:

```
free(p);
```

Lássuk, hogyan használhatjuk a mutatókat, akkor, ha a mutatók struktúrákat címeznek. Tekintsük az alábbi kódrészletet:

```
struct hallgato
{
  char neptun_kod[7];
  int szul_ev;
};

struct hallgato h;
struct hallgato* hp;
hp= &h;
```

A kérdés, hogy a `hp` mutatóból hogyan érhetjük el az általa címzett struktúra mezőit. A legkézenfekvőbb megoldás természetesen a `*` operátor használata. Mivel a `*hp` kifejezés minden esetben a `h` objektumot jelenti, használhatjuk a `.` minősítő operátort, azaz a `h` változó mezőinek értéket adhatunk az alábbi utasításokkal:

```
(*hp).szul_ev= 1990;
strcpy((*hp).neptun_kod, "AAABBB");
printf("neptun kód: %s születési év: %d\n", (*hp).neptun_kod, (*hp).szul_ev);
```

A fenti kód jól szemlélteti, hogy `*hp` minden esetben a `hp` mutató által címzett objektumot jelenti, így ha az struktúra, használahatjuk akár a `.` minősítő operátort is mezőinek elérésére. A C nyelv lehetőséget biztosít a `(*hp).szul_ev` jellegű szerkezetek egyszerűsítésére a `->` operátor használatával. A `(*hp).szul_ev` kifejezés teljesen ekvivalens a `hp->szul_ev` kifejezéssel. Fontos azonban megjegyeznünk, hogy a `->` operátor csak és kizárólag akkor használható, ha a bal oldalán egy olyan mutató áll, amely valamilyen struktúrát címez. A fenti kódrészlet tehát ekvivalens az alábbival:

```
hp->szul_ev= 1990;
strcpy(hp->neptun_kod, "AAABBB");
```

A továbbiakban a `->` operátort használjuk ahol lehetséges.

Lássuk hogyan oldhatjuk meg most azt a feladatot, amelyet a szakasz elején még nem tudtunk megoldani!

13.1. Feladat: Írjon programot, amely a bemenetről beolvas egy egész számot (n), majd n darab egész számot és fordított sorrendben kiírja azokat.

13.1. Megoldás:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int n;
  int *p;
  int i;

  scanf("%d", &n);
  p= (int*)malloc(sizeof(int)*n);
  for ( i= 0; i < n; ++i )
    scanf("%d", &(p[i]));
  for ( i= n-1; i >= 0; --i )
    printf("%d", p[i]);

  free(p);
  return 0;
}
```

Mivel a tömbök tekinthetők változók sorozatának, a `scanf` függvény paraméterlistáján bátran alkalmazhatjuk az `&` operátort a `p[i]`, mint változó előtt. Ekvivalens módon, a `scanf` paraméterlistáján szerepelhetne a `&(*(p+i))` kifejezés is. Vegyük észre, hogy az `&` és a `*` operátor egymás ellentétei, tehát közvetlenül egymás után alkalmazva őket kioltják egymás hatását, így a `scanf` paraméterlistáján a korábbiakkal szintén ekvivalens módon használhatjuk a `p+i` kifejezést is, hiszen az szintén annak a memóriaterületnek a címét adja, amelyre az `i`-edik egész értéket be kell olvasni.

13.2. Feladat: Írjon programot, amely beolvas egy n számot, majd beolvas n darab lebegőpontos számot, és ezt követően kiírja a számok mediánját!

14. Egy irányba láncolt lista

Az egy irányba láncolt lista a legegyszerűbb dinamikus adatszerkezet, amellyel foglalkozunk. Visszatérve az előző szakasz végén lévő feladat megoldására, vegyük észre, hogy bár a tömböt dinamikusan hoztuk létre, miután az létrejött, annak mérete rögzítetté vált, hiszen a tömb statikus adatszerkezet, így legfeljebb annyi elemet tudunk tárolni benne, amekkora területet lefoglaltunk. A tömb tehát bár dinamikusan hoztuk létre, továbbra is statikus adatszerkezet. Célunk azonban olyan adatszerkezet létrehozása, amelyben tetszőleges számú elemet tárolhatunk, dinamikusan hozzáadhatunk elemeket növelve az adatszerkezet méretét, és dinamikusan eltávolítva elemeket, csökkentve ezzel az adatszerkezet méretét. Vegyük észre azonban, hogy dinamikus adatszerkezeteket nem tárolhatunk egy folytonos memóriaterületen, hiszen a lefoglalt folytonos memóriaterület méretét nem tudjuk hatékonyan növelni vagy csökkenteni. Mindez azt jelenti, hogy a dinamikus adatszerkezet elemei külön-külön, önállóan helyezkednek el a memóriában. Szükség van azonban arra, hogy egy elemből kiindulva be tudjuk járni az adatszerkezetet, végig tudjunk menni rajta, azaz az elemek között legyen valamilyen kapcsolat. Erre kiváló eszközt nyújt a mutatók használata. A legegyszerűbb esetben, azaz egyirányba láncolt listánál minden egyes adatelemhez, amelyet a listában tárolunk, hozzárendelünk egy mutatót, amely a valamilyen értelemben vett "következő" elemre mutat. Mivel egy mutatóból az előző szakaszban tárgyaltak alapján el tudjuk érni a mutató által címzett adatot, a mutatók használatával az adatszerkezetetben tárolandó elemeket lényegében összefűzzük, és az első, kitüntetett, általában *fej*nek nevezett elemből kiindulva az adatszerkezet bejárható, azaz minden eleme elérhető. A lista végét az utolsó elem `NULL` mutató értéke jelzi, hiszen az nem mutat további elemre.

Mivel minden elemhez hozzárendelünk egy mutatót, összetett, saját típust kell használnunk. Az alábbiakban olyan listát hozunk létre, amely egész számok tárolására lesz alkalmas. A saját típus,

amely egy-egy önálló szerkezetet reprezentál a memóriában, az alábbi lesz:

```
struct lista
{
  int adat;
  struct lista* kov;
};
```

Értelemszerűen a `struct lista` típus `adat` mezőjében tároljuk azt az adatrészt, amelyet dinamikus adatszerkezetben szeretnénk kezelni, míg a `kov` mező a következő ilyen típusú objektumra mutathat.

Üres lista létrehozása A további kódjainkban egy láncolt listát egyetlen változó reprezentál, mégpedig az első elem mutatója, hiszen ezen mutatóból kiindulva a lista minden elemét elérhetjük. Az alábbi példában tehát két különböző láncolt listát hozunk létre, `a` és `b` néven:

```
struct lista* a= NULL;
struct lista* b= NULL;
```

A változó `NULL` értékkel történő inicializálása lényegében üres lista létrehozását jelenti. A továbbiakban az egyirányba láncolt listákat kezelő leggyakoribb függvényeket vesszük végig.

14.1. Függvényekkel

Beszúrás Adatszerkezetek esetén a legalapvetőbb művelet a beszúrás. Mivel a láncolt lista egy szekvenciális adatszerkezet, azaz van első és utolsó elem, valamint minden elemnek van rákövetkezője. Ezek alapján több különböző helyre is elvégezhetjük a beszúrást. Az eljárásorientált programozás jegyében azonban a beszúrást végző függvények közös részét, azaz egy-egy új terület lefoglalását és inicializálását végző műveleteket kiemeljük az `ujelem` függvénybe, amely visszatérési értéke egy újonnan lefoglalt, a listába történő befűzésre készen álló objektum címe.

```
struct lista* ujelem(int x)
{
  struct lista* tmp;

  tmp= (struct lista*)malloc(sizeof(struct lista));
  tmp->adat= x;
  tmp->kov= NULL;

  return tmp;
}
```

Az `ujelem` függvény felhasználásával a láncolt lista elejére történő beszúrást megvalósító függvény az alábbi:

```
struct lista* elejere_beszur(int x, struct lista* fej)
{
  struct lista* tmp;

  tmp= ujelem(x);
  tmp->kov= fej;

  return tmp;
}
```

A lista végére történő beszúrás során két esetet különböztethetünk meg. Az első esetben, ha a lista üres lista, megváltozik a `fej` értéke, a második esetben nem. Ezen két esetet meg kell különböztetnünk:

```
struct lista* vegere_beszur(int x, struct lista* fej)
{
  struct lista* tmp;

  if ( fej == NULL )
    return ujelem(x);

  tmp= fej;
  while ( tmp->kov != NULL )
    tmp= tmp->kov;
  tmp->kov= ujelem(x);

  return fej;
}
```

Ügyeljünk arra, hogy a `fej` változót vissza kell adnunk, ezért a lista végére történő pozícionálást egy `tmp` segédváltozó segítségével oldjuk meg. A lista vége felé a `tmp= tmp->kov` iterált alkalmazásával pozícionálhatunk, a lista utolsó elemét akkor érjük el, amikor a `while` ciklus feltétele igazzá válik, hiszen az utolsó elemre igaz csak, hogy a rákövetkezője `NULL`.

A fenti kódot természetesen `while` ciklus helyett `for` ciklussal is megvalósíthatjuk:

```
struct lista* vegere_beszur(int x, struct lista* fej)
{
  struct lista* tmp;

  if ( fej == NULL )
    return ujelem(x);

  for ( tmp= fej; tmp->kov != NULL; tmp= tmp->kov );

  tmp->kov= ujelem(x);

  return fej;
}
```

Vegyük észre, hogy a fenti megoldásban a `for` ciklus magja üres.

Törlés A listából történő törlést megvalósító függvényekből is kettőt valósítunk most meg, a lista elejéről és végéről történő törlést. Amikor a lista elejéről törlünk, két esetet kell megkülönböztetnünk, azt, amikor üres listából szeretnénk törölni, és azt, amikor legalább egy elemet tartalmazó listából törlünk. Ne felejtsük el, hogy nem elegendő egy elemet kifűzni a listából azt fel is kell szabadítani a `free` függvénnyel:

```
struct lista* elejerol_torol(struct lista* fej)
{
  struct lista* tmp;

  if ( fej == NULL )
    return NULL;

  tmp= fej->kov;
  free(fej);

  return tmp;
}
```

A lista végéről történő törlés során három esetet kell megkülönböztetnünk: első eset, amikor üres listából törlünk, második eset, amikor egy elemet tartalmazó listából és harmadik eset, amikor több, mint egy elemet tartalmazó listából törlünk.

```
struct lista* vegerol_torol(struct lisat* fej)
{
  struct lista* tmp;

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

  tmp= fej;
  while ( tmp->kov->kov != NULL )
    tmp= tmp->kov;
  free(tmp->kov);
  tmp->kov= NULL;

  return fej;
}
```

Bejárás A bejárás művelet azt jelenti, hogy valamilyen módon végig szeretnénk menni az adatszerkezeten, és minden elemet elérni, feldolgozni. A feldolgozás jelentheti a legegyszerűbb esetben az elem kiírását a konzolra. Ezt egy nagyon egyszerű struktúrájú függvénnyel valósíthatjuk meg:

```
void bejar(struct lista* fej)
{
  while ( fej )
  {
    printf("%d ", fej->adat);
    fej= fej->kov;
  }
}
```

A bejárás során a kiírás, tehát a `printf` függvény helyett tetszőleges feldolgozási lépés szerepelhet.

Keresés A keresés során, hasonlóan a tömbökben történő kereséshez, a keresett értéket (adatot) tartalmazó listaelem címét adjuk vissza, ha megtaláljuk (hasonlóan a tömbökben történő kereséshez, ahol az indexet adtuk vissza), és `NULL` mutatót adunk vissza, ha nincs benne a keresett elem a listában. A függvény nagyon hasonló a bejáráshoz, a feldolgozás itt egy egyszerű feltétel ellenőrzése:

```
struct lista* keres(int x, struct lista* fej)
{
  while ( fej )
  {
    if ( fej->adat == x )
      return fej;
  }
  return NULL;
}
```

Csere A csere függvényhez felhasználjuk a `keres` függvényt:

```
struct lista* csere(int mit, int mire, struct lista* fej)
{
  struct lista* tmp= keres(mit, fej);

  if ( tmp == NULL )
    return fej;
```

```
  tmp->adat= mire;
  return fej;
}
```

Adott elem után történő beszúrás A függvényben felhasználjuk a már létező `keres` függvényünket:

```
struct lista* elem_utan_beszur(int x, int ez_utan, struct lista* fej)
{
  struct lista* tmp= keres(ez_utan, fej);
  struct lista* tmp2;

  if ( tmp == NULL )
    return vegere_beszur(x, fej);

  tmp2= ujelem(x);
  tmp2->kov= tmp->kov;
  tmp->kov= tmp2->kov;

  return fej;
}
```

Adott elem törlése Mivel a lista közepén található elem törlése során az elemet megelőző elemnek a `kov` mezőjét kell módosítani, meg kell találnunk az adott adat értéket megelőző listaelemet.

```
struct lista* adott_elem_torlese(int mit, struct lista* fej)
{
  struct lista* tmp= fej;
  struct lista* elozo= NULL;

  while ( tmp && tmp->adat != mit )
  {
    elozo= tmp;
    tmp= tmp->kov;
  }

  if ( tmp == NULL )
    return fej;

  if ( elozo == NULL )
    fej= fej->kov;
  else
    elozo->kov= tmp->kov;
  free(tmp);

  return fej;
}
```

Lista törlése A lista törlése művelet a lista minden elemének felszabadítását jelenti. Ügyeljünk kell azonban arra, hogy miután egy elemet felszabadítottunk, a mezőit már nem érhetjük el, ezért egy elem felszabadítása előtt el kell tárolnunk egy segédváltozóban az őt követő elem mutatóját, hogy továbbléphessünk majd és azt is felszabadíthassuk:

```
struct lista* felszabadit(struct lista* fej)
{
  struct lista* tmp;
```

```
  while ( fej )
  {
    tmp= fej->kov;
    free(fej);
    fej= tmp;
  }

  return NULL;
}
```

Példa Az eddig megírt függvények mind arra építenek, hogy a hívásuk után a nekik átadott paraméter értékül kapja a visszatérési értéküket, azaz a függvények hívása az alábbi sablon alapján történik:

```
fej= fuggveny(fej);
```

Ezek alapján ha a függvényeket egy forráskód elejére írjuk, az alábbi főprogram lefordítható és működik:

```
int main(int argc, char** argv)
{
  struct lista* a= NULL;
  struct lista* b= NULL;

  a= elejere_beszur(2, a);
  a= vegere_beszur(3, a);
  b= vegere_beszur(3, b);
  b= vegere_beszur(4, b);
  a= elejerol_torol(a);
  b= elem_utan_beszur(5, 4, b);
  a= csere(3, 6, a);

  bejar(a);
  printf("\n");
  bejar(b);

  felszabadit(a);
  felszabadit(b);
}
```

A fenti kód futása után az alábbi kimenetet láthatjuk:

```
6
3 4 5
```

14.2. Eljárásokkal

A függvények helyett eljárások segítségével is megírhatjuk az egyes listakezelő műveleteket, azonban ekkor, mivel a hívási környezetben lévő `fej` változó megváltozik, nem elegendő a `fej` értékét átadni paraméterként, hiszen ekkor nem tudjuk megváltoztatni a hívási környezetben a változót. Megolásként annak címét kell átadnunk, a cím átadása azonban kétszeres indirekcióval történik, azaz mutatóra mutató mutatót kell átadnunk paraméterként, aminek kezelése kicsit körülményesebb, azonban az eredmény mindenképpen elegánsabb és hatékonyabb is. Az alábbiakban a lista elejére történő beszúrást és a lista elejéről történő törlést valósítjuk meg eljárások segítségével. A megoldások során az algoritmus nem változik, csak a `fej` változó kezelése. A további algoritmusok átírása analóg módon történhet.

Lista elejére beszúrás

```
void elejere_beszur(int x, struct lista** fejp)
{
  struct lista* tmp;

  tmp= ujelem(x);
  tmp->kov= *fejp;

  *fejp= tmp;
}
```

Lista elejéről törlés

```
void elejerol_torol(struct lista** fejp)
{
  struct lista* tmp;

  if ( *fejp == NULL )
    return;

  tmp= (*fejp)->kov;
  free(*fejp);

  *fejp= tmp;
}
```

Könnyű látni, hogy a fenti kódokban mindkét esetben csak a `fej` változót cseréltük `*fejp` változóra, hogy ugyanaz az információ jelenjen meg az adott helyen, ami a függvényekkel történő megvalósításban, valamint a `return` utasításokat kell értelemszerűen módosítani, ugyanis a függvények használatakor a `return` utasítás utáni visszatérési érték bekerül a hívási környezet `fej` változójába a `fej= fuggveny(fej)` jellegű használat miatt. Namost ha nincs visszatérési értékünk, akkor a függvényeknél megjelenő visszatérési értéket egy egyszerű `*fejp= visszateresi_ertek` utasítással is elhelyezhetjük a hívási környezet `fej` változójában, hiszen a `fejp` változó éppen annak címét tartalmazza. Üres lista létrehozása után az eljárásokat az alábbi módon használhatjuk:

```
struct lista* a= NULL;

elejere_beszur(3, &a);
elejere_beszur(4, &a);
elejerol_torol(&a);
```

14.1. Feladat: Írjuk át a többi lista kezelő függvényt is eljárásra!

14.3. Rekurzió

Rekurziónak azt nevezzük, amikor egy függvény vagy eljárás önmagát hívja meg. Rekurzió használatával nagyon sok problémát elegánsan és röviden oldhatunk meg, azonban nem mindig a leghatékonyabb megközelítés. Leggyakrabban bejárásoknál és adatszerkezetekben történő pozícionálásnál használják a rekurziót. Lássuk, néhány függvény rekurzív megvalósítását.

Bejárás Elsőként nézzük, hogyan írhatjuk ki a lista elemeit a tárolás sorrendjében, rekurzívan.

```
void bejar_elejerol(struct lista* fej)
{
  if ( fej == NULL )
```

```
    return;
  printf("%d ", fej->adat);
  bejar_elejerol(fej->kov);
}
```

A lista elemeinek kiírása fordított sorrendben:

```
void bejar_vegerol(struct lista* fej)
{
  if ( fej == NULL )
    return;
  bejar_vegerol(fej->kov);
  printf("%d ", fej->adat);
}
```

Lista végére történő beszúrás

```
struct lista* vegere_beszur(int x, struct lista* fej)
{
  if ( fej == NULL )
    return ujelem(x);
  fej->kov= vegere_beszur(x, fej->kov);
  return fej;
}
```

Lista végéről történő törlés

```
struct lista* vegerol_torol(struct lista* fej)
{
  if ( fej == NULL )
    return NULL
  if ( fej->kov == NULL )
  {
    free(fej);
    return NULL;
  }
  fej->kov= vegerol_torol(fej->kov);
  return fej;
}
```

14.4. typedef

Egyes megvalósításokban kihasználják a C nyelv azon lehetőségét, mellyel elnevezhetjük a `struct lista` típust egy rövidebb névvel, a következő módon, a `typedef` utasítás felhasználásával:

```
typedef struct lista
{
  int adat;
  struct lista* kov;
} LISTA;
```

Ezt követően a `LISTA` típust bárhol használhatjuk, ahol a `struct lista` típust a korábbiakban, tehát ez csak egy rövidítés, semmit sem változtat az algoritmusok működésén. A lista elejére történő beszúrást az alábbi módon valósíthatjuk meg a fenti saját típus definícióval:

```
LISTA* elejere_beszur(int x, LISTA* fej)
{
  LISTA* tmp;

  tmp= ujelem(x);
  tmp->kov= fej;

  return tmp;
}
```

15. Kétirányba láncolt lista

A kétirányba láncolt listát úgy lehet elképzelni, mint egy egyirányba láncolt listát, amit "hátulról visszafelé IS felfűzünk", azaz minden elem tartalmaz egy mutatót az őt megelőzőre is. A listát az első és utolsó elemének mutatójával reprezentáljuk, s ennek az az előnye, hogy a lista végére történő beszúrásnál nincs szükség arra, hogy újra és újra a lista végére kelljen pozicionálni. Ebben az esetben a listát tehát az első és utolsó elemének mutatója reprezentálja, ami azt jelenti, hogy az elegancia és hatékonyság érdekében össze kell fognunk ezen két változót egy újabb struktúrába. Tehát szemben az egyirányba láncolt listáknál, ahol egy listaelem típusú mutató reprezentálta a listát, itt most egy olyan struktúra reprezentálja azt, amely két mezőt tartalmaz, az egyik mező a lista első elemét, a másik mező a lista utolsó elemét címzi. Emellett persze szükség van a szokásos, listaelemeket reprezentáló struktúrára is, hiszen ebben tároljuk az adatokat. A kétirányba láncolt lista kezeléséhez tehát két struktúrát hozunk létre:

```
struct klistaelem
{
  int adat;
  struct klistaelem* elozo;
  struct klistaelem* kovetkezo;
};

struct klista
{
  struct klistaelem* fej;
  struct klistaelem* veg;
}
```

Léterhozás Ezek alapján üres listát úgy hozhatunk létre, hogy egyszerűen deklarálunk egy `struct klista` típusú változót, és beállítjuk annak mezőit `NULL` értékekre, hiszen nincs sem `fej`, sem `veg` eleme egy üres listának:

```
struct klista a= {NULL, NULL};
```

A fenti `a` változó reprezentál egy kétirányba láncolt listát.

15.1. Függvények

Beszúrás A kétirányba láncolt listába történő beszúrást is több módon valósíthatjuk meg, az alábbiakban a lista elejére és végére történő beszúrást végző függvényeket írjuk meg. Természetesen az `ujelem` függvény megírásával kezdünk.

```
struct klistaelem* ujelem(int x)
{
  struct klistaelem* tmp= (struct klistaelem*)malloc(sizeof(struct klistaelem));
```

```
  tmp->adat= x;
  tmp->elozo= tmp->kovetkezo= NULL;

  return tmp;
}


struct klista elejere_beszur(int x, struct klista a)
{
  if ( a.fej == NULL )
    a.fej= a.veg= ujelem(x);
  else
  {
    a.fej->elozo= ujelem(x);
    a.fej= a.fej->elozo;
  }

  return a;
}


struct klista vegere_beszur(int x, struct klista a)
{
  if ( a.veg == NULL )
    a.veg= a.fej= ujelem(x);
  else
  {
    a.veg->kovetkezo= ujelem(x);
    a.veg= a.veg->kovetkezo;
  }

  return a;
}
```

Nagyban megkönnyíti a lista végein operáló függvények működését, hogy a `veg` és `fej` mezőknek köszönhetően a kétirányba láncolt lista szimmetrikus, így szimmetriai okokból ha megírtunk egy függvényt a lista egyik végére, azt nagyon könnyen átírhatjuk egyszerű szimmetrikus mezőnév helyettesítéssel úgy, hogy a lista másik végén operáljon.

Törlés

```
struct klista elejerol_torol(struct klista a)
{
  if ( a.fej == NULL )
    return a;
  if ( a.fej == a.veg )
  {
    free(a.fej);
    a.fej= a.veg= NULL;
    return a;
  }
  a.fej= a.fej->kovetkezo;
  free(a.fej->elozo);
  a.fej->elozo= NULL;

  return a;
}


struct klista vegerol_torol(struct klista a)
{
  if ( a.veg == NULL )
```

```
    return a;
  if ( a.veg == a.fej )
  {
    free(a.veg);
    a.veg= a.fej= NULL;
    return a;
  }
  a.veg= a.veg->elozo;
  free(a.veg->kovetkezo;
  a.veg->kovetkezo= NULL;

  return a;
}
```

Bejárás A bejárás esetünkben történhet két irányból, az lista fejétől a vége felé és a vége felől a fej felé haladva is.

```
void bejar_elejerol(struct klista a)
{
  while ( a.fej )
  {
    printf("%d ", a.fej->adat);
    a.fej= a.fej->kovetkezo;
  }
}


void bejar_vegerol(struct klista a)
{
  while ( a.veg )
  {
    printf("%d ", a.veg->adat);
    a.veg= a.veg->elozo;
  }
}
```

Így, hogy az `a` formális paramétert érték szerinti paraméterátadással vettük át, felhasználhatjuk annak mezőit hiszen az értékük megváltozása nincs hatással a hívási környezetben lévő, a kétirányú listát reprezentáló `struct klista` típusú változóra. Természetesen segédváltozó segítségével is megvalósíthatjuk a bejárásokat.

```
void bejar_elejerol(struct klista a)
{
  struct klistaelem* tmp= a.fej;
  while ( tmp )
  {
    printf("%d ", tmp->adat);
    tmp= tmp->kovetkezo;
  }
}
```

Keresés A keresés művelet az egyirányba láncolt listához hasonlóan a bejárás vázán épülhet fel:

```
struct klistaelem* keres(int x, struct klista a)
{
  while ( a.fej )
  {
    if ( a.fej->adat == x )
      return a.fej;
```

```
    a.fej= a.fej->kovetkezo;
  }
  return NULL;
}
```

Csere A keresés függvény segítségével aztán nagyon könnyen megírhatjuk a csere műveletet:

```
void csere(int mit, int mire, struct klista a)
{
  struct klistaelem* tmp= keres(mit, a);
  if ( tmp != NULL )
    tmp->adat= mire;
}
```

Beszúrás adott elem után Adott elem után történő beszúráshoz szintén felhasználjuk az eddigi függvényeinket:

```
struct klista beszur_elem_utan(int x, int ez_utan, struct klista a)
{
  struct klistaelem* tmp= keres(ez_utan, a);
  struct klistaelem* uj;

  if ( tmp == NULL )
    return a;
  if ( tmp == a.veg )
    return vegere_beszur(x, a);
  uj= ujelem(x);
  uj->elozo= tmp;
  uj->kovetkezo= tmp->kovetkezo;
  uj->kovetkezo->elozo= uj;
  tmp->kovetkezo= uj;

  return a;
}
```

Adott elem törlése

```
struct klista elem_torles(int mit, struct klista a)
{
  struct klistelem* tmp= keres(mit, a);

  if ( tmp == NULL )
    return a;
  if ( tmp == a.veg )
    return elejerol_torol(a);
  if ( tmp == a.fej )
    return vegerol_torol(a);
  tmp->elozo->kovetkezo= tmp->kovetkezo;
  tmp->kovetkezo->elozo= tmp->elozo;
  free(tmp);
  return a;
}
```

Felszabadítás A listát törölni megintcsak az egyirányba láncolt listánál is alkalmazott stratégia szerint tudunk:

```
struct klista felszabadit(struct klista a)
{
  struct klistaelem* tmp;
  while ( a.fej )
  {
    tmp= a.fej->kovetkezo;
    free(a.fej);
    a.fej= a.fej->kovetkezo;
  }
  a.fej= a.veg= NULL;
  return a;
}
```

Példa Az eddig megírt függvényeket tehát az alábbi módon használhatjuk:

```
struct klista a= {NULL, NULL};
a= elejere_beszur(4, a);
a= vegerol_torol(a);
bejar_elejerol(a);
```

15.2. Eljárások

Hasonlóan az egyirányba láncolt listák esetéhez, a kétirányba láncolt listákat kezelő függvények helyett is írhatunk eljárásokat, ekkor azonban a `struct klista` változó címét kell átvennünk paraméterként. A lista elejére történő beszúrás esetén ez az alábbi módon alakul:

```
void elejere_beszur(int x, struct klista* a)
{
  if ( a->fej == NULL )
    a->fej= a->veg= ujelem(x);
  else
  {
    a->fej->elozo= ujelem(x);
    a->fej= a->fej->elozo;
  }
}
```

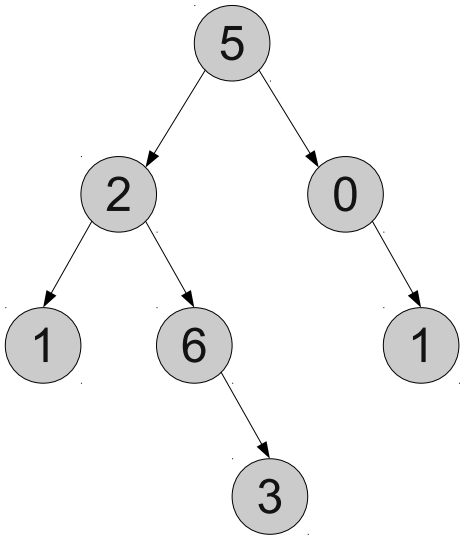
Maga az algoritmus, tehát az egymást követő lépések ebben az esetben sem változtak, arra azonban ügyelnünk kell, hogy egyrészt most mutató paramétert kaptunk, így értelemszerűen a mezői eléréséhez a `->` operátort kell használnunk, másrészt most a hívási környezetben lévő változót kezeljük közvetlenül, így nincs szükség visszatérési értékre, és ha szükséges, segédváltozókat kell bevezetnünk.

16. Általános bináris fa

A bináris fák az egyirányba láncolt lista általánosításai abban az értelemben, hogy egy helyett kettő rákövetkezője van minden faelemnek. Ezeket a rákövetkezőket *bal-* és *jobb*oldali rákövetkezőknek hívjuk. Egy egészeket tartalmazó bináris fát megvalósító struktúra az alábbi:

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

A fát egyetlen mutató reprezentálja, amely a fa gyökér elemére mutat:



Ábra 1: Általános bináris fa

```
struct faelem* gyoker= NULL;
```

A `NULL` mutató specifikálásával egy üres fát hoztunk létre. Ahhoz, hogy a fába elemeket tudjunk beszúrni, elsőként megírjuk a szokásos `ujelem` függvényt, amely létrehoz egy megfelelő struktúrát és inicializálja annak mezőit:

```
struct faelem* ujelem(int x)
{
  struct faelem* tmp;
  tmp= (struct faelem*)malloc(sizeof(struct faelem));
  tmp->adat= x;
  tmp->bal= tmp->jobb= NULL;

  return tmp;
}
```

Az általános bináris fa felépítését azonban nem végezhetjük olyan általánosan, mint tettük azt a láncolt listák esetén, azaz nem tudunk olyan függvényt/eljárást írni, amely egy adott helyen bővíti a fát, a fának ugyanis nem egy, vagy két vége van, hanem ahogy nő a fa, úgy nő azoknak a helyeknek a száma, amelyet a fa "végének" nevezhetnénk. A fa bővítése ezért csak akkor oldható meg automatikusan, ha a bővítés során bevezetünk valamilyen szabályt, amely szabály aztán pontosan meghatározza, hogy egy adott elemet hová kell beszúrni a fában. A következő szakaszban, a bináris keresőfák esetén láthatunk példát egy ilyen szabályra. Az alábbiakban egy néhány elemből álló fát explicit módon, az `ujelem` függvény segítségével építünk fel. Az alábbi kódrészletben az 1. ábrán látható bináris fát építjük fel.

```
struct faelem* gyoker= NULL;
gyoker= ujelem(5);
gyoker->bal= ujelem(2);
gyoker->bal->bal= ujelem(1);
gyoker->bal->jobb= ujelem(6);
gyoker->bal->jobb->jobb= ujelem(3);
gyoker->jobb= ujelem(0);
gyoker->jobb->jobb= ujelem(1);
```

Ami közös a legkülönbözőbb bináris fákban, az a lehetséges rekurzív bejárások módja. Rekurzív bejárásokkal már találkoztunk egyirányba láncolt listáknál, ahol megoldást találtunk arra, hogy megfelelő rekurzióval a lista elejéről a vége felé, és a lista végéről az eleje felé haladva is ki tudjuk írni a lista elemeit. Ami különbség volt a két módszer között, az az, hogy hová helyeztük el a feldolgozást és a rekurzív hívást a kódban, egymáshoz viszonyítva. Mivel a bináris fák esetén minden faelemnek egy bal és egy jobb oldali leszármazottja van, hasonló meggondolásból három különböző módon tudjuk bejárni a bináris fát rekurzívan:

- preorder módon: egy elem feldolgozását a leszármazottai feldolgozása előtt végezzük;
- inorder módon: egy elem feldolgozását a bal oldali leszármazottjának feldolgozása után, de a jobb oldali leszármazottjának feldolgozása előtt végezzük;
- postorder módon: egy elem feldolgozását a leszármazottai feldolgozása után végezzük.

Ezek alapján az alábbi eljárásokat írhatjuk meg arra az esetre, ha a bináris fa feldolgozása az elemeinek a kiírását jelenti:

```
void preorder(struct faelem* gyoker)
{
  if ( gyoker == NULL )
    return;
  printf("%d ", gyoker->adat);
  preorder(gyoker->bal);
  preorder(gyoker->jobb);
}

void inorder(struct faelem* gyoker)
{
  if ( gyoker == NULL )
    return;
  inorder(gyoker->bal);
  printf("%d ", gyoker->adat);
  inorder(gyoker->jobb);
}

void postorder(struct faelem* gyoker)
{
  if ( gyoker == NULL )
    return;
  postorder(gyoker->bal);
  postorder(gyoker->jobb);
  printf("%d ", gyoker->adat);
}
```

A három eljárás sémája hasonló, rekurzívan hívja mindhárom eljárás saját magát, a rekurzív hívást az szakítja meg, ha a paraméterként kapott faelem, azaz a feldolgozandó részfa gyökéreleme `NULL`.

A példában megadott bináris fa bejárásakor a három különböző bejárással három különböző sorrendet kapunk:

- preorder: `5 2 1 6 3 0 1`;
- inorder: `1 2 6 3 5 0 1`;
- postorder: `1 3 6 2 1 0 5`.

Egy elem törlésének menetét elég algoritmikusan ismerni: a legegyszerűbb a levélelemek törlése, mivel nincsenek leszármazottjaik, egyszerűen csak fel kell szabadítani őket és a szülő elemük megfelelő mutatóját `NULL` értékre állítani. Ha egy olyan elemet szeretnénk törölni, amely nem levélelem, akkor mivel a fában az elemek sorrendjét semmilyen szabály nem köti, a legegyszerűbb, ha a törlendő elem `adat` mezőjét felülírjuk valamelyik levélelem `adat` mezőjével és aztán a levélelemet töröljük az előző mondatban leírt módon.

Részfa (vagy akár az egész fa) törlését rekurzívan valósíthatjuk meg, azonban ügyelnünk kell arra, hogy előbb mindig a leszármazott elemeket kell törölnünk és utána törölhetjük csak az aktuális gyökér elemet. A három rekurzív bejárási mód közül ezt a stratégiát éppen a postorder bejárás valósítja meg, azaz

```
struct faelem* felszabadit(struct faelem* gyoker)
{
  if ( gyoker == NULL )
    return NULL;
  felszabadit(gyoker->bal);
  felszabadit(gyoker->jobb);
  free(gyoker);
  return NULL;
}
```

A korábban felépített fa esetén az utóbb megírt 4 függvényt az alábbi módon használhatjuk:

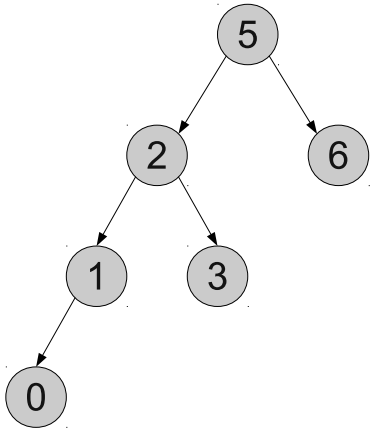
```
preorder(gyoker);
inorder(gyoker);
postorder(gyoker);
gyoker->bal= felszabadit(gyoker->bal);
gyoker= felszabadit(gyoker);
```

A `felszabadit` függvénye első hívása csak a baloldali részfát törli, a második hívás pedig az egész fát.

17. Bináris keresőfa

A bináris keresőfa tehát egy speciális bináris fa, speciális abban az értelemben, hogy egy szabályt rendelünk hozzá, s ez a szabály egyértelműen meghatározza minden elem helyét a fában. A szabály a következő: minden elemre teljesülnie kell, hogy az aktuális elemből induló baloldali részfában az aktuális elem `adat` mezőjétől csak kisebb `adat` mezőjű faelemek szerepelhetnek, a jobb oldali részfában pedig csak nagyobb `adat` mezőjű faelemek. Könnyű látni, hogy ez a szabály egy rendezettséget visz a fába és egy adott fa esetén egyértelműen meghatározza, hogy egy új elemet hová kell beszúrnunk a fába. Vegyük észre azt is, hogy egy bináris keresőfa minden elemből csak egyet tartalmazhat! Tekintsük az előző szakaszban vizsgált bináris fát, amelybe az elemek a `5 2 1 6 3 0 1` sorrendben kerültek be. Ezen érkezési sorrendből a 2. ábrán látható bináris keresőfát építhetjük fel a megadott szabály alapján. Jegyezzük meg tovább, hogy a fa függ a bérkező elemek sorrendjéből, ugyanazon elemekből több olyan bináris fát is fel lehet építeni, amelyre teljesül a bináris keresőfákat jellemző rendezettségi szabály. Egy bináris keresőfa megvalósítható és kezelhető az általános bináris fák esetén létrehozott eszközökkel (struktúrákkal, eljárásokkal), egyedül a fát módosító függvényeket kell úgy megírni, hogy a módosítás után is teljesüljön a bináris keresőfákat jellemző szabály a fára. A bináris keresőfába történő beszúrást megvalósító szabály tehát a következő:

- Ha a gyökérelem üres, akkor hozzunk létre egy új elemet és adjuk vissza ennek mutatóját.
- Ha a gyökérelemben tárolt adat kisebb, mint a beszúrandó adat, akkor a jobb oldali részfába kell beszúrnunk azt ugyanezen szabállyal.
- Ha a gyökérelemben tárolt adat nagyobb, mint a beszúrandó adat, akkor a bal oldali részfába kell beszúrnunk azt ugyanezen szabállyal.



Ábra 2: Bináris keresőfa

Az algoritmus megvalósítása rekurzióval:

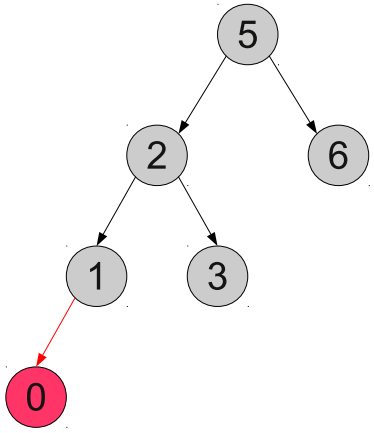
```
struct faelem* beszur(int x, struct faelem* gyoker)
{
  if ( gyoker == NULL )
    return ujelem(x);
  if ( gyoker->adat > x )
    gyoker->bal= beszur(x, gyoker->bal);
  if ( gyoker->adat < x )
    gyoker->jobb= beszur(x, gyoker->jobb);
  return gyoker;
}
```

A `beszur` függvény felhasználásával tehát a beszúrandó `5 2 1 6 3 0 1` számsorból éppen a 2. ábrán látható fát építhetjük fel:

```
struct faelem* gyoker;
gyoker= beszur(5, gyoker);
gyoker= beszur(2, gyoker);
gyoker= beszur(1, gyoker);
gyoker= beszur(6, gyoker);
gyoker= beszur(3, gyoker);
gyoker= beszur(0, gyoker);
gyoker= beszur(1, gyoker);
```

Könnyű látni, hogy ha olyan elemet szeretnék beszúrni, amely már szerepel a fába (a példában az 1-es érték), akkor az nem fog még egyszer bekerülni, azaz a példában felépített fa egy elemmel kevesebbet tartalmaz, mint ahány elemre meghívtuk a `beszur` függvényt.

Jegyezzük meg azt is, hogy a bináris keresőfa inorder bejárása éppen a benne tárolt elemek növekvő sorrendben történő kiírását eredményezi, hiszen minden elem előtt kiírjuk a bal oldali részfájának elemeit, azaz a tőle kisebb elemeket.



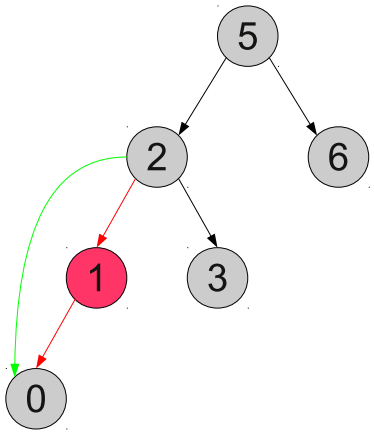
Ábra 3: Levélelem törlése (a pirossal jelölt értékeket töröltük)

A keresést szintén rekurzívan célszerű megvalósítani, hiszen ha keresünk egy elemet, potnosan tudjuk, hogy hol kell lennie. Ha nincs ott, azaz `NULL` értéket találunk a helyén, akkor `NULL` mutatóval térünk vissza, ellenekző esetben az őt tartalmazó faelem címével:

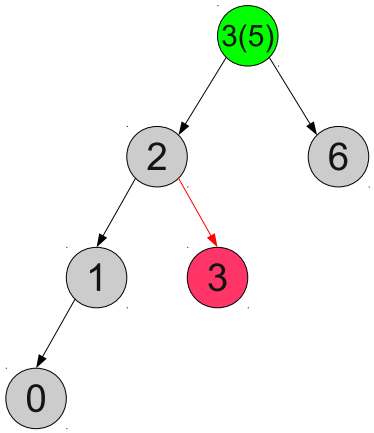
```
struct faelem* keres(int x, struct faelem* gyoker)
{
  if ( gyoker == NULL || gyoker->adat == x )
    return gyoker;
  if ( gyoker->adat < mit )
    return keres(x, gyoker->jobb);
  if ( gyoker->adat > mit )
    return keres(x, gyoker->bal);
}
```

A bináris fákhoz kapcsolódó utolsó algoritmusként nézzük meg a törlés menetét, csak algoritmikusan:

- Ha levélelemet akarunk törölni, azt egyszerűen megtehetjük, az általános bináris fáknál tárgyalt módon, azaz felszabadítjuk az elemet, majd a szülő elem rá mutató mutatóját `NULL`-ra állítjuk. (3. ábra)
- Ha olyan elemet szeretnénk törölni, amelynek csak egy rákövetkezője van, akkor a szülő elemnek a törlendő elemre mutató mutatóját felülírjuk a törlendő elem egyetlen rákövetkezőjének mutatójával, majd felszabadítjuk a törlendő elemet. (4. ábra)
- Ha olyan elemet szeretnénk törölni, amelynek két rákövetkezője van, akkor a törlendő elemet felülírjuk a baloldali részfájának legjobboldalibb elemével, vagy a jobboldali részfájának legbaloldalibb elemével (ugyanis ezekben az esetekben nem sérül a rendezettségi szabály), majd töröljük azt az elemet, amelyikkel felülírtuk a törlendő elemet, az első két szabály valamelyikével. (5. ábra)



Ábra 4: Egy leszármazottal rendelkező elem törlése (a pirossal jelölt értékeket töröltük, a zölddel jelölt kapcsolatot frissen hoztuk létre)



Ábra 5: Két leszármazottal rendelkező elem törlése (a törlendő elem a gyökerében lévő 5-ös érték, zöld jelölés azt jelenti, hogy csak módosítottuk a törlendő elem értékét, a beldolali részfa legjobboldalibb elemére (3), majd fizikailag a baloldali részfa legjobboldalibb elemét töröltük (a pirossal jelölt 3-as))

Irodalomjegyzék

[1] CORMEN, T. H. AND LEISERSON, C. E. AND RIVEST, L. R. (1996). *Algoritmusok* Műszaki Könyvkiadó, Budapest, HU, ISBN: 963 16 3029 3.