Magasszintű programozási nyelvek 1 gyakorlat

Kovács György

2007.05.04.

Tartalomjegyzék

EZ A DOKUMENTUM NEM C TUTORIAL, A GYAKORLATAIMON TÁBLÁRA FELKERÜLT, VAGY LEMARADT KÓDOKAT, ÉS AZOK MAGYARÁZATAIT TARTALMAZZA.

A PÉLDAPROGRAMOKAT A LEHETŐ LEGEGYSZERŰBBEN IGYEKSZEM BEMUTATNI, NEM A LEGHATÉKONYABBAN, VAGY LEGELEGÁNSABBAN. A PROBLÉMÁKRA LEHETNEK JOBB MEGOLDÁSOK.

A JEGYZET NEM LEKTORÁLT, TARTALMAZ(HAT) HIBÁKAT. HASZNÁLATÁT NEM JAVASLOM ÉS HASZNÁLATÁÉRT FELELŐSSÉGET NEM VÁLLALOK.

1. Függvények, eljárások

Függvényeket, és eljárásokat egy programban azért hozunk létre, hogy a kódot rövidebbé, és átláthatóbbá tegyük. Az alprogramok használatával a forráskód hasonló részleteit kiemelhetjük, oly módon, hogy a problémát általános, formális paraméterek használatával oldjuk meg. Nézzük a következő programot.

Írjon programot, amely beolvas egy számot (n), majd n darab egész számot egy tömbbe. A feladat: rendezzük a tömböt (növekvőleg), majd minden elemét szorozzuk meg a 10%index -el, aztán megint rendezzük a tömböt, majd minden eleméből vonjuk ki az indexét, majd megint rendezzük a tömböt, és írjuk ki az elemeit a kimenetre.

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

int main()
{
   int *t, i, n, tmp;

   scanf("%d",&n);

   t=(int *)malloc(sizeof(int)*n);

   /* Beolvasás: */
   for(i=0;i<n;++i)
      scanf("%d",t+i);

   /* Buborék rendezés: */
   for(i=0;i<n;++i)
      for(j=n;j>i;++j)
         if(t[j]<t[j-1]){
            tmp=t[j];
            t[j]=t[j-1];
            t[j-1]=tmp;
         }

   /*Elemek szorzása az 10%index-el: */
   for(i=0;i<n;++i)
```

```
      t[i]*=(10%i);

   /* Buborék rendezés: */
   for(i=0;i<n;++i)
      for(j=n;j>i;++j)
         if(t[j]<t[j-1]){
            tmp=t[j];
            t[j]=t[j-1];
            t[j-1]=tmp;
         }

   /*Indexek kivonása az elemből: */
   for(i=0;i<n;++i)
      t[i]-=i;

   /* Buborék rendezés: */
   for(i=0;i<n;++i)
      for(j=n;j>i;++j)
         if(t[j]<t[j-1]){
            tmp=t[j];
            t[j]=t[j-1];
            t[j-1]=tmp;
         }

    /*Elemek kiírása: */
    for(i=0;i<n;++i)
       printf("%d\t",t[i]);

    return 0;
}
```

Látható, hogy ez egy egész hosszú kód, annak ellenére, hogy szinte alig csinál valamit. Azt is észre lehet venni, hogy háromszor szerepel benne a buborék rendezés kódrészlet, mivel háromszor kell rendezni a tömböt. Sokkal szebb kódot kapunk, ha a rendezést kiemeljük egy külön eljárásba, és amikor rendezni kell, ezt az eljárást hívjuk meg:

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

void rendezes(int meret, int *tomb)
{
   int i,tmp;
   for(i=0;i<meret;++i)
      for(j=meret;j>i;++j)
         if(tomb[j]<tomb[j-1]){
            tmp=tomb[j];
            tomb[j]=tomb[j-1];
            tomb[j-1]=tmp;
         }
}

int main()
{
   int *t, i, n, tmp;

   scanf("%d",&n);

   t=(int *)malloc(sizeof(int)*n);
```

```
   /* Beolvasás: */
   for(i=0;i<n;++i)
      scanf("%d",t+i);

   rendez(n,t);

   /*Elemek szorzása az indexével: */
   for(i=0;i<n;++i)
      t[i]*=i;

   rendez(n,t);

   /*Indexek kivonása az elemből: */
   for(i=0;i<n;++i)
      t[i]-=i;

   rendez(n,t);

   /*Elemek kiírása: */
    for(i=0;i<n;++i)
       printf("%d\t",t[i]);

    return 0;
}
```

Kiemeltük a kódból a rendezést, ezáltal azt csak egyszer kellett implementálnunk. További előnye az alprogramok használatának, hogy ha kiderül, hogy valamely függvényben, vagy eljárásban hiba van, akkor azt csak egy helyen kell kijavítanunk. Ha egy kódban 100 helyen kell rendezést alkalmazni, és azt 100 helyen mindig leimplementáljuk, és kiderül, hogy rosszul, félnapi munka lenne kijavítani. De ha az egy adott feladatot végző kódrészlet csak egy helyen, egy függvényben van megírva, akkor csak egy helyen kell javítani.

Bár első látásra nem sok különbség van köztük, az eljárások és a függvények nagyban különböznek egymástól. Más programozási nyelvekben sokkal jobban eltérnek formailag is, létrehozásukhoz külön kulcsszó tartozik, a C viszont a void típussal kicsit összemossa a határokat közöttük. A legfontosabb különbségek:
A függvénynek:

- Mindig van visszatérési értéke!
- Kifejezésben hívható!

Az eljárásnak:

- Soha nincs visszatérési értéke! Azaz a visszatérési típusa mindig `void`, üres típus.
- Csak eljárás hívás kifejezésben hívható!

Tehát a függvények olyanok, mint a matematikai függvények. Például a tangens függvény tan(45) visszatérési értéke akárcsak a matematikában, a 45 fok tangense, ami 1. Ezek után a tan(45)-öt bárhol használhatom, akármilyen összetettebb kifejezésben, az mindig a helyettesítési értékét fogja jelenteni: `(x+tan(45))*(x-tan(45))`. Ezzel szemben az eljárásoknak nincs visszatérési értéke, ezért azokat nem hívhatom meg kifejezésekben. A fenti `rendez` eljárás hívás csak önálló kifejezésként állhat.

Ha a kód egy pontján meghívunk egy eljárást, vagy függvényt, az a célunk, hogy a hívás helyén látható változókban valamilyen változást érjünk el. Függvények használatánál ez magától értetődő, hiszen a függvény visszatérési értékét általában értékül adjuk egy változónak:

```
float x=tan(60);
```

Azonban az eljárásoknak nincsen visszatérési értéke, ezért azoknak vagy globális változókat, vagy valamely paraméterüket kell megváltoztatniuk. Ha egy eljárás nem változtatja meg egyik paraméterét sem, és nem változtat meg egyetlen globális változót sem, és nem ír valamely kimenetre, akkor annak az eljárásnak nincs értelme. Hiába számolunk ki, vagy oldunk meg valamit benne, az eredményt nem juttatjuk vissza a hívási környezetbe, ezért azt nem tudjuk felhasználni a későbbiekben, az eljárás lefutása után elvész. Ezért ha eljárásokat készítünk, mindig figyeljünk oda rá, hogy amit kiszámol, vagy megold, az visszajusson a hívás helyére. A fenti rendez eljárás megváltoztatja a paraméterként kapott tömböt. Rendezi az elemeit. Most nézzünk meg egy olyan eljárást, amely paraméterként kap egy tömböt, és megkeresi annak maximális elemét:

```
void max(int n, int *t)
{
   int m=t[0], i;
   for(i=1;i<n;++i)
      if(t[i]>m)
         m=t[i];

}
```

Miután lefut a ciklus, a maximális elem benne van az m változóban. Azonban mivel végetér az eljárás, az m lokális változó megsemmisül. Hogy az eljárásnak értelme is legyen, az m értéket vagy egy globális változóban, vagy egy 3. paraméterben kell visszaadni:
globális változóban: feltételezem, hogy van egy maximum nevű, egész típusú globális változó deklarálva:

```
int maximum;

void max(int n, int *t)
{
   int m=t[0], i;
   for(i=1;i<n;++i)
      if(t[i]>m)
         m=t[i];

   maximum=m;
}
```

Mivel a változó globális, ezért azt az eljáráson kívül kell létrehozni.
mutatóval: felveszünk egy 3. paramétert, egy mutatót, és az az által mutatott helyre rakjuk be az m értékét:

```
void max(int n, int *t, int *mutato)
{
   int m=t[0], i;
   for(i=1;i<n;++i)
      if(t[i]>m)
         m=t[i];

   *mutato=m;
}
```

Arra kell figyelni, hogy ha valamilyen x típusú értéket szerenénk visszaadni, akkor ahhoz egy x* típusú formális paramétert kell felvennünk, az indirekció miatt.

2. Keresés

Az alábbi példákban a keresési és rendezési algoritmusok egészeket tartalmazó tömbökre vannak megvalósítva, de kis változtatással más adatszerkezetekre is alkalmazhatóak!

2.1. Lineáris (teljes) keresés

A legegyszerűbb, és egyben leglassabb keresési algoritmus a teljes keresés. Adott egy nem rendezett tömb, és a kérdés az, hogy egy adott szám szerepele benne, és esetleg arra is kíváncsiak lehetünk, hogy hol? Mivel nem rendezett a tömb, az egészen végig kell mennünk, annak minden elemét megvizsgálva. Lineáris keresés egészeket tartalmazó tömbben:

```
int linearis_kereses(int * tomb, int n, int ertek)
{
   /*
      linearis keresés

      tomb: a tomb mutatója,
      n: a mérete,
      ertek: a keresett érték
  */

  int i;

  for(i=0;i<n;++i)
    if(tomb[i]==ertek)
      return i;

  return -1;
}
```

A `tomb` mutatóval címzett `n` elemből álló tömbben keressük az `ertek` számot. Ha megtaláltuk, visszaadjuk a szám első előfodulásának indexét, ha

nincs benne a tömbben, -1 -et adunk vissza. (Mivel tömbben negatív indexű helyen nem állhat elem, -1 -gyel jelöljük a hívási környezet számára, hogy nem szerepelt a tömbben a keresett szám)

2.2. Keresés rendezett adatszerkezetben

Azt feltételezzük, hogy a tömb rendezett. Erre építve a lineáris keresésnél általában gyorsabb algoritmust kapunk, ha csak addig keresünk, amíg a keresett szám a rendezési feltételnek megfelelően kissebb/nagyobb, mint a tömb éppen vizsgált eleme. Tehát növekvőleg rendezett tömb esetén elég addig vizsgálnunk a tömb elemeit, amíg a keresett számnál nagyobbat nem találunk, mert a tömb rendezettsége miatt a továbbiakban biztos, hogy nem fog szerepleni a keresett szám:

```
int rendezett_kereses(int * tomb, int n, int ertek)
{
   /*
      keresés növekvőleg rendezett tömbben

      tomb: a tömb mutatója
      n: a mérete
      ertek: a keresett érték
   */

   int i;

   for(i=0;!(tomb[i]>ertek) && i<n;++i)
     if(tomb[i]==ertek)
        return i;

   return -1;
}
```

A kód szinte teljesen megegyezik az előzővel, annyi eltérés van, hogy a `for`-ciklus megállási felétételében elsőként azt vizsgáljuk, hogy a tömb aktuális elemének az értéke nagyobb-e, mint a keresett érték. Amennyiben ha nem, akkor a ciklus magjában megvizsgáljuk, hogy egyenlő-e, és ha igen, visszaadjuk az indexét. Látható, hogy minden ciklus lépés során 3 feltétel vizsgálata történik: `tomb[i]<=ertek`, `i<n`, `tomb[i]==ertek`. Gyorsíthatunk a kódon, ha ezt kettőre csökkentjük:

```
int rendezett_kereses_mod(int * tomb, int n, int ertek)
{
   /*
      módosított keresés rendezett tömbben

      tomb: a tomb mutatója
      n: a mérete
      ertek: a keresett érték
   */

   int i;

   for(i=0;tomb[i]<ertek && i<n;++i);

   if(i<n && tomb[i]==ertek)
      return i;
```

```
   else
      return -1;
}
```

Ebben az esetben a ciklusmag egyetlen üres utasításból áll, tahát a ciklussal csak az a célunk, hogy a ciklusváltozót olyan pozícióba léptessük, amelyből már eldönthetjük, hogy szerepel-e a keresett érték a tömbben, vagy sem. A tömb minden vizsgált elmére csak 2 hasonlítás történik, ami gyorsabb, mint a sima rendezett keresésnél. Elegánsabbá tehetjük a kódot, ha a háromoperandusú operátort használjuk az `if`-utasítás helyett:

```
int rendezett_kereses_mod2(int * tomb, int n, int ertek)
{
   /*
      módosított keresés rendezett tömbben

      tomb: a tomb mutatója
      n: a mérete
      ertek: a keresett érték
   */

   int i;

   for(i=0;tomb[i]<ertek && i<n;++i);

   return (i<n && tomb[i]==ertek)?i:-1;
}
```

Valójában még az i változóra sincs szűkségünk, ugyanis használhatjuk a tömb méretét tartalmazó `n` változót ciklusváltozóként:

```
int rendezett_kereses_mod3(int * tomb, int n, int ertek)
{
   /*
      módosított rendezett keresés

      tomb: a tomb mutatója
      n: a mérete
      ertek: a keresett érték
   */

   for(--n;tomb[n]>ertek && n>=0;--n);

   return (n>0 && tomb[n]==ertek)?n:-1;
}
```

Ha annak az adatszerkezet a végén , amelyben keresünk, van még egy szabad hely (dinamikus adatszerkezet, pl. láncolt lista esetén ez nem kérdés), és oda berakjuk a keresett elemet, akkor elég minden ciklusléptésben csak azt vizsgálni, hogy az aktuális elem a keresett elemmel egyenlő-e. A ciklus mindenképpen megáll, hiszen az utolsó elem éppen a keresett elem, és ezután csak azt kell vizsgálnunk, hogy a megtalált elem indexe kisebb-e, mint a tömb mérete. Ha nem kisebb, tehát az utolsó elemet találtuk meg, akkor a keresett szám nem szerepel a tömbben. Ezt a módszert strázsamódszernek hívják, és az az előnye, hogy ciklusléptésenként csak egy hasonlítást végez:

```
int strazsamodszer_kereses(int * tomb, int n, int ertek)
{
   /*
```

```
      keresés strázsa módszerrel
      (feltételezzük, hogy a paraméterül kapott
tömb végén még van 1 hely)

      tomb: a tomb mutatója
      n: a mérete
      ertek: a keresett érték
   */

   int i;

   tomb[n]=ertek;
   for(i=0;tomb[i]<ertek;++i);

   return (tomb[i]==ertek && i<n)?i:-1;
}
```

Mivel a kereséseket sokszor nagyon nagy adatszerkezeteken alkalmazzák, nagyon fontos, hogy azok a lehető leggyorsabban fussanak. Kellően nagy adatszerkezeteknél a fenti 1,2,3 hasonlítással működő keresések érezhető különbséggel működnek.

2.3. Bináris keresés

Gyors keresési algoritmus, ami szintén azt feltételezi, hogy rendezett tömbben keresünk. Első lépésben megvizsgáljuk a tömb középső elemét, ha megegyezik a keresett számmal, akkor visszaadja az indexét. Ha a keresett szám kisebb, mint a középső elem, akkor a tömb első felét vizsgáljuk meg ugyanígy, ha nagyobb, akkor a második felét. Addig folytatjuk ezt, amíg az éppen vizsgált résztömb egyelemű nem lesz. Ha ez megegyezik a keresett számmal, visszadjuk az indexét, ha nem, megszakad a ciklus a feltétel miatt, és -1 -et adunk vissza:

```
int binaris_kereses(int * tomb, int n, int ertek)
{
   /*
       bináris keresés

       tomb: a tomb mutatója
       n: a mérete
       ertek: a keresett érték
   */
   int also_index = 0;
   int felso_index = n-1;
   int kozep = also_index+((felso_index-also_index)/2);

   while ( also_index <= felso_index ) {

       if ( tomb[kozep] > ertek )
           felso_index = kozep - 1;
       else if ( tomb[kozep] < ertek )
           also_index = kozep + 1;
       else
           return kozep;  /*itt biztos, hogy tomb[kozep]==ertek*/

       kozep = also_index+((felso_index-also_index)/2);
   }
```

```
   return -1;
}
```

3. Rendezés

Ahhoz, hogy egy tömbben hatékonyan tudjunk keresni, azt rendezni kell. A rendezési algoritmusoknak nagyon fontos jellemzője, hogy helyben rendeznek, vagy sem. A szélsőérték kiválasztásos rendezés és a buborék kiválasztásos rendezés helyeben rendező algoritmus, ami azt jelenti, hogy a rendezendő tömb elemeit cserélgetik, az összefésüléses rendezés viszont nem helyben rendez, segéd tömböket használ.

3.1. Szélsőérték kiválasztásos rendezés

```
void min_kivalasztasos_rendezes(int * tomb, int n)
{
  /*
     minimum kiválasztásos rendezés

     tomb: a rendezendő tömb
     n: a tömb mérete
  */

  int i,j,tmp,min;

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

     min=i;

     for(j=i+1;j<n;++j)
         if(tomb[j]<tomb[min])
            min=j;

     tmp=tomb[min];
     tomb[min]=tomb[i];
     tomb[i]=tmp;

  }
}
```

A külső for ciklus `i` ciklusváltozója szemléletesen az jelenti, hogy hányadik eleméig rendezett már a tömbünk. Ezt a ciklusváltozót a tömb méretéig növelve azt érjük el, hogy az egész tömb rendezett legyen. A külső ciklus magjában ezután azt kell elérnünk, hogy valóban az `i`. eleméig rendezett legyen a tömb.

Első lépésben az `i=0`, vagyis az első eleméig kell rendezetté tennünk (mivel a C-ben a tömbök indexelése 0-val kezdődik). Ezt úgy érjük el, hogy kiválasztjuk a tömbből a legkisebb elemet, és megcseréljük az első (0.) elemmel.

A belső ciklus végzi a legkisebb elem kiválasztását úgy, hogy annak indexét berakja a `min` változóba. (A `min` változót minden keresés előtt annak a résztömbnek az első indexére állítjuk, amelyben keresünk.) Miután tudjuk a legkisebb elem helyét, megcseréljük a legkisebb, és az i ciklusváltozó által indexelt (az első lépésben 0.) elemet.

Ezzel elértük, hogy a tömbünk az első eleméig rendezett, mivel az első eleme a legkisebb elem. Ezután véget ér a külső ciklus 1. lépése és növeljük az i ciklusváltozó értékét 1-el.

Most ugyanúgy kiválasztjuk a legkisebb elemet, de mivel a belső ciklus a `j=i` indextől indul (most `i=1`), ezért nem vesszük bele a keresésbe az előbb kiválasztott, és a 0. helyre berakott legkisebb elemet. Tehát a tömb második legkisebb elemét fogjuk kiválasztani, és azt cseréljük meg az 1. helyen álló elemmel, s ezzel már 2 elem lesz biztosan rendezett a tömb elején.

3.2. Buborék-rendezés

Hasonló az szélsőérték kiválasztásos rendezéshez, abban különbözik, hogy ahelyett, hogy a belső ciklusban kiválasztanánk a legkisebb/legnagyobb elemet, és azt megcserélnénk az külső ciklusváltozó által indexelt elemmel, ebben az esetben a belső ciklus által érintett elemet mindig a szomszédjával hasonlítjuk össze, és amennyiben kisebb/nagyobb, megcseréljük vele. Lényegében a külső ciklusváltozó megint a már rendezett részt indexeli, míg a ciklus magja szintén arról gondoskodik, hogy a legkisebb elemet eljuttassuk a rendezetlen tömb elejére:

```
void buborek_rendezes(int * tomb,int n)
{
/*
   buborék rendezés

   tomb: egészeket tartalmazó tömb címe
   n: a tömb elemszáma
*/

   int i,j,tmp;

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

}
```

4. Példa

Feladat: Legfeljebb 20 egész szám beolvasása a standard input-ról, azok elhelyezése egy tömbben, és ezután a standard inputól olvasott számokról el-

dönteni, hogy benne vannak-e a tömbben, és ha igen, akkor kiirni a standart output-ra, hogy `IGEN` vagy `NEM`.

```
#include<stdio.h>
#define SIZE 20

int binaris_kereses(int * tomb, int n, int ertek)
{
   /*
       bináris keresés

       tomb: a tomb mutatója
       n: a mérete
       ertek: a keresett érték
   */
   int also_index = 0;
   int felso_index = n-1;
   int kozep = also_index+((felso_index-also_index)/2)

   while ( also_index <= felso_index ) {

       if ( tomb[kozep] > ertek )
           felso_index = kozep - 1;
       else if ( tomb[kozep] < ertek )
           also_index = kozep + 1;
       else
           return kozep      /*itt biztos, hogy kozep==ertek*/

       kozep = also_index+((felso_index-also_index)/2)
   }

   return -1
}

void buborek_rendezes(int * tomb,int n)
{
/*
   buborék rendezés

   tomb: egészeket tartalmazó tömb címe
   n: a tömb elemszáma
*/

   int i,j,tmp;

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

}

int main()
{

   int t[SIZE],i=0,j,k;

   /* tömb feltöltése max 20 elemmel:*/
```

```
   while(i++<SIZE && scanf("%d",t+i)==1);

   /* buborék rendezéssel a tömb rendezése:*/

   buborek_rendezes(t,i);

   /* számok beolvasása és eldöntése, hogy benne vannak-e a tömbben:*/

   while(scanf("%d",&j)==1){
      if(binaris_kereses(t,i,j)!=-1)
          printf("IGEN\n");
      else
          printf("NEM\n");
   }
}
```

Látható, hogy egyszerűen csak összemásoltuk a korábban megírt függvényeinket a néhány soros `main` függvény megírásához. Ha a programunk nem csak egy `main` függvényből áll, nagyon fontos, hogy a felhasznált függvények a `main` függvény futtatásakor már láthatóak legyenek, ezért azokat a kódban a `main` függvény előtt kell elhelyezni. A fenti kód fordítási hibát okozna, ha a buborék-rendezés függvényt a `main` függvény után adnánk meg.

Hogy szokjuk a függvények használatát, a fenti szélsőérték kiválasztásos rendezés függvényünket programozzuk le úgy, hogy az két újabb, logikailag önálló függvény hívásával működjön:

```
void csere(int * a,int * b)
{
   /*
      megcseréli a két címen található egész számokat

      a: 1. cím
      b: 2. cím
   */

   int tmp;

   tmp=*a;
   *a=*b;
   *b=tmp;
}

int legkisebb_elem_indexe(int * tomb,int n)
{
   /*
      visszaadja a paraméterül kapott tömb legkisebb elemének indexét

      tomb: a tömb címe
      n: a tömb mérete
   */

   int i,min;

   min=0;

   for(i=0;i<n;++i){
      if(tomb[min]>tomb[i])
          min=i;
```

```
   }
   return min;
}

void min_kivalasztasos_rendezes(int * tomb, int n)
{
  /*
     minimum kiválasztásos rendezés

     tomb: a rendezendő tömb
     n: a tömb mérete
  */

  int i;

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

     csere(tomb+i,tomb+legkisebb_elem_indexe(tomb+i,n-i);

  }
}
```

5. Sztringek

A legtöbb program célja, hogy az ember számára értelmezhető dolgokon végezzen műveleteket. Ezek a dolgok általában számok, illetve szövegek, vagyis karaktersorozatok, amelyeket sztring-nek nevezünk. Sok programozási nyelvvel ellentétben a C nem ad külön típust a sztringek kezelésére. A C-ben a sztringeket ezért karaktertömbökkel kezeljük. A sztring végét mindig a `'\0'` karakter jelzi, ami nem más, mint a nulla-kódú karakter. Ha ez nem szerepel a sztring végén, akkor nem tudjuk, hogy hol van vége, ekkor nagyon kevés hasznos műveletet lehet vele végezni. A `string.h header` -ben nagyon sok hasznos, a sztrigek kezelésével kapcsolatos függvény van. Ezek néhány soros egyszerű függvények, de a `string.h` -t `include` -olva nem kell őket minden alkalommal megírnunk, ha sztringekkel dolgozunk. Nézzünk ezek közül néhányat:

5.1. strlen

Ez a függvény a paraméterül kapott sztring hosszát adja vissza előjeles egész számként.

```
int strlen(char * s)
{
   int i=0;
   while(s+i) ++i;
   return i;
}
```

Mivel a sztring végén szereplő `'\0'` a nulla karakterkódú karakter, ezért az a tömbben, amelynek az elemein a `while` ciklus-ban végiglépkedünk, 0-ként jelenik meg, ami feltételként HAMIS-nak felel meg. Ekkor a `while` -ciklus megáll, és a függvény visszaadja az `i` -t , ami pont a paraméterül kapott sztring hosszával lesz egyenlő.

5.2. strcmp

Ez a függvény összehasonlítja a paraméterül kapott két sztringet. Amennyiben megegyeznek, 0-t ad vissza, ha az első paramétere kisebb, akkor negatív számot, ha az első paramétere nagyobb, akkor pozitívat.

```
int strcmp(char * s, char * t)
{
   while(*s==*t && *s){
        ++s;
        ++t;
    }

    return *s-*t;
}
```

A ciklus addig megy, amíg a két tömb elemei megyegyeznek. Teljesen mindegy, hogy a tömb aktuális elemére `*s` -el hivatkozunk, és magát az `s` mutatót növeljük, vagy `s[i]` -vel, és egy különálló `i` változót növelünk. Ha valamely elemében különbözik a két sztring, vagy elértük a végüket, akkor megáll a ciklus. (`*s` magában az a feltétel, hogy a sztring végén vagyunk-e, mivel ha a `*s` valamilyen `'\0'` -tól különböző karakter, akkor IGAZ a feltétel, ha viszont `'\0'`, akkor HAMIS, tehát a sztring végén vagyunk, és mivel ezen a ponton biztos, hogy `*s==*t`, ez azt jelent, hogy a `t` sztring-nek is a végén vagyunk) Ezek után kivonjuk egymásból a két karaktert, amelynél megállta a ciklus. Amennyiben egyenlő a két sztring, akkor ez a kivonás biztos, hogy 0-0, azaz 0 lesz, tehát 0-t adunk vissza. Ha az első kisebb, mint a második, akkor negatív, ha nagyobb, akkor pozitív értéket.

5.3. strcpy

A második paraméterként kapott sztringet bemásolja az első paraméterbe. A sztringmásolásnál arra kell ügyelni, hogy az első paraméterként kapott karaktertömbbe biztos, hogy beleférjen a második paraméterként kapott sztring, tehát az első legalább annyi elemet tartalmazzon, amennyi a második sztring hossza + 1. Az első paraméterét adja vissza a függvény.

```
char * strcpy(char * s, char * t)
{
   int i=0;
   do{
      s[i]=t[i];
   }
```

```
   while(t[i++]);
   return s;
}
```

Addig megy a ciklus, amíg a t sztring aktuális eleme nem 0, tehát nem a sztring záró `'\0'` karakter. Azért használunk hátultesztelős ciklust, mert egy karaktert mindenképpen át kell másolni, mert ha üres a második paraméterként kapott másolandó sztring, akkor is szerepel benne legalább egy `'\0'` karakter. A hátultesztelős ciklusnak pedig pontosan az a tulajdonsága, hogy legalább egyszer mindig lefut a ciklus magja. Természetesen meg lehet csinálni elől-tesztelőssel is, csak nem szabad elfelejtkezni a `'\0'` karakterről.

5.4. strcat

Ez a függvény a paraméterként kapott második sztringet bemásolja az első végére, felülírva az első paramétere végén szereplő `'\0'` -t. Itt is arra kell figyelnünk, hogy a függvény hívásakor az első paraméterként kapott karakter tömb elég nagy legyen, hogy beleférjen még a második paraméterként kapott sztring. Szintén az első paraméterét adja vissza.

```
char * strcat(char * s, char * t)
{
   int i=0, j=strlen(s);
   do{
      s[i+j]=t[i];
   }
   while(t[i++]);

   return s;
}
```

Szinte teljesen megegyezik a `strcpy` sztring másoló függvénnyel, a különbség, hogy a második sztringet nem az első elejére, hanem a végére másolja.

5.5. strchr

Első paramétere egy sztring, a második egy karakter. megkeresi az első előfordulását a karakternek a sztringben. Ha szerepel benne, visszaadja az előfordulásra mutató mutatót, ha nem szerepel benne, NULL mutatót ad vissza.

```
char * strchr(char * s, char c)
{
   for(;*s;++s)
      if(*s==c)
         return s;

   return NULL;
}
```

5.6. sscanf

Az `sscanf` függvény ugyanúgy működik, mint a `scanf` , azzal a különbséggel, hogy a formátum sztringet nem a standard bemenetre, hanem az első paramétereként kapott sztringre próbálja illeszteni. Példaként írjunk eljárást, amely paraméterként kap egy sztringet, amely egy neptun-kódot tartalmaz, és utána 4 számot, vesszővel elválasztva: `aaabbb,1,2,3,4` . Az eljárás írja ki a neptunkódot, és az azt követő számok összegét tabulátorral elválasztva.

```
void pelda(char * string)
{
   char t[7];
   int a,b,c,d;

   sscanf(string,"%s,%d,%d,%d,%d",t,&a,&b,&c,&d);

   printf("%s\t%d\n",t,a+b+c+d);
}
```

5.7. sprintf

Az `sprintf` függvény működése a `printf` -éhez hasonló, a különbség az, hogy a paramétereit a formátumsztringnek megfelelő formában nem a standard kimenetre, hanem az első paraméterként megkapott sztring-be írja. Példaként írjunk olyan függvényt, amely paraméterként kap egy sztringet, amely egy neptun-kódot tartalmaz, és utána 4 számot, vesszővel elválasztva: `aaabbb,1,2,3,4` . A függvény adja vissza azt a sztringet, amely a neptunkódot, és attól tabulátorral elválasztva a számok összegét tartalmazza. (Feltételezzük, hogy a számok összege maximum 3 jegyű.)

```
char * pelda2(char * string)
{
   char t[7];
   char e[11};
   int a,b,c,d;

   sscanf(string,"%s,%d,%d,%d,%d"',t,&a,&b,&c,&d);

   sprintf(e,"%s\t%d\n"',t,a+b+c+d);

   return e;
}
```

Ellentétben a korábban tárgyalt sztringkezelő függvényekkel, a `sscanf` , és a `sprintf` NEM a `string.h header` állományban van, hanem az `stdio.h` -ban!

6. Standard input/output

A korábbiakban a standard input/output kezelésére csak a `scanf` , és `printf` függvényeket használtuk. Mindkét függvény nevében az `f` a formázott be és kimenetre utal. Ennek megfelelően ezen függvényeknek az első paramétere

minden esetben egy formátum sztring, amelyet a `scanf` megpróbál illeszteni a bemenetre, míg a `printf` ennek megfelelően ír a kimenetre. Azonban az `stdio.h` állomány nem csak ezeket a függvényeket tartalmazza az input/output kezelésére.

6.1. getchar, putchar

A `getchar` függvény visszaadja a standard bemenet következő karakterét. Ha eléri az EOF-ot, vagy hiba történt, akkor azt adja vissza. A `putchar` függvény ezzel szemben a paraméterként kapott karaktert írja ki a standard kimenetre, visszatérési értéke a kiírt karaker kódja, ha hiba történik, akkor `EOF` . A következő példaprogram egyszerűen kiírja a kimenetre azt a karaktert, amit a bemeneten olvas. Mindezt addig teszi, amíg '.' karaktert nem olvas.

```
int main()
{
   char c;
   do{
      c=getchar();
      putchar(c);
   }
   while(c!='.');
}
```

6.2. gets, puts

A `gets` függvény egy paramétert kap, egy karaktertömböt, amelybe a standard bemenetről egy sort olvas be. Egész pontosan addig olvas, amíg `'\n'` karaktert nem talál. A `'\n'`-t már nem rakja be a paraméterként kapott tömb végére, azonban egy `'\0'` -t automatikusan odarakja. Ha nem olvasott be egy karaktert sem, akkor `NULL` -t ad vissza. A `puts` függvény a paraméterként kapott karaktertömböt (sztringet) kiírja a standard kimenetre, és a végére ír még egy `'\n'` újsor karaktert. VIsszatérési értéke egy pozitív szám, ha nem történt hiba, ha hiba történt, akkor `EOF` -t ad vissza. Nézzük meg, hogyan készíthetnénk `gets` és `puts` függvényeket az eddig megismert `scanf, printf, getchar, putchar` függvények segítségével!

```
char * gets(char * s)
{
   int i=0;
   while((s[i]=getchar())!='\n' && s[i]!=EOF) ++i;
   s[i]='\0';
   return i==0?NULL:s;
}
```

A `gets` függvény megvalósítása nagyon egyszerű, beolvasunk egy karaktert, majd megvizsgáljuk, hogy az sorvége, avagy `EOF` -e. Ha egyik sem, akkor beolvassuk a következő karaktert. Ha sorvége karaktert, vagy `EOF` -t olvastunk, akkor a beolvasott sztringet lezárjuk egy `'\0'` karakterrel. Ezután visszaadjuk a beolvasott sztring mutatóját, ha legalább egy karaktert beolvastunk, különben NULL-t adunk vissza.

```
int puts(char * s)
{
   while(*s!='\0')
      if(putchar(*s)==EOF)
        return EOF;
      else
         s++;

    return putchar('\n');
}
```

Ebben az esetben karakterenként írjuk ki a kiírandó sztringet, és minden kiírásnál megvizsgáljuk a `putchar` visszatérési értékét. Ha az `EOF` akkor abbahagyjuk a kiírást, és a hívási környezetbe visszadjuk az `EOF` -t. Ha sikerül kiírni az egész sztringet, kiírunk még egy újsor karaktert, és visszaadjuk ennek a visszatérési értékét. Ugyanis ha hiba nélkül sikerül kiírni a `'\n'` karaktert, akkor annak a kódját adja vissza a `putchar` függvény, ami pozitív szám, tehát a `puts` helyes lefutását jelzi, azonban ha az utolsó kiírás `EOF` -ot ad vissza, akkor valami hiba történt a `'\n'` kiírásánál, és ezt jelezni kell a hívási környezetnek, ezért közvetlenül ezt az `EOF` -t adjuk vissza.

7. Kétdimenziós tömbök

Kétdimenziós tömbök kezelésére a C-ben több lehetőség is kínálkozik. A legkézenfekvőbb azonban nem biztos, hogy a legegyszerűbb. A kétdimenziós tömbök a memóriában minden esetben egydimenzióként jelennek meg, sorfolytonosan ábrázolva, ami azt jelenti, hogy egyik sor a másik után következik. A gyakorlatban a kétdimenziós tömböket kezelhetjük egydimenzióként, és ekkor csak logikailag használjuk őket kétdimenzióként, valamint kezelhetjük őket a C nyelv tömb típusának kompozíciójával nyert összetett típusként is.

7.1. Egydimenzióként kezelve

A kétdimenziós tömbben jelölje s a sorok, és o az oszlopok számát. Adott az i, és j indexpár, ahol i a sorindexet, j pedig az oszlopindexet jelöli. Sorfolytonos ábrázolásnál, az i sorindex azt jelenti, hogy az adott sor első elemének memóriacíme előtt i-1 teljes sor helyezkedik el. Mivel a C-ben a tömbök indexelése 0-val kezdődik, ezért valójában nem i-1, hanem i sor helyezkedik el az i. sor előtt. Vagyis, ha i-vel megszorozzuk a sorok hosszát (ami az oszlopok száma) , akkor megkapjuk az i. sor kezdőindexét: i*o. Ezek után már csak hozzá kell adnunk az oszlop indexet, j-t, hogy megkapjuk, hogy a keresett i,j indexű elemünk hol helyezkedik el sorfolytonos ábrázolással: i*(oszlopok száma)+j: i*o+j. Tehát:

```
t[i*o+j]   /* a t által reprezentált 2 dimenziós tömb i,j indexű eleme*/
```

Ebben az esetben a kétdimenziós tömb valójában egydimenziós, csak logikailag kezeljük 2 dimenzióként. Ha alprogramot írunk, a tömb paraméterként történő átadása ezért ugyanúgy történik, mint az egydimenziós tömböké, csak ne feledkezzünk meg arról, hogy mivel 2D tömbről van szó, annak 2 irányú kiterjedése van, ezért a kezelhetőség érdekében mindkét méretet adjuk át paraméterként!

7.2. Kétdimenzióként kezelve

Ha létrehozunk egy kétdimenziós tömböt a `[ ]` operátor kétszeres alkalmazásával: `int t[3][10]` , az így létrejött változó már más tulajdonságokkal rendelkezik, mint a fenti esetben. Az így létrejött 2D tömb `i,j` indexű elemére a következő módon hivatkozhatunk: `t[i][j]` . Ez kényelmesebb, mint a fenti esetben, azonban ha paraméterként akarjuk átadni egy alprogramnak, problmába ütközhetünk, ugyanis ez a `t` változó 10 elemű egészeket tartalmazó tömböknek a 3 elemű tömbje. Ezt kezelhetjük olyan változóként, amely `(int *)t[10]` típusként egy 10 elemű tömbre mutató mutató. Ekkor a `t` -t növelve egyel, a típusnak megfelelően 10*4 bájttal tovább mutat majd a t, a következő 10 elemű tömbre. Azonban nem kezelhetjük a `t` -t `int ** t` -ként, mert ez már egy egészre mutató mutatóra mutató mutató, ami nyilván nem egyezik meg egy a 10 elemő egész tömbre mutató mutatóval.

A fenti `int t[3][10]` -hez hasonló módon létrehozott változókat NEM adhatjuk át olyan alprogramoknak, amelyek formális paraméter listáján `int **` típusú változó szerepel. Ez fordítási hibát okoz. A 2 típus nem azonos, és nem is konvertálható! Az ilyen tömböket úgy adhatjuk át paraméterként, ha a formális paraméter listán `(int *)tomb[10]` típusú 10 elemű tömbre mutató mutatót, vagy `int tomb[3][10]` típusú változót használunk. Tehát az alprogram létrehozásakor legalább a 2. dimenzióját ismernünk kell a kezelendő 2D tömbnek. Ez viszont ellentmond annak, amiért alprogramokat hozunk létre. Alprogramokat azért (is) hozunk létre, hogy bizonyos problémákat általánosan, paraméterezhetően oldjunk meg, és aztán ezt a megoldást sok egyedi esetben felhasználhassuk. Ha a formális paraméterlistán egy adott méret szerepel, az nem általános megoldás.

A `int t[3][10]` tömböt úgy tudjuk átadni általánosan egy alprogramnak, ha létrehozunk egy mutatótömböt, amely minden sornak az első elemére mutat:

```
int *t2[3],i;
for(i=0;i<3;++i)
   t2[i]=t[i];
```

Ekkor a `t2` már egy mutatókat tartalmazó tömb, melyet átadhatunk olyan alprogramnak, mely paraméterlistája `int **tomb` , mivel a `t2` mutatók tömbje, vagyis mutatók sorozatának első elemére mutató mutató, azaz `int**` típusú. Az így paraméterként kapott `int **` típusú változóra alkalmazva a `[ ]` operátort, egy mutatót kapunk vissza, például `t[i]` esetben a 2 dimenziós tömb `i.` sorának mutatóját. Ezt tekinthetjük innentől egy egydimenziós tömbnek, és így indexelhetjük még egy indexel: `t[i][j]` .

7.3. Kétdimenziós tömb bejárása

A következő két függvényben egy kétdimenziós tömböt járunk be, és írjuk ki az elemeit. Az elsőben egydimenziós tömbként ábrázoljuk, a másodikban kétdimenziósként.

```
void ket_dimenzios_tomb_kiirasa(int * tomb, int s, int o)
{
/*
   két dimenziós tömb bejárása, és kiírása két index használatával

   tomb: a tömb címe
   s: sorok száma
   o: oszlopok száma
*/

   int i,j;

   for(i=0;i<s;++i){
      for(j=0;j<o;++j)
         printf("%d ",tomb[i*o+j]); //az i,j indexű elem kiírása
      printf("\n");
   }
}

void ket_dimenzios_tomb_kiirasa_mod(int **tomb, int s, int o)
{
/*
   két dimenziós tömb bejárása, és kiírása két index használatával

   tomb: a tömb mutatótömb
   s: sorok száma
   o: oszlopok száma
*/

   int i,j;

   for(i=0;i<s;++i){
      for(j=0;j<o;++j)
         printf("%d ",tomb[i][j]); //az i,j indexű elem kiírása
      printf("\n");
   }
}


int main()
{
   int t[3][10]={{1,2,3,4,5,6,7,8,9,10},
                 {10,9,8,7,6,5,4,3,2,1},
                 {1,2,3,4,5,6,7,8,9,10}};
   int *t2[3];
   int i;
   for(i=0;i<3;++i)
      t2[i]=t[i];
   ket_dimenzios_tomb_kiirasa_mod(t2,3,10);
   return 0;
}
```

7.4. Mátrix szorzás

A következő példában 2 mátrix szorzását fogjuk elvégezni a matematikai sor-oszlop kompozíciós szorzással:

```
int * matrix_szorzas(int * a,int m,int n, int * b,int p,int q)
{
/*
   mátrix szorzás sor-oszlop kompozícióval

   a: az 1. mátrix(két-dimenziós tömb) címe
   m: a sorainak száma
   n: az oszlopainak száma
   b: a 2. mátrix címe
   p: a sorainak száma
   q: az oszlopainak száma
*/

   /* e-vel címezzük majd az eredmény mátrixot: */

   int * e;
   int i,j,k,sum;

   /*az 1. mátrix második, és a 2. mátrix első dimenziójának egyezni kell
   ahhoz, hogy összeszorozhassuk őket*/

   if(n!=p){
      printf("Nem összeszorozható mátrixok!");
      return NULL;
   }

   /*lefoglaljuk az eredménymátrixnak megfelelő méretű területet*/
   e=(int *)malloc(sizeof(int)*m*q);

   for(i=0;i<m;++i)
      for(j=0;j<q;++j){
         sum=0;
         for(k=0;k<n;++k){
             sum+=a[i*m+k]*b[k*q+j];
         }
         e[i*q+j]=sum;
      }

   return e;
}
```

Példa a függvény alkalmazására:

```
int main(int argc, char **argv)
{
   int t[6]={1,2,3
             4,5,6};
   int s[12]={1,2,3,4,
              5,6,7,8,
              9,10,11,12};

int * eredmeny,i,j;

   eredmeny=matrix_szorzas(t,2,3,s,3,4);
   for(i=0;i<2;++i){
       for(j=0;j<4;++j)
           printf("%d ",eredmeny[i*4+j]);
```

```
        printf("\n");
    }
}
```

8. Parancssori argumentumok

Parancssoros programok általában használnak parancssori argumentumokat, amelyekkel a működésüket szabályozhatjuk. Gondoljunk csak a legegyszerűbb dir, vagy ls könyvtár listázó parancsokra. Egy C-ben írt program a parancssori argumentumokat a `main` függvény paramétereiben kapja meg, amennyiben léteznek. Ha létezenek, akkor lennie kell egy `int` típusú formális paraméternek, amelyben a parancssori argumentumok számát kapja meg, valamint egy `char **` típusú sztringtömbnek, amelyben pedig magukat a paramétereket. Amire oda kell figyelni, hogy az első parancssori argumentum, vagyis a sztringtömb mindig létező 0-indexű eleme az maga a programindítási parancs.

A következő program kilistázza a parancssori argumentumait.

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int i;
   for(i=0;i<argc;++i)
      printf("%s\n",argv[i]);

   return 0;
}
```

Ezek után a programot a következő argumentumokkal futtatva az azt követő kimenetet láthatjuk:

```
C:\Projekt1.exe alma korte szilva
C:\C:\Projekt1.exe
C:\alma
C:\korte
C:\szilva
```

8.1. Kapcsolók használata

Parancssoros futtatásnál általában megadhatunk egy programnak bizonyos opciókat, melyek a futását befolyásolják. Ezeket parancssori argumentumként kapja meg C-nyelvű programok esetén a `main` -függvény. A konvenció az, hogy ha egy ilyen argumentum NEM kötőjellel kezdődik, akkor az valamilyen adat, mellyel a program dolgozik. Ha kötőjellel kezdődik, akkor viszont valamilyen opcionális logikai információ, amely a program futását befolyásolja.

Készítsünk programot, amely parancssori argumentumként egy angol mondatot kap, és kiírja azt a standard kimenetre. Ha szerepel az argumentumok közt a `-r` kapcsoló, akkor hátulról visszafelé írja ki, és ha szerepel a `-minus kar1kar2...` kapcsoló, akkor kihagyja a kiírandó sztringből a kar1, kar2... karaktereket. Tehát a program a következő módon működik:

```
C:\Projekt1.exe We are the champions, my friends!
C:\We are the champions, my friends!
C:\Projekt1.exe We are the champions, my friends! -r
C:\!sdneirf ym ,snoipmahc eht era eW
C:\Projekt1.exe We are the champions, my -minus abcdef friends!
C:\We r th hmpions, my rins!
C:\Projekt1.exe We are -r the chamions, my -minus abcdef friends!
C:\!snir ym ,snoipmh ht r W
```

A program:

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

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

   int r=0,i,j,l=0;
   char * s=NULL;
   char * kar=NULL;

   for(i=1;i<argc;++i){
      if(strcmp(argv[i],"-r")==0){
         r=1;
         continue;
      }
      if(strcmp(argv[i],"-minus")==0){
         kar=argv[++i];
         continue;
      }

      l+=strlen(argv[i]);
   }

   s=(char *)malloc(sizeof(char)*(l+argc));

   l=0;
   for(i=1;i<argc;++i){
      if(strcmp(argv[i],"-r")==0)
         continue;
      if(strcmp(argv[i],"-minus"==0){
         ++i;
         continue;
      }

      strcpy(s+l,argv[i]);
      l+=argv[i];
      s[l++]=' ';
   }
   s[l-1]='\0';


   if(!r){
      for(i=0;i<l-1;++i)
         if(!strchr(kar,s[i])
            printf("%c",s[i]);

   }
   else{
      for(i=l-1;i>=0;--i)
         if(!strchr(kar,s[i])
```

```
            printf("%c"',s[i]);
   }

   return 0;
}
```

Kapcsolók feldolgozásánál érdemes minden kapcsolóhoz létrehozni egy külön logikai változót, és azt 0 értékkel inicializálni. Itt is ez történik, az r, és kar változók az r és minus kapcsolóknak felelnek meg. Az r kapcsoló logikai kapcsoló, ezért az int típusú, a kar változóban viszont a -minus kapcsolót követő karaktersorozatot fogjuk tárolni, amennyiben van ilyen. Az első ciklusban végigmegyünk az argumentumokon, ha valamely kapcsolóval találkozunk, akkor megváltoztatjuk a kapcsolónak megfelelő változó értékét, ezután a változót megvizsgálva bármikor eldönthetjük, hogy szerepelt-e valamelyik kapcsoló. Ezenkívül az első ciklusban mérjük le a kapcsolókon kívüli argumentumok hosszát. Dinamikusan lefoglalunk megfelelő méretű területet a mondatnak, majd a szavait bemásoljuk erre a területre a következő ciklusban. Ezután már csak a kiírás van hátra. If feltételben vizsáljuk, hogy szerepelt-e az -r kapcsoló, majd felhasználjuk a strchr függvényt, hogy migtudjuk, az éppen kiírandó karakter szerepel-e a nem kiírandó karaktereket tartalmazó, paraméterként kapott sztringben.

9. Gyakorló feladatok

9.1. 2005/06, 2. zh, 1. feladat

Írj eljárást, amely paraméterül megkap egy tetszőleges méretű kétdimenziós, long típusú elemeket tartalmazó tömböt, és kiírja minden olyan elemét, amely értéke indexei szorzatával egyenlő!

```
void kiir(long **tomb, int s, int o)
{
   int i,j;
   for(i=0;i<s;++i)
      for(j=0;j<o;++j)
         if(tomb[i][j]==i*j)
            printf("%d\n",tomb[i][j]);
}
```

A feladat megoldásához felhasználhatjuk azt a kódot, amelyeben a statikusan kétdimenziós tömbként deklarált tömböt járjuk be. Annyi teendőnk van, hogy a kiirás előtt egy if utasításban ellenőrizzük, hogy az adott elem egyenlő-e az indexei szorzatával. Amennyiben a feladatban nem lett volna megadva, hogy a tömböt mutatótömbként, long **tomb változóval kell megkapnia az eljárásnak, a következő kód is jó lett volna:

```
void kiir(long *tomb, int s, int o)
{
   int i,j;
   for(i=0;i<s;++i)
      for(j=0;j<o;++j)
         if(tomb[i*o+j]==i*j)
             printf("%d\n"',tomb[i*o+j]);
}
```

9.2. 2005/06, 2. zh, 3. feladat

Írj függvényt, amely paraméterként kap két darab, egy napon belüli időpontot másodperc pontossággal tároló sztringet, és visszaadja a köztük eltelt másodpercek számát!

```
int eltelt_masodpercek(char *t0, char *t1);
```

Az időpontok az alábbi két alak egyikében vannak megadva:

```
ŰŰ óra PP perc MM másodperc

ŰŰ:PP:MM
```

Példa:

```
eltelt_masodpercek("18:12:09", "07 óra 11 perc 46 másodperc");
```

Kimenet:

```
39623
```

A megoldáshoz valahogyan a paraméterként kapott sztringekből ki kell nyernünk a számokat, vagyis az óra, perc és másodperc értékeket. Ezt csak azután tudjuk megtenni, hogy eldöntöttük, hogy milyen formátumú sztringel van dolgunk. Ehhez mindkét paraméterként kapott sztringet ugyanúgy fogjuk feldolgozni: megvizsgáljuk, hogy milyen formátumban van megadva az időpont. Legegyszerűbb, ha megnézzük a 2 indexű karakter :-e, vagy sem. Ha a 3. karakter, vagyis a 2 indexű kettőspont, akkor az első formátumban lévő string-el van dolgunk, ha nem :, akkor a második formátumúval. Eztuán a számjegyeket könnyen számmá alakíthatjuk: az adott számjegyből, ami még egy karakter, kivonjuk a '0' karakter kódját, és így megkapjuk a számjegy értékét, amit aztán megszorzunk a helyiértékének megfelelő 10 hatvánnyal.

```
int eltelt_masodpercek(char * t1,char * t2))
{

   int o1, p1, mp1, o2, p2, mp2;

   /*ezekbe a változókba nyerjük ki a sztringekből a számokat*/

   if(t1[2]==':'){
      o1=(t1[0]-'0')*10+(t1[1]-'0');
      p1=(t1[3]-'0')*10+(t1[4]-'0');
      mp1=(t1[5]-'0')*10+(t1[6]-'0');
   }
   else{
    o1=(t1[0]-'0')*10+(t1[1]-'0');
```

```
    p1=(t1[7]-'0')*10+(t1[8]-'0');
    mp1=(t1[15]-'0')*10+(t1[16]-'0');
   }

   if(t2[2]==':'){
      o2=(t2[0]-'0')*10+(t2[1]-'0');
      p2=(t2[3]-'0')*10+(t2[4]-'0');
      mp2=(t2[5]-'0')*10+(t2[6]-'0');
   }
   else{
    o2=(t2[0]-'0')*10+(t2[1]-'0');
    p2=(t2[7]-'0')*10+(t2[8]-'0');
    mp2=(t2[15]-'0')*10+(t2[16]-'0');
   }


mp1+=o1*60*60+p1*60;
mp2+=o2*60*60+p2*60;

return mp1>mp2?mp1-mp2:mp2-mp1;

}
```

Rövidebb megoldást kaphatunk azonban, ha az `sscanf` függvényt használjuk:

```
int eltelt_masodpercek(char * t1,char * t2))
{

   int o1, p1, mp1, o2, p2, mp2;

   /*ezekbe a változókba nyerjük ki a sztringekből a számokat*/

  if(sscanf(t1,"%d:%d:%d",&o1,&p1,&mp1)==3);
    else sscanf(t1,"%d óra %d perc %d másodperc",&o1,&p1,&mp1);
  if(sscanf(t2,"%d:%d:%d",&o2,&p2,&mp2)==3);
    else sscanf(t2,"%d óra %d perc %d másodperc",&o2,&p2,&mp2);

mp1+=o1*60*60+p1*60;
mp2+=o2*60*60+p2*60;

return mp1>mp2?mp1-mp2:mp2-mp1;

}
```

Az sscanf függvény ugyanúgy működik, mint a scanf, csak a formátum sztring a második paramétere, amelyet aztán az első paraméterként megadott sztringre próbál illeszteni, csakúgy, mint a scanf az standard bemenetre. Az első `if` utsításban próbáljuk illeszteni az első típusú időformátumnak megfelelő formátumsztringet. Ha a `scanf` függvény 3-at ad vissza, az azt jelenti, hogy sikerült illesztenie a sztringet. Ha nem hármat ad vissza, akkor az `else` ágban illesztjük a másik típusú formátumsztringet, mivel ekkor már biztos, hogy ilyen formátumú a bemenet.

9.3. 2006/05, 2. házifeladatsor, 1. feladat

Írj programot, amely a szabványos bemenetről egész értékeket olvas be soronként. A beolvasást vagy a bemenet vége állítja meg, vagy az, ha már 30 számot

beolvastunk. Ezután a program írja a kimenetre azon indexek összegét (0-tól indexelünk), amelyekhez tartozó értékek az összes beolvasott érték átlagánál nem nagyobbak!

```
#include <stdio.h>

int main()
{

   int i,j,t[30],s=0;
   float atlag;

   for(i=0; i<30 && scanf("%d",t+i)==1; ++i))
      sum+=t[i];

   atlag=(float)sum/i;

   for(j=0;j<i;++j)
      if(t[j]<=sum)
         printf("%d\n",j);
}
```

Arra kell figyelni, hogy az i változót a sehol ne írjuk felül, mivel az első ciklus lefutása után abban tároljuk a tömb tényleges hosszát, vagyis a beolvasott számok számát.

9.4. 2006/05, 2. házifeladatsor, 4. feladat

Írj programot, amely parancssori argumentumként megkap egy sztringet, és a bemenetről soronként szóközzel elválasztott számpárokat olvas. A számpárok a sztring egy részét azonosítják, ahol az első szám a részsztring kezdőindexe (0-tól indexelünk), a második pedig a részsztring hossza. Mindkét szám nemnegatív egész. A program minden beolvasott számpárra törölje a sztringből a számpár által kijelölt részsztringet, majd az eredményt írja ki egy sorban! (A sztring végül elfogyhat.) A számpár által kijelölt intervallum túlnyúlhat a sztringen, ilyenkor a sztringbe eső részt kell törölni (esetleg semmit). A példában szereplő idézőjelek nem részei a sztringnek, azok csak az argumentum határolására szolgálnak.

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

int strlen(char * s)
{

   /* Visszaadja a paraméterként kapott string hosszát */

   int i=0;
   while(s[i]!='\0')
      ++i;

   return i;
}

void strcpy(char * s, char * t)
```

```
{

   /* Bemásolja a második paraméterként kapott stringet az elsőbe */

   int i,j;
   for(i=0;t[i]!='\0';++i)
      s[i]=t[i];
   s[i]='\0';
}

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

   int n=0,i,j,a,b,k;
   char * s;

   /* Lemérjük a parancssori argumentumok össz-hosszát. */

   for(i=;i<argc;++i)
      n+=strlen(argv[i]);

   /* Dinamikusan lefoglaljuk az össz-hossz + szavak száma
   hosszú területet, a szavak számát azért adjuk hozzá, hogy
   legyen hely a szavak közti szóközöknek */

   s=(char *)malloc(sizeof(char)*(n+argc));

   /* Most bemásoljuk a parancssori argumentumokat egymás
   után az s string-be, közéjük szóközöket írva. */

   j=0;
   for(i=0;i<argc;++i){
      strcpy(s+j,argv[i]);
      j=j+strlen(argv[i]);
      s[j]=' ';
   }

   /* Nem feledkezünk meg a sztring végén a '\0' -ról */

   s[j]='\0';
   printf("%s\n",s);

   /* Számokat olvasunk be kettesével */
   /* A j változóban mindig a sztring aktuális végét tartjuk nyilván */

   while(scanf("%d %d",&a,&b)==2){

   /* Ha a kivágandó részsztring túlnyúlik a sztringünkön, akkor egyszerűen
   a kivágandó részsztring első beűje helyett /0-t írunk, ezzel levágtuk a
   teljes hátralévő részét a sztringnek */

   if(b>j){
           s[a]='\0';
           j=0;
           }

   /* Ha a kivágandó részsztring benne van az aktuálisban, akkor a kivágandó
   rész végétől kezdődő részt rámásoljuk a kivágandó részre */

   else{
        for (k=b;s[k]!='\0';++k)
            s[a+k-b]=s[k];
```

```
        s[a+k-b]=s[k];
        j=k;
   }

   printf("%s\n",s);
   }
   return 0;
}
```

10. Dinamikus adatszerkezetek

A láncolt listák előnye a tömbökkel szemben, hogy nem kell előre tudnunk, hány darab adatot szerenék elraktározni az adatszerkezetben. Mindig lennie kell egy mutatónak, ami a lista első elemére mutat, ezt `fej`-nek nevezzük.

10.1. Egyirányba láncolt listák

Mivel a fej változhat, és a listakezelő függvények egyszeres mutatóként kapják azt, ezért azokat úgy fogjuk megírni, hogy mindig a lista fejét adják vissza, s ezáltal ha megváltozik a lista feje, a változás eljut hívási környezetbe is. (ezt nagyon sokféle módon meg lehet oldani) Egy egész számot tartalmazó listaelemet a következő struktúra fog reprezentálni:

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

A `kov` mező, arra szolgál, hogy a lista következő elemének a címét tartalmazza.

10.1.1. Listakezelés függvényekkel

Új listaelem létrehozása:

```
struct elem * uj_elem(int adat)
{
/*
   új listaelem létrehozása, az új elem az adat-ot fogja tartalmazni

   adat: az új elem tartalma
*/
   struct elem * uj=(struct elem *)malloc(sizeof(struct elem));

   uj->kov=NULL;
   uj->adat=adat;

   return uj;
}
```

Beszúrás a lista elejére:

```
struct elem * lista_elejere_beszur(struct elem * fej, int adat)
{
/*
   beszúrás a lista elejére

   fej: a lista aktuális feje;
   adat: a beszúrandó elem;
*/

   struct elem * uj=uj_elem(adat);

   uj->kov=fej;

   return uj;
}
```

Mivel megváltozik a lista feje, és éppen ezt adjuk vissza, nagyon fontos, hogy a hívási környezetben megfelelő módon hívjuk a függvényt:

```
fej=lista_elejere_beszur(fej,3);
```

Lista bejárása és elemeinek kiírása:

```
void bejar_kiir(struct elem * fej)
{
/*
   lista bejárása, és elemeinek kiírása

   fej: a lista aktuális feje
*/

   while(fej!=NULL){
      printf("%d\t",fej->adat);
      fej=fej->kov;
   }
}
```

A ciklus magjában a `fej=fej->kov;` utasítás lépteti a fej lokális változót a lista következő elemére.

Beszúrás lista végére:

```
struct elem * lista_vegere_beszur(struct elem * fej, int adat)
{
/*
   lista végére beszúrás

   fej: a lista feje
   adat: a beszúrandó adat
*/
   struct elem * tmp=fej, *uj=uj_elem(adat);

   if(fej==NULL)
      return uj;

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

  tmp->kov=uj;

  return fej;
}
```

Mivel a lista végére akarunk beszurni egy elemet, és nincs mutatónk az utolsó elemre, el kell mennünk a lista végére. Ez a `while`-ciklusban történik. Azonban mielőtt elindulnánk a lista végére, ellenőriznünk kell, hogy a lista nem üres-e. Amennyiben üres a lista, vagyis a `fej` a `NULL`-t tartalmazza, abban az esetben nem hivatkozhatunk a mezőire, vagyis nem adhatjuk ki a `fej->kov` utasítást, mert az futási hibát okozna. Ha tehát üres a lista, akkor visszaadjuk az új elemet, mint egyelemű listát, ami így természetesen utolsó elem is egyben.

Törlés lista elejéről:

```
struct elem * elejerol_torol(struct elem * fej)
{
/*
   a fej parameterkent kapott lista elso elemenek torlese

   fej: mutato a listara
*/
   struct elem * tmp=fej;

   if(fej==NULL)
      return NULL;

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

   return fej;
}
```

Dinamikus adatszerkezetből történő törlésnél nagyon fontos, hogy használjuk a `free(void *)` függvényt, és felszabadítsuk a törölt listaelemnek lefoglalt memóriaterületet! Annak vizsgáltatára, hogy a `fej` nem egyenlő-e `NULL`-al, ugyanazért van szűkség, mint a fenti esetben: Az `if` utasítást követő utasításban a `fej kov` mezőjére hivatkozunk, azonban egy üres `NULL` elemnek nincsenek mezői, ezért futási hibát kapnánk! Amennyiben a fej üreslistára mutat, visszaadjuk a hívási környezetnek az üreslistát.

Törlés lista végéről:

```
struct elem * vegerol_torol(struct elem * fej)
{
/*
   a parameterkent kapott lista utolso elemenek torlese

   fej: a lista elso eleme
*/
   struct elem * tmp=fej;

   if(fej==NULL)
      return NULL;

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

   while(tmp->kov->kov!=NULL)
      tmp=tmp->kov;
```

```
   free(tmp->kov);
   tmp->kov=NULL;

   return fej;
}
```

Az első lépés ebben az esetben is, hogy ellenőrizzük, hogy a lista üres-e. Amennyiben üres, visszaadjuk az üreslistát. Ha nem üres, akkor megnézzük, hogy egyelemű-e a lista, vagyis a `fej->kov==NULL`. Ha egyelemű, akkor felszabadítjuk azt az egy elemet, és visszaadjuk az üreslistát. Ha többelemű, akkor elmegyünk az utolsó előtti elemig, és annak a mutatóját állítjuk `NULL`-ra.

Lista törlése:

```
void lista_torles(struct elem * fej)
{
/*
   lista minden elemének felszabadítása

   fej: a felszabadítandó lista feje
*/
   struct elem * tmp;

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

Példa:

```
#include <stdlib.h>

int main()
{

   /* kezdetben a fej-et inicializáljuk az üreslistával, vagyis NULL-al */

   struct elem * fej=NULL;

   fej=elejere_beszur(fej,3);
   fej=elejere_beszur(fej,4);
   fej=vegere_beszur(fej,1);
   kiir(fej);

   /* A standard outputon a 4 3 1 számsorozat jelenik meg. */

   fej=vegerol_torol(fej);
   kiir(fej);

   /* A standard outputon a 4 3 számsorozat jelenik meg. */

   lista_torles(fej);

   return 0;
}
```

10.1.2. Listakezelés eljárásokkal

Eljárások használata esetén a listakezelő algoritmusok megegyeznek a fentiekkel, csak a hívási környezet nem a függvény visszatérési értékeként kapja vissza a fejet, hanem a módosításokat csak a paraméterként, kétszeres indirekcióval kapott `struct elem ** fej`-ben végezzük. Az `uj_elem` függvény megegyezik a fentivel. **Lista elejére beszúrás**

```
void elejere_beszur(struct elem ** fej, int adat)
{
/*
   lista elejére beszúrás

   fej: a hívási környezetben a listafejmutató címe
   adat: a beszúrandó szám
*/
struct elem * uj=uj_elem(adat);

uj->kov=*fej;
*fej=uj;
}
```

Lista végére beszúrás

```
void vegere_beszur(struct elem ** fej, int adat)
{
/*
   lista végére beszúrás

   fej: a hívási környezetben a listafejmutató címe
   adat: a beszúrandó szám
*/
   struct elem *tmp=*fej, *uj=uj_elem(adat);

   if(*fej==NULL)
      *fej=uj;
   else{
      while(tmp->kov!=NULL)
         tmp=tmp->kov;
      tmp->kov=uj;
   }
}
```

Törlés lista elejéről

```
void elejerol_torol(struct elem ** fej)
{
/*
lista elejéről törlés

fej: a hívási környezetben a listafejmutató címe
*/
struct elem *tmp=*fej;

if(*fej!=NULL){
   *fej=(*fej)->kov;
   free(tmp);
}
}
```

Törlés lista végéről

```
void vegerol_torol(struct elem ** fej)
{
/*
   lista utolsó elemének törlése

   fej: a hívási környezetben a listafejmutató címe
*/
   struct elem *tmp = *fej;

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

Példa

```
int main()
{

   /* kezdetben a fej-et inicializáljuk az üreslistával, vagyis NULL-al */

   struct elem * fej=NULL;

   elejere_beszur(&fej,3);
   elejere_beszur(&fej,4);
   vegere_beszur(&fej,1);
   kiir(fej);

   /* A standard outputon a 4 3 1 számsorozat jelenik meg. */

   vegerol_torol(&fej);
   kiir(fej);

   /* A standard outputon a 4 3 számsorozat jelenik meg. */

   lista_torles(fej);

   return 0;
}
```

Érdemes összehasonlítani az listakezelő eljárásokat a listakezelő függvényekkel!

10.2. Két irányba láncolt listák

A két irányba láncolt listák előnye, hogy mivel két végük van, akármelyik elemen állva meghatározhatjuk a lista végeit. Egyirányba láncolt listáknál ha elveszítettük a fejmutatót, később nem tudtuk felhasználni az első elemeit a listának. Kétirányba láncolt listánál minden listaelemnek 2 mutatója van, az egyik a megelőző, a másik a rákövetkező elemre mutat. Ennek megfelelően bármely elemen állva el tudunk menni a lista bármelyik végéig. Az egészeket

tároló kétirányba láncolt lista egy eleme a következő struktúrával reprezentálható:

```
struct elem{
              int adat;
              struct elem * prev;
              struct elem * next;
};
```

Ebben az esetben egy listát nem egy mutató, hanem kettő reprezentál. Egy két irányba láncolt lista egy fej, és egy vég mutatókból, és köztük tetszőleges számú elemből áll. Ahhoz, hogy a két mutatót együtt tudjuk kezelni, létrehozunk egy újabb struktúrát, amivel egy egységbe foghatjuk őket:

```
struct lista{
              struct elem * fej;
              struct elem * veg;
};
```

Hogy a listakezelő függvények vagy eljárások a listát ábrázoló struktúrát egy *-os mutatóként kapják, és visszaadják, vagy 2 *-os mutatóként kapják, és nem adnak vissza semmit, az a fenti esethez hasonlóan teljesen mindegy, csak a függvények meghívására kell odafigyelni.

Új elem létrehozása

```
struct elem * uj_elem{int adat)
{
/*
   új listaelem létrehozása

 adat: a listaelem tartalma
*/

   struct elem * uj=(struct elem *)malloc(sizeof(struct elem));

   uj->adat = adat;
   uj->next=uj->prev=NULL;

   return uj;
}
```

A függvény megegyezik az egyirányba láncolt lista ujelem függvényével, annyi a különbség, hogy most nem a `kov` mutatót nullázzuk, ki, hanem a `prev` és a `next` mutatókat is. Tehát ez a függvény létrehoz egy listaelemet, amely a beszúrandó számot tartalmazza, és a mutatói sehová sem mutatnak.

Lista elejére beszúrás

```
void elejere_beszur(struct lista * l, int adat)
{
/*
   kétirányba láncolt lista elejére beszúrás

   l: a listát reprezentáló struktúra
   adat: a beszúrandó szám
*/

   struct elem * uj=uj_elem(adat);
```

```
   if(l->fej=NULL){
      l->fej=uj;
      l->veg=uj;
   }
   else{
      uj->next=l->fej;
      l->fej->prev=uj;
      l->fej=uj;
   }
}
```

Mivel a kétirányba láncolt lista teljsen szimmetrikus adatszerkezet, ezért a lista elejét és végét módosító eljárások szerkezetileg megegyeznek, csak a mutatók térnek el egymástól.

Lista végére beszúrás

```
void vegere_beszur(struct lista * l, int adat)
{
/*
   kétirányba láncolt lista elejére beszúrás

   l: a listát reprezentáló struktúra
   adat: a beszúrandó szám
*/

   struct elem * uj=uj_elem(adat);

   if(l->fej==NULL){
      l->fej=uj;
      l->veg=uj;
   }
   else{
      uj->prev=l->veg;
      l->veg->next=uj;
      l->veg=uj;
   }
}
```

Törlés lista elejéről

```
void elejerol_torol(struct lista * l)
{
/*
   kétirányba láncolt lista elejéről törlés

   l: a listát reprezentáló struktúra
*/
   if(l->fej!=NULL){
      if(l->fej==l->veg){
         free(l->fej);
         l->fej=l->veg=NULL;
      }
      else{
         l->fej=l->fej->next;
         free(l->fej->prev);
         l->fej->prev=NULL;
      }
   }
}
```

Lista végéről törlés

```
void vegerol_torol(struct lista * l)
{
/*
   kétirányba láncolt lista végéről törlés

   l: a listát reprezentáló struktúra
*/
   if(l->fej!=NULL){
      if(l->fej==l->veg){
         free(l->fej);
         l->fej=l->veg=NULL;
      }
      else{
         l->veg=l->veg->prev;
         free(l->veg->next);
         l->veg->next=NULL;
      }
   }
}
```

```
void vegerol_torol(struct lista * l)
{
/*
   kétirányba láncolt lista végéről törlés

   l: a listát reprezentáló struktúra
*/
   if(l->fej!=NULL){
      if(l->fej==l->veg){
         free(l->fej);
         l->fej=l->veg=NULL;
      }
      else{
         l->veg=l->veg->prev;
         free(l->veg->next);
         l->veg->next=NULL;
      }
   }
}
```

Mindkét törlő függvény ugyanolyan szerkezetű. Ha a `fej` `NULL`, akkor nyilván a `veg` is `NULL`, vagyis üres a lista, és ekkor nem teszünk semmit, mivel üres listából nem tudunk törölni. Ha a lista feje és vége megegyezik, akkor a listának csak egy eleme van. Nincs más dolgunk, minthogy ezt felszabadítsuk, és beállítsuk a lista végeit `NULL` értékekre, mivel kiürült a lista. Ha viszont 1-nél több eleme van a listának, akkor a megfelelő mutatót a következő(, vagy megelőző) elemre állítjuk, és az eddigi első(, vagy utolsó) elemet felszabadítjuk.

```
void elejerol_kiir(struct lista * l)
{
struct elem * tmp=l->fej;

while(tmp!=NULL){
printf("%d\t", tmp->adat);
tmp=tmp->next;
}
```

```
}

void vegerol_kiir(struct lista * l)
{
struct elem * tmp=l->veg;

while(tmp!=NULL){
printf("%d\t", tmp->adat);
tmp=tmp->prev;
}
}

int main(int argc, char *argv[])
{
struct lista l1;
struct lista l2;
l1.fej=l1.veg=NULL;
l2.fej=l2.veg=NULL;

/* a két lista feltöltése ugyanolyan elemekkel,
egyiket a végéről, másikat az elejéről */

elejere_beszur(&l1,2);
vegere_beszur(&l2,2);
elejere_beszur(&l1, 4);
vegere_beszur(&l2, 4);
elejere_beszur(&l1, 5);
vegere_beszur(&l2, 5);

/* a két lista elemeienk kiírása: a két számsornak
meg kell egyeznie */
elejerol_kiir(&l1);
printf("\n");
vegerol_kiir(&l2);

/* egy-egy elem eltávolítása */
elejerol_torol(&l1);
vegerol_torol(&l2);

printf("\n");
elejerol_kiir(&l1);
printf("\n");
vegerol_kiir(&l2);

  return 0;
}
```

A fenti példában a `main` függvényben a két listát reprezentáló `l1`, és `l2` változókat statikusan, egyszerű deklarációval hoztuk létre, majd a címüket adtuk át a listakezelő eljárásoknak, amelyek egy-egy címet várnak paraméterként. A két `struct lista *` típusú változót hoztuk volna létre, akkor nem kellett volna alkalmazni az `&` (címe) operátort az eljárások hívásánál, mivel mutatókat, vagyis címeket adtunk volna át az eljárásoknak. Azonban: az utóbbi esetben csak a mutatóknak foglalódik le hely a memóriában, és magának a struktúrának nem. Ezért: ha mutatókat hoztunk volna létre, akkor külön utasításban le kellett volna foglalni helyet egy `struct lista` típusú struktúrának, amire mutathatnak:

```
struct lista * l1, *l2;
```

```
l1=(struct lista *)malloc(sizeof(struct lista));

l1->fej=l1->veg=NULL;
l2->fej=l2->veg=NULL;

elejere_beszur(l1, 2);
vegere_beszur(l2, 2);
```

A fenti kódrészlet futtatásakor az `l2->fej=l2->veg=NULL` utasításnál kapnánk futási hibát, mert az `l2` változó még nem mutat sehová! Hogy elkerüljük a főprogramban a dinamikus memóriafoglalást, és mutató manipulációt, létrehozhatunk egy `uj_lista` függvényt, ami az `uj_elem`-hez hasonlóan egy új, NULL-ra állított mezőkkel rendelkező listát ad vissza:

```
struct lista * uj_lista()
{
   struct lista * uj=(struct lista *)malloc(sizeof(struct lista));
   uj->fej=uj->veg=NULL;

   return uj;
}
```

Ezután a főprogramban használhatjuk a következőt:

```
struct lista * l1=uj_lista();

elejere_beszur(l1, 2);
```

10.3. Verem

A verem adatszerkezet a legegyszerűbb dinamikus adatszerkezetek közé tartozik. 3 műveletet értelmezünk rajta:

- PUSH - elem belerakása a verembe
- TOP - az felső elem elérése a veremből
- POP - a felső elem eltávolítása a veremből

A verem LIFO adatszerkezet, azaz Last In First Out, vagyis amit utoljára beleraktunk, azt tudjuk elsőként kivenni. Úgy lehet elképzelni, mint ténylegesen egy vermet, gödröt. Ha beleteszünk valamit egy gödörbe, akkor azt utána kivehetjük belőle. Ha viszont beleteszünk még valamit, szemléletesen rátesszük az előző elemre, akkor már csak azt tudjuk kivenni a veremből, amit utoljára beleraktunk, hiszen az azt megelőző elemken rajta van, és azokat felülről nem tudjuk elérni. Egészeket tároló verem megvalósítása egyirányba láncolt listával:

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

struct verem * uj_elem(int szam)
```

```
{
   /*
      létrehoz egy új listaelemet
   */
   struct verem *  uj=(struct verem *)malloc(sizeof(struct verem));
   uj->kov=NULL;
   uj->adat=szam;
   return uj;
}

void PUSH(struct verem ** v, int szam)
{
   /*PUSH művelet, valójában lista elejére beszúrás*/

   struct verem * uj=uj_elem(szam);
   uj->kov=*v;

   *v=uj;
}

int TOP(struct verem * v)
{
   /*Az első elem elérése*/

   return v->adat;
}

void POP(struct verem ** v)
{
   /*POP művelet, valójában lista elejéről törlés*/

   struct verem * tmp=*v;

   if(*v==NULL) return;
   *v=(*v)->kov;
   free(tmp);
}
```

A `TOP` művelet több módon megvalósítható: ebben az esetben a verem elemeiben tárolt adatot adja vissza, látható az `int` típusú visszatérési értékéből. Ha egy verem elemeiben több összetartozó adatot kell tárolni, megvalósítható úgy, hogy definiálunk egy struktúrát, ami pontosan a veremben tárolandó összetartozó elemeket tartalmazza, és ezt használjuk visszatérési típusként a `TOP` műveletben. Ha egy elem több különböző adatot tartalmaz további megoldás lehet, ha az adatokat output paramétereken keresztül adjuk át a hívási környezetnek.

A `POP` műveletnek lehet olyan megvalósítása, amely magában foglal egy `TOP`-ot is, azaz valamilyen módon visszaadja a verem legfelső elemében tárolt adatokat is.

Valójában a műveleteket nagyon sokféle módon lehet implementálni, a fő konvenció az, hogy a `PUSH` belerak a verembe egy elemet, a `POP` kivesz belőle egyet, a `TOP` pedig eléri a legfelső elemet.

Verem adatszerkezetet általában ott használnak, ahol fordított sorrendben kell feldolgozni a bemenő, előre nem ismert számú adatot.

10.4. Sor

A sor a veremhez hasonlóan nagyon egyszerű adatszerkezet, az a különbség, hogy míg a verem LIFO adatszerkezet, addig a sor FIFO, azaz First In First Out. Ez azt jelenti, hogy azt az elemet vesszük ki először az adatszerkezetből, tehát azt dolgozzuk fel először, amelyiket legrégebben raktunk bele az adatszerkezetbe. A két alapművelet sorokkal a

- `PUT` - elem belerakása a sorba
- `GET` - elem kivétele a sorból

Egészeket tartalmazó sor megvalósítása egyirányba láncolt listával:

```
struct sor{
   int adat;
   struct sor kov;
};

struct sor * uj_elem(int szam)
{
   /*
      létrehoz egy új listaelemet
   */
   struct sor *  uj=(struct sor *)malloc(sizeof(struct sor));
   uj->kov=NULL;
   uj->adat=szam;
   return uj;
}

void PUT(struct sor ** s, int szam)
{
   /* elem belerakása a sorba, valójában lista végére beszúrás */

   struct sor * tmp=*s;

   if(*s==NULL){ *s=uj_elem(szam); return;}
   while(tmp->kov!=NULL) tmp=tmp->kov;
   tmp->kov=uj_elem(szam);

}

int GET(struct sor ** s)
{
   /* elem kivétele a sorból, valójában lista elejéről törlés */

   int i;
   struct sor * tmp=*s;

   if(*s==NULL) return NULL;
   i=(*s)->adat;
   (*s)=(*s)->kov;
```

```
   free(tmp);

   return i;
}
```

Sor használatat akkor javasolt, ha az előre nem ismert számú bemenetet szeretnénk letárolni későbbi, az érkezésnek megfelelő sorrendben történő feldolgozásra.

10.5. Bináris kereső fák

A bináris keresőfák olyan dinamikus adatszerkezetek, amelyekben minden elemnek 2 rákövetkezője lehet. (Szemben a láncolt listákkal, ahol minden elemnek 1 rákövetkezője volt.) A bináris keresőfák használata a nagy adathalmazban történő keresést hivatott felgyorsítani. Az adatszerkezetben az elemeket rendezve tároljuk. A fa elemeinek terminológiája: amelyik elemnek nincs szülőeleme, az a fa gyökere, amelyeknek pedig nincsenek rákövetkezői, azok a fa levélelemei. Minden fának csak 1 gyökéreleme van, és tetszőlegesen sok leveléleme lehet.

Egész számokat tároló bináris keresőfa a következő típusú elemekből épül fel:

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

Az adatszerkezet kezeléséhez készítsük el először az `uj_elem` függvényt!

```
struct fa * uj_elem(int szam)
{
   /*
      új faelem dinamikus lefoglalása, és mezőinek beállítása
   */

   struct fa * uj=(struct fa *)malloc(sizeof(struct fa));
   uj->adat=szam;
   uj->bal=uj->jobb=NULL;

   return uj;
}
```

A bináris keresőfába történő beszúrást megvalósító függvény a következő:

```
struct fa * beszur(struct fa * gy, int szam)
{
   /*
      beszúrás bináris keresőfába

      gy: a fa gyökerének mutatója
      szam: a beszúrandó egész
   */

   if(gy==NULL)
```

```
      return uj_elem(szam);

   if(gy->adat>szam)
      gy->bal=beszur(gy->bal, szam);

   if(gy->adat<szam)
      gy->jobb=beszur(gy->jobb, szam);

   return gy;
}
```

A `beszur` függvény rekurzívan hívja magát a jobb- vagy baloldali részfára, attól függően, hogy a beszúrandó elem kisebb, vagy nagyobb, mint a fa aktuális, gyökérelemében tárolt szám.

Miután feltöltöttük a fát a `beszur` függvénnyel, az így létrejött adatszerkezetet valahogy fel kell dolgoznunk. Listáknál ezzel nem volt probléma, végigmentünk a listán valamelyik irányba, és biztos, hogy minden elemét feldolgoztuk a listának (például kiírtuk a képernyőre). Mivel a fa elemei nem egy irányban helyezkednek el, nem tudunk rajtuk a fent említett módon végigmenni. Valamilyen rekurzív bejárást kell alkalmaznunk, amivel a fa minden elemét elérjük, és pontosan egyszer. Három ilyen bejárást fogunk megnézni: `preorder, inorder, postorder` bejárásokat. A három eljárás között az a különbség, hogy milyen sorrendben dolgozzák fel az egyes fa csúcsokat. A következő eljárásokban a feldolgozás az aktuális elem kiírását jelenti majd:

```
void preorder(struct fa * gy)
{
   /*
      a fa elemeinek bejárása, és kiírása preorder módon
   */

   printf("%d\n", gy->adat);
   if(gy->bal) preorder(gy->bal);
   if(gy->jobb) preorder(gy->jobb);
}

void inorder(struct fa * gy)
{
   /*
      a fa elemeinek bejárása, és kiírása inorder módon
   */

   if(gy->bal) inorder(gy->bal);
   printf("%d\n", gy->adat);
   if(gy->jobb) inorder(gy->jobb);
}

void postorder(struct fa * gy)
{
   /*
      a fa elemeinek bejárása, és kiírása postorder módon
   */

   if(gy->bal) postorder(gy->bal);
   if(gy->jobb) postorder(gy->jobb);
   printf("%d\n", gy->adat);
}
```

11. Fájlkezelés

A processzor csak a memóriában lévő adatokkal tud dolgozni, ezért a fájlokat sem közvetlenül a merevlemezen, hanem a memóriában kezeljük. A fájlokat a memóriában reprezentáló adatszerkezet a `FILE` adatszerkezet. Mivel C-ben érték szerinti paraméterátadás van, ha egy függvény egy memóriában ábrázolt fájlt kapna paraméterként le kellene másolnia az egészet még egyszer, ami idő és memória pazarlás, ezért mindig `FILE*` mutatókkal fogunk dolgozni. Ahhoz, hogy egy fájllal dolgozhassunk, azt meg kell nyitni, majd ha befejeztük a munkát, be kell zárni. Fájl megnyitására a `fopen("hello.txt","r");` függvény szolgál, aminek első paramétere a megnyitni kívánt fájl neve, második paramétere pedig, hogy milyen céllal akarjuk megnyitni a fájlt. Ez lehet `"r", "w", "a"` amelyek rendre olvasás, írás és hozzáfűzés műveleteket jelentenek. Fájl lezárására az `fclose(f)` függvény szolgál, amely egyetlen paramétere egy `FILE` mutató.

```
/*fájlmutató deklarálása és fájl megnyitása: */

FILE * f = fopen("hello.txt", "r");

/*ITT szerepelhetnek végrehajtandó műveletek*/

/*fájl lezárása:*/
fclose(f);
```

A fájlokból történő olvasásra és írásra hasonló függvényeket használhatunk, mint a `standard input` és `output` kezelésére, csak az első paraméterük általában egy `FILE` mutató.

Fájlból olvasásra a következő függvények használhatók:

- int fscanf(FILE *, char *, ...);
- char * fgets(char * , int, FILE *);
- int fgetc(FILE *);

Fájlba írásra pedig a következők:

- int fprintf(FILE *, char *, ...);
- char * fputs(char *, FILE *);
- int fputc(int, FILE *);

A függények leírása a A C programozási nyelv című könyv végén megtalálható.
Példaként a következő program parancssori argumentumként kap két fájlnevet, az elsőből maximum 100 karakter hosszú sorokat olvas, letárolja őket egy veremben, és a második paraméterként kapott fájlba fordított sorrendben írja ki a sorokat.

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

struct verem{ char s[101];
              struct verem * kov;
};

struct verem * uj_elem(char * str)
{
   /*
      létrehoz egy új listaelemet
   */
   struct verem *  uj=(struct verem *)malloc(sizeof(struct verem));
   uj->kov=NULL;
   strcpy(uj->s, str);
   return uj;
}

void PUSH(struct verem ** v, char * str)
{
   /*PUSH művelet, valójában lista elejére beszúrás*/

   struct verem * uj=uj_elem(str);
   uj->kov=*v;

   *v=uj;
}

char * TOP(struct verem * v)
{
   /*Az első elem elérése*/

   return v->s;
}

void POP(struct verem ** v)
{
   /*POP művelet, valójában lista elejéről törlés*/

   struct verem * tmp=*v;

   if(*v==NULL) return;
   *v=(*v)->kov;
   free(tmp);
}

int main(int argc, char ** argv)
{
   /*fájlmutatók deklarálása, és fájlok megnyitása*/
   FILE * in=fopen(argv[1], "r");
   FILE * out=fopen(argv[2], "w");

   char t[101];

   struct verem * v = NULL;

   /*sorok beolvasása a verembe*/
   while(fgets(t, 100, in))
      PUSH(&v,t);

   /*sorok kiírása a veremből*/
```

```
   while(v){
      fputs(TOP(v), out);
      POP(&v);
   }

   fclose(in);
   fclose(out);

   return 0;
}
```

12. Fakezelő algoritmusok

Pár fejezettel korábban megnéztünk néhány bináris fákhoz kapcsolódó függvényt, eljárást, most további hasznos függvényekről lesz szó.

Adott egy bináris keresőfa, melynek elemeiben egész számokat tárolunk, tehát a struktúra, amelyből felépítjük a fát, a következő:

```
struct fa
{
   int szam;
   struct fa * bal, * jobb;
};
```

A fában tárolt elemek összege:

```
int osszeg(struct fa * gy)
{
   if(!gy) return 0;
   return osszeg(gy->bal)+osszeg(gy->jobb)+gy->szam;
}
```

Rekurzívan hívjuk meg a függvényt mind a baloldali, mind a jobboldali részfájára. A rekurzió akkor áll meg, amikor a levélelemekből azok jobb és bal oldali részfáira, vagyis `NULL` értékekre hívjuk meg az `osszeg` függvényt. Most nézzük meg, hogyan számolhatjuk meg a fa elemeinek a számát!

```
int elemszam(struct fa * gy)
{
   if(!gy) return 0;
   return elemszam(gy->bal)+elemszam(gy->jobb)+1;
}
```

A függvény felépítése megegyezik az előzővel. Míg az előző esetben mindig a csúcsban tárolt értéket adjuk hozzá a részfákban tárolt értékek összegéhez, addig ebben az esetben 1-et adunk hozzá, ami az adott csúcsot reprezentálja.

```
int max(struct fa * gy)
{
   int tmp, m=gy->szam;

   if(gy->bal){
      tmp=max(gy->bal);
      if(tmp>m) m=tmp;
   }
```

```
   if(gy->jobb){
      tmp=max(gy->jobb);
      if(tmp>m) m=tmp;
   }
   return m;
}
```

A fában tárolt páratlan számok számát visszaadó függvény:

```
int paratlan(struct fa * gy)
{
   if(!gy) return 0;
   return paratlan(gy->bal)+paratlan(gy->jobb)+((gy->szam%2)==1)?1:0;
}
```

A fában tárolt Fibonacci-számok számát visszaadó függvény, felhasználva az első házifeladatsor megoldását. Feltehetjük, hogy van egy függvény, amely visszatérési értéke 1, ha a paraméterként kapott szám Fibonacci-szám, és 0, ha nem az.

```
int fibonacci(struct fa * gy)
{
   if(!gy) return 0;
   return fibonacci(gy->bal)+fibonacci(gy->jobb)+(fibonacci(gy->szam)==1)?1:0;
}
```

Szám keresése bináris fában (visszatérési értéke 1, ha benne van, 0 ha nincs benne)

```
int keres(struct fa * gy, int szam)
{
   if(gy==NULL) return 0;
   if(gy->szam==szam) return 1;
   return keres(gy->bal,szam) || keres(gy->jobb,szam);
}
```

Szám keresése bináris KERESŐfában (visszatérési értéke 1, ha benne van, 0 ha nincs benne). Ebben az esetben az a különbség az előzőhöz képest, hogy nem kell rekurzív függvénnyel bejárnunk az egész fát, mivel kereséfáról van szó, amely rendezett, pontosan tudjuk, hogy hol keressük a számot. A bináris keresőfák a keresőfák részhalmazai, tehát ha egy algoritmus működik bináris fákra, akkor az alkalmazható bináris keresőfákra is.

```
int keres(struct fa *gy, int szam)
{
   while(gy!=NULL){
      if(gy->szam==szam)
         return 1;
      else if(gy->szam<szam)
         gy=gy->jobb;
      else if(gy->szam>szam)
         gy=gy->bal;
    }
    return 0;
}
```

Adott szám keresése bináris keresőfában, visszatérési értéke a tartalmazó elem mutatója:

```
struct fa * keres(struct fa *gy, int szam)
{
   while(gy!=NULL){
      if(gy->szam==szam)
         return gy;
      else if(gy->szam<szam)
         gy=gy->jobb;
      else if(gy->szam>szam)
         gy=gy->bal;
   }
   return NULL;
}
```

13. Tippek

Ha úgy gondoljuk, készen vagyunk a függvénnyel/programmal, nézzük át, hogy:

- Minden változó kapott-e értéket felhasználása előtt?
- Minden ciklus megáll-e valamikor?
- Minden függvénynek meghatározzuk-e a visszatérési értékét?
- Ha sztringekkel dolgoztunk karakterenként, nem felejtettük-e le a végéről a `'\0'` -t?
- Eljárást kellett-e írni, vagy függvényt, és azt írtunk-e?
- Annak az eljárásnak nincs értelme, ami nem ír ki semmit, nem változtat meg semmit a hívási környezetben, csak kiszámol valamit egy változójába!
- Minden kapcsoszárójelet bezártunk-e?