Adatszerkezetek és algoritmusok

Gyakorlati segédlet gazdasági informatikusok részére

Készítette: Szeghalmy Szilvia
Módosítva: 2017.05.11.

Tartalomjegyzék

1. BEVEZETÉS 3

2. ADATSZERKEZETEK ÉS MŰVELETEIK 4

2.1. TÖMB, MÁTRIX 4

2.1.1. Keresés 5

2.1.2. Rendezés 6

2.1.3. Néhány alapvető algoritmus 8

2.2. HALMAZ 10

2.3. MULTIHALMAZ 12

2.4. RITKAMÁTRIX 13

2.5. TÁBLÁZATOK 15

2.5.1. Soros táblázat 15

2.5.2. Önátrendező táblázat 17

2.5.3. Rendezett táblázat 17

2.5.4. Kulcstranszformációs táblázatok 19

2.6. SOR 26

2.7. VEREM 28

2.8. EGYIRÁNYBAN LÁNCOLT LISTA 29

2.9. KÉTIRÁNYBAN LÁNCOLT LISTA 33

2.10. BINÁRIS FA 36

2.11. BINÁRIS KERESŐ FA 38

2.12. ÁLTALÁNOS FA 41

2.13. GRÁF 43

3. FELADATOK ÉS MEGOLDÁSAIK 47

3.1. TÉMAKÖR: TÖMB 47

3.2. TÉMAKÖR: HALMAZ, MULTIHALMAZ 48

3.3. TÉMAKÖR: RITKAMÁTRIX 49

3.4. TÉMAKÖR: EGYIRÁNYBAN LÁNCOLT LISTA 50

3.5. TÉMAKÖR: ÁLTALÁNOS FA 53

4. GYAKORLÓ FELADATOK MEGOLDÁSOK NÉLKÜL 57

4.1. BEMELEGÍTŐ GYAKORLATOK (C ALAPOK) 57

4.2. RÖVID FÜGGVÉNYEK 57

4.3. TÖMBÖS, MÁTRIXOS FELADATOK 57

4.4. EGYIRÁNYBAN LÁNCOLT LISTÁS FELADATOK: 59

4.5. KÉTIRÁNYBAN LÁNCOLT LISTÁS FELADATOK 59

1. Bevezetés

Jelen anyag célja az otthoni gyakorlás elősegítése. A jegyzet nem lektorált, tehát hibákat tartalmazhat, és minden bizonnyal tartalmaz is. Használata csak kellő körültekintéssel javasolt! A felfedezett hibákat a szeghalmy.szilvia@inf.unideb.hu címre lehet jelezni.

A jegyzet első része a C nyelv bemutatására szolgál, olyan mértékben, amennyire azt a gyakorlati anyag megköveteli. A jegyzet második részében az órán is tanult adatszerkezetekről szól. A további részekben az első két részhez kapcsolódó gyakorló feladatok találhatóak.

A jegyzetben használt jelölések

Szintaktikai leírásban szereplő jelölések:

{a|b}	alternatíva (vagy *a* vagy *b* írandó a helyére)
[a]	a szögletes zárójelben lévő rész opcionális (*a*-t nem kötelező kiírni)
[a]...	a zárójelben lévő rész, *a* tetszőlegesen sokszor ismételhető

A programozási eszközök ismertetésénél a felső sorban az általános eset szerepel, alatta pedig egy konkrét példa.

```
típus változónév;
int x;
```

Kódrészletekben előforduló jelölések:

...	tetszőleges kódrészletetet jelöl a forráskódban
…	
// leírás	a három sor helyére a leírásnak megfelelő kódot kell írni
…	

2. Adatszerkezetek és műveleteik

2.1. Tömb, mátrix

Létrehozás

Meg kell adni az elemek típusát, a tömb nevét, és minden dimenzióhoz az elemszámot. Egy N dimenziós tömb létrehozása általánosan, ahol az elemszám helyén konstans és nevesített konstans állhat. (A félkövér zárójelben lévő rész elhagyható, a többi [] itt a szintaktika része!)

```
típus név[elemszám1][elemszám2]…[elemszámN][=kezdőérték]
```

5 elemű, t nevű, valós számokból álló 1D-s tömb létrehozása:

```
float t[5];
```

2 elemű *z* nevű egészekből álló 1D-s tömb 2, 5, 7 kezdőértékkel:

```
int z[3]={2, 5, 7};
int z[]={2, 5, 7}; //az elemszámot a fordító "beírja"
```

2x3-es *m* nevű egészekből álló 2D-s tömb. (első sora 1, 2, 0, második sora: 2, 8, 9):

```
int m[2][3]={{1,2,0},{2,8,9}};
```

10x15x30-as 3 dimenziós egészekből álló tömb létrehozása:

```
int tomb3D[10][15][30];
```

*Megj.: C-ben a több dimenziós tömb tömbök tömbjeként értelmezett, de az elemek számára folytons terület kerül lefoglalásra a memóriában. A * operátorral (n-1) dimenziós tömbként kezelve indexelhetjük. A leképezés sorfolytonsan történik. Például az m[1][2], és (*m)[1*3 + 2] azonos (az m mátrix 3 sorból és 2 oszlopból áll, ezért az 1-es indexű sor 2-es eleme az 6. (ötös indexű) eleme).*

Elérés

A tömb elemeinek elérése dimenziónként egy indexszel történik. Az index minden indextartomány esetén [0; elemszám-1] egész értékeket tartalmazó, zárt intervallumba esik. A fenti *t* tömb első, második, harmadik, negyedik, és ötödik elemének elérése rendre:

```
t[0], t[1], t[2], t[3], t[4]
```

A fenti z tömb 5-ös értékű elemének elérése:

```
z[1]
```

Az *m* mátrix első sorában lévő elemeinek elérése:

```
m[0][0], m[0][1], m[0][2]
```

Az *m* mátrix második sorában lévő elemeinek elérése:

```
m[1][0], m[1][1], m[1][2]
```

Bejárás

A tömb minden elemét fel kell dolgozni. A bejárás jellemzően *for* ciklussal történik, ahol a

ciklusváltozó az aktuális elem indexeként szerepel. A fenti *t* tömb bejárása:

```
int i;
for(i = 0; i<5; ++i)
   t[i]; //az elem feldolgozása feladatfüggő. Most emmi sem történik.
```

A fenti *m* mátrix bejárása sorfolytonos módon (ez azt jelenti, hogy az egy sorban lévő elemek egymás után kerülnek feldolgozásra):

```
int i, j;
for(i = 0; i<2; ++i)
   for(j = 0; j<3; ++j) //az oszlop index változik gyorsabban
      m[i][j];
```

A fenti *m* mátrix bejárása oszlopfolytonos módon (ez azt jelenti, hogy az egy oszlopban lévő elemek egymás után kerülnek feldolgozásra):

```
int i, j;
for(j = 0; j<3; ++j)
   for(i = 0; i<2; ++i) //a sor indexe változik gyorsabban
      m[i][j];
```

2.1.1. Keresés

Teljes keresés

Legyen adva egy *N* elemű *t* tömb és legyen *K* a keresett elem. A függvény a keresett elem indexével tér vissza, ha az szerepel a tömbben, különben -1-el.

```
int keres(int t[],int N, int K){
   int i;
   for( i = 0; i<N; ++i)
      if(t[i] == K) return i;
   return -1;
}
```

Lineáris keresés

Legyen adva egy *N* elemű *t* **rendezett** tömb és legyen *K* a keresett elem. A függvény a keresett elem indexével tér vissza, ha az szerepel a tömbben, különben -1-el.
Tf.: növekvő rendezettség áll fenn.

```
int keres(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;
}
```

Megj.: Az implementációt hatékonyabbá tehetjük, ha a cikluson belüli ellenőrzést kivesszük, hiszen a ciklusfeltétel miatt a keresés le fog állni, amint megtaláltuk K-t.

```
int keres(int t[], int N, int K){
   int i;
   for( i = 0; i<N && t[i]<K; ++i)
      ; //semmit sem csinálunk a cikluson belül
   return i<N && t[i] == K ? i: -1; //A feltétel azt nézi megtaláltuk-e K-t.
}
```

Bináris keresés

Legyen adva egy *N* elemű *t* **rendezett** tömb és legyen *K* a keresett elem. A függvény a keresett elem indexével tér vissza, ha az szerepel a tömbben, különben -1-el. A bináris keresés működéséhez szükséges, az elemek **közvetlen elérés**e.

Tf.: növekvő rendezettség áll fenn.

```
int keres(int t[],int N, int K){
   int ah = 0, fh = N-1, k = (ah+fh)/2;
   for(   ; ah<=fh; k = (ah+fh)/2){
      if(t[k] == K) return k;
      if(t[k]< K)
         ah = k+1
      else
         fh = k-1;
   return -1;
}
```

2.1.2. Rendezés

Közvetlen kiválasztásos rendezés

```
void kozvetlenKivalasztasos(int t[], int N){
   int i, j, tmp;
   for(i = 0; i<N-1; ++i)
      for(j = i+1; j<N; ++j)
         //novekvo sorrendben rendez. csokkeno: t[i]<t[j]
         if(t[i]>t[j]){
            tmp = t[i];
            t[i] = t[j];
            t[j] = tmp;
         }
}
```

Az algoritmus lépéseit a következő példán keresztül követhetjük nyomon, ahol az aláhúzott elemek a tömb *i.* elemét, a pirossal kiemelt számok a tömb *j.* elemét jelölik. Mindig ezt a két elemet hasonlítjuk egymáshoz, és ha szükséges, akkor ezeket az elemeket cseréljük fel.

Rendezendő tömb: t[] = {4, 8, 5, 3, 6};

i	j	t[0]	t[1]	t[2]	t[3]	t[4]	
0	1	4	8	5	3	6	
0	2	4	8	5	3	6	
0	3	4	8	5	3	6	csere!
0	4	3	8	5	4	6	
1	2	3	8	5	4	6	csere!
1	3	3	5	8	4	6	csere!
1	4	3	4	8	5	6	
2	3	3	4	8	5	6	csere!
2	3	3	4	5	8	6	
3	4	3	4	5	8	6	csere!
		3	4	5	6	8	

Gyorsrendezés

A rendezés során az elemeket két résztömbre bontjuk, majd a kapott tömböket rendezzük gyorsrendezéssel (rekurzió). A résztömbök képzésénél kiválasztunk egy elemet, és az elemtől kisebbeket a bal, a nagyobbakat a jobb oldali résztömbbe helyezzük. Az alábbi kódnál mindig a középsőelemet választjuk ki, de ez nem kötelező érvényű.

```
void gyors(int *t, int ah, int fh) {
   int i, j, k, tmp;
   if (fh <= ah) //üres, vagy 1 elem
      return;
   i = ah; j = fh;
   k = t[(ah + fh) / 2];
   while (i <= j){ [1]
      while (t[i] < k) i++;
      while (t[j] > k) j--;
      if (i <= j) {
         tmp = t[i];
         t[i] = t[j];
         t[j] = tmp;
         i++; j--;
      }
   }
   gyors(t, ah, j);
   gyors(t, i, fh);
}
```

Az algoritmus lépéseit a következő példán keresztül követhetjük nyomon, ahol az aláhúzott elemek a tömb *i.* elemét, a pirossal kiemelt számok a tömb *j.* elemét jelölik. A szürke háttér az aktuálisan vizsgált résztömböt jelöli.

gyors(t, 0, 6)	1 5 -2 4 3 8 2	
	1 5 -2 4 3 8 2	csere!
ah = 0, fh = 6, k = 4	1 2 -2 4 3 8 5	
	1 2 -2 4 3 8 5	csere!
	1 2 -2 3 4 8 5	
gyors(t, 0, 3)	1 2 -2 3 4 8 5	
	1 2 -2 3 4 8 5	csere!
ah = 0, fh = 3, k = 2	1 -2 2 3 4 8 5	
gyors(t, 0, 1)	1 -2 2 3 4 8 5	csere!
	-2 1 2 3 4 8 5	
gyors(t, 0, 0)	-2 1 2 3 4 8 5	fh <= ah teljesül
gyors(t, 1, 1)	-2 1 2 3 4 8 5	fh <= ah teljesül
gyors(t, 2, 3)	-2 1 2 3 4 8 5	
	-2 1 2 3 4 8 5	j, i egybeesik
ah = 2, fh = 3, k = 2	-2 1 2 3 4 8 5	j = 1 (ah alatt)
gyors(t, 2, 1)	üres tartomány	
gyors(t, 3, 3)	-2 1 2 3 4 8 5	fh <= ah teljesül
gyors(t, 4, 6)	-2 1 2 3 4 8 5	
	-2 1 2 3 4 8 5	csere!
ah = 4, fh = 6, k = 8	-2 1 2 3 4 5 8	

[1] i<j esetén a résztömbök átfedőek lehetnek, ez csak párhuzamos futtatás esetén okozhaz zavart.

gyors(t, 4, 5) ah = 4, fh = 5, k = 4	-2 1 2 3 4 5 8	
	-2 1 2 3 4 5 8	i, j egybeesik
	-2 1 2 3 4 5 8	j = 3 (ah alatt)
gyors(t, 4, 3)	üres tartomány	
gyors(t, 5, 5)	-2 1 2 3 4 5 8	fh <= ah teljesül
gyors(t, 6, 6)	-2 1 2 3 4 5 8	fh <= ah teljesül

2.1.3. Néhány alapvető algoritmus

Elemek összege:

```
int osszeg(int t[], int n){
   int i, sum = 0; //kezdőérték adás ne maradjon le!
   for(i = 0; i<n; ++i)
      sum += t[i];
   return sum;
}
```

Elemek átlaga:

```
double atlag(int t[], int n){
   int i, sum = 0;
   for(i = 0; i<n; ++i)
      sum += t[i];
   return sum/(double)n; //egészosztás:1/2 = 0, de 1/(double)2 = 0.5
}
```

Számlálás (felételtől függően)
A feltétel a feladattól függően helyettesítendő. Pl. ha a negatív értékek számát akarjuk megkapni, akkor a <feltétel> helyére a `t[i] < 0` kifejezés kerül.

```
int szamlalas(int t[], int n){
   int i, count = 0;
   for(i = 0; i<n; ++i)
      if( <feltetel>  )
         ++count;
   return count;
}
```

Elemek szorzata (n>=1)

```
double szorzat(int t[], int n){
   int i;
   double s = 1; //kezdőérték!
   for(i = 0; i<n; ++i)
      s *= t[i];
   return s;
}
```

Elemek maximuma (n>=1)

```
int maximum (int t[], int n){
   int i, max = t[0];
```

```
   for(i = 1; i<n; ++i)
   if( max < t[i])
      max = t[i];
   return max;
}
```

Elemek minimuma (n>=1)

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

Maximumhely keresése, több minimum esetén az elsőt adja vissza (n>=1)

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

Maximumhelyek megjelenítése (n>=1)

```
int maximumhely(int t[], int n){
   int i, m = maximum(t, n);
   for(i = 0; i<n; ++i)
      if(t[i] == m)
         printf("%d, ", i);
}
```

Az algoritmusok alapelve mátrixok esetén sem változik. A mátrix elemeinek bejárása az, ami elgér. Például:

Elemek összege:

```
int osszeg(int mat[N][M]){
   int i,j, sum = 0;
   for(i = 0; i<N; ++i)
      for(j = 0; j<M; ++j)
         sum += mat[i][j])
   return sum;
}
```

Mátrix adott sorának feldolgozása (a példában az elemek kiírása)

```
void egySor(int mat[N][M], int sorIndex){
   int j;
   for(j = 0; j<M; ++j)
         printf("%d ", mat[sorIndex][j]);
```

```
}
```

Mátrix adott oszlopának feldolgozása (a példában az elemek kiírása)

```
void egyOszlop(int mat[N][M], int oszlopIndex){
   int i;
   for(i = 0; i<N; ++i)
      printf("%d ", mat[i][oszlopIndex]);
}
```

2.2. Halmaz

A matematikai halmazfogalom megjelenítése adatszerkezet szintjén. Homogén és véges. Ábrázolása általában folytonosan történik a karakterisztikus függvény megadásával. Ehhez definiálni kell egy alaphalmazt, rögzített elemsorrenddel. A halmaz elemeit az alaphalmaz elemeinek számával egyező hosszúságú 0-1 sorozat adja meg. A sorozatban 0 jelzi, hogy az alaphalmaz azonos pozícióján lévő elem nincs a halmazban, 1 hogy benne van.

A továbbiakban az alaphalmaz az első N nemnegatív egész szám, növekvő sorrend szerint rögzítve. Ekkor a halmazban a [0, N-1] zárt intervallumbeli értékek szerepelhetnek. Az érthetőség kedvéjért a műveletek egészeket tartalmazó tömbbel kerülnek bemutatásra, bár 1 elem tárolására 1 bit is elegendő lenne. Ez a megoldás, azért kényelmes, mert minden halmazelem, a tárolásra használt tömb azonos indexével esik egybe, tehát ha a halmazban egy *x* elem szerepel, akkor a tömb *x.* eleme 1-es értékű lesz, különben 0.

Megj.: Amennyiben a halmaz lehetséges elemei nem felelnek meg ennek a követelménynek, akkor adjunk meg kölcsönösen egyértelmű megfeleltetést az alaphalmaz és a [0, N-1] intervallum között.

Létrehozás

Pl.: Legyen a tárolandó halmaz *{1, 4, 2, 5}*, és *N = 10*.

```
//A tárolandó halmazelemek értékével azonos indexen áll 1-es, különben
0
int halmaz[] = {0,1,1,0,1,1,0,0,0,0};
```

A halmazműveletek tárgyalása előtt lássunk két alapvető utasítást. Tf.: *x* lehetséges halmazelem. (Az alaphalmaz eleme.)
Egy halmaz *x* elemmel való bővítéséhez, az tömb *x.* elemét kell *1*-re állítani.

```
halmaz[x] = 1;
```

Az x elem törléséhez pedig a tömb x. elemét 0-ra kell állítani.

```
halmaz[x] = 0;
```

Eleme

Az alábbi függvény 1-et ad vissza, ha *x* eleme a halmaznak, különben 0-át. A függvényt érdemes felkészíteni olyan eshetőségre is, amikor a keresett elem nem esik bele az alaphalmazként rögzített

tartományába.

```
int eleme(int halmaz[N], int x){
   //azt is le kell ellenőrizni, hogy x beleesik-e a [0, N-1]-be!
   return x>=0 && x<N && halmaz[x];
}
```

Unió

Két halmaz uniója a karakterisztikus függvényük közti (logikai) *vagy* művelettel számítható.
Tf. hogy N>=M:

```
int unio(int halmaz[N], int halmaz2[M], int eredmeny[N]){
   int i = 0;
   for(i = 0; i<M; ++i)
      eredmeny[i] = halmaz[i] || halmaz2[i];
   for( ; i<N; ++i)
      eredmeny[i] = halmaz[i];
}
```

Metszet

Tf. hogy N>=M:
Két halmaz metszete a karakterisztikus függvényük közti (logikai) *és* művelettel számítható.

```
int metszet(int halmaz[N], int halmaz2[M], int eredmeny[N]){
   int i = 0;
   for(i = 0; i<M; ++i)
      eredmeny[i] = halmaz[i] && halmaz2[i];
   for( ; i<N; ++i)
      eredmeny[i] = 0;
}
```

Különbség

Tf. hogy N>=M. Halmaz-halmaz2 az alábbi függvénnyel írható le:

```
int kulonbseg(int halmaz[N], int halmaz2[M], int eredmeny[N]){
   int i = 0;
   for(i = 0; i<M; ++i)
      eredmeny[i] = halmaz[i] && !halmaz2[i];
   for( ; i<N; ++i)
      eredmeny[i] = halmaz[i];
}
```

Komplementer

A halmaz komplementere a halmaz karakterisztikus függvényének bitenkénti negálásával valósítható meg.

```
int komplementer(int halmaz[N], int eredmeny[N]){
   int i = 0;
   for(i = 0; i<N; ++i)
      eredmeny[i] = !halmaz[i];
}
```

2.3. Multihalmaz

Ábrázolása a halmazéhoz hasonló. A multihalmaz elemeit az alaphalmaz elemeinek számával egyező hosszúságú egész szám sorozat adja meg. Egy adott pozíción szereplő érték azt adja meg, hogy az elem hányszor szerepel a halmazban. A műveleteknek sajnálatos módon többféle definíciójuk is létezik.

Létrehozás

Pl.: Legyen a tárolandó multihalmaz {1, 1, 4, 2, 5, 5, 5, 5}, és N = 10.

```
int halmaz[] = {0,2,1,0,1,4,0,0,0,0};
```

A multihalmaz műveleteinek tárgyalása előtt lássunk két alapvető utasítást. Tf.: x lehetséges halmazelem. (Az alaphalmaz része.) Egy multihalmaz x elemmel való bővítéséhez, az tömb x. elemét kell 1-el növelni.

```
++halmaz[x];
```

Az x elem törléséhez pedig a tömb x. elemét 1-el csökkenteni, ha x eleme a halmaznak.

```
if(halmaz[x]>0)
   --halmaz[x];
```

Eleme

Az alábbi függvény 1-et ad vissza, ha x eleme a halmaznak, különben 0-át.

```
int eleme(int halmaz[N], int x){
   return x>=0 && x<N && halmaz[x];
}
```

Unió

Két halmaz uniója a karakterisztikus függvényük közti összeadás művelettel számítható.
Tf. hogy N>=M:

```
int unio(int halmaz[N], int halmaz2[M], int eredmeny[N]){
   int i = 0;
   for(i = 0; i<M; ++i)
      eredmeny[i] = halmaz[i] + halmaz2[i];
   for( ; i<N; ++i)
      eredmeny[i] = halmaz[i];
}
```

Megj.: Más definíció szerint halmaz[i] + halmaz2[i] helyett: max(halmaz[i], halmaz2[i]) szerepel.

Metszet

Tf. hogy N>=M:
Két halmaz metszete a karakterisztikus függvényük közti *min* művelettel számítható.

```
int metszet(int halmaz[N], int halmaz2[M], int eredmeny[N]){
   int i = 0;
   for(i = 0; i<M; ++i)
      eredmeny[i] = min(halmaz[i],halmaz2[i]);
   for( ; i<N; ++i)
```

```
        eredmeny[i] = 0;
}
```

Különbség

Tf. hogy N>=M. Halmaz-halmaz2 az alábbi függvénnyel írható le:

```
int kulonbseg(int halmaz[N], int halmaz2[M], int eredmeny[N]){
   int i = 0;
   for(i = 0; i<M; ++i){
   eredmeny[i] = halmaz[i]>halmaz2[i] ? halmaz[i] - halmaz2[i] : 0;
   for( ; i<N; ++i)
      eredmeny[i] = halmaz[i];
}
```

2.4. Ritkamátrix

Olyan mátrix melyben egy elem nagyságrendekkel többször szerepel, mint a többi. Ezt az elemet *gyakori elem*nek nevezzük. A gyakori elem általában a 0. A cél az elemek helytakarékosabb módon való tárolása.

Háromsoros reprezentáció (koordinátalista)

NxM-es mátrix háromsoros reprezentációja 3db K elemű vektorral történik (Sor, Oszlop, Érték), ahol K a nem gyakori elemek darabszáma. A mátrix (sor v. oszlopfolytonos) bejárása közben talált nem gyakori elemek sora, oszlopa, és értéke a Sor, Oszlop és Érték vektorok következő szabad helyére kerül bejegyzésre.

Legyen a kiinduló mátrix:

$$\begin{pmatrix} 0 & 0 & 2 & 1 & 0 \\ 2 & 0 & 1 & 0 & 5 \\ 1 & 0 & 4 & 0 & 0 \end{pmatrix}$$

Sor	0	0	1	1	1	2	2
Oszlop	2	3	0	2	4	0	2
Érték	2	1	2	1	5	1	4

Négysoros reprezentációnál az alapmátrix kiegészül egy mutató vektorral, mely (sorfolytonos esetben) az alapmátrixban azonos oszlopban lévő elemeket láncolja össze, ezzel elősegítve az elemek oszlopfolytonos feldolgozását. A felépítéséhez nincs szükség az eredeti mátrixra, csak az *Oszlop* tömb értékei alapján dolgozunk. A *Mutató* tömb *i.* értéke az a sorszám lesz, ahol az *Oszlop* tömbben az *i.* helytől jobbra először találunk az oszlop *i.* elemével azonos értéket. Ha nem találunk azonosat, akkor -1 lesz az érték.

Sorszám	0.	1.	2.	3.	4.	5.	6.
Sor	0	0	1	1	1	2	2
Oszlop	2	3	0	2	4	0	2
Érték	2	1	2	1	5	1	4
Mutató	3	-1	5	6	-1	-1	-1

Felépítése (sorfolytonos)

A *gy* a gyakori elem. *K* a nem gyakori elemek száma.

```
void felepit(int t[N][M], int sor[K], int oszlop[K], int ertek[K], int
```

```
mutato[K], int gy){
   //3 soros
   int i, j, p = 0;
   for(i = 0; i<N; ++i)
      for(j=0; j<M; ++j)
      if(t[i][j] != gy){
         sor[p] = i;
         oszlop[p] = j;
         ertek[p] = t[i][j];
         ++p;
      }

//4. sor (mutato)
for(i = 0; i<p; ++i){
   //i-től balra áll-e i.-el egyező érték az oszlop tömbben?
   for(j = i+1; j<p; ++j)
      if(oszlop[i] == oszlop[j])
         break;
   //ha nem, akkor a mutató legyen -1
   if(j == p)
      mutato[i] = -1;
   //ha igen, akkor mutato[i]-be a megtalált elem pozíciója kerül.
   else
      mutato[i] = j;
   }
}
```

Elem elérése

A *gy* a gyakori elem. Az alapmátrix s. sorának, és o. oszlopban lévő elem értékét adja vissza. (A példa feltételezi, hogy helyes s, és o értékkel hívják a függvényt.)

```
void eler(int sor[K], int oszlop[K], int ertek[K], int s, int o){
   for(int i=0; i<K; ++i)
      if(sor[i] == s && oszlop[i] == o)
         return ertek[i];
   return gy;
}
```

Sorfolytonos leképezés esetén sor, oszlopfolytonos esetén oszlop szerinti lineáris keresés is elegendő teljes keresés helyett: *i<K* helyén *i<K && t[i]<=s, vagy i<K && t[i]<=o*.

Mutató oszlop felhasználása

A mutató oszlop felhasználása során kikeressük az első az adott oszlop első elemét a 4 soros reprezentációban egy ciklus segítségével. Ha az i. helyen találtuk meg az elemet, akkor a következő azonos oszlopban lévő elem, a mutato[i]. helyen lesz, ezért az i index értékét mutato[i]-re állítjuk át.

Példa: Jelenítsük meg az eredeti mátrix o. oszlopában lévő nem gyakori elemeket!

```
void ir(int sor[K], int oszlop[K], int ertek[K], int mutato[K], int
o){
    int i;
    //megkeressuk az adott oszlop elso elemet
    for(i = 0; i<K; ++i){
```

```
      if(oszlop[i] == o){
         //feldolgozzuk az adott oszlopot a mutatokon vegiglepdelve
         for(  ; mutato[i] != -1; i = mutato[i])
            printf("%d ", ertek[i]); //ha van még elem az oszlopban
         printf("%d ", ertek[i]); //a legutolsó oszlopemelem kiírása
      }
    }
}
```

2.5. Táblázatok

A táblázatok asszociatív címzésű, homogén adatszerkezetek. A táblázat elemei eleve összetettek, mert az elemek a tárolni kívánt adat mellett egy kulcs részt is tartalmaznak, mely segítségével az adatelem azonosítható. A gyakorlaton eddig elsajátított ismereteink alapján csak folytonos ábrázolási módot tudunk megvalósítani, ezért csak azokat a táblázatokat nézzük meg részletesen, ahol ez (is) hatékony.

Az összetett elem leírására struktúrával történik, ami természetesen a konkrét alkalmazástól függ. Az adatszerkezet bemutatásánál egész típusú szám lesz az adat és a kulcs rész is. (A *typedef* rész a megadott típust ELEM-re nevezi át.)

```
typedef struct elem{
   int adat;
   int kulcs;
} ELEM;
```

2.5.1. Soros táblázat

A soros táblázatnál a kulcsok semmilyen hatással nincsenek az elem táblázatban elfoglalt helyére, ezt a műveleteknél is ki fogjuk használni. Akkor érdemes ilyen táblázatot választanunk az adataink tárolására, ha általában az összes vagy legalábbis sok elemet kell feldolgozni. A törlés és a bővítés is hatékonyan kivitelezhető, ezért akkor is jól jöhet ez a fajta tárolás, ha gyakran kell ilyen módosító műveleteket végeznünk.

Létrehozás

Folytonos reprezentáció esetén a létrehozás során egy ELEM típusú elemekből álló tömböt hozunk létre. Maximálisan annyi elemet tárolhatunk majd a táblázatban, amilyen méretű a tömb. A táblázatban szereplő (érvényes) elemek számát is tároljuk, kezdetben ez nulla.

```
#define MERET 50000 //Maximalisan tarolhato elemszam.

ELEM tabla[MERET];
int elemszam = 0;
```

Bejárás

A táblázatban az elemek folytonosan lesznek tárolva (nem lesznek üres helyek két érvényes elem között), ezért a bejárást egy 1D-s tömb bejárását jelenti, azzal a különbséggel, hogy a tömb vége helyett csak az utolsó érvényes elemig megyünk el.

```
void bejar(){
   int i;
   for( i = 0; i<elemszam; ++i)
```

```
        feldolgoz(tabla[i]);
    }
```

Keresés

Keresés kulcs alapján: A keresést teljes kereséssel valósítjuk meg, hiszen a kulcs értéke nem utal az elem helyére, ne feledjük, most is csak az utolsó érvényes elemig tart a keresés. Ha a keresett kulcs nincs benne a táblázatban -1-et ad vissza a függvény.

```
    int holVan(int kulcs){
       int i;
       for( i = 0; i<elemszam; ++i){
          if( tabla[i].kulcs == kulcs)
             return i;
       return -1;
    }
```

Beszúrás

A folytonos feltöltés miatt az új elemet közvetlenül az utolsó táblázatban lévő elem után szúrjuk be, ami C-ben annyit jelent, hogy ha *x* elemünk van, akkor a tömb *x.* indexére pakoljuk az elemet. Persze előtte ellenőriznünk kell, hogy van-e hely és azt is, hogy van-e már olyan kulccsal elem.

```
void beszur(int kulcs, int adat){
   if( elemszam < MERET ){
       if(holVan(kulcs) == -1 ){
           tabla[ elemszam ].kulcs = kulcs;
           tabla[ elemszam ].adat = adat;
           ++elemszam;
       }
       else
           printf("Foglalt a kulcs.");
   }
   else
      printf("Nincs hely az elem beszurasara.\n");
}
```

Törlés

Ha a sorrend megörzésével kellene kitörölni az elemet, a törlendő elem után lévő összes elemet előre kellene mozgatnunk. Szerencsénkre azonban a soros táblázatnál a kulcsoknak nem kell sorrendben lenniük és nincsenek összefüggésben a kulcs értéke az elem táblázatban elfoglalt helyével, ezért egyszerűbb megoldást is használhatunk. Az utolsó elemmel felülírjuk a törlendő elemet (a kulcs és adat részt is), és az utolsó elmet vesszük ki a táblázatból, ami a gyakorlatban azt jelenti, hogy az elemszám csökken eggyel.

```
    void torol(int kulcs){
       int idx = holVan(kulcs);
       if( idx != -1 ){
            /*az utolso elemet "toroljuk", de nem fizikailag*/
            --elemszam;
            /*az utolso elemmel felulirjuk a torlendot*/
            tabla[idx] = tabla[elemszam];
       }
       else{
```

```
        printf("Az elem nincs a tablazatban.");
    }
}
```

Csere

Az csere az adat rész felülírását jelenti.

```
void cserel(int kulcs, int ujadat){
    int idx = holVan(kulcs);
    if( idx != -1 )
         tabla[idx].adat = ujadat;
    else
        printf("Az elem nincs a tablazatban.");
}
```

2.5.2. Önátrendező táblázat

Az önátrendező táblázatnál az a cél, hogy a gyakran használt elemek a táblázat az adatszerkezet elejére kerüljenek. A módszerrel akkor nyerünk, ha általában nem az összes adatelemet kell feldolgozoni, hanem csak néhányat, és ha az elemek feldoglozásának gyakorisága eltérő. Gondoljunk csak arra, hogy a lakásban sem ássuk el a szekrény legmélyére azokat a dolgokat, amire minden nap szükségünk van, hanem olyan helyre tesszük, ahol szem előtt van, amikor kell, akkor gyorsan megtalálhatjuk.

Az önátrendező táblázatot szétszórt reprezentációval érdemes tárolni, ezért ennek ismertetésétől eltekintünk.

2.5.3. Rendezett táblázat

Ahogy arra a neve is utal, a rendezett táblázat elemei rendezett sorrendben jelennek meg a táblázatban. Méghozzá kulcs szerint rendezve. Az elemek kulcs szerinti keresése a táblázat szétszórt ábrázolás esetén lineáris kereséssel, folytonos ábrázolása esetén bináris kereséssel történik. A rendezettség több elem keresése során is jól kihasználható, különösen akkor, ha a kulcsok egy összefüggő tartományát kell feldolgozni.
Ha az elemek száma gyakran változik, akkor a szétszórt ábrázolás az előnyösebb, hiszen a módosítás folytonos tárolás esetén jelentős mértékű adatmozgatással járhat együtt. Amennyiben a feldolgozás a gyakoribb művelet, és az elemek száma viszonylag stabil érdemes lehet a bináris keresés előnyeit kihasználni. Az alábbiakban a folytonos reprezentációt tekintjük át.

Létrehozás

A soros táblázatnál ismertetett módon.

Bejárás

A soros táblázatnál ismertetett módon.

Keresés

Keresés kulcs alapján: A keresést bináris kereséssel valósítjuk meg.

```
int holVan(int kulcs){
    int ah = 0, fh = elemszam-1, k;
    for(  ; ah<=fh; ){
       k = (ah+fh)/2;
       if(tabla[k].kulcs == kulcs)
          return k;
       if(tabla[k].kulcs < kulcs)
          ah = k+1;
       else
          fh = k-1;
    }
    return -1;
}
```

Beszúrás

Ahhoz, hogy a beszúrás során az elemek megőrizzék a sorrendjüket, tudnunk kell, hogy az új elemnek hol kell elhelyezkednie a tömbben. (Egy speciális lineáris kereső: ha megtaláltuk az elemet, akkor adunk vissza -1-es értéket, és ha nem találtuk, akkor azzal térünk vissza, hol jöttünk erre rá, ugyanis az adott elemet oda kell majd beszúrni.)

```
int holKelleneLennie(int kulcs){
    int i;
    //lehetne hatékonyabban (ld. lin ker), de így könnyen
áttekinthető:
    for(i = 0 ; i < elemszam ; ++i){
        if( tabla[i].kulcs == kulcs)
          return -1; //foglalt kulcs
        if( tabla[i].kulcs > kulcs)
           return i;
    }
    return elemszam;
}
```

Felhasználva a fenti függvényeket, a beszúrás kódja:

```
void beszur(int kulcs, int adat){
   if(elemszam < MERET){
        int idx = holKelleneLennie(kulcs);
        /*ha még nem foglalt a kulcs (idx < MERET, hiszen van hely)*/
       if( idx >= 0 ){
           /*helyet csinálunk az elemnek az elemek elmozgatásával*/
           int i;
           for(i = elemszam; i > idx; --i){
               tabla[ i ] = tabla[i-1];
           }
           /*beszúrjuk az új elemet*/
           tabla[ idx ].kulcs = kulcs;
           tabla[ idx ].adat = adat;
           ++elemszam;
       }
```

```
        else
            printf("Foglalt a kulcs.");
    }
    else
       printf("Nincs hely az elem beszurasara.\n");
}
```

Törlés

Sajnos most nem élhetünk a soros táblázatoknál megismert trükkel, hiszen az elronthatná a kulcsok rendezettségét. Az elem törlését ezúttal a mögötte lévő elemek előremozdításával fogjuk elérni.

```
void torol(int kulcs){
   int idx = holVan(kulcs);
   if( idx != -1 ){
       /*a törlendő mögötti elemeket előremozgatjuk*/
       int i;
       for(i = idx; i < elemszam-1; ++i){
            tabla[ i ] = tabla[i+1];
       }
       --elemszam;
   }
   else
       printf("Az elem nincs a tablazatban.");
}
```

A rendezett táblázatot szétszórt reprezentációval való tárolásának ismertetésétől is eltekintünk. (A listák megismerése után önállóan is képesek lesztek elkészíteni.)

Csere

A csere az adat rész felülírását jelenti.

```
void cserel(int kulcs, int ujadat){
   int idx = holVan(kulcs);
   if( idx != -1 )
        tabla[idx].adat = ujadat;
   else
       printf("Az elem nincs a tablazatban.");
}
```

2.5.4. Kulcstranszformációs táblázatok

A kulcstranszformációs táblázatok céja, hogy az egyes elemeket a kulcs alapján gyorsan (közvetlen, kvázi közvetlen) módon lehessen elérni.

2.5.4.1. Közvetlen szervezésű struktúra (direkt táblázat)

A közvetlen szervezésű struktúrát azoknál a feladatoknál alkalmazhatjuk, ahol a kulcsok halmaza kölcsönösen egyértelmű módon leképezhető[2] a tárolóhelyek halmazára (index tartományra). Az

[2] Kölcsönösen egyértelmű **leképezés** (injekció), ami lehet **ráképezés** (szürjekció) is, de nem feltétlenül az. A

ábrázolás folytonos módon történik. A leképezésre szolgáló függvényt hash függvénynek nevezzük.

Létrehozás

A táblázat létrehozáshoz elengedhetetlen, hogy ismerjük a lehetséges kulcsok halmazát/tartományát. Kialakítjuk a megfelelő méretű tárhelyet, valamint jelezzük, hogy az összes hely üres. Utóbbira azért van szükség, mert az elemek a rendelkezésre álló tárhelyen belül nem feltétlenül folytonosan helyezkednek el. Ezen túl rögzítenünk kell a használni kívánt hash függvényt is.

A táblázat mérete és a hash függvény is a rendelkezésre álló kulcsoktól függ. Tekintsünk most egy olyan példát, ahol az 1900-as években felépülő iskolák számát tároljuk el. Az adatelemek kulcsa az 1900-as évekből való évszám, az adat rész az iskolák száma. A kulcsok tartománya tehát az [1900, 1999], az egész számok körében.

```
#define MERET 1000     /*A kulcsok halmazának elemszáma*/
#define URES -1        /*Nincs a táblázat adott helyén elem*/
#define ERVENYTELEN -2 /*Érvénytelen index.*/

typedef struct elem{
  int kulcs;
  int adat;
} ELEM;

/*Táblázat létrehozása*/
ELEM tabla[MERET];
```

Az alábbi függvény (meghívása!) a táblázat minden elemét üresre állítja.

```
void inicializalas(){
    int i;
    for(i = 0; i < MERET; ++i)
        tabla[i].kulcs = URES;
}
```

A fenti probléma esetén egy egyszerű kivonással elintézhető, hogy az index a [0, MERET-1] tartományra essen. A függvény ERVENYTELEN-t ad vissza, ha a kulcs kívül esett a lehetséges tartományon (hibás kulcs, pl.: elgépelte a felh.).

```
int hash(int kulcs){
    int idx = kulcs - 1900;
    return 0 <= idx && idx < MERET ? idx : ERVENYTELEN;
}
```

Beszúrás

A beszúrás során az új elem kulcsértékét leképezzük a hash függvénnyel. Ha az érték érvényes, arra

---

hash függvény minden kulcshoz (értelmezési tartomány) különböző indexszet (értékkészlet) rendel, de lehet olyan index, melyre nem képeztünk le kulcsot. Egyszerűbben fogalmazva: a tárhelyek száma meghaladhatja a kulcsok számát.

a helyre pakoljuk be az adott elemet, ha még nincs ott másik elem. (Ha van, akkor azonos adattal kellene ott lennie, különben valamilyen probléma van: nem lehet két azonos kulcs, ezért vagy ugyanazt akartuk még egyszer beszúrni, vagy hibás az adat, legrosszabb esetben hibásan adtuk meg a hash függvényt.)

```
void beszur(int kulcs, int adat){
    int idx = hash(kulcs);
    if( idx == ERVENYTELEN )
        printf("Érvénytelen kulcs.\n");
    else if ( tabla[idx].kulcs != URES )
        printf("Foglalt kulcs (van ott elem).\n");
    else{
        tabla[idx].kulcs = kulcs;
        tabla[idx].adat = adat;
    }
}
```

Törlés

Az adott kulccsal rendelkező elem törlésénél a hash függvény segítségével határozzuk meg az elem táblázatban elfoglalt helyét. Amennyiben az adott helyen van elem, a kulcs részt felülírjuk az üres hely jelzésére szolgáló értékkel.

```
void torol(int kulcs){
    int idx = hash(kulcs);
    if( idx == ERVENYTELEN )
        printf("Érvénytelen kulcs.\n");
    else if ( tabla[idx].kulcs == URES )
        printf("Nincs ilyen kulccsal elem.\n");
    else
        tabla[idx].kulcs = URES;

}
```

Csere

A csere mindig az elem adat részének felülírását jelenti. A kulcs módosítása nem engedélyezett. A megvalósítás során az elem kulcsa és a hash függvény alapján meghatározzuk az elem helyét. Ha ott ténylegesen van elem, akkor az adatrészt felűlírjuk.

```
void csere(int kulcs, int ujadat){
    int idx = hash(kulcs);
    if( idx == ERVENYTELEN )
        printf("Érvénytelen kulcs.\n");
    else if ( tabla[idx].kulcs == URES )
        printf("Nincs ilyen kulccsal elem.\n");
    else
        tabla[idx].adat = ujadat;
}
```

Bejárás

Legegyszerűbb a táblázat soros végigjárásával, miközben minden helyen ellenőrizzük, hogy a táblázat adott pozícióján van-e elem.

```
void bejar(){
    int i;
    for( i = 0; i<MERET; ++i)
        if( tabla[i].kulcs != URES)
           feldolgoz(tabla[i]);
    }
```

De bejárható a táblázat a kulcsértékeken végigmenve, a hash függvény alapján elérve az elemeket. Utóbbi természetesen a kulcsok halmazától függ, a fenti évszámos példánál:

```
void bejar(){
    int i, idx;
    for( i = 1990; i<2000; ++i)
        idx = hash(i);
        if( tabla[idx].kulcs != URES)
           feldolgoz(tabla[idx]);
}
```

Keresés

A kulcs alapján tényleges keresésre nincs szükség, a hash függvényt használva közvetlenül megkapjuk az elem táblázatbeli helyét. Az alábbi függvény az adott kulcsú elem táblázatbeli helyét adja vissza, illetve ha az nem szerepel a táblázatban, akkor -1-et.

```
int holVan(int kulcs){
   idx = hash(kulcs);
   if( idx != ERVENYTELEN && tabla[idx].kulcs == kulcs)
      return idx;
   return -1;
}
```

Az adatok keresése teljes kereséssel történik, de csak az érvényes elemek lesznek figyelembe véve. Az alábbi függvény a keresett adat első előfordulásának helyét adja vissza, vagy ha nincs az elem a táblázatban, akkor -1-et.

```
int adatKereso(int adat){
    int i;
    for( i = 0; i<MERET; ++i)
        if( tabla[i].kulcs != URES && tabla[i].adat == adat)
           return i;
    return -1;
 }
```

Újraszervezés

Amennyiben megváltozik a kulcsok tartománya (túlnő a lefoglalt tárhelyen a szám, vagy annyival lecsökken, hogy pazarló lenne fenntartani a tárolást) újraszervezhetjük a táblázatot.

A részletek ismertetésétől eltekintünk, mert az újraszervezésshez szükséges programozási eszközök ismerete még nem elvárás a gyakorlatnak ezen a pontján.

2.5.4.2. Véletlen szervezésű struktúra (random táblázat, hasító táblázat)

A közvetlen szervezésű struktúánál láthattuk, hogy kölcsönösen egyértelmű leképzelésre van szükség a lehetséges kulcsok és a tároló helyek halmaza között. A gyakorlati életben viszont sokszor előfordul, hogy a kulcsok lehetséges tartománya annyival meghaladja a táblázatban ténylegesen tárolásra kerülő elemek számát, hogy nincs értelme, vagy nem is lehet annyi helyet fenntartani. A közvetlen szervezésű struktúránál a táblázat mérete az elemek várható számának alapján kerül meghatározásra (felülről becsülve az elemszámot). Ha kevesebb a hely, mint a lehetséges kulcsok száma, már nem lehet kölcsönösen egyértelmű leképezést létrehozni, tehát elő fog fordulni olyan eset, amikor két különböző kulcshoz azonos tárhelyet rendel a hash függvény, amit *kulcsütközés*nek nevezünk.

Az ütköző elemek (szinonímák) kezelésére és a címképzésre szolgáló különböző módszerek az előadás anyagát képezik. Az alábbiakban a műveleteket folytonos ábrázolás, osztó módszeres címképzés és nyílt címzéses szinonímakezelést használva mutatjuk be.

Létrehozás

A táblázat mérete és a hash függvény is a rendelkezésre álló kulcsoktól függ. Egész kulcsot fogunk használi. Az elemek számára adott felső becslést a MERET konstans tartalmazza.

```
#define MERET 1000      /*Felső becslés az elemszámra.*/
#define KOZELI_PRIM 997 /*A hash függvénynél használjuk majd*/
#define URES -1         /*Nincs a táblázat adott helyén elem*/
#define ERVENYTELEN -2  /*Érvénytelen index.*/

typedef struct elem{
  int kulcs;  /*cikkszam*/
  int adat;   /*eladott mennyiseg*/
} ELEM;

/*A táblázat létrehozása*/
ELEM tabla[MERET];
```

Az alábbi függvény (meghívása!) a táblázat minden elemét üresre állítja.

```
void inicializalas(){
    int i;
    for(i = 0; i < MERET; ++i)
        tabla[i].kulcs = URES;
}
```

Végül még meg kell adnunk a hash függvényt. Az alábbi függvény egy a táblázat méretéhez közeli prímszámmal osztja el a kulcsot az index meghatározásához.

```
int hash(int kulcs){
    int idx = kulcs % KOZELI_PRIM;
    return idx >= 0 ? idx : ERVENYTELEN;
}
```

Beszúrás

Az elem kulcsát a hash függvénnyel leképezzük, majd elvégezünk pár ellenőrzést, hogy be lehet-e

egyáltalán szúrni a megadott elemet. A kedvező eset, ha a hash függvény olyan indexet adott meg, ahol a táblázat üres, ekkor csak az adott helyre betesszük az elemet. Ha viszont ott már volt egy elem (egy a mostanitól eltérő kulcsú elem már be lett szúrva az adott helyre: kulcsütközés van), akkor a példában nyílt címzést hasznáva tároljuk el az elemet. Ez azt jelenti, hogy az adott elem is a táblázatba kerül, de egy másik, még szabad helyre. A hely megkeresését a *szabadhelyKereso* végzi.

```
void beszur(int kulcs, int adat){
    int idx = hash(kulcs);

    if( idx == ERVENYTELEN ){
        printf("Hibás kulcs.\n");
        return;
    }
    if( holVan(kulcs) != -1){
        printf("Foglalt kulcs.");
        return;
    }

    /*egyedi kulcs, de utkozes van a lekepezesenel*/
    if ( tabla[idx].kulcs != URES ){
       idx = szabadhelyKereso(kulcs);
       if( idx == -1){
          printf("Nincs szabad hely.\n");
          return;
       }
    }
    /*beszúrás*/
    tabla[idx].kulcs = kulcs;
    tabla[idx].adat = adat;
}
```

A beszúrás során használt szabad hely kereséső a beszúrandó elem kulcs alapján számolt helyétől felvelé található első szabad hely indexét adja vissza. Ha nem volt üres hely, akkor a táblázat elejétől indulva keresi tovább a szabad helyet, ameddig körbe nem ér. A -1-es visszatérési érték a szabad hely hiányát jelzi.

```
int szabadhelyKereso(int kulcs){
    int idx = hash(kulcs); /*idx-re kellett volna beszúrni eredetileg*/

    int i;
    for(i = idx+1; i < MERET; ++i)
        if( tabla[i].kulcs == URES)
            return i;

    for(i = 0; i < idx; ++i)
       if( tabla[i].kulcs == URES)
            return i;

    return -1;
}
```

Keresés

Ahogy a beszúrás is, ez is attól függ, milyen szinonímakezlési módot használunk. A nyílt címzésnek megfelelően, ha az elemet a saját helyén nem találtuk meg (hash függvény által megadott index), akkor a szabad hely keresésénél ismertett módon haladunk körbe a táblázatban, ameddig az elemet meg nem találjuk, vagy el nem dől, hogy nincs az elem a táblázatban.

Megj.: Vigyázat, ha üres helyet találunk a hash függvény által megadott helyen, az még nem jelenti azt, hogy ez elem nincs a táblázatban. Törölni is lehet a táblázatból, ezért az az elem, ami miatt egy elem netán más helyre került, már nem feltétlenül van benne a táblázatban.

```
int holVan(int kulcs){
    int idx = hash(kulcs);

    if( idx == ERVENYTELEN )
        return -1;

    if( tabla[idx].kulcs == kulcs) /*hash függvény alapján megtaláltuk*/
       return idx;

    int i;                         /*nyilt címzésnek megfelelő keresés*/
    for(i = idx+1; i < MERET; ++i)
        if( tabla[i].kulcs == kulcs)
           return i;

    for(i = 0; i < idx; ++i)
        if( tabla[i].kulcs == kulcs)
           return i;

    return -1;                     /*Nem szerepel.*/
}
```

Törlés

Megkeressük az elemet az előbb megismert kereső segítségével, és ha szerepelt a táblázatban, akkor a kulcs értékét felülírjuk az üres helyet jelző speciális értékkel.

```
void torol(int kulcs){
    int idx = holVan(kulcs);
    if(idx == -1)
        printf("Nincs ilyen kulccsal elem.\n");
    else
        tabla[idx].kulcs = URES;

}
```

Csere

Megkeressük az elemet az előbb megismert kereső segítségével, és ha szerepelt a táblázatban, akkor az adat részét felülírjuk az új értékkel.

```
void csere(int kulcs, int ujadat){
    int idx = holVan(kulcs);
    if(idx == -1)
        printf("Nincs ilyen kulccsal elem.\n");
    else
        tabla[idx].adat = ujadat;

}
```

Bejárás

A bejárás során most is csak arra kell ügyelni, hogy az üres helyekkel ne foglalkozzunk.

```
void bejar(){
    int i;
    for( i = 0; i<MERET; ++i)
        if( tabla[i].kulcs != URES)
           feldolgoz( tabla[i]);
}
```

Újraszervezés

A közvetlen szervezésű struktúráknál írtak mellett akkor is szükség lehet újraszervezésre, sőt leginkább akkor van rá szükség, ha a szinonímák száma túlságosan megnő, lelassítva az elemekhez való hozzáférést. A részletek ismertetésétől most is eltekintünk.

2.6. Sor

A sor olyan adatszerkezet, mely biztosítja, hogy az elemek a bekerülés sorrendjükkel azonos sorrendben kerüljenek ki onnan. Két alapvető művelete van:

put(érték)	beteszi az *érték*et a sor végére
get()	kivesz egy értéke a sor elejéről

Folytonos tárolás esetén, a sor elemei tömbben kerülnek tárolásra. Ez azt jelenti, hogy bár a sor dinamikus adatszerkezet, valójában a tömb előre rögzített mérete behatárolja a sor elemszámát.

Hibakezeléssel csak érintőlegesen foglalkozunk. Később megismerkedünk olyan eszközzel, mellyel kulturált módon jelezhetjük, ha a függvény futása során hiba történt. Az itt lévő kód csak szöveges értesítést ad a hibáról! A get műveletnél lévő visszatérési érték is csak akkor tekinthető hibajelzésnek, ha a -1 NEM eleme a sornak!

Fix sor

A sor első eleme mindig a tömb első helyén áll (*sor[0]*), ezért a *get* művelet közben a sor megmaradó elemeit eggyel előrébb kell mozgatni a tömbben. A put művelet nem jár elemmozgatással, csak a sor végét jelző változó értékét növeljük, és a jelzett helyre beírjuk az új értéket.

```
const int N = 50; //A példában legfeljebb 50 elem lehet a sorban.
int sor[N]; //a sor elemeinek tárolására.
// fix sornál csak a vég mutatót szükséges tárolni
int v=-1;

int get(){
   int i, tmp = sor[0];
   if (v == -1){
      printf("Nincs elem a sorban!");
      return -1;
   }

   //a törlés eggyel balra csúsztatja az elemeket
   for(i = 0; i < v; ++i)
      sor[i] = sor[i+1];
   --v;
   return tmp;
}

void put(int x){
   if(v+1 == N)
      printf("A sor megtelt!");
   else
      sor[++v] = x;
```

```
}
```

Vándorló sor

Itt már sem a sor eleje, sem a vége nincs rögzítve a tömbben. A *get* műveletnél az sor elejét jelző mutató értéke növekszik eggyel, elemmozgatás nem történik. *Put* műveletnél a sor végére kerül be az elem. Ha ott már nincs hely, akkor a sor összes elemét balra visszük annyival, hogy a sor eleje a tömb elejére (tömb 0. pozíciójára) kerüljön, majd ez után kerül az elem beszúrásra.

```
#define N 50
int sor[N];    //a sor elemeinek tárolására
// vándorló sornál eleje, és vége mutatót is szükséges használni.
int e =0, v=-1;

//a törlés nem csúsztatja az elemeket
int get(){
   int i;
   if (v == e-1){
      printf("Hiba: a sor üres!\n");
      return -1;
   }
   ++e;
   return sor[e-1];
}

void put(int x){
   int i;
   if(e == 0 && (v+1) == N){
      printf("Hiba: a sor megtelt!\n");
      return;
   }
   //a beszúrás csúsztatja vissza az elemeket a tömb elejére,
   //ha már nincs hely a tömb végén
   else if( v+1 == N){
     for(i = e; i< N; ++i)
        sor[i-e] = sor[i];
     v -= e;
     e = 0;
   }
   sor[++v] = x;
}
```

Ciklikus sor

Sem a sor eleje, sem a vége nincs rögzítve a tömbben. A *get* műveletnél a sor elejét jelző mutató értéke eggyel nő, a *put* műveletnél a végét jelző mutató nő eggyel (modulo maximális elemszám). A modulo biztosítja, hogy a mutató ne menjen az indexhatáron kívülre. Eredményeként, ha a valamelyik mutató eléri a tömb végét, akkor 0-tól fog tovább növekedni.

```
#define N 50
int sor[N];    //a sor elemeinek tárolására
//ciklikus sornál eleje, és vége mutatót is szükséges használni.
int e = 0, v = -1;
```

```
int get(){
   int tmp;
   if (v < e){
      printf("Hiba: a sor üres!\n");
      return -1;
   }
   tmp = sor[ e % N ];
   ++e;
   //Az elmélettel ellentétben e és v korlátozva van az int
tartományra
   //Ezzel biztosíthatjuk, hogy ne lépjük túl.
   if(e>N) {
     e -= N;,
     v -= N;
   }
   return tmp;
}

void put(int x){
   if( v+1-e >= N ){
      printf("Hiba: a sor megtelt!\n");
      return;
   }
   sor[++v % N] = x;
}
```

Legyen a sor 5 elemű tömbben tárolva, valamint legyen adott *x=2*, és *y=3* egész típusú változók. A műveletek az alábbi változásokat okozzák a sorban. A sor vége félkövérrel jelölve, a sor eleje piros színnel.

Művelet	x,y	Fix sor	Vándorló sor	Ciklikus
put(3)	2, 3	**3** _ _ _ _	**3** _ _ _ _	**3** _ _ _ _
put(5)	2, 3	3 **5** _ _ _	3 **5** _ _ _	3 **5** _ _ _
put(2+x)	2, 3	3 5 **4** _ _	3 5 **4** _ _	3 5 **4** _ _
x = get()	3, 3	**5** 4 _ _ _	_ **5** 4 _ _	_ **5** 4 _ _
get()	3, 3	**4** _ _ _ _	_ _ **4** _ _	_ _ **4** _ _
get()	3, 3	_ _ _ _ _	_ _ _ _ _	_ _ _ _ _
put(1)	3, 3	**1** _ _ _ _	_ _ _ **1** _	_ _ _ **1** _
put(y-3)	3, 3	**1** 0 _ _ _	_ _ _ **1** 0	_ _ _ **1** 0
put(x*y)	3, 3	**1** 0 9 _ _	**1** 0 9 _ _	9 _ _ **1** 0
put(2)	3, 3	**1** 0 9 **2** _	**1** 0 9 **2** _	9 **2** _ **1** 0
put(get()+2) z=get()	3, 3	**0** 9 **2** _ _	_ **0** 9 **2** _	9 **2** _ _ **0**
y = get()	3, 0	**9** 2 3 _ _	_ _ **9** 2 3	**9** 2 3 _ _

2.7. Verem

Az elemek a bekerülés sorrendjéhez képest fordított sorrendben kerülnek ki a veremből. Két fontos művelettel foglalkozunk:

push(érték)	beteszi az értéket a verem tetejére.
pop()	kivesz egy eleme a verem tetejéről.

Folytonos reprezentáció esetén az elemek természetesen egy tömbben kerülnek tárolásra. Az egyszerűség kedvéjért a tömböt, és a verem tetejét jelző mutatót a sorokhoz hasonlóan, itt is globális változóként hozzuk létre.

Hibakezeléssel csak érintőlegesen foglalkozunk. Később megismerkedünk olyan eszközzel, mellyel kulturált módon jelezhetjük, ha a függvény futása során hiba történt. Az itt lévő kód csak szöveges értesítést ad a hibáról! A pop műveletnél lévő visszatérési érték is csak akkor tekinthető hibajelzésnek, ha a -1 NEM eleme a veremnek!

```
#define N 50   //Ennyi elem tarolhato a veremben.
int verem[N];  //A verem tarolasara szolgalo tomb.
int  v=-1;          //Verem teteje mutato: az utoljara bepakolt elem
indexe.

int pop(){
   if (v == -1){
      printf("Hiba: a verem üres!\n");
      return -1;
   }
   --v;
   return verem[v + 1];
}

void push(int x){
   if( v+1 == N ){
      printf("Hiba: a verem megtelt!\n");
   }else
   verem[++v] = x;

}
```

2.8. Egyirányban láncolt lista

Az adatelemek szétszórtan helyezkednek el a memóriában. Minden adatelem tartalmaz egy mutató típusú tagot is, ami a következő elem memóriabeli helyét tartalmazza (a következő elemre mutat). Ha nincs következő elem a mutató értéke *NULL* [3].

Megj.: Mivel a cél csak az adatszerkezet kezelésének bemutatása, feltételezni fogjuk, hogy a memória lefoglalása minden esetben sikeres. Egy-egy esetben a függvény alatti megjegyzésnél elolvasható milyen változtatást kell tenni, ha nem élünk ezzel a feltételezéssel, ami alapján a többi függvényhez is hozzáadható az ellenőrző rész.

Az összetett elem leírására struktúrával történik. A *typedef* a megadott típust ELEM-re nevezi.

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

Létrehozás

Mivel dinamikus adatszerkezetről van szó, üres listát hozunk létre. Tehát a lista feje jelenleg nem mutat érvényes elemre.

```
ELEM *fej = NULL;
```

[3] A C fordítók a NULL-t nullaként, vagy (void*)0-ként szokták definiálni. Egy *x* mutató típusú változónál ezért többféle módon is ellenőrizhető, hogy *x* értéke érvényes-e, pl: *x != NULL*, *!x*. Az *x != 0* használata viszont nem javasolt, sem a nulla NULL-al való helyettesítése!

Bővítés

A bővítéshez az elemnek megfelelő méretű helyet kell foglalni a memóriába, majd beszúrni az elemet a láncba. Az alábbi kódok megadásánál feltételezzük, hogy a memóriafoglalás sikeresen ment végbe (a malloc függvény nem NULL-al tér vissza).

Elem létrehozása:

```
ELEM* ujelem(int adat){
   //az uj elemnek lefoglaljuk a helyet
   ELEM *uj =(ELEM*)malloc(sizeof(ELEM));
   uj->adat = adat; //ertek beirasa
   uj->kov = NULL;  //ezzel sok bajt megspórolhatunk
   return uj;
}
```

Megj.: Ha a memóriafoglalás sikerességét is ellenőrizni szeretnénk, a malloc meghívása után szúrjuk be az alábbi sorokat:

```
if(uj == NULL){
   printf("Nincs elég memória.");
   return NULL;
}
```

Elem beszúrása a lista elejére:

```
ELEM* beszur_elore(ELEM *fej, int adat){
   ELEM *uj = ujelem(adat);
   /*az uj elem mutasson a lista elso elemere*/
   uj->kov = fej;
   /*az uj elemmel terunk vissza, hogy mostantol ez legyen a fej*/
   return uj;
}
```

Megj.: Ha az elem nem jött létre, akkor természetesen beszúrni sem tudjuk. Ha erre az opcióra is fel akarunk készülni, az alábbi sorokat szúrjuk be az ujelem függvény után:

```
if(uj == NULL){
   printf("Az elem nem került beszúrásra.");
   return fej;
}
```

Hasonló módon minden beszúrás függvénybe beépíthetjük az ellenőrzést.

Elem beszúrása a lista végére:

```
ELEM* beszur_vegere(ELEM *fej, int adat){
   ELEM *tmp, *uj = ujelem(adat);
   //Ha az új elem függvényben nem állítottuk be, itt kell
   //uj->kov = NULL;
   //ha ures volt a lista az uj elem lesz a fej
   if(fej == NULL)
      return uj;
   //kulonben elmegyunk a lista vegere
   for(tmp = fej; tmp->kov; tmp = tmp->kov)
```

```
        ;
    //az uj elemet az utolso moge fuzzuk
    tmp->kov = uj;
    return fej;
}
```

Elem beszúrása adott elem után:

Ennél a megoldásnál az elem az első *hova* értékű elem után kerül beszúrásra. Ez akkor lehet kényelmes, ha a listában egyedi értékek vannak. Természetesen írhatunk olyan függvényt is, ahol a felhasználótól már magát az elemet kérjük, ami után be akarja szúrni az elemet.

```
ELEM* beszur_adott_elem_utan(ELEM *fej, int hova, int mit){
    ELEM *tmp, *uj = ujelem(mit);
    uj->kov = NULL;
    //megkeressuk az adat erteku elemet
    for(tmp = fej; tmp && tmp->adat != hova; tmp = tmp->kov)
        ;
    //ha a for ciklus azert ert veget, mert megtalaltuk az elemet
    if(tmp != NULL){
        //akkor befuzzuk az elemet a tmp utani helyre
        uj->kov = tmp->kov;
        tmp->kov = uj;
    }
    return fej;
}
```

Törlés

Adott elem törlése

```
ELEM *torol(ELEM *fej, int mit){
    ELEM *tmp, *elozo = NULL;
    //megkeressuk a mit erteku elemet (a tmp-be lesz a vegen)
    for(tmp = fej; tmp && tmp->adat != mit; tmp = tmp->kov)
        elozo = tmp;

    //nem talaltuk a torlendo elemet
    if(tmp == NULL)
        return fej;

    //a fejet fogjuk torolni, ezert a fej a fej kovetkezoje lesz
    if(elozo == NULL)
        fej = fej->kov;
    //kulonben, a torlendo elem kovetkezoje a torlendo kovetkezoje lesz
    else
      elozo->kov = tmp->kov;
    free(tmp); //felszabaditjuk az elem helyet a memoriaban
    return fej;
}
```

Lista törlése

A lista minden elemének külön-külön foglaltunk helyet a memóriában, most minden lefoglalt helyet kell szabadítanunk. A lista elejétől indulva bejárjuk azt, és minden lépésben felszabadítjuk az

aktuális elemet. Egyetlen dologra kell figyelnünk, mielőtt töröljük az elemet, "kérjük el" a következő elem címét.

```
void felszabadit(ELEM *fej){
   ELEM *tmp;
   for(; fej; fej = tmp){
      //el kell menteni a kov. elem mutatojat.
      tmp = fej->kov;
      //mert ez utan a sor utan a fej->kov mar nem ervenyes hivatkozas
      free(fej);
   }
   return NULL;
}
```

Csere

Meg kell keresni a *fej*el adott listában az adott elemet (keres függvény lentebb), és az adatrészét felülírni!

```
ELEM* mmit = keres(fej, mit);
if(mmit) //ha megtaláltuk
   mmit->adat = mire;
```

Bejárás

A lista elejétől haladunk a lista végéig a láncon végiglépkedve. A *feldolgoz* egy képzeletbeli függvény, akármit csinálhat az adattal.

```
void bejar(ELEM *fej){
   for(; fej; fej = fej->kov)
      feldolgoz(fej->adat);
}
```

Keresés

Keressük meg a fejel adott listában a mit értékű elemet. Ha nem szerepel NULL-al, ha szerepel, akkor az adott elem címével térünk vissza.

```
ELEM *keres(ELEM *fej, int mit){
   for( ; fej && fej->adat != mit; fej = fej->kov)
     ;
   return fej;
}
```

Példák a fenti függvények hívására a főfüggvényből:

```
int main(){
   ELEM *fej = NULL, *keresett;
   fej = beszur_vegere(fej, 2);
   fej = beszur_elore(fej, 5); fej = beszur_vegere(fej, 1);
   fej = beszur_adott_elem_utan(fej, 2, 8);
   fej = beszur_adott_elem_utan(fej, 1, 9);
   fej = torol(fej, 5);
   /*A keresés eredményét nem írjuk vissza a fejbe!
     Elveszne a lista keresett elem előtti része*/
   keresett = keres(fej, 2);
```

```
    fej = felszabadit(fej);
    return 0;
}
```

2.9. Kétirányban láncolt lista

Az adatelemek szétszórtan helyezkednek el a memóriában. Minden adatelem tartalmaz két mutató típusú tagot is. Az egyik a megelőző, a másik a rákövetkező elemre mutat, amennyiben léteznek, különben értékük NULL.

Megj.: Most is feltételezni fogjuk, hogy a dinamikus memóriafoglalásnál sikeresen lefoglaltuk az elemnek szükséges helyet, de az egyirányban láncolt listáknál tett megjegyzések alapján itt is könnyűszerrel beépíthetjük azokat az ellenőrzéseket, mely a sikertelen memóriafoglalás esetre is felkészíti a függvényeket.

A lista elem leírására struktúrával történik.

```
typedef struct elem{
    int adat;
    struct elem *kov, *elozo;
} ELEM;
```

A lista *fej* és *vég* mutatóját egy külön struktúrába tároljuk. Ez a típus írja majd le a listát:

```
typedef struct lista{
    ELEM *fej;
    ELEM *veg;
} LISTA;
```

Létrehozás

Üres listát hozunk létre.

```
LISTA ujlista(){
    LISTA l = {NULL, NULL};
    return l;
}
```

Bővítés

Elem létrehozása:

A létrehozás során lefoglaljuk az elem számára a memóriaterületet. Lusták, és feledékenyek számára a *calloc* függvény használata javasolt, mert ki is nullázza a lefoglalt memóriaterületet, így az új elem *elozo* és *kov* mutatója automatikusan NULL lesz.

```
ELEM* ujelem(int adat){
    ELEM *uj = (ELEM*)calloc(1, sizeof(ELEM));
    //ertek beirasa uj->adat = adat;
    return uj;
}
```

Beszúrás a lista elejére:

A művelet során létrehozunk egy új elemet, és az elem következőjének beállítjuk a lista jelenlegi fejét. Megelőző eleme nem lesz, ezért az marad NULL (a calloc következtében). Ezután a lista fejét átállítjuk az új elemre. Egy speciális esettel is foglalkozni kell. Ha a lista üres, akkor az új elem lesz a lista feje, és vége is.

```
LISTA beszur_elore(LISTA lista, int adat){
   ELEM *uj = ujelem(adat);

   //ures lista volt
   if(!lista.fej){
      lista.fej = uj;
      lista.veg = uj;
      return lista;
   }
   //van elem
   uj->kov = lista.fej; lista.fej->elozo = uj; lista.fej = uj;
   return lista;
}
```

Beszúrás a lista végére:

A lista végére való beszúrásnál azt a speciális esetet, amikor a lista üres, elintézzük a lista elejére való beszúrással. Ha van már elem a listában, akkor létrehozunk egy elemet, és a megelőző elemeként beállítjuk a lista aktuális vég mutatóját. Majd a lista vég mutatóját az új elemre állítjuk.

```
LISTA beszur_vegere(LISTA lista, int adat){
   ELEM *uj;
   if(!lista.fej)
      return beszur_elore(lista, adat);

   uj = ujelem(adat);
   uj->elozo = lista.veg;
   lista.veg->kov = uj;
   lista.veg = uj;

   return lista;
}
```

Beszúrás adott elem után:

Ennél a megoldásnál az elem az első *hova* értékű elem után kerül beszúrásra (*miUtan*). Ehhez először megkeressük az elemet a listában (keres fv-t lásd lenetebb.). Ha nincs ilyen elem, nem csinálunk semmit, ha az elem megegyezik a lista végével, akkor a lista végére való beszúrást hajtunk végre. Egyébként létrehozzuk az új elemet, és befűzzük a láncba:

1. A *miUtan* elem következőjének a megelőzőjét állítsuk át az új elemre.
2. Az új elem következőjét állítsuk a *miUtan* következőjére.
3. Az új elem előzőjét állítsuk át a *miUtán*ra.
4. A *miUtán* elem következőjét állítsuk át az új elemre.

```
LISTA beszur_adott_elem_utan(LISTA lista, int hova, int mit){
   ELEM*tmp, *uj;

   ELEM* miUtan = keres(lista, hova);
   //nincs meg az elem
   if( miUtan == NULL )
      return lista;
   if( miUtan == lista.veg)
      return beszur_vegere(lista, mit);
```

```
    //kulonben az uj elemet letrehozzuk
    uj = ujelem(mit);
    miUtan->kov->elozo = uj;
    uj->kov = miUtan->kov;
    uj->elozo = miUtan;
    miUtan->kov = uj;
    return lista;
}
```

Törlés

Ahogy a beszúrásnál a törlésnél is több eshetőség van. Törölhetjük a lista fejét, végét vagy közbenső elemet. Ezeket most egy függvényen belül intézzük el, hogy teljes legyen a gyönyör ☺.

```
LISTA torol(LISTA lista, int mit){
    //megkeressuk a torlendo elemet
    ELEM *tmp = keres(lista, mit);

    //nincs meg
    if(tmp == NULL)
        return lista;
    //egy elemu lista torlese
    if( lista.fej == lista.veg){
        lista.fej = NULL;
        lista.veg = NULL;
    }
    //elso elem torlese (nincs tmp->elozo)
    else if( lista.fej == tmp){
        lista.fej = lista.fej->kov;
        lista.fej->elozo = NULL;
    }//utolso elem torlese (nincs tmp->kov)
    else if( lista.veg == tmp){
        lista.veg = lista.veg->elozo;
        lista.veg->kov = NULL;
    }//kozbenso elemet torlunk
    else{
        tmp->elozo->kov = tmp->kov;
        tmp->kov->elozo = tmp->elozo;
    }
    //felszabaditjuk a torlendo elemnek lefoglalt memoriat
    free(tmp);
    return lista;
}
```

Lista törlése

A lényeg ugyan az, mint az egyirányban láncolt listáknál. Ne felejtsük el a lista fej, és vég mutatóját NULL-ra állítani!

```
LISTA felszabadit(LISTA lista){
    ELEM *tmp, *e;
    for(e = lista.fej; e; e = tmp){
        tmp = e->kov;
        free(e);
```

```
    }
    lista.fej = NULL;
    lista.veg = NULL;
    return lista;
}
```

Bejárások

Bejárás a lista elejétől a vége felé haladva.

```
void bejar(LISTA lista){
    ELEM *e = lista.fej;
    for(; e; e = e->kov)
        feldolgoz(e->adat);
}
```

Bejárás a lista végétől az eleje felé haladva.

```
void bejar2(LISTA lista){
    ELEM *e = lista.veg;
    for(; e; e = e->elozo)
        feldolgoz(e->adat);
}
```

Keresés

```
ELEM* keres(LISTA lista, int mit){
    ELEM *f = lista.fej;
    for(   ; f && f->adat != mit; f = f->kov)
        ;
    return f;
}
```

Példa néhány függvény hívására a main függvényből:

```
int main(){
    LISTA lst = ujlista();
    lst = beszur_elore(lst, 4);
    bejar(lst);
    lst = beszur_vegere(lst, 9);
    lst = beszur_adott_elem_utan(lst, 4, 8);
    lst = torol(lst, 9);
    bejar2(lst);
    lst = felszabadit(lst);
}
```

2.10. Bináris fa

Hierarchikus adatszerkezet, ahol minden elemre igaz, hogy legfeljebb két rákövetkező eleme (gyerekeleme) lehet, és minden elemnek pontosan egy szülőeleme van, kivéve az úgynevezett gyökér elemet, melynek nincs szülő eleme. A fák tárolására szétszórt ábrázolást szokás alkalmazni. Az adat rész mellett minden adatelem tartalmaz két mutató típusú tagot is, melyek az elem bal, illetve jobb oldali leszármazottjára mutatnak. Ha a leszármazott nem létezik, akkor az adott oldali mutató NULL értéket vesz fel. A példában egészek tárolására alkalmas struktúrát hozunk létre:

```
typedef struct fa{
```

```
   int adat;
   struct fa *bal, *jobb;
} FA;
```

Létrehozás

Üres fát hozunk létre. Ezt bővítjük.

```
FA *gy = NULL
```

Bővítés

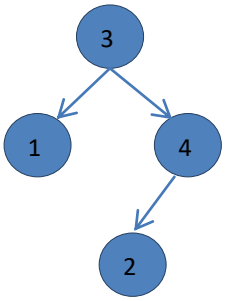
Elem létrehozása:

```
//Egy adott erteket "becsomagolunk egy elembe"
FA *ujelem(int adat){
   FA*uj = (FA*)malloc(sizeof(FA));
   if(!uj) return NULL;
   uj->adat = adat;
   uj->bal = uj->jobb = NULL; //nincs gyerekeleme return uj;
   return uj;
}
```

Elem beszúrása

Az új elemek mindig levélelemként kerülnek beszúrásra. A bináris fában az elemek elhelyezkedésére csak annyi megkötés van, hogy minden elemnek legfeljebb két gyerekeleme lehet, ezért a fa számos módon felépíthető. Használhatjuk például a keresőfáknál bemutatásra kerülő bővítő algoritmust egy aprócska módosítással: akkor is szúrjuk be az elemet, ha találtunk hasonló elemet a fában. (> jelet cseréljük le >= jelre)

Az ábrán látott aprócska fát felépíthetjük akár úgy is, hogy közvetlenül megadjuk melyik elem hova kerüljön. (Feltételezve, hogy az elem létrejött.)




```
FA* gyoker = NULL;
gyoker = ujelem(3);
gyoker->bal = ujelem(1);
gyoker->jobb = ujelem(4);
(gyoker->jobb)->bal = ujelem(2);
```

Bejárások

Preorder

Egy elem feldolgozása a gyerekelemei feldolgozása előtt történik meg. A feldolgozás a példában az adott elem kiírása lesz. Tehát feldolgozzuk az elemet, majd bejárjuk a baloldali részfáját preorder módon, végül bejárjuk a jobboldali részfáját preorder módon.

```
void preorder(FA* gy){
   if(gy != NULL){ //ha nem létezik a részfa nem csinálunk semmit
      printf("%d ", gy->adat);
      //bejarjuk az aktualis gyokerelem baloldali reszfajat
      preorder(gy->bal);
      //bejarjuk az aktualis gyokerelem jobboldali reszfajat
      preorder(gy->jobb);
   }
```

```
}
```

Inorder

Egy elem feldolgozása a bal oldali gyerekelemének feldolgozása után, de a jobb oldali gyerekelemének feldolgozása előtt történik meg.

```
void inorder(FA* gy){
   if(gy == NULL) return; //ha nem létezik a részfa azonnal kilépünk
   inorder(gy->bal);
   printf("%d ", gy->adat);
   inorder(gy->jobb);
}
```

Postorder

Egy elem feldolgozása gyerekelemének feldolgozása után történik meg.

```
void postorder(FA* gy){
   if(gy){ //így is ellenőrizhető, hogy létezik-e a részfa
      postorder(gy->bal);
      postorder(gy->jobb);
      printf("%d ", gy->adat);
}
```

Törlés

Elem törlés

Legegyszerűbben levél elem törölhető, hiszen csak az elem számára lefoglalt memóriát kell felszabadítani és a megfelelő oldali szülő elem mutatóját NULL-ra állítani. A bináris fában az elemek sorrendjére nincs semmilyen megkötés, ezért ha nem levélelem törlésére van szükség, akkor az elem érték részét felülírható egy levélelem érték részével, ami után már csak a levélelemet kell törölni a fából.

Részfa törlés

A függvénynek azt az elemet kell átadni, ahonnan induló részfát törölni akarunk.

```
FA* felszabadit(FA* gy){
   if(!gy) return NULL;
   felszabadit(gy->bal);
   felszabadit(gy->jobb);
   free(gy);
   return NULL;
}
```

A fenti példában a 4, 2 elemekből álló részfa törlése:

```
gyoker->jobb = felszabadit(gyoker->jobb);
```

2.11. Bináris kereső fa

A bináris keresőfa olyan bináris fa, melynek minden elemére igaz, hogy a bal oldalon lévő részfájában csak nála kisebb, a jobb oldali részfájában csak nála nagyobb kulcsú elemek

helyezkednek el. A *kulcs* kifejezés nem véletlen, nem lehet két azonos elem a fában! Mi most az adat részt fogjuk kulcsnak tekinteni, de természetesen létrehozható olyan faelem (struktúra) is, ahol a kulcs és adatrész elkülönül egymástól.

Létrehozás
Üres fát hozunk létre és bővítjük.

```
FA *gyoker = NULL;
```

Bővítés
Ha üres a fa, az új elem lesz a fa gyökere, különben az alábbi módon megkeressük az első olyan elemet, ahová beszúrható az elem.

1. Ha az új elem kisebb, mint a gyökér, akkor
    a. ha a gyökérnek nincs bal részfája, akkor szúrjuk be az elemet, és vége.
    b. különben legyen a gyökér a gyökér bal részfájának gyökere. Ugrás 1-re.
2. Különben ha az elem nagyobb, mint a gyökér
    a. ha a gyökérnek nincs jobb részfája, akkor szúrjuk be az elemet, és vége.
    b. különben legyen a gyökér a gyökér jobb részfájának a gyökere. Ugrás 1-re.
3. Különben hiba (két azonos kulcsú elem nem lehet)

Az algoritmus "szó szerinti" megvalósítása:

```
FA *beszur(FA *gy, int adat){
   if(!gy)
      return ujelem(adat);
   if(gy->adat > adat){
      if(!gy->bal){
         gy->bal = ujelem(adat);
         return gy;
      }
      beszur(gy->bal, adat);
   }
   if(gy->adat < adat){
      if(!gy->jobb){
         gy->jobb = ujelem(adat);
            return gy;
      }
      beszur(gy->jobb, adat);
   }
   return gy;
}
```

A fenti helyett használhatunk egy tömörebb megoldást. Itt akkor is felülírjuk egy elem bal vagy jobb oldali gyerekére mutató részét, ha az már létezett. De ez semmi problémát nem fog okozni, mert a megfelelő részfa gyökerét fogjuk visszaírni. Ha elsőnek talán furcsának is tűnik ez a módszer, a kód tömörsége bizonyára megfelelő kárpótlást nyújt:

```
FA *beszur(FA *gy, int adat){
```

```
    if(!gy)
       gy = ujelem(adat);
    else if(gy->adat > adat)
       gy->bal = beszur(gy->bal, adat);
    else if(gy->adat < adat)
       gy->jobb = beszur(gy->jobb, adat);
    return gy;
}
```

Megj.: A rekurzív függvényeket átírhatjuk nem rekurzív formára is. Az alábbi függvény hossza talán kellően rávilágít arra, hogy miért használunk olyan gyakran önmagát hívó függvényeket.

```
FA *beszur(FA *gy, int adat){
   FA *aktualis, *uj = ujelem(adat);

   //ha nem volt meg elem a faban, akkor az uj elem lesz a gyoker
   if(gy == NULL) return uj;

   aktualis = gy;
   for( ; ; ){
      //az aktualis elemtol jobbra kell beszurni az elemet
      if(aktualis->adat < adat){
         //nincs jobb oldali gyerek, az elem beszurhato
         if(aktualis->jobb == NULL){
            aktualis->jobb = uj;
            return gy;
         }
         //a jobb oldalon keressuk tovabb az elem helyet
         aktualis = aktualis->jobb;
      }
      //a bal reszfaba kell szurni az elemet
      else if(aktualis->adat>adat){
         if(aktualis->bal == NULL){
            aktualis->bal = uj;
            return gy;
         }
         //ha nem tudtunk beszurni a bal reszfaban vizsgalodunk tovabb
         aktualis = aktualis->bal;
      }
      // mar szerepel az elem, nem szurjuk be ujra
      else
         return gy;
      }
}
```

Törlés

A törlést úgy kell végrehajtani, hogy a fa továbbra is keresőfa maradjon. Ezt a következő módon tehetjük meg:

1. Ha az elem levélelem, akkor töröljük: szülője rá mutató mutatóját NULL-ra állítjuk, és töröljük az elemet (memória felszabadítás).
2. Különben, ha az elemnek egy gyereke van, akkor a törlendő elem szülőjének a törlendő elemre mutató mutatóját átállítjuk a törlendő elem gyerekére. Majd töröljük az elemet.
3. Különben (mindkét oldalon van gyerek) a törlendő elem bal részfának a"legjobboldalibb", vagy a jobb részfájának a "legbaloldalibb" elemével (levélelem) felülírjuk a törlendő elemet, és a levélelemet töröljük a fából.

Keresés

Az elem keresésénél a fa felépítésénél használt szabály alapján minden lépésnél tudjuk, hogy az

adott elemtől merre helyezkedik el a keresett elem. Az alábbi függvény a keresett elemmel tér vissza, ha az szerepel a fában, különben NULL-al.

```
FA *keres_rekurzivan(FA *gy, int mit){
   if(gy == NULL) return NULL;
   if(gy->adat == mit)
      return gy;
   if(gy->adat < mit)
      return keres_rekurzivan(gy->jobb, mit);
   // ha gy->adat > mit
   return keres_rekurzivan(gy->bal, mit);
}
```

2.12. Általános fa

A bináris fával ellentétben az általános fáknál egy elemnek tetszőleges számú leszármazottja lehet, ezért az elem gyerekelemeinek nyilvántartása láncolt listával, vagy ha ismert a maximális gyerekszám, akkor tömbbel történhet. A jegyzetben az utóbbi megoldás kerül ismertetésre.
Feltételezzük, hogy minden elemnek legfeljebb *N* számú leszármazottja lehet, melyek folytonosan kerülnek tárolásra egy tömbben. Minden elemnél tároljuk azt is, hogy aktuálisan hány gyerekeleme van. Egy egészek tárolására alkalmas faelemet leíró struktúra ekkor a következően alakul:

```
typedef struct afa{
   int adat;
   struct afa *gyerekek[N];
   int gyerekszam;
} AFA;
```

Létrehozás
Üres fát hozunk létre, ezt bővítjük.

```
AFA *gyoker = NULL;
```

Bővítés
Elem létrehozása:
Az eddigiekhez hasonló módon történik. A *calloc* függvény biztosítja, hogy a *gyerekszam* kezdetben nulla legyen.

```
AFA *ujelem(int adat){
   AFA*uj = (AFA*)calloc(1, sizeof(AFA));
   if(!uj) return NULL;
   uj->adat = adat;
   return uj;
}
```

Beszúrás
Levélelemmel bővítünk. Az alábbi függvény azt a létező szülőelemet várja 1. paraméterként, mely gyerekelemeként az új elemet be akarjuk szúrni. A 2. paraméter a beszúrandó adat. Ha a szülőelemnél van hely, akkor beszúrjuk az elemet, és növeljük eggyel a szülőnél tárolt gyerekszám információt. Eltérő esetben nem kerül beszúrásra az új elem.

```
AFA* beszur( AFA* szulo, int adat){
   if (szulo && szulo->gyerekszam < N){
      szulo->gyerekek[ szulo->gyerekszam ] = ujelem(adat);
      ++(szulo->gyerekszam);
   }else{
      printf("Az elem nem beszúrható az adott elem gyerekelemeként.");
   }
   return szulo;
}
```

Megj.: Természetesen most is készíthető olyan függvény, mely a fa gyökerétől indulva keresi meg hova szúrható be az elem. Az alábbi kód véletlenszerűen keresi meg az új elem leendő helyét. Első paraméterként a fa gyökerét, második paraméterként a beszúrandó adatot várja.

```
AFA* beszurRandom( AFA *gy, int adat){
   int idx = 0;
   if(!gy) return ujelem(adat);
   idx = rand() % N;
   //gy-nek nincs idx számú gyereke. Az adat beszúrásra kerül.
   if ( idx >= gy->gyerekszam){
      gy->gyerekek[gy->gyerekszam] = ujelem(adat);
      ++gy->gyerekszam;
   }
   //gy->gyerek[idx] letezik, abba az ágba kerül majd beszúrásra az elem
   else
      beszurRandom( gy->gyerekek[ idx ], adat);
   return gy;
}
```

Bejárás

A *preorder* módon való bejárásnál a (rész)fa gyökere kerül először feldolgozásra, majd létező gyerekelemektől induló részfa kerül bejárásra.

```
void preorder(FA* gy){
   int i;
   if( !gy ) return;
   printf("%d ", gy->adat);
   for(i = 0; i < gy->gyerekszam; ++i)
      preorder(gy->gyerekek[i]);
}
```

Ha nem létezik a (részfa), akkor egyből kilépünk. Ha létezik, akkor kiírjuk az elemet, majd egy *for* ciklus segítségével végigjárjuk a gyerekelemeket tartalmazó tömböt. (Az *i* értékét természetesen nem befolyásolja a függvényhívás, hiszen a függvények lokális változóiból minden hívásnál új "készletet" kapunk.)

A *postorder* módon való bejárásnál egy elem mindig csak akkor kerül feldolgozásra, ha a gyerekelemek már feldolgozásra kerültek.

```
void postorder(FA* gy){
   int i;
   if( !gy ) return;
   for(i = 0; i < gy->gyerekszam; ++i)
      postorder(gy->gyerekek[i]);
   printf("%d ", gy->adat);
}
```

Törlés

Egész fa törlése

A függvény a fa gyökerét várja paraméterként. A felszabadításnál most is postorder bejárást használunk, hiszen az elem gyerekelemei törlése után törölhető csak ki maga az elem.

```
AFA* felszabadit(AFA* gy){
   int i;
   if( !gy ) return NULL;
   for(i = 0; i < gy->gyerekszam; ++i)
      felszabadit(gy->gyerekek[i]);
   free(gy);
   return NULL;
}
```

Részfa törlése

Természetesen részfa törléséhez is felhasználható a *felszabadit* függvényt, viszont ha nem az egész fát töröljük, akkor a szülő elemmel kapcsolatos "adminisztrációs" teendőket is el kell látni. Egyrészt csökkenteni kell a gyerekelemek számát, másrészt biztosítani kell, hogy továbbra is folytonosan helyezkedjenek el a gyerekelemek.

```
AFA *torolReszfa(AFA *szulo, AFA* torlendo){
   int i;
   if(!szulo || !torlendo) return szulo;
   for( i = 0; i < szulo->gyerekszam; ++i)
      if(szulo->gyerekek[i] == torlendo){
         --szulo->gyerekszam;
         //a törlendő helyett mutasson a korábbi jobb szélső elemre
         szulo->gyerekek[i] = szulo->gyerekek[szulo->gyerekszam];
         felszabadit(torlendo);
         break;
      }
}
```

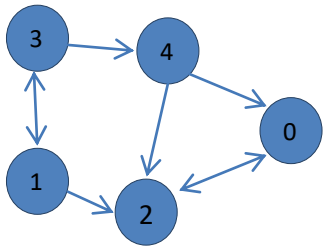
Keresés

```
AFA* keres(AFA* gy, int mit){
   int i;
   AFA *talalt = NULL;
   if( !gy || gy->adat == mit) return gy;
   for(i = 0; i < gy->gyerekszam; ++i){
      talalt = keres(gy->gyerekek[i], mit);
      if (talalt) return talalt;
   }
   return NULL;
}
```

2.13. Gráf

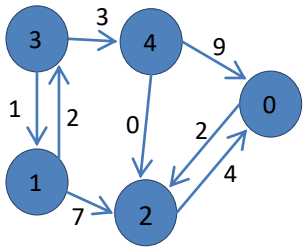
A matematika gráf fogalma megjelenése adatszerkezet szintjén. A véges gráfok reprezentálásának elterjedt módja a szomszédsági (kapcsolat) mátrixszal való leírás. Ez egy N csúcsból álló gráf esetén *NxN-es mátrixot jelent.* A gráf csúcsait feleltessük meg *[0, N-1]* zárt intervallumban lévő egész számoknak kölcsönösen egyértelmű módon. A mátrix *i.* sorának,

és *j*. oszlopának metszéspontjában *1*-es áll, ha *i*-ből *j*-be vezet él, különben *0*. Ezzel egy bitmátrixot kapunk. Irányítatlan gráf esetén a mátrix természetesen szimmetrikus lesz.



csúcs	0	1	2	3	4
0	0	0	1	0	0
1	0	0	1	1	0
2	1	0	0	0	0
3	0	1	0	0	1
4	1	0	1	0	0

Ha a gráf éleihez súlyokat rendeltünk, akkor *1*-es helyett, az adott súlyt szerepeltetjük a mátrixban. Ilyenkor, ha *0*-s súly is előfordulhat a gráfban, akkor az él hiányát végtelennel szokás jelölni (programban egy olyan érték, amit nem lehet valós súly).



csúcs	0	1	2	3	4
0	∞	∞	2	∞	∞
1	∞	∞	7	2	∞
2	4	∞	∞	∞	∞
3	∞	1	0	0	3
4	9	∞	0	∞	∞

Megj.: Kapcsolatmátrixot akkor érdemes használni, ha az elemek száma állandó, vagy nagyon ritkán változik, ugyanis egy új csúcs felvétele a mátrixban is új sor, és oszlop beszúrását eredményezné, a törlés pedig sor és oszlop törléssel járna, és mint tudjuk a mátrix statikus adatszerkezet.

Létrehozás

Az *N* elemhez tartozó kapcsolatmátrix egy *NxN*-es mátrix létrehozását és megfelelő elemekkel való feltöltését jelenti (irányítatlan mátrix esetén használhatunk háromszögmátrixot is a tárolásra). Súlyozatlan gráf esetében akár bitmátrixot is használhatunk. Súlyozott esetben a súly típusa határozza meg a mátrix típusát. Mi most egész számokból álló mátrixot fogunk használni az elemek tárolására.

```
TIPUS kapcsMatrixNeve[CsucsokSzama][CsucsokSzama];
int M[N][N];
//...
//Bejegyezzük a mátrixba a kapcsolatokat.
//...
```

Bővítés

Az *i.* csúcsból *j.* csúcsba vezető él beszúrás:

```
M[i][j] = 1;
```

Súlyozott gráfnál, *i.*-ből *j.*-be futó *w* súlyú él hozzáadása:

```
M[i][j] = w;
```

Megj.: Ha a gráf irányítatlan és ennek ellenére nem háromszögmátrixot használunk a tárolásra, akkor az M[j][i] értékét is át kell írni.

Törlés
Az *i.* csúcsból *j.* csúcsba vezető él törlése:

```
M[i][j] = 0;
```

Súlyozott esetben az él hiányát jelző speciális érték kerül a 0 helyére.

Megj.: Ha a gráf irányítatlan és ennek ellenére nem háromszögmátrixot használunk a tárolásra, akkor az M[j][i]-t is át kell írni.

Csere
A kapcsolatmátrixot nem érinti. (A csúcsok *[0, N-1]* intervallumra való leképezése változik.)

Keresés
A klasszikus értelemben vett keres itt nincs, hiszen az eredeti csúcsok közt és nem a kapcsolatmátrixban kell az elemet keresni. Ehelyett az lehet a kérdés elérhető-e egy elem egy adott elemből, lásd bejárások.)

Rendezés: -

Bejárás: A bejárás során az elemeket összefüggő részenként érjük el, az elemek elérés sorrendje a választott bejárás mellett attól is függ, hogy mely elemtől indítjuk a bejárást.

Szélességi bejárás
A bejárás során egy nyílt, és egy zárt halmazt használunk. A nyílt halmazt a sorként kezeljük, tehát a végére bővítünk, és az elejéről vesszük ki a kiterjesztendő elemet! A gráf tetszőleges csúcspontjából kiindulhatunk. (Ha az adott csúcsból nem érünk el minden elemet, a bejárás újraindul egy másik, még nem bejárt csúcstól.)

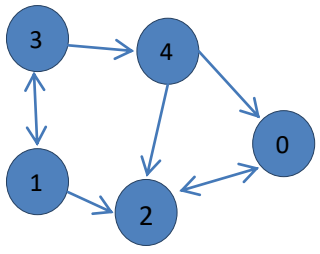
1. A kezdő elemet betesszük a nyíltak közé.
2. Ameddig a sor nem üres

   Vegyük ki a sor első elemét X-be. X feldolgozása.
   Terjesszük ki X-et.
   Tegyük X-et a zárt halmazba.

A kiterjesztés során X szomszédos elemei bekerülnek a nyílt halmazba, ha sem a nyílt sem a zárt halmazban nem szerepelnek.

Az algoritmus "adminisztrálásához" egy csúcshoz kapcsolódva annak szülő elemét, és a kiinduló elemtől való távolságát (költségét) szokás tárolni (*Elem(szülő, lépésszám/költség)*). Az elemek sorként való kezelése automatikusan biztosítja, hogy mindig a(z egyik) legkisebb költségű elem kerül kiterjesztésre.

Járjuk be az alábbi gráfot a 3-as csúcsból kiindulva!



Nyílt	Zárt	Megjegyzés
3(-, 0)	-	
1(3,1), 4(3, 1)	3(-, 0)	
4(3, 1), 2(1, 2)	1(3, 1)	A 3-ast nem kell újra felvenni, mert szerepel a zártban.
2(1, 2), 0(4, 2)	4(3, 1)	A 2(4,2)-t nem kell felvenni, mert a 2-es szerepel a nyíltban.
0(4, 2)	2(1, 2)	A 0(4,2)-t nem felvenni, mert a 0 szerepel a nyíltban

Mélységi bejárás

Az egyetlen változás, hogy sor helyett a nyílt halmazt verem adatszerkezetként kezeljük, emiatt mindig a legnagyobb költségű elem(ek egyike) kerül kiterjesztésre. A fenti gráf bejárása a hármas csúcsból:

Nyílt	Zárt	Megjegyzés
3(-, 0)	-	
1(3, 1), 4(3, 1)	3(-, 0)	
2(1, 2), 4(3, 1)	1(3, 1)	A 3-ast nem kell újra felvenni, mert szerepel a zártban.
0(2, 3), 4(3, 1)	2(1, 2)	
4(3, 1)	0(2, 3)	A 0-ásból csak a 2-es érhető el, de az már szerepel a zártak közt.
-	4(3, 1)	A 4-esből csak olyan elem érhető el, ami szerepel a zártban.

Ha a fenti bejárásokat keresésre használva az algoritmus akkor ér sikeresen véget, ha a kiterjesztés során elértük az elemet. Ha a nyílt halmaz kiürül az elem megtalálása előtt, akkor az elem nem szerepel a gráfban.

Megj.: Ha egy elemhez vezető legjobb utat akarjuk megkeresni, akkor a kiterjesztés részt változtatjuk: Vegyük sorra X szomszédjait. Amennyiben egy szomszéd nem X és nem szerepel sem a nyílt sem a zárt halmazban, vagy ott nagyobb költséggel szerepel, akkor tegyük bele a nyílt halmazba, a rosszabb költségűt pedig vegyük ki.

3. Feladatok és megoldásaik

A feladatok megadásánál a függvény deklarációját fogjuk megadni. A feladat minden esetben a függvény kiegészítése. A feladatokat lehetőleg gép mellett oldjuk meg, az elkészített függvényeket ellenőrizzük is. A legjobb, ha a teljes programot sikerül önállóan megírni, de ameddig ez nem megy, használd fel a jegyzethez mellékelt kódrészleteket (honlapon).

3.1. Témakör: Tömb

1. Egy mérőállomás által meghatározott hőmérsékleti adatokat egy 1 dimenziós tömbben tároljuk. A tömb minden eleme egy-egy napra vonatkozó érték.
   a. Írj függvényt, mely paraméterben egy hónapra vonatkozó hőmérsékleti adatokat (*n* elemű *meres* nevű tömb) vár és meghatározza az adott hónap átlaghőmérsékletét.

      ```
      float haviAtlag(int meres[], int n);
      ```

   b. Írj függvényt, mely paraméterben egy adott időszakra vonatkozó hőmérsékleti adatokat (*n* elemű *meres* nevű tömb) vár és meghatározza mennyi volt a maximális hőmérséklet.

      ```
      int maxHomerseklet(int meres[], int n);
      ```

   c. Írj függvényt, mely paraméterben egy adott időszakra vonatkozó hőmérsékleti adatokat (*n* elemű *meres* nevű tömb) vár és meghatározza melyik napon volt a legmelegebb (maximum hely). Ha több is volt, az 1. ilyen napot adja vissza a függvény.

      ```
      int legmelegebbNap(int meres[], int n);
      ```

   d. Írj függvényt, mely paraméterben egy adott időszakra vonatkozó hőmérsékleti adatokat (*n* elemű *meres* nevű tömb) vár és **kiírja** mely **napo(ko)n** volt a legmelegebb. Használd fel hozzá a korábban megírt maxHomerseklet függvényt.

      ```
      void legmelegebbNap(int meres[], int n);
      ```

2. Számolja ki egy N elemű, egészekből álló tömb elemeinek átlagát!
3. Számolja meg hány pozitív szám áll egy N elemű, valós tömbben!
4. Számolja ki két N dimenziós vektor belső szorzatát. (Mindkét vektor egy N elemű tömbbel van reprezentálva.)
5. Határozza meg egy egészekből álló tömb legnagyobb elemének helyét!
6. Határozza meg egy NxM-es mátrix legnagyobb elemét!

Megoldások

1.

```
int osszeg(int t[], int n){
   int i, sum = 0; //kezdőérték adás ne maradjon le!
   for(i = 0; i<n; ++i)
      sum += t[i];
}
```

2.

```
double atlag(int t[], int n){
   int i, sum = 0;
   for(i = 0; i<n; ++i)
      sum += t[i];
   return sum/(double)n; //egészosztás: 1/2 = 0, de 1/(double)2 = 0.5
}
```

3.

```
int pozitiv(double t[], int n){
   int i, count = 0;
   for(i = 0; i<n; ++i)
      if(t[i]>0)
         ++count;
   return count;
}
```

4.

```
double szorzat(double v[], double v2[], int n)
   int i;
   double s = 0;
   for(i = 0; i<N; ++i)
      s += v[i]*v2[i];
   return s;
}
```

5.

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

6.

```
int maximum(int t[N][M]){
   int i,j, max = t[0][0];
   for(i = 0; i<N; ++i)
      for(j = 0; j<M; ++j)
         if(max < t[i][j])
            max = t[i][j];
   return max;
}
```

3.2. Témakör: halmaz, multihalmaz

Minden feladatnál feltételezzük, hogy az alaphalmaz egész számokból álló [0, N-1] intervallum.

1. Add meg a {0, 3, 5, 2} halmaz 6 hosszú karakterisztikus függvényét.
2. Add meg a {0, 0, 3, 3, 3, 4} halmaz 6 hosszúságú karakterisztikus függvényét.
3. Mi az eredeti halmaz, ha a karakterisztikus függvénye: 0010110?
4. Mi az eredeti multihalmaz, ha a karakterisztikus függvénye: 3,3,0,0,4,1,2
5. Egészítsd ki az alábbi kódot, úgy, hogy E a következő halmazművelet eredményét tartalmazza: A∪ (B∩C)

```
void AuBmC(int A[N], int B[N], int C[N], int E[N]){
   int i;
   for( i = 0;___;++i)
      E[i] = ____;
}
```

6. Egészítsd ki az alábbi kódot, úgy, hogy *R* a következő halmazművelet eredményét tartalmazza: Ā∪ (B\C)

```
void ABC(int A[N], int B[N], int C[N], int R[N]){
int k;
for( k = 0; k<N ;      ++k)
     _ = ___;
}
```

Megoldások

1. 1 0 1 1 0 1
2. 2, 0, 0, 3, 1, 0
3. {2, 4, 5}
4. {0, 0, 0, 1, 1, 1, 4, 4, 4, 4, 5, 6, 6}
5. i<N A[i] || (B[i] && C[i])
6. R[k] !A[k] || (B[k] && !C[k])

3.3. Témakör: Ritkamátrix

1. Adott a következő 3 soros reprezentáció. Mi volt az eredeti mátrix, ha tudjuk, hogy 3x4-es volt, és a gyakori elem 0.

Sor	0	0	1	1	2
Oszlop	0	2	1	3	2
Érték	4	6	5	2	2

2. Adott a következő ritkamátrix. Készítse el a 4 soros reprezentációját, ha a gyakorielem a 0!

$$\begin{pmatrix} 2 & 0 & 6 & 0 & 0 \\ 1 & 3 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 1 \end{pmatrix}$$

Megoldások

1.

$$\begin{pmatrix} 4 & 0 & 6 & 0 \\ 0 & 5 & 0 & 2 \\ 0 & 0 & 2 & 0 \end{pmatrix}$$

Sor	0	0	1	1	1	2	2
Oszlop	0	2	0	1	2	2	4
Érték	2	6	1	3	1	2	1
Mutató	2	4	-1	-1	5	-1	-1

3.4. Témakör: Egyirányban láncolt lista

A listaelem definíciója legyen a következő:

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

1. Írj függvényt, mely kiszámolja a fejével megadott lista elemeinek összegét!
2. Írj függvényt, mely kiszámolja a fejével megadott lista elemeinek átlagát!
3. Írj függvényt, mely kiszámolja a páros értékű adatelemek szorzatát!
4. Írj függvényt, mely fordított sorrendben kiírja a listaelemeket! (rekurzív fv.)

Megoldások

1.

```
int osszeg(ELEM *x){
   int ossz = 0;
   for( ; x ; x = x->kov)
   ossz += x->adat;
   return ossz;
}
```

2.

```
float atlag(ELEM *fej){
   int osszeg = 0, elemszam = 0;
   for( ; fej ; fej = fej->kov){
      osszeg += fej->adat;
      ++elemszam;
   }
   //nullával való osztás elkerülésére
   if(elemszam == 0)
      return .0f;
   return osszeg/(float)elemszam;
}
```

3.

```
int szorzat(ELEM *f){
   int s = 1;
```

```
   for( ; f; f = f->kov)
   if( f->adat % 2 == 0)
      s *= f->adat;
   return s;
}
```

4.

```
void kiirForditva(ELEM *f){
   if( !f ) return;
   kiirForditva(f->kov);
   printf("%d ", f->adat);
}
```

Egy egyirányban láncolt listában versenyzők adatait akarjuk tárolni. Minden versenyzőnek van egy egész típusú azonosítója, egy maximum 19 betűből álló neve, valamint egy 0-10 közötti egész pontszáma.

1. Hozd létre az ennek megfelelő struktúrát, és nevezd el ezt a típust *LISTELEM*-nek.
2. Írj egy függvényt, mely kiírja a listában tárolt versenyzők minden adatát!
3. Írj függvényt, mely visszaadja a legmagasabb ponszámot!
4. Írj egy függvényt, mely azonosító, név, pontszám hármasból elkészít egy listaelemet!
5. Írj egy függvényt, mely egy nevet és egy pontszámot vár paraméterként és a lista végére beszúrja új elemként! Az azonosító legyen eggyel nagyobb, mint az előző elemé!
6. Írj függvényt, mely a nyertest (legmagasabb pontszám) leíró listaelemmel tér vissza! (Feltételezheted, hogy a pontszámok egyediek.)
7. Írj eljárást, mely kiírja a nyertest (nyerteseket)! (Használható a 3. feladatban megírt függvény, vagy feltételezhető hogy létezik.)
8. Írjon függvényt mely visszaadja azoknak a versenyzőknek a listáját, melyek 5 pont alatt teljesítettek! (Használja az 5. feladatban megírt függvényt, v. feltételezze, hogy létezik.)

Megoldások

1. Természetesen más struktúranév és mezőnevek is használhatók.

```
typedef struct elem{
   int azonosito;
   char nev[20]; //'\0' záró karakter miatt foglalunk 20-nak helyet
   unsigned char pontszam; //más számtípus is használható
   struct elem *kov;
}LISTELEM;
```

2. A %-20s húsz karakteren való balra igazított kiírást jelent, ami a feladat megoldás szempontjából lényegtelen de áttekinthetőbb lesz a megjelenített lista.

void kiir(LISTELEM *fej){

```
for(  ; fej; fej = fej->kov)
   printf("%2d %-20s %2d\n", fej->azonosito, fej->nev, fej->pontszam);
   printf("\n");
}
```

3.

```
unsigned int maxpontszam(LISTELEM *p){
```

```
   //azért indíthatjuk 0-ról, mert ennél kisebb pontszám
   //nem érhető el.
   unsigned int max = 0;
   for( ; p; p = p->kov)
      if(p->pontszam > max)
         max = p->pontszam;
   return max;
}
```

4.

```
LISTELEM* ujelem(int azonosito, char *nev, unsigned char pontszam){
   LISTELEM *uj = (LISTELEM*)malloc(sizeof(LISTELEM));
   if(!uj) return NULL;
   uj->azonosito = azonosito;
   uj->pontszam = pontszam;
   strcpy(uj->nev, nev);
   uj->kov = NULL;
   return uj;
}
```

5.

```
LISTELEM*  beszur_vegere(LISTELEM  *fej,  char  *nev,  unsigned  int
pontszam){
   LISTELEM *tmp;
   if(fej == NULL)
      return ujelem(0, nev, pontszam );
   for(tmp = fej; tmp->kov; tmp = tmp->kov)
      ;
   tmp->kov = ujelem(tmp->azonosito+1, nev, pontszam );
   return fej;
}
```

6.

```
LISTELEM* nyertes(LISTELEM *p){
   LISTELEM *legjobb = p; //az aktuális legjobb legyen a lista első
eleme
   for(; p; p = p->kov)
      if(p->pontszam > legjobb->pontszam)
         legjobb = p;
   return legjobb;
}
```

7.

```
void nyertesekKiirasa(LISTELEM *p){
   int max;
   if ( !p )
      printf("Nincs versenyző.");
   else{
      max = maxpontszam(p);
      for( ; p; p = p->kov)
         if( p->pontszam == max )
```

```
            printf("%2d                %20s                %2d\n",
                p->azonosito,p->nev,p->pontszam);
    }
}
```

8.

```
LISTELEM* otPontAlatt(LISTELEM* ls){
   LISTELEM *lista5 = NULL; //ez lesz az új lista feje.
   for(; ls; ls = ls->kov)
   if( ls->pontszam < 5 )
   lista5 = beszur_vegere(lista5, ls->nev, ls->pontszam);
   return lista5;
}
```

3.5. Témakör: Általános fa

Legyen a faelemét leíró struktúra a következő:

```
typedef struct t{
   int data;
   struct t *children[4];
   int numOfChildren;
} Tree;
```

1. A gyökér elemre mutató változó neve *root*. Hogyan éred el a 8-as elemet?
2. Járd be a fát preorder és postorder módon!
3. A gyökér elemre mutató változó neve *root*. Írasd ki hány gyerekeleme van a 7-es elemnek! (root-on, keresztül elérve!)
4. Írj függvényt, mely megadja mennyi a faelemek összege!
5. Írass ki minden páros értékű elemet!
6. Szorozd meg az összes 5-nél nagyobb elemet kettővel!

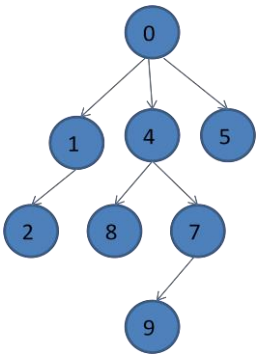
Megoldások

1. `root->children[1]->children[0]`
2. preorder: 0 1 2 4 8 7 9 5 postorder: 2 1 8 9 7 4 5 0
3. `printf("%d ", root->children[1]->children[1]->numOfChildren);`
4.

```
int osszeg(Tree *r){
   int i, s = 0;
   if (!r) return 0;
   for (i = 0; i<r->numOfChildren; ++i)
      s += osszeg(r->children[i]);
   return s + r->data;
}
```

5.

```
void postorder(Tree * gy){
   int i;
   if( !gy ) return;
   for (i = 0; i<gy->numOfChildren; ++i)
      postorder(gy->children[i]);
```



```
    if( gy->data % 2 == 0)
       printf("%d ", gy->data);
}
```

6.

```
void otnelNagyobbatDuplazo(Tree* gy){
   int i;
   if( !gy ) return;
   for (i = 0; i<gy->numOfChildren; ++i)
      otnelNagyobbatDuplazo(gy->children[i]);
   if( gy->data > 5 )gy->data *= 2;
}
```

A faelemet leíró struktúrában egy vállalat alkalmazottainak adatait tároljuk: név, fizetés, közvetlen beosztottakra mutató mutatókat tartalmazó tömb, és a beosztottak száma. N a közvetlenbeosztottak számának lehetséges maximuma.

```
typedef struct dolgozo{
   char nev[20];
   float fizetes;
   struct dolgozo *beosztottak[N];
   int unsigned beosztottakSzama;
} DOLGOZO;
```

1. Listázd ki a dolgozók nevét és fizetését!
2. Hány olyan alkalmazott dolgozik, akiknek van beosztottja?
3. Hány dolgozó van a cégnél?
4. Mennyi a legmagasabb fizetés a cégnél?
5. Írd ki azokat a dolgozókat, akiknek egy megadott értékkel egyező számú közvetlen
1. beosztottja van!
6. Listázd ki, melyik dolgozó irányítása alatt hányan dolgoznak összesen (Tehát nem csak a közvetlen beosztottak számát.)
7. Keresd meg, hogy egy adott dolgozó szerepel-e a cég alkalmazottai közt. Ha igen, térj vissza a dolgozót tároló elem mutatójával, különben NULL-al!

Megoldások

1.

```
void dolgozokListaja(DOLGOZO* gy){
   int i;
   if( !gy ) return;
   printf("%s %0.2f\n", gy->nev, gy->fizetes);
   for(i = 0; i < gy->beosztottakSzama; ++i)
      dolgozokListaja(gy->beosztottak[i]);
}
```

2. A feladat nem más, mint a NEM levélelemek megszámlálása.

```
int csoportVezetokSzama(DOLGOZO *gy){
   int i, s = 0;
   if (! gy) return 0;
```

```
   for (i = 0; i<gy->beosztottakSzama; ++i)
      s += csoportVezetokSzama(gy->beosztottak[i]);
   return s + (gy->beosztottakSzama != 0);
}
```

3.

```
int dolgozokSzama(DOLGOZO *gy){
   int i, s = 0;
   if (! gy) return 0;
   for (i = 0; i<gy->beosztottakSzama; ++i)
      s += dolgozokSzama(gy->beosztottak[i]);
   return s + 1;
}
```

4.

```
float maxFizetes(DOLGOZO* gy){
   int i;
   float max = 0, tmp;
   if(!gy) return 0;
   max = gy->fizetes;
   for (i = 0; i<gy->beosztottakSzama; ++i)
      if( (tmp = maxFizetes(gy->beosztottak[i])) > max )
         max = tmp;
   return max;
}
```

5.

```
void xKozvetlenBeosztott(DOLGOZO *gy, int x){
   if(gy){
      int i;
      if(gy->beosztottakSzama == x)
         printf("%s\n", gy->nev);
      for( i = 0; i<gy->beosztottakSzama; ++i)
         xKozvetlenBeosztott(gy->beosztottak[i], x);
   }
}
```

6.

```
void hanyBeosztott(DOLGOZO *gy){
   int i;
   if(!gy) return;
   printf("%s %d\n", gy->nev, dolgozokSzama(gy)-1);
   for(i = 0; i<gy->beosztottakSzama; ++i)
      hanyBeosztott(gy->beosztottak[i]);
}
```

7.

```
DOLGOZO* keres(DOLGOZO* gy, char* kit){
   int i;
   DOLGOZO *talalt = NULL;
   if( !gy || !strcmp(gy->nev, kit)) return gy;
```

```
   for(i = 0; i < gy->beosztottakSzama; ++i){
      talalt = keres(gy->beosztottak[i], kit);
      if (talalt) return talalt;
   }
   return NULL;
}
```

4. Gyakorló feladatok megoldások nélkül

A kódírásos feladatokat érdemes gép előtt gyakorolni. A fordító majd "szól" a szintaktikai hibák miatt, a kiírások alapján pedig látszik, ha nem jól működik a program. A folyamatos visszajelzés rengeteget segít a fejlődésben.

4.1. Bemelegítő gyakorlatok (C alapok)

1. Kérj be egy számot, és dönts el róla, hogy pozitív-e?
2. Kérj be egy számot, és döntsd el, hogy páros-e? (maradékos osztás jele: %)
3. Kérj be két számot, és írd vissza a nagyobbat.
4. Kérj be három számot, és írd vissza a legnagyobbat.
5. Írj programot mely 10-szer kiírja a "Márpedig én átmegyek!" mondatot! (*for ciklus*)
6. Írasd ki az [5, 10] intervallumba első számok köbét!
7. Olvass be számokat a konzolról, amíg nullát nem kapsz.
8. Olvass be számokat a konzolról nulláig. Az aktuális értéket mindig add hozzá a korábbiakhoz, és írd is ki.
9. Írj 21-es játékot! ([0, N-1] intervallumba eső véletlenszám generálása: rand()%N)

4.2. Rövid függvények

A függvényeket minden esetben hívd is meg!

1. Írj függvényt, mely két egész számot vár paraméterként, és azok összegét adja vissza.
2. Írj eljárást mely két egész számot vár paraméterként és kiírja azok szorzatát.
3. Írj függvényt mely igaz értékkel tér vissza, ha a paramétere páros, hamissal, ha nem.
4. Írj függvényt mely igaz értékkel tér vissza, ha az 1. paramétere nagyobb, mint a 2.
5. Írj függvényt, mely két egész szám hányadosát adja vissza! A nullával való osztás elvégzése helyett írj ki valamilyen hibaüzenetet.
6. Írj függvényt mely annyiszor írja ki az első paraméter értékét, amennyi a 2. paraméter értéke. Az első paraméter legyen egy karakter, a második legyen egy nemnegatív egész szám.
7. Írj függvényt az n faktoriális kiszámítására! Az n-t kérd a felhasználótól.

4.3. Tömbös, mátrixos feladatok

A teszteléshez érdemes olyan méretű tömböt, és olyan elemeket választani, hogy a megoldást le is tudd ellenőrizni. Az alábbi kód használható keretnek a tömb feldolgozására irányuló feladatoknál.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 12 // ha más elemszámot akarsz ezt változtasd

//min max közti értékekkel lesz tele a tömb
int feltolt(int t[], int n, int min, int max){
   int i, m = max - min + 1;
   for(i = 0; i<n; ++i)
      t[i] = rand() % m + min;
   }
```

```
void kiir(int a[], int meret){
   int i;
   for(i = 0; i<meret; ++i){
      printf("%d ", a[i]);
   }
}


int main(){
   //véletlenszám generátor inicializálása
   srand(time(NULL));
   int tomb[N];

   // így tudod feltölteni például -20 és 20 közti értékekkel
   feltolt(tomb, N, -20, 20);

   //írhatod ide a kódod, vagy készíthetsz függvényt.

   kiir(tomb, N);

   getchar();
   return 0;
}
```

1. Hozz létre egy 5 elemű egész számokat tartalmazó tömböt, és írd ki a benne lévő értékeket. (Ne rémülj meg, most éppen azt látod, mi történhet, ha nem adsz kezdőértéket.)
2. Hozz létre egy 5 elemű egészeket tartalmazó tömböt, és olvasd be konzolról az elemeket.
3. Tölts fel 0-10 közti értékekkel egy 10 elemű egész tömböt, és add össze az elemeit! (*rand()%11*)
4. Tölts fel véletlen értékekkel egy 8 elemű valós tömböt, és számold ki az elemek átlagát!
5. Tölts fel véletlen értékekkel egy 10 elemű valós tömböt, és minden elemet cseréld ki a négyszeresére.
6. Tölts fel egy tetszőleges méretű egész tömböt, és minden 0-s értéket cserélj ki 100-ra!
7. Keresd meg egy tömbben a legnagyobb páratlan értékű elemet! (Alakítsd át a maximum keresést.)
8. Keresd meg egy tömbben a legkisebb értékű elemet a páros indexűek közül! (Alakítsd át a minimumkeresést.)
9. Minden tömbelemet csúsztass 2-vel balra. Az utolsó két elemet nullázd ki.
10. Olvass be egy számot, és az attól balra lévő tömbelemeket csúsztasd el eggyel jobbra.

11. Hozz létre 3 sorból és 4 oszlopból álló mátrixot és írd ki a 2. sorának elemeit. (Index szerint 1-est.) 12. Keresd meg egy mátrixban a legnagyobb elemet!
13. Határozd meg a mátrixban lévő 4-el osztható elem átlagát!
14. Számold meg oszloponként hány olyan elem van, amely 0-nál nagyobb!
15. Keresd meg egy mátrixban azt az oszlopot, melyben az elemek összege minimális!
16. Tárold el egy *t* tömbben azt, hogy melyik mátrixelemből hány darab van. Feltételezve, hogy a mátrixban lévő értékek a 0 és 5 zárt intervallumba esnek. (pl.: ha 3 nullát, és 1 kettest és 2 hármast és 4 darab 5-öst találnál a mátrixban, akkor t = {3, 0, 1, 2, 0, 5}, mint a multihalmaz reprezentációnál.)
17. Írj fel egy tetszőleges mátrixot, amelyben a gyakori elem az 5-ös! Készítsd el a 3 és 4 soros

reprezentációját!
18. Adott a következő 3 soros reprezentáció. Mi volt az eredeti mátrix, ha tudjuk, hogy 4x6- os volt, és a gyakori elem 0.

Sor	0	0	1	1	3	3
Oszlop	0	4	3	5	3	5
Érték	4	6	5	2	2	5

4.4. Egyirányban láncolt listás feladatok:

1. Írasd ki az egyirányban láncolt lista minden második elemét!
2. Írasd ki az egyirányban láncolt lista páros elemeit!
3. Számold ki a 10-nél nagyobb listaelemek összegét!
4. Számold ki a 10-nél nagyobb listaelemek átlagát! (Vigyázat, nem az össze elem számával kell osztani!)
5. Keress meg egy adott k számot a listában! (Teljes keresés)
6. Keress meg egy adott k számot egy rendezett listában! (Lineáris keresés)
7. Számold meg hány 5-ös érték van a listában!
8. Írjon függvényt, mely eldönti van-e két azonos érték a listában?
9. Írjon függvényt, mely x értékű elemek értékét y-ra cseréli!
10. Rajzold fel, milyen mutatókat, és milyen sorrendben kell átállítania, ha egy 9, 10, 13, 15-ös listában a 13-as elé szúr be egy 3-as elemet!
11. Tf.: van egy rendezett listája! Írjon függvényt, mely beszúr egy új elemet a megfelelő helyre.
12. Írjon olyan függvényt, mely egy adott elem elé szúr be elemet!

4.5. Kétirányban láncolt listás feladatok

Minden, ami az egyirányú listánál is szerepel itt is jó gyakorlásnak!

1. Írassa ki a lista elemeket fordított sorrendben!
2. Rajzolja le, milyen mutatókat milyen sorrendben állítana át, ha az elejére szúr be egy elemet!
3. Rajzolja le, milyen mutatókat milyen sorrendben állítana át, ha a végére szúr be egy elemet!
4. Rajzolja le, milyen mutatókat milyen sorrendben állítana át, ha egy közbenső elem elé szúr be egy elemet!
5. Rajzolja le, milyen mutatókat milyen sorrendben állítana át, ha egy közbenső elem utá szúr be egy elemet!
6. Keressen meg az első *k* értékű számot a listában! (Teljes keresés)
7. Keressen meg az utolsó *k* értékű számot egy rendezett listában! (Lineáris keresés)
8. Tételezze fel, hogy betűket tartalmaz a lista! Ellenőrizze le, hogy palindroma-e a szó (pl.: a, abba, apa, görög)
9. Adott egy *t[N] egész* tömb. Hozzon létre listát, mely a *t*-ben szereplő, különböző értékeket tartalmazza. (Segítség: végig kell menni a tömbben, az adott értékű elemet keresni a listában. Ha benne volt, akkor nem kell csinálni semmit, ha hiányzott be kell szúrni! Használhatók a jegyzetben szereplő beszúrásra és keresésre szolgáló függvények.)

Irodalomjegyzék

- N1570 Committee Draft (April 12, 2011) ISO/IEC 9899:201x
  http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1570.pdf

- G. Ivanyos, L. Rónyai, R. Szabó, Algoritmusok, Műszaki Könyvkiadó, Budapest, 1996