

A PROGAMOZÁS ALAPJAI 1

6. előadás

Vitéz András
egyetemi adjunktus
BME Híradástechnikai Tanszék
vitez@hit.bme.hu

2012. március 13.

Miről lesz ma szó?

- Felsorolás típus
- Indirekció
  - Mutatók
  - Mutatók függvény paramétereként
  - Mutató aritmetika
  - Tömbök és mutatók
  - Többdimenziós tömbök
  - Karakter tömbök, string típus, string kezelő függvények
- Vektoralgoritmusok (folytatás) - Keresések
  - Lineáris
  - Logaritmikus
  - Várható lépésszám
  - Kereső függvények paraméterezése

Felsorolás

felsorolás
(enumeráció)

szimbolikus néven hivatkozott
egész típusú állandók
összefogása egy típussá

Felsorolás

felsorolás deklaráció alakja:

enum [<felsorolás címke>]
{
felsorolás lista (vesszővel elválasztott értékek)
}
[<felsorolás változó azonosítók>] ;

felsorolt érték
felsorolt érték = konstans kifejezés

Felsorolás

értékek hozzárendelése:

Ha van = jel,

akkor a megadott kifejezéstől,

egyébként 0-tól

egyesével növekvő értékek

Felsorolás

Példa:

```
enum evszak {
                tavasz, nyar, osz, tel
                } e1,e2;
enum evszak e3,e4;
typedef enum {
                 piros, narancs=2, sarga,
                 zold, kek=8, lila
                } szin;
szin sz1,sz2;
```

Mutatók

- A mutató típusú adat egy másik adat címét tárolja
- Bár minden mutatót ugyanolyan belső ábrázolással tárolunk, típusa lényeges,
  mert a hivatkozott objektum programban játszott szerepét ez határozza meg.

Mutatók

Deklarációja:

<mutatott típus> * <mutató azonosítója>

Itt a * a bal oldalhoz tartozik,
vagyis a mutató típusa
<mutatott típus>*

Mutatók

Deklarációja:

A * az indirekció operátora, és
- mint minden operátor-
elválasztható az operandusoktól

Mutatók

Deklarációja:

Például

```
int*a;
int* a;
int *a;
int * a;
```

egyenértékűek

Mutatók

Deklarációja:

Ez a többalakúság hasznos, ha egyetlen deklarációs utasítással ezt is – azt is szeretnénk deklarálni.

Például

```
int a,*b,c,*d;
```


Mutatók

Használata:

*<mutató azonosítója>

jelenti a mutatott objektumot

Mutatók

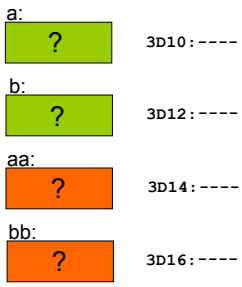
Használata:

```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
aa=bb;
```


Mutatók

Használata:

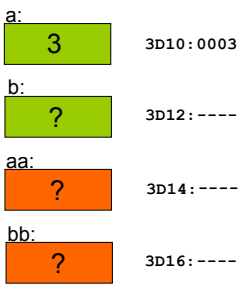
```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
aa=bb;
```



Mutatók

Használata:

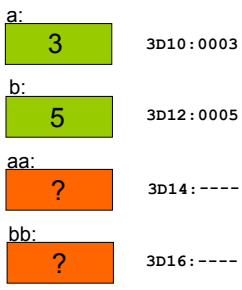
```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
aa=bb;
```



Mutatók

Használata:

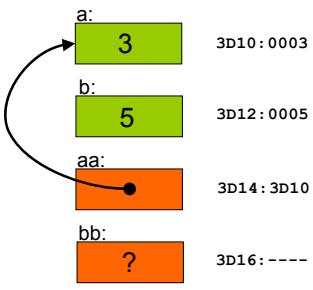
```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
aa=bb;
```



Mutatók

Használata:

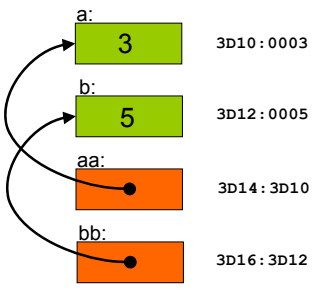
```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
aa=bb;
```



Mutatók

Használata:

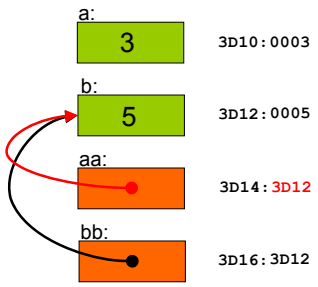
```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
aa=bb;
```



Mutatók

Használata:

```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
aa=bb;
```



Mutatók

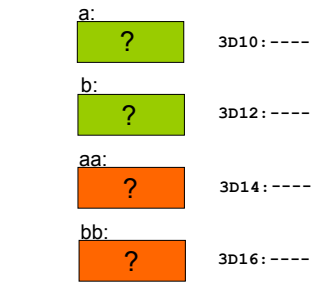
Használata:

```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
*aa=*bb;
```


Mutatók

Használata:

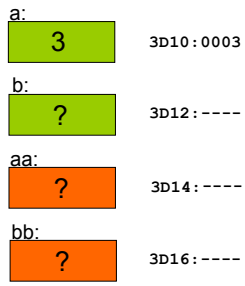
```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
*aa=*bb;
```



Mutatók

Használata:

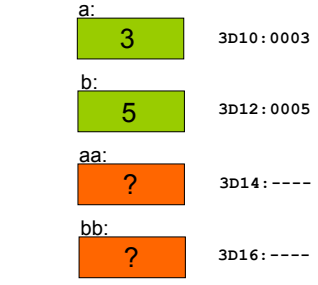
```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
*aa=*bb;
```



Mutatók

Használata:

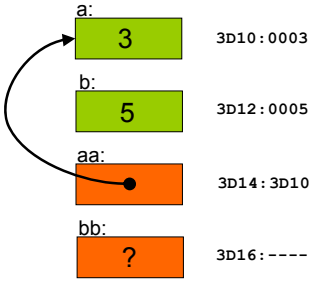
```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
*aa=*bb;
```



Mutatók

Használata:

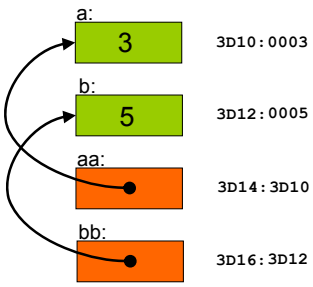
```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
*aa=*bb;
```



Mutatók

Használata:

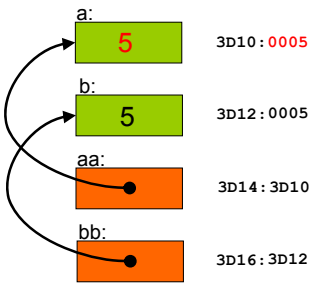
```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
*aa=*bb;
```



Mutatók

Használata:

```
int a,b,*aa,*bb;
a=3;
b=5;
aa=&a;
bb=&b;
*aa=*bb;
```



Mutatók

Különlegességek:

Minden mutatótípussal kompatibilis,
de nem hivatkozható

void*

Mutató, amely nem mutat sehová

0

Mutatók függvény paramétereként

Mikor használjuk?

Ha meg akarunk változtatni egy adatot,
de nem tudjuk előre, hogy melyiket,
vagy a függvényből nem volna látható.

Mutatók függvény paramétereként

Példa:

```
void csere (int *a,int *b)
{
   int c=*a;
   *a=*b;
   *b=c;
}
```


Mutatók függvény paramétereként

Példa:

```
void hegylako (int *a,int *b)
{
   *a>*b
   ?
   ((*a+=*b),(*b=0))
          /* a elveszi b életerejét */
   :
   ((*b+=*a),(*a=0));
          /* b veszi el a életerejét */
}
```

Mutató aritmetika

Alapgondolata:

A lépésköz a mutatott típus ábrázolási mérete

Mutató aritmetika

Például

int *a,*b;

esetén értelmes kifejezések

a+3

b++

b-a

Kivonni csak azonos típusú mutatókat szabad!

Tömbök és mutatók

- A tömb azonosítójának jelentése
  a 0 indexű elem címét tartalmazó konstans mutató
- A fordítóprogram
  minden a[x] alakú hivatkozást
  *(a+x) alaknak értelmez
- Mutatót függvény formális paramétereként így is deklarálhatunk:
  int a[]

Tömbök és mutatók

Például

Szeretnénk egy hat elemű egész tömböt b néven, de az első elem indexe ne 0, hanem 3 legyen, vagy egy c nevű első elemének indexe -2 legyen.

int a[6],*b,*c;

b=a-3;

c=a+2;

b[3]	b[4]	b[5]	b[6]	b[7]	b[8]
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
c[-2]	c[-1]	c[0]	c[1]	c[2]	a[3]

Vigyázat! b és c szabadon változtatható, de *a* konstans!

Többdimenziós tömbök

- Igazából egydimenziós

Például

int a[4][3];

elemtípusa nem int, hanem három elemű int tömb

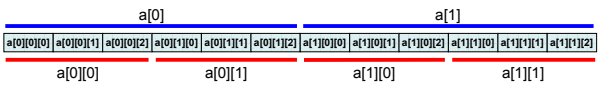
formális paraméterként
int b[][3]
ugyanilyen mutató

Többdimenziós tömbök

- A tömbelemek a memóriában egymás után helyezkednek el
- Ezt nevezik sorfolytonos tárolásnak

Például

int a[2][2][3]; esetén

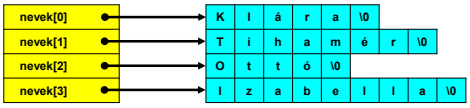


Mutató tömbök

- Ha a tömb elemei tömbök lennének, de nem egyforma hosszúságúak, mutatók tömbjét szoktunk használni
- Tipikusan szövegek kezelésekor szükséges

Például

```
char *nevek[4];
```



Többdimendiós tömbök paraméterként

A következő mátrixot adatként tároljuk programunkban, és szeretnénk mátrix formában kiíratni:

1 0 2 1
3 0 0 1
2 2 1 0

Többdimendiós tömbök paraméterként

Az első változatban csak a sorfolytonos feltöltést mutatjuk meg.

A mátrixot globális változóként érjük el.

Így nincs is paraméter átadás.

Többdimendiós tömbök paraméterként

```
#include <stdio.h>
int x[3][4]={1,0,2,1,3,0,0,1,2,2,1,0};
void printa(){
      int i,j;
      for (i=0;i<3;i++){
            for (j=0;j<4;j++)
                  printf("%4d",x[i][j]);
            printf("\n");}
}
int main(){
      printa();
      return 0;
}
```


Többdimendiós tömbök paraméterként

A mátrixot paraméterként átadhatjuk, mint 4 elemű tömbök tömbjét.

Voltaképpen mutatót adunk át, ezért alkalmazzuk a már látott szintaxist.

Többdimendiós tömbök paraméterként

```
#include <stdio.h>
int x[3][4]={1,0,2,1,3,0,0,1,2,2,1,0};
void printb(int b[][4]){
      int i,j;
      for (i=0;i<3;i++){
            for (j=0;j<4;j++)
                  printf("%4d",b[i][j]);
            printf("\n");}
}
int main(){
      printb(x);
      return 0;
}
```

Többdimendiós tömbök paraméterként

A tömb elemtípusához (4 egész tömbje) új nevet is rendelhetünk.

Ezek tömbjét adhatjuk át mutatóként.

Többdimendiós tömbök paraméterként

```
#include <stdio.h>
int x[3][4]={1,0,2,1,3,0,0,1,2,2,1,0};
typedef int i4[4];
void printc(i4*c){
      int i,j;
      for (i=0;i<3;i++){
            for (j=0;j<4;j++)
                  printf("%4d",c[i][j]);
            printf("\n");}
}
int main(){
      printc(x);
      return 0;
}
```


Többdimendiós tömbök paraméterként

Az univerzális megoldásban a tömb mérete is paraméter lesz.

Kihasználjuk a sorfolytonos tárolást.

A függvény hívásakor kényszerített típusmódosításra van szükség.

Többdimendiós tömbök paraméterként

```
#include <stdio.h>
int x[3][4]={1,0,2,1,3,0,0,1,2,2,1,0};
void printd(int*d,int m,int n){
      int i,j;
      for (i=0;i<3;i++){
            for (j=0;j<4;j++)
                  printf("%4d",d[n*i+j]);
            printf("\n");}
}
int main(){
      printd((int*)x,3,4);
      return 0;
}
```


Többdimendiós tömbök paraméterként

Átadhatunk mutatótömböt is.

Az adatnak is ebben a formában kell lennie.

Többdimendiós tömbök paraméterként

```
#include <stdio.h>
int x[3][4]={1,0,2,1,3,0,0,1,2,2,1,0};
int*y[3]={x[0],x[1],x[2]};
void printe(int**e,int m,int n){
      int i,j;
      for (i=0;i<m;i++){
            for (j=0;j<n;j++)
                  printf("%4d",e[i][j]);
            printf("\n");}
}
int main(){
      printe(y,3,4);
      return 0;
}
```

Karakter tömbök

Szövegek ábrázolása a C nyelvben

String típus

Karakter tömbként

```
char ct [5] = {'a','l','m','a', '\0'};
```

Körülményes!

Helyette:

```
char ct [5] ="alma";
```

Ugyanazt jelenti:

ct:	a	l	m	a	\0

String típus

Karakter tömbként

```
char ct [] ="alma";
```

Ugyanaz!

Sőt:

```
char ct [] ="al" "ma" ;
```

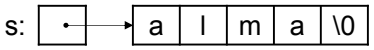
ct:	a	l	m	a	\0

ugyanis az előfeldolgozó egyesíti!

String típus

Karakter mutatóként

```
char *s ="alma";
```



String típus

Különbség karakter és string között

'a'	'a'	1 byte
"a"	'a' \0	2 byte

String típus

Különbség karakter és string között

''		HIBA!
""	\0	üres string

String kezelő függvények (string.h)

Összehasonlítás

```
int strcmp (char *op1, char *op2)
  {
    while (*op1 && *op1 == *op2)
        {op1++; op2++;}
    return *op1-*op2;
  }
```


String kezelő függvények (string.h)

Másolás (ész nélkül!)

```
char *strcpy (char *dest, char *source)
  { char *d=dest;
    while (*d++ = *source++);
    return dest;
  }
```


String kezelő függvények (string.h)

Másolás (ésszel)

```
char *strmove (char *dest,char *source);
```


String kezelő függvények (string.h)

String hossza (a lezáró \0 nélkül)

```
size_t strlen (char *str)
  { char *s =str;
    while (*s++);
    return s-str-1;
  }
```


String kezelő függvények (string.h)

Megtalált karakter címe a stringben

```
char *strchr (char *str, char ch)
  {while (*str && *str != ch) str++;
   if (*str == ch)  return str;
   return 0;
  }
```


Vektoralgoritmusok

Amit már ismerünk:

- elemek összege
- elemek szorzata
- elemek átlaga
- elemek minimuma
- elemek maximuma
- eldöntés
- leszámlálás
- szétválogatás

Keresések

Emlékeztető:

Az ***eldöntési feladat*** egyik megfogalmazása:

Van-e a vektornak olyan eleme,
amely rendelkezik egy adott tulajdonsággal?

Keresések

Keresési feladat:

Van-e a vektornak olyan eleme,
amely rendelkezik egy adott tulajdonsággal?

Ha van, melyik az első?

A tulajdonság legyen az, hogy a tárolt elem
valamelyik része megegyezik egy konkrét értékkel.

Az elemnek ezt a részét nevezzük
a keresés kulcsának.

Keresések

Lineáris keresés

Az első elemtől kezdve egyesével vizsgáljuk
az elemeket, amíg
- a keresett elemet meg nem találjuk,
- vagy ki nem derül, hogy nincs ilyen elem.

Keresések

Lineáris keresés

A vektor elemtípusa legyen struktúra,
aminek egyik tagja legyen a kulcs!

```
typedef /* valami */ . . .  kulcs_tipus;
typedef struct {
          . . .  /* tetszőleges tagok */
         kulcs_tipus kulcs;
          . . .  /* további tagok */
        } tombelem;
```


Keresések

Lineáris keresés

Ha függvényként valósítjuk meg,
- milyen paramétereket adjunk át,
- mi legyen a visszatérési érték?

```
??? keres(tombelem*t,int n,kulcs_tipus k)
```


Keresések

Lineáris keresés

Visszaadhatjuk a megtalált elemet:

```
tombelem keres()
```

Kényelmes, de nem tudjuk, hol volt!
Mit adjunk vissza, ha nem találtunk megfelelőt?!!

Keresések

Lineáris keresés

Visszaadhatjuk a megtalált elem indexét:

```
int keres()
```

Egyszerű, az elemet indexeléssel elérhetjük!
Mit adjunk vissza, ha nem találtunk megfelelőt?!!
- negatív indexet (Pl. -1)
- n-et (ilyen indexű elem már nincs!)

Keresések

Lineáris keresés

Visszaadhatjuk a megtalált elem címét:

```
tombelem* keres()
```

Egyszerű, az elemet indirekcióval elérhetjük!
Mit adjunk vissza, ha nem találtunk megfelelőt?!!
null-pointert (könnyű tesztelni!)

Keresések

Lineáris keresés

```
tombelem* keres(tombelem*t,
                int n,
                kulcs_tipus k)
{
   while (n) if(t->kulcs==k)return t;
             else t++,n--;
   return 0;
}
```


Keresések

Lineáris keresés

A keresés várható lépésszáma
(ha a tömb mérete N)

?

Keresések

Lineáris keresés

A keresés várható lépésszáma
(ha a tömb mérete N)

N

Olyan kulcs érték, amely nincs tárolva a tömbben, sokkal több létezik, mint olyan, amely tárolva van.

Keresések

Lineáris keresés

A lépésszám csökkenthető, ha a tömbben tárolt elemek rendezett halmazt alkotnak, és a tömb a kulcs szerint rendezett.

Keresések

Lineáris keresés rendezett tömbben

A keresés várható lépésszáma
(ha a tömb mérete N)

?

Keresések

Lineáris keresés rendezett tömbben

A keresés várható lépésszáma
(ha a tömb mérete N)

N/2

A tárolt kulcsok megtalálásához átlagosan N/2 lépés szükséges. Nem tárolt kulcsok keresésekor átlagosan N/2 lépés után dől el, hogy nincs meg. (meghaladtuk)

Keresések

Lineáris keresés rendezett tömbben

```
int keres(tombelem*t,int n,kulcs_tipus k)
{
  int i;
  for(i=0;(i<n)&&(t[i].kulcs<k);i++);
  return i;
}
```

Ha megvan, az indexét, ha nincs, a helyét adjuk vissza.
Ez további vizsgálatot igényel, de később még jól jön!

Keresések

Ha a tömb rendezetlen, csak
a lineáris keresés használható!

Ha a tömb rendezett, van jobb!

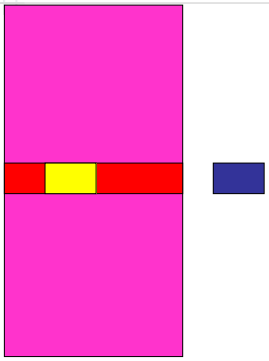
Keresések

Logaritmikus (bináris) keresés

Minden egyes összehasonlító lépésben
a keresési tartomány középső elemét vizsgáljuk.

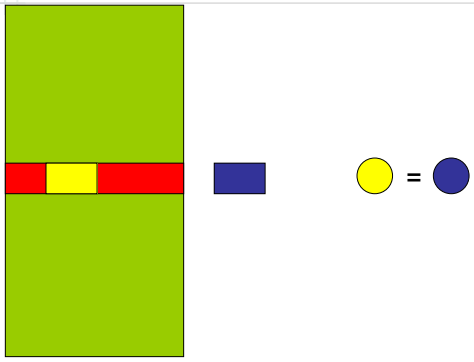
Ezt az elvet már láttuk, amikor számkitaláló játékot játszottunk.

Logaritmikus (bináris) keresés

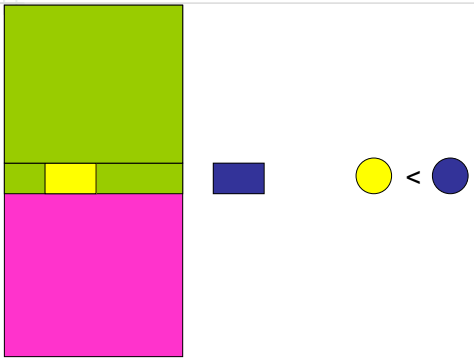


? ? ?

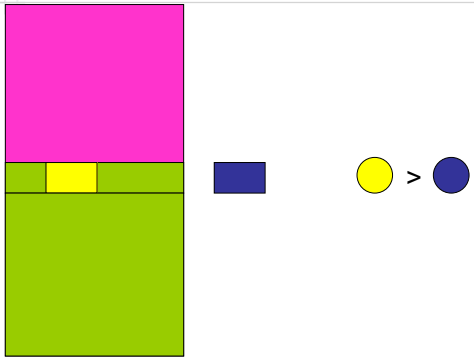
Logaritmikus (bináris) keresés



Logaritmikus (bináris) keresés



Logaritmikus (bináris) keresés



Keresések

Logaritmikus keresés rendezett tömbben

A keresés várható lépésszáma
(ha a tömb mérete N)

?

Keresések

Logaritmikus keresés rendezett tömbben

A keresés várható lépésszáma
(ha a tömb mérete N)

$\log_2 N$

Olyan kulcs érték, amely nincs tárolva a tömbben, sokkal több létezik, mint olyan, amely tárolva van.
A keresési tartomány minden egyes lépésben feleződik!

Keresések

Logaritmikus keresés rendezett tömbben

```
int keres(tombelem*t, int n,kulcs_tipus k)
{
  int i=0,j;
  if(t[--n].kulcs<k)return n+1;
  while(i<n){
    if(t[j=(i+n)/2].kulcs==k)return j;
    if(t[j].kulcs>k)n=j-1; else i=j+1;
  }
  return t[i].kulcs>=k?i:i+1;
}
```