Programozás alapjai
C nyelv
9. gyakorlat

Szeberényi Imre
BME IIT
<szebi@iit.bme.hu>

Rekurzió

- A feladat algoritmusa eleve rekurzív formában adott (ld: n!).
- A valójában nem rekurzív de valami hasznot húzunk a rekurzióból, pl. sorrend fordítás (ld: számkiíró).

Rekurzív algoritmus

- Megkeressük azt a legegyszerűbb esetet amiben a megoldás már magától értetődik.
- Megkeressük, hogy hogyan vezethető vissza ismételt egyszerűsítésekkel a legegyszerűbb esetre a feladat.

Rekurzív algoritmus fajtái.

- Hogyan hívja önmagát
  - közvetlen (a hívja a-t)
  - közvetett (a hívja b-t és b hívja a-t)
- Hányszor, hány helyen hívja magát
  - egyszerű
  - többszörös

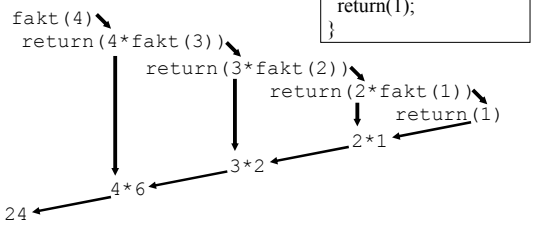
Példa: n!

- Rekurzívan adott az algoritmus. (Ennek ellenére nem a rekurzió a leghatékonyabb megvalósítás.)
- n! = n * (n-1)!, ha n > 1, egyébként 1.

```
long fakt(int n)
{
  if (n > 1)
    return(n*fakt(n-1));
  return(1);
}
```


Példa: n! (2)

```
long fakt(int n)
{
 if (n > 1)
    return(n*fakt(n-1));
 return(1);
}
```



Példa: számkiíró

Írjunk ki 3-as számrendszerben:

$n = a_n*3^n + a_{n-1}*3^{n-1} + ... + a_1*3^1 + a_0$

- Ha elosztjuk 3-mal akkor a maradék adja az utolsó jegyet.
- Gond: ezt a jegyet kellene utoljára kiírni.
  - lehetne tárolni egy tömbben, de ....
- Megoldás: a rekurzív hívás után írunk ki (a rekurzív hívások során tárolódnak a már kiszámított jegyek).

Példa: számkiíró (2)

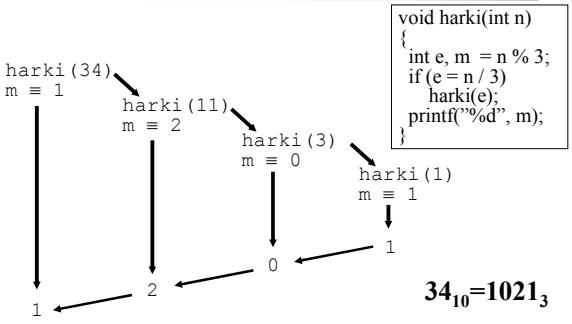
```
void harki(int n)
{
  int e, m;

  m = n % 3;
  if (e = n / 3)
    harki(e);
  printf("%d", m);
}
```

maradék

egészrész

csak visszatérés után írunk ki

Példa: számkiíró (3)

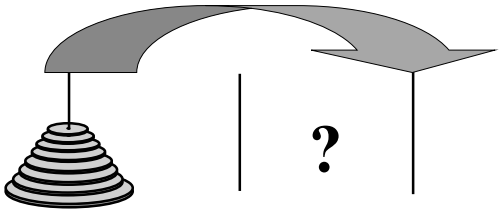



```
void harki(int n)
{
  int e, m  = n % 3;
  if (e = n / 3)
     harki(e);
  printf("%d", m);
}
```

$34_{10}=1021_3$

Példa: Hanoi tornyai legenda

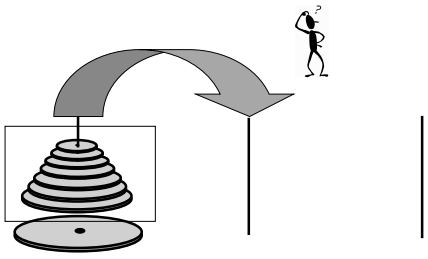
Feladat:



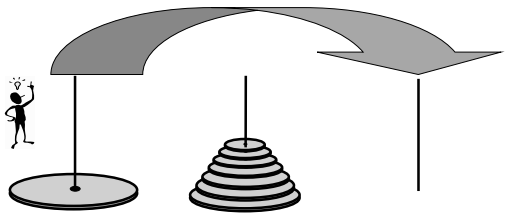
Algoritmus a legenda szerint

- „Átviszem a felső 99 korongot az ezüstrúdra, majd a 100. korongot átrakom az aranyrúdra. Ezután átviszem az ezüstrúdon levő korongokat az aranyrúdra.”
- „Elég öreg vagyok, ezért inkább a tanítványomra bízom a 99 korong átrakását. Elegendő nekem a 100. korong mozgatása.”

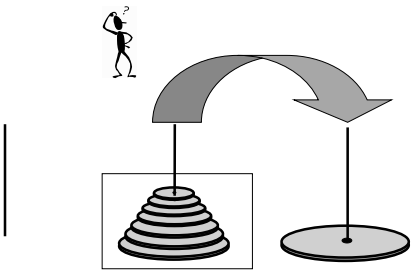
Hanoi tornyai folyt.



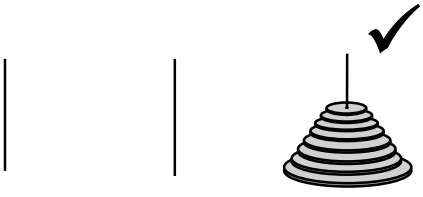
Hanoi tornyai folyt.



Hanoi tornyai folyt.



Hanoi tornyai folyt.



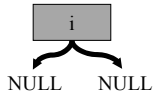
Hanoi tornyai folyt.

```
void Hanoi(int n, char forras, char cel, char seged)
{
  if (n > 0) {
     Hanoi(n-1, forras, seged, cel);
     printf("%d. korongot %c -> %c\n", n, forras, cel);
     Hanoi(n-1, seged, cel, forras);
  }
}
main()
{
   Hanoi(4, 'R', 'A', 'E');
   Hanoi(6, 'R', 'A', 'E');
}
```


Rekurzió összefoglalása

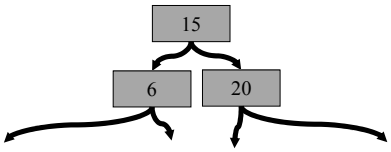
- A rekurzív algoritmusok sokszor kézenfekvőnek tűnnek, de nem biztos, hogy rekurzívan kapjuk a leghatékonyabb megoldást.
- Rekurzív algoritmus helyességét sokszor egyszerűbb belátni.
- A legtöbb rekurzió ciklussá alakítható.
- Rekurzív adatszerkezetek

Fa

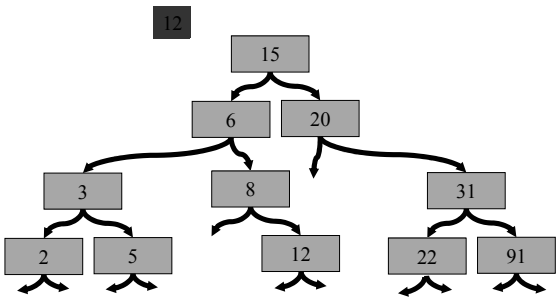



```
typedef struct fa_str {
   int i;
   struct fa_str *bal;
   struct fa_str *jobb;
} fa_elem, *fa_poi;
```

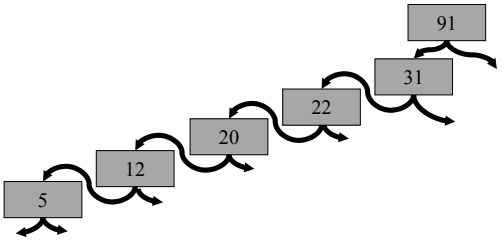
Fa



Fa



Fa (elfajuló)



Feladat 1

- Olvassunk be a „binfa.txt” állományból file végéig egész számokat. Tároljuk az adatokat bináris fában!
- Keressünk meg egy adott elemet, és írjuk ki, hogy hányszor fordult elő!
- Írjuk ki nagyság szerint rendezve az adatokat és azok előfordulási számát!

Vázlat (adatszerkezet)

```
typedef  struct fa_str {
   int i;                  /* értek */
   int sz;                 /* számláló */
   struct fa_str *bal;     /* bal mutató */
   struct fa_str *jobb;    /* jobb mutató */
} fa_elem, *fa_poi;
```


Vázlat (algoritmus)

```
gyoker = NULL
fp = állomány_megnyitása()
while folvas(fp, i)
   gyoker = faepit(gyoker, i)
folvas(stdin, i)
p = keres(gyoker, i)
if (p != NULL) elem_kiirása
else nem_találtuk_meg
fakir(gyoker)
```

Alprogram spec. - faepit

fa_poi faepit(fa_poi p, int i);

- A paraméterként kapott bináris fába felveszi az integer paraméterként kapott értéket. Azonos érték esetén növeli a számlálót.
- bemenet:
  - fa gyökerére mutató pointer. NULL, ha üres
  - felveendő érték
- kimenet:
  - függvényérték: fa gyökerére mutató pointer

Alprogram spec. - folvas

int folvas(FILE *fp, int *n); (≈ olvas 8. ea)

- Integer értékű függvény, beolvassa a következő egészet a paraméterként kapott állományból. A nem számjegy karaktereket eldobja.
- bemenet:
  - megnyitott állomány pointere
  - pointer a kimenő adatra
- kimenet:
  - olvasott egész (pointer paraméter)
  - függvényérték 1, ha sikerült az olvasás, egyébként 0

Alprogram spec. - keres

fa_poi keres(fa_poi p, int i);

- Függvény, amely a paraméterként kapott bináris fában megkeres egy elemet.
- bemenet:
  - fa gyökerére mutató pointer. NULL, ha üres
  - keresett érték
- kimenet:
  - függvényérték: megtalált elem pointere NULL, ha nincs.

Alprogram spec. - fakir

void fakir(fa_poi p);

- A paraméterként kapott fa elemeit nagyság szerint kiírja.
- bemenet:
  - fa gyökerére mutató pointer. NULL, ha üres
- kimenet: standard output

Algoritmus - faepit

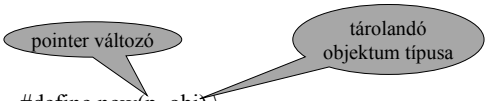
```
if a_fa_üres then
    új_elemet_veszünk_fel
else érték_azonos then
    növeljük_a_számlálót
else if kisebb_elemet_kell_felvenni then
    bal_részfát_építjük
else
    jobb_részfát_építjük
```


Algoritmus - faepit

- Az építés algoritmusa rekurzív, ami nem meglepő, hiszen maga az adatszerkezet is rekurzív.
  A fa és annak minden részfája
  - vagy üres, vagy
  - egy gyökérelemből és annak bal és jobb oldali részfájából áll.
- Nem feltétlenül rekurzív, de sokkal egyszerűbb.

Implementáció - faepit

- A tömörebb írásmód miatt vezessük be a new makrót, ami megfelelő méretű helyet foglal, és hibát is kezel (visszatér NULL-lal):




```
#define new(p, obj) \
   if ((p = malloc(sizeof(obj))) == NULL) return(NULL)
```


Implementáció - faepit

```
fa_poi faepit(fa_poi p, int i)
{
   if (p == NULL) {
     new(p, fa_elem); p->bal = p->jobb = NULL;
     p->i = i;  p->sz = 1;
   } else if (p->i == i)
     p->sz++;
   else if (p->i > i)
     p->bal = faepit(p->bal, i);
   else
     p->jobb = faepit(p->jobb, i);
  return(p);
}
```

p csak itt változik

bal részfán tovább

a változást visszaírjuk (!)

Algoritmus - keres

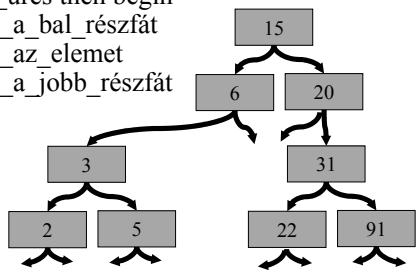
```
if a_fa_üres then
     nincs_meg_az_elem
else érték_azonos then
     megtaláltuk
else if a_keresett_elem_kisebb then
     bal_részfában_keresünk
else
     jobb_részfában_keresünk
```

Implementáció - keres

```
fa_poi keres(fa_poi p, int i)
{
  if (p == NULL)
    return(NULL);
  else if (p->i == i)
    return(p);
  else if (p->i > i)
    return(keres(p->bal, i));
  else
    return(keres(p->jobb, i));
}
```


Algoritmus - fakir

```
if fa_nem_üres then begin
    kiírjuk_a_bal_részfát
    kiírjuk_az_elemet
    kiírjuk_a_jobb_részfát
end
```



Implementáció - fakir

```
void fakir(fa_poi p)
{
  if (p != NULL) {
      fakir(p->bal);          /* bal részfa */
      printf("%5d%6d\n", p->i, p->sz);
      fakir(p->jobb);         /* jobb részfa */
  }
}
```

Implementáció - olvas

```
int olvas(FILE *fp, int *i)
{
  int r ;

  while ((r = fscanf(fp, ”%d”, i)) == 0)
    fscanf(fp, ”%*c”);
  return(r != EOF);
}
```


Implementáció - program

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

typedef  struct fa_str {
    int i;                  /* értek */
    int sz;                 /* számláló */
    struct fa_str *bal;     /* bal mutató */
    struct fa_str *jobb;    /* jobb mutató */
} fa_elem, *fa_poi;

… alprogramok …
```


Implementáció - program

```
main ()
{
  fa_poi gyoker = NULL, p; FILE *fp; int i;
  if ((fp = fopen(”binfa.txt”, ”r”)) == NULL)
    printf(”A binfa.txt nem olvasható\n”), exit(-1);
  while (olvas(fp, &i))
    gyoker = faepit(gyoker, i);
  olvas(stdin, &i);
  p = keres(gyoker, i);
  if (p !=  NULL)  printf(”%d:%10d\n”, p->i, p->sz);
  else printf(”Sajnos nincs meg!\n”);
  fakir(gyoker);
}
```

Feladat 2

- Keressünk a fában nem rekurzív algoritmussal! (A legtöbb esetben a rekurzió ciklussá alakítható.)
- „Rajzoljuk” ki a fát!

Algoritmus - keres

```
if a_fa_üres then
    nincs_meg_az_elem
else érték_azonos then
    megtaláltuk
else if a_keresett_elem_kisebb then
    bal_részfában_keresünk
else
    jobb_részfában_keresünk
```


Implementáció - keres2

```
fa_poi keres2(fa_poi p, int i)
{
   while (p != NULL) {
      if (p->i == i)
         break;
      if (p->i > i)
         p = p->bal;
      else
         p = p->jobb;
   }
   return(p);
}
```

ciklus elhagyása, mert megtaláltuk

megtaláltuk, vagy NULL

Alprogram spec. - farajz

void farajz(fa_poi p, int m);

- A paraméterként kapott bináris fát „kirajzolja” a paraméterként kapott margóval.
- bemenet:
  - fa gyökerére mutató pointer. NULL, ha üres
  - margó
- kimenet:
  - 90 fokkal elforgatott „rajz” a fa elemeiről.

Algoritmus - farajz

```
if fa_nem_üres then begin
    növeljük_a_margót
    kiírjuk_a_jobb_részfát
    kiírjuk_a_margót
    kiírjuk_az_elemet
    kiírjuk_a_bal_részfát
    csökkentjük_a_margót
end
```


Implementáció - farajz

```
void farajz(fa_poi p, int m)
{
   if (p != NULL) {
        m++;
        farajz(p->jobb, m);          /* jobb részfa */
        printf("%*s%5d\n", (m-1)*5, "", gy->i);
        farajz(p->bal, m);           /* bal részfa */
        m--;
   }
}
```

növeljük a szintet

(m-1)*5 szóköz

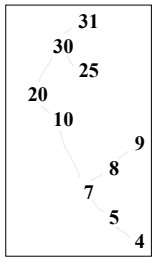
csökkentjük a szintet

Próbafuttatás eredménye

Input: (file)
20 30 25 10 7
8 5 4 31 20
30 9 7 7 7 7

Input: (standard)
7

```
7:          5
   4        1
   5        1
   7        5
   8        1
   9        1
  10        1
  20        2
  25        1
  30        2
  31        1
```



Összefoglalás

- A keresés gyorsítása érdekében a láncolt adatszerkezetet fába rendeztük.
- Fa: az adatszerkezet az egyes elemeknél elágazhat.
- Kétfelé ágazó fákat bináris fának nevezzük.
- A bináris fa és annak minden részfája vagy üres, vagy a gyökérelemből és annak bal és jobboldali részfájából áll.

Összefoglalás (2)

- Ha egy elemnek nincs utódja, akkor azt levélelemnek nevezzük.
- Ha az összes levél azonos szinten van, akkor a fa kiegyensúlyozott.
- Rekurzív adatszerkezet. Rekurzív algoritmusokkal egyszerűbb kezelni.
- A keresés az elemek 2-es alapú logaritmusával arányos műveletet igényel.