Rekurzió. Fák

A programozás alapjai I.

Hálózati Rendszerek és Szolgáltatások Tanszék
Farkas Balázs, Fiala Péter, Vitéz András, Zsóka Zoltán

2021. november 23.

Tartalom

1. fejezet

Rekurzió

Rekurzió – definíció

Sok matematikai problémát rekurzívan fogalmazunk meg

- a_n sorozat összege

$$S_n = \begin{cases} S_{n-1} + a_n & n > 0 \\ a_0 & n = 0 \end{cases}$$

- Faktoriális

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

- Fibonacci-számsorozat

$$F_n = \begin{cases} F_{n-2} + F_{n-1} & n > 1 \\ 1 & n = 1 \\ 0 & n = 0 \end{cases}$$

Rekurzió – definíció

Sok hétköznapi problémát rekurzívan fogalmazunk meg

- Felmenőm-e Dózsa György?

$$\text{Felmenőm-e?} = \begin{cases} \text{Apám/anyám felmenője-e?} \\ \text{Apám-e?} \\ \text{Anyám-e?} \end{cases}$$

- Általában

$$\text{Probléma} = \begin{cases} \text{Egyszerűbb, hasonló problém(ák)} \\ \text{Triviális eset(ek)} \end{cases}$$

Rekurzió – kitekintés

- Sokminden lehet rekurzív
  - Bizonyítás pl. teljes indukció
  - Definíció pl. Fibonacci-sorozat
  - Algoritmus pl. útvonalkeresés labirintusban
  - Adatszerkezet pl. láncolt lista, számítógép könyvtárstruktúrája
  - Geometriai konstrukció pl. fraktál
- Mi rekurzív adatszerkezetekkel és rekurzív algoritmusokkal foglalkozunk

Rekurzív algoritmusok C-ben

- Faktoriális

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

Másoljuk be C-be!

```
unsigned factorial(unsigned n)
{
  if (n > 0)
    return factorial(n-1) * n;
  else
    return 1;
}
```

- A függvény hívása

```
unsigned f = factorial(5); /* működik! */
printf("%u\n", f);
```

Kitérő

- Hogyan képzejük el?

```
unsigned f0(void) { return 1; }
unsigned f1(void) { return f0() * 1; }
unsigned f2(void) { return f1() * 2; }
unsigned f3(void) { return f2() * 3; }
unsigned f4(void) { return f3() * 4; }
unsigned f5(void) { return f4() * 5; }
...
unsigned f = f5();
```

- Egyazon függvénynek sok különböző, egyszerre létező alakja
- A paramétereik különböztetik meg őket

A rekurzió megvalósítása

Hogyan létezhet egy függvénynek egyszerre sok példánya?

```c
/*
  faktoriális rekurzív függvény
 */
unsigned factorial(unsigned n)
{
  if (n > 0)
    return factorial(n-1) * n;
  else
    return 1;
}

int main(void)
{
  ...
   factorial(4);
  ...
}
```

regiszter: 24

A rekurzió megvalósítása

- A C függvényhívási mechanizmusa eleve alkalmas a rekurzív függvényhívás megvalósítására
- A függvényt közvetve vagy direkt módon hívó függvények összes adatát (lokális változók, visszatérési cím) a veremben tároljuk
- A működés szempontjából közömbös, hogy egy függvény önmagát hívja vagy egy másik függvényt hív.
- A rekurzív hívások maximális mélysége: ami a verembe belefér

Rekurzió vagy iteráció – faktoriális

n! számítása rekurzívan – elegáns, de pazarló

```
unsigned fact_rec(unsigned n)
{
  if (n == 0)
    return 1;
  return fact_rec(n-1) * n;
}
```

link

és iterációval – „fapados”, de hatékony

```
unsigned fact_iter(unsigned n)
{
  unsigned f = 1, i;
  for (i = 2; i <= n; ++i)
    f *= i;
  return f;
}
```

link

Rekurzió vagy iteráció – Fibonacci

F_n számítása rekurzívan – elegáns, de kivárhatatlan!
A számítási idő n-nel exponenciálisan nő!

```
unsigned fib_rec(unsigned n)
{
  if (n <= 1)
    return n;
  return fib_rec(n-1) + fib_rec(n-2);
}
```

link

és iterációval – „fapados”, de hatékony

```
unsigned fib_iter(unsigned n)
{
  unsigned f[2]= {0, 1}, i;
  for (i = 2; i <= n; ++i)
    f[i%2] = f[(i-1)%2] + f[(i-2)%2];
  return f[n%2];
}
```

link

Rekurzió vagy iteráció

1. Minden rekurzív algoritmus megoldható iterációval (ciklusokkal)
   - Nincs általános módszer az átírásra, sokszor igen nehéz
2. Minden iterációval megoldható algoritmus megoldható rekurzívan
   - Könnyen automatizálható, általában nem hatékony

A problémától függ, hogy melyik módszert érdemes használni

Iterációk rekurzívan

Tömb bejárása rekurzívan (for ciklus kiváltása)

```c
void print_array(int array[], int n)
{
  if (n == 0)
    return;
  printf("%3d", array[0]);
  print_array(array+1, n-1); /* rekurzív hívás */
}
```

Lista bejárása rekurzívan

```c
void print_list(list_elem *head)
{
  if (head == NULL)
    return;
  printf("%3d", head->data);
  print_list(head->next); /* rekurzív hívás */
}
```

Csak elvileg érdekesek, ezen esetekben is az iteráció hatékonyabb

Szám kiírása adott számrendszerben

rekurzívan

```
void print_base_rec(unsigned n, unsigned base)
{
  if (n >= base)
    print_base_rec(n/base, base);
  printf("%d", n%base);
}
```

link

iterációval

```
void print_base_iter(unsigned n, unsigned base)
{
  unsigned d; /* n-nél nem nagyobb base-hatvány */
  for (d = 1; d*base <= n; d*=base);
  while (d > 0)
  {
    printf("%d", (n/d)%base);
    d /= base;
  }
}
```

link

Amikor a rekurzió már egyértelműen hasznos

Az alábbi tömb egy labirintust tárol

```c
char lab[9][9+1] = {
    "+-------+",
    "|       |",
    "+-+ ++ ++",
    "|       |",
    "| + +-+ |",
    "| | |   |",
    "+-+ +-+ |",
    "|     | |",
    "+-----+-+"
};
```

link

Járjuk be a teljes labirintust adott (x,y) kezdőpozícióból

```c
traverse(lab, 1, 1);
```

Minden lehetséges irányban elindulunk, és bejárjuk a még be nem járt labirintusrészeket.

Amikor a rekurzió már egyértelműen hasznos

A megoldás rekurzióval pofonegyszerű

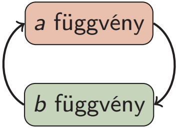
```
void traverse(char lab[][9+1], int x, int y)
{
  if (lab[x][y] != ' ')
    return;
  lab[x][y] = '.';          /* itt jártam */
  traverse(lab, x-1, y);
  traverse(lab, x+1, y);
  traverse(lab, x, y-1);
  traverse(lab, x, y+1);
}
```

link

Iterációval embert próbáló – de nem lehetetlen – feladat lenne

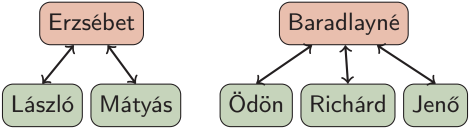
Közvetett rekurzió

Közvetett rekurzió: Függvények „körbehívják egymást”




```c
/* elődeklaráció */
void b(int); /* név, típus, paraméterek típusai */

void a(int n) {
    ...
    b(n); /* b hívható elődeklaráció miatt */
    ...
}

void b(int n) {
    ...
    a(n);
    ...
}
```

Elődeklaráció – kitekintés



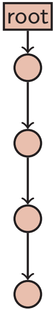
Elődeklaráció közvetve rekurzív adatszerkezetek esetén is szükséges

```
/* elődeklaráció */
struct child_s;

struct mother_s { /* anya típus */
  char name[50];
  struct child_s *children[20]; /*gyerekek ptrtömbje*/
};

struct child_s { /* gyerek típus */
  char name[50];
  struct mother_s *mother;       /*mutató anyára*/
};
```

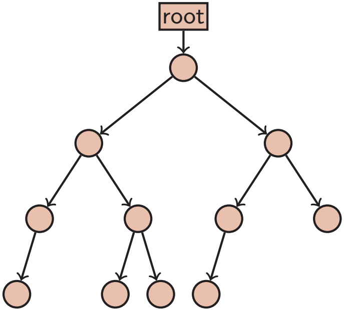
2. fejezet

Bináris fák

Fák



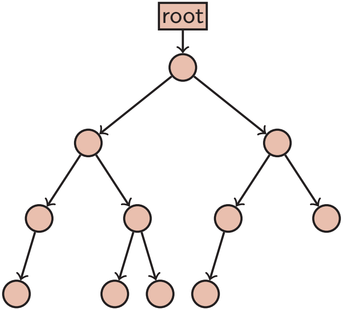
$K = 1$ (láncolt lista)



$K = 2$ (bináris fa)

- Körmentes gráf
- Minden csomópontba egy él fut be
- K-ágú fa: minden csomópontból legfeljebb K él fut ki

Bináris fák



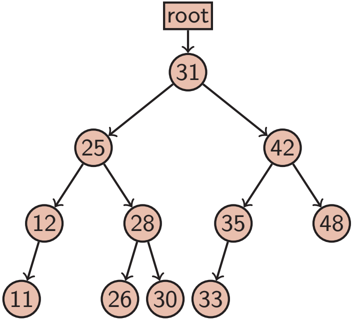
- A bináris fa adatszerkezetének deklarációja

```
typedef struct tree {
  int data;
  struct tree *left, *right;
} tree_elem, *tree_ptr;
```

link

- Szokványos egyben deklarálni a mutató típust is

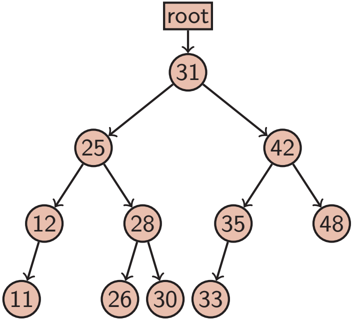
Bináris rendezőfa



31 25 28 42 12 35 11 48 30 33 26

- Elem bal oldali részfájában csak nála kisebb elemek vannak
- Elem jobb oldali részfájában csak nála nagyobb elemek vannak
- A fa struktúrája az elemek érkezési sorrendjétől függ!

Elem megkeresése a fában

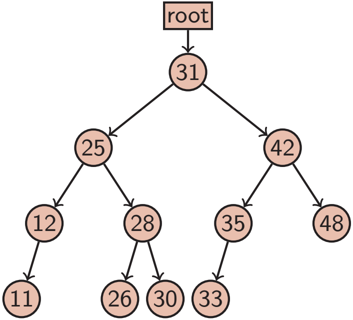



```c
tree_ptr find(tree_ptr root,
              int data)
{
  while (root != NULL &&
    data != root->data)
  {
    if (data < root->data)
      root = root->left;
    else
      root = root->right;
  }
  return root;
}
```

link

- Ez még nem rekurzió!
- d mély fában max. d lépés alatt megvan az eredmény
- Kiegyensúlyozott fában n elem közül $\approx \log_2 n$ lépés!

Bejárás – inorder




```c
void inorder(tree_ptr root)
{
  if (root == NULL)
    return;
  inorder(root->left);
  printf("%d ", root->data);
  inorder(root->right);
}
```

11 12 25 26 28 30 31 33 35 42 48

- inorder bejárás
  1. bal részfa
  2. gyökérelem
  3. jobb részfa

Ebben a sorrendben nagyság szerinti sorrendben dolgozzuk fel az elemeket

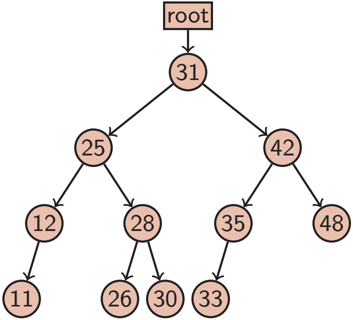
Bejárás – inorder

Másként is szervezhetjük a bejárást

```c
void inorder(tree_ptr root)
{
  if (root->left != NULL)
    inorder(root->left);
  printf("%d ", root->data);
  if (root->right != NULL)
    inorder(root->right);
}
```

Ebben az esetben a hívó függvénynek kell vizsgálnia a `root != NULL` feltételt

Bejárás – preorder




```c
void preorder(tree_ptr root)
{
  if (root == NULL)
    return;
  printf("%d ", root->data);
  preorder(root->left);
  preorder(root->right);
}
```

31 25 12 11 28 26 30 42 35 33 48

- preorder bejárás
  1. gyökérelem
  2. bal részfa
  3. jobb részfa

Ebben a sorrendben kimentve majd visszaolvasva az elemeket, a fa struktúrája visszaállítható.

Faépítés

Új elem beillesztése a fába

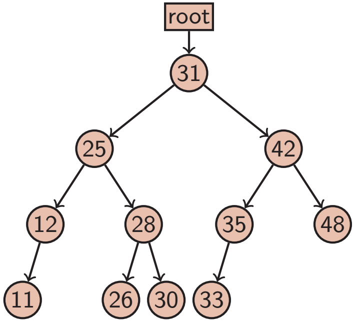
```c
tree_ptr insert(tree_ptr root, int data)
{
  if (root == NULL) {
    root = (tree_ptr)calloc(1, sizeof(tree_elem));
    root->data = data;
  }
  else if (data < root->data)
    root->left = insert(root->left, data);
  else
    root->right = insert(root->right, data);
  return root;
}
```

link

A függvény használata

```c
tree_ptr root = NULL;
root = insert(root, 2);
root = insert(root, 8);
...
```

Bejárás – posztorder




```
void postorder(tree_ptr root)
{
  if (root == NULL)
    return;
  postorder(root->left);
  postorder(root->right);
  printf("%d ", root->data);
}
```

11 12 26 30 28 25 33 35 48 42 31

- posztorder bejárás
  1. bal részfa
  2. jobb részfa
  3. gyökérelem

Ebben a sorrendben először a levélelemeket dolgozzuk fel → alkalmazás: pl. fa lebontása

Fa lebontása posztorder bejárással

```
void delete(tree_ptr root)
{
  if (root == NULL) /* üres fa leállási feltétel */
    return;
  delete(root->left);    /* postorder bejárás */
  delete(root->right);
  free(root);
}
```

link

Egy teljes programrész (memóriaszivárgás nélkül)

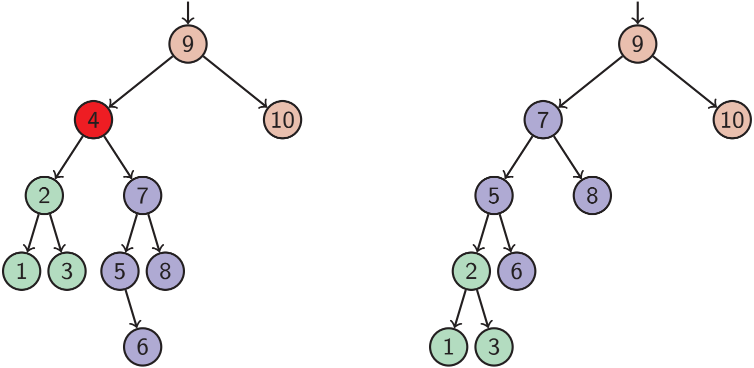
```
tree_ptr root = NULL;
root = insert(root, 2);
root = insert(root, 8);
...
delete(root);
root = NULL;
```

Egyszerű házi feladatok

...senki nem ellenőrzi ☺

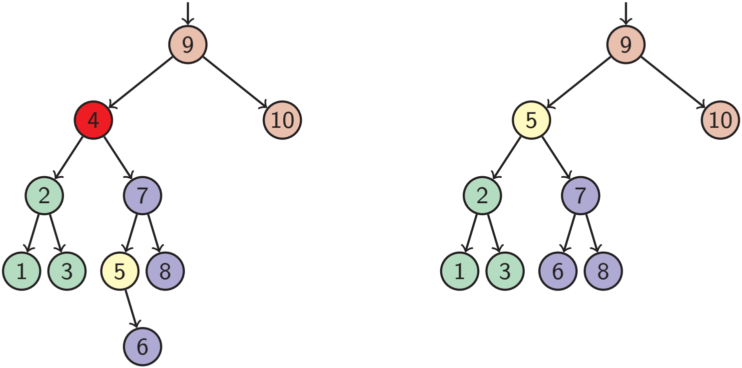
- Írj max. 10 soros rekurzív függvényt, amely
  - megállapítja, hogy milyen mély a fa
  - kiszámolja a faelemek összegét / szorzatát / átlagát
- Írj max. 10 soros iteratív függvényt, amely
  - kiszámolja a faelemek minimumát / maximumát
  - visszaadja a legkisebb / legnagyobb adatot tartalmazó faelem címét

Elem törlése bináris rendezőfából – bután



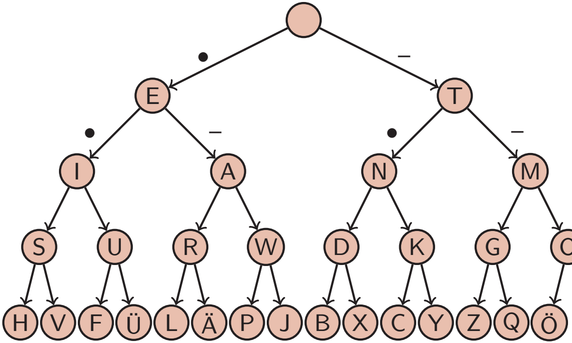
- Jobb részfát felvisszük a törlendő elem helyére
- Bal részfát beillesztjük a jobb részfa legkisebb eleme alá
- Kiegyensúlyozottság romlik!

Elem törlése bináris rendezőfából – okosan



- Jobb részfa legkisebb elemét felvisszük a törlendő helyére
- A felvitt elemnek csak jobb oldali részfája lehetett, ezt gond nélkül feljebbvisszük.

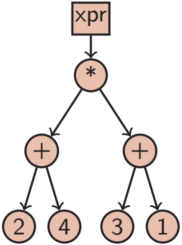
Morse dekódoló fa



SOSOS: ••• – – – ••• – – – •••

A válasz: •••• – – – • •••• – – – • •••• – – – •

Matematikai kifejezések kiértékelése



- Matematikai kifejezések tárolása fában
- Levél $\rightarrow$ konstans
- Elágazás $\rightarrow$ kétoperandusú operátor
- Példában $(2+4)*(3+1)$

```c
int eval(tree_ptr xpr)
{
  char c = xpr->data;
  if (isdigit(c))       /* leállási feltétel */
    return c - '0';
  if (c == '+')
    return eval(xpr->left) + eval(xpr->right);
  if (c == '*')
    return eval(xpr->left) * eval(xpr->right);
}
```

link

Függvény kiértékelése

Vezessük be az x változót is levélelemként:

```c
double feval(tree_ptr xpr, double x)
{
  char c = xpr->data;
  if (isdigit(c))
    return c - '0';
  if (c == 'x')
    return x;
  if (c == '+')
    return feval(xpr->left, x) + feval(xpr->right, x);
  if (c == '*')
    return feval(xpr->left, x) * feval(xpr->right, x);
}
```

link

Függvény deriváltjának kiértékelése

Deriváljuk a függvényt:

- $c' = 0$
- $x' = 1$
- $(f + g)' = f' + g'$
- $(f \cdot g)' = f' \cdot g + f \cdot g'$

```
double deval(tree_ptr xpr, double x)
{
  char c = xpr->data;
  if (isdigit(c))      /* leállási feltétel */
    return 0.0;
  if (c == 'x')        /* leállási feltétel */
    return 1.0;
  if (c == '+')
    return deval(xpr->left, x) + deval(xpr->right, x);
  if (c == '*')
    return deval(xpr->left, x) * feval(xpr->right, x) +
      feval(xpr->left, x) * deval(xpr->right, x);
}
```

link

3. fejezet

Többdimenziós tömbök

Többdimenziós tömbök

1D tömb Azonos típusú elemek a memóriában egymás mellett tárolva

2D tömb Azonos méretű és típusú 1D tömbök a memóriában egymás mellett tárolva

3D tömb Azonos méretű és típusú 2D tömbök a memóriában egymás mellett tárolva

... ...

Kétdimenziós tömbök

- 2D tömb deklarációja:

```c
char a[3][2]; /* 3 soros két oszlopos karaktertömb */
              /* 2 elemű 1D tömbök 3 elemű tömbje */
```

a[0][0]	a[0][1]
a[1][0]	a[1][1]
a[2][0]	a[2][1]

- C-ben sorfolytonos tárolás, vagyis a hátsó index fut gyorsabban

a[0]		a[1]		a[2]	
a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

- `a[0]`, `a[1]` és `a[2]` 2 elemű 1D tömbök

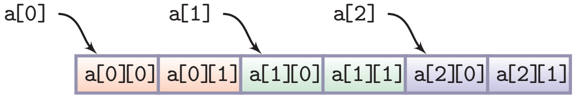
Kétdimenziós tömb átvétele soronként

- 1D tömb (sor) feltöltése adott elemmel

```c
void fill_row(char row[], size_t size, char c)
{
  size_t i;
  for (i = 0; i < size; ++i)
    row[i] = c;
}
```

- 2D tömb feltöltése soronként

```c
char a[3][2];
fill_row(a[0], 2, 'a'); /* 0. sor csupa 'a' */
fill_row(a[1], 2, 'b'); /* 1. sor csupa 'b' */
fill_row(a[2], 2, 'c'); /* 2. sor csupa 'c' */
```



Kétdimenziós tömb átvétele egyben

- átvétel 2D tömbként – csak ha az oszlopok száma ismert

```c
void print_array(char array[][2], size_t nrows)
{
  size_t row, col;
  for (row = 0; row < nrows; ++row)
  {
    for (col = 0; col < 2; ++col)
      printf("%c", array[row][col]);
    printf("\n");
  }
}
```

- A függvény használata

```c
char a[3][2];
...
print_array(a, 3);      /* 3 soros tömb kiírása */
```

Kétdimenziós tömb átvétele egyben

- 2D tömb átvétele mutatóként

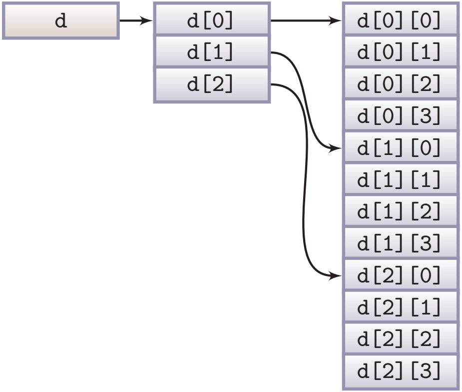
```c
void print_array(char *array, int nrows, int ncols)
{
  int row, col;
  for (row = 0; row < nrows; ++row)
  {
    for (col = 0; col < ncols; ++col)
      printf("%c", array[row*ncols+col]);
    printf("\n");
  }
}
```

- A függvény használata

```c
char a[3][2];
...
print_array((char *)a, 3, 2); /* 3 sor 2 oszlop */
```

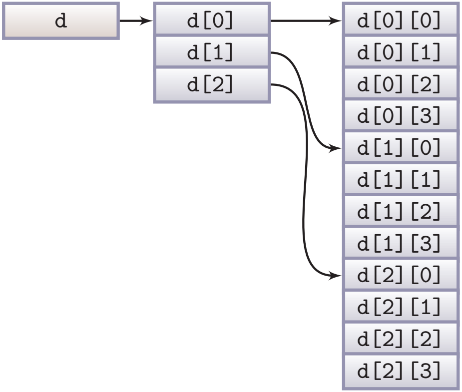
2D dinamikus tömb

Foglaljunk dinamikusan kétdimenziós tömböt, melyet a szokásos módon, `d[i][j]` indexeléssel használhatunk




```c
double **d =(double**)malloc(3*sizeof(double*));
d[0] = (double*)malloc(3*4*sizeof(double));
for (i = 1; i < 3; ++i)
  d[i] = d[i-1] + 4;
```

2D dinamikus tömb



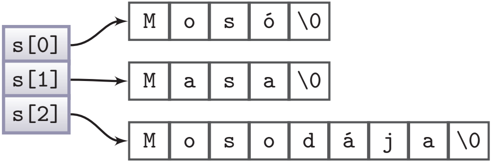
A tömb felszabadítása

```
free(d[0]);
free(d);
```

Mutatótömb

- Mutatótömb definiálása és átadása függvénynek

```
char *s[3] = {"Mosó", "Masa", "Mosodája"};
print_strings(s, 3);
```



- Mutatótömb átvétele függvénnyel

```
void print_strings(char *strings[], size_t size)
/*                 char **strings is lehet       */
{
  size_t i;
  for (i = 0; i < size; ++i)
    printf("%s\n", strings[i]);
}
```

Köszönöm a figyelmet.