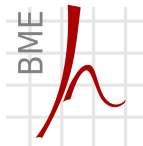


Bináris fák. Függvénymutatók

A programozás alapjai I.



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

2020. november 23.

Tartalom

1 Bináris fák

- Definíció
- Bináris rendezőfa
- Bejárások

- Törlés
- Egyéb alkalmazások

2 Függvénymutatók

- Motiváció
- Megoldás

1. fejezet

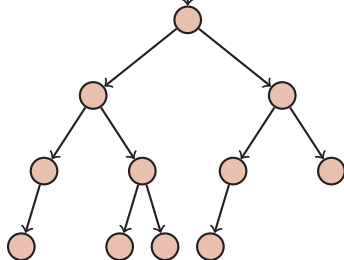
Bináris fák

Fák

root

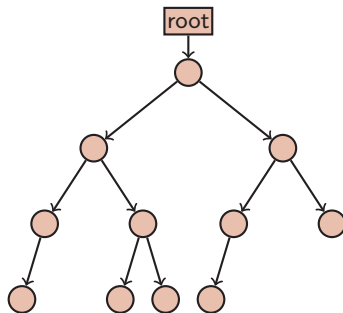
 $K = 1$ (láncolt lista)

root

 $K = 2$ (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

```
1 typedef struct tree {  
2     int data;  
3     struct tree *left, *right;  
4 } tree_elem, *tree_ptr;
```

[link](#)

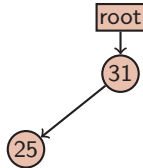
- Szokványos egyben deklarálni a mutató típust is

Bináris rendezőfa



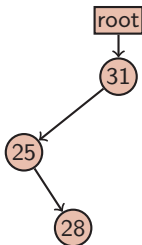
- Elem bal oldali részfájában csak nála kisebb elemek vannak
- Elem jobb oldali részfájában csak nála nagyobb elemek vannak
- A fa struktúrája az elemek érkezési sorrendjétől függ!

Bináris rendezőfa



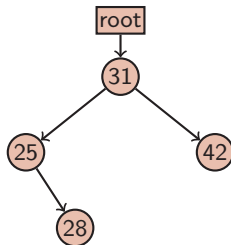
- Elem bal oldali részfájában csak nála kisebb elemek vannak
- Elem jobb oldali részfájában csak nála nagyobb elemek vannak
- A fa struktúrája az elemek érkezési sorrendjétől függ!

Bináris rendezőfa



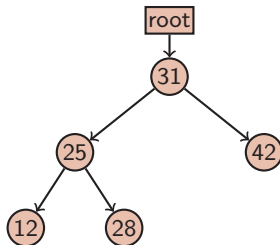
- Elem bal oldali részfájában csak nála kisebb elemek vannak
- Elem jobb oldali részfájában csak nála nagyobb elemek vannak
- A fa struktúrája az elemek érkezési sorrendjétől függ!

Bináris rendezőfa



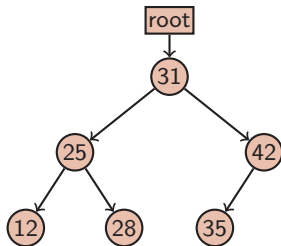
- Elem bal oldali részfájában csak nála kisebb elemek vannak
- Elem jobb oldali részfájában csak nála nagyobb elemek vannak
- A fa struktúrája az elemek érkezési sorrendjétől függ!

Bináris rendezőfa



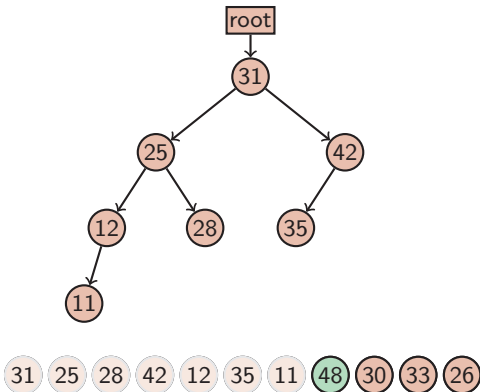
- Elem bal oldali részfájában csak nála kisebb elemek vannak
- Elem jobb oldali részfájában csak nála nagyobb elemek vannak
- A fa struktúrája az elemek érkezési sorrendjétől függ!

Bináris rendezőfa



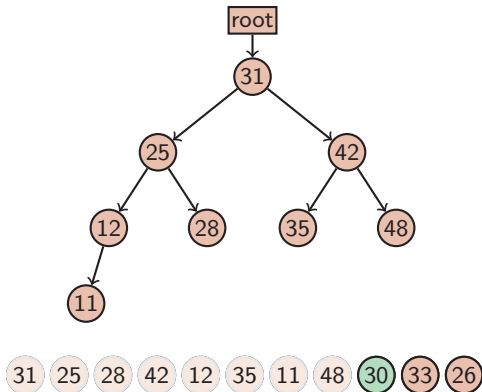
- Elem bal oldali részfájában csak nála kisebb elemek vannak
- Elem jobb oldali részfájában csak nála nagyobb elemek vannak
- A fa struktúrája az elemek érkezési sorrendjétől függ!

Bináris rendezőfa



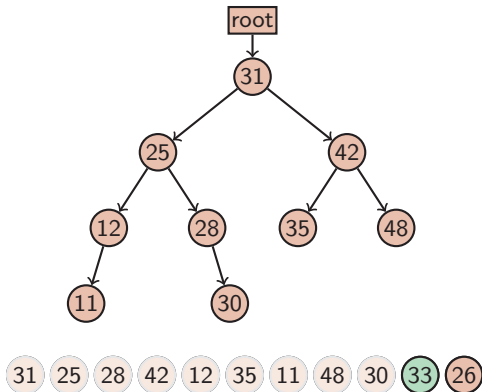
- Elem bal oldali részfájában csak nála kisebb elemek vannak
- Elem jobb oldali részfájában csak nála nagyobb elemek vannak
- A fa struktúrája az elemek érkezési sorrendjétől függ!

Bináris rendezőfa



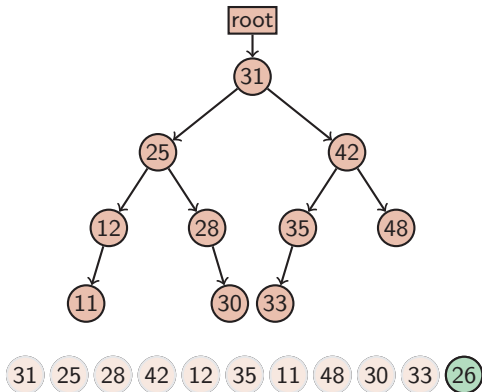
- Elem bal oldali részfájában csak nála kisebb elemek vannak
- Elem jobb oldali részfájában csak nála nagyobb elemek vannak
- A fa struktúrája az elemek érkezési sorrendjétől függ!

Bináris rendezőfa



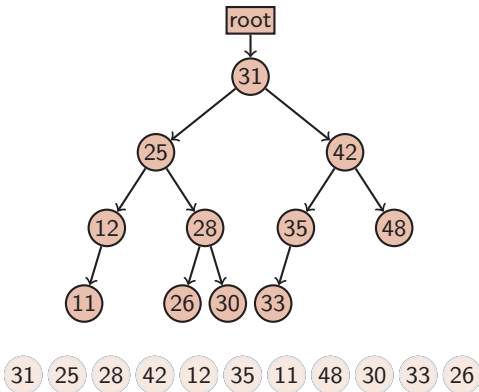
- Elem bal oldali részfájában csak nála kisebb elemek vannak
- Elem jobb oldali részfájában csak nála nagyobb elemek vannak
- A fa struktúrája az elemek érkezési sorrendjétől függ!

Bináris rendezőfa



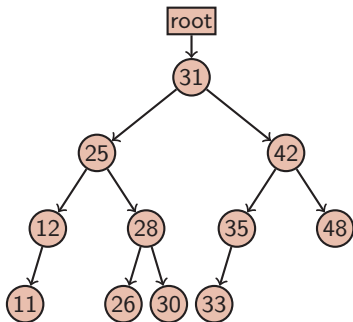
- Elem bal oldali részfájában csak nála kisebb elemek vannak
- Elem jobb oldali részfájában csak nála nagyobb elemek vannak
- A fa struktúrája az elemek érkezési sorrendjétől függ!

Bináris rendezőfa



- 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!

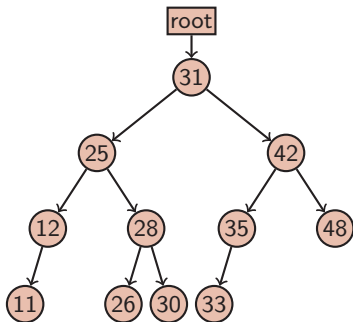
Elem megkeresése a fában



```
1 tree_ptr find(tree_ptr root,
2               int data)
3 {
4     while (root != NULL &&
5           data != root->data)
6     {
7         if (data < root->data)
8             root = root->left;
9         else
10            root = root->right;
11    }
12    return root;
13 }
```

[link](#)

Elem megkeresése a fában

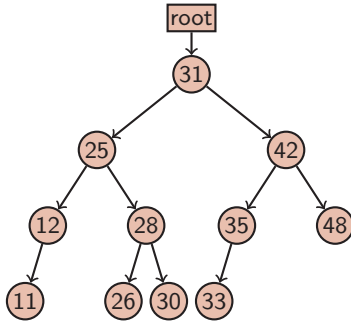


```
1 tree_ptr find(tree_ptr root,
2               int data)
3 {
4     while (root != NULL &&
5           data != root->data)
6     {
7         if (data < root->data)
8             root = root->left;
9         else
10            root = root->right;
11    }
12    return root;
13 }
```

[link](#)

■ Ez még nem rekurzió!

Elem megkeresése a fában

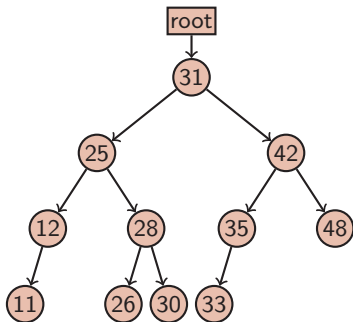


```
1 tree_ptr find(tree_ptr root,  
2               int data)  
3 {  
4     while (root != NULL &&  
5           data != root->data)  
6     {  
7         if (data < root->data)  
8             root = root->left;  
9         else  
10            root = root->right;  
11    }  
12    return root;  
13 }
```

[link](#)

- Ez még nem rekurzió!
- d mély fában max. d lépés alatt megvan az eredmény

Elem megkeresése a fában

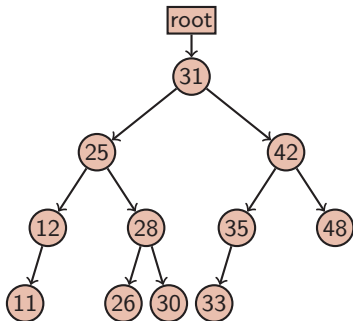


```
1 tree_ptr find(tree_ptr root,  
2               int data)  
3 {  
4     while (root != NULL &&  
5           data != root->data)  
6     {  
7         if (data < root->data)  
8             root = root->left;  
9         else  
10            root = root->right;  
11    }  
12    return root;  
13 }
```

[link](#)

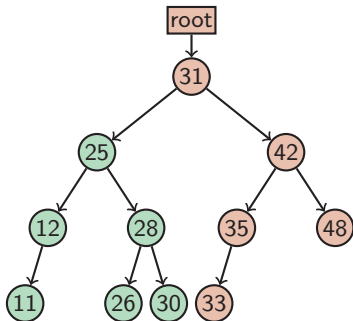
- Ez még nem rekurzió!
- d mély fában max. d lépés alatt megvan az eredmény
- Kiegyensúlyozott fában n elem közül $\approx \log_2 n$ lépés!

Bejárás – inorder



```
1 void inorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     inorder(root->left);
6     printf("%d ", root->data);
7     inorder(root->right);
8 }
```

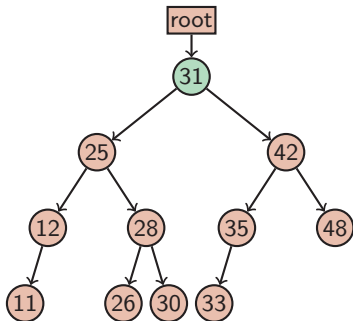
Bejárás – inorder



```
1 void inorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     inorder(root->left);
6     printf("%d ", root->data);
7     inorder(root->right);
8 }
```

11 12 25 26 28 30

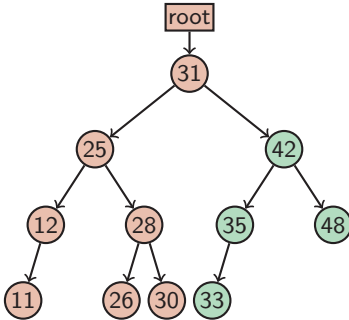
Bejárás – inorder



```
1 void inorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     inorder(root->left);
6     printf("%d ", root->data);
7     inorder(root->right);
8 }
```

11 12 25 26 28 30 31

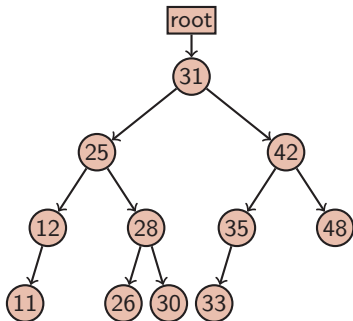
Bejárás – inorder



```
1 void inorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     inorder(root->left);
6     printf("%d ", root->data);
7     inorder(root->right);
8 }
```

11 12 25 26 28 30 31 33 35 42 48

Bejárás – inorder



```
1 void inorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     inorder(root->left);
6     printf("%d ", root->data);
7     inorder(root->right);
8 }
```

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

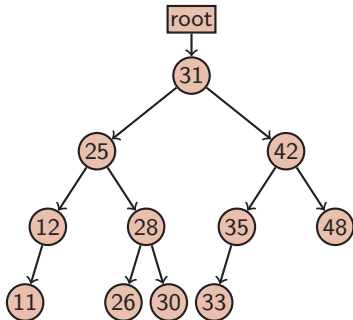
Bejárás – inorder

Másként is szervezhetjük a bejárást

```
1 void inorder(tree_ptr root)
2 {
3     if (root->left != NULL)
4         inorder(root->left);
5     printf("%d ", root->data);
6     if (root->right != NULL)
7         inorder(root->right);
8 }
```

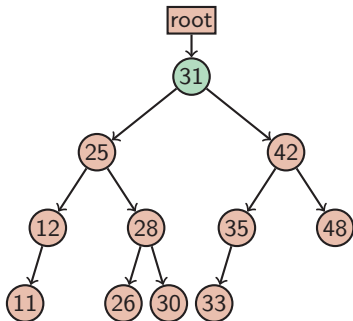
Ebben az esetben a hívó függvénynek kell vizsgálnia a `root != NULL` feltételt

Bejárás – preorder



```
1 void preorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     printf("%d ", root->data);
6     preorder(root->left);
7     preorder(root->right);
8 }
```

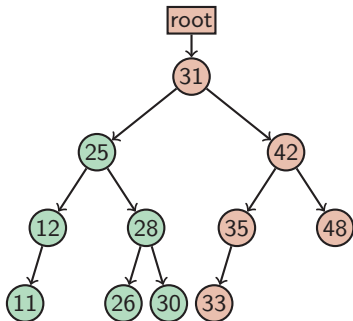
Bejárás – preorder



```
1 void preorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     printf("%d ", root->data);
6     preorder(root->left);
7     preorder(root->right);
8 }
```

31

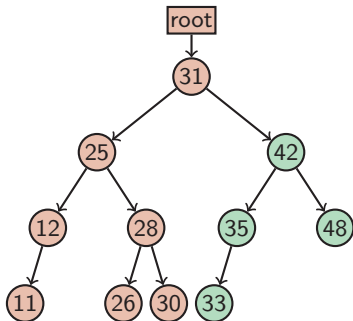
Bejárás – preorder



```
1 void preorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     printf("%d ", root->data);
6     preorder(root->left);
7     preorder(root->right);
8 }
```

31 25 12 11 28 26 30

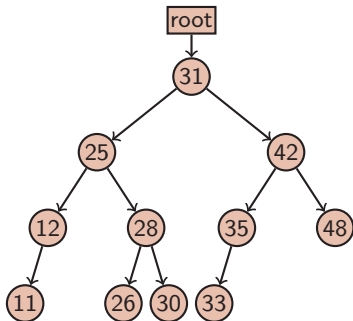
Bejárás – preorder



```
1 void preorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     printf("%d ", root->data);
6     preorder(root->left);
7     preorder(root->right);
8 }
```

31 25 12 11 28 26 30 42 35 33 48

Bejárás – preorder



```

1 void preorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     printf("%d ", root->data);
6     preorder(root->left);
7     preorder(root->right);
8 }

```

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

```
1 tree_ptr insert(tree_ptr root, int data)
2 {
3     if (root == NULL) {
4         root = (tree_ptr)calloc(1, sizeof(tree_elem));
5         root->data = data;
6     }
7     else if (data < root->data)
8         root->left = insert(root->left, data);
9     else
10        root->right = insert(root->right, data);
11    return root;
12 }
```

[link](#)

Faépítés

Új elem beillesztése a fába

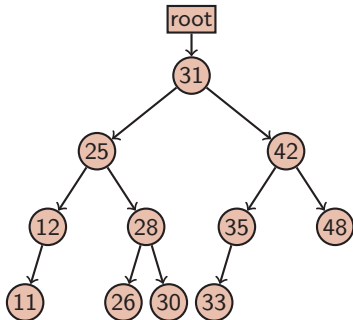
```
1 tree_ptr insert(tree_ptr root, int data)
2 {
3     if (root == NULL) {
4         root = (tree_ptr)calloc(1, sizeof(tree_elem));
5         root->data = data;
6     }
7     else if (data < root->data)
8         root->left = insert(root->left, data);
9     else
10        root->right = insert(root->right, data);
11    return root;
12 }
```

[link](#)

A függvény használata

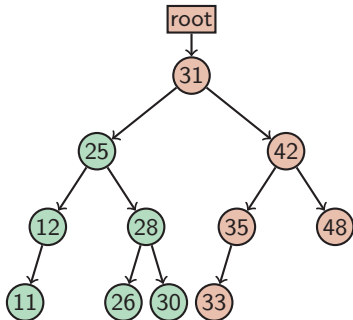
```
1 tree_ptr root = NULL;
2 root = insert(root, 2);
3 root = insert(root, 8);
4 ...
```

Bejárás – posztorder



```
1 void postorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     postorder(root->left);
6     postorder(root->right);
7     printf("%d ", root->data);
8 }
```

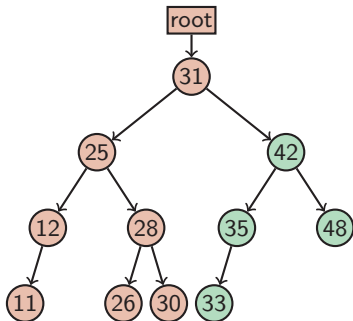
Bejárás – posztorder



```
1 void postorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     postorder(root->left);
6     postorder(root->right);
7     printf("%d ", root->data);
8 }
```

11 12 26 30 28 25

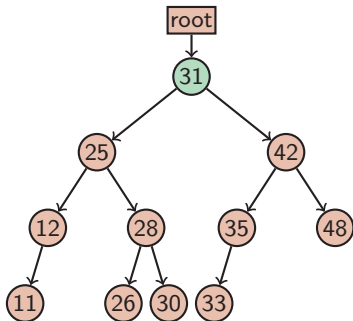
Bejárás – posztorder



```
1 void postorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     postorder(root->left);
6     postorder(root->right);
7     printf("%d ", root->data);
8 }
```

11 12 26 30 28 25 33 35 48 42

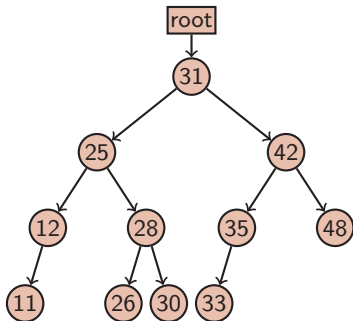
Bejárás – posztorder



```
1 void postorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     postorder(root->left);
6     postorder(root->right);
7     printf("%d ", root->data);
8 }
```

11 12 26 30 28 25 33 35 48 42 31

Bejárás – posztorder



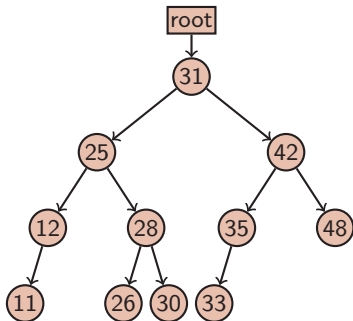
```
1 void postorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     postorder(root->left);
6     postorder(root->right);
7     printf("%d ", root->data);
8 }
```

11 12 26 30 28 25 33 35 48 42 31

■ posztorder bejárás

- 1 bal részfa
- 2 jobb részfa
- 3 gyökérelem

Bejárás – posztorder



```

1 void postorder(tree_ptr root)
2 {
3     if (root == NULL)
4         return;
5     postorder(root->left);
6     postorder(root->right);
7     printf("%d ", root->data);
8 }
  
```

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

```
1 void delete(tree_ptr root)
2 {
3     if (root == NULL) /* üres fa leállási feltétel */
4         return;
5     delete(root->left);    /* postorder bejárás */
6     delete(root->right);
7     free(root);
8 }
```

[link](#)

Fa lebontása posztorder bejárással

```
1 void delete(tree_ptr root)
2 {
3     if (root == NULL) /* üres fa leállási feltétel */
4         return;
5     delete(root->left);    /* postorder bejárás */
6     delete(root->right);
7     free(root);
8 }
```

[link](#)

Egy teljes programrész (memóriaszivárgás nélkül)

```
1 tree_ptr root = NULL;
2 root = insert(root, 2);
3 root = insert(root, 8);
4 ...
5 delete(root);
6 root = NULL;
```

Egyszerű házi feladatok

...senki nem ellenőrzi 😊

- Írj max. 10 soros rekurzív függvényt, amely

Egyszerű házi feladatok

...senki nem ellenőrzi 😊

- Írj max. 10 soros rekurzív függvényt, amely
 - megállapítja, hogy milyen mély a fa

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

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

Egyszerű házi feladatok

...senki nem ellenőrzi 😊

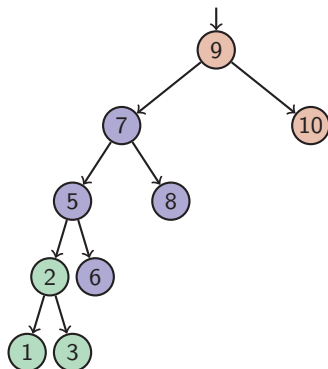
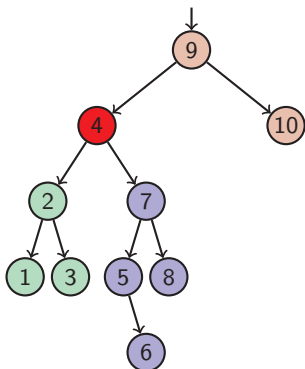
- Írj max. 10 soros rekurzív függvényt, amely
 - megállapítja, hogy milyen mély a fa
 - kiszámolja a faelemek összegét / szorzatát / átlagát
- Írj max. 10 soros iteratív függvényt, amely
 - kiszámolja a faelemek minimumát / maximumát

Egyszerű házi feladatok

...senki nem ellenőrzi 😊

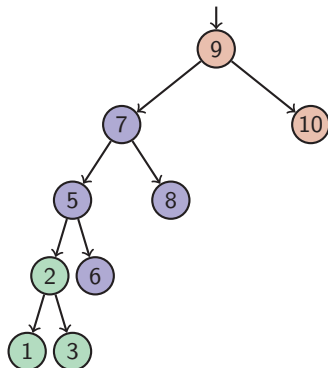
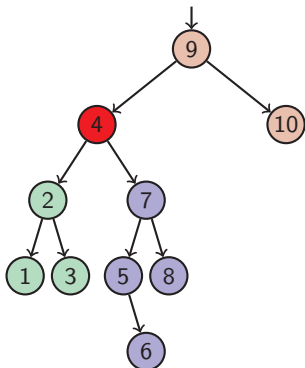
- Írj max. 10 soros rekurzív függvényt, amely
 - megállapítja, hogy milyen mély a fa
 - kiszámolja a faelemek összegét / szorzatát / átlagát
- Írj max. 10 soros iteratív függvényt, amely
 - kiszámolja a faelemek minimumát / maximumát
 - visszaadja a legkisebb / legnagyobb adatot tartalmazó faelem címét

Elem törlése bináris rendezőfából – bután



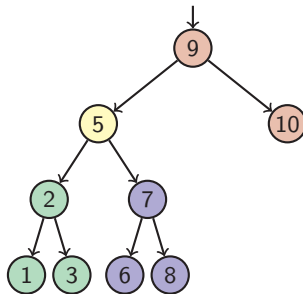
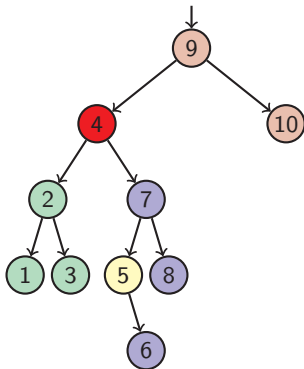
- Jobb részfat felvisszük a törölendő elem helyére
- Bal részfat beillesztjük a jobb részfa legkisebb eleme alá

Elem törlése bináris rendezőfából – bután



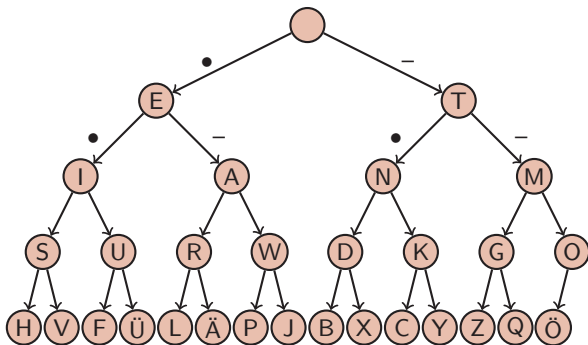
- Jobb részfat felvisszük a törlendő elem helyére
- Bal részfat beillesztjük a jobb részfa legkisebb eleme alá
- Kiegyensúlyozottság romlik!

Elem törlése bináris rendezőfából – okosan



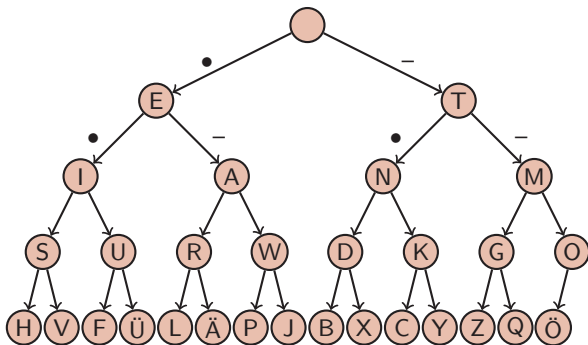
- Jobb részfa legkisebb elemét felvisszük a törlendő helyére
- A felvitt elemnek csak jobb oldali részfája lehetett, ezt gond nélkül feljebbvisszük.

Morse dekódoló fa



SOSOS: ••• - - - ••• - - - •••

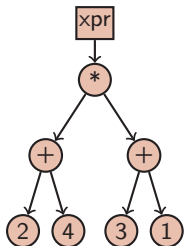
Morse dekódoló fa



SOSOS: ●●● - - - ●●● - - - ●●●

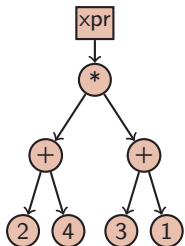
A válasz: ●●●● - - - ● ●●●● - - - ● ●●●● - - - ●

Matematikai kifejezések kiértékelése



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

Matematikai kifejezések kiértékelése



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

```
1 int eval(tree_ptr xpr)
2 {
3     char c = xpr->data;
4     if (isdigit(c)) /* leállási feltétel */
5         return c - '0';
6     if (c == '+')
7         return eval(xpr->left) + eval(xpr->right);
8     if (c == '*')
9         return eval(xpr->left) * eval(xpr->right);
10 }
```

[link](#)

Függvény kiértékelése

Vezessük be az x változót is levélelemként:

```
1 double feval(tree_ptr xpr, double x)
2 {
3     char c = xpr->data;
4     if (isdigit(c))
5         return c - '0';
6     if (c == 'x')
7         return x;
8     if (c == '+')
9         return feval(xpr->left, x) + feval(xpr->right, x);
10    if (c == '*')
11        return feval(xpr->left, x) * feval(xpr->right, x);
12 }
```

[link](#)

Függvény deriváltjának kiértékelése

Deriváljuk a függvényt:

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

```
1 double deval(tree_ptr xpr, double x)
2 {
3     char c = xpr->data;
4     if (isdigit(c))          /* leállási feltétel */
5         return 0.0;
6     if (c == 'x')            /* leállási feltétel */
7         return 1.0;
8     if (c == '+')
9         return deval(xpr->left, x) + deval(xpr->right, x);
10    if (c == '*')
11        return deval(xpr->left, x) * feval(xpr->right, x) +
12            feval(xpr->left, x) * deval(xpr->right, x);
13 }
```

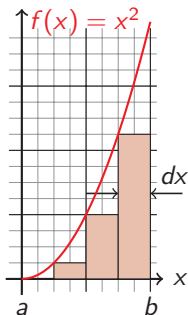
[link](#)

2. fejezet

Függvénymutatók

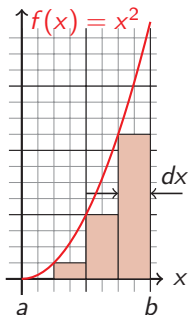
Bemelegítő feladat

Írjunk függvényt, amely az $f(x) = x^2$ függvény görbéje alatti területet közelíti az $[a, b]$ intervallumon, n darab téglalappal



Bemelegítő feladat

Írjunk függvényt, amely az $f(x) = x^2$ függvény görbéje alatti területet közelíti az $[a, b]$ intervallumon, n darab téglalappal

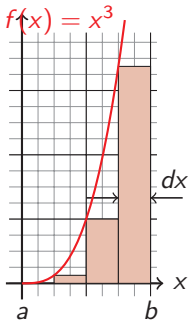


```
1 double xsquare(double x)
2 {
3     return x*x;
4 }
```

```
1 double integral_x2(double a, double b,
2                     unsigned n)
3 {
4     double dx = (b-a)/n, sum=0.0, x;
5     unsigned i;
6
7     for (x=a, i=0; i<n; x+=dx, i++)
8         sum += xsquare(x) * dx;
9
10    return sum;
11 }
```

Bemelegítő feladat

Írjunk függvényt, amely az $f(x) = x^3$ függvény görbéje alatti területet közelíti az $[a, b]$ intervallumon, n darab téglalappal



```
1 double xcube(double x)
2 {
3     return x*x*x;
4 }
```

```
1 double integral_x3(double a, double b,
2                     unsigned n)
3 {
4     double dx = (b-a)/n, sum=0.0, x;
5     unsigned i;
6
7     for (x=a, i=0; i<n; x+=dx, i++)
8         sum += xcube(x) * dx;
9
10    return sum;
11 }
```

Motiváció

Határ a csillagos ég...

```
1 double integral_x2(double a, double b, unsigned n);  
2 double integral_x3(double a, double b, unsigned n);  
3 double integral_sin(double a, double b, unsigned n);  
4 double integral_sqrt(double a, double b, unsigned n);
```

- ha módosítjuk a számítást, minden ilyen függvényt át kell írunk

Motiváció

Határ a csillagos ég...

```
1 double integral_x2(double a, double b, unsigned n);  
2 double integral_x3(double a, double b, unsigned n);  
3 double integral_sin(double a, double b, unsigned n);  
4 double integral_sqrt(double a, double b, unsigned n);
```

- ha módosítjuk a számítást, minden ilyen függvényt át kell írunk
- Helyette jó lenne a görbét megadó függvényt is átadnunk a számolónak

Függvénymutató

Függvénymutató

- a függvény is a memóriában van, ezért címe is képezhető → függvényre mutató pointernek is van értelme

```
1 double (*fx)(double);  
2 fx = sqrt;  
3 printf("%f\n", fx(5.0)); /* sqrt(5.0) */  
4 fx = sin; /* <math.h> */  
5 printf("%f\n", fx(1.57)); /* sin(1.57) */
```

Függvénymutató

Függvénymutató

- a függvény is a memóriában van, ezért címe is képezhető → függvényre mutató pointernek is van értelme
- értékadásnál egy megfelelő típusú függvény azonosítóját kell megadnunk
típus: paraméterek típusai + visszatérési érték típusa

```
1 double (*fx)(double);  
2 fx = sqrt;  
3 printf("%f\n", fx(5.0)); /* sqrt(5.0) */  
4 fx = sin; /* <math.h> */  
5 printf("%f\n", fx(1.57)); /* sin(1.57) */
```


Függvénymutató

Függvénymutató

- a függvény is a memóriában van, ezért címe is képezhető → függvényre mutató pointernek is van értelme
- értékadásnál egy megfelelő típusú függvény azonosítóját kell megadnunk
típus: paraméterek típusai + visszatérési érték típusa
- a mutatott függvényt a mutatón keresztül meghívhatjuk

```
1 double (*fx)(double);  
2 fx = sqrt;  
3 printf("%f\n", fx(5.0)); /* sqrt(5.0) */  
4 fx = sin; /* <math.h> */  
5 printf("%f\n", fx(1.57)); /* sin(1.57) */
```

Függvénymutató

Függvénymutató

- a függvény is a memóriában van, ezért címe is képezhető → függvényre mutató pointernek is van értelme
- értékadásnál egy megfelelő típusú függvény azonosítóját kell megadnunk
típus: paraméterek típusai + visszatérési érték típusa
- a mutatott függvényt a mutatón keresztül meghívhatjuk

```
1 double (*fx)(double);  
2 fx = sqrt;  
3 printf("%f\n", fx(5.0)); /* sqrt(5.0) */  
4 fx = sin; /* <math.h> */  
5 printf("%f\n", fx(1.57)); /* sin(1.57) */
```

Görbe alatti - bármire

```
13 double integral(double (*fx)(double),
14                 double a, double b, unsigned n)
15 {
16     double dx=(b-a)/n, sum=0.0, x;
17     unsigned i;
18
19     for(x=a, i=0; i<n; x+=dx, i++)
20         sum += fx(x) * dx;
21
22     return sum;
23 }
24
25 int main(void)
26 {
27     printf("%f\n", integral(xsquare, 1.0, 5.0, 100));
28     printf("%f\n", integral(xcube, 1.0, 5.0, 100));
29     return 0;
30 }
```

[link](#)

Függvény mint függvény paramétere

```
1 double integral(/* double (*fx)(double) */  
2                 double fx(double),  
3                 double a, double b, unsigned n) {  
4     ...  
5     sum += fx(x) * dx;  
6     ...  
7 }  
8  
9 printf("%f\n", integral(xsquare, 1.0, 5.0, 100));
```

- A függvényt mutatóval adjuk át
Egyszerűsítés: a függvény fejlécét is írhatjuk
- A mutatót a függvényhívás operátorral () használjuk
- Aktuális paraméterként csak a függvény azonosítóját kell megadnunk, hasonló formában, mint a tömb átadásánál

Függvény mint függvény paramétere

```
1 double integral(/* double (*fx)(double) */  
2                 double fx(double),  
3                 double a, double b, unsigned n) {  
4     ...  
5     sum += fx(x) * dx;  
6     ...  
7 }  
8  
9 printf("%f\n", integral(xsquare, 1.0, 5.0, 100));
```

- A függvényt mutatóval adjuk át
Egyszerűsítés: a függvény fejlécét is írhatjuk
- A mutatót a függvényhívás operátorral () használjuk
- Aktuális paraméterként csak a függvény azonosítóját kell megadnunk, hasonló formában, mint a tömb átadásánál

Függvény mint függvény paramétere

```
1 double integral(/* double (*fx)(double) */  
2                 double fx(double),  
3                 double a, double b, unsigned n) {  
4     ...  
5     sum += fx(x) * dx;  
6     ...  
7 }  
8  
9 printf("%f\n", integral(xsquare, 1.0, 5.0, 100));
```

- A függvényt mutatóval adjuk át
Egyszerűsítés: a függvény fejlécét is írhatjuk
- A mutatót a függvényhívás operátorral () használjuk
- Aktuális paraméterként csak a függvény azonosítóját kell megadnunk, hasonló formában, mint a tömb átadásánál

Függvény mint függvény paramétere

```
1 double integral(/* double (*fx)(double) */  
2                 double fx(double),  
3                 double a, double b, unsigned n) {  
4     ...  
5     sum += fx(x) * dx;  
6     ...  
7 }  
8  
9 printf("%f\n", integral(xsquare, 1.0, 5.0, 100));
```

- A függvényt mutatóval adjuk át
Egyszerűsítés: a függvény fejlécét is írhatjuk
- A mutatót a függvényhívás operátorral () használjuk
- Aktuális paraméterként csak a függvény azonosítóját kell megadnunk, hasonló formában, mint a tömb átadásánál

Függvény mint függvény paramétere

```
1 double integral(/* double (*fx)(double) */  
2                 double fx(double),  
3                 double a, double b, unsigned n) {  
4     ...  
5     sum += fx(x) * dx;  
6     ...  
7 }  
8  
9 printf("%f\n", integral(xsquare, 1.0, 5.0, 100));
```

- A függvényt mutatóval adjuk át
Egyszerűsítés: a függvény fejlécét is írhatjuk
- A mutatót a függvényhívás operátorral () használjuk
- Aktuális paraméterként csak a függvény azonosítóját kell megadnunk, hasonló formában, mint a tömb átadásánál

Függvény mint függvény paramétere

```
1 double integral(/* double (*fx)(double) */  
2                 double fx(double),  
3                 double a, double b, unsigned n) {  
4     ...  
5     sum += fx(x) * dx;  
6     ...  
7 }  
8  
9 printf("%f\n", integral(xsquare, 1.0, 5.0, 100));
```

- A függvényt mutatóval adjuk át
Egyszerűsítés: a függvény fejlécét is írhatjuk
- A mutatót a függvényhívás operátorral () használjuk
- Aktuális paraméterként csak a függvény azonosítóját kell megadnunk, hasonló formában, mint a tömb átadásánál

Függvénymutatók tömbje

- Függvénytömbökből álló tömböt is képezhetünk

```
1 double (*(ftrig[2]))(double) = {sin, cos};  
2  
3 for (i = 0; i < 2; ++i)  
4     printf("%f\n", ftrig[i](3.14));
```

Függvénymutatók tömbje

- Függvénytömbökből álló tömböt is képezhetünk

```
1 double (*(ftrig[2]))(double) = {sin, cos};  
2  
3 for (i = 0; i < 2; ++i)  
4     printf("%f\n", ftrig[i](3.14));
```

- Könnyebben érthető a típus, ha `typedef`-et használunk

```
1 typedef double (*fvptr)(double);  
2 fvptr fhyp[] = {sinh, cosh};
```

Függvénymutatók tömbje

- Függvénytömbökből álló tömböt is képezhetünk

```
1 double (*(ftrig[2]))(double) = {sin, cos};  
2  
3 for (i = 0; i < 2; ++i)  
4     printf("%f\n", ftrig[i](3.14));
```

- Könnyebben érthető a típus, ha `typedef`-et használunk

```
1 typedef double (*fvptr)(double);  
2 fvptr fhyp[] = {sinh, cosh};
```

- Megjegyzések

Függvénymutatók tömbje

- Függvénytömbökből álló tömböt is képezhetünk

```
1 double (*(ftrig[2]))(double) = {sin, cos};  
2  
3 for (i = 0; i < 2; ++i)  
4     printf("%f\n", ftrig[i](3.14));
```

- Könnyebben érthető a típus, ha `typedef`-et használunk

```
1 typedef double (*fvptr)(double);  
2 fvptr fhyp[] = {sinh, cosh};
```

- Megjegyzések

- A függvénymutatókkal nem végezhetünk mutatóaritmetikai műveleteket

Függvénymutatók tömbje

- Függvénytömbökből álló tömböt is képezhetünk

```
1 double (*(ftrig[2]))(double) = {sin, cos};  
2  
3 for (i = 0; i < 2; ++i)  
4     printf("%f\n", ftrig[i](3.14));
```

- Könnyebben érthető a típus, ha `typedef`-et használunk

```
1 typedef double (*fvptr)(double);  
2 fvptr fhyp[] = {sinh, cosh};
```

- Megjegyzések

- A függvénymutatókkal nem végezhetünk mutatóaritmetikai műveleteket
- A függvénymutatók read-only területre mutatnak, a függvények nem írhatók felül

Köszönöm a figyelmet.