ADATSZERKEZETEK ÉS ALGORITMUSOK

Bináris keresőfa

„Hierarchikus adatszerkezetek, keresési fák”

Rendezési (kereső) fák

- A rendezési fa (vagy keresőfa) olyan bináris fa adatszerkezet, amelynek kialakítása a különböző adatelemek között meglévő rendezési relációt követi
- A fa felépítése olyan, hogy minden csúcsra igaz az, hogy
  - a csúcs értéke nagyobb, mint tetszőleges csúcsé a tőle balra lévő leszálló ágon és
  - a csúcs értéke kisebb minden, a tőle jobbra lévő leszálló ágon található csúcs értékénél
- A T fa bármely x csúcsára és $\mathrm{bal}(x)$ bármely y csúcsára és $\mathrm{jobb}(x)$ bármely z csúcsára:

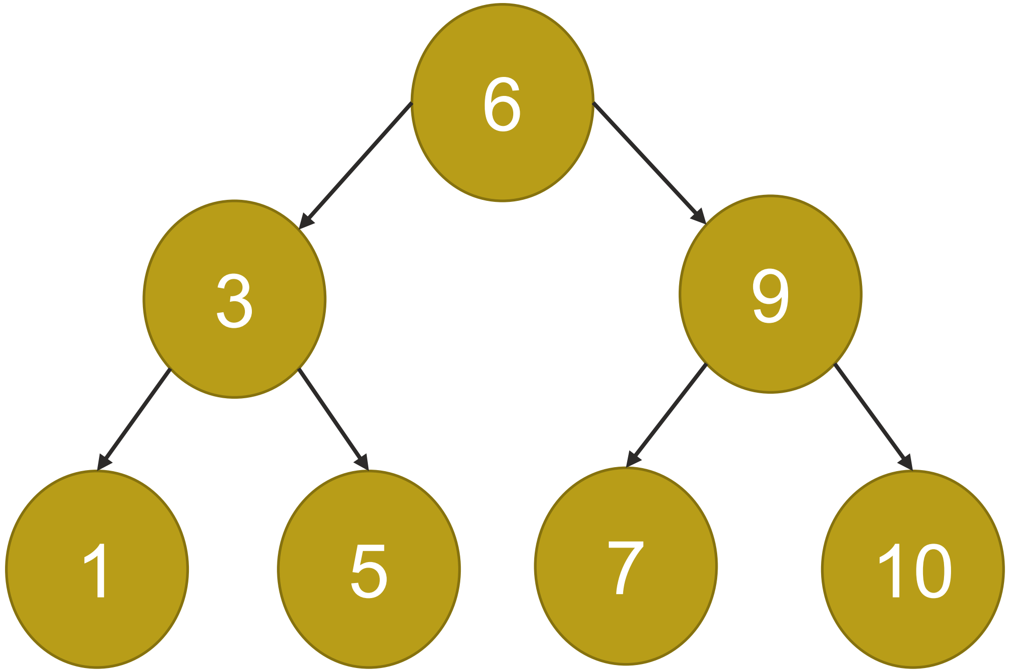
$$y < x < z$$

Rendezési (kereső) fák

- A rendezési fa az őt tartalmazó elemek beviteli sorrendjét is visszatükrözi.
- Ugyanazokból az elemekből különböző rendezési fák építhetők fel.

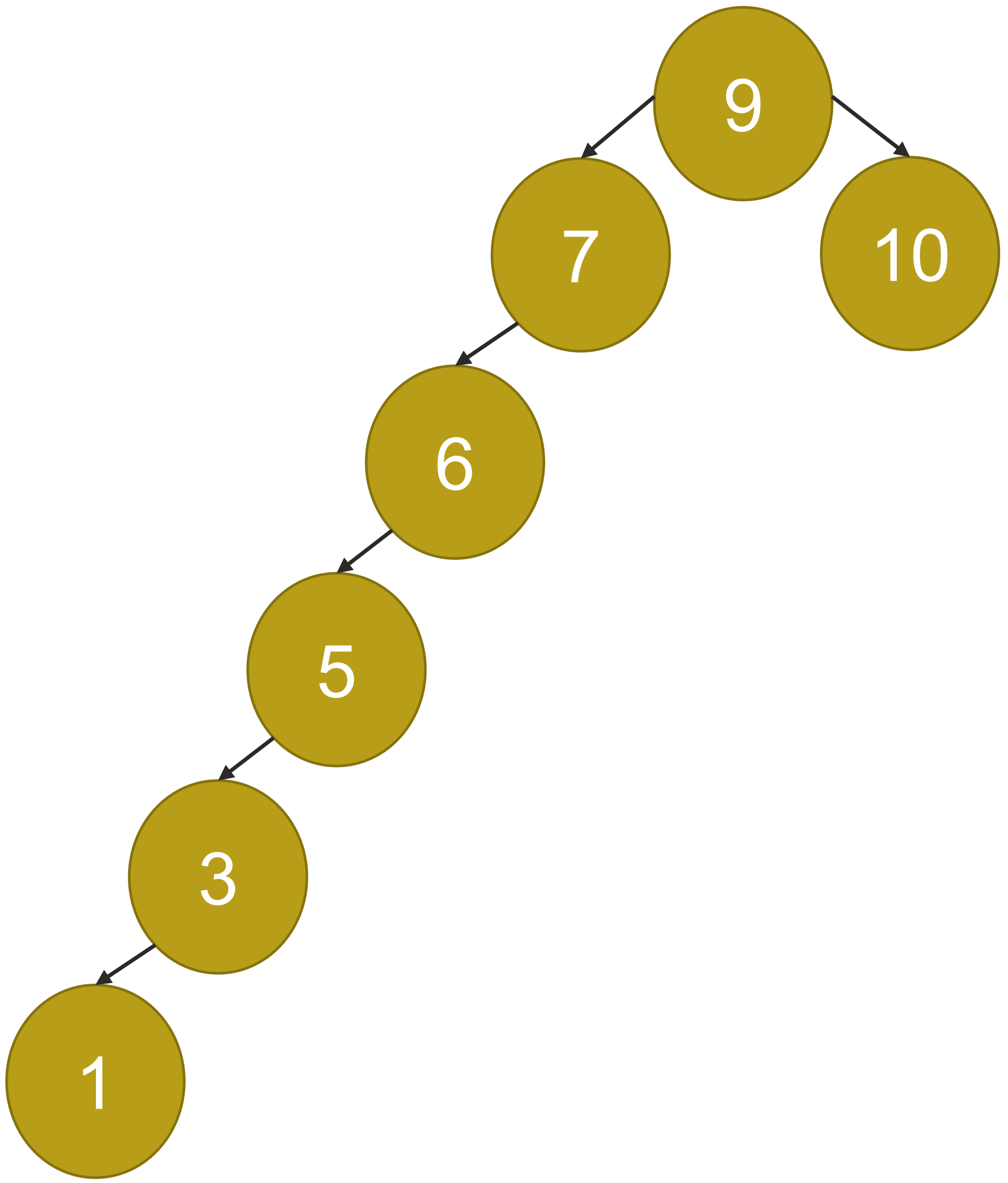
Elemek beszúrása példa

- 6,3,1,9,7,5,10



Elemek beszúrása példa

- 9,7,6,5,10,3,1



Rendezési (kereső) fák

- Fontos tulajdonság
  - inorder bejárással a kulcsok rendezett sorozatát kapjuk
- Az algoritmus pszeudokódja:

```
Inorder-fa-bejárás(x)
if x≠NIL
  then Inorder-fa-bejárás(bal[x])
    print(kulcs[x])
  Inorder-fa-bejárás(jobb[x])
```

- Egy T bináris keresőfa összes értékének kiíratásához

```
Inorder-fa-bejárás(gyökér[T])
```

Rendezési (kereső) fák

- Az algoritmus helyessége a bináris-kereső-fa tulajdonságból indukcióval adódik.

- Egy n csúcsú bináris kereső fa bejárása $\mathcal{O}(n)$ ideig tart
  - A kezdőhívás után a fa minden csúcspontja esetében pontosan kétszer (rekurzívan) meghívja önmagát
    - egyszer a baloldali részfára
    - egyszer a jobboldali részfára

Műveletek

- Keresés
  - A T fában keressük a k kulcsú elemet (csúcsot)
  - ha ez létezik, akkor visszaadja az elem címét, egyébként NIL-t.
- Az algoritmust megadjuk rekurzív és iteratív megoldásban is, ez utóbbi a legtöbb számítógépen hatékonyabb.

Műveletek

- Keresés
  - A rekurzív algoritmus pszeudokódja

```
Fában-keres(x, k)
    if x = NIL or k = kulcs[x]
      then return x
    if k < kulcs[x]
      then return Fában-keres(bal[x], k)
      else return Fában-keres(jobb[x], k)
```

Műveletek

- Keresés:
  - Az iteratív algoritmus pszeudokódja

```
Fában-iteratívan-keres(x, k)
   while x ≠ NIL and k  ≠ kulcs[x] do
     if k < kulcs[x]
       then x ← bal[x]
       else x ← jobb[x]
   return x
```

Műveletek

- Minimum keresés
  - Tegyük fel, hogy T ≠ NIL. Addig követjük a baloldali mutatókat, amíg NIL mutatót nem találunk
  - Az iteratív algoritmus pszeudokódja:

```
Fában-minimum (T)
  x ← gyökér[T]
  while bal[x] ≠ NIL
    do  x ← bal[x]
  return x
```

  - Helyessége a bináris-kereső-fa tulajdonságból következik
  - Lefut $\mathcal{O}(h)$ idő alatt, ahol h a fa magassága

Műveletek

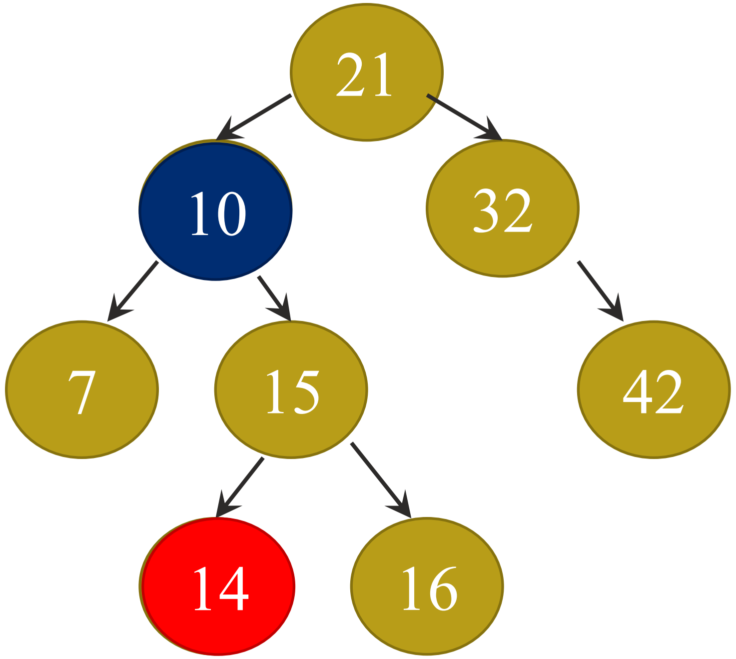
- Maximum keresés
  - Tegyük fel, hogy T $\neq$ NIL. Addig követjük a jobboldali mutatókat, amíg NIL mutatót nem találunk
  - Az iteratív algoritmus pszeudokódja:

```
Fában-maximum (T)
  x ← gyökér[T]
  while jobb[x] ≠ NIL
    do  x ← jobb[x]
  return x
```

  - Helyessége a bináris-kereső-fa tulajdonságból következik
  - Lefut $\mathcal{O}(h)$ idő alatt, ahol h a fa magassága

Műveletek

- Következő elem: x csúcs rákövetkezőjét adja vissza, ha van, NIL különben
  - Több eset lehetséges
  - Például a 10 rákövetkezője a 14
    - Létezik a megfelelő jobb részfa



  - Algoritmusa

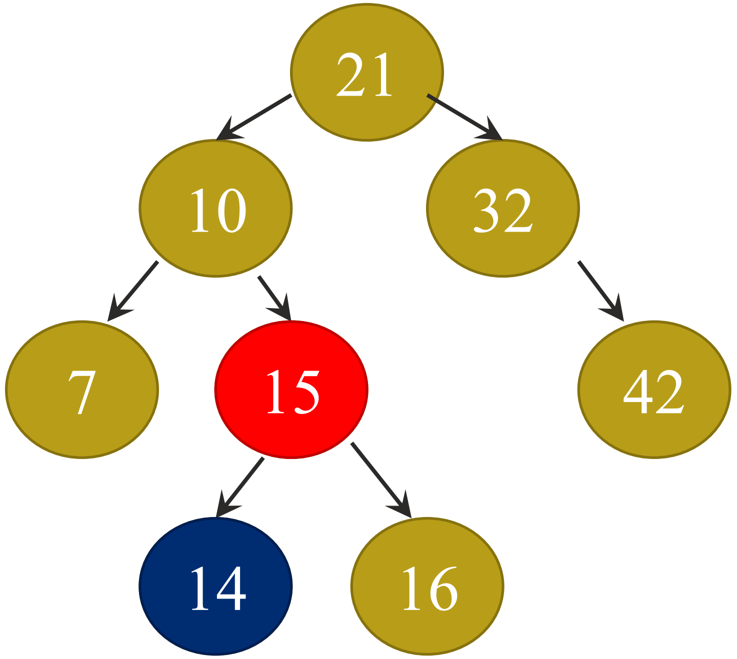
```
if jobb[x] ≠ NIL
   then return Fában-minimum (jobb[x])
```

Műveletek

- Következő elem: x csúcs rákövetkezőjét adja vissza, ha van, NIL különben
  - A 14 rákövetkezője a 15
    - Nem létezik a jobb részfa
      - Felfelé kell keresni

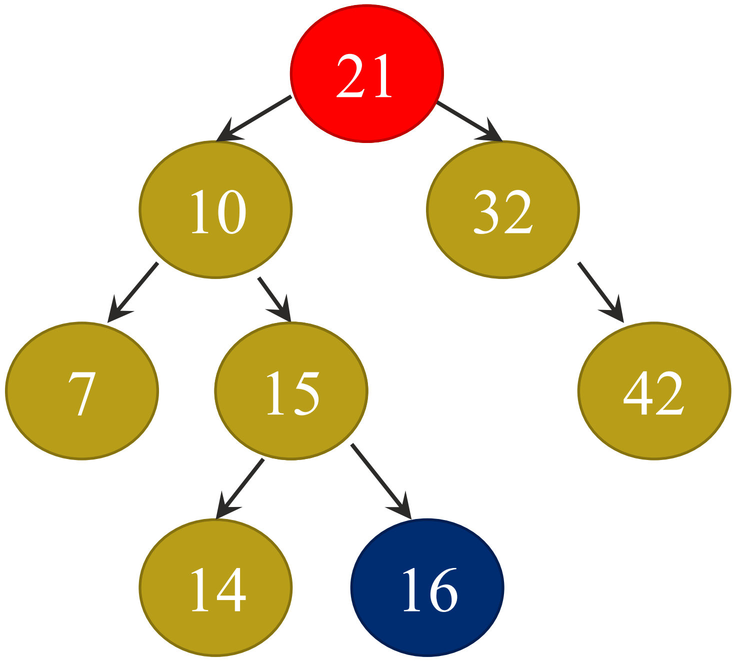
  - Algoritmusa

```
y ← szülő[x]
while y ≠ NIL és x = jobb[y] do
  x ← y
  y ← szülő[x]
return y
```



Műveletek

- Következő elem: x csúcs rákövetkezőjét adja vissza, ha van, NIL különben
  - A 16 rákövetkezője a 21
    - Nem biztos, hogy mindig a gyökér!
  - Algoritmusa




```
y ← szülő[x]
while y ≠ NIL és x = jobb[y] do
  x ← y
  y ← szülő[x]
return y
```

Műveletek

- Következő elem: x csúcs rákövetkezőjét adja vissza, ha van, NIL különben
- Teljes algoritmus

```
Fában-következő(T, x)
if jobb[x] ≠ NIL
   then return Fában-minimum (jobb[x])
y ← szülő[x]
while y ≠ NIL és x = jobb[y] do
  x ← y
  y ← szülő[x]
return y
```

Műveletek

- Fában-következő(T, x) futási ideje h magasságú fák esetén $\mathcal{O}(h)$.
- Megelőző elem: x csúcs megelőzőjét adja vissza, ha van, NIL különben.
  - `Fában-megelőző(T, x)`
  - Házi feladat

Műveletek

- Tétel: A dinamikus halmazokra vonatkozó Keres, Minimum, Maximum, Következő és Előző műveletek h magasságú bináris keresőfában $\mathcal{O}(h)$ idő alatt végezhetők el
  - Bizonyítás: az előzőekből következik

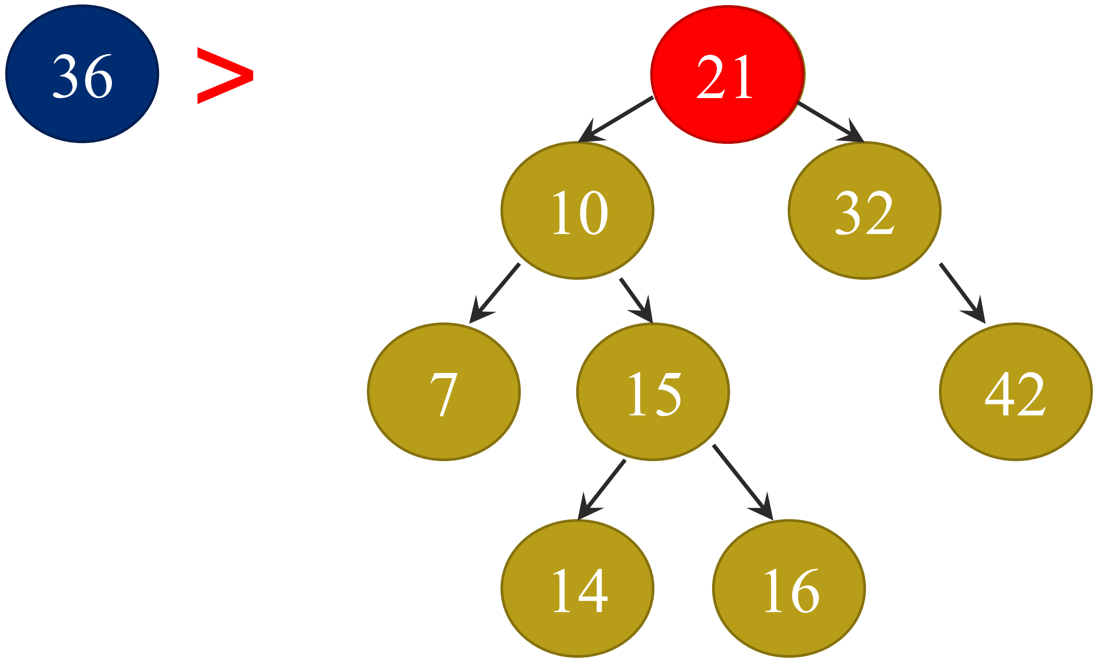
Műveletek

- Beszúrás
  - A T bináris keresőfába a p csúcsot szúrjuk be.
  - Kezdetben:
    - `kulcs[p]=k`
    - `bal[p] = NIL`
    - `jobb[p] = NIL`
    - `szülő[p] = NIL`

- Feltételezzük, hogy a fában még nincs k kulcsú csúcs!
  - Otthoni feladat megnézni, hogyan változik az algoritmus, ha ez a feltételezés nem igaz

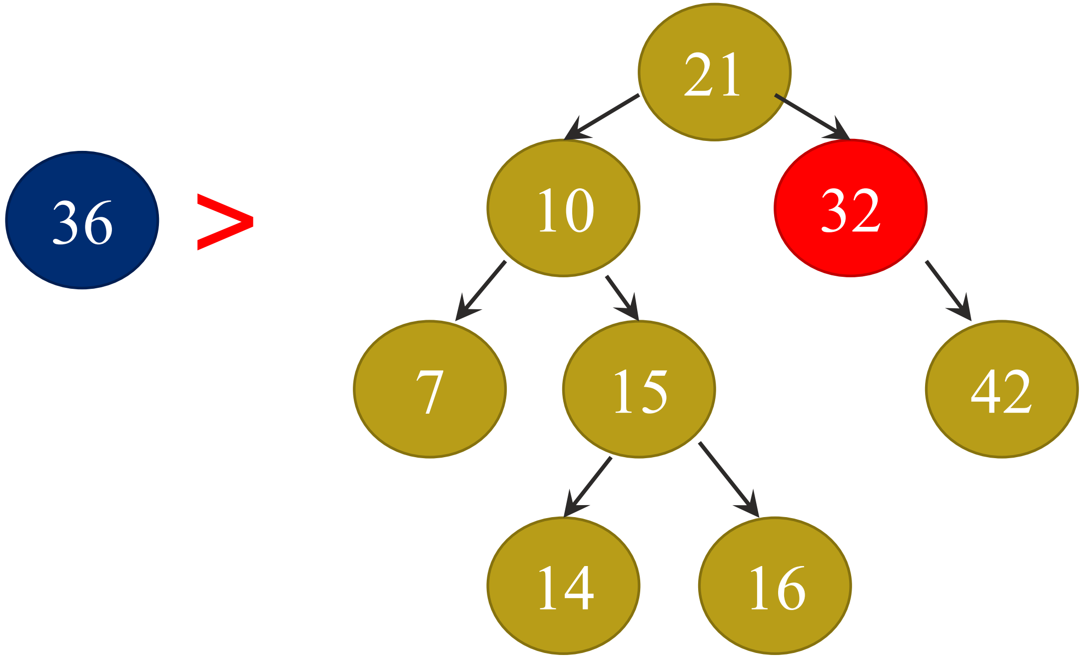
Műveletek

- Fába beszúr: szúrjuk be például a 36-t!
  - 1. megkeressük a helyét



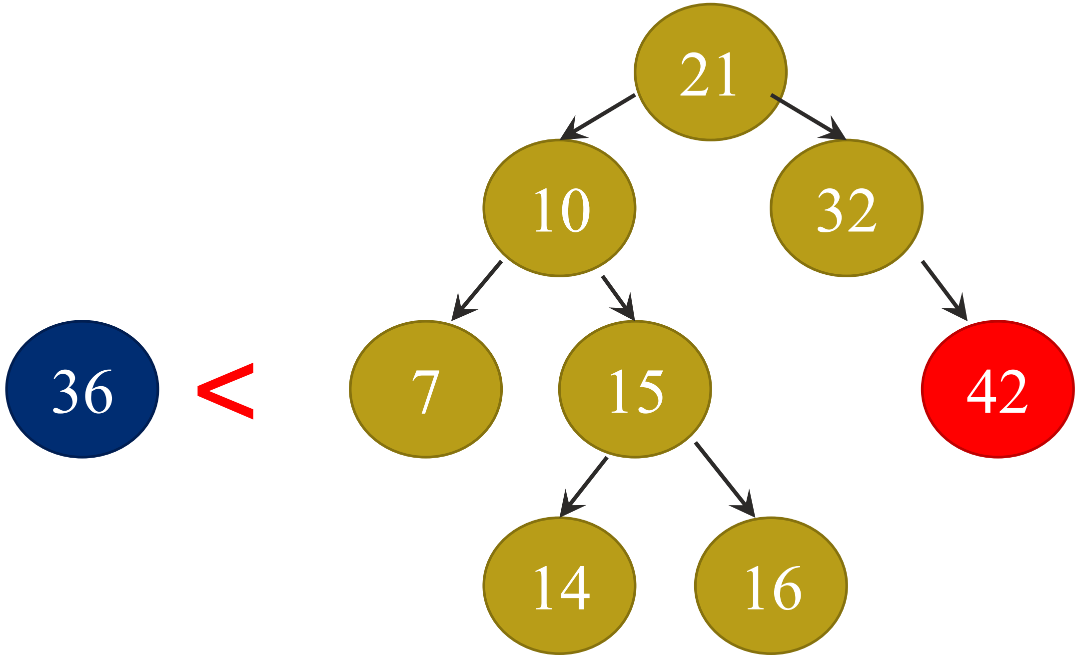
Műveletek

- Fába beszúr: szúrjuk be például a 36-t!
  - 1. megkeressük a helyét



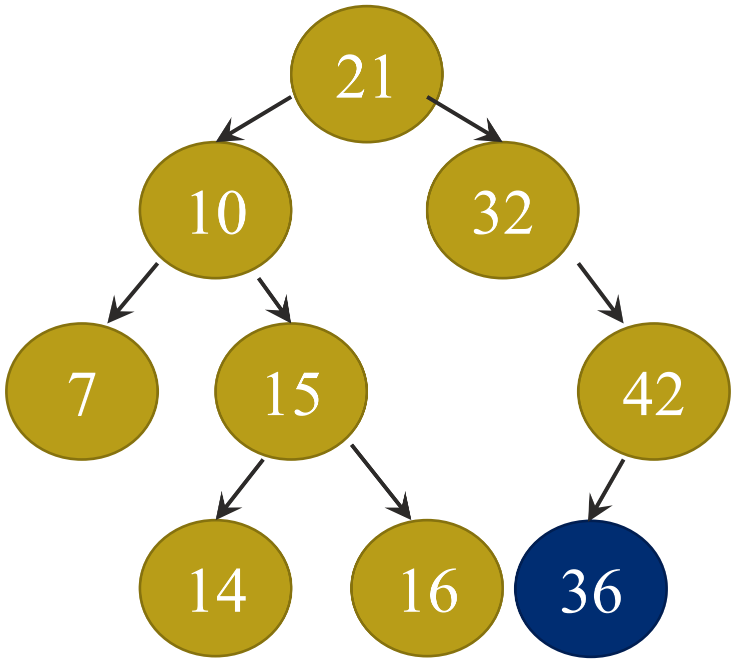
Műveletek

- Fába beszúr: szúrjuk be például a 36-t!
  - 1. megkeressük a helyét



Műveletek

- Fába beszúr: szúrjuk be például a 36-t!
  - 1. megkeressük a helyét
  - 2. beláncoljuk



Műveletek

- Algoritmus

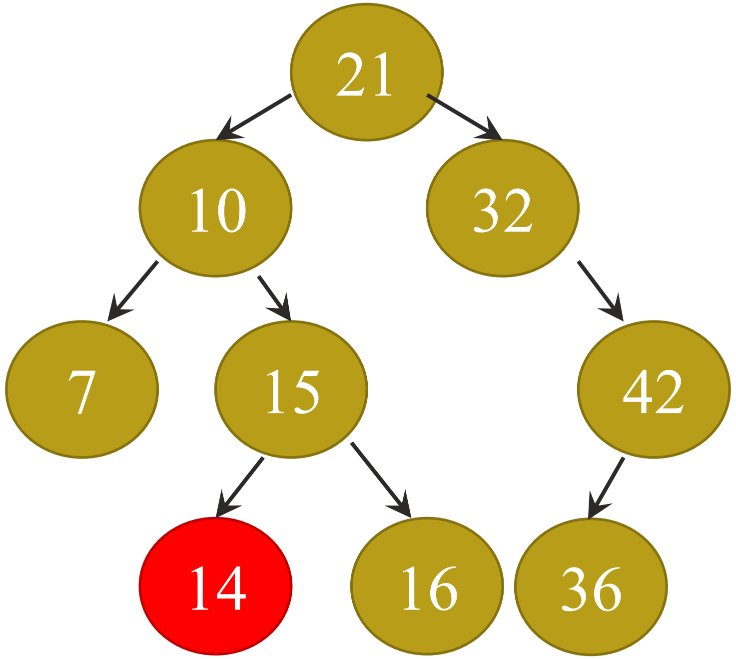
```
Fába-beszúr (T,p)
y ←NIL; x ← gyökér[T]
while x  ≠ NIL
  do y ← x
    if kulcs[p] < kulcs[x]
      then x ← bal[x]
      else x ← jobb[x]
szülő[p] ← y
if y = NIL
  then gyökér[T] ← p
  else if kulcs[p] < kulcs[y]
    then bal[y] ←p
    else jobb[y] ←p
```

Műveletek

- Törlés:
  - A T bináris keresőfából a p csúcsot töröljük
- Lehetőségek:
  1. p-nek még nincs gyereke: szülőjének mutatóját NIL-re állítjuk
  2. p-nek egy gyereke van: a szülője és a gyermeke között építünk ki kapcsolatot
  3. p-nek két gyereke van: átszervezzük a fát: kivágjuk azt a legközelebbi rákövetkezőjét, aminek nincs balgyereke így 1., vagy 2. típusú törlés, majd ennek tartalmát beírjuk p-be

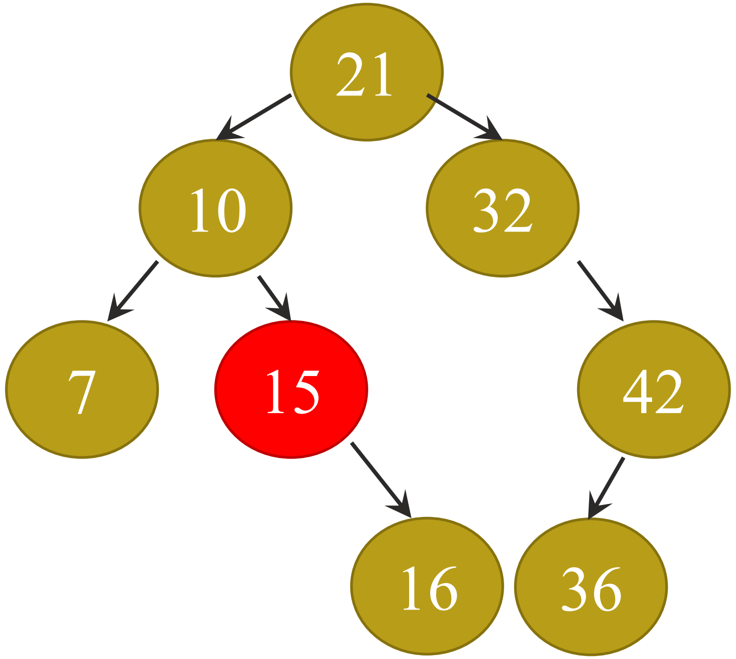
Műveletek

- Törlés: töröljük ki például a 14-t!
  - Ez a legegyszerűbb eset, alkalmazzuk az 1. szabály szerinti teendőket



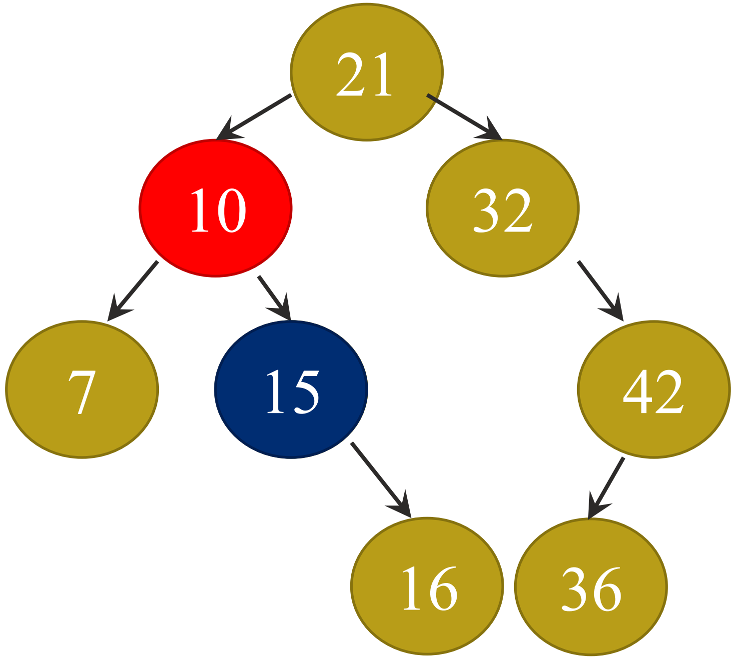
Műveletek

- Törlés: töröljük ki például a 15-t!
  - Ebben az esetben egy gyereke van a törlendőnek, tehát a 2. szabályt alkalmazzuk



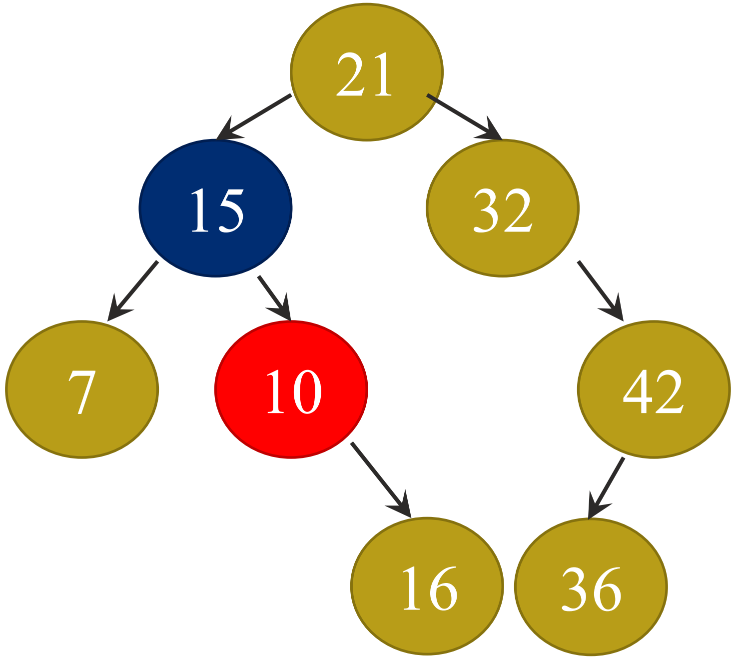
Műveletek

- Törlés: töröljük ki például a 10-et!
  - Ebben az esetben két gyereke van a törlendőnek, tehát a 3. szabályt alkalmazzuk
    - A megfelelő, rákövetkező elem a 15.
    - Ennek nincsen balgyereke
      - Ha lenne nem az lenne a rákövetkező
    - Előfordulhat, hogy jobbgyereke sincs



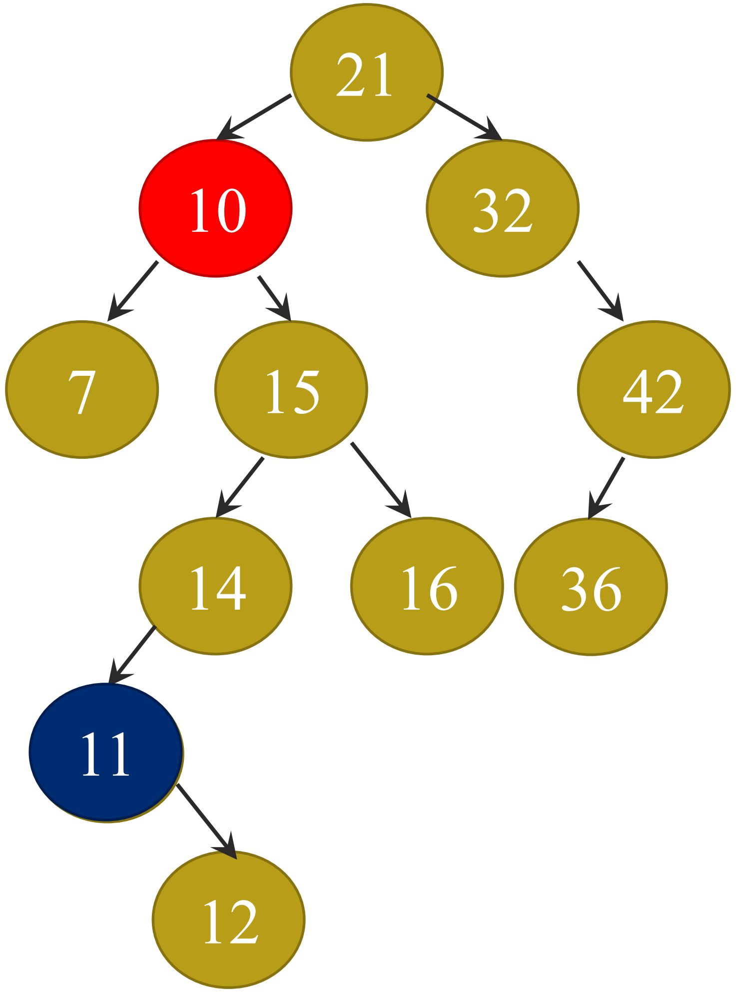
Műveletek

- Törlés: töröljük ki például a 10-et!
  - Ebben az esetben két gyereke van a törlendőnek, tehát a 3. szabályt alkalmazzuk
    - A megfelelő, rákövetkező elem a 15.
    - Ennek nincsen balgyereke
      - Ha lenne nem az lenne a rákövetkező
    - Előfordulhat, hogy jobbgyereke sincs
  - Helyet cserél a törlendő és a rákövetkező
    - Majd végrehajtjuk a törlést az eddigiek szerint



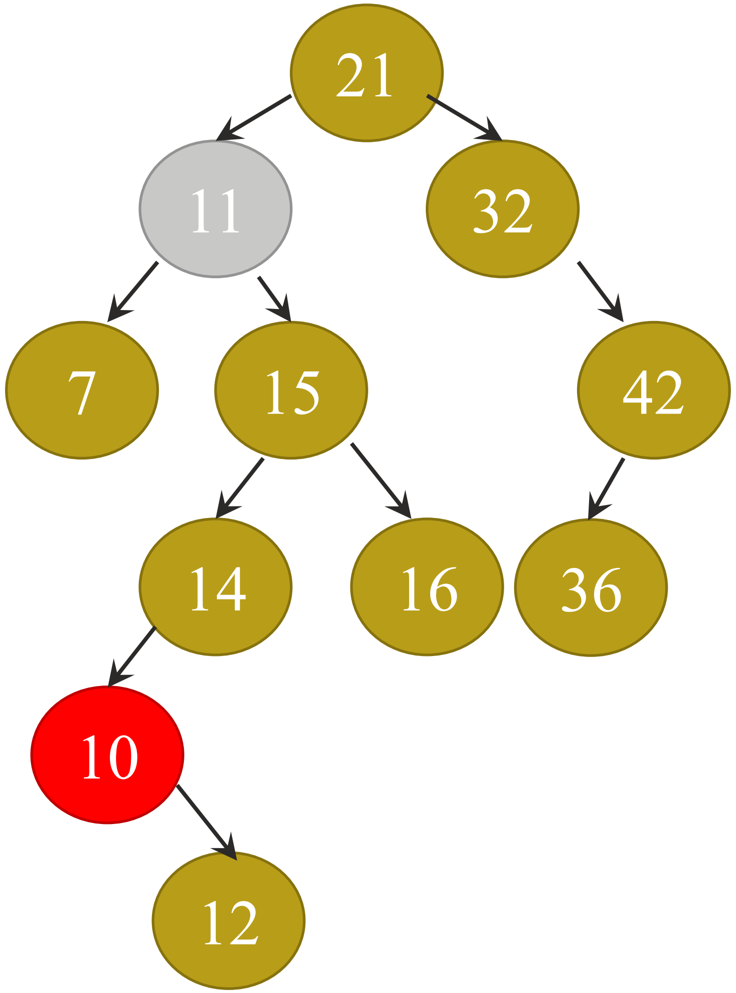
Műveletek

- Törlés: töröljük ki például a 10-et!
  - Ugyanez komplikáltabb példán



Műveletek

- Törlés: töröljük ki például a 10-et!
  - Ugyanez komplikáltabb példán



Műveletek

- Feltesszük, hogy p a T-ben létezik
- Fából-töröl (T,p)

```
if bal[p] = NIL vagy jobb[p] = NIL
  then y ←p                                    -- 0, vagy 1 gyerek
  else y ← Fában-következő(T, p)               -- 2 gyerek
if bal[y]  ≠ NIL
  then x ← bal[y]                              -- x az y 0, vagy 1 gyerekére mutat
  else x ← jobb[y]
if x  ≠ NIL                                    -- ha volt gyereke, akkor befűzi az új szülőhöz
  then szülő[x] ← szülő[y]
if szülő[y] =NIL                               -- ha a gyökeret töröltük akkor be kell állítani az újat
  then gyökér[T ] ←x                           -- különben a szülőnek megfelelő oldalhoz tartozó
  else  if  y = bal[szülő[y]]                     mutatót kell az x-re állítani
    then bal[szülő[y]] ← x
    else  jobb[szülő[y]] ← x
if  y  ≠ p                                     -- amennyiben a ténylegesen törlendő csúcs nem
  then kulcs[p] ← kulcs[y]                        azonos azzal, amit kiláncolunk át kellírni az
return y                                          adatot is
```

Bináris keresőfa megvalósítás

És összefoglalás

Bináris keresőfa műveletei

- Keresés
- Minimumkeresés
- Maximumkeresés
- Következő elem keresése
- Megelőző elem keresése
- Beszúrás
- Törlés

Bináris keresőfa műveletei I.

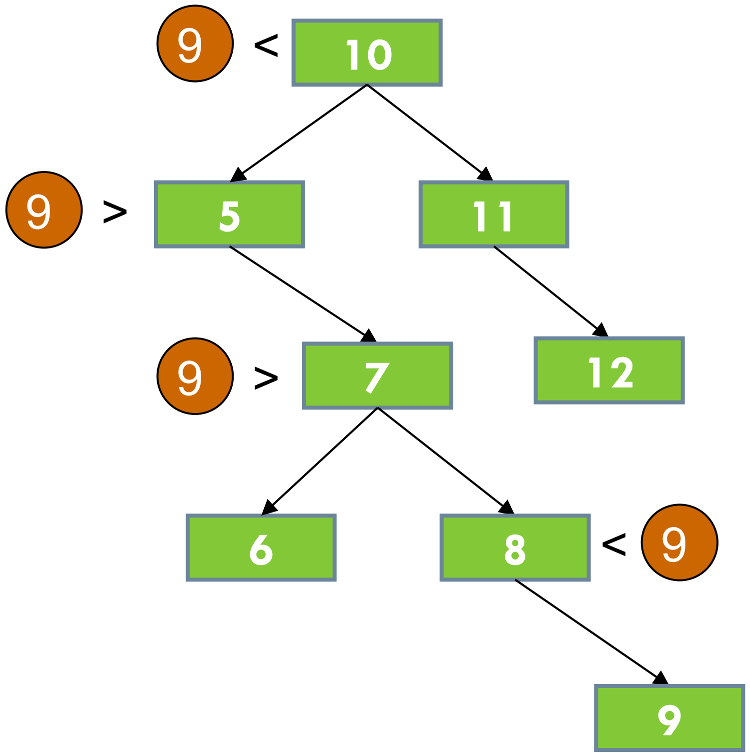
- **Beszúrás:** alapszabály:
  - nagyobb: jobbra lépünk
  - kisebb: balra lépünk
  - ha nincs gyereke ott, ahova lépnénk: beszúrunk
- **Keresés:** k kulcsú elemet keressük.
  - gyökértől indulunk
  - ha kisebb a keresett érték az aktuálisnál balra lépünk, ha nagyobb jobbra
- **Minimumkeresés részfában:**
  - addig lépünk balra, ameddig csak lehet.
  - Globális minimum: ha a gyökértől indulunk.
- **Maximumkeresés részfában:**
  - addig lépünk jobbra, ameddig csak lehet.
  - Globális maximum: ha a gyökérből indulunk.

Bináris keresőfa műveletei II.

- *Inorder bejárás esetén*
- **Következő elem:**
  - van jobb részfája: a részfa minimuma
  - nincs jobb részfája: lépjünk felfelé a fában. Az első elem aminek ő a bal részfájában van (tehát az első aminél kisebb) a rákövetkező elem.
- **Megelőző elem:**
  - van bal részfája: a részfa maximuma
  - nincs bal részfája: lépjünk felfelé a fában. Az első elem aminek ő a jobb részfájában van a keresett (megelőző értékű) elem.

Beszúrás

- A bináris keresőfába a 9 kulcsú csúcsot szúrjuk be.
- Feltételezzük (vagy ellenőrizzük), hogy a fában még nincs 9 kulcsú csúcs!
- Először megkeressük a helyét, majd beláncoljuk.



Ennek a csúcsnak már nincs jobb gyereke.
Következésképpen megtaláltuk a szülőt!

Mivel 9>8, ezért a 9 kulcsú csúcs a 8 kulcsú jobb gyereke lesz.

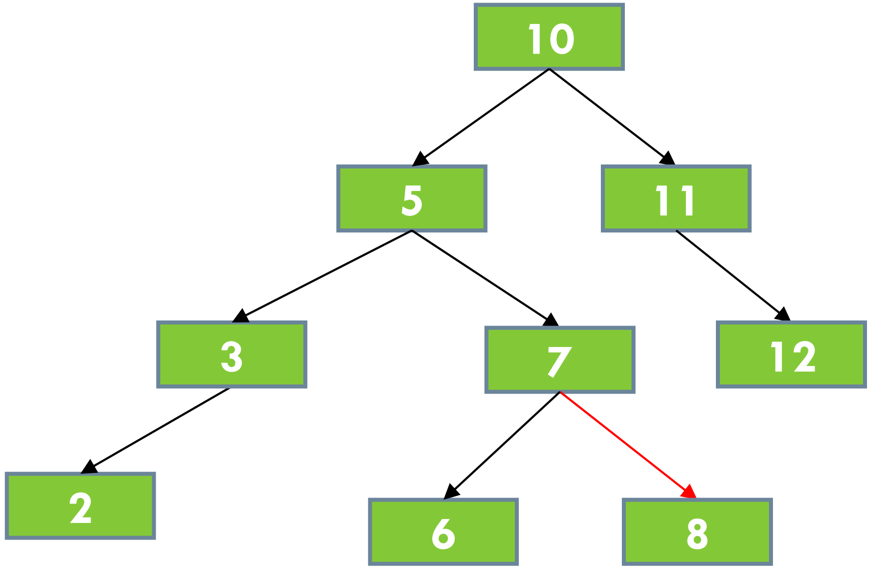
Törlés

- A bináris keresőfából a *p* kulcsú csúcsot töröljük.
- Lehetőségek:

1. A *p* kulcsú csúcsnak még nincs gyereke: szülőjének mutatóját `nullptr`-re állítjuk.
2. A *p* kulcsú csúcsnak egy gyereke van: a szülője és a gyermeke között építünk ki kapcsolatot.
3. A *p* kulcsú csúcsnak két gyereke van: átszervezzük a fát: kivágjuk azt a legközelebbi rákövetkezőjét, aminek nincs bal gyereke, így 1., vagy 2. típusú törlés, majd ennek tartalmát beírjuk a *p* kulcs helyére.

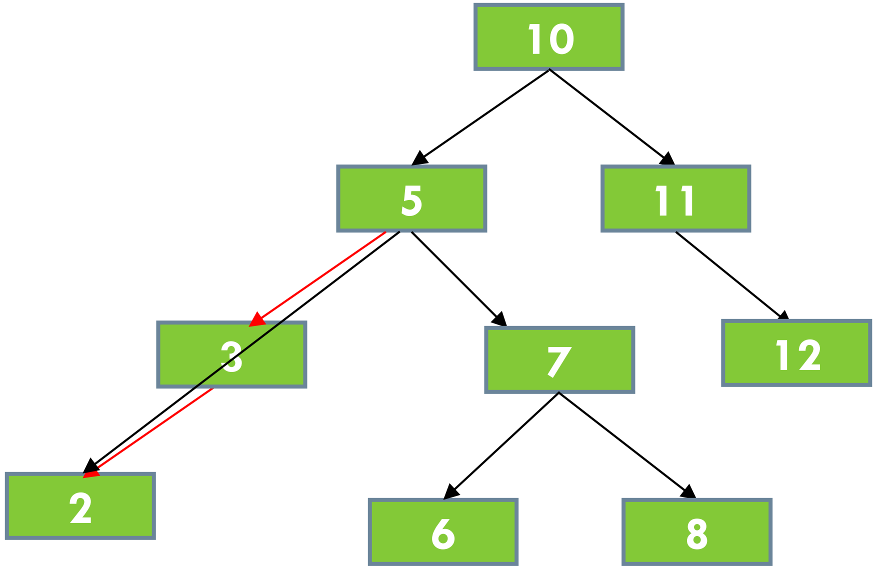
Törlés – 1. eset

- Töröljük a fából a **8** kulcsú csúcsot. Ezen csúcsnak nincs gyereke, tehát szülőjének megfelelő mutatóját `nullptr`-re állítjuk.



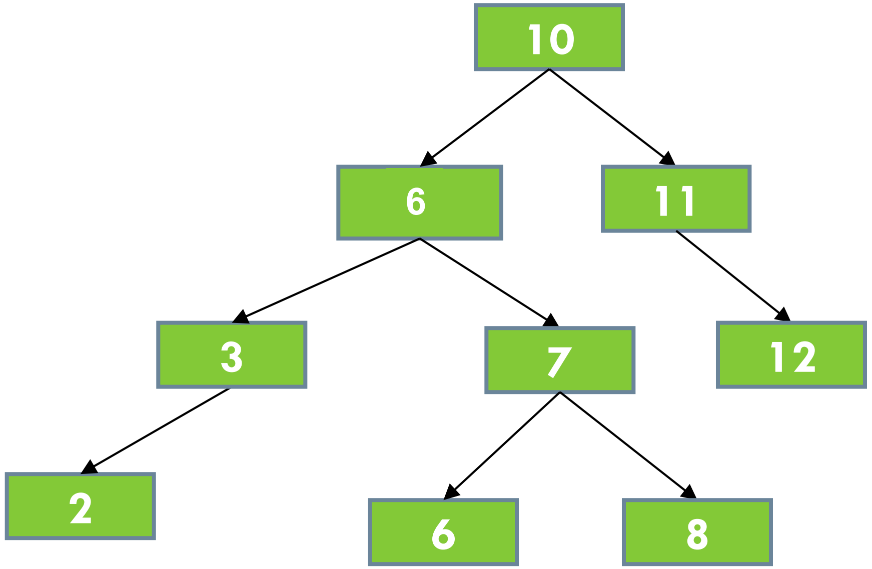
Törlés – 2. eset

- Töröljük a fából a **3** kulcsú csúcsot. Ezen csúcsnak egy gyereke van, tehát szülő és gyermeke között építünk ki kapcsolatot.



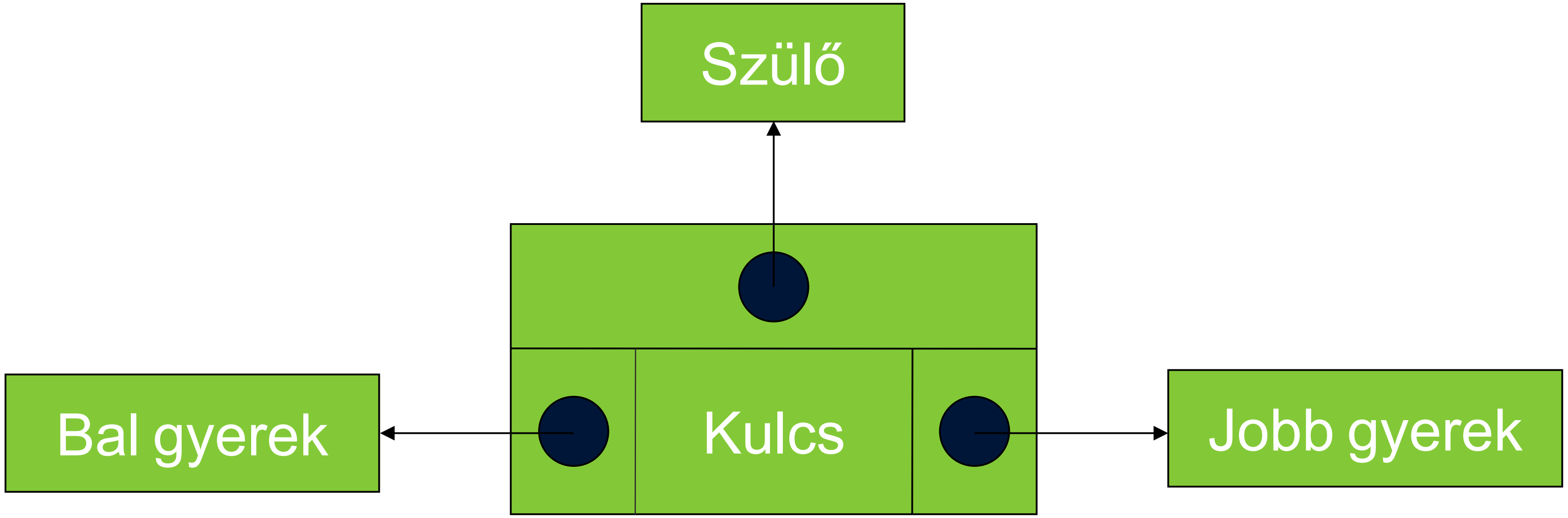
Törlés – 3. eset

- Töröljük a fából a 5 kulcsú csúcsot. Ezen csúcsnak két gyereke van, tehát megkeressük a rákövetkezőjét, ez a 6 kulcsú csúcs, és ennek nincs bal gyermeke. Így a 6 értéket beírjuk az 5 értékű csúcsba, és töröljük a 6-ost.



Reprezentáció

- A bináris keresőfák reprezentációja során is láncolt ábrázolást használunk.
- Az adatelemeket a következő módon reprezentáljuk:



C++-ban

Beágyazott elem osztály és külön kitüntetett gyökérelem:

```
template<class T>
class Bs_tree {
    // Belső csúcs struktúra
    struct Node {
        Node *parent;
        Node *left, *right;
        T key;
        Node(const T& k) : parent(nullptr), left(nullptr), right(nullptr), key(k){}
        Node(const T& k, Node *p) : parent(p), left(nullptr), right(nullptr), key(k){}
    };
    Node *root;
};
```

Fontos!
T olyan kell legyen, amire tudunk értelmezni „rendező operátort”!

Megírandó műveletek:

Amiket kívülről látunk (public):

```
size_t size();
bool isempty();
bool find(T k);
void insert(T k);
void remove(T k);
T min();
T max();
ostream& InOrder(ostream& o);
ostream& PreOrder(ostream& o);
ostream& PostOrder(ostream& o);
```

Amiket kívülről nem látunk (private):

```
Node* _next(Node* p);
Node* _prev(Node* p);
Node* _min(Node* p);
Node* _max(Node* p);
void _remove(Node* p);
size_t _size(Node *x);
ostream& _inorder(Node* i, ostream& o);
ostream& _preorder(Node* i, ostream& o);
ostream& _postorder(Node* i, ostream& o);
```