

## 1. Bináris fák

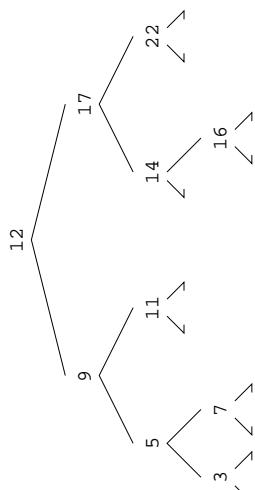
A listához hasonlóan rekurzív adatszerkezet a fa. Most bináris fák deklarációját és használatát mutatjuk be.

### 1.1. A bináris fa mint adattípus

Először olyan bináris fát dekláralunk, amelynek a levelei üresek, a csomópontjaiban pedig előbb a bal részt, majd az 'a típusú értéket, és végül a jobb részt adjuk meg:

`datatype 'a tree = L | B of 'a tree * 'a * 'a tree`

Tekintünk például az alábbi fát:



Az 'a tree adattípus L és B adatkonstruktoraival ez a fa így írható le:

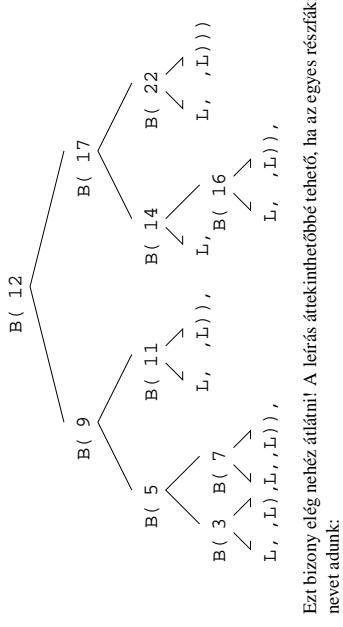
```

B(B(B(L, 3, L),
      5,
      B(L, 7, L)
    ),
  9,
  B(B(L, 11, L)
    ),
  12,
  B(B(L,
        14,
        B(L, 16, L)
      ),
    17,
    B(L, 22, L)
  )
)
  
```

A fastruktúra szöveges leírását megkönyvítő, ha az ábrába beírjuk a megfelelő adatkonstruktorkat:

```

datatype 'a badtree = L of 'a badtree
                     | B of 'a badtree * 'a * 'a badtree
  
```



Ez bizony elég nehéz általános! A leírás áttekinthetőbb lehet, ha az egyes részfáknak nevet adunk:

```

val tr3 = B(L, 3, L);
val tr7 = B(L, 7, L);
val tr5 = B(tr3, 5, tr7);
val tr11 = B(L, 11, L);
val tr9 = B(tr5, 9, tr11);
val tr16 = B(L, 16, L);
val tr14 = B(L, 14, tr16);
val tr22 = B(L, 22, L);
val tr17 = B(tr14, 17, tr22);
val tr12 = B(tr9, 12, tr17);
  
```

Természetesen másfél fastruktúrákat is deklaráthatunk, pl. kezdhetjük az 'a típusú értékekkel, majd folytatjuk előbb a bal, azután a jobb részt megadásával. Felhasználhatjuk a levelet is értékek tárolására, vagy előírhatjuk, hogy csak a levelek lehet érték stb. A

```

datatype 'a tree = E | L of 'a
                  | B of 'a tree * 'a * 'a tree
  
```

deklaráció például abban különbözik a korábban már látott deklarációtól, hogy a bináris fa leveleiben is tárolunk értéket, az értéket nem tároló üres csomókot pedig Evel jelöljük.  
A rekurzív függvényekhez hasonlóan a rekurzív adattípusoknak is kell hogy legyen trivialis eset. Szintaktikailag helyesek az alábbi deklarációk is, de a trivialis eset hiányában alkalmatlannak adatok létrehozására:

```

datatype 'a badtree = B of 'a badtree * 'a * 'a badtree
                     | B of 'a badtree * 'a * 'a badtree
  
```

## 1.2. Egyszerű műveletek bináris fákon

Most bináris fákra alkalmazható, jól ismert műveletekre írunk SML-függvényeket. A példákban az alábbi típusdeklarációi használjuk:

```
datatype 'a tree = L | N of 'a * 'a tree * 'a tree
```

nodes egy fa csomópontjai számlálja meg. Ehhez hasonlóan számolhatók meg a fa levelei.

```
(* nodes f = az f fa csomópontjainak a száma
   nodes : 'a tree -> int
*)
fun nodes L = 0
| nodes (N(_, t1, t2)) = 1 + nodes t1 + nodes t2

Jobbrekurzív változata jobbkötésekhez.
```

```
(* nodes f = az f fa csomópontjainak a száma
   nodes : 'a tree -> int
*)
fun nodes f =
  let (* nodes0(f, n) = n + a csomópontok száma
      f-bei
      nodes0 : 'a tree * int -> int
      *)
    fun nodes0 (L, n) = n
    | nodes0 (N(_, t1, t2), n) =
        nodes0(t1, nodes0(t2, n+1))
  in
    nodes0(f, 0)
  end
```

depth egy fa mélysége (más szóhasználat a magasságat) határozza meg. A fa gyökérhől a levelshez vezető úton az élék számát (az út hosszát) az adott levél szintjének is nevezik. A szintek közül a legnagyobbat a fa mélységek hívjuk.

```
(* depth f = az f fa mélysége
   depth : 'a tree -> int
*)
fun depth f =
  let fun depth0 (L, d) = d
    | depth0 (N(_, t1, t2), d) =
        Int.max(depth(t1, d+1),
                depth(t2, d+1))
  in
    depth0(f, 0)
```

end

fulltree  $n$  mélysegről teljes bináris fát épít, és a fa csomópontjait 1-től  $2^n$ -ig beszámza. Egy teljes bináris fában minden csomópontból pontosan két elindul ki, és minden levelek nyújanak a szintje.

```
(* fulltree n = n mélysegről teljes fa
   fulltree : int -> 'a tree
*)
fun fulltree n =
  let fun ftree (_ , 0) = L
    | ftree (k, n) = N(k, ftree(2*k, n-1),
                           ftree(2*k+1, n-1))
  in ftree(1, n)
  end
```

reflect a fát a függőleges tengelyre mentén tükrözi.

```
(* reflect =
   reflect : 'a tree -> 'a tree
*)
fun reflect L = L
| reflect (N(v, t1, t2)) = N(v, reflect t2, reflect t1)
```

## 1.3. Lista előállítása bináris fa elemeiből

preorder, inorder és postorder bináris fárból lista állít el. Ahogy a nevük sugallja, a három függvény abban különbözik egymástól, hogy az esy csomópontból az ott található értéket mikor veszik ki, és milyen sorrendben járják be a bal, ill. a jobb részfát.

preorder először az értéket veszi ki, majd bejárja a bal, és azután a jobb résztát. inorder először bejárja a bal résztát, majd kiveszi az értéket, és végi bejárja a jobb résztát. postorder először bejárja a bal, majd a jobb résztát, és utoljára veszi ki az értéket.

Az alábbi megalakítások egyszerűek, érthetőek, de nem elég hatékonyak (@ operátor használata miatt).

```
(* preorder f = az f fa elemeinek preorder sorrendű
   listája
*)
preorder : 'a tree -> 'a list
(*)
fun preorder L = []
| preorder (N(v,t1,t2)) =
  preorder (N(v,t1,t2)) @
  preorder t1 @ preorder t2

(* inorder f = az f fa elemeinek inorder sorrendű
   listája
*)
inorder : 'a tree -> 'a list
(*)
fun inorder L = []
| inorder (N(v,t1,t2)) =
  inorder (N(v,t1,t2)) @
  inorder t1 @ (v :: inorder t2)
```

(\* postorder f = az f fa elemeinek postorder sorrendű  
listája  
postorder : 'a tree -> 'a list

\*)  
fun Postorder L = []  
| postorder (N(v,t1,t2)) =  
postorder t1 @ postorder t2 @ [v]

A gyűjtőargumentum használata miatt nehezebben érthetőek, de *hatékonyabbak* a jobbkurzív változatok. (A függvények „erjéjeséül” most elektintik.)

(\* preord(f, vs) = az f fa elemeinek a vs lista elé  
fűzött, preorder sorrendű listája  
preord : 'a tree \* 'a list -> 'a list

\*)  
fun Preord (L, vs) = vs !!!: rev Postord !!!:  
| Preord (N(v,t1,t2), vs) =  
V::preord(t1, preord(t2, vs))

(\* inord(f, vs) = az f fa elemeinek a vs lista elé  
inorder : 'a tree \* 'a list -> 'a list

\*)  
fun Inord (L, vs) = vs  
| Inord (N(v,t1,t2), vs) = inord(t1, v::inord(t2, vs))

(\* postord(f, vs) = az f fa elemeinek a vs lista elé  
fűzött, postorder sorrendű listája  
Postord : 'a tree \* 'a list -> 'a list

\*)  
fun Postord (L, vs) = vs  
| Postord (N(v,t1,t2), vs) =  
Postord(t1, Postord(t2, v::vs))

flügvenyt, amelynek egy egészből (xs) és egy listából (xs) álló pár az argumentuma, és ugyancsak egy pár az eredménye. E pár első tagja a lista első k db eleme, második tagja pedig a lista többi eleme legyen.

(\* take'ndrop(k, xs) = olyan pár, amelynek első tagja  
xs első k db eleme, második  
tagja pedig xs maradéka

\*)  
take'ndrop : int \* 'a list -> 'a list \*

in tdi(k, xs, [])

end

take'ndrop felhasználása, nevezetesen az eredményül átadt pár miatt módosítani kell balpreorder felírásán.

(\* balpreorder xs = az xs lista elemeiből álló,  
preorder bejárású, kiegyszűlyozott fa  
balpreorder: 'a list -> 'a tree

\*)  
fun balpreorder (x:xs) =  
let val k = length xs div 2  
val (ts, ds) = take'ndrop(k, xs)  
in N(x, balpreorder ts, balpreorder ds)

end

| balpreorder [] = L

balinorder take'ndrop-pal val definiált gyakorló feladatait az olvasóra bízzuk.

(\* balinorder xs = az xs lista elemeiből álló, inorder  
bejárású, kiegyszűlyozott fa  
balinorder: 'a list -> 'a tree

\*)  
fun balinorder (x:xs as x:xs) =  
let val k = length xxs div 2  
val (y:ys) = drop(k, xxs)  
in N(y, balinorder (take(k, xxs)), balinorder ys)

end

| balinorder [] = L

(\* balpostorder xs = az xs lista elemeiből álló,  
postorder bejárású, kiegyszűlyozott fa  
balpostorder: 'a list -> 'a tree

\*)  
fun balpostorder xs =  
let val k = length xs div 2 !!! kiemeini!!!  
in N(xs, balpreorder (take(k, xs)),  
balpreorder (drop(k, xs)))

end

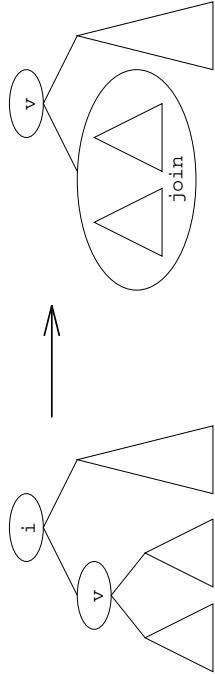
| balpostorder [] = L

A hatékonyiságot kisebb mértékben rontja, hogy take és drop egymástól függetlenül készítenek végeg a lista első részén. Írfunk take'ndrop néven olyan

## 1.5. Elem törlése bináris fárból

Bináris fában adott értékű elemet rekurzív módszerrel megkeresni egyszerű feladat. *Új elemet hozzáírni* sem nehéz: rekurzív módszerrel keressük egy levelet, és ennek a helyére berakjuk az új értéket. Ha a fa rendezi van, ügyelünk kell arra, hogy a rendezettség megmaradjon.

Bináris fárból adott értékű elemet vagy elemeket rekurzív módszerrel *kiterölni* valamivel nehezebb: ha a fölrendű értéket az éppen vizsgált részfa gyökérében van, a két részre szétszakítva részfáit valamilyen módon *egyesíteni* kell, miután a törlést a két részfán már végrehoztottuk.



A vázolt műveletre mutat példát az alábbi remove függvény: rendezetten bináris fárból törli az i értékű elem összes előfordulását. A join segédfüggvényel egyszerűíti a törlés hatásra létrejövő két részfát, megpedig úgy, hogy a bal fát lebonjuk, és közben az elemeit berakjuk a jobb fába.

```
(* join(b, j) = a b és a j fák egysítésével
   létrehozott fa
   join : 'a tree * 'a tree -> 'a tree
   *) fun join (L, tr) = tr
      | join (N(v, lt, rt), tr) = N(v, join(lt, rt), tr)
      (* remove(i, f) = i összes előfordulását törli f-ból
         remove : 'a * 'a tree -> 'a tree
         *) fun remove (i, L) = L
            | remove (i, N(v, lt, rt)) =
               if i <> v
                  then N(v, remove(i, lt), remove(i, rt))
                  else join(remove(i, lt), remove(i, rt))
            *)
            fun bremove (i, f) = raise Bsearch("remove: " ^ i)
               | bremove (N((a,x), t1, t2), f) =
                  if a = b
                     then N((a,x), bupdate(t1, (b,y)), t2)
                  else if a < b
                     then N((a,x), t1, bupdate(t2, (b,y)))
                  else (* a=b *) raise Bsearch("remove: " ^ i)
```

Megjegyük azt is, hogy előbb egyszerűk a két részfát, majd az eredményül kapott fárból törljük az adott értékű elemet. Ez a feladatot gyakorlásként az olvasóra bízzuk.

## 1.6. Bináris keresőfák

Ebben a szakaszban *bináris keresőfákon* alkalmazható műveleteket definíálunk.

Rendszerint adott kulcsú elemet keresünk, ehhez értékeket kell összehasonítanunk egymással; a keresett kulcsnak tehát *egyenlősségi tipusának* kell lennie. A példában a string típus használjuk, de a típus természetesen tetszőleges más egyenlőségi típus

is lehet. Szebb lenne, ha a *generikus függvényeket* írnánk; ezt a feladatot gyakorlatáshoz megihigynuk az olvasónak. A függvények *kivételes helyezet* jeleznek, ha a keresett kulcsú elem nincs a keresőfában.

**exception Bsearch of string**

A blookup függvény adott kulcshoz tartozó értéket ad vissza egy rendezett bináris fárból:

```
(* blookup(f, b) = az f fában a b kulcsúhoz tartozó érték
   blookup : (string * 'a) tree * string -> 'a
   *) fun blookup (N((a,x), t1, t2), b) =
      if b < a
         then blookup(t1,b)
      else if a < b
         then blookup(t2, b)
      else x
      | blookup (L, b) = raise Bsearch("LOOKUP: " ^ b)
```

A binsert függvény egy új kulcsú elemet rak be egy rendezett bináris fába, ha még nincs benne:

```
(* binsert(f, (b,y)) = az új (b,y) kulcs-érték párral bővíttet f fa
   binsert : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   *) fun binsert (L, (b,y)) = N((b,y), L, L)
      | binsert (N((a, x), t1, t2), (b,y)) =
         if b < a
            then N((a, x), binsert(t1, (b,y)), t2)
         else if a < b
            then N((a, x), t1, binsert(t2, (b,y)))
         else (* a=b *) raise Bsearch("INSERT: " ^ b)
```

A bupdate függvény meglévő kulcsú elembe új értéket raktározza a rendezett bináris fában:

```
(* bupdate(f, (b,y)) = az f fa, a b kulcsúhoz tartozó érték helyén az y értékkel bupdate : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   *) fun bupdate (L, (b,y)) = raise Bsearch("UPDATE: " ^ i)
      | bupdate (N((a,x), t1, t2), (b,y)) =
         if b < a
            then N((a,x), bupdate(t1, (b,y)), t2)
         else if a < b
            then N((a,x), t1, bupdate(t2, (b,y)))
         else (* a=b *) N((b,Y), t1, t2)
```