1. Bináris fák

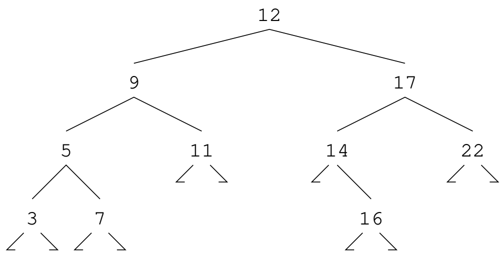
A listához hasonlóan rekurzív adatszerkezet a fa. Most bináris fák deklarációját és használatát mutatjuk be.

1.1. A bináris fa mint adattípus

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

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

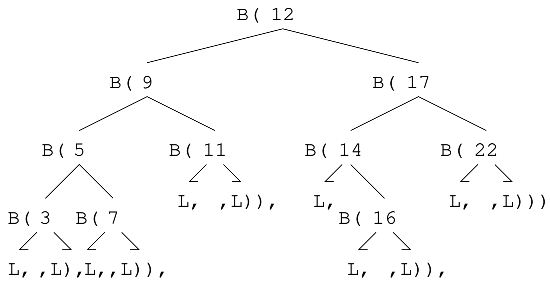
Tekintsük például az alábbi fát:



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

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

A fastruktúra szöveges leírását megkönnyíti, ha az ábrába beírjuk a megfelelő adatkonstruktorokat:



Ezt bizony elég nehéz átlátni! A leírás áttekinthetőbbé tehető, 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 másféle fastruktúrákat is deklarálhatunk, pl. kezdhetjük az `'a` típusú értékkel, majd folytathatjuk előbb a bal, azután a jobb részfa megadásával. Felhasználhatjuk a levelet is értékek tárolására, vagy előírhatjuk, hogy csak a levélben 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

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

deklarációtól, hogy a bináris fa leveleiben is tárolunk értéket, az értéket nem tároló üres csonkokat pedig `E`-vel jelöljük.
A rekurzív függvényekhez hasonlóan a rekurzív adattípusoknak is kell hogy legyen triviális esete. Szintaktikailag helyesek az alábbi deklarációk is, de a triviális eset hiánya miatt alkalmatlanok adatok létrehozására:

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

datatype 'a badtree = L of '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 típusdeklarációt használjuk:

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

`nodes` egy fa csomópontjait 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 jobb hatékonyságú.

```
(* nodes f = az f fa csomópontjainak a száma
   nodes :  'a tree -> int
*)
fun nodes f =
      let (* nodes0(f, n) = n + a csomópontok száma
f-ben
             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 mélységét (más szóhasználattal a magasságát) határozza meg. A fa gyökeréből a leveléhez vezető úton az élek számát (az út hosszát) az adott levél szintjének is nevezzük. A szintek közül a legnagyobbat a fa mélységének hívjuk.

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

`depth` jobbrekurzív változata:

```
(* 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 mélységű *teljes bináris fát* épít, és a fa csomópontjait 1-től $2n$-ig beszámozza. Egy teljes bináris fában minden csomópontból pontosan két él indul ki, és minden levelének ugyanaz a szintje.

```
(* fulltree n = n mélységű 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 tengelye 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 fából listát* állít elő. Ahogy a nevük is sugallja, a három függvény abban különbözik egymástól, hogy az egy csomópontból az ott tárolt 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 részfát. `inorder` először bejárja a bal részfát, majd kiveszi az értéket, és végül bejárja a jobb részfát. `postorder` először bejárja a bal, majd a jobb részfát, és utoljára veszi ki az értéket.
Az alábbi megvalósítások egyszerűek, érthetőek, de nem elég hatékonyak (a `@` 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)) =
                    v ::  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 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 jobbrekurzív változatok. (A függvények „elrejtésétől” most eltekintünk.)

```
(* 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
                  fűzött, inorder sorrendű listája
   inord :  '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))
```

1.4. Bináris fa előállítása lista elemeiből

Listát *kiegyensúlyozott bináris fává* alakítanak a következő függvények; a különbség közöttük most is a bejárási sorrendben van.

```
(* balpreorder xs = az xs lista elemeiből álló,
                preorder bejárású, kiegyensúlyozott fa
    balpreorder:  'a list -> 'a tree
*)
fun balpreorder (x::xs) =
      let val k = length xs div 2 !!!  kiemelni!!!
      in N(x, balpreorder(take(k, xs)),
              balpreorder(drop(k, xs)))
      end
  | balpreorder [] = L
```

A hatékonyságot kisebb mértékben rontja, hogy `take` és `drop` egymástól függetlenül *kétszer* mennek végig a lista első felén. Írjunk `take'ndrop` néven olyan

függvényt, amelynek egy egészből (`k`) 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 * 'a list
*)
fun take'ndrop (k, xs) =
      let fun td (0, xs, ts) = (rev ts, xs)
            | td (k, x::xs, ts) = td(k-1, xs, x::ts)
            | td (_, [], ts) = (rev ts, [])
      in
          td(k, xs, [])
      end
```

`take'ndrop` felhasználása, nevezetesen az eredményül átadott pár miatt módosítani kell `balpreorder` felépítésén.

```
(* balpreorder xs = az xs lista elemeiből álló,
                preorder bejárású, kiegyensúlyozott 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
```

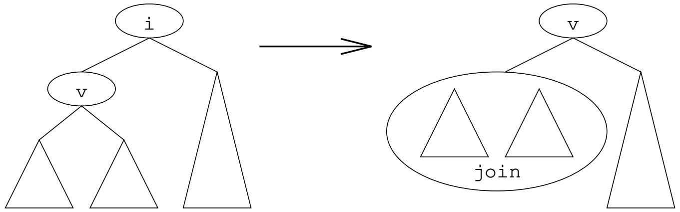
`balinorder` `take'ndrop`-pal való definiálását gyakorló feladatként az olvasóra bízzuk.

```
(* balinorder xs = az xs lista elemeiből álló, inorder
                   bejárású, kiegyensúlyozott fa
    balinorder:  'a list -> 'a tree
*)
fun balinorder (xxs 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, kiegyensúlyozott fa
    balpostorder:  'a list -> 'a tree
*)
fun balpostorder xs = balpreorder(rev xs)
```

1.5. Elem törlése bináris fából

Bináris fában adott értékű *elemet* rekurzív módszerrel *megkeresni* egyszerű feladat. *Új elemet beszúrni* sem nehéz: rekurzív módszerrel keresünk egy levelet, és ennek a helyére berakjuk az új értéket. Ha a fa rendezve van, ügyelnünk kell arra, hogy a rendezettség megmaradjon.
Bináris fából adott értékű *elemet* vagy *elemeket* rekurzív módszerrel *kitörölni* valamivel nehezebb: ha a törlendő érték az éppen vizsgált részfa gyökerében van, a két részre széteső fa részfáit valamilyen módon *egyesíteni* kell, miután a törlést a két részfán már végrehajtottuk.



A vázolt műveletre mutat példát az alábbi `remove` függvény: rendezetlen bináris fából törli az `i` értékű elem összes előfordulását. A `join` segédfüggvénnyel egyesítjük a törlés hatására létrejövő két részfát, mégpedig úgy, hogy a bal fát lebontjuk, és közben az elemeit berakjuk a jobb fába.

```
(* join(b, j) = a b és a j fák egyesítésével
                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-ből
   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))
```

Megtehetjük azt is, hogy előbb egyesítjük a két részfát, majd az eredményül kapott fából töröljük az adott értékű elemet. Ezt a feladatot gyakorlásként az olvasóra bízzuk.

1.6. Bináris keresőfák

Ebben a szakaszban *bináris keresőfákon* alkalmazható műveleteket definiálunk. Rendszerint adott kulcsú elemet keresünk, ehhez értékeket kell összehasonlítanunk egymással; a keresett kulcsnak tehát *egyenlőségi típusúnak* kell lennie. A példákban a `string` típust használjuk, de a típus természetesen tetszőleges más egyenlőségi típus

is lehet. Szebb lenne, ha *generikus függvényeket* írnánk; ezt a feladatot gyakorlásképpen meghagyjuk az olvasónak. A függvények *kivételes helyzetet* 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 fából:

```
(* blookup(f, b) = az f fában a b kulcshoz 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
                       bővített 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 ír be egy rendezett bináris fában:

```
(* bupdate(f, (b,y)) = az f fa, a b kulcshoz tartozó
                       érték helyén az y értékkel
   bupdate :  (string * 'a) tree * (string * 'a) ->
                                    (string * 'a) tree
*)
fun bupdate (L, (b,y)) = raise Bsearch("UPDATE: " ^ b)
  | 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)
```