

HALMAZMŰVELETEK

Halmazműveletek: „benne van-e?” (isMem) és „ha új, tedd bele” (newMem)

- isMem igaz értéket ad eredményül, ha a keresett elem benne van a listában.

```
(* isMem : 'a * 'a list -> bool
   isMem(x, ys) = x eleme-e ys-nek
*)
fun op isMem (_, []) = false
  | op isMem (x, y::ys) = x = y orelse op isMem(x, ys)
infix isMem
```

Megjegyzés: az op operátor nélkül az infix deklarációt követően a függvénydefiníciót nem lehetne újrafordítani.

- newMem egy új elemet rak be egy listába, ha még nincs benne.

```
(* newMem : 'a * 'a list -> 'a list
   newMem(x, xs) = [x] és xs listaként ábrázolt uniója
*)
fun newMem (x, xs) = if x isMem xs
                    then xs
                    else x::xs
```

newMem, ha a sorrendtől eltekintünk, halmazt hoz létre.

Halmazműveletek: „listából halmaz” (setof)

- setof halmazt készít egy listából úgy, hogy kiszedi belőle az ismétlődő elemeket. Rossz hatékonyságú.

```
(* setof : 'a list -> 'a list
   setof xs = xs elemeinek listaként ábrázolt halmaza
*)
fun setof []      = []
  | setof (x::xs) = newMem(x, setof xs)
```

- Öt halmazműveletet definiálunk:

- unió (union, $S \cup T$),
- metszet (inter, $S \cap T$),
- részhalmaza-e (isSubset, $T \subseteq S$),
- egyenlők-e (isSetEq, $S = T$),
- hatványhalmaz (powerSet, pS).

- Rendezetlen listával ábrázoljuk most a halmazokat.
- Gyakorlófeladatnak meghagyjuk a halmazműveletek megvalósítását rendezett listákkal és rendezett fákkal. (A vizsgán is kaphatnak ilyen feladatokat.)

Halmazműveletek: „unió” (union) és „metszet” (inter)

- Két halmaz uniója

```
(* union : 'a list * 'a list -> 'a list
   union(xs, ys) = az xs és ys elemeiből álló halmazok uniója
*)
fun union ([], ys)      = ys
  | union (x::xs, ys) = newMem(x, union(xs, ys))
```

- Két halmaz metszete

```
(* inter : 'a list * 'a list -> 'a list
   inter(xs, ys) = az xs és ys elemeiből álló halmazok metszete
*)
fun inter ([], _)      = []
  | inter (x::xs, ys) = let val zs = inter(xs, ys)
                        in
                          if x isMem ys then x::zs else zs
                        end
```

Halmazműveletek: „részalmaz-e” (`isSubset`) és „egyenlők-e” (`isSetEq`)

- Részalmaz-e egy halmaz egy másiknak?

```
(* isSubset : 'a list * 'a list -> bool
   isSubset (xs, ys) = az xs elemeiből álló halmaz részalmaz-e
                       az ys elemeiből álló halmaznak
*)
fun op isSubset ([], _)      = true
  | op isSubset (x::xs, ys) = x isMem ys andalso
                               op isSubset(xs, ys)

infix isSubset
```

- Két halmaz egyenlősége (a listák egyenlőségvizsgálata beépített művelet az SML-ben, halmazokra mégsem használható, mert pl. $[3, 4]$ és $[4, 3]$ listaként ugyan különböznek, de halmazként egyenlők)

```
(* isSetEq : 'a list * 'a list -> bool
   isSetEq(xs, ys) = az xs elemeiből álló halmaz egyenlő-e
                    az ys elemeiből álló halmazzal
*)
fun isSetEq (xs, ys) = (xs isSubset ys) andalso (ys isSubset xs)
```

Halmazműveletek: „halmaz hatványhalmaza” (`powerSet`)

- Az S halmaz hatványhalmaza összes részalmazának a halmaza, az S -t és a $\{\}$ -t is beleértve.
- S hatványhalmaza úgy állítható elő, hogy kivesszük S -ből az x elemet, majd *rekurzív módon* előállítjuk az $S - \{x\}$ hatványhalmazát.
- Ha tetszőleges T halmazra $T \subseteq S - \{x\}$, akkor $T \subseteq S$ és $T \cup \{x\} \subseteq S$, így mind T , mind $T \cup \{x\}$ eleme S hatványhalmazának.
- Miközben a fenti elvet rekurzív módon alkalmazzuk, tehát felsorolhatjuk az $S - \{x\}$ stb. részalmazait, gyűjtjük a *már kiválasztott* elemeket. Egy-egy rekurzív lépésben a gyűjtő vagy változatlan (T), vagy kiegészül az x elemmel ($T \cup \{x\}$).
- A `pws` függvényben a `base` argumentumban gyűjtjük a halmaz *már kiválasztott* elemeit; kezdetben üres.
- $\text{pws}(xs, \text{base}) = \{S \cup \text{base} \mid S \subseteq xs\}$, azaz $xs \cup \text{base}$ azon részalmazainak a listája, amelyek teljes egészében tartalmazzák a `base` halmazt.

Halmazműveletek: „halmaz hatványhalmaza” (folyt.)

- Ezzel a pws függvény:

```
(* pws : 'a list * 'a list -> 'a list list
   pws(xs, base) = mindazon halmazok listája, amelyek előállnak xs egy
                   részalmazának és a base halmaznak az uniójaként *)
fun pws ([], base) = [base]
  | pws (x::xs, base) = pws(xs, base) @ pws(x, base)
```

- $pws(xs, base)$ valósítja meg az $S - \{x\}$ rekurzív hívást (hiszen $x :: xs$ felel meg S -nek), azaz állítja elő az összes olyan halmazt, amelyekben x nincs benne.
- $pws(xs, x :: base)$ rekurzív módon $base$ -ben gyűjti az x elemeket, vagyis előállítja az összes olyan halmazt, amelyben x benne van.
- `powerSet`-nek már csak megfelelő módon hívnia kell `pws`-t:

```
(* powerSet : 'a list -> 'a list list
   powerSet xs = az xs halmaz hatványhalmaza *)
fun powerSet xs = pws(xs, [])
```

- Példa:

```
powerSet [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,19,20];
```

Halmazműveletek: „halmaz hatványhalmaza”, hatékonyabban

- `pws` rossz hatékonyságú a kétfelé ágazó rekurzió miatt; 20 egész szám hatványalmazának előállítása már a verem túlsordulását okozza. Írjunk valamivel hatékonyabb változatot.
- Az `insAll` segédfüggvény egy elemet szűr be egy listákból álló lista minden eleme elé.

```
(* insAll : 'a * 'a list list * 'a list list -> 'a list list
   insAll(x, yss, zss) = az yss lista ys elemeinek zss elé fűzött
                       listája, amelyben minden ys elem elé x van beszúrva *)
fun insAll (x, [], zss) = zss
  | insAll (x, ys::yss, zss) = insAll(x, yss, (x::ys)::zss)
```

- `powerSet` `insAll`-t használó rekurzív változata

```
fun powerSet [] = [[]]
  | powerSet (x::xs) = let val pws = powerSet xs
                       in pws @ insAll(x, pws, []) end
```

- `powerSet` `insAll`-t használó iteratív változata

```
fun powerSet [] = [[]]
  | powerSet (x::xs) = let val pws = powerSet xs
                       in insAll(x, pws, pws) end

powerSet [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,19,20,21,22];
```

LISTÁK RENDEZÉSE

Listák rendezése FP-9-169

Listák rendezése

- **inssort** (beszúró rendezés),
- **selsort** (kiválasztó rendezés),
- **quicksort** (gyorsrendezés),
- **tmsort** (felülről lefelé haladó összefésülő rendezés),
- **bmsort** (alulról felfelé haladó összefésülő rendezés),
- **smsort** (simarendezés).

Beszűrő rendezés

- Az `ins` segédfüggvény az `x` elemet a megfelelő helyre rakja be az `ys` listában:

```
(* ins : real * real list -> real list
   ins (x, ys) = ys kibővítve x-szel a <= reláció szerint
   PRE: ys a <= reláció szerint rendezve van *)
fun ins (x, y::ys) = if x <= y then x::y::ys else y::ins(x, ys)
| ins (x : real, []) = [x]
```

- `inssort`-tal rekurzívan rendezzük a lista maradékát; végrehajtási ideje $O(n^2)$:

```
(* inssort : ('a * 'b list -> 'b list) -> 'a list -> 'b list
   inssort f xs = az xs elemeiből álló, az f felhasználásával
                 rendezett lista *)
fun inssort f (x::xs) = f(x, inssort f xs)
| inssort _ [] = [];
```

- Példa `inssort` alkalmazására:

```
inssort ins [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0] =
           [0.33, 1.12, 4.1, 4.1, 4.24, 5.67, 8.0];
```

Beszűrő rendezés, generikus változat

- Az `ins` függvényt generikussá tesszük:

```
(* ins : ('a * 'a -> bool) -> 'a * 'a list -> 'a list
   ins cmp (x, ys) = ys kibővítve x-szel a cmp reláció szerint
   PRE: ys a cmp reláció szerint rendezve van *)
fun ins cmp (x, ys) =
    let fun ins0 (y::ys) =
          if cmp(x, y) then x::y::ys else y::ins0 ys
        | ins0 [] = [x]
    in ins0 ys
    end
```

- Ezzel `inssort` egy újabb változata:

```
(* inssort : ('a * 'a -> bool) -> 'a list -> 'a list
   inssort cmp xs = az xs elemeiből álló, a cmp reláció
                   szerint rendezett lista *)
fun inssort cmp (x::xs) = ins cmp (x, inssort cmp xs)
| inssort _ [] = []
```

Beszűrő rendezés, generikus változat (folyt.)

- `inssort` eddigi változatai előbb elemeire szedik szét a rendezendő listát, majd hátulról visszafelé haladva, rendezés közben építik fel az újat.
- A jobbrekurziót és akkumulátort használó változatnak (`inssort2`) kisebb veremre van szüksége, mivel a listáról leválasztott elemeket balról jobbra haladva azonnal berakja a helyükre az eredménylistában. (A két megoldás futási idejét később összehasonlítjuk).

```
(* inssort2 : ('a * 'a -> bool) -> 'a list -> 'a list
   inssort2 cmp xs = az xs elemeiből álló, a cmp reláció
                     szerint rendezett lista *)

fun inssort2 cmp xs =
  let (* sort : 'a list -> 'a list -> 'a list
       sort xs zs = zs kibővítve az xs-nek a cmp reláció
                     szerint rendezett elemeivel
       PRE: zs cmp szerint rendezve van *)
      fun sort (x::xs) zs = sort xs (ins cmp (x, zs))
        | sort [] zs = zs
    in
      sort xs []
    end
```

Beszűrő rendezés `foldr`-rel és `foldl`-lel

- A második argumentumát akkumulátorként használó `foldl` kisebb vermet használ `foldr`-nél, ezért `inssortL` hosszabb listákat tud rendezni:

```
fun inssortR cmp = foldr (ins cmp) [];
fun inssortL cmp = foldl (ins cmp) [];
```

- Példák `inssort`-tal és `inssort2`-vel:

```
inssort op<= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0] =
              [0.33, 1.12, 4.1, 4.1, 4.24, 5.67, 8.0];
inssort2 op>= [4, 4, 5, 1, 0, 8] = [8, 5, 4, 4, 1, 0];
inssort op< (explode "qwer") = ["#e", "#q", "#r", "#w"];
```

- Példák `foldr` és `foldl` felhasználásával:

```
fun inssortRi cmp = foldr (ins cmp) [];
fun inssortLr cmp = foldl (ins cmp) ([] : real list);

inssortRi op>= [4, 4, 5, 1, 0, 8] = [8, 5, 4, 4, 1, 0];
inssortLr op<= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0] =
              [0.33, 1.12, 4.1, 4.1, 4.24, 5.67, 8.0];
```

A futási idők mérése, összehasonlítása

- 5000 elemet tartalmazó, véletlenszerű és növekvő sorrendű listák rendezéséhez szükséges futási időt mérünk.

- Véletlen eloszlású egészlistát állít elő a Random könyvtárbeli `rangelist` függvény:

```
val xs5000R =
    Random.rangelist (1, 100000) (5000, Random.newgen());
```

- Növekvő sorrendű egészlistát állít elő a `---` operátor:

```
infix ---;
fun fm --- to =
    let fun upto to zs =
            if to < fm then zs else upto (to-1) (to::zs)
        in
            upto to []
        end;
    val xs5000N = 1 --- 5000;
```

A futási idők mérése, összehasonlítása (folyt.)

- A futási időt az alábbi függvénnyel mérhetjük:

```
app load ["Timer", "Time", "Int"];

fun futIdo (sort, sortFn) (cmp, cmpFn) (xs, kind) =
    let val starttime = Timer.startCPUTimer()
        val zs = sort cmp xs
        val usr=tim,... = Timer.checkCPUTimer starttime
    in
        "Int sort with " ^ sortFn ^ ", " ^ cmpFn ^
        ", length = " ^ Int.toString(length xs) ^ " (" ^
        kind ^ "), time = " ^ Time.fmt 2 tim ^ " sec\n"
    end;

futIdo (inssort, "inssort, recursive") (op<=, "op<=") (xs5000N, "incr");
futIdo (inssort2, "inssort, iterative") (op<=, "op<=") (xs5000N, "incr");
futIdo (inssort, "inssort, recursive") (op>=, "op>=") (xs5000N, "incr");
futIdo (inssort2, "inssort, iterative") (op>=, "op>=") (xs5000N, "incr");
futIdo (inssort, "inssort, recursive") (op<=, "op<=") (xs5000R, "rand");
futIdo (inssort2, "inssort, iterative") (op<=, "op<=") (xs5000R, "rand");
futIdo (inssort, "inssort, recursive") (op>=, "op>=") (xs5000R, "rand");
futIdo (inssort2, "inssort, iterative") (op>=, "op>=") (xs5000R, "rand");
```


A futási idők mérése, összehasonlítása (folyt.)

- Egy Intel Pentium M 1.4 GHz CPU-n, linux operációs rendszer alatt a következő időket mértük:

```
Int sort with inssort, recursive, op<=, length = 5000 (incr), time = 0.00 sec
Int sort with inssort, iterative, op<=, length = 5000 (incr), time = 4.24 sec
Int sort with inssort, recursive, op>=, length = 5000 (incr), time = 4.65 sec
Int sort with inssort, iterative, op>=, length = 5000 (incr), time = 0.00 sec
Int sort with inssort, recursive, op<=, length = 5000 (rand), time = 1.96 sec
Int sort with inssort, iterative, op<=, length = 5000 (rand), time = 1.78 sec
Int sort with inssort, recursive, op>=, length = 5000 (rand), time = 1.95 sec
Int sort with inssort, iterative, op>=, length = 5000 (rand), time = 1.77 sec
```

- Jól látszik, hogy az akkumulátort nem használó (rekurzív) és az akkumulátort használó (iteratív) változatok futási ideje csak kismértékben különbözik egymástól.
- A rekurzív és az iteratív változat között lényeges különbség a veremhasználatban van.
 - A rekurzív változat már egy 500 ezer elemű listát sem tud kezelni:

```
inssort op< (1---500000);
! Uncaught exception:
! Out_of_memory
```

A futási idők mérése, összehasonlítása (folyt.)

- • Az iteratív változatnak egy 5 millió elemű listával sincs még gondja:

```
inssort2 op< (rev(1---5000000));
> val it = [1, 2, ...] : int list
```

- A futási idő rendezett lista esetén attól függ, hogy mi a listaelemek sorrendje a lista bejárás irányához képest:
 - ha a lista kezdetben megfelelő sorrendű és jobbról balra járjuk be (első rekurzív eset), akkor hátulról visszafelé haladva a már addig létrehozott eredménylista elejére kell befűzni a következő elemet, ami gyorsan megy;
 - ha a lista kezdetben megfelelő sorrendű és balról jobbra járjuk be (első iteratív eset), akkor előlről hátrafelé haladva a következő elemet a már addig létrehozott lista végére kell befűzni, ami lassan megy;
 - ha a lista kezdetben fordított sorrendű és jobbról balra járjuk be (második rekurzív eset), akkor hátulról visszafelé haladva a már addig létrehozott eredménylista végére kell befűzni a következő elemet, ami lassan megy;
 - ha a lista kezdetben fordított sorrendű és balról jobbra járjuk be (második iteratív eset), akkor előlről hátrafelé haladva a következő elemet a már addig létrehozott lista elé kell befűzni, ami gyorsan megy.

Kiválasztó rendezés

```
(* selsort : ('a * 'a -> order) -> 'a list -> 'a list
   selsort cmp xs = az xs elemei cmp szerint növekvő sorrendben
*)
fun selsort cmp xs =
  let
    (* max : 'a * 'a -> 'a
       max (x, y) = x és y közül cmp szerint a nagyobb
    *)
    fun max (x, y) = if cmp(x, y) = GREATER then x else y

    (* min : 'a * 'a -> 'a
       min (x, y) = x és y közül cmp szerint a kisebb
    *)
    fun min (x, y) = if cmp(x, y) = LESS then x else y

    (* maxSelect : 'a * 'a list * 'a list -> 'a * 'a list
       maxSelect (x, ys, zs) = pár, amelynek első tagja az
       (x::ys) cmp szerinti legnagyobb eleme, második
       tagja az x::ys többi eleméből és a zs
       elemeiből álló lista
    *)
    fun maxSelect (x, [], zs) = (x, zs)
      | maxSelect (x, y::ys, zs) =
        maxSelect(max(x, y), ys, min(x,y)::zs);
  end
```

 Deklaratív programozás. BME VIK, 2005. tavaszi félév

(Funkcionális programozás)

Listák rendezése FP-9-179

Kiválasztó rendezés (folyt.)

```
(* sSort : 'a list * 'a list -> 'a list
   sSort (xs, ws) = az xs elemei cmp szerint növekvő
   sorrendben a ws elé fűzve
*)
fun sSort ([], ws) = ws
  | sSort (x::xs, ws) =
    let val (z, zs) = maxSelect(x, xs, [])
        in
          sSort (zs, z::ws)
        end
  in
    sSort (xs, [])
  end;

app load ["Int", "Char", "Real"];

selsort Int.compare [1,2,3,4,5,6,7,8,9];
selsort Int.compare [9,8,7,6,5,4,3,2,1];
selsort Real.compare [4.5,6.7,3.6,4.3,1.2,0.9,8.9,9.8,2.0];
selsort Char.compare (explode "Ej mi a ko tyukanyo");
```

 Deklaratív programozás. BME VIK, 2005. tavaszi félév

(Funkcionális programozás)

Gyorsrendezés akkumulátor nélkül

```
(* quicksort1 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort1 : ('a * 'a -> order) -> 'a list -> 'a list *)
fun quicksort1 cmp xs =
  let (* qs : 'a list -> 'a list
       qs ys = ys elemeinek cmp szerint rendezett listája
       *)
      fun qs (m::ys) =
          let (* partition : 'a list * 'a list * 'a list -> 'a list
               partition (xs, ls, rs) = olyan pár, amelynek első tagja
               az xs m-nél kisebb elemeinek a listája ls elé fűzve,
               második tagja pedig az xs többi eleme rs elé fűzve *)
               fun partition (x::xs, ls, rs) =
                   if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                   else partition(xs, ls, x::rs)
                   | partition ([], ls, rs) = qs ls @ (m::qs rs)
               in
                   partition (ys, [], [])
               end
          | qs [] = []
      in
          qs xs
      end;
```

Deklaratív programozás. BME VIK, 2005. tavaszi félév

(Funkcionális programozás)

Listák rendezése FP-9-181

Gyorsrendezés akkumulátorral

```
(* quicksort2 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort2 : ('a * 'a -> order) -> 'a list -> 'a list *)
fun quicksort2 cmp xs =
  let (* qs : 'a list -> 'a list -> 'a list
       qs ys zs = ys elemeinek cmp szerint rendezett listája zs elé fűzve
       *)
      fun qs (m::ys) zs =
          let (* partition : 'a list * 'a list * 'a list -> 'a list
               partition (xs, ls, rs) = olyan pár, amelynek első tagja
               az xs m-nél kisebb elemeinek a listája ls elé fűzve,
               második tagja pedig az xs többi eleme rs elé fűzve *)
               fun partition (x::xs, ls, rs) =
                   if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                   else partition(xs, ls, x::rs)
                   | partition ([], ls, rs) = qs ls (m :: qs rs zs)
               in
                   partition (ys, [], [])
               end
          | qs [] zs = zs
      in
          qs xs [] end;
```

- A `List.partition` függvény használható egy lista két rész részre bontására.

Deklaratív programozás. BME VIK, 2005. tavaszi félév

(Funkcionális programozás)

A futási idők mérése, összehasonlítása

```

app load ["Listsort","Int"];
futIdo (inssort2, "inssort2") (op>=, "op>=") (xs5000R, "rand");
                                                    (* ~ 12.5 M összehasonlítás! *)
futIdo (quicksort2, "quicksort2")
      (Int.compare, "Int.compare") (xs5000R, "rand");
futIdo (Listsort.sort, "Listsort.sort")
      (Int.compare, "Int.compare") (xs5000R, "rand");
                                                    (* ~ 60 E összehasonlítás *)

Int sort with inssort2, op>=, length = 5000 (random), time = 2.34 sec
Int sort with quicksort2, Int.compare, length = 5000 (rand), time = 0.04 sec
Int sort with Listsort.sort, Int.compare, length = 5000 (rand), time = 0.04 sec

futIdo (quicksort2, "quicksort2") (Int.compare, "Int.compare")
      (Random.rangelist (1, 100000) (200000, Random.newgen()), "rand");
Int sort with quicksort2, Int.compare, length = 200000 (rand), time = 2.53 sec

futIdo (Listsort.sort, "Listsort.sort") (Int.compare, "Int.compare")
      (Random.rangelist (1, 100000) (200000, Random.newgen()), "rand");
! Uncaught exception:
! Out_of_memory

```

Összefésülő rendezések

- Az összefésülő rendezéshez kell egy olyan függvény, amely két listát növekvő sorrendben egyesít.

```

(* merge(xs, ys) = xs és ys elemeinek <= szerint
   egyesített listája
   merge : int list * int list -> int list
*)
fun merge (xxs as x::xs, yys as y::ys)=
  if x <= y
  then x::merge(xs, yys)
  else y::merge(xxs, ys)
| merge ([], ys) = ys
| merge (xs, []) = xs;

```

- Korlátot jelent, ha a részeredményeket a veremben tároljuk.
- Akkumulátor használata esetén meg kell fordítani az eredménylistát.

Föülről lefelé haladó összefésülő rendezés

- A fölülről lefelé haladó összefésülő rendezés (*top-down merge sort*) akkor hatékony, ha közel azonos hosszúságú az a két lista, amelyekre a rendezendő listát szétszedjük.

```
(* tmsort xs = az xs elemeinek a <= reláció szerint
rendezett listája
tmsort : int list -> int list
*)
fun tmsort xs = let val h = length xs
                  val k = h div 2
                in
                  if h > 1
                  then merge(tmsort(List.take(xs, k)),
                             tmsort(List.drop(xs, k)))
                  else xs
                end;
```

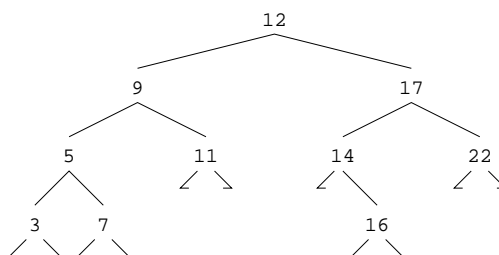
- A legrosszabb esetben $O(n \cdot \log n)$ lépésre van szükség.

Bináris fák datatype deklarációval

- A listához hasonlóan rekurzív adatszerkezet a *fa*.
- 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:

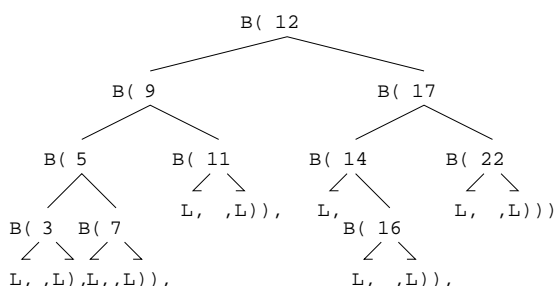


- Az 'a tree adattípus L és B adatkonstruktorokkal ez a fa pl. a következő lapon látható módon írható le.

Bináris fák datatype deklarációval (folyt.)

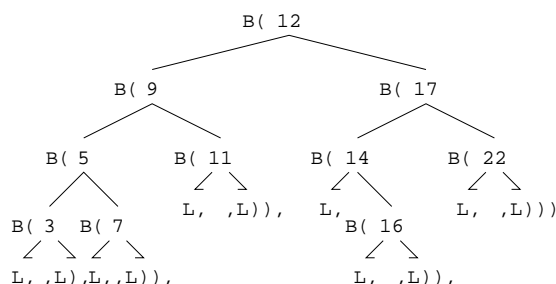
```
B(B(B(B(L, 3, L),
      5,
      B(L, 7, L)
    ),
    9,
    B(L, 11, L)
  ),
  12,
  B(B(L,
      14,
      B(L, 16, L)
    ),
    17,
    B(L, 22, L)
  )
);
```

A bal oldali kifejezést elég nehéz átlátni. A fastruktúra szöveges leírását megkönnyíti, ha az ábrába beírjuk a megfelelő adatkonstruktorokat.



Bináris fák datatype deklarációval (folyt.)

- A fastruktúra szöveges leírása átláthatóbb, ha az egyes részfáknak nevet adunk, és a részfákból építjük fel a teljes fát:



```

val tr3  = B(L,3,L);
val tr5  = B(tr3,5,tr7);
val tr9  = B(tr5,9,tr11);
val tr14 = B(L,14,tr16);
val tr17 = B(tr14,17,tr22);

val tr7  = B(L,7,L);
val tr11 = B(L,11,L);
val tr16 = B(L,16,L);
val tr22 = B(L,22,L);
val tr12 = B(tr9,12,tr17);
  
```

Bináris fák datatype deklarációval (folyt.)

- 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,
 - az értéket nem tároló üres csonkokat pedig E-vel jelölhetjük.

- A leírtak szerinti bináris fát hoz létre a következő deklaráció:

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

- A rekurzív függvényekhez hasonlóan a rekurzív adattípusok deklarációjában is kell lennie nemrekurzív ágaknak (ún. triviális esetnek).
- A nemrekurzív ág hiánya miatt az alábbi, szintaktikailag helyes deklarációk használhatatlanok:

```

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