

# POLIMORF 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 isMem (_, []) = false
  | isMem (x, y::ys) = x = y orelse isMem(x, ys)
```

```
infix isMem
```

- 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,  $pS$ ).

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

---

- Listaként kezeljük a halmazokat, később hatékonyabb ábrázolást választhatunk, pl. rendezett listát vagy bináris fát.

- Két halmaz uniója

```
(* union : 'a list * 'a list -> 'a list
   union(xs, ys) = az xs és ys elemei halmazának 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 elemei halmazának 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: „részhalmaza-e” (isSubset) és „egyenlők-e” (isSetEq)

---

- Részhalmaza-e egy halmaz egy másiknak?

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

*A hatványhalmaz megvalósítása SML-ben ezen és a következő két fólián csak olvasmány haladóknak, nem vizsgaanyag.*

- 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 felsoroltatjuk 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.
- $\text{pws}(\mathbf{xs}, \text{base}) = \{S \cup \text{base} \mid S \subseteq \mathbf{xs}\}$ , azaz  $\mathbf{xs} \cup \text{base}$  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(xs, 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, [])
```

- `pws` rossz hatékonyságú, mert kétfelé ágazó rekurziót használ. Pl. egy 19 egész számból álló lista hatványhalmazának előállítását nem lehet kivárni. Írjunk hatékonyabb változatot.

## Halmazműveletek: „halmaz hatványhalmaza”, hatékonyabban

---

- Az `insAll` segédfüggvény egy elemet szúr 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
```



# ESETSZÉTVÁLASZTÁS

---

## Esetszétválasztás (case)

```
case E of P1 => E1 | P2 => E2 | ... | Pn => En
```

Az SML-értelmező – balról jobbra és fölülről lefelé haladva – megpróbálja E-t P1-re illeszteni, ha nem sikerül, P2-re s.í.t. A case-kifejezés eredménye az E kifejezésre illeszkedő első P<sub>i</sub> mintához tartozó E<sub>i</sub> kifejezés lesz.

A case is csak szintaktikus édesítőszer, ui. helyettesíthető fn-jelöléssel:

```
(fn P1 => E1 | P2 => E2 | ... | Pn => En) E
```

Például egy függvényt lady néven így (is) definiálhatunk:

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron
```

```
(* lady : degree -> string
   lady p = p főnemes
           hitvesének rangja *)
```

```
fun lady p =
  case p of
    Duke      => "Duchess "
  | Marquis   => "Marchioness"
  | Earl      => "Countess"
  | Viscount  => "Viscountess"
  | Baron     => "Baroness"
```

```
(* lady : degree -> string
   lady p = p főnemes
           hitvesének rangja *)
```

```
fun lady p =
  (fn
    Duke      => "Duchess "
  | Marquis   => "Marchioness"
  | Earl      => "Countess"
  | Viscount  => "Viscountess"
  | Baron     => "Baroness"
  ) p
```

## AZ ORDER TÍPUS

---

## Az order típus

---

Az order típus definíciója (ld. `General.sig`)

```
datatype order = LESS | EQUAL | GREATER
```

[order] is used as the return type of comparison functions.

Példák az SML-alapkönyvtárból (SML Basis Library)

```
Int.compare      : int * int -> order
Char.compare    : char * char -> order
Real.compare    : real * real -> order
String.compare  : string * string -> order
Time.compare    : time * time -> order
```

# LISTÁK RENDEZÉSE

---

# 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 = xs elemeinek f szerint rendezett listája *)
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]
```

## 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];
inssort2 op>= [4, 4, 5, 1, 0, 8];
inssort op< (explode "qwerty")
```

- 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];
inssortLr op>= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0]
```

## A futási idők mérése, összehasonlítása

---

- 2000 elemet tartalmazó, véletlenszerűen előállított, illetve eredetileg éppen fordított sorrendű listák rendezéséhez szükséges futási időt mérünk.
- Véletlen eloszlású egészlistát állít elő a `Random.konyvtarbeli.rangelist` függvény:

```
val xs2000R =  
    Random.rangelist (1, 100000) (2000, 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 xs2000N = 1 -- 2000;
```

## A futási idők mérése, összehasonlítása (folyt.)

---

- A futási időt az alábbi függvénnyel mérhetjük:

```

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;

val t1N =
  futIdo (inssort, "inssort") (op>=, "op>=") (xs2000N, "increasing");
val t2N =
  futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000N, "increasing");
val t1R =
  futIdo (inssort, "inssort") (op>=, "op>=") (xs2000R, "random");
val t2R =
  futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000R, "random");

```

## A futási idők mérése, összehasonlítása (folyt.)

---

- **A 2000 elemű, fordított sorrendű lista rendezése az akkumulátort nem használó inssort-változatokkal több mint 5 s-ig, az akkumulátort használó változatokkal csak 0.01 s-ig tart (linux, 233 MHz-es Pentium).**

```
Int sort with inssort, op>=, length = 2000 (increasing), time = 5.18 sec
Int sort with inssort2, op>=, length = 2000 (increasing), time = 0.01 sec
Int sort with inssortRi, op>=, length = 2000 (increasing), time = 5.14 sec
Int sort with inssortLi, op>=, length = 2000 (increasing), time = 0.01 sec
```

- **Eltűnik a különbség, ha ugyanolyan hosszú, de véletlenszerűen előállított listákat rendezünk.**

```
Int sort with inssort, op>=, length = 2000 (random), time = 2.39 sec
Int sort with inssort2, op>=, length = 2000 (random), time = 2.26 sec
Int sort with inssortRi, op>=, length = 2000 (random), time = 2.40 sec
Int sort with inssortLi, op>=, length = 2000 (random), time = 2.24 sec
```

## 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

```

## 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");

```

## 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 (* qsort : 'a list -> 'a list
       qsort ys = ys elemeinek cmp szerint rendezett listája
       *)
      fun qsort (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) = qsort ls @ (m :: qsort rs)
          in
              partition (ys, [], [])
          end
      | qsort [] = []
  in
      qsort xs
  end;

```



## 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 (* qsort : 'a list -> 'a list -> 'a list
       qsort ys zs = ys elemeinek cmp szerint rendezett listája zs elé fűzve
       *)
    fun qsort (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) = qsort ls (m :: qsort rs zs)
          in
            partition (ys, [], [])
          end
        | qsort [] zs = zs
      in
        qsort xs []
      end;

```

## A futási idők mérése, összehasonlítása

---

```

val t1 = futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000R, "random");
                                           (* ~ 2 M összehasonlítás! *)
val t3 = futIdo (quicksort2, "quicksort2")
               (Int.compare, "Int.compare") (xs20000R, "random");
val t4 = futIdo (Listsort.sort, "Listsort.sort")
               (Int.compare, "Int.compare") (xs20000R, "random");
                                           (* ~ 300 E összehasonlítás *)

Int sort with inssort2, op>=, length = 2000 (random), time = 2.30 sec

Int sort with quicksort1, Int.compare, length = 20000 (random), time = 2.18 sec
Int sort with quicksort2, Int.compare, length = 20000 (random), time = 1.72 sec
Int sort with Listsort.sort, Int.compare, length = 20000 (random), time = 1.76 sec

Int sort with quicksort2, Int.compare, length = 200000 (random), time = 27.13 sec
Int sort with quicksort1, Int.compare, length = 200000 (random), time = 32.59 sec

val t7 = futIdo (Listsort.sort, "Listsort.sort") (Int.compare, "Int.compare")
              (Random.rangelist (1, 100000) (200000, Random.newgen()), "random");
! Uncaught exception:
! Out_of_memory

```

# LISTÁK RENDEZÉSE

---

## Listák rendezése (folyt.)

---

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

## 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.

## Fölülről 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.

## Alulról fölfelé haladó összefésülő rendezés

- Az alulról fölfelé haladó összefésülő rendezés (*bottom-up merge sort*) legegyszerűbb változata az eredeti  $k$  hosszúságú listát  $k$  darab egyelemű listára bontja, majd a szomszédos listákat összefuttatja, így 2, 4, 8, 16 stb. elemű listákat állít elő.
- R. O'Keefe algoritmus (1982) lépésről lépésre futtatja össze az egyforma hosszú részlistákat, de csak az utolsó lépésben rendez az összeset. Az alábbi példában az összefuttatott részlistákat *egymás mellé írással* jelöljük:

```

A B C D E F G H I J K
AB  C D E F G H I J K
AB  CD  E F G H I J K
ABCD   E F G H I J K
ABCD   EF  G H I J K
ABCD   EF  GH  I J K
ABCD   EFGH   I J K
ABCDEFGH       I J K
ABCDEFGH       IJ  K
...

```

## Alulról fölfelé haladó összefésülő rendezés (folyt.)

---

- **bmsort** a **sorting** segédfüggvényt használja, amelynek
  - első argumentuma a rendezendő lista,
  - második argumentuma a már rendezett részlisták akkumulátora,
  - harmadik argumentuma az adott lépésben összefuttatandó elem sorszáma.

```
(* bmsort xs = az xs elemeinek a <= reláció szerint
           rendezett listája
   bmsort : int list -> int list
*)
fun bmsort xs = sorting(xs, [], 0)
```



## Alulról fölfelé haladó összefésülő rendezés (folyt.)

---

- Ha a rendezendő lista ( $xs$ ) még nem fogyott el, soron következő eleméből `sorting` egyelemű listát ( $[x]$ ) képez, és ezt a már rendezett részlisták listája ( $lss$ ) elé fűzve meghívja a `mergepairs` segédfüggvényt. `mergepairs` az argumentumként átadott lista két azonos hosszúságú bal oldali részlistáját fűzi egybe, feltéve persze, hogy vannak ilyenek.  $k$  az éppen átadott elem sorszáma. Ha a rendezendő lista kiürült, `sorting` a kétszintű lista egyetlen elemét, a rendezett listát adja eredményül.

(\* `sorting(xs, lss, k)` = a még rendezetlen  $xs$  lista elemeit  
berakja a rendezett részlisták összesen  
már  $k$  elemet tartalmazó  $lss$  listájába

`sorting : int list * int list list * int -> int list`

PRE:  $k \geq 0$

\*)

```
fun sorting (x::xs, lss, k) =
    sorting(xs, mergepairs([x]::lss, k+1), k+1)
| sorting ([], lss, k) = hd(mergepairs(lss, 0))
```

## Alulról fölfelé haladó összefésülő rendezés (folyt.)

---

- `mergepairs` egyetlen listában gyűjti a már összefuttatott részlistákat. Az éppen átadott elem `k` sorszámából dönti el, hogy mit kell csinálnia a következő részlistával.

```
(* mergepairs(llss, n)= az n elemet tartalmazó, már
    rendezett llss lista első két részlistáját,
    ha egyforma a hosszuk, összefuttatja
mergepairs : int list list -> int list list
PRE: n >= 0
```

\*)

```
fun mergepairs (llss as ls1::ls2::lss, n) =
  (* legalább kételemű a lista *)
  if n mod 2 = 1
  then llss
  else mergepairs(merge(ls1, ls2)::lss, n div 2)
| mergepairs (lss, _) = lss (* egyelemű a lista *)
```

- Ha `n` páratlan, `mergepairs` a listát változtatás nélkül adja vissza, ha páros, akkor az `llss` lista elején álló két, egyforma hosszú listát egyetlen rendezett listává futtatja össze. `n=0`-ra `mergepairs` az összes listák listáját olyan listává futtatja össze, amelynek egyetlen eleme maga is lista.

## Alulról fölfelé haladó összefésülő rendezés (folyt.)

---

- A legrosszabb esetben  $O(n \cdot \log n)$  lépésre van szükség.
- A függvények működését egy példán is bemutatjuk. A kezdőhívás legyen

```
bmsort [1,2,3,4,5,6,7,8,9]
      ---> sorting ([1,2,3,4,5,6,7,8,9], [], 0)
```

- Amíg `sorting` első argumentuma a nem üres (`x::xs`) lista, `sorting` saját magát hívja meg. A rekurzív hívás
  - első argumentuma a lépésenként egyre rövidülő `xs` lista,
  - második argumentuma a `mergepairs([x]::lss, k+1)` függvényalkalmazás eredménye, ahol kezdetben `lss = []`,
  - harmadik argumentuma (`k+1`) a már feldolgozott listaelemek száma.

```
fun sorting (x::xs, lss, k) =
      sorting(xs, mergepairs([x]::lss, k+1), k+1)
| sorting ([], lss, k) = hd(mergepairs(lss, 0))
```

## Alulról fölfelé haladó összefésülő rendezés (folyt.)

---

- A következő táblázatos elrendezés
  - `mergepairs` mindkét argumentumát,
  - a rekurzív `sorting` hívás itt `j`-vel jelölt 3. argumentumát, `k+1`-et, és
  - bináris számként `k`-t mutatja lépésről lépésre.
- A `sorting` függvény hívja `mergepairs`-t azokban a sorokban, amelyekben a `j` új értéket vesz föl, a többi helyen `mergepairs` hívása rekurzív.
- Ne feledjük, hogy `mergepairs`-nek listák listája az első argumentuma!
- A táblázat utolsó oszlopa a vonatkozó magyarázatra hivatkozik.
- Vegyük észre, hogy kapcsolat van az `lss` első eleme utáni listaelemek hossza és a `k` bitjei között! Ha `k` valamelyik bitje 1, akkor (balról jobbra haladva) az `lss` megfelelő listaelemének a hossza az adott bit helyiértékével egyenlő. A 0 értékű biteknek megfelelő listaelemek „hiányoznak” `lss`-ből.

```
fun sorting (x::xs, lss, k) =
    sorting(xs, mergepairs([x]::lss, k+1), k+1)
| sorting ([], lss, k) = hd(mergepairs(lss, 0))
```

## Alulról fölfelé haladó összefésülő rendezés (folyt.)

lss	n	j	k	
[[1]]	1	1	0	m1
[[2],[1]]	2	2	1	m2
[[1,2]]	1			m3
[[3],[1,2]]	3	3	10	m3
[[4],[3],[1,2]]	4	4	11	m2
[[3,4],[1,2]]	2			m2
[[1,2,3,4]]	1			m3
[[5],[1,2,3,4]]	5	5	100	m3
[[6],[5],[1,2,3,4]]	6	6	101	m2
[[5,6],[1,2,3,4]]	3			m3
[[7],[5,6],[1,2,3,4]]	7	7	110	m3
[[8],[7],[5,6],[1,2,3,4]]	8	8	111	m2
[[7,8],[5,6],[1,2,3,4]]	4			m2
[[5,6,7,8],[1,2,3,4]]	2			m2
[[1,2,3,4,5,6,7,8]]	1			m3
[[9],[1,2,3,4,5,6,7,8]]	9	9	1000	m3
[[9],[1,2,3,4,5,6,7,8]]	0	0		m4
[[1,2,3,4,5,6,7,8,9]]				

```

fun sorting (x::xs, lss, k) =
  sorting(
    xs,
    mergepairs([x]::lss, k+1),
    k+1
  )
| sorting ([], lss, k) =
  hd(mergepairs(lss, 0))

```

m1: Az argumentumként átadott listának egyetlen eleme van (maga is lista), ezért az argumentumot mergepairs második klóza változtatás nélkül visszaadja az őt hívó sorting-nak.

m2: n páros, ez azt jelzi, hogy az argumentumként átadott lista első két eleme egyforma hosszú lista, amelyeket merge egyetlen rendezett listává futtat össze, majd az eredménnyel mergepairs első klóza meghívja saját magát.

m3: n páratlan, ez azt jelzi, hogy az argumentumként átadott lista első két eleme nem egyforma hosszú lista, ezért az argumentumot mergepairs első klóza változtatás nélkül visszaadja az őt hívó sorting-nak.

m4: n=0, az összes listák listáját olyan listává kell összefuttatni, amelynek egyetlen lista az eleme.

## Simarendezés

- Az applikatív simarendezés (*smooth sort*) algoritmus a O'Keefe alulról fölfelé haladó rendezéséhez hasonló, de nem egyelemű listákat, hanem növekvő *futamokat* állít elő.
- Ha a futamok száma  $n$ -től független, azaz a lista majdnem rendezve van, akkor az algoritmus végrehajtási ideje  $O(n)$ , és a legrosszabb esetben is legfeljebb csak  $O(n \cdot \log n)$ .

(\* nextrun (run, xs) = olyan pár, amelynek első tagja xs egy növekvő sorrendű futama, második tagja pedig xs maradéka

```
nextrun : int list * int list -> int list * int list
```

\*)

```
fun nextrun (run, x::xs) =
  if x < hd run
  then (rev run, x::xs)
  else nextrun(x::run, xs)
| nextrun (run, []) = (rev run, [])
```

- nextrun eredménye egy pár, amelynek első tagja a futam (egy növekvő számsorozat), a második tagja pedig a rendezendő lista maradéka.
- A futam csökkenő sorrendben bővül, kilépéskor a futamot meg kell fordítani.

## Simarendezés (folyt.)

---

- `smsorting` a futamokat ismételten előállítja és összefuttatja:

(\* `smsorting(xs, lss, k)` = a még rendezetlen `xs` lista elemeit berakja a rendezett részlisták összesen már `k` elemet tartalmazó `lss` listájába

```
smsorting : int list * int list list * int -> int list
```

```
PRE: k >= 0
```

```
*)
```

```
fun smsorting (x::xs, lss, k) =
    let val (run, tail) = nextrun([x], xs)
        in smsorting(tail, mergepairs(run::lss, k+1), k+1)
        end
| smsorting ([], lss, k) = hd(mergepairs(lss, 0))
```

- (\* `smsort xs` = az `xs` elemeinek `<=` szerint rendezett listája

```
smsort : int list -> int list
```

```
*)
```

```
fun smsort xs = smsorting(xs, [], 0)
```

- A simarendezés egy változata `sort` néven megtalálható a `Listsort` könyvtárban.

## A futási idők összehasonlítása

---

```

fun futIdo2 (sort, sortFn) (xs, kind) =
  let val starttime = Timer.startCPUTimer()
      val zs = sort xs
      val usr=tim,... = Timer.checkCPUTimer starttime
  in  "Int sort with " ^ sortFn ^ ", length = " ^ Int.toString(length xs) ^
      " (" ^ kind ^ "), time = " ^ Time.fmt 2 tim ^ " sec\n"
  end

val t101 = futIdo2 (tmsort, "tmsort")
           ((Random.rangelist (1, 100000) (100000, Random.newgen())), "random");
val t102 = futIdo2 (bmsort, "bmsort")
           ((Random.rangelist (1, 100000) (100000, Random.newgen())), "random");
val t103 = futIdo2 (smsort, "smsort")
           ((Random.rangelist (1, 100000) (100000, Random.newgen())), "random")

Int sort with tmsort, length = 100000 (random), time = 10.96 sec
Int sort with bmsort, length = 100000 (random), time = 7.69 sec
Int sort with smsort, length = 100000 (random), time = 7.70 sec
Int sort with quicksort2, Int.compare,
           length = 100000 (random), time = 11.98 sec
Int sort with Listsort.sort, Int.compare,
           length = 100000 (random), time = 14.17 sec

```



# **BINÁRIS FÁK**

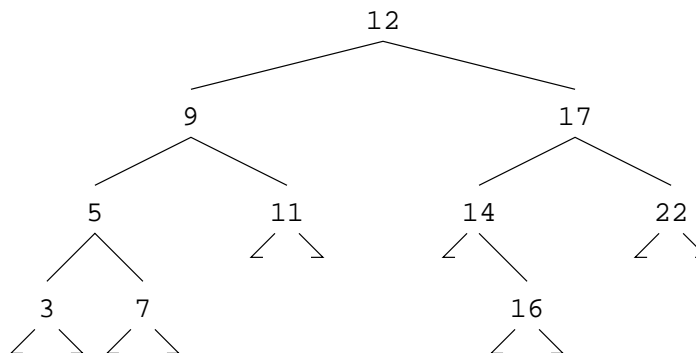


## Bináris fák datatype deklarációval

- 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 adatkonstruktoraival ez a fa pl. a következő lapon látható módon írható le.

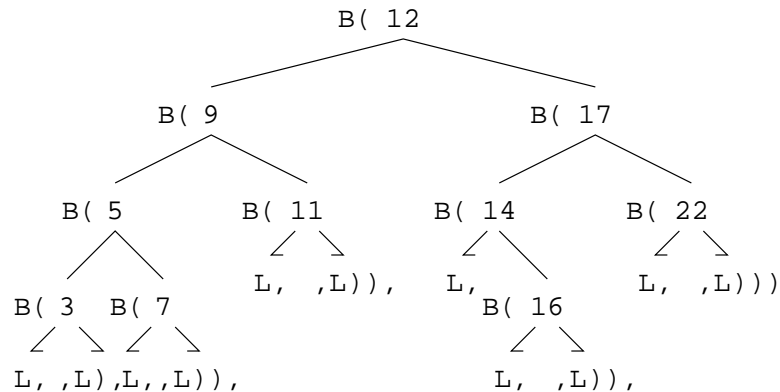
## Bináris fák datatype deklarációval (folyt.)

```

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)
  )
)

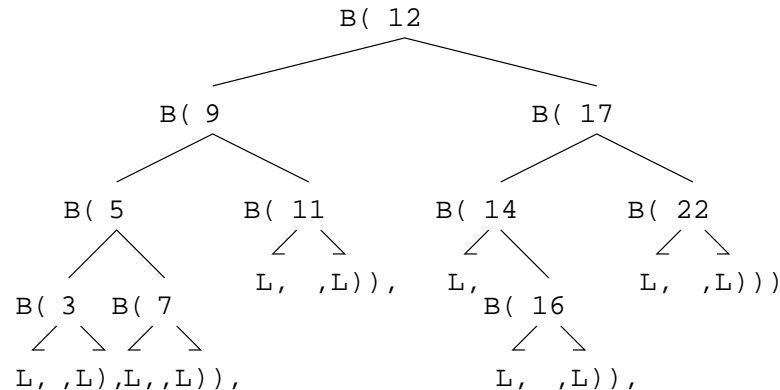
```

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
```

## Egyszerű műveletek bináris fákon

---

- `nodes` egy fa csomópontjait számlálja meg. Legyen

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

```
(* nodes : 'a tree -> int
```

```
    nodes f = az f fa csomópontjainak a száma *)
```

```
fun nodes (N(_, t1, t2)) = 1 + nodes t2 + nodes t1
```

```
  | nodes L = 0
```

- `nodes` akkumulátort használó változata (`nodesa`):

```
fun nodesa f =
```

```
    let (* nodes0(f, n) = n + a csomópontok száma f-ben
```

```
        nodes0 : 'a tree * int -> int *)
```

```
    fun nodes0 (N(_, t1, t2), n) =
```

```
        nodes0(t1, nodes0(t2, n+1))
```

```
      | nodes0 (L, n) = n
```

```
    in nodes0(f, 0)
```

```
end
```

## Egyszerű műveletek bináris fákon (folyt.)

- 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` egy fa mélységét határozza meg.

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

- `depth` akkumulátort használó változata (`deptha`):

```
fun deptha f = let fun depth0 (N(_, t1, t2), d) =
                    Int.max(depth0(t1, d+1), depth0(t2, d+1))
                  | depth0 (L, d) = d
                in
                    depth0(f, 0)
                end
```

## Egyszerű műveletek bináris fákon (folyt.)

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

```
(* fulltree : int -> 'a tree
   fulltree n = n mélységű teljes fa *)
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 : 'a tree -> 'a tree
   reflect t = a függőleges tengelye mentén tükrözött t fa *)
fun reflect L = L
    | reflect (N(v,t1,t2)) = N(v, reflect t2, reflect t1)
```



## Lista előállítás bináris fa elemeiből

- Mindhárom függvény *bináris fából listát* állít elő. Abban különböznek egymástól, hogy a tárolt értékeket milyen sorrendben veszik ki a részfák bejárása közben:
  - preorder először az értéket veszi ki, majd bejárja a bal, és azután a jobb részfát;
  - inorder bejárja a bal részfát, majd kiveszi az értéket, végül bejárja a jobb részfát;
  - postorder először bejárja a bal, majd a jobb részfát, és utoljára veszi ki az értéket.
- Az akkumulátort nem használó változatok egyszerűek, érthetőek, de nem elég hatékonyak a @ operátor használata miatt.

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

## Lista előállítás bináris fa elemeiből (folyt.)

---

- Ha `inorder` előző változatában az `inorder t1 @ (v :: inorder t2)` kifejezésben a `v :: inorder t2` részkifejezést nem tesszük zárójelbe, a fordító hibát jelez, mivel `::` és `@` azonos precedenciájú, és ezért zárójelek nélkül a nyilvánvalóan hibás `inorder t1 @ v` részkifejezést akarná kiértékelni.
- `inorder` előző megvalósításával kb. egyenértékű a következő változata, amelyben a `v` elem helyett az egyelemű `[v]` listát fűzzük `inorder t2` elé:

```
fun inorder L = []
  | inorder (N(v,t1,t2)) = inorder t1 @ ([v] @ inorder t2)
```

Ez a változat azonban *roppant sérülékeny*, ugyanis a hatékonysága függ a zárójelek kirakásától. Ha a `[v] @ inorder t2` részkifejezést nem tesszük zárójelbe, akkor a fordító először a `inorder t1 @ [v]` részkifejezést fogja kiértékelni, azaz egy egyelemű listához fűz egy (általában) jóval hosszabbat!

## Lista előállítása bináris fa elemeiből (folyt.)

---

- Az elmondottakhoz hasonló okból postorder bemutatott változata is *rendkívül sérülékeny!* Ha ugyanis a `postorder t1 @ (postorder t2 @ [v])` kifejezésben az amúgyis rossz hatékonyságú `postorder t2 @ [v]` részkifejezést nem tesszük zárójelbe, akkor a fordító először a `postorder t1 @ postorder t2` részkifejezést értékeli ki, azaz a két, feltehetően hosszú listát fűzi egybe, majd a létrehozott eredménylistát fűzi az egyelemű listához!

## Lista előállítás bináris fa elemeiből (folyt.)

---

Az akkumulátort használó változatok nehezebben érthetőek meg, de *hatékonyabbak*, elsősorban a veremhasználat szempontjából.

```
(* preord : 'a tree * 'a list -> 'a list
   preord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                   preorder sorrendű listája *)
fun preord (L, vs) = vs
  | preord (N(v,t1,t2), vs) = v::preord(t1, preord(t2,vs))

(* inord : 'a tree * 'a list -> 'a list
   inord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                  inorder sorrendű listája *)
fun inord (N(v,t1,t2), vs) = inord(t1, v::inord(t2,vs))
  | inord (L, vs) = vs

(* postord : 'a tree * 'a list -> 'a list
   postord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                    postorder sorrendű listája *)
fun postord (N(v,t1,t2), vs) = postord(t1, postord(t2, v::vs))
  | postord (L, vs) = vs
```

## Bináris fa előállítása lista elemeiből: `balPreorder`

---

- Listát *kiegyensúlyozott* (balanced) *bináris fává* alakítanak a következő függvények: `balPreorder`, `balInorder` és `balPostorder`; a különbség közöttük most is a bejárási sorrendben van.
- (\* `balPreorder: 'a list -> 'a tree`  
`balPreorder xs = az xs lista elemeiből álló, preorder`  
`bejárású, kiegyensúlyozott fa`  
 \*)  

```

fun balPreorder [] = L
  | balPreorder (x::xs) =
    let val k = length xs div 2
    in
      N(x, balPreorder(List.take(xs, k)),
        balPreorder(List.drop(xs, k)))
    end
  
```
- A hatékonyságot kisebb mértékben rontja, hogy `List.take` és `List.drop` egymástól függetlenül *kétszer* mennek végig a lista első felén.

## take és drop egyetlen függvénnyel: take'ndrop

---

- Írjunk take'ndrop néven olyan függvényt, amelynek egy xs listából és egy k egészből álló pár az argumentuma, és egy olyan pár az eredménye, amelynek első tagja a lista első k db eleme, második tagja pedig a lista többi eleme.

```
(* take'ndrop : 'a list * int -> 'a list * 'a list
   take'ndrop(xs, k) = olyan pár, amelynek
                       első tagja xs első k db eleme,
                       második tagja pedig xs maradéka
```

```
*)
```

```
fun take'ndrop (xs, k) =
    let fun td (xs, 0, ts) = (rev ts, xs)
        | td ([], _, ts) = (rev ts, [])
        | td (x::xs, k, ts) = td(xs, k-1, x::ts)
    in
        td(xs, k, [])
    end
```

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

## Bináris fa előállítás a lista elemeiből: `balPreorder`, újra

---

- Ez volt:

```
fun balPreorder [] = L
  | balPreorder (x::xs) =
    let val k = length xs div 2
    in N(x, balPreorder(List.take(xs, k)),
        balPreorder(List.drop(xs, k)))
    end
```

- Ez lett:

```
(* balPreorder: 'a list -> 'a tree
   balPreorder xs = az xs lista elemeiből álló, preorder ... *)
fun balPreorder [] = L
  | balPreorder (x::xs) =
    let val k = length xs div 2
    in N(x, balPreorder (take'ndrop(xs, k)
        val (ts, ds) = take'ndrop(xs, k)
    in N(x, balPreorder ts, balPreorder ds)
    end
```

## Bináris fa előállítás a lista elemeiből

---

- ```

(*) (* balInorder: 'a list -> 'a tree
    balInorder xs = az xs lista elemeiből álló, inorder bejárású,
                    kiegyensúlyozott fa
*)
fun balInorder [] = L
  | balInorder (x::xs) =
    let val k = length xs div 2
        val ys = List.drop(x, k)
    in
      N(hd xs, balInorder(List.take(x, k)),
        balInorder(tl xs))
    end

```
- ```

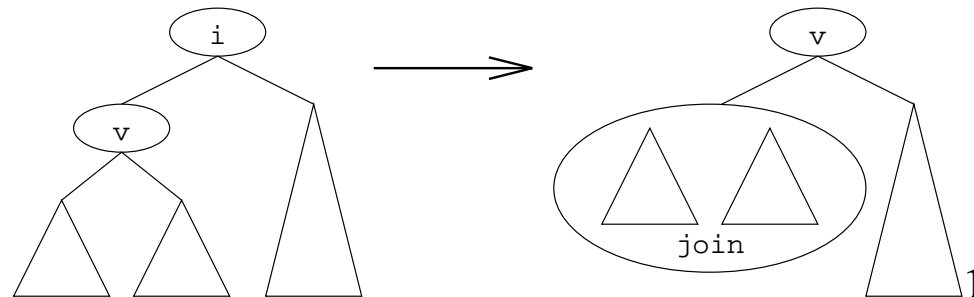
(*) (* balPostorder: 'a list -> 'a tree
    balPostorder xs = az xs lista elemeiből álló, postorder
                    bejárású, kiegyensúlyozott fa
*)
fun balPostorder xs = balPreorder(rev xs)

```
- balInorder take'ndrop-pal való definiálását meghagyjuk gyakorló feladatnak.**



## Elem törlése bináris fából

- 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.
- 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 *egyesíteni* kell, miután a törlést a két részfán már végrehajtottuk.



- Megtehetjük, hogy előbb egyesítjük a két részfát, majd az eredményül kapott fából töröljük az adott értékű elemet.

## Elem rekurzív törlése bináris fából (folyt.)

- A `join`-nal egyesítjük a törlés hatására létrejövő két részfát: a bal részfát lebontja, és közben az elemeit egyesével berakja a jobb részfába.

```
(* join : 'a tree * 'a tree -> 'a tree
   join(b, j) = a b és a j fák egyesítésével létrehozott fa *)
fun join (L, tr) = tr
  | join (N(v, lt, rt), tr) = N(v, join(lt, rt), tr)
```

- A `remove` rendezetlen bináris fából törli az `i` értékű elem összes előfordulását.

```
(* remove : 'a * 'a tree -> 'a tree
   remove(i, f) = i összes előfordulását törli f-ből *)
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))
```

## Bináris keresőfák: `blookup`, `binsert`

---

- Rendszerint adott kulcsú elemet keresünk egy rendezett bináris fában, ehhez értékeket kell összehasonlítanunk egymással, ehhez a keresett kulcsnak *egyenlőségi típusúnak* kell lennie (a példában a `string` típust használjuk).
- A függvények *kivételt* jeleznek, ha a keresett kulcsú elem nincs a keresőfában: `exception Bsearch of string`.
- A `blookup` függvény adott kulcshoz tartozó értéket ad vissza:

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

## Bináris keresőfák: `bupdate`

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

```
(* binsert : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   binsert(f, (b,y)) = az új (b,y) kulcs-érték párral bővített f fa *)
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 : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   bupdate(f, (b,y)) = az f fa, a b kulcshoz tartozó érték helyén
                       az y értékkel *)
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)
```

- A függvények *generikussá* tételét meghagyjuk gyakorló feladatnak.

# EGYIDEJŰ DEKLARÁCIÓ, ÚJBÓL

---

## Egyidejű deklaráció, újból

---

- Nemcsak értékek, típusok is deklaráálhatók *egyidejűleg* az `and` kulcsszó alkalmazásával.
- Vegyük a következő deklarációsorozatokat:

```
type sor = int; type osz = int;
datatype fa = L | B of fa * fa;
  datatype 'a verem = > | | >> of 'a * 'a verem;
val v1 = "a"; val v2 = "z";
fun f1 i = i + 1; fun f2 i = i - 1;
```

Ezeket a deklarációkat az SML-értelmező a *megadott sorrendben* értékeli ki.

```
type sor = int and osz = int;
datatype fa = L | B of fa * fa and
  'a verem = > | | >> of 'a * 'a verem;
val v1 = "a" and v2 = "z";
fun f1 i = i + 1 and f2 i = i - 1;
```

Az `and` szócskával elválasztott deklarációkat az SML-értelmező *egyidejűleg* értékeli ki.

## Egyidejű deklaráció, újból (folyt.)

---

- Egyidejű deklarációt kell használnunk kölcsönösen rekurzív függvények definiálására.

Példa:

```
fun even 0 = true | even n = odd(n-1)
and odd 0 = false | odd n = even(n-1);
```

- Egyidejű deklarációt használhatunk két vagy több kötés egyidejű felcserélésére. Példa:

```
val v1 = "a"; val v2 = "z"; val v1 = v2 and v2 = v1;
```

- Egyidejű deklarációt használhatunk, ha fölülről lefelé haladva akarunk programot írni.

Példa:

```
fun length zs = len zs 0
and len [] i = i | len (_ :: xs) i = len xs (i+1);
```

## Egyidejű deklaráció, újból (folyt.)

---

- A polimorf függvényeket a szekvenciális és az egyidejű deklaráció eltérően kezeli, mivel a típuslevezetést az SML-értelmező a teljes kifejezésre alkalmazza. Példa:

```
fun id x = x; fun hi () = id 3; fun nr () = id 4.0;  
fun id x = x and hi () = id 3 and nr () = id 4.0;
```

Az első sor kiértékelésekor `id 'a -> 'a` típusú. A második sor kiértékelésekor az `id` név `int -> int` és `real -> real` típusú lenne egyszerre, ami lehetetlen.



# FÜGGVÉNYEK KOMPOZÍCIÓJA

---

## Függvények kompozíciója

- Az  $f \circ g$  függvénykompozíció az SML-ben

(\* f o g = az f és g függvények kompozíciója \*)

```
infix 2 o;
```

```
fun (f o g) = fn x => f(g x); vagy
```

```
fun (f o g) x = f(g x);
```

- Az  $\circ$  típusa  $? * ? \rightarrow ?$  szerkezetű. Mit írjunk a  $?$ -ek helyébe? Vezessük le!

- A függvénydefiníció jobb oldalán álló kifejezés elemzésével kezdjük.

$x : 'a$        $g : 'a \rightarrow 'b$        $(g\ x) : 'b$        $f : 'b \rightarrow 'c$

- A `fun (f o g) x = f(g x)` függvénydefinícióban az egyenlőségjel (=) bal és jobb oldalán álló kifejezéseknek azonos értéket kell eredményül adniuk, ezért  $f \circ g$  és  $f$  eredményének azonos a típusa (azaz  $'c$ ).

$(f \circ g) : 'a \rightarrow 'c$      $o : ('b \rightarrow 'c) * ('a \rightarrow 'b) \rightarrow ('a \rightarrow 'c)$

- Példa: `round : real -> int, chr : int -> char`  
`chr o round : real -> char`