

LISTÁK RENDEZÉSE



Listák rendezése (folyt.)

- `insort` (beszúró rendezés),
- `selsort` (kiválasztó rendezés),
- **`quicksort` (gyorsrendezés),**
- **`tmsort` (fölről lefelé haladó összefésülő rendezés),**
- `bmsort` (alulról fölfelé haladó összefésülő rendezés),
- `smsort` (simarendezés).

Quicksort algoritmus:

$$\text{qs}(m::xs) = \text{qs}([\dots, x_i, \dots] \mid x_i < m) @ [m] @ \text{qs}([\dots, x_j, \dots] \mid x_j \geq m), x_i, x_j \in xs$$
$$\text{qs} [] = []$$

Ismeretes, hogy a Quicksort egy n elemű sorozatot $O(n \cdot \log n)$ lépésben rendez.

Gyorsrendezés akkumulátor nélkül

```

(* quicksort1 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort1 : ('a * 'a -> order) -> 'a list -> 'a list
   qs ys = ys elemeinek cmp szerint rendezett listája
   qs : '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
   partition : 'a list * 'a list * 'a list -> 'a list *)
fun quicksort1 cmp xs =
  let
    fun qs (m::ys) =
      let
        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;

```

Gyorsrendezés akkumulátorral

```

(* quicksort2 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort2 : ('a * 'a -> order) -> 'a list -> 'a list
   qs ys zs = ys elemeinek cmp szerint rendezett listája zs elé fűzve
   qs : '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
   partition : 'a list * 'a list * 'a list -> 'a list *)
fun quicksort2 cmp xs =
  let
    fun qs (m::ys) zs =
      let
        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;

```

Gyorsrendezés akkumulátorral, `List.partition`-nal

- Egy lista két részre bontására a `List.partition` is függvény használható.

- `List.partition : ('a -> bool) -> 'a list -> ('a list * 'a list)`

(* quicksort3 cmp xs = az xs elemeinek cmp szerint rendezett listája

quicksort3 : ('a * 'a -> order) -> 'a list -> 'a list

qs ys zs = ys elemeinek cmp szerint rendezett listája zs elé fűzve

qs : 'a list -> 'a list -> 'a list

*)

```
fun quicksort3 cmp xs =
```

```
  let
```

```
    fun qs (m::ys) zs =
```

```
      let
```

```
        val (ls,rs) = List.partition (fn x => cmp(x,m)=LESS) ys
```

```
      in
```

```
        qs ls (m :: qs rs zs)
```

```
      end
```

```
    | qs [] zs = zs
```

```
  in
```

```
    qs xs []
```

```
end;
```

A futási idők mérése és összehasonlítása

Ha mosml használunk, be kell tölteni a szükséges programkönyvtárakat:

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

A futási idő mérésére és kiírására futIdo alábbi változatát használjuk:

```
(* futIdo (sort, sortFn, cmpFn) (xs, kind) = megméri és kiírja a képernyőre
      sort xs futási idejét a függvény, a reláció, és az xs lista
      jellemzőivel együtt
   futIdo : ('a list -> 'b) * string * string -> 'a list * string -> unit
*)
fun futIdo (sort, sortFn, cmpFn) (xs, kind) =
  let val starttime = Timer.startCPUTimer()
      val zs = sort xs
      val {usr=tim,...} = Timer.checkCPUTimer starttime
  in
    print ("Int sort with " ^ sortFn ^ ", " ^ cmpFn ^
           ", length = " ^ Int.toString(length xs) ^ " (" ^
           kind ^ "), time = " ^ Time.fmt 2 tim ^ "s\n")
  end;
```

Megjegyzés: sort néven egy n elemű listát legfeljebb $O(n \cdot \log n)$ lépésben rendező függvényt találunk az mosml Listsort, ill. az smlnj ListMergeSort könyvtárában.

A futási idők mérése és összehasonlítása (folyt.)

10000 elemet tartalmazó rendezetlen egészlistával mérjük a futási időket:

```
val xs10000R = Random.rangelist (1, 100000) (10000, Random.newgen());
  (* kb. 25 M összehasonlítás! *)
futIdo (inssort2 op>=, "inssort2", "op>=") (xs10000R, "rand");

  (* kb. 130 E összehasonlítás *)
futIdo (quicksort1 Int.compare, "quicksort1", "Int.compare") (xs10000R, "rand");
futIdo (quicksort2 Int.compare, "quicksort2", "Int.compare") (xs10000R, "rand");
futIdo (quicksort3 Int.compare, "quicksort3", "Int.compare") (xs10000R, "rand");

  (* kb. 130 E összehasonlítás *)
futIdo (Listsort.sort Int.compare, "Listsort.sort", "Int.compare")
                                             (xs10000R, "rand");
```

Az mosml válaszai:

Int sort with inssort2, op>=, length = 10000 (random), time = 9.49s

Int sort with quicksort1, Int.compare, length = 10000 (rand), time = 0.10s

Int sort with quicksort2, Int.compare, length = 10000 (rand), time = 0.08s

Int sort with quicksort3, Int.compare, length = 10000 (rand), time = 0.09s

Int sort with Listsort.sort, Int.compare, length = 10000 (rand), time = 0.08s

A futási idők mérése és összehasonlítása (folyt.)

Vizsgáljuk meg mosml alatt – inssort2 kivételével – a többi rendező algoritmus viselkedését 200000 elemet tartalmazó rendezetlen egészlistára is:

```
val xs200000R = Random.rangelist (1, 100000) (200000, Random.newgen());
futIdo (quicksort1 Int.compare, "quicksort1", "Int.compare") (xs200000R, "rand");
futIdo (quicksort2 Int.compare, "quicksort2", "Int.compare") (xs200000R, "rand");
futIdo (quicksort3 Int.compare, "quicksort3", "Int.compare") (xs200000R, "rand");
```

Az mosml válaszai:

```
Int sort with quicksort1, Int.compare, length = 200000 (rand), time = 3.44s
Int sort with quicksort2, Int.compare, length = 200000 (rand), time = 2.37s
Int sort with quicksort3, Int.compare, length = 200000 (rand), time = 3.29s
```

Vajon mit kezd Listsort.sort egy 200000 elemből álló listával?

```
futIdo (Listsord.sort Int.compare, "Listsord.sort", "Int.compare")
                                             (xs200000R, "rand");
```

Az mosml válasza:

```
! Uncaught exception:
! Out_of_memory
```

Ejnye!

Összefésülő rendezések

- Ebben a részben csak egészeket tartalmazó listák rendezésével foglalkozunk.
- A függvényeket generikussá tehetjük, ha az összehasonlító műveletet paraméterként adjuk át.
- Az összefésülő rendezéshez kell egy olyan függvény, amely két listát növekvő sorrendben egyesít.

(* merge(xs, ys) = xs és ys elemeinek <= szerint egyesített listája

PRE: xs, ys <= szerint rendezve vannak

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, hogy a részeredményeket a veremben tároljuk.
- Akkumulátor használata esetén meg kell fordítani az eredménylistát vagy az összefésülendő listákat.

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.

BINÁRIS FÁK

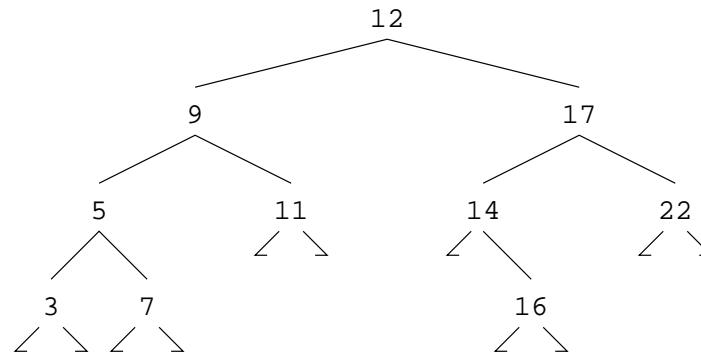


Bináris fák datatype deklarációval

- 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.

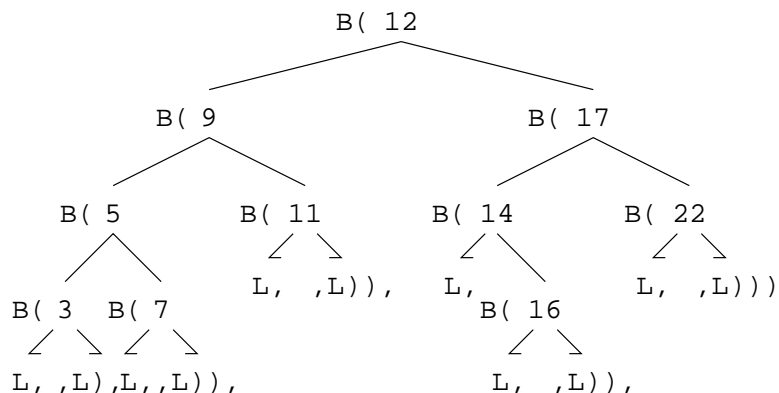
Bináris fák datatype deklarációval (folyt.)

```

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)
  )
);

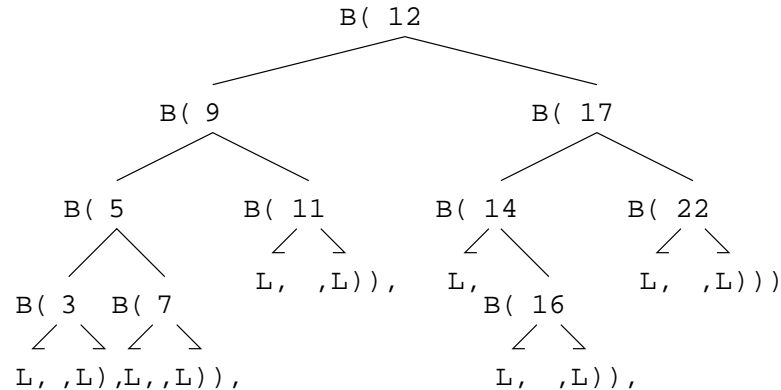
```

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.

- kezdhethetjük az 'a típusú értékkel, majd folytathatjuk előbb a bal, azután a jobb részfa megadásával:

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

- felhasználhatjuk a levelet is értékek tárolására:

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

- az értéket nem tároló üres csomópontokat pedig E-vel jelölhetjük:

```
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 ágak (a rekurzió leállításához).
- 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

Legyen

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

- nodes egy fa csomópontjait számlálja meg.

```
(* 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
```

- Akkumulátort használó változata: nodesa – a kétfelé ágazó rekurzió miatt nem nyerünk vele szinte semmit, de sikerült kevésbé érthetővé tennünk. :-)

```
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` – a kétfelé ágazó rekurzió miatt most sem nyerünk vele szinte semmit.

```
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.

```
(* fulltree : int -> int 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
```

Egy teljes bináris fában minden csomópontból pontosan két él indul ki, és minden levelének ugyanaz a szintje.

- `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, hogy mikor veszik ki a csomópontokban tárolt értékeket, és milyen sorrendben járják be a részfákat:
 - `preorder` először az értéket veszi ki, majd bejárja a bal, és azután a jobb részfát;
 - `inorder` 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 akkumulátort nem használó változatok egyszerűek, érthetőek, de nem elég hatékonyak a `@` operátor használata miatt.

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

- 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 megint nehezebben érthetőek, de talán valamivel *hatékonyabbak*, főleg a veremhasználat szempontjából (@ helyett ::-ot használunk).

```
(* 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 előállítás a lista elemeiből: `balPreorder`

- Listát *kiegyensúlyozott* (**balanced**) *bináris fává* alakítanak a következő függvények: `balPreorder`, `balInorder` és `balPostorder`; a különbség közöttük most is a bejárás 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
 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(xxs, k)
    in
      N(hd ys, balInorder(List.take(xxs, k)),
        balInorder(tl ys))
    end

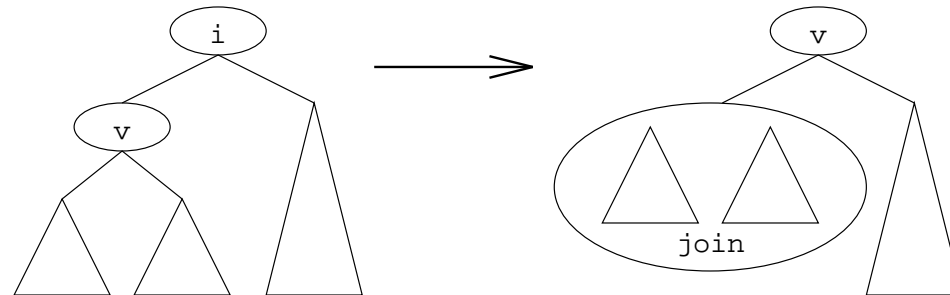
```
- ```

(*) (* 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 rekurzív 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 szétcsúszó 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 részfat: a bal részfat lebontja, és közben az elemeit egyesével berakja a jobb részfatba.

```
(* 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.

# ABSZTRAKCIÓ FÜGGVÉNYEKSEL (ELJÁRÁSOKKAL)

---

## Legnagyobb közös osztó

- Fontos, hogy algoritmusaink futási idejét és tárigényét ismerjük – különösen korlátos erőforrású (pl. beágyazott) rendszerekben fontos ez.
- Következő példánk  $a$  és  $b$  legnagyobb közös osztóját számolja ki az euklideszi algoritmussal.
- Az alapgondolat az, hogy ha  $a$ -t  $b$ -vel osztva  $r$  a maradék, akkor  $a$  és  $b$  közös osztói azonosak  $b$  és  $r$  közös osztóival.
- A matematikai definíciót most is pontosan követi az SML-függvény.

$$\begin{array}{l|l} \text{gcd}(a, 0) = a & \text{fun gcd (a, 0) = a} \\ \text{gcd}(a, b) = \text{gcd}(b, a \bmod b) & \text{| gcd (a, b) = gcd(b, a mod b)} \end{array}$$

- A függvény *rekurzív*, a *folyamat* iteratív. A lépések száma logaritmikusan nő – miért is?

Az ún. *Lamé-tétel* szerint – ld. SICP, 1.2.5. szakasz – ha az euklideszi algoritmus egy számpár legnagyobb közös osztóját  $k$  lépésben számítja ki, akkor a számpár kisebbik tagja nem lehet kisebb a  $k$ -adik Fibonacci-számnál.

Legyen  $n$  az algoritmus kisebbik paramétere. Ha a legnagyobb közös osztó kiszámításához  $k$  lépésre van szükség, akkor  $n \geq F(k) \approx \Phi^k / \sqrt{5}$ . A  $k$  lépésszám tehát valóban az  $n - \Phi$  alapú – logaritmusával arányos.

Idézzük föl, hogy  $\Phi = (1 + \sqrt{5})/2 \approx 1.61803$  az *arany metszés* arányszáma.

## Prímteszt

---

- A `prime` predikátum egy  $n$  szám prím voltát teszteli. A `findDivisor` függvény 2-től kezdve megkeresi az  $n$  szám legkisebb osztóját. Az  $n$  szám prím, ha legkisebb osztója maga az  $n$ .
- Az  $n$  osztóit 2-től  $\sqrt{n}$ -ig kell keresni, így a lépések száma  $O(\sqrt{n})$ .

```
(* prime n = true if n is prime
 prime : int -> bool
*)
fun prime n =
 let
 infix divides
 fun smallestDivisor n = findDivisor(n, 2)
 and findDivisor (n, testDivisor) =
 if square testDivisor > n
 then n
 else if testDivisor divides n
 then testDivisor
 else findDivisor(n, testDivisor+1)
 and square x = x * x
 and a divides b = b mod a = 0
 in
 n = smallestDivisor n
 end;
```



## Prímteszt (folyt.)

---

- A következő SML-predikátum egy szám prím voltát *valószínűségi módszerrel* teszteli. A lépések száma  $O(\lg n)$ .
- Az algoritmus a kis Fermat-tételre alapul, amely azt mondja ki, hogy:  
ha  $n$  prím és  $0 < a < n$ , akkor  $a^n$  modulo  $n$  szerint *kongruens*  $a$ -val, azaz  $a^n \bmod n = a$  (ld. pl. SICP, 1.2.6. szakasz).
  - Két szám akkor *kongruens* modulo  $n$  szerint, ha  $n$ -nel osztva mindkettőnek ugyanaz a maradéka. Egy  $a$  szám  $n$ -nel való osztásának maradékát  $a$  modulo  $n$  szerinti maradékának, vagy röviden  $a$  modulo  $n$ -nek is nevezik.
- Ha  $n$  nem prím, akkor az  $a < n$  számok nagy hányadára nem teljesül a fenti reláció.
- A prímteszt algoritmusában ezek után a következő :
  - Adott  $n$ -re véletlenszerűen válasszunk egy  $a < n$  számot: ha  $a^n \bmod n \neq a$ , akkor  $n$  nem prím. Ellenkező esetben nagy a valószínűsége, hogy  $n$  prím.
  - Válasszunk véletlenszerűen egy másik  $a < n$  számot: ha  $a^n \bmod n = a$ , akkor növekedett annak a valószínűsége, hogy az  $n$  prím. Újabb és újabb  $a$  értékeket választva egyre biztosabbak lehetünk abban, hogy az  $n$  prím.

## Prímteszt (folyt.)

---

- Az `expmod` segédfüggvény a `base` szám `exp`-edik hatványának modulo `m` szerinti maradékát adja eredményül.

```
(* expmod (base, exp, m) = base exp-edik hatványa modulo m
 expmod : int * int * int -> int
```

```
*)
```

```
fun expmod (_, 0, _) = 1
 | expmod (b, e, m) =
 if even e
 then square(expmod(b, e div 2, m)) mod m
 else b * expmod(b, e-1, m) mod m
and even n = n mod 2 = 0
and square x = x * x;
```

- Nagyon hasonló felépítésű `exptFast`-hoz. A lépések száma a kitevő logaritmusával arányos.

## Prímteszt (folyt.)

---

- Ha `mosml`-t használunk, betöltjük a `Random` könyvtárat:

```
load "Random" ;
```

- `fermatTest` generál egy álvéletlen-számot, és egyszer elvégzi a vizsgálatot:

```
(* fermatTest n = false if n is not prime
 fermatTest : int -> bool
*)
fun fermatTest n =
 let fun tryIt a = expmod(a, n, n) = a
 in tryIt(Random.range (1, n) (Random.newgen()))
 end;
```

- `fastPrime times`-szor megismétli a vizsgálatot:

```
(* fastPrime (n, times) = true if n passes the prime test times times
 fastPrime : int * int -> bool
*)
fun fastPrime (n, 0) = true
 | fastPrime (n, t) = fermatTest n andalso fastPrime(n, t-1);
```

- Ez a megoldás csak nagy valószínűséggel, de nem teljes bizonyossággal ad választ a kérdésre. Például az 561 átmegy a Fermat-teszten, bár nem prím.

## Függvények mint általános számítási módszerek

---

- Láttuk, hogy a függvény (ill. általában az eljárás) olyan *absztrakció*, amely – a paraméterként átadott adatok konkrét értékétől függetlenül – összetett műveleteket ír le.
- Az olyan magasabbrendű függvény, amelynek függvény a paramétere, még *magasabb szintű* absztrakció, hiszen az általa megvalósított összetett műveletet nemcsak egyes konkrét adatoktól, hanem egyes konkrét műveletektől is függetlenné tesszük.
- A magasabbrendű függvény (eljárás) tehát valamilyen *általános számítási módszert* fejez ki.
- A következő lapokon két nagyobb példát ismertetünk a SICP alapján: általános számítási módszert függvények *zérushelyeinek* és *fixpontjának* a megtalálására.

## Egyenlet gyökeinek meghatározása intervallumfelezéssel

---

- Az intervallumfelezés módszere hatékony eljárás az  $f(x) = 0$  egyenlet gyökeinek megtalálására, ahol  $f$  folytonos függvény.
- A közismert alapötlet a következő:
  - Megfelelően megválasztott  $a$ -ra és  $b$ -re, amelyekre  $f(a) < 0 < f(b)$ ,  $f$ -nek legalább egy zérushelye van  $a$  és  $b$  között.
  - A zérushely megtalálásához legyen  $x = (a + b)/2$ . Ha  $f(x) > 0$ , akkor  $f$  zérushelyét  $a$  és  $x$  között, ha  $f(x) < 0$ , akkor  $x$  és  $b$  között kell keresnünk.
  - A keresést – a rekurziót – akkor hagyjuk abba, amikor két egymás utáni közelítő érték *eltérése* egy előre meghatározott értéknél kisebb lesz.
- Mivel az eltérés minden lépésben a felére csökken, az  $f$  zérushelyének megtalálásához szükséges lépések száma  $O(L/T)$ , ahol  $L$  az intervallum hossza kezdetben, és  $T$  a megengedett eltérés.
- A leírt algoritmust valósítja meg a `search` függvény (ld. a következő lapon):

## Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

---

```
(* search (f, negPoint, posPoint) = root of f x in the
 negPoint <= x <= posPoint interval
 PRE: f negPoint <= 0 and f posPoint >= 0
 search : (real -> real) * real * real -> real
*)
fun search (f, negPoint, posPoint) =
 let val midPoint = average(negPoint, posPoint)
 in
 if closeEnough(negPoint, posPoint)
 then midPoint
 else let val testValue = f midPoint
 in
 if positive(testValue)
 then search(f, negPoint, midPoint)
 else if negative(testValue)
 then search(f, midPoint, posPoint)
 else midPoint
 end
 end
 end
and average (x, y) = (x+y)/2.0
and closeEnough (x, y) = abs(x-y) < 0.001
and positive x = x > 0.0
and negative x = x < 0.0;
```

## Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

---

- Az előfeltételek betartását célszerű `search` alkalmazása előtt ellenőrizni, nehogy rossz választ kapjunk az SML értelmezőtől.

```
load "Math";
```

- Helyes az eredménye:

```
search(Math.sin, 4.0, 2.0);
```

```
> val it = 3.14111328125 : real
```

- **Hibás** az eredménye:

```
search(Math.sin, 2.0, 4.0);
```

```
> val it = 2.00048828125 : real
```

- A `halfIntervalMethod` függvény (ld. a következő lapon) elvégzi az ellenőrzést, és jelzi, ha `negPoint` vagy `posPoint` kezdeti értéke nem jó.
- Figyeljük meg az *ügyek szétválasztása* elv alkalmazását: `search` a gyökkeresési stratégiát valósítja meg, `halfIntervalMethod` pedig az előfeltételek meglétét ellenőrzi.

## Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

```

(*) (* halfIntervalMethod (f, a, b) = root of f x in the a <= x <= b interval
 halfIntervalMethod : (real -> real) * real * real -> real
*)
fun halfIntervalMethod(f, a, b) =
 let val (aValue, bValue) = (f a, f b)
 in
 if negativeOrZero aValue andalso positiveOrZero bValue
 then search(f, a, b)
 else if negativeOrZero bValue andalso positiveOrZero aValue
 then search(f, b, a)
 else print ("Values " ^ makestring a ^ " and " ^
 makestring b ^ " are not of opposite sign.\n")
 end
and positiveOrZero x = x >= 0.0
and negativeOrZero x = x <= 0.0;

```

- A függvénynek ez a változata hibás, mert az `if-then-else` feltételes kifejezés összes ágának *ugyanolyan típusú* eredményt *kell* adnia, márpedig `print` eredménye nem `int` típusú.
- (Csúnya) megoldás az `(e; f)` alakú *szekvenciális kifejezés* használata: az értelmező kiértékeli `e`-t és `f`-et a felírt sorrendben, eredményül pedig `f` értékét adja (ld. a következő lapon).
- Szébb, ha *kivételt* jelzünk: ezt meghagyjuk gyakorló feladatnak.



## Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

---

```
(* halfIntervalMethod (f, a, b) = root of f x in the a <= x <= b interval
 halfIntervalMethod : (real -> real) * real * real -> real *)
fun halfIntervalMethod(f, a, b) =
 let val (aValue, bValue) = (f a, f b)
 in if negativeOrZero aValue andalso positiveOrZero bValue
 then search(f, a, b)
 else if negativeOrZero bValue andalso positiveOrZero aValue
 then search(f, b, a)
 else (print ("Values " ^ makestring a ^ " and " ^ makestring b ^
 " are not of opposite sign.\n"); 0.0)
 end
and positiveOrZero x = x >= 0.0
and negativeOrZero x = x <= 0.0;
halfIntervalMethod(Math.sin, 2.0, 4.0);
> val it = 3.14111328125 : real
halfIntervalMethod(fn x => x*x*x-2.0*x-3.0, 1.0, 2.0);
> val it = 1.89306640625 : real
halfIntervalMethod(Math.sin, 2.0, 2.5);
Values 2.0 and 2.5 are not of opposite signs
> val it = 0.0 : real
```

## Függvény fixpontjának meghatározása

---

- Az  $f(x) = x$  egyenletet kielégítő  $x$  az  $f$  függvény *fixpontja*.
- Egy  $f$  függvény valamely fixpontját megfelelő kezdőértékből kiindulva  $f$  rekurzív alkalmazásával határozhatjuk meg:

$fx, f(fx), f(f(fx)), f(f(f(fx))), \dots$

A rekurzió akkor fejezhető be, amikor már elhanyagolható mértékű a változás.

- A `fixedPoint` függvény paramétere egy pár; ennek első tagja egy függvény, amelynek a fixpontját keressük, a második tagja pedig a fixpont egy első közelítése:

`fixedPoint (f, firstGuess) = fixpoint of f in the proximity of firstGuess with tolerance tolerance.`

- Szükségünk van tehát még a közelítés megkívánt pontosságára:

`val tolerance = 0.00001;`

## Függvény fixpontjának meghatározása (folyt.)

---

```
(* fixedPoint (f, firstGuess) = fixpoint of f in the proximity of
 firstGuess with tolerance tolerance
 fixedPoint : (real -> real) * real -> real
*)
fun fixedPoint (f, firstGuess) =
 let
 fun closeEnough (v1, v2) = abs(v1-v2) < tolerance
 fun try guess =
 let
 val next = f guess
 in
 if closeEnough(guess, next)
 then next
 else try next
 end
 in
 try firstGuess
 end;

fixedPoint(Math.cos, 1.0);
fixedPoint(fn y => Math.sin y + Math.cos y, 1.0);
```

## Függvény fixpontjának meghatározása (folyt.)

- A fixpontszámítás hasonlít a négyzetgyökvonás korábban megbeszélte folyamatára: mindkettő azon alapul, hogy addig finomítjuk a közelítést, amíg valamilyen feltétel nem teljesül.
- A négyzetgyökvonás könnyedén megfogalmazható fixpontszámításként: ha  $x$  négyzetgyöke  $y$ , akkor  $y * y = x$ , azaz  $y = x/y$ . Az  $f y = x/y$  függvény fixpontja tehát az  $x$  négyzetgyöke.

```
fun sqrt x = fixedPoint (fn y => x/y, 1.0);
```

- A megoldásunk rossz, ugyanis nem konvergál! Könnyen belátható:  
Legyen  $x$  négyzetgyökének első közelítése  $y_1$ , a második  $y_2 = x/y_1$ , a harmadik  $y_3 = x/y_2 = x/(x/y_1) = y_1$ . Látható, hogy a folyamat sohasem ér véget.
- Az oszcillációt pl. úgy gátolhatjuk meg, hogy *korlátozzuk* két közelítő érték között a változás mértékét.
- Mivel a helyes válasz mindig az  $y$  közelítő érték és  $x/y$  között van,  $y$ -hoz  $x/y$ -nál *közelebb eső* új közelítő értéként  $y$  és  $x/y$  átlagát választhatjuk:  $y \leftarrow (y + x/y)/2$ .

```
fun sqrt x = fixedPoint (fn y => (y+x/y)/2.0, 1.0);
```

- Ezt a gyakran használható módszert *átlagcsillapításnak* (angolul *average damping*) nevezik.

## Az átlagcsillapítás módszere

---

- A függvényekről mint absztrakciós eszközökről szólva eddig olyan magasabbrendű függvényeket használtunk, amelyeknek más függvények voltak a paraméterei.
- Most olyan magasabbrendű függvényeket mutatunk be, amelyek *függvényt* (pontosabban *függvényértéket*) adnak eredményül.
- A korábban bemutatott *átlagcsillapítás* sokszor használható módszer, ezért érdemes önálló függvényként megírni: ha adott az  $f$  függvény, elő kell állítani  $x$  és  $f x$  átlagát.

```
(* averageDamp f = f valamely x-re alkalmazva előállítja x és f x átlagát
 averageDamp : (real -> real) -> real -> real
*)
fun averageDamp f = fn x => (x + f x) / 2.0;
```

- Jól látható, hogy `averageDamp`, ha csak egyetlen paraméterre alkalmazzuk, függvényértéket ad eredményül. `averageDamp` részlegesen alkalmazható függvény.
- Példa `averageDamp` alkalmazására:

```
(averageDamp (fn x => x*x)) 10.0; (* 10.0 és 100.0 átlaga *)
```

- A kiértékelés sorrendje miatt a külső zárójelpár el is hagyható:

```
averageDamp (fn x => x*x) 10.0;
```

## Az átlagcsillapítás módszere (folyt.)

---

- `averageDamp` definíciója felírható a `fun` *szintaktikai édesítőszerez*rel is:

```
fun averageDamp f x = (x + f x) / 2.0;
```

- `sqrt` `averageDamp`-pel felírt változata explicitté teszi a *fixpontmeghatározás* és az *átlagcsillapítás* módszerét, továbbá az  $y = x/y$  *egyenlet használatát*.

```
fun sqrt x = fixedPoint(averageDamp (fn y => x/y), 1.0);
sqrt 4.0;
```

- Tanulság: egy folyamatot sokféle eljárással leírhatunk, de a *lényeg*et sokkal könnyebb megérteni, ha *megfelelően megválasztott absztrakciókat* vezetünk be.
- Még egy példa a bemutatottak alkalmazására: az  $x$  köbgyöke az  $y \mapsto x/y^2$  – SML-jelöléssel az `fn y => x/(y*y)` – függvény fixpontja. A megoldás már kész is van!

```
fun cubeRoot x = fixedPoint(averageDamp (fn y => x/y/y), 1.0);
cubeRoot 8.0;
```

## Az általános Newton-módszer

- Legyen  $x \mapsto g(x)$  egy differenciálható függvény és  $f(x) = x - g(x)/g'(x)$ , ahol  $g'(x)$  a  $g$  függvény  $x$  szerinti deriváltja. Ekkor a  $g(x) = 0$  egyenlet  $x$  megoldása az  $x \mapsto f(x)$  függvény egy fixpontja.
- Az *általános Newton-módszer* tehát a fixpontmódszer egy alkalmazása az  $f$  függvény egy fixpontjának megtalálására. Számos  $g$  függvényre és megfelelően megválasztott  $x$  értékre az általános Newton-módszer gyorsan konvergál.
- Először is azt a `deriv` függvényt kell definiálnunk, amelynek (az `averageDamp` függvényhez hasonlóan) függvény a paramétere, és függvényt ad eredményül.
- Ha  $g$  függvény és  $dx$  egy kis szám, akkor a  $g$  függvény  $g'$  deriváltja az a függvény, amelynek értéke bármely  $x$  számra a következő:  $g'(x) = (g(x + dx) - g(x))/dx$ .

```
(* deriv g = g deriváltja
 deriv : (real -> real) -> real -> real
```

```
*)
```

```
val dx = 0.00001;
```

```
fun deriv g = fn x => (g(x+dx) - g x) / dx;
```

- Például az  $x \mapsto x^3$  függvény deriváltja  $x = 5$ -re (pontos értéke 75):

```
let fun cube x = x*x*x in deriv cube 5.0 end;
```

## A Newton-módszer fixpont-folyamatként

---

- `deriv` felhasználásával az általános Newton-módszer *fixpont-folyamatként* definiálható:

```
(* newtonTransform g x = g
 transzformáltja az általános
 Newton-módszer alkalmazásához
```

```
newtonTransform : (real -> real) -> real -> real
```

```
newtonsMethod g guess = g
 gyökhelye, ahol guess a
 gyökkeresés kezdőértéke
```

```
newtonsMethod : (real -> real) -> real -> real
```

```
*)
```

```
fun newtonTransform g x = x - (g x / deriv g x)
```

```
and newtonsMethod g guess = fixedPoint(newtonTransform g, guess);
```

- Példa `newtonsMethod` használatára:

```
fun sqrt x = newtonsMethod (fn y => y*y-x) 1.0;
```

```
sqrt 16.0;
```



## A fixpontmódszer kétféle alkalmazása

---

- Két általános módszer egy-egy alkalmazását láttuk egy szám négyzetgyökének kiszámítására: az egyik a fixpont-, a másik a Newton-módszer.
- Mivel az utóbbi is a fixpontmódszeren alapul, valójában a fixpontmódszer kétféle alkalmazását láttuk.
- Mindkét esetben egy függvényből indulunk ki, és kiszámítjuk valamely transzformáltjának egy fixpontját.
- Ezt az általános módszert is definiálhatjuk eljárásként (függvényként):

```
(* fixedPointOfTransform (g, transform, guess) = a fixed point
 of transform g with the initial guess guess
 fixedPointOfTransform :
 'a * ('a -> real -> real) * real -> real
*)
fun fixedPointOfTransform (g, transform, guess) =
 fixedPoint(transform g, guess);
```

## A fixpontmódszer kétféle alkalmazása (folyt.)

---

- Ez volt például `sqrt` fixpontkeresésen alapuló első változata:

```
fun sqrt x = fixedPoint(averageDamp (fn y => x/y), 1.0)
```

- Átírva az általános módszert megvalósító függvénnyel:

```
fun sqrt x = fixedPointOfTransform (fn y => x/y,
 averageDamp, 1.0)
```

- Ez volt például `sqrt` Newton általános módszerét használó második változata:

```
fun sqrt x = newtonsMethod (fn y => y*y-x) 1.0;
```

- Átírva az általános módszert megvalósító függvénnyel:

```
fun sqrt x = fixedPointOfTransform (fn y => y*y-x,
 newtonTransform, 1.0)
```

# LISTÁK RENDEZÉSE

---

## Listák rendezése (folyt.)

---

- `insort` (beszúró rendezés),
- `selsort` (kiválasztó rendezés),
- `quicksort` (gyorsrendezés),
- `tmsort` (fölről lefelé haladó összefésülő rendezés),
- **`bmsort` (alulról fölfelé haladó összefésülő rendezés),**
- **`smsort` (simarendezés).**

## 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 rendezi az egészet. 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
```

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

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

```
*)
```

```
and 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
```

```
PRE: n >= 0
```

```
mergepairs : int list list * int -> int list list
```

```
*)
```

```
and 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.)

| l 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.)

---

- `msorting` a futamokat ismételten előállítja és összefuttatja:

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

PRE: `k >= 0`

`msorting : int list * int list list * int -> int list`

\*)

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

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

`msort : int list -> int list`

\*)

```
fun msort xs = msorting(xs, [], 0)
```

- A simarendezés egy változata `msort` néven található meg az `mosml Listsort` könyvtárban.

## Futási idők összehasonlítása smlnj alatt

---

- Az eddig használt `Random` struktúra az `mosml` alatt működik.
- Most az `smlnj` értelmezővel hasonlítjuk össze a gyorsrendezés, az alulról fölfelé haladó összefésülő rendezés és a `simarendezés` bemutatott változatainak, valamint az `smlnj`-ben megvalósított `ListMergeSort.sort` függvénynek a futási idejét.
- Az `smlnj` `Random` struktúrája többek között a `Random.rand (i, j)` és `Random.randInt rand` függvényeket definiálja (ld. <http://www.smlnj.org/doc/smlnj-lib/Manual/random.html>).
- `smlnj` alatt 800000 véletlenszámból álló egészlistát állítunk elő a mérésekhez:

```
(*
local
 val rand = Random.rand(57,87641)
 fun rg 0 zs = zs
 | rg i zs = rg (i-1) (Random.randInt rand :: zs)
 fun rgen n = rg n []
in
 val xs800000R = rgen 800000
end
*)
```

## Futási idők összehasonlítása smlnj alatt (folyt.)

---

A mérések:

```
(*
futIdo (quicksort1 Int.compare,"quicksort1","Int.compare") (xs800000R,"rand");
futIdo (quicksort2 Int.compare,"quicksort2","Int.compare") (xs800000R,"rand");
futIdo (quicksort3 Int.compare,"quicksort3","Int.compare") (xs800000R,"rand");
futIdo (bmsort,"bmsort","built-in op<=") (xs800000R,"rand");
futIdo (smsort,"smsort","built-in op<=") (xs800000R,"rand");
futIdo (ListMergeSort.sort op>=,"ListMergeSort.sort","op>=") (xs800000R,"rand");
*)
```

Az smlnj válaszai:

```
Int sort with quicksort1, Int.compare, length = 800000 (rand), time = 5.61s
Int sort with quicksort2, Int.compare, length = 800000 (rand), time = 2.80s
Int sort with quicksort3, Int.compare, length = 800000 (rand), time = 5.26s
Int sort with bmsort, built-in op<=, length = 800000 (rand), time = 5.19s
Int sort with smsort, built-in op<=, length = 800000 (rand), time = 5.45s
Int sort with ListMergeSort.sort, op>=, length = 800000 (rand), time = 5.70s
```

Feltűnő, hogy quicksort2 kétszer gyorsabb quicksort1-nél: ennek oka feltehetően az, hogy quicksort1-ben a @-ot, quicksort2-ben pedig a ::-ot alkalmazzuk. quicksort3 lassúságának oka minden bizonnyal a List.partition alkalmazásában keresendő.

## bmsort generikus változata: genbmsort

---

A következőkben megírjuk bmsort, majd smsort generikus változatát, azután pedig összehasonlítjuk a futási idejüket a nemgenerikus változatokkal.

```
(* genbmsort cmp xs = az xs elemeinek cmp szerint rendezett listája
 genbmsort : ('a * 'a -> order) -> 'a list -> 'a list
```

```
merge (xs, ys) = xs és ys elemeinek cmp szerint egyesített listája
 PRE: xs, ys cmp szerint rendezve vannak
merge : 'a list * 'a list -> 'a list
```

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

```
sorting (xs, lss, k) = a még rendezetlen xs lista elemeit berakja a
 rendezett részlisták összesen már k elemet tartalmazó lss listájába
 PRE: k >= 0
sorting : 'a list * 'a list list * int -> 'a list
```

\*)



## bmsort generikus változata: genbmsort (folyt.)

---

```

fun genbmsort cmp xs =
 let
 fun merge (xxs as x::xs, yys as y::ys) =
 if cmp(y, x) = GREATER then x :: merge (xs, yys)
 else y :: merge (xxs, ys)
 | merge ([], ys) = ys
 | merge (xs, []) = xs

 fun mergepairs (lss as ls1::ls2::lss, n) = (* lss min. kételemű *)
 if n mod 2 = 1 then lss
 else mergepairs (merge (ls1, ls2) :: lss, n div 2)
 | mergepairs (lss, _) = lss (* lss egyelemű *)

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

```

## msort generikus változata: gensmsort

---

(\* gensmsort cmp xs = az xs elemeinek cmp szerint rendezett listája  
 gensmsort : ('a \* 'a -> order) -> 'a list -> 'a list

merge (xs, ys) = xs és ys elemeinek cmp szerint egyesített listája  
 PRE: xs, ys cmp szerint rendezve vannak  
 merge : 'a list \* 'a list -> 'a list

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

nextrun (run, xs) = olyan pár, amelynek első tagja xs egy cmp szerint  
 növekvő sorrendű futama, második tagja pedig xs maradéka  
 nextrun : 'a list \* 'a list -> 'a list \* 'a list

smsorting (xs, lss, k) = a még rendezetlen xs lista elemeit berakja a  
 rendezett részlisták összesen már k elemet tartalmazó lss listájába  
 PRE: k >= 0  
 smsorting : 'a list \* 'a list list \* int -> 'a list

\*)

## msort generikus változata: gensmsort (folyt.)

---

```

fun gensmsort cmp xs =
 let fun merge (xxs as x::xs, yys as y::ys)=
 if cmp(y, x) = GREATER then x :: merge (xs, yys)
 else y :: merge (xxs, ys)
 | merge ([], ys) = ys
 | merge (xs, []) = xs

 fun mergepairs (lss as ls1::ls2::lss, n) = (* lss min. kételemű *)
 if n mod 2 = 1 then lss
 else mergepairs (merge (ls1, ls2) :: lss, n div 2)
 | mergepairs (lss, _) = lss (* lss egyelemű *)

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

 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))
 in smsorting (xs, [], 0) end;

```

## Futási idők összehasonlítása smlnj alatt (folyt.)

---

Az smlnj alatt összehasonlítjuk az alulról fölfelé haladó összefésülő rendezés és a simarendezés nemgenerikus és generikus változatainak futási idejét:

```
(*
futIdo (bmsort,"bmsort","built-in op<=") (xs800000R,"rand");
futIdo (genbmsort Int.compare,"genbmsort","Int.compare") (xs800000R,"rand");

futIdo (smsort,"smsort","built-in op<=") (xs800000R,"rand");
futIdo (gensmsort Int.compare,"gensmsort","Int.compare") (xs800000R,"rand");
*)
```

Az smlnj válaszai:

```
Int sort with bmsort, built-in op<=, length = 800000 (rand), time = 5.31s
Int sort with genbmsort, Int.compare, length = 800000 (rand), time = 5.81s

Int sort with smsort, built-in op<=, length = 800000 (rand), time = 4.88s
Int sort with gensmsort, Int.compare, length = 800000 (rand), time = 5.65s
```

A generikus változatok azért lassúbbak, mert az értelmezők az `Int.compare` relációt lassabban értékeli ki a `op<=` relációnál.

Az alulról fölfelé haladó összefésülő rendezés és különösen a simarendezés akkor hatékonyabb a gyorsrendezésnél, ha a rendezendő lista – több része – már kezdetben rendezve van. A vizsgálatokat meghagyjuk gyakorló feladatnak.