

BINÁRIS FÁK

Bináris fák FP-11..12-2

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 csomópontokban tárolt értékeket mikor veszik ki, és milyen sorrendben járják be a részfákat:
 - `preorder` először az értéket veszi ki, majd bejárja a bal, és azután a jobb részfát;
 - `inorder` először bejárja a bal részfát, majd kiveszi az értéket, 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!

- 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 előállítás lista elemeiből: balPreorder

- Listát *kiegyensúlyozott (balanced) bináris fává* alakítanak a következő függvények: balPreorder, balInorder és balPostorder; a különbség közöttük most is a bejárési 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 listaelemeiből: 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:

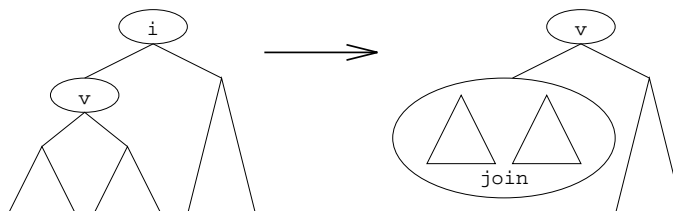
```
(* balPreorder: 'a list -> 'a tree
   balPreorder xs = az xs lista elemeiből álló, preorder ... *)
fun balPreorder [] = L
  | balPreorder (x::xs) =
    let val k = length xs div 2
        val (ts, ds) = take'ndrop(xs, k)
    in N(x, balPreorder ts, balPreorder ds)
    end
```

Bináris fa előállítása 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 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 szétválasztott részfa 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.

Kivételkezelés

- Kivételt az `exception` kulcsszóval deklarálunk, a `raise` kulcsszóval jelzünk, a `handle` kulcsszóval bevezetett kifejezésben kezelünk.
- A kivételt általában hibák jelzésére használjuk, de használhatjuk visszalépés kezelésére is (az utóbbira példa a `valtas` függvényben látható a következő fóliák egyikén).
- A kivételdeklaráció az adattípus-deklarációra (`datatype`-deklarációra) emlékeztet:
`exception name; exception name of ty.`
- Példák kivétel deklarálására: `exception Valt; exception Hiba of char * int.`
- A kivételkonstruktor állandó vagy függvény lehet. Példák: `Valt : exn, Hiba : char * int -> exn.`
- A kivételdeklaráció speciális adattípus-deklaráció, ui. az utóbbival ellentétben dinamikusan *bővíti* a kivételkonstruktorok halmazát.
- Kivétel jelzésére a `raise` kulcsszóval kezdődő speciális kifejezést kell használnunk.
- Példák kivétel jelzésére: `raise Valt, raise Hiba("#N", 4).`
- `raise` (hipotetikus) típusa: `exn -> 'a.`

Kivételkezelés (folyt.)

- `raise` alkalmazásának eredménye az ún. *kivételcsomag*. Mivel a kivételcsomag polimorf típusú, bármely más típussal kompatibilis.
- A kivétel kezelése a `case`-szerkezetre emlékeztet: `E handle P1 => E1 | ... | Pn => En`
- Ha `E` „közönséges” értéket ad eredményül, a kivételkezelő egyszerűen továbbadja az eredményt.
- Ha `E` eredménye *kivételcsomag*, az SML megpróbálja illeszteni a `P1`, ..., `Pn` mintákra.
 - Ha `Pi` ($1 \leq i \leq n$) az első illeszkedő minta, akkor `Ei` a kivételkezelő eredménye.
 - Ha egyetlen minta sem illeszkedik a kivételcsomagra, a kivételkezelő továbbpasszolja.
- Példák kivétel kezelésére:
 - `erme :: valtas (erme::ermelista) (osszeg-erme)`
`handle Valt => valtas ermelista osszeg`
 - `(fn i => kivKez i handle Hiba(c, i) => (print(str c); i-1)) 0`
- `handle` (hipotetikus) típusa: `exn -> 'a.`
- Legyen `Ex` `exn` típusú kivétel, `e` pedig tetszőleges kifejezés; ekkor az `e handle Ex => c` (kivételkezelőt tartalmazó) kifejezésben `c`-nek `e`-vel azonos típusúnak kell lennie.

Kivételkezelés (folyt.)

- A következő programrészlet példa kivétel deklarálására, jelzésére és kezelésére

```
exception Hiba of char * int;

fun kivKez 0 = raise Hiba("#N", 4)
  | kivKez ~9 = raise Hiba("#M", 9)
  | kivKez n = n;

fun kivKezel i =
    kivKez i handle Hiba("#N", i) => (print "N"; i)
                | Hiba("#M", i) => (print "M"; i-1);

kivKezel 0 = 4;
kivKezel ~9 = 8;
kivKezel 7 = 7;
```

Kivételkezelés (folyt.)

- Példa visszalépés programozására kivételkezeléssel

```
exception Valt;

(* váltás : int list -> int -> int list
   váltás ermelista osszeg = a lehető legkevesebb érmét tartalmazó olyan
   ermelista, amely elemeinek összege osszeg
   PRE : ermelista = a váltásra használható érmék csökkenő értéksorrendben
   osszeg >= 0
*)
fun váltás _ 0 = []
  | váltás [] _ = raise Valt
  | váltás (erme::ermelista) osszeg =
    if (* ha az adott érme túl nagy, a következővel próbálkozunk *)
        erme > osszeg then váltás ermelista osszeg
    (* ha az adott érmétől kezdve sikerül felváltani, az jó;
       ha nem, a következő érmével kezdjük újra az adott ponttól *)
    else erme :: váltás (erme::ermelista) (osszeg-erme)
        handle Valt => váltás ermelista osszeg;

váltás [50, 20, 10, 5, 2] 197 = [50, 50, 50, 20, 20, 5, 2];
```

Kivételkezelés (folyt.)

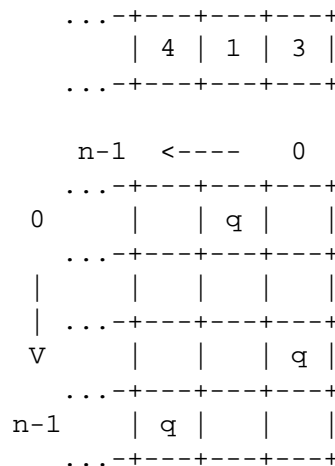
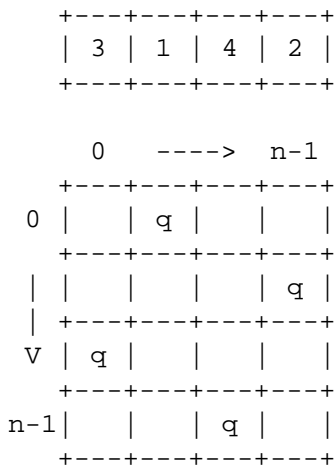
- A leggyakoribb belső kivételek

Név	Művelet, amely a kivételt kiválthatja
Bind	Értékdeklarációban a jobb oldali kifejezés nem illeszkedik a bal oldali mintára.
Chr	chr pred succ
Div	/ div mod
Domain	Az érték kilóg az értelmezési tartományból.
Empty	hd tl last
Fail	compile load loadOne Fail : string -> exn
Interrupt	Megszakítás ctrl/c-vel.
Io	Ki/beviteli hiba. Io : {cause : exn, function : string, name : string}
Match	Mintaillesztési hiba case és handle kifejezésben, vagy függvényalkalmazásban.
Option	Hiba egy Option könyvtárbeli függvény alkalmazásakor.
Overflow	~ + - * / div mod abs ceil floor round trunc
Size	^ array concat fromList implode tabulate translate vector
Subscript	copy drop extract nth sub substring take update

- Fail és Io kivételkonstruktorfüggvények, a többi exn típusú kivételkonstruktorállandó.
- Option csak Option.Option néven használható, ha nem nyitjuk meg az Option könyvtárat.

n vezér a sakktáblán

- Hányféleképpen rakható *n* vezér a sakktáblára úgy, hogy ne üssék egymást?
- A vezéreket tartalmazó mezők sorának *j* sorszámát az egyes oszlopokon belül egy *n* hosszú sorvektor adott oszlophoz rendelt mezőjébe írt szám adja meg, ahol $j \leq s < n$.
Példa $n=4$ esetén:
- A sorvektort (egy egyre bővülő) listával valósítjuk meg. Egy listához balról könnyű új elemeket fűzni, ezért a táblát és a vezérek helyzetét leíró listát hossz tengelye mentén tükrözzük.

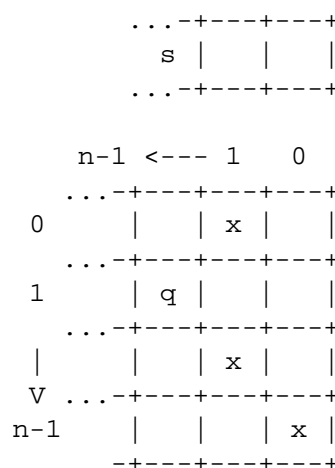


n vezér a sakktáblán (folyt.)

Azt, hogy az új vezért üti-e egy korábban a táblára rakott másik vezér, a sorvektor vizsgálatával dönthetjük el: a sorvektor azt adja meg, hogy a listaelem indexe által meghatározott oszlopban és a listaelem értéke által meghatározott sorban vezér van.

- Az új vezér sorának sorszáma, azaz az új listaelem értéke nem fordulhat elő a lista már felépített részében.
- Az új vezér átlós irányban sem lehet egy vonalban más vezérrel a táblán. Ez azt jelenti, hogy ha a sorvektort jelentő lista elejére az *s* sorindexet akarjuk rakni, akkor az *i*-edik elemének az értéke, ha van ilyen eleme, nem lehet $s - (i + 1)$, ill. $s + (i + 1)$.
- A következő példa segít megvilágítani az esetet.

Ha a 2-es oszlopba és az $s=1$ -es sorba akarjuk lerakni az új vezért, akkor az *x*-szel jelölt mezőket kell megvizsgálnunk. Az eddig létrehozott listának (sorvektornak) két eleme van, ahol a lista fejének az indexe 0. A listafej értéke nem lehet $s-1$, sem $s+1$. A lista rekurzív algoritmussal dolgozható fel.



***n* vezér a sakktáblán: „ütésben van”-vizsgálat**

```

(* utesbenVan : int list -> bool
    utesbenVan zs = igaz, ha a (hd zs) vezért legalább egy
        (tl zs)-beli vezér üti
*)
fun utesbenVan [] = false
  | utesbenVan (z::zs) =
    let (* uV : int -> int -> int list -> bool
        uV s1 s2 rs = igaz, ha a z vezért s1, s2 vagy r, vagy
            egy másik rs-beli vezér közül legalább egy üti
        *)
        fun uV _ _ [] = false
          | uV s1 s2 (r::rs) = z = r orelse
                                s1 = r orelse
                                s2 = r orelse
                                uV (s1-1) (s2+1) rs

        in
            uV (z-1) (z+1) zs
        end
    end

```

***n* vezér a sakktáblán: egy megoldás előállítás**

```

exception Zsakutca

(* vezerek0 : int -> int list
    vezerek0 n = a feladvány egy megoldása n vezér esetén
*)
fun vezerek0 n =
  let (* vez : int -> int list -> int list
      vez z zs: egy megoldás n vezér esetén
      *)
      fun vez z zs =
          if z = 0 andalso utesbenVan zs orelse z = n
          then raise Zsakutca
          else if length zs = n
          then rev zs
          else vez 0 (z::zs) handle Zsakutca => vez (z+1) zs

      in
          vez 0 []
      end
  end

```

***n* vezér a sakktáblán: több megoldás előállítása visszalépéssel**

```

(* vezerek1 : int -> int list list
   vezerek1 n = a feladvány összes megoldásának listája
                 n vezér esetén
*)
fun vezerek1 n =
  let (* vez: int -> int list -> int list list
       vez z zs: az összes megoldás listája n vezér esetén
     *)
      fun vez z zs =
          if z = 0 andalso utesbenVan zs orelse z = n
          then raise Zsakutca
          else if length zs = n
          then [rev zs]
          else (vez 0 (z::zs) handle Zsakutca => []) @
               (vez (z+1) zs handle Zsakutca => [])
      in
        vez 0 []
      end

```

***n* vezér a sakktáblán: több megoldás előállítása listák listájával**

```

(* vezerek2 : int -> int list list
   vezerek2 n = a feladvány összes megoldásának listája
                 n vezér esetén
*)
fun vezerek2 n =
  let (* vez: int -> int list -> int list list
       vez z zs: az összes megoldás listája n vezér esetén
     *)
      fun vez z zs =
          if z = 0 andalso utesbenVan zs orelse z = n
          then []
          else if length zs = n
          then [rev zs]
          else vez 0 (z::zs) @ vez (z+1) zs
      in
        vez 0 []
      end

```

***n* vezér a sakktáblán: több megoldás előállítása listák listájával (folyt.)**

Akkumulátor alkalmazásával:

```
(* vezerek3 : int -> int list list
   vezerek3 n = a feladvány összes megoldásának listája
               n vezér esetén
*)
fun vezerek3 n =
  let (* vez: int -> int list -> int list list -> int list list
       vez z zs: az összes megoldás listája n vezér esetén
       *)
      fun vez z zs zss =
          if z = 0 andalso utesbenVan zs orelse z = n
          then zss
          else if length zs = n
               then rev zs :: zss
               else vez 0 (z::zs) (vez (z+1) zs zss)
      in
          vez 0 [] []
      end
```

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 elemeiből álló halmazok uniója
*)
fun union ([], ys)      = ys
  | union (x::xs, ys) = newMem(x, union(xs, ys))
```

- Két halmaz metszete

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

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

- Részhalmaza-e egy halmaz egy másiknak?

```
(* isSubset : 'a list * 'a list -> bool
   isSubset (xs, ys) = az xs elemeiből álló halmaz 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 részhalmazának a halmaza, az S -t és a $\{\}$ -t is beleértve.
- S hatványhalmaza úgy állítható elő, hogy kivesszük S -ből az x elemet, majd *rekurzív módon* előállítjuk az $S - \{x\}$ hatványhalmazát.
- Ha tetszőleges T halmazra $T \subseteq S - \{x\}$, akkor $T \subseteq S$ és $T \cup \{x\} \subseteq S$, így mind T , mind $T \cup \{x\}$ eleme S hatványhalmazának.
- Miközben a fenti elvet rekurzív módon alkalmazzuk, tehát felsorolhatjuk az $S - \{x\}$ stb. részhalmazait, 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.
- `pws(xs, base) = \{S \cup base \mid S \subseteq xs\}`, azaz `xs \cup base` azon részhalmazainak 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
                   részhalmazának 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, [])
```

Halmazműveletek: „halmaz hatványhalmaza”, hatékonyabban

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

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

EGYIDEJŰ DEKLARÁCIÓ

Egyidejű deklaráció

- Típusok, ill. értékek *egyidejűleg* is deklarálhatók 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 (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 főlülről lefelé haladva akarunk programot írni. Példa:

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

- 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 `id int -> int` és `real -> real` típusú lenne egyszerűen, ami lehetetlen.

AZ ORDER TÍPUS

Az order típus FP-11..12-40

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 FP-11..12-42

Listák rendezése

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

Beszűrő rendezés

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

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

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

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

- Példa `inssort` alkalmazására:

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

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)
```

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

```

Gyorsrendezés akkumulátorral

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

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.