KIÍRÁS

Kiírás

- `{TextIO.}print : string -> unit`
  `print s` = kiírja az `s` értékét a standard kimenetre, és azonnal kiüríti a puffert.
- `{General.}makestring : numtxt-> string`
  `makestring v` = eredménye a `v` érték *ábrázolása.*
- `{Meta.}printVal : 'a -> 'a`
  `printVal e` = kiírja az `e` kifejezés értékét a standard kimenetre pontosan úgy, ahogyan az SML értelmező írja ki a „legfelső szinten”, és azonnal kiüríti a puffert. Eredményül visszaadja az `e` kifejezés értékét. *Csak interaktív módban használható.*
- Példák:

```
- print("alma"^"Korte\n");
almaKorte
> val it = () : unit
- makestring ~5.8e~3;
> val it = "~0.0058" : string
```

```
- printVal("alma"^"Korte\n");
"almaKorte\n"> val it = "almaKorte\n" : string

- makestring("alma"^"Korte\n");
> val it = "\"almaKorte\\n\"" : string
```

Megjegyzések. A kapcsos zárójelek – { és } – között opcionálisan megadható modulnév áll. Például `{TextIO.}print` azt jelenti, hogy a függvény a `TextIO` modulban van definiálva, de az SML-értelmező a `print` nevet rövid alakban is felismeri. *`numtxt`* = `int | real | word | word8 | char | string`

Kiírás (folyt.)

- `printVal`-lal tetszőleges típusú érték íratható ki. További példák:

```
- printVal (3, 5.0);
(3, 5.0)> val it = (3, 5.0) : int * real

- printVal [#"A",#"Z",#":"];
[#"A", #"Z", #":"]> val it = [#"A", #"Z", #":"] : char list

- datatype t = L | B of t * t;
> New type names: =t
  datatype t = (t,con B : t * t -> t, con L : t)
  con B = fn : t * t -> t
  con L = L : t
- val fa = B(B(B(L,B(L,B(L,B(B(L,L),L)))),L),B(L,L));
> val fa = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
- printVal fa;
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))> val it = B(B(B(L, B(L, B(L
```


Kiírás (folyt.)

- Az utolsó példában a kiírt sor túl hosszú lett, jó lenne eltörni a `>` jel előtt. Hogyan írathatunk ki egy újsor-jelet úgy, hogy az eredmény a `fa` érték maradjon? Például így, de ez elég körülményes:

```
- let val res = printVal fa;
      val _ = print "\n"
  in
     res
  end;
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```

- A `before` operátort az ilyen és hasonló dolgok kezelésére találták ki.

Szekvenciális kifejezés: `before`

- Az `x before y` kifejezés az ún. *szekvenciális kifejezés* egy változata.
  `{General.}before : 'a * 'b -> 'a`
  `x before y` = először az `x`-et, majd az `y`-t értékeli ki, eredménye az `x` értéke.
  Precedenciaszintje 0.
- Példa `before` használatára:

```
- printVal fa before print "\n";
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```

- Az `x before y`-hoz hasonló a `(x; y)` szekvenciális kifejezés, amely azonban az *utolsó* részkifejezésének az értékét adja eredményül.

```
- (print "A fa változó értéke =\n"; printVal fa be-
fore print "\n");
A fa változó értéke =
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```


Kiírás (folyt.)

- Hosszú lista, ill. egymásba skatulyázott adatszerkezetek esetén `printVal` (és maga az SML-értelmező is) alapesetben csak az első 200 listaelemet, ill. legfeljebb 20 szintet ír ki. A hosszat a `printLength`, a szintek számát a `printDepth` *frissíthető változó* szabályozza. Mindkét érték felülírható.

```
printLength : int ref        | printLength := 7;  !printLength;
printDepth  : int ref        | printDepth  := 3;  !printDepth;
```

- Példák:

```
- printVal [1,2,3,4,5,6,7,8,9,10] before print "\n";
[1, 2, 3, 4, 5, 6, 7, ...]
> val it = [1, 2, 3, 4, 5, 6, 7, ...] : int list
- printVal fa before print "\n";
B(B#, B#)
> val it = B(B#, B#) : t
```

- Figyelem: a `printLength` és a `!printLength` kifejezések különböznek!

```
- printLength;                | - !printLength;
> val it = ref 7 : int ref    | > val it = 7 : int
```


Kiírás, szekvenciális kifejezés (folyt.)

- Különböző típusú egyszerű értékeket alakítanak át füzérré a `toString` függvények:

```
Char.toString : char -> string
Int.toString  : int -> string
Real.toString : real -> string
Bool.toString : bool -> string
Word.toString : word -> string
```

- Szekvenciális kifejezést felírhatunk a `;` (pontosvessző) alkalmazásával is.
- Az `(x; y)` szekvenciális kifejezés, akárcsak az `x before y`, szintaktikai édesítőszer. Az `(x; y)` ekvivalens az alábbi kifejezéssel:

```
let val _ = x in y end
```


NYOMKÖVETÉS, LISTÁK

Nyomkövetés: `length` (nem iteratív)

- Az MOSML-ben nyomkövetés csak a program szövegébe beírt kiíró függvényekkel lehetséges.
- Példa: a `length` függvény két változatának kiértékelése
- A `length` „naív” változata

```
fun length (_::xs) = 1 + length xs
  | length []      = 0
```

- A `length` „naív” változata kiíró függvényekkel **(félkövér** szedéssel az eredeti szöveg látható)

```
fun length ((_ : int) :: xs) =
        printVal(1 + (print " & "; printVal(length(printVal xs))
                                   before print " $ "
                     )
                 )
        before print " #\n"
  | length []      = (print " * "; printVal 0
                                   before print " %\n")
```


Nyomkövetés: `lengthi` (iteratív)

- A `length` iteratív változata

```
fun lengthi xs = let fun len (i, _::xs) = len(i+1, xs)
                       | len (i, [])    = i
                 in len(0, xs)
                 end
```

- A `length` iteratív változata kiíró függvényekkel **(félkövér** szedéssel az eredeti szöveg látható)

```
fun lengthi xs =
    let fun len (i, (_ : int) :: xs) =
              len((print " "; printVal((printVal i
                                        before print " $ ") + 1)),
                  (print " & "; printVal xs)
                 )
                  before print "#\n"
        | len (i, []) = (print " * "; printVal i
                                      before print " %\n")
    in len(0, xs)
    end
```


Nyomkövetés: `length` egy alkalmazása

- `length` egy alkalmazása

```
fun length ((_ : int) :: xs) =
        printVal(1 + (print " & "; printVal(length(printVal xs))
                                   before print " $ "
                     )
                 )
        before print " #\n"
  | length []      = (print " * "; printVal 0
                                   before print " %\n")

length [1,2,3];
 & [2, 3] & [3] & [] * 0 %
0 $ 1 #
1 $ 2 #
2 $ 3 #
```


Nyomkövetés `lengthi` egy alkalmazása

- `lengthi` egy alkalmazása

```
fun lengthi xs =
    let fun len (i, (_ : int) :: xs) =
              len((print " "; printVal((printVal i
                                        before print " $ ") + 1)),
                  (print " & "; printVal xs)
                 )
                  before print "#\n"
        | len (i, []) = (print " * "; printVal i
                                      before print " %\n")
    in len(0, xs)
    end

lengthi [1,2,3];
 0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %
#
#
#
```

Nyomkövetés: `length` és `lengthi` összehasonlítása

- `length` és `lengthi` kiértékelésének összehasonlítása

```
length [1,2,3];

 & [2, 3] & [3] & [] * 0 %
0 $ 1 #
1 $ 2 #
2 $ 3 #
```

```
lengthi [1,2,3];

 0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %
#
#
#
```

- Korábban tárgyaltuk a `nodes` és `depth` függvényeket, valamint akkumulátort használó `nodesa` és `deptha` változatukat.

  A következő fóliákon e függvények kiértékelésének nyomkövetésére mutatunk megoldást.

  A szöveg olvashatóságát (szintenként növekvő) *behúzással* javítjuk. A megfelelő számú szóköz beszúrására szolgál a `tab` függvény. A változó számú szóközből álló füzért paraméterként adjuk át, ezért olyan segédfüggvényeket vezetünk be, amelyeknek az őket meghívó függvényhez képest eggyel több paraméterük van.

  A függvények *kiírást szolgáló részek nélküli* szövegét **félkövér** szedéssel jelöljük.

BINÁRIS FÁK, NYOMKÖVETÉS

Nyomkövetés: `nodes` (akkumulátort nem használ)

```
(* tab : string -> string
   tab i = a sorok behúzásához használandó i füzér szóközökkel kiegészítve *)
fun tab i = i ^ "    "

fun nodes f =
      let (* nodes0 : string -> 'a tree -> int
             nodes0 i f = a csomópontok száma az f fában;
                          i a behúzáshoz használt füzér *)
          fun nodes0 i (N(a, t1, t2)) =
                (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
                 printVal(1 +
                          nodes0 (tab i) (printVal t2 before print " *") +
                          nodes0 (tab i) (printVal t1 before print " %")
                          before print "$ "
                         )
                 before print(" #\n" ^ i)
                )
            | nodes0 i L = (print("\n" ^ i); 0)
      in
          nodes0 "" f
      end
```


Nyomkövetés: `nodesa` (akkumulátort használ)

```
fun nodesa f =
  let (* nodes0 i (f, n) = n + a csomópontok száma f-ben;
                           i a behúzáshoz használt füzér
         nodes0 : string -> 'a tree * int -> int
      *)
      fun nodes0 i (N(a, t1, t2), n) =
            (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
             nodes0 (tab i) (printVal t1 before print(" %\n" ^ (tab i)),
                             nodes0 (tab i) (printVal t2 before print(" *\n" ^
                                                                      (tab i)),
                                             printVal(n+1) before print " $"
                                            )
                            )
             before print(" #" ^ i)
            )
        | nodes0 i (L, n) = (* (print("\n" ^ i); n) *) n
  in
      nodes0 "" (f, 0)
  end
```

nodes és nodesa alkalmazása hét csomópontból álló teljes fára

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree

- nodes f7;                          | - nodesa f7;
                                     |
<1> N(3, N(6, L, L), N(7, L, L)) *   | <1> N(2, N(4, L, L), N(5, L, L)) %
    <3> N(7, L, L) *                 |     N(3, N(6, L, L), N(7, L, L)) *
        <7> L *                      |     1 $
            L %                      |     <3> N(6, L, L) %
            $ 1 #                    |         N(7, L, L) *
        N(6, L, L) %                 |         2 $
        <6> L *                      |         <7> L %
            L %                      |             L *
            $ 1 #                    |             3 $ #
        $ 3 #                        |         <6> L %
    N(2, N(4, L, L), N(5, L, L)) %   |             L *
                                     |             4 $ #         #
```

Folytatása a következő lapon.

nodes és nodesa alkalmazása . . . (folyt.)

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree

 (nodes f7)                          |  (nodesa f7)
                                     |
    <2> N(5, L, L) *                 |     <2> N(4, L, L) %
        <5> L *                      |         N(5, L, L) *
            L %                      |         5 $
            $ 1 #                    |         <5> L %
        N(4, L, L) %                 |             L *
        <4> L *                      |             6 $ #
            L %                      |         <4> L %
            $ 1 #                    |             L *
        $ 3 #                        |             7 $ #         #     #
    $ 7 #                            |
                                     |
> val it = 7 : int                   | > val it = 7 : int
```


Nyomkövetés: depth (akkumulátort nem használ)

```
fun depth f =
  let (* depth0 i f = az f fa mélysége; i a behúzáshoz használt füzér
         depth0 : string -> 'a tree -> int
      *)
      fun depth0 i (N(a : int, t1, t2)) =
            (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
             printVal(1 +
                      Int.max(depth0 (tab i) (printVal t2 before print " *"),
                              depth0 (tab i) (printVal t1 before print " %")
                             )
                     )
             before print(" #\n" ^ i))
        | depth0 i L = (print( "\n" ^ i) ; 0)
  in
      depth0 "" f
  end
```


Nyomkövetés: deptha (akkumulátort használ)

```
fun deptha f =
  let (* depth0 i (f, d) = d + az f fa mélysége; i a behúzáshoz használt füzér
         depth0 : string -> 'a tree * int -> int *)
      fun depth0 i (N(a : int, t1, t2), d) =
            (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
             printVal(Int.max(depth0 (tab i) (printVal t2 before print(" *\n" ^
                                                                       (tab i)),
                                              printVal(d+1) before print " $ "
                                             ),
                              depth0 (tab i) (printVal t1 before print(" %\n" ^
                                                                       (tab i)),
                                              printVal(d+1) before print " & "
                                             )
                             )
                     )
             before print(" #\n" ^ i)
            )
        | depth0 i (L, d) = (print( "\n" ^ i) ; d)
  in
      depth0 "" (f, 0)
  end
```

`depth` és `deptha` alkalmazása hét csomópontból álló teljes fára

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree

- depth f7;
<1> N(3, N(6, L, L), N(7, L, L)) *
    <3> N(7, L, L) *
        <7> L *
            L %
            1 #
        N(6, L, L) %
        <6> L *
            L %
            1 #
        2 #
    N(2, N(4, L, L), N(5, L, L)) %
```

```
- deptha f7;

<1> N(3, N(6, L, L), N(7, L, L)) *
    1 $
    <3> N(7, L, L) *
        2 $
        <7> L *
            3 $
            L %
            3 &
            3 #
        N(6, L, L) %
        2 &
        <6> L *
            3 $
            L %
            3 &
            3 #
        3 #
    N(2, N(4, L, L), N(5, L, L)) %
    1 &
```


`depth` és `deptha` alkalmazása hét csomópontból álló teljes fára (folyt.)

Folytatás az előző lapról.

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree

 (depth f7)
    <2> N(5, L, L) *
        <5> L *
            L %
            1 #
        N(4, L, L) %
        <4> L *
            L %
            1 #
        2 #
    3 #

> val it = 3 : int
```

```
 (deptha f7)

    <2> N(5, L, L) *
        2 $
        <5> L *
            3 $
            L %
            3 &
            3 #
        N(4, L, L) %
        2 &
        <4> L *
            3 $
            L %
            3 &
            3 #
        3 #
    3 #
> val it = 3 : int
```


BINÁRIS FÁK

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 előállítása 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 előállítása bináris fa elemeiből (folyt.)

Az akkumulátort használó változatok nehezebben érthetőek meg, de *hatékonyabbak*.

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

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

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


Bináris fa előállítása lista elemeiből: `balPreorder`

- Listát *kiegyensúlyozott* (**bal**anced) *bináris fává* alakítanak a következő függvények: `balPreorder`, `balInorder` és `balPostorder`; a különbség közöttük most is a bejárási sorrendben van.
- ```
  (* balPreorder: 'a list -> 'a tree
     balPreorder xs = az xs lista elemeiből álló, preorder
                      bejárású, kiegyensúlyozott fa
  *)
  fun balPreorder [] = L
    | balPreorder (x::xs) =
        let val k = length xs div 2
        in
           N(x, balPreorder(List.take(xs, k)),
                balPreorder(List.drop(xs, k)))
        end
  ```
- A hatékonyságot kisebb mértékben rontja, hogy `List.take` és `List.drop` egymástól függetlenül *kétszer* mennek végig a lista első felén.

`take` és `drop` egyetlen függvénnyel: `take'ndrop`

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

```
(* take'ndrop : 'a list * int -> 'a list * 'a list
   take'ndrop(xs, k) = olyan pár, amelynek első tagja xs első k db
                       eleme, második tagja pedig xs maradéka
*)
fun take'ndrop (xs, k) =
      let fun td (xs, 0, ts) = (rev ts, xs)
            | td ([], _, ts) = (rev ts, [])
            | td (x::xs, k, ts) = td(xs, k-1, x::ts)
      in
          td(xs, k, [])
      end
```

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

Bináris fa előállítása 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 előállítása lista elemeiből

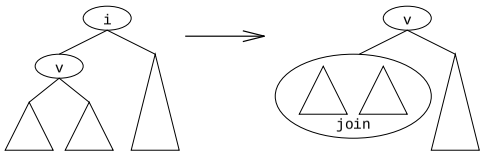
-
```
(* balInorder: 'a list -> 'a tree
   balInorder xs = az xs lista elemeiből álló, inorder bejárású,
                   kiegyensúlyozott fa
*)
fun balInorder [] = L
  | balInorder (xxs as x::xs) =
      let val k = length xxs div 2
          val ys = List.drop(xxs, k)
      in
          N(hd ys, 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 széteső fa részfáit *egyesíteni* kell, miután a törlést a két részfán már végrehajtottuk.



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

Elem rekurzív törlése bináris fából (folyt.)

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

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

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

```
(* remove : 'a * 'a tree -> 'a tree
   remove(i, f) = i összes előfordulását törli f-ből *)
fun remove (i, L) = L
  | remove (i, N(v,lt,rt)) =
      if i<>v
      then N(v, remove(i,lt), remove(i,rt))
      else join(remove(i,lt), remove(i,rt))
```

Bináris keresőfák: `blookup`, `binsert`

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

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


Bináris keresőfák: `bupdate`

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

```
(* binsert : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   binsert(f, (b,y)) = az új (b,y) kulcs-érték párral bővített f fa *)
fun binsert (L, (b,y)) = N((b,y), L, L)
  | binsert (N((a, x), t1, t2), (b,y)) =
      if b < a      then N((a, x), binsert(t1, (b,y)), t2)
      else if a < b then N((a, x), t1, binsert(t2, (b,y)))
      else (* a=b *) raise Bsearch("INSERT: " ^ b)
```

- A `bupdate` függvény meglévő kulcsú elembe új értéket ír be egy rendezett bináris fában:

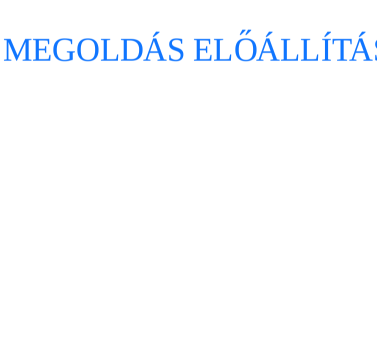
```
(* bupdate : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   bupdate(f, (b,y)) = az f fa, a b kulcshoz tartozó érték helyén
                       az y értékkel *)
fun bupdate (L, (b,y)) = raise Bsearch("UPDATE: " ^ b)
  | bupdate (N((a,x), t1, t2), (b,y)) =
      if b < a      then N((a,x), bupdate(t1, (b,y)), t2)
      else if a < b then N((a,x), t1, bupdate(t2, (b,y)))
      else (* a=b *) N((b,y), t1, t2)
```

- A függvények *generikussá* tételét meghagyjuk gyakorló feladatnak.

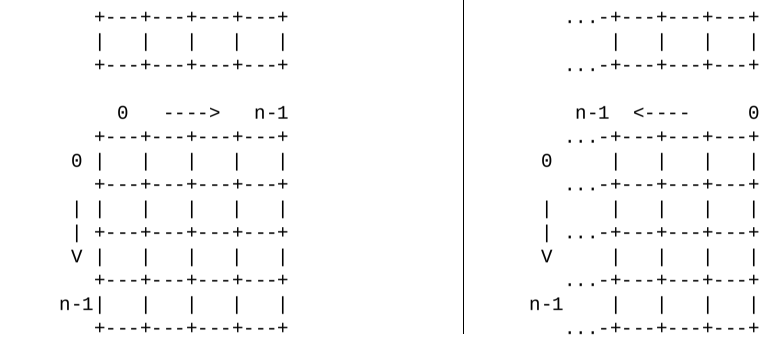
TÖBB MEGOLDÁS ELŐÁLLÍTÁSA VISSZALÉPÉSSEL

n vezér a sakktáblán

- Hányféleképpen rakható *n* vezér a sakktáblára úgy, hogy ne üssék egymást?
- A vezéreket tartalmazó mezők sorának számát az egyes oszlopokon belül egy `n` hosszú sorvektor adott oszlophoz rendelt mezőjébe írt `<= s < n` szám adja meg. Példa `n = 4` esetén:



- A sorvektort (egy egyre bővülő) listával valósítjuk meg. Egy listához balról könnyű új elemeket fűzni, a táblát és a vezérek helyzetét leíró listát hossztengelye mentén tükrözzük.

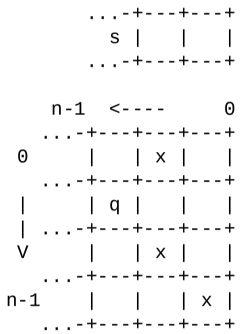


n vezér a sakktáblán (folyt.)

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

1. Az új vezér sorának száma, azaz az új listalem értéke nem fordulhat elő a lista már felépített részében.
2. 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)`.
3. 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) vezér nincs ütésben
                   egyetlen (tl zs)-beli vezérrel sem
*)
fun utesbenVan [] = false
  | utesbenVan (z::zs) =
      let 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
```


n vezér a sakktáblán: egy megoldás előállítása

```
exception Zsakutca

(* vezerek0 : int -> int list
   vezerek0 n = a feladvány egy megoldása n vezér esetén
*)
fun vezerek0 n =
  let
      fun vez0 z zs =
            if z = 0 andalso utesbenVan zs orelse z = n
            then raise Zsakutca
            else if length zs = n
            then rev zs
            else vez0 0 (z::zs) handle Zsakutca => vez0 (z+1) zs
  in
      vez0 0 []
  end
```


n vezér a sakktáblán: több megoldás előállítása visszalépéssel

```
(* vezerek : int -> int list list
   vezerek n = a feladvány összes megoldásának listája
               n vezér esetén
*)
fun vezerek n =
  let
      fun vez0 z zs =
            if z = 0 andalso utesbenVan zs orelse z = n
            then raise Zsakutca
            else if length zs = n
            then [rev zs]
            else (vez0 0 (z::zs) handle Zsakutca => []) @
                 (vez0 (z+1) zs handle Zsakutca => [])
  in
      vez0 0 []
  end
```

n vezér a sakktáblán: több megoldás előállítása listák listájával

```
fun vezerek n =
  let fun vez0 z zs =
            if z = 0 andalso utesbenVan zs orelse z = n
            then []
            else if length zs = n
            then [rev zs]
            else vez0 0 (z::zs) @ vez0 (z+1) zs
  in  vez0 0 [] end
```

Ugyanez akkumulátor alkalmazásával:

```
fun vezerek n =
  let fun vez0 z zs ws =
            if z = 0 andalso utesbenVan zs orelse z = n
            then ws
            else if length zs = n
            then rev zs :: ws
            else vez0 0 (z::zs) (vez0 (z+1) zs ws)
  in  vez0 0 [] [] end
```