```
(* insert(m, ns)  inserts  m  in the correct position within list  ns.
   Precondition:  m  is an int  and  ns  is a sorted list of ints.
   Postcondition:  the returned answer is a sorted list containing exactly
      m  and the elements in  ns.
   Example:  insert(3, [2,4]) returns [2,3,4].
*)
fun insert(m, nil) = m
|           insert(m, hd::tl) = if m>n then hd @ insert(m, tl) else [m] @ [n] @ tl


(* isortloop(ns, ans)  calls  insert  to insert the elements of  ns  into  ans.
   Precondition:  ns  is a list of ints and  ans  is a sorted list of ints
   Postcondition: the answer returned is a sorted list of ints containing
     exactly the elements of  ns   and  ans.
   Call it like this:   isortloop(ns, []),
       e.g.,   isortloop([4,2,3,5,1], [])  returns  [1,2,3,4,5]
*)
fun isortloop(nil, ans) = ans
|           isortloop(hd::tl, ns) = isortloop(tl, insert(hd, ns))


(* isort(ns)  calls  insert  to sort  ns.
   Precondition:  ns  is a list of ints
   Postcondition:  the answer returned is a sorted list of ints containing
     exactly the elements of  ns.
   Example:  isort([4,2,3,5,1])  returns  [1,2,3,4,5]
*)
fun isort(hd::tl) = insert(hd, isort(tl))


(* lookup(k, db)   finds the value associated with a key in the database
   params:  k - the key,   db - the database
   returns the value, v,  such that  k,v  is saved in  db.
   If  k  isn't found in  db,  raises LookupError.
   The type of this function should be    lookup : char * DB -> int
*)
fun lookup(k, Leaf) = raise LookupError
|           lookup(k, Update(key, value, rest)) = if k = key then value else lookup(k, rest)
|           lookup(k, Node(key, value, left, right)) = if k = key then value else if k < key then lookup(k, left) else lookup(k,
right)

fun delete(k, nil) = nil
|           delete(k, (key, value)::tl) = if k = key then delete(k, tl) else [(key,value)] @ delete(k, tl)

(* collect(db)  traverses the database and
   returns a list of all the visible (key,value) pairs in  db, removing any entries
   that are cancelled by updates.   For example,
   collect( Update(("b",8),
            Update(("a",5),
              Node("c",6, Node("a",2,Leaf,Leaf), Node("d",7,Leaf,Leaf)))) )
    returns  [("b",8), ("a",5), ("c",6), ("d",7)]
   The type of this function should be   collect : DB -> (char * int) list
*)
fun collect(Leaf) = nil;
|           collect(Update(key, value, rest)) = [(key, value)] @ delete(key, collect(rest))
|           collect(Node(key, value, left, right)) = [(key, value)] @ delete(key, collect(left)) @ delete(key, collect(right));
```

```
(* lookup(x, env)  searches for  x  in  env  and returns the denotable value
   paired with it,  e.g.,
     lookup("b",  Binding("a", Loc(0), Binding("b", Loc(1), Empty)))  returns  Loc(1).
   Raises  EnvLookupError  when  x  is not found in  env.                    binding(a, b, c) --> (a, b):: rest
      lookup : string * Env -> Denotable
*)
fun lookup(x, nil) = raise EnvLookupError
|            lookup(x, (h,t)::rest) = if x = h then t else lookup(x, rest)

(* store(i, v, mem)  returns a store that looks like  mem  with the int
     at location  i  replaced by  v.  E.g.,
       store(1, 4, [3,5,7,9])  returns  [3,4,7,9]
   Raises  StoreError  if  location  i  not found in  mem
    store : int * int * Store -> Store
*)
fun store(i, v, nil:Store) = raise StoreError
|            store(i, v, (hd::tl)) = if i = 0 then [v] @ tl else [hd] @ store(i-1, v, tl)
```