1 Signatures

In the `splay_set.mli` file, we have to declare all the module types.

```
(*
 * Signature for the module that defines
 * the elements in the set.  The elements
 * are totally ordered by the compare function.
 *)
module type EltSig =
sig
   type elt
   val compare : elt -> elt -> int
end

(*
 * Finite sets, and operations on those sets.
 *)
module type SetSig =
sig
   type t
   type elt

   val empty : t
   val mem : elt -> t -> bool
   val insert : elt -> t -> t
   val delete : elt -> t -> t
end
```

We also have to declare the `Make` functor, with a sharing constriant.

```
(*
 * Make a set.
 *)
module Make (Ord : EltSig)
: SetSig with type elt = Ord.elt
```

2 Implementation

The `splay_set.ml` file also has to define all the module types, using the same definitions as the in `splay_set.mli` file (I won't repeat the definitions here).

```
module type EltSig = ...
module type SetSig = ...
```

2.1 Type definitions

First, we need to define the `elt` type (which is the same as the `Ord.elt` type), and we define the type of trees. For this implementation of sets, we also compute the number of elements in

the set as the fourth element in a node.

```
(*
 * Build a set over a totally ordered type.
 *)
module Make (Ord : EltSig) =
struct
   type elt = Ord.elt

   (*
    * Table is a binary tree.
    * Each node has four fields:
    *    1. a key
    *    2. a left child
    *    3. a right child
    *    4. the total number of elements in the tree
    *)
   type tree =
      Leaf
    | Node of elt * tree * tree * int

   (*
    * The tree is mutable
    * so that we can rearrange the tree in place.
    * However, we all splay operations are functional,
    * and we assume that the rearranged tree can be
    * assigned atomically to this field.
    *)
   type t =
      { mutable splay_tree : tree }
```

The hardest part of this problem is to get the `splay` function right. For the solution, we compute a *path* to the node that is to be splayed, which will be passed to a `lift` function that moves it to the root of the tree. The `splay_result` type is used for the return value of the `splay` function; it indicates whether the node was found.

```
   (*
    * Directions are used to define
    * paths in the tree.
    *)
   type direction =
      Left of tree
    | Right of tree

   (*
    * Result of a splay operation.
    *)
   type splay_result =
      SplayFound of tree
    | SplayNotFound of tree
```

2.2 Implementation

The `splay` operation is divided into two parts. The `splay` function computes the path to the node to be splayed, and then it calls the `lift` function to lift that node to the root. The `lift`

function does all the work for the three cases. This code has been expanded for efficiency into 7 cases: 1) the tree is empty, 2,3) the node has no grandparent, 4,5) the node has a grandparent, and it parent is on the same "side" of the tree, 6,7) the node has a grandparent, and it parent is on the opposite "side" of the tree.

```
(*
 * Add two nodes.
 *)
let new_node key left right =
   Node (key, left, right, cardinality left + cardinality right + 1)

(*
 * This function performs the action of moving an entry
 * to the root.  The argument is the path to the entry.
 *)
let rec lift key left right = function
   [] ->
      new_node key left right
 | [Left (Node (key', _, right', _))] ->
      new_node key left (new_node key' right right')
 | [Right (Node (key', left', _, _))] ->
      new_node key (new_node key' left' left) right
 | Left (Node (key_left, _, left_right, _)) :: Left (Node (key', _, right', _)) :: ancestors ->
      lift key left (new_node key_left right (new_node key' left_right right')) ancestors
 | Right (Node (key_right, right_left, _, _)) :: Right (Node (key', left', _, _)) :: ancestors ->
      lift key (new_node key_right (new_node key' left' right_left) left) right ancestors
 | Left (Node (key_right, _, right_right, _)) :: Right (Node (key', left', _, _)) :: ancestors ->
      lift key (new_node key' left' left) (new_node key_right right right_right) ancestors
 | Right (Node (key_left, left_left, _, _)) :: Left (Node (key', _, right', _)) :: ancestors ->
      lift key (new_node key_left left_left left) (new_node key' right right') ancestors
 | _ ->
      raise (Invalid_argument "lift")

(*
 * Find an entry in the tree.
 * Returns SplayFound iff the entry is found.
 * Transforms the tree so that either the
 * entry becomes the root, or an adjacent entry
 * becomes the root if the entry is not found.
 *)
let rec splay compare key0 path = function
   Node (key, left, right, _) as node ->
      let comp = compare key0 key in
         if comp = 0 then
            SplayFound (lift key left right path)
         else if comp < 0 then
            if left = Leaf then
               SplayNotFound (lift key left right path)
            else
               splay compare key0 (Left node :: path) left
         else if right = Leaf then
            SplayNotFound (lift key left right path)
         else
            splay compare key0 (Right node :: path) right

 | Leaf ->
      SplayNotFound Leaf
```

2.3 The `empty` set

The empty set is pretty easy; it is just a `Leaf`.

```
(*
 * An empty tree is just a leaf.
 *)
let empty =
   { splay_tree = Leaf }
```

2.4 The `mem` function

The `mem` function splays the tree and saves it (this is important to get the complexity right). It returns `true` iff the node was found.

```
(*
 * check if a key is listed in the table.
 *)
let mem key t =
   match splay Ord.compare key [] t.splay_tree with
      SplayFound tree ->
         t.splay_tree <- tree;
         true
    | SplayNotFound tree ->
         t.splay_tree <- tree;
         false
```

2.5 The `insert` function

The insert function also splays the tree, and if the node was not found, it adds the new node at the root.

```
(*
 * Add an entry to the tree.
 *)
let insert key t =
   match splay Ord.compare key [] t.splay_tree with
      SplayFound tree ->
         t.splay_tree <- tree;
         t
    | SplayNotFound tree ->
         let tree =
            match tree with
               Node (key', left, right, size) ->
                  if Ord.compare key key' < 0 then
                     new_node key left (new_node key' Leaf right)
                  else
                     new_node key (new_node key' left Leaf) right
             | Leaf ->
                  (* Tree is empty, so make a new root *)
                  new_node key Leaf Leaf
         in
            { splay_tree = tree }
```

2.6 The `delete` function

The delete function splays the tree. If the node was not found, then it saves the result and returns. Otherwise it has to delete the root node and join the two children. In this case, we should splay the left tree so that the rightmost node is at the root, then we can add the right tree as the right child of the root. (We could also splay the right tree so that the smallest node is at the root). To do this, we call the `splay` function with a comparison function that always returns `-1`.

Now the `delete` function can be defined using the `splay_right` function.

```
(*
 * Remove the first entry from the hashtable.
 * If the value list becomes empty, remove the
 * entire entry from the tree.
 *)
let delete key t =
   match splay Ord.compare key [] t.splay_tree with
      SplayFound tree ->
         begin
            match tree with
               Node (_, Leaf, right, _) ->
                  { splay_tree = right }
             | Node (_, left, Leaf, _) ->
                  { splay_tree = left }
             | Node (_, left, right, _) ->
                  begin
                     match splay (fun _ _ -> -1) key [] left with
                        SplayNotFound (Node (key, left_left, Leaf, _)) ->
                           { splay_tree = new_node key left_left right }
                      | _ ->
                           raise (Failure "Fun_splay_set.remove")
                  end
             | Leaf ->
                  raise (Failure "Fun_splay_set.remove")
         end
    | SplayNotFound tree ->
         t.splay_tree <- tree;
         t
```