1 Splay trees

One of the most important data structures used in compilers are *sets* and *tables* of arbitrary elements. For example, when we do type checking, we will need a table that contains the type of each bound variable. This table can be used both to check that the variable is being used properly, and it can also be used to check for unbound variables.

Since sets and tables are used so frequently, their implementation has a major impact on the performance of the compiler. We need an *efficient*, *functional* table implementation.

One suitable data structure is a tree representation. For this assignment, we'll implement *functional splay trees*. This description is taken from Kozen, "The Design and Analysis of Algorithms," (Springer, 1991).

A functional splay tree is a data structure invented by Sleator and Tarjan. It is an ordered binary tree S, that supports the following *functional* operations.

- **member**(i, S): determine whether element i is in splay tree S
- **insert**(i, S): insert element i into splay tree S (return a new tree S')
- **delete**(i, S): delete element i from splay tree S (return a new tree S')

All operations have an amortized cost $O(\log n)$. The most interesting thing about splay trees is that, unlike red-black trees or 2-3 trees, it is not necessary to rebalance the tree after each operation—it happens automatically.

The splay tree operations are all implemented in terms of a single, basic operation called **splay**.

- **splay**(i, S): reorganize the splay tree S so that element i is at the root if $i \in S$, and otherwise the new root is either

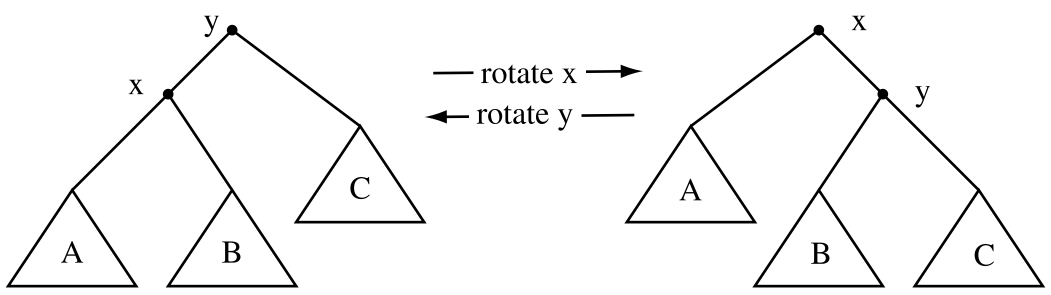
$$max_{k \in S} k < i \quad \text{or} \quad min_{k \in S} k > i.$$

All of the other operations can be implemented in terms of **splay**.

- **member**(i, S): call **splay**(i, S) to bring i to the root if it is there, then check the root against i. The result of the splay should be saved.
- **insert**(i, S): call **splay**(i, S) to produce a new tree S'. If i is not at the root, create a new root node labeled i, and split S' to produce the children.
- **delete**(i, S): call **splay**(i, S) to bring i to the root if it is there; then remove i and join the two subtrees.

2 Implementation of splay

The **splay** operation can be implemented in terms of an even more elementary operation **rotate**. Given a binary tree S and a node x with parent y, the operation **rotate**(x) moves x up and y down, according to the following picture.

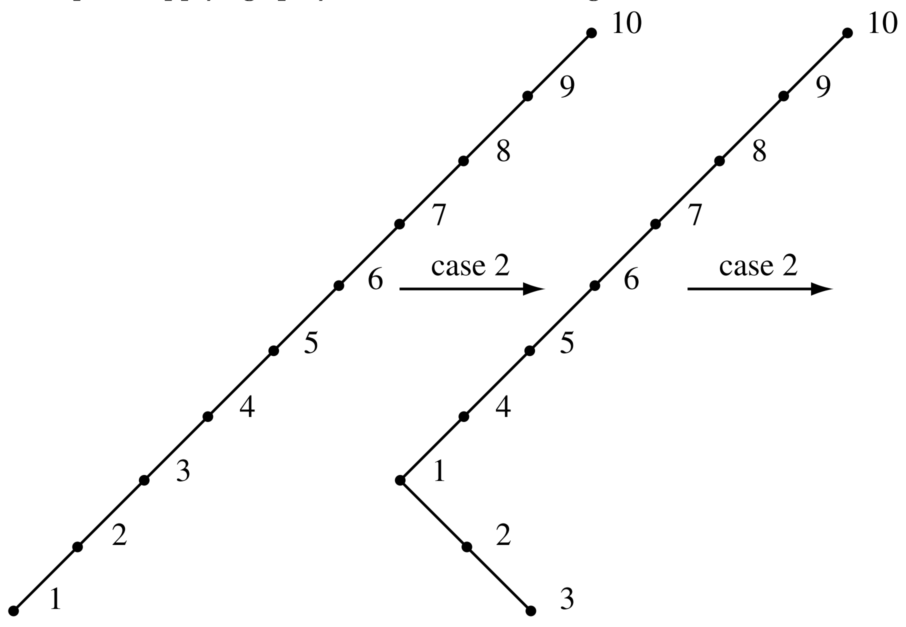


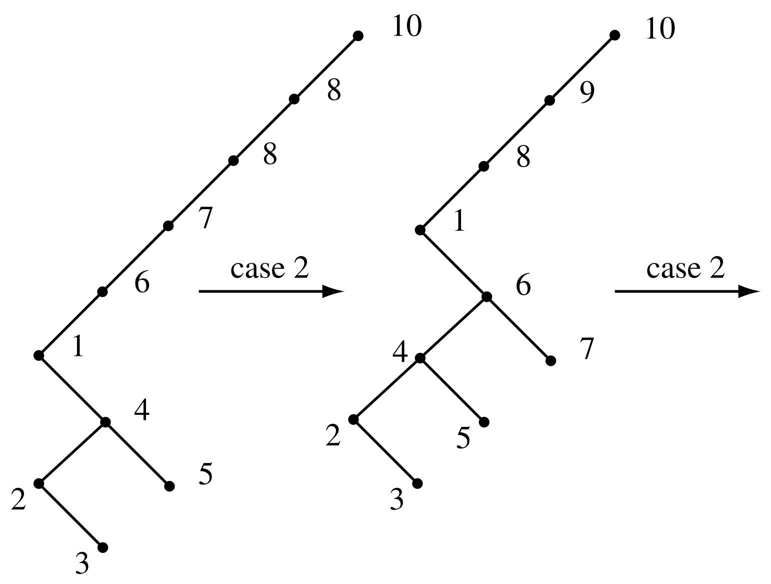
Note that the **rotate** operation preseves the inorder numbering of the tree.

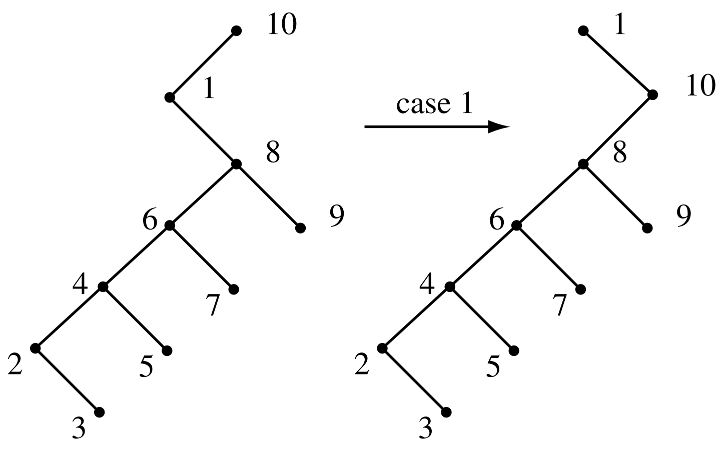
To implement **splay**(x, S), we distinguish three separate cases:

1. If x has a parent, but no grandparent, we just **rotate**(x).
2. If x has parent y and a grandparent, and if x and y are either both left children or right children, we first **rotate**(y) and then **rotate**(x).
3. If x has parent y, and a grandparent, and if one of x or y is a left child and the other is a right child, we first **rotate**(x) and then **rotate**(x) again.

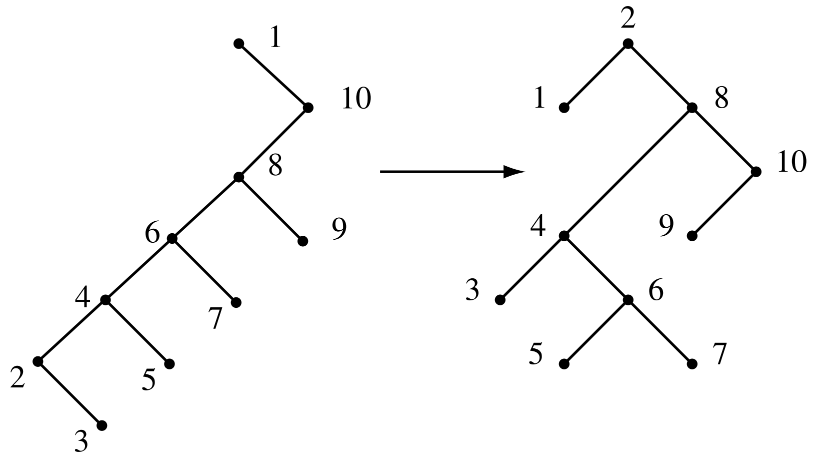
Here is an example of applying **splay**$(1, S)$ to the following tree S:







Aplying **splay** 2 to the resuling tree yields:



Note that the tree seems to become more balanced with each **splay** operation.

3 Getting Started

You will need to use the OCaml module system to help with this assignment. You should use splay trees to implement a module with the following signature.

```
module type SetSig =
sig
   type elt     (* type of elements in the set *)
   type t       (* type of sets *)

   (* Create a new set *)
   val create : unit -> t

   (* Test for membership *)
   val mem : elt -> t -> bool

   (* Add an element to the set *)
   val insert : elt -> t -> t

   (* Delete an element from the set *)
   val delete : elt -> t -> t
end
```

The type of elements is also defined as a module with a `compare` function. The `compare` function takes two `elt` arguments and returns a) a negative number of the first argument is smaller than the second, b) zero if they are equal, or c) a positive number if the first argument is larger than the second.

```
module type EltSig =
sig
   type elt    (* type of elements *)

   (* Comparison function *)
   val compare : elt -> elt -> int
end
```

Your splay set implementation will be implemented as a *functor* that takes a module of type `EltSig`, and returns a module with type `SetSig`.

```
module MakeSplaySet (Elt : EltSig) =
struct
   (* Same type of elements *)
   type elt = Elt.elt

   (* Splay trees *)
   type tree =
      Leaf
    | Node of tree * elt * tree

   (* Have to use a reference cell to save result of splay *)
   type t = tree ref

   (* Splay operation *)
   let rec splay x = function
      Leaf -> Leaf
    | Node (left, y, right) ->
         ...

   (* Membership *)
   let mem x s =
      let s' = splay x !s in
      let found = (* check if x is the root *) in
         s := s';
         found

   ...
end
```

The type of splay trees (t) uses a reference cell. You should use this cell to save the result of the splay after a mem operation. Note that the set will still appear functional, since the **splay** operation does not change the set membership.

The functor has to be declared with a *sharing constraint*. The splay_set.mli file will contain a declaration like this.

```
module MakeSplaySet (Elt : EltSig)
   : SetSig with type elt = Elt.elt
```

4 What to turn in

You should place the following in your submit/lab1 directory on mojave (contact the TAs if you don't have an account).

- A README file that describes your implementation.
- The splay_set.mli and splay_set.ml files that implement the splay trees.
- A test.ml file that defines a test program for your splay set.