Burt Rosenberg

Problem Set 8

Out: 28 October, 1993
Due: 5 November, 1993

Goals

Implementing a balanced tree structure.

Reading Assignment

Read Chapters 10, 14 and 15 from *Algorithms.*

Introduction to Splay Trees

When using linked-lists we found a useful heuristic *move-to-front.* A similar heuristic for trees would be *move-to-root*: whenever an element x is accessed, the tree is mutated to move x to the root. A *splay tree* does this, but also improves the overall balance of the tree with each mutation. After x is "splayed" to the root of the tree, the tree is shorter and bushier than it was before the splay.

The splay is a sequence of local tree transformations called *rotations.* The diagrams below show how to rotate a tree at x. Node y is the parent of x before the rotation, and A, B and C represent subtrees attached to x and y. (They might be empty.)

```
If x is a left child of y:
         y              x
        / \     @x     / \
       x   C   ===>   A   y
      / \               / \
     A   B             B   C

If x is a right child of y:
       y                x
      / \       @x     / \
     A   x     ===>   y   C
        / \         / \
       B   C       A   B
```

It is important to note that the search tree order is not disturbed by a rotation. Consider in detail the first form of rotation. The initial configuration informs us that the nodes are in size order:

$$(a \in A) < x < (b \in B) < y < (c \in C),$$

that is, any node a in the subtree A has a smaller key than the key of x, and the key of x is smaller than that of any node b in B, and so on. The final configuration agrees with this order. The second form of rotation exhibits this agreement as well: both the initial and final configurations order the nodes as:

$$(a \in A) < y < (b \in B) < x < (c \in C).$$

It was worked out by some extremely clever people how to glue together rotations to make a splay. It does not work to repeatedly rotate at x until x appears at the root! It is true that this simple recipe will result in x at the root, and will not harm search tree order, but it will not result in a tree of overall improved balance. The trick, it turns out, lies it changing the order of rotations for the case that x is the left child of a left child, or right child of a right child.

Let us first deal with the easy cases. If x is the root, there is nothing to be done. If x is a child of the root, then rotate at x and you are done. Else, x has a grandparent, call it z, and a parent, call it y. Node x stands in one of two relations to its grandparent,

1. x forms a *zig-zag* with z if either x is a right child of y and y is a left child of z, or x is a left child of y and y is a right child of z. In this case, rotate twice at x.

```
    z              z             x
   / \     @x     / \     @x     / \
  y   D   ===>   x   D  ===>   y   z
 / \            / \           / \ / \
A   x          y   C         A  B C  D
   / \        / \
  B   C      A   B
```

2. x forms a *zig-zig* with z if either x is a left child of y and y is a left child of z, or x is a right child of y and y is a right child of z. In this case, rotate first at y and then at x.

```
      z              y             x
     / \     @y     / \    @x     / \
    y   D   ===>   x   z  ===>   A   y
   / \            / \ / \           / \
  x   C          A  B C  D         B   z
 / \                                  / \
A   B                                C   D
```

The drawings show only one of two variants for each the zig-zag and zig-zig.

Inserting and Deleting in Splay Trees

A *splay tree* is a tree structure which, in addition to the usual tree operations, supports the operation "splay at x", for any node x in the tree. Insertion in a splay tree proceeds according to insertion of a normal binary tree, however, it is finished up with a splay at the newly inserted node, thus bringing it to the root. This insures that no matter how fiendishly the input data is arranged, the tree will remain within height $O(\log n)$ for an n node tree. Since search and splay run in time proportional to the height of the tree, insertion therefore is an $O(\log n)$ operation worst-case. This is good.

Deletion in a splay tree is easily accomplished using only splay operations. Suppose x is in the tree and is to be deleted. Splay x to put the tree into the form:

```
  x
 / \
A   B
```

Now delete x, preserving pointers to A and B. Find the smallest item y in B and splay y, now the situation is:

```
      y
A      \
        B'
```

so make A the left child of y and you are done. Deletion is also easily seen to be an $O(\log n)$ operation in an n node tree.

The Assignment

Write a program implementing a splay tree, with the following operations,

```
function SplayTreeCreate : SplayTree ;
   { Returns a newly created, empty Splay Tree. }
function SplayTreeIsEmpty( st : SplayTree ) : boolean ;
   { Returns true if and only if st is empty. }
procedure SplayTreeSearch( st : SplayTree ;
                            d : DataType ) : boolean ;
   { Searches st for d.
     If found splays at d, bringing it to the root
        and returns True.
     Else st is unchanged and returns False. }
procedure SplayTreeInsert( st : SplayTree ; d : DataType ) ;
   { If d is not in st, inserts it into st and moves
        it to the root.
     If d is already in st, moves it to the root. }
procedure SplayTreeDelete( st : SplayTree ) ;
   { Removes the item at the root of st. }
function SplayTreeAccess( st : SplayTree) : DataType ;
   { Returns the value of the element in the root of
     st. If st empty, returns the empty value for the
     type DataType. }
procedure SplayTreePrettyPrint ( st : SplayTree ) ;
   { Pretty-Prints splay tree st, that is, in
     preorder with indentation. }
procedure SplayTreePrint ( st : SplayTree ) ;
   { Prints st in-order. }
```

Put these into a program which has two modes, test and run, depending on the value of the `CONST` variable `DEBUG`. In test mode, words are taken from the keyboard and inserted into an initially empty tree. If the word begins with "–", then the dash is stripped and the word deleted from the tree. After each insert or delete a SplayTreePrettyPrint is commissioned. In run mode, text is taken from a named file and only a single SplayTreePrint is commissioned at the close of execution.