Burt Rosenberg

Problem Set 5

Out: 10 October, 1995
Due: 24 October, 1995

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 of integers. Integers are taken from the keyboard and, found, inserted or deleted in an initially empty tree, according to the sign of the integer. If the integer is 0, exit the program. If the integer is positive, then insert the integer if it is not already in the tree, splaying the inserted element to the root, or splaying to the root the already present element. If the integer is negative, change the sign and delete the resulting integer, if it is found in the tree, else leave the tree unchanged. After each operation, print the splay tree.

This is a tricky program to write. There will not be a template for you to work from, but you are to adopt some sort of plan of attack, similar to having a template. Here is a recommended plan of attack:

1. Start from your already working tree program of the last homework. You might get rid of the deletion software, and you should upgrade to a "dummy" root node.

2. Write functions `rotate_left` and `rotate_right`. These functions take a tree node pointer as argument and return a tree node pointer. Test on your simple tree.

3. Write a function `splay_at` that does a single splay operation on a node, i.e. zig-zig or zig-zag to the left or right. Use the `rotate_left` and `rotate_right` functions which you have working. A simple calling method is to give a tree node pointer and an integer as argument, with the assertion that the integer is a grandchild of the pointed to node, and returns a tree node pointer of the splayed result, the grandchild indicated by the integer argument having been brought to the root. Test on your simple tree.

4. Begin to build the recursive splay function, called `splay_aux`. The function does a find on the way down and a series of splay-at's on the way up. The tricky part is the splay-at's are done only on every even number of returns. Create a global variable `tricky` and set it EVEN or ODD, reversing its polarity each return level. Splay-at when tricky is EVEN.

5. Place the recursive splay function in a non-recursive shell which also handles the special case of splaying at the root. Call this function `splay`, which takes a splay tree (pointer to a dummy root) and a value and returns void or an integer flag, but the splay tree has been modified. The special case for roots is that if splay sees tricky ODD it does a rotate rather than a splay-at.

6. Finish up the program with routines for insertion and deletion and so on. Add toppings and salt to taste.

Please recall and abide by the class policy on group effort and homeworks. This course is meant to be challenging. The department has provided a tutor and students are encouraged to help each other to master the material. But it does no good to simply copy another student's code! Once you believe you understand the program, go off by yourself to write the code.

Work slowly, writing just five lines of code between each compilation. Set up meaningful stopping points along the route to the final program. This generally means attacking a problem via small detours ... solving an easier problem first then dismantling some of the easier problem before heading again forward towards a more difficult goal. These extra lines of code are called "scaffolding". Some expert programmers estimate that half the lines they write are scaffolding, which are removed and thrown away before the project is completed. Take big steps only if you don't have far to fall.