CS 2420 – Splay Trees

Program 3 – 20 points  Due Oct 1, 2010

Part 1: You are to create a splay tree using a small data set.

For ease it coding, we will do a “top down” splay, in which we start splaying before we ever get to the node we care about. This makes it a bit easier than what the book shows in Chapter 4 as we don’t have to find the node and look up two levels. We make another simplification. Instead of doing the double rotation (for a zigzag case), we just go down ONE level and do a left or right rotation (a zig). This makes the coding easier as we don’t have to code a double rotation.

Create a splay tree abstract data type, with the following operations:
- void insert(string)
- bool find(string)
- string deleteMax()
- string deleteMin()
- printTree(string msg): prints the msg before printing the tree prettily [Print prettily means that nodes are indented to show their level in the tree]
- makeEmpty(): remove all nodes from the tree, recursively

Create a splay tree from tiny3.txt and test the various operations. TestSplay.cpp contains code to do this checking. (Leave this code in your program so the grader can verify all operations work.)

Part 2: Create a new splay tree from the input dictionary3.txt. Read in search3.txt and count the number of words which were not found. TestSplay.cpp contains the code to display the output.

Hints:
Note, the splay routine is used in many other operations.

I had a the following enumerated type
enum DIR {DONE, LEFTLEFT, LEFT, RIGHT, RIGHTRIGHT};

And a routine to print the direction as a string (for use in debugging).

template <class Etype>
string SplayTree<Etype>::: getDir(DIR d)
{
    switch(d) {
        case DONE: return "DONE";
        case LEFTLEFT: return "LEFTLEFT";
        case LEFT : return "LEFT";
        case RIGHTRIGHT: return "RIGHTRIGHT";
        case RIGHT : return "RIGHT";
    }
}
return "OOPS";
}

I had a routine which, when given the current root and the value I was looking for, it
would return the direction needed to get there. So if the value I sought was in my left
child's left child, I would return "LEFTLEFT". If it was in my left child's right child, I
would return "LEFT" (as we only do single rotations). This helped me to separate
functionality into a named routine which I could easily describe.

Clarifying Example
In the following example, notice that leftMax and rightMin are pointers which move to the next place of
insertion. In this small example, it is difficult to see the advantage, but in a huge tree, having leftMax
means you don’t have to search from the root for an insertion point for the newly cast off part of the tree.

Original Tree before trying to find node n:
Splay on n (zigzag)

leftExtra

a

h

m

leftMax

f

k

l

j

i

c

c

d

e

rightExtra

q

rightMin

s

r

t

Tree

Splay on n (reassemble)

Attach n’s kids to rightExtra and leftExtra