CSE 332: Data Abstractions
Assignment #4
July 11, 2011
due: Friday, July 22, 10:00 p.m.

Implement the Dictionary class using splay trees. Here are the details:

1. Implement the procedures `rotate(t,dir)`, `splay(k)`, `lookup(k)`, `insert(k)`, `concat(t1,t2)`, and `delete(k)` as described in Section 4.5 and the attached Figures 7.17-7.20 from Lewis and Denenberg's textbook *Data Structures & Their Algorithms*. The procedure `rotate(t,dir)` does a single rotation at node `t` with the child in direction `dir` (either L or R) and is used by `splay`. Also implement a procedure `display` that you can invoke after each Dictionary operation to display the entire splay tree, so that you can watch its shape change. For the display, use preorder traversal and the outline form as in Figure 4.7. If a node does not have a sibling, it must be made clear whether it is a left or right child, so print out a "−" to indicate the empty sibling.

   With the exception of `splay`, these procedures are very short and easy, so don't be concerned about the number of procedures you have to implement. The procedures `lookup`, `insert`, `delete`, and `display` should all be public members of your Dictionary class. You are free to modify the interface and implementation of the private members, as long as they accomplish their task by the same algorithms as given in the text.

   The data type for the key field should be `int`. There is no need to have an `info` field for this assignment; just keep in mind that in a real application there would be one. Since there is no `info` field, `lookup` should simply return a boolean that is `true` if and only if the key was found.

   For full credit, your data structure should not have a parent pointer for each node. Instead, design `splay` recursively, so that the recursion stack will do the job of remembering the path back to the root. In fact, the only fields in each node should be the key, left child, and right child.

   (Hint: Because the type of rotation done depends on the path to the splayed node X from the grandparent of X, you need some ancestral context after returning from a recursive call. Design your procedure so that when the recursive call returns, X is either at depth 1 or depth 2 from the current root. If it is at depth 1, then simply return without doing any rotation; if at depth 2, then do the appropriate two single rotations before returning. You will need the recursive procedure to store X or the key at X, in order for the invoking procedure to find X from its current root. Notice that when all the recursion ends, X may be left at depth 1 of the entire tree, so some zig cleanup may be necessary.)

   (Antihint: It may occur to you that each recursive call could leap down 2 levels rather than 1, and then do the obvious zig-zag or zig-zig rotation when the recursive call returns. This won't work. The problem is that, if the path length to the splayed node is odd, then you will do the zig rotation as the very first rotation rather than the very last. This results in entirely different splay behavior from the algorithm in the book, and I cannot give you any guarantee that the amortized analysis holds anymore.)

   To implement `concat`, you can use any convenient key from T_2 in place of $+\infty$. If T_2 is empty, it is not necessary to splay T_1 at all.

2. Your program should be called `RunDictionary` and will take two filenames as arguments. The first one is the input file. If it does not already exist, your program should print a warning and exit. The second one is the output file. It should be created if it does not exist and overwritten if it does. The input file should consist of a sequence of Dictionary commands, one per line, each command being of one of the three following forms:

insert n
delete n
lookup n

where n is an integer. The output file should contain the displayed splay trees after each command in the input file is executed, starting from an initially empty Dictionary. If your program encounters errors in the input file (such as a word other than insert, delete, and lookup), output an appropriate error message to the output file and quit your program. We should be able to invoke your program from the command line via the command

```
java RunDictionary infile.txt outfile.txt
```

3. Run some experiments with your Dictionary package, trying to find a sequence of operations from an initially empty tree that causes some of its operations to take $\Omega(n)$ time (even though the average time must be $O(\log n)$). What happens to the shape of the tree after a few such expensive operations? Once the tree is balanced, can you find some operations that cause it to become completely unbalanced? Include a short report on these experiments in a `README` file.

   Keep a counter of the total number T of single rotations done over the course of all m Dictionary operations in the input file. Since the running time is proportional to the number of single rotations, T is a measure of total running time. Let n be the maximum size of the dictionary during these operations. Run enough experiments with various large values of m and n so that you can produce the following plot: graph T/m on the vertical axis and $\log_2 n$ on the horizontal axis (equivalently, plot n using a log scale for the horizontal axis). If everything goes well, these points should lie on (or below) a roughly straight line whose slope will tell you the constant factor c in the relation $T \leq cm \log_2 n$. Estimate c and include this estimate in your `README` file. What I suggest for these experiments is that your operations consist of n insertions followed by $m - n$ lookups, where each of these lookups splays one of the deepest nodes in the splay tree. If you do this, hopefully most of your plotted points will lie close to the straight line. Choose n to be successive large powers of 2 and $m >> n$. If you run into a java stack overflow, you can increase the allocated stack for the virtual machine to 256 Mb with

   ```
   java -Xss256m RunDictionary in.txt out.txt
   ```

   Turn in a spreadsheet file with your plotted graph. You will want to turn off production of the output file during these experiments, because it would be huge.

Turn in all your source, README, and spreadsheet files at

`https://catalyst.uw.edu/collectit/dropbox/summary/tompa/16472`