Red Black Trees : Building and Searching
Lab Assignment #8

by
Justin Anthony Timberlake

CS 303 Algorithms and Data Structures
October 18, 2017

1. Problem Specification

The purpose of this lab was to first implement an RBTree class with the methods for inserting new nodes into the tree, printing out the values in an in-order traversal walk of the tree, and also a search inherited from BST, which takes a starting RBNode and a key value (in this case, a long) to locate specific values in the RBTree. The next goal was to read in the "UPC-random.csv" file which contained all the records, in which to search for the given search keys and records from the "input.dat" file.

2. Program Design

The steps taken to develop the program were:

a) Extend the existing Node class to be "RBNode" that also has a boolean color value field, isRed followed by extending the BST class to be the RBTree class that now contains leftRotate, rightRotate, RBInsert, and RBInsertFixup instead of treeInsert and the BST constructor.

b) Generate some simple integer inputs to build the RBTree upon and demonstrate the inOrderTreeWalk for the in-class portion of the lab.

c) Copying over the input/output code from the main method of the last lab, change "UPC.csv" to "UPC-random.csv" and otherwise leave everything exactly the same, as the assignment is near identical with a different data structure and input file. All necessary printouts and benchmarks are already added in print statements.

No additional methods were used in the primary class.

The following methods were used in secondary class, RBNode.

a) RBNode(long thing, String words) - constructor for objects of type RBNode, setting the data equal to the given String and the key value equal to the given long.

The following methods were used in secondary class, RBTree.

a) RBTree() - base constructor for objects of type RBTree.

b) RBInsert(Node z) – Puts the given node z in its proper place within an existing RBTree based on comparison values of the keys, which are long values.

c) inOrderTreeWalk(Node x) – starting at the x node, recursively prints out values of the keys of the left child, the current node, and then the right child, recursively. This generates sorted order.

d) treeSearch(Node x, long k) – starting at node x, this method searches recursively through the BST for the given long value, k. For searching through large trees recursively, stack overflows can occur with the use of recursion (using the stack). However, when testing my program, all such issues were not present in the final product of the code.

e) leftRotate(RBNode x) – shifts the RBTree in position structure in the case for which parent of x is the left child of the grandparent of x.

f) rightRotate(RBNode x) – shifts the RBTree in position structure in the case for which parent of x is the right child of the grandparent of x.

g) RBInsertFixup(RBNode z) – Given a starting node to iterate through the RBTree, recolors the RBNodes as necessary in order to maintain the RBTree properties.

3. Testing Plan

Once the RBTree and RBNode classes were built and tested in-class and only a few modifications were made for the homework portion, the only thing left to test was the actual time elapsed performing the operations mentioned in this assignment. This was calculated by getting the nanoTime() directly before and after each insert or search operation only and then adding their respective totals as the loop continues to parse and generate results to be searched, as seen in the code. These two values are then printed out for benchmark testing purposes. The specific descriptions of the given search values are printed after the bench mark of each of the operations to signify that all the elements have been found.

4. Test Cases

For building the given 177, 650 records in the "UPC-random.csv" file into a RBTree, the total time elapsed was 1,779,015,687 nanoseconds in a given test, whereas giving an already sorted list of items to the BST took 76,461,176,294 nanoseconds.

For searching the given 17 values in the "input.dat" file out of the aforementioned 177,650 records, the total time elapsed was 142,619 nanoseconds, where this same search took 1,986,371 nanoseconds with the BST.

5. Analysis and Conclusions

Though the BST was incredibly fast in delivering search values from within "UPC.csv", RBTree proved to do the task much faster on its own list. However, in addition to a faster search time based on the (RB) tree built from the values read in (this time from "UPC-random.csv", the values were read in from an unsorted, randomized list, meaning that the time for building the RBTree was significantly less in this case than building the BST with the same number of values because the randomized values developed a more balanced tree, thus taking about 1/76th of the total time based on each benchmark of building time.

6. References

All code in this assignment was either presented in prior labs and assignments or provided through means of pseudocode within the in-class portion of the lab.