Lab 6 - Trees
Due (via Bitbucket and hard copy) Wednesday, 4 March 2015
50 points

Lab Goals

• Learn how to traverse Binary Search Trees
• Evaluate the performance of adding nodes to Red-Black Trees
• Answer some questions about the application of trees to search problems

Assignment Details

We have spent the last few classes talking about using trees as a data structure to hold our symbol table. In this lab, we will write a few short functions that demonstrate ways that we can work with these trees and evaluate their performance.

Part One: Traversing Binary Search Trees (20 points)

One useful feature of Binary Search Trees is that the keys are ordered. We know that if we visit any node of a tree, its left subtree only contains smaller keys, and its right subtree only contains larger keys. Because of this, we can traverse the tree to obtain a sorted list of keys. First, we recursively print all of the keys in the left subtree, then we print the key of the current node, and finally we recursively print all of the keys in the right subtree. This is called In Order Traversal.

For example, consider the tree in Figure 1 on the next page. Our goal is to print a sorted list of keys A-B-C-D-E-F-G-H-I. To do so, we begin at the root node F. We recursively call a function that will print the In Order traversal of the left subtree. This recursive call, now rooted at node B, will again call itself, moving the node under consideration to node A. Because node A has no left subtree, the recursive call does not occur. Instead, node A prints its key, attempts and fails to print its right subtree, then returns the recursive call back to node B. Since the left subtree call has completed, node B can then print its key. Next, node B will recursively call a function to print the In Order traversal of its right subtree. And so the process continues until the last node has been printed.

You should implement a function to print both the Key and Value of each node in a Binary Search Tree using this In Order traversal method. You may use the BST.java file provided in the src folder as your Binary Search Tree data structure, or you can write your own from scratch. You can test your In Order traversal using the tinyTale.txt file also located in the src folder. Your output should look similar to the following:
In Order traversal: (key, value)
--------------------------------
(age, 21)
(belief, 29)
(best, 3)
(darkness, 47)
(despair, 59)
(epoch, 33)
...

Once you have implemented In Order traversal, you should also implement two other traversal methods. The first, known as Preorder Traversal, prints the current node, then the left subtree, and finally the right subtree. Using the tree from Figure 1 as an example, the key order printed will be F-B-A-D-C-E-G-I-H. You can think of Preorder traversal as “swimming” around the tree counterclockwise, printing out each node the first time you see it.

The second traversal method, known as Postorder Traversal, prints the left subtree, then the right subtree, and finally the current node. You can similarly think of it as “swimming” around the tree counterclockwise, but this time you will print out each node the LAST time you see it. Using the tree from Figure 1 as an example again, the key order printed will be A-C-E-D-B-H-I-G-F.

Part Two: 2-3 and Red-Black Trees (20 points)

We noted in the lecture that 2-3 Trees and Red-Black Trees are equivalent data structures. We represent a 3-Node by connecting two nodes with a Red link, signifying that they are treated as a single node. Because of our rules governing these Red links (they must go left, there should only be one per parent node), we had to make our tree self-correcting through the use of the rotateLeft(), rotateRight(), and flipColors() functions. Processing these structural swaps
involves some additional computation time. The question is, how much extra?

In this section, you will write a function to compute the number of Red links in a randomly generated Red-Black Tree. You can count the number of Red links by traversing the tree similarly to how you traversed it in Part One. You can use the RedBlackBST.java file included in the src folder as your data structure, or you can write your own from scratch.

You will also need to add a timing function to measure the time it takes to create your tree. You should insert $N$ randomly-generated keys (integers or floating-point keys and values are fine) into your Red-Black Tree, measuring the time it takes to add the keys. Try $N$ values of 10,000, 100,000, and 1,000,000. You should perform at least 10 runs for each $N$ size to get a reasonable sample.

Our goal is to see if there is a relationship between the time required to build the tree and the number of Red links in the completed tree. You can determine that relationship by plotting your results for each trial on a graph. Your x-axis should be the number of Red links, and your y-axis should be the time required to construct the tree. Do you see any relationship? (In other words, do trees with more Red links take significantly longer to build?) Do you think the additional computation time for Red links is a significant penalty to the performance of a Red-Black Tree?

Part Three: While You Have Some Downtime (10 points)

Answer the following questions thoroughly:

1. In a Binary Search Tree, is the delete() operation commutative? (Does deleting $x$ then $y$ always give the same resulting tree as deleting $y$ then $x$? Give an example to back up your answer.

2. Draw the 2-3 Tree that results when you insert the keys REDBLACKXYZ in that order into an initially empty tree.

3. Draw the Red-Black Tree that results from inserting the keys ABCDEFGHIJK in order into an initially empty tree. What happens in general when trees are build by inserting keys in ascending order?

4. Find a sequence of keys to insert into a Binary Search Tree and into a Red-Black Tree such that the height of the Binary Search Tree is less than the height of the Red-Black Tree. Alternatively, prove that no such sequence of keys is possible.
Submission Details

For this lab, please submit a paper copy of everything listed below. Additionally, please upload all of your files to a folder in your BitBucket repository clearly labeled as “Lab 6.”

1. Your source code for traversing the Binary Search Tree, as well as sample output showing In Order, Preorder, and Postorder traversals of tinyTale.txt.

2. Your source code for counting the links of a Red-Black Tree, as well as your data and graph from your trials.

3. The answers to the questions in Part Three.


Please remember that all files that you submit should be your own work, though you are welcome to discuss high-level topics and algorithms with classmates.