Programming Project 4

CS146 Data Structures and Algorithms

Benjamin Garcia Section 02

Introduction

For Project 4, we were asked to create a working spell checker that spell checked all words in a given file using a dictionary file that would be implemented as a Red Black Tree. We first would have to first get all the words from the dictionary file and the file that is being spell checked and store it. We then add all the dictionary words into a Red Black Tree. The spell checker then looks up every single word and sees if it is in the dictionary or not.

Background/Review

A Red Black Tree is a special type of binary search tree. A binary search tree must hold the properties that: the value in every node is larger than the value of the left child (or any value in the left subtree) and smaller than the value of the right child (or any value in the right subtree). The additional properties that make it a Red Black Tree is that: every node is either red or black, every leaf is black, if a node is red then both children are black, every path from the node to the descendent leaf contains the same number of black nodes, and the root is always black. If a tree follows all these properties, it ensures that the tree is balanced, and that one side is not much longer than the other. The longest path from the root to a leaf in a Red Black tree cannot be more than twice the shortest path from the root to a leaf. This means that all operations are always O(lgn). Because the dictionary gets implemented as a Red Black Tree, lookup times are very fast for each word and makes a spell checker super-efficient.

Project Specification

The first thing we had to do was create a method that would grab all the words in both the dictionary file and the file being spell checked. I did this by having a Scanner scan through each line and extract all words, remove any special characters, and making the word all lowercase. It would then add each word to an ArrayList and return that. This was good enough to store the file being checked. But, to implement the dictionary, I had to use a Red Black Tree. To do this, there were two classes: a class for the Red Black Tree (RBTree) and a class for the nodes of the tree (Node). The tree would be initialized with a null root. Every node created would be initialized with the proper data and both the children null. To insert a new root, I would first have to find the proper position by traversing the Red Black Tree. If the value being inserted was less than the node, go left, otherwise go right. It would do this until a child was null which meant there was a spot to insert the node. Insert the new node here and then run a method called fixTree that would fix the tree to verify the Red Black Tree properties were maintained. This was done by using multiple cases and two functions, rotateLeft and rotateRight. In each case, if needed, the subtree was rotated, and surrounding nodes were recolored to maintain these properties.

Cases to Consider

A special case I had to consider was adding a node when the tree was empty. In this case, that meant the node being added would have to be the root. So, set the root to the new node and continue

inserting other nodes onto the root. Another case that was considered often was if the node I was trying to address was null. This led to a lot of Null Pointer Exceptions throughout my testing. I had to put in a lot of checks to do certain things if a node was null versus when it was an actual node.

Problem Analysis and Solution Design

As previously mentioned, the biggest problem I ran into were Null Pointer Exceptions. I had to insert checks for null pointers in almost all the methods. Because binary trees have so many null nodes as well as Red Black Trees having to check and reassign so many different pointers to fix the tree, it became obvious that almost every time a pointer was referenced, it should be prefaced by an if null condition.

Running Time

When calculating the running time, I was super surprised. The time to take the dictionary file and implement it as a Red Black Tree took anywhere from 600-800 milliseconds. But the time it took to perform lookup calls for all words in the poem took under 1 millisecond. This really caught me off guard and showed me how efficient a balanced binary tree really is.