Lab Report 06

Nathan Bickel

Problem

The problem was to allow the user to input however many Strings they wanted, and then for the program to sort the Strings they input in ascending order based on the number of times the word “sort” (or other capitalizations) was contained in the String. For example, “sorting is an important concept that can be implemented in Java in a variety of sorting algorithms” would contain two instances of “sort”, as would “sOrTs orTSOrT!?” The sorting was to occur with O(nlg(n)).

Solution

The program involves a number of internal classes to organize the data. They are each described:

StringCounter is a fairly simple class that combines the String the user input with the number of times it contains “sort”. In the constructor, each substring of the length of the target is checked against the target. In this case where the target is “sort”, each substring of 4 characters is checked to see if it is equal to “sort” (so the first four characters were checked, then the second through fifth, up to the last four). The integer numWords is incremented every time there is a match.

LinkedList is a fairly typical linking structure of type StringCounter. It is composed of ListNodes, which each contain data and a ListNode linking to the next node in the list. The class has an **add** method where the list is iterated through until the end and then a new node is connected there, and a **print** method that iterates through and prints out the data (specifically the String the user inputs and not the number of “sorts”) at each ListNode.

SearchTree is a bit more complicated—it is a binary search tree composed of LinkedLists. It has an **add** method that takes in a StringChecker as a parameter, and then determines which LinkedList to add it to. The method is first called on the root, and it determines where to go from there:

- If the node is null, a new linked list is created and the StringChecker is added as the first element.

- If the StringChecker's numWords is less than that of the head of the linked list at the node, the method is recursively called on the node's left child.
- If the StringChecker's numWords is equal to that of the head of the linked list at the node, the StringChecker is added to the LinkedList using its **add** method.
- If the StringChecker's numWords is greater than that of the head of the linked list at the node, the method is recursively called on the node's right child.

The tree also contains a **print** method that uses recursion to pass through each node and call the LinkedList at that node's **print** method. It first goes left until it can't anymore, prints, and then begins backtracking and printing values as it goes. This means that nodes will be printed from most left to most right, which corresponds to ascending order based on the StringCheckers' numWords attributes in the LinkedList. Thus, this will print the tree out in the order that we want it.

Since these internal classes provide most of the functionality required, the main method was fairly simple. First, a Scanner and a SearchTree are constructed (the Scanner is actually constructed outside of the main method). Then, the program greets the user and allows the user to input Strings until they're done and enter "quit". For each String, it constructs a corresponding StringCounter and then adds it to the SearchTree. The tree's **print** method is then called, and the user is asked if they want to run the program again. It keeps running until the user enters "no" when prompted.

Implementation Problems Encountered

I didn't encounter many problems writing this. I had a little bit of trouble working out the syntax for the LinkedList's **add** method, and I didn't initially know that this was what was causing problems. However, I just needed to check if the first element was null to handle the case of the head being set to the first element being added. Other than that, everything was fairly straightforward to write and implement.

Lab Report Questions

1. The Big O complexity of quick sort is n^2. However, the worst case is quite rare, and the average time complexity is n*lg(n).
2. The Big O complexity of merge sort is n*lg(n). This is also the average time complexity.