Landon Crowther
CS 2420 – Assignment 7
3/2/017

Analysis:

What I learned:

I learned that BSTs, although sometime tricky to construct, are much more effective than LinkedLists for organizing data. I can see how trees could become unbalanced or eventually defective, but overall, they do a great job at efficiently organizing and structuring data.

It was interesting to see how BSTs could become unbalanced just by a bad starting point. If the starting point was the "median" of the data set, the BST would generally turn out alright; however, if the starting point was towards the end or the middle of the data, the BST slowly turned into a LinkedList

Experiments:

We ran a few different experiments to test the effectiveness of our BST implementation and Javas. All the results of the figures are listed below. All tests were performed with random data, as well as sorted data. All experiments were averaged over 100 trials.

BST Implementation:

We tested the addAll and the containsAll methods for BST of lengths ranging from 1000 to 10,000. Because of the nature of a BST, it would make sense that a random list would implement much faster than a sorted list. This is because when adding a sorted list, we are essentially working with a LinkedList, which functions poorly in this case.

Because the BST implementation is not fully optimized, it was hard to test more than 10,000 elements without getting a Stack Overflow Error or having a ridiculous wait time. Java's TreeSet implementation was much more effective at this.

TreeSet Implementation:

We performed the same tests as mentioned above on Java's TreeSet, to compare efficiency. Those figures are listed below. It was noted that the TreeSet was much more effective, and could handle immensely more values (1,000,000) than our BST implementation (10,000). Additionally, the timing was consistently shorter for the BST implementation and the Java implementation.

BST vs. Tree Set:

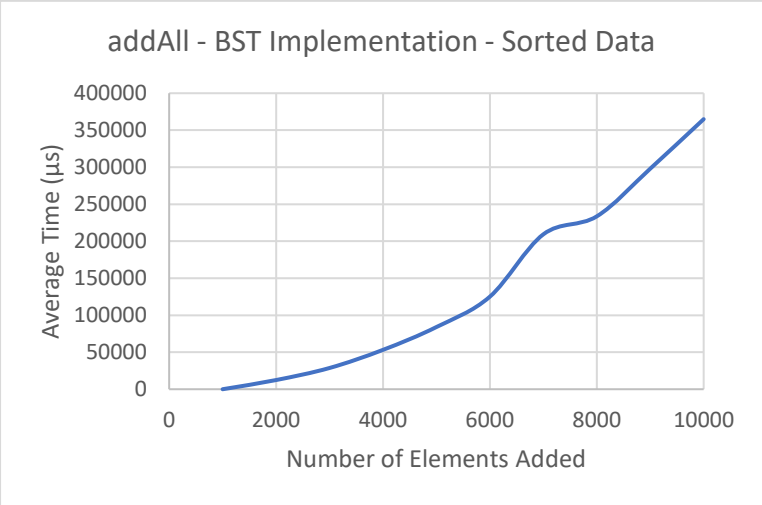
We tested the addAll method on the same sizes of structures using both the BST implementation and the Java implementation with random data. To my surprise, the BST implementation is just barely better than the Java implementation. However, when performing the same test with sorted sets, Java's implementation is MUCH better than ours. For example, to add 10,000 sorted elements with the BST, it took 365,000 microseconds; however, with Java's implementation, it only took 944 microseconds.

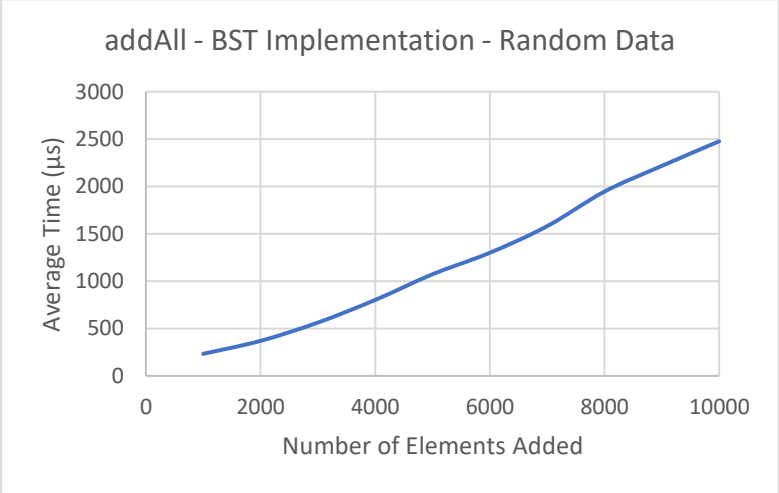
Big O Behavior:

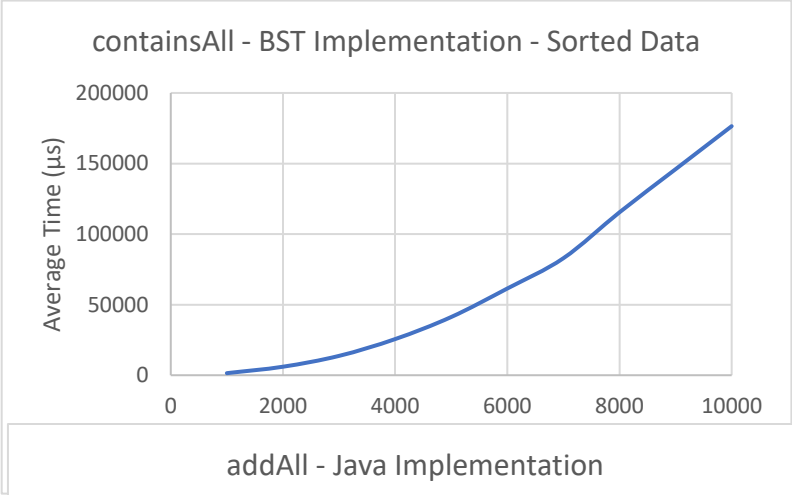
In general, the add and contains method is $\log_2(N)$ for balanced trees. This is because if the tree is balanced, these methods would only have to look at half of the data each time it goes down a level. For non-balanced trees, the complexity would be closer to N in the worst-case scenario (basically a linked

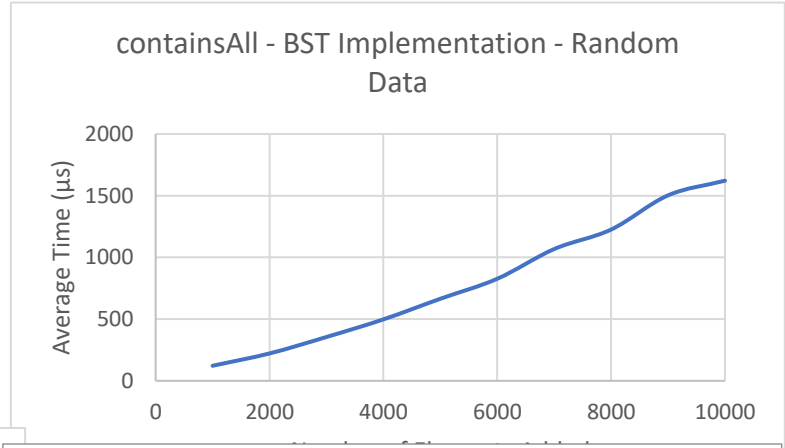
list). Because our implementation doesn't have any auto-balancing features, the complexity is between N and $\log_2(N)$, depending on how the BST was constructed.

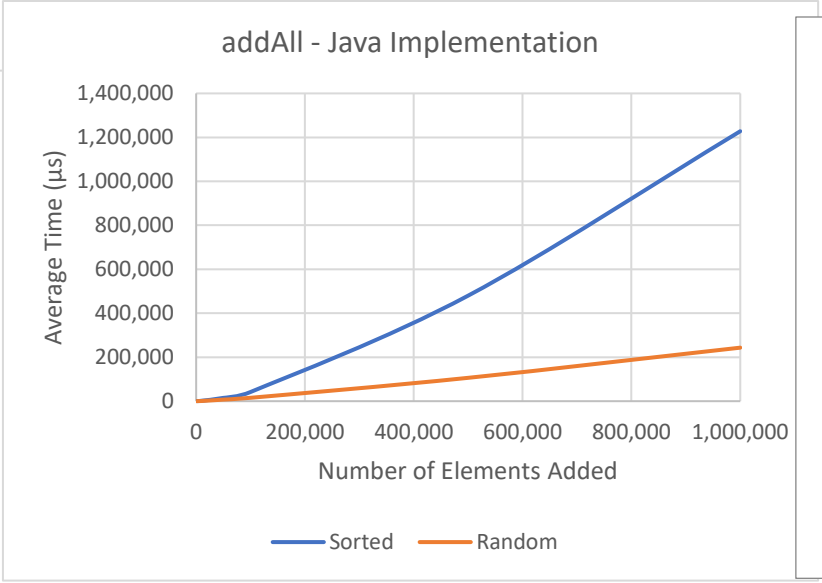
Figures:

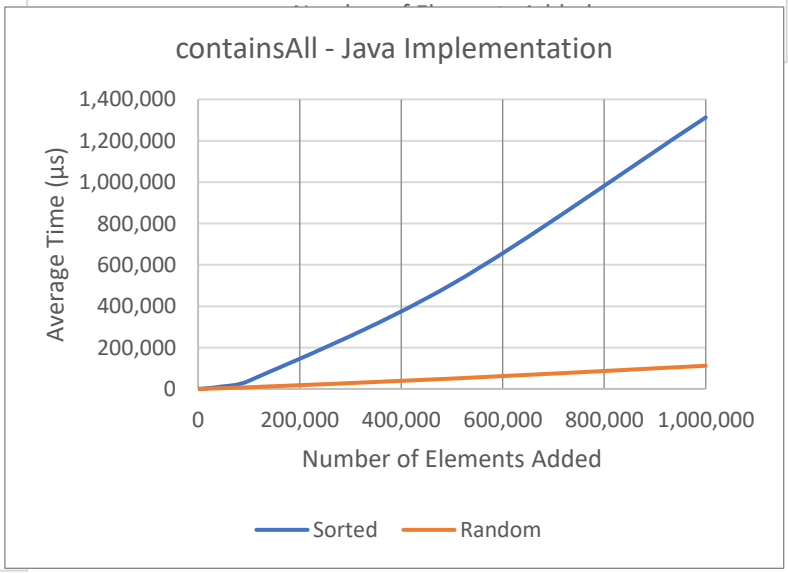




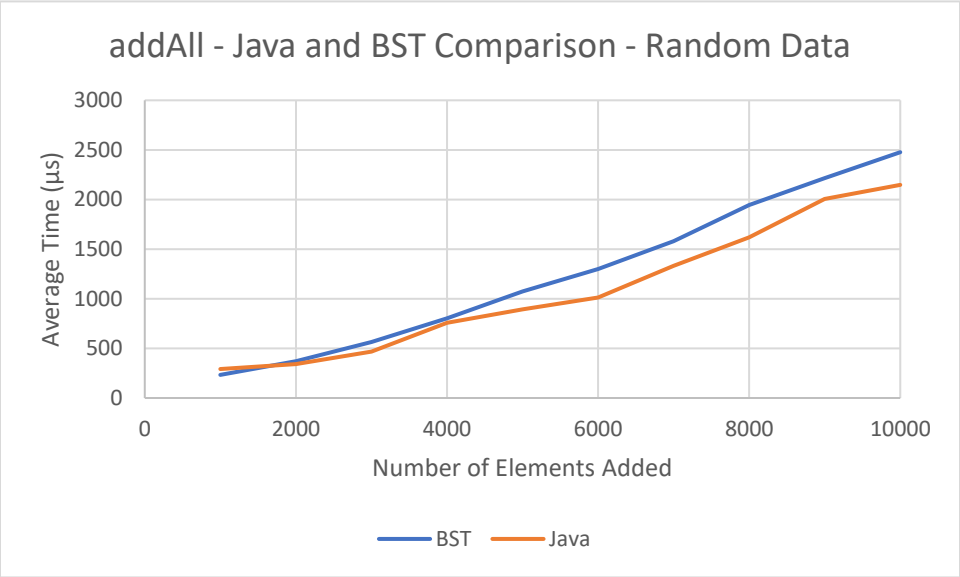








Landon Crowther
CS 2420 – Assignment 7
3/2/017



Class Hierarchy:

The class hierarchy is relatively simple for this project. The SortedSet contains the interface that the BinarySearchTree class uses. It contains all the main methods, such as add, remove, size,

The BinarySearchTree class is fairly stand alone. It's generic so that it can be used with any type of data. Within the BinarySearchTree class, there is a Node class. This Node class does most the "brute work." The Node class contains all they actual data, as well as the all references to left and right nodes. The Node class creates the "tree" structure. In addition, the Node class contains all the recursive methods that are used in the BST.

The SpellChecker class is just a BST with strings.

Landon Crowther
CS 2420 – Assignment 7
3/2/017

Iteration vs. Recursion:

Recursion was a super useful tool for this assignment. In some cases, the problems would have been impossible to solve without recursion. For example, traversing the BST to make an ArrayList of the BST in sorted order would have been incredibly complicated without recursion. This is because the method would have to make a virtual stack, which is what recursion does anyway. It becomes even when the fact that the depth of the BST is unknown, so it is difficult to account for how many "levels" the virtual stack should travel.

Recursion was useful in the contains method as well, because the value of the "item" only needed to be compared to one node, and then a simple if statement could make a recursive call to the appropriate node.

The only time iteration loops were used was during the first and last methods, using a "current = current.next" style in a while loop. These methods could have easily been made recursively. Additionally, the containsAll, addAll, and removeAll used a for loop, but that's because they had to check every element in the collection, and it makes more sense to use an iterative loop.

Development Log:

This assignment took roughly 15 hours, which was a lot less than the previous two assignments in my opinion. I think I had last assignment prepared me with a better understanding of the concepts involved in this assignment.

We kept over complicating things (like a lot). After talking with multiple TAs, we finally could write out pseudocode that worked for our implementation. Because we used special helper methods & switch statements (mentioned above), the logic for remove was slightly different than the logic we discussed in class.

The most frustrating part of this assignment was the fact that we would "fix" the remove method, run the tests, and sadly find out that we created even more problems. This happened way too many times to count. Each time, however, we used the debugger to find where the problem was, and could work through it for a successful implementation.

Thought Problems:

Non-Balanced Tree:

If a sorted dictionary were to be added to a BST, it would add in sequential order, and we would end up with a LinkedList. This is very inefficient. To combat this, one could randomize the elements in the dictionary, and then add them to a BST. This would not always prove to be effective, though. Perhaps the best way to create a balanced BST would be to find the "median" word in the dictionary, and then work outwards in both directions from there, alternating sides of the median.

Landon Crowther
CS 2420 – Assignment 7
3/2/017

Spell Checker

The spell checker tree from the SpellCheckerDemo class was balanced. This is because the words that were added were not in alphabetical order. In fact, they were added as a sentence, and the order of the words varied throughout the sentence.

Drawings: