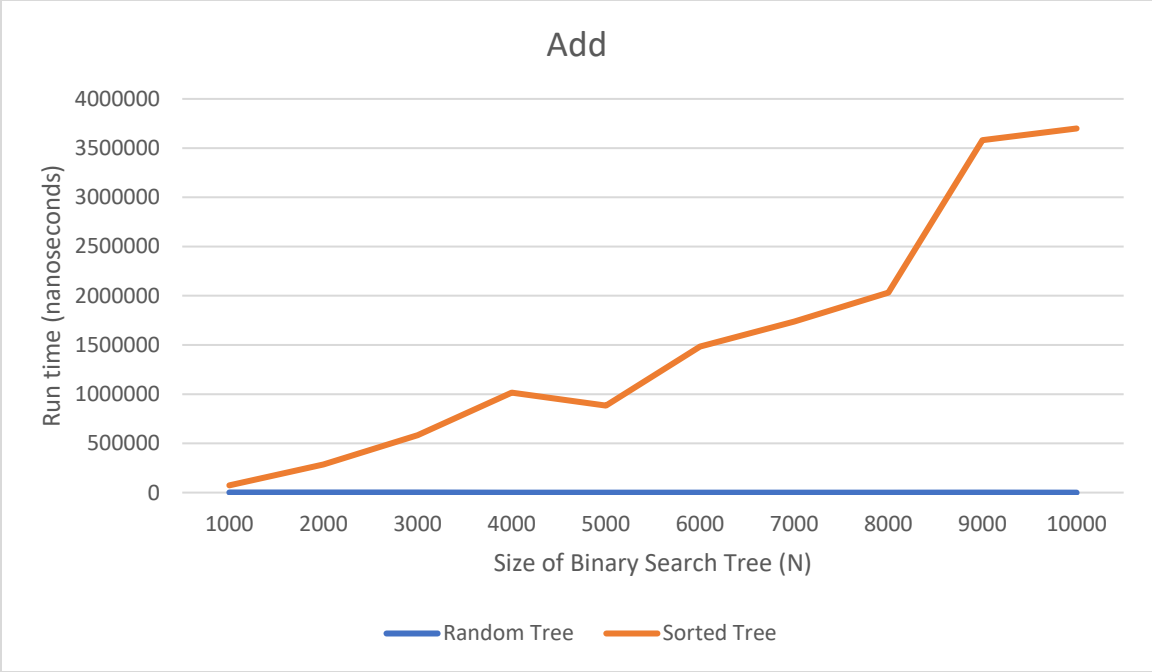
Assignment 7 Analysis

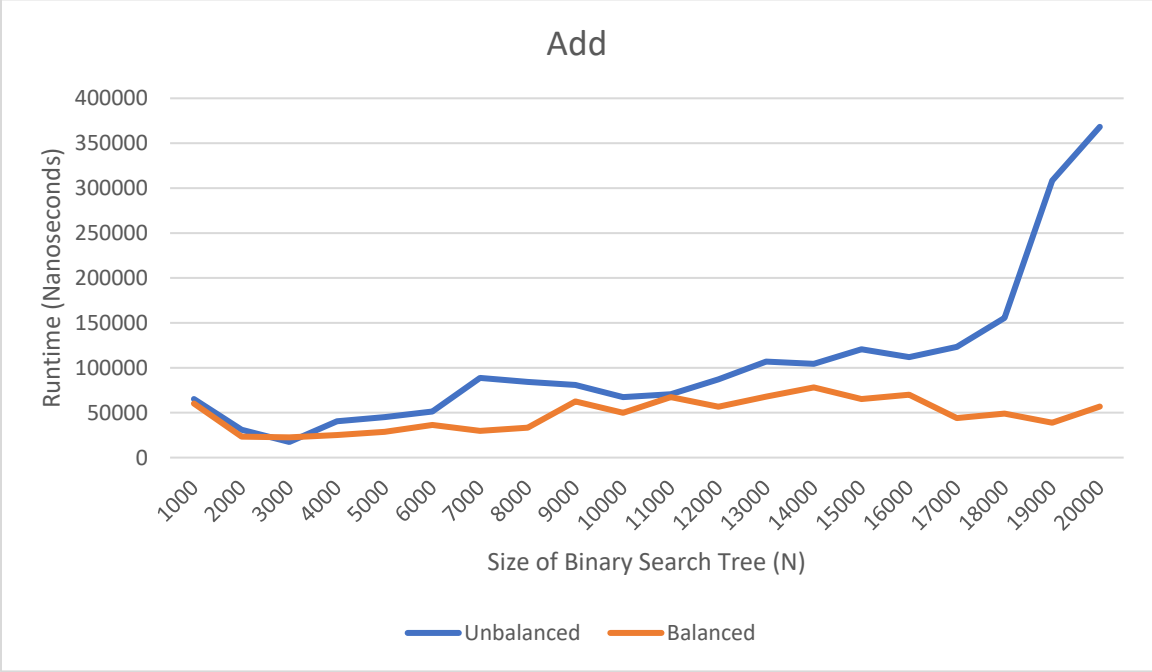
1. I completed this assignment with a partner. Her name is Nandini Goel and I submitted the program files to Gradescope.
2. The order in which items are inserted affects the balance of the tree. If they are inserted in a pseudo random order, then the tree is more likely to be balanced. However, if they are inserted in order from smallest to largest, then the tree will be right heavy with all items on the right. If they are inserted from largest to smallest, then the tree will be left heavy with all items on the right. This will affect the running time, because in the average and worst cases for add, contains, and remove for a balanced tree will be O(logN). However, in an unbalanced tree, the worst cases will be O(N), because if we need to add or remove a value that is the largest or smallest, we must traverse through every single value in the tree before we access the correct one. This is why balance in a tree is important, because binary search trees can be very fast, but if they are unbalanced it is significantly slower to perform the same operations.
3. To demonstrate the difference between a BST built in sorted order vs. unsorted order, I will create two different Binary Trees of size N, and perform timing tests for a few methods in our Binary Search Tree class that are not constant, i.e. where the size of N will impact the running time. To create the trees of size N, I will have a for-loop that has two statements contained in it. Statement 1 will add the current integer i to the sorted order tree, and statement 2 will add a random value from 0-N to the unsorted order tree. This guarantees the lists will be of size N and that they meet the sorted/unsorted requirements.

   After creating the trees, I will measure the average amount of time it takes to perform a function on the tree—for example, how long it takes on average to add a random integer to each tree. This timing method is performed by similarly to how the timing was done for the last assignment—start a timer, perform the action the given number of times, and the perform a simple calculation to ensure the time only includes the actual method itself, not any time it takes to add an item back in after the operation, remove an item, etc.



These two graphs above show that an unbalanced binary tree (the tree where each element was inserted in order) causes significantly slower run times compared to a relatively balanced binary tree. This is because there is no way to 'cut' the problem size in half, as is done in a binary search, so it takes longer to loop through all the elements because there is no way of reducing the problem size. The worst case for an unbalanced tree is N, meaning you must go through every element in the tree until you reach the last one, if that is your desired value. In a balanced tree, however, the worst case is O(logN) because we can make fewer comparisons to find our desired value. In each experiment above, I created two types of binary trees for each value of N; the only difference was

what values were added and when they were added. Thus, the problem size was not the factor in the difference in running times, it was solely the imbalance of the trees.

4. 

My experiment to compare a non-balanced tree against a balanced tree was to time how long it took to add a value to the tree for each size of N. I created two different binary trees, one of my written Binary Search Tree class and the other was a Java TreeSet class. I created a for loop that timed how long it took to add the values from 0 to N to the tree, and plotted them above. The balanced tree obviously added values much faster than the unbalanced tree, especially when the problem size got larger. They were relatively similar for the smaller tree sizes, but for the larger sets it becomes a lot more obvious that the unbalanced tree takes more time to traverse through, and thus takes longer to add a new value to the tree. If the tree is balanced, it will run in log(N) time at worst, while this is not a guarantee for the unbalanced tree.

5. I think a Binary Search Tree is a good data structure for a dictionary, because we don't have to add or remove values often and instead just use it to primarily search for a value. The algorithm for searching for a value is relatively simple, and as long as the tree is relatively balanced, it has a fast runtime. While an array, ArrayList, etc. using a binary search would be about as fast (a run time of O(logN)), it would be more difficult to add values or remove them. It is not a primary concern because it doesn't happen often, but as a rule, it is better to use the more efficient data structure, even if you don't anticipate using certain functions often. However, I do think that the binary search tree would be an inefficient data structure for a dictionary if we added the values already sorted. As seen above, an unbalanced, already sorted tree runs significantly slower than a balanced tree. An array or ArrayList would be far more useful at that point, because we would just need to loop over each value and add it to the end of the list. At that point, the run time is O(logN), and while adding and removing aren't as efficient as a binary search tree, as stated above, these aren't used as often and thus it is worth the extra effort to use a more

efficient structure for the methods that are used most often. In general, I believe that a binary search tree is efficient as a dictionary if we add words to the dictionary unsorted, but it should not be used if we add the words in their already sorted order.

6. As stated above, if we add values to the binary search tree in order it will create an unbalanced tree that will be incredibly inefficient and run slowly. It would defeat the purpose of using a binary search tree, and would instead be better suited with an array or an ArrayList. To solve the problem of the binary search tree being unsorted, we can either create a method that balances the binary search tree as we go, we can use an implementation that has a self-balancing method built into it (such as Java's TreeSet), or if neither are possible, we can instead continually select random values from the list of words and add them into the tree until the list no longer contains any values. Though that isn't the most efficient method, it would ensure a pseudo-random order and would help solve the imbalance of the tree without having to create an entirely new method to balance the tree later on.