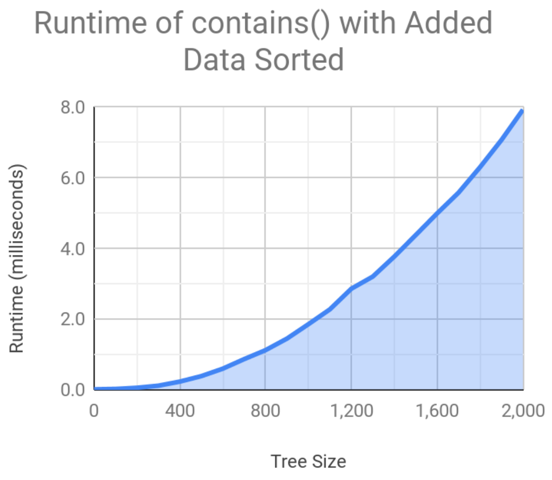
Assignment 8: Binary Search Tree

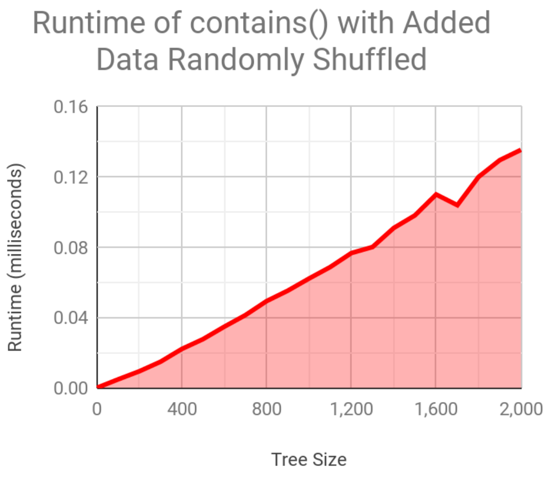
Analysis Document by Qianlang Chen (u1172983)

As in the previous programming assignment, I had Kevin Song as my programming partner in this one. Kevin did a great job at understanding the presented problems and quickly coming up with proper solutions, which made us work efficiently. Because of this, I plan to work with Kevin in future assignments.

During the assignment, we have noticed and understood that the performance of our *BinarySearchTree* (abbreviated as BST) was brought by its mechanism but relied more on the order of which the data is constructed and added to it. Since in this assignment, we did not have to keep the tree balanced at all time, there is a high probability where the elements are laid out heavily on one side of the tree and the sub-trees after a certain amount of adding or removing actions. When the tree is not balanced, the methods of our BST, such as *add()*, *contains()*, and *remove()*, will spend more time searching for the element on the heavy side of the tree, therefore making the algorithms slower. In fact, the more the tree is unbalanced (i.e., the more elements laid out heavily on one side), the closer the methods' performance would be to a linked-list (more linear than logarithmic).

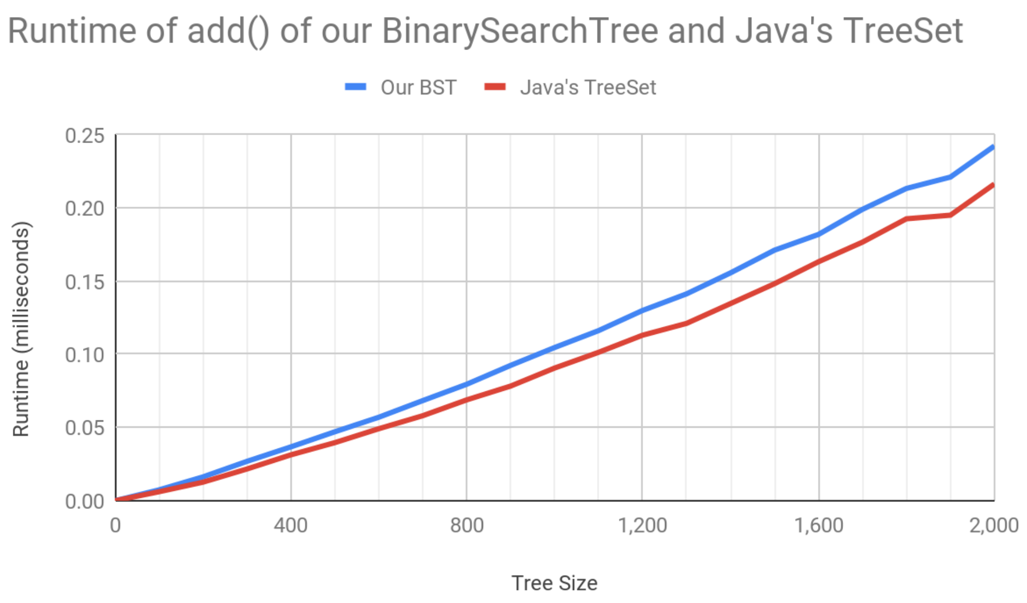
To experiment how much the runtime of our methods is affected by the order of which the data is added into our BST, we decided to time running the *contains()* method on trees whose data was added in sorted order versus in random order and made a side-by-side comparison. Here are our results:

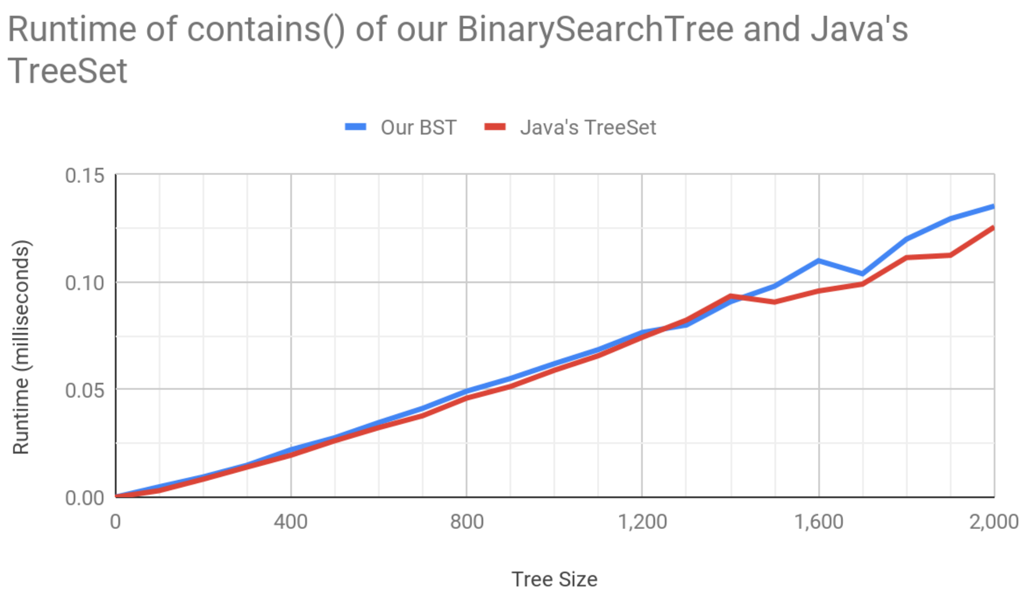




As we can see from the graphs, our BST did a better job when the data was added in random order, which matches our prediction. Since we ran the *contains()* method for every single element added to the tree, the runtime behavior should be the behavior of the *contains()* method itself multiplied by the number of elements. Dividing by the runtime behaviors of the above two tested cases by the number of elements in the BST, which are N, we get **linear** and **logarithmic** behaviors, respectively. This result makes sense, because according to our theories in mind, when all nodes in a tree are laid out on one side, the tree is no different than a linked-list, and that gave us the linear runtime behavior.

How about using a BST that always tries and guarantees the child nodes are balanced on both sides and for all of its sub-trees? To see and understand how much different it would make to runtime, we called out Java's native BST, the TreeSet class, which always keeps the nodes balanced. We timed the *add()* methods for each BST by adding elements in a random permuted order, as well as the *contains()* methods for both trees with data added in random order. Here is what we saw:





Our results show that Java's *TreeSet* had a slightly faster performance over our BST because of its auto-balance functionality. According to this, Java's *TreeSet* should always give us the same fast performance regardless of the order of which the data is added, and this is a significant advantage that our BST does not provide.

Since BST is a data structure which gives a good performance, we discussed where we might use it. Dictionary might be an excellent place to use our BST because even though it might be expansive for our BST to add and remove data compared to using arrays or linked-lists, dictionaries themselves do not deserve many modifications once they are constructed. On the other hand, dictionaries more often need to be looked up, for which our BST has high performance. In conclusion, if a dictionary is well-constructed, which means that the BST used to hold the dictionary is perfectly balanced, it can be a good idea to use BST as its data structure. Simply constructing the dictionary with words in alphabetical order would make it miserable because the BST would, therefore, be all-on-one-side and would perform exactly like a linked-list. Instead, although making the BST perfectly balanced is a hard job and requires some advanced algorithms, randomly shuffling the dictionary before constructing the BST would do most of the trick since it would make the BST nearly balanced, giving us almost best performances.