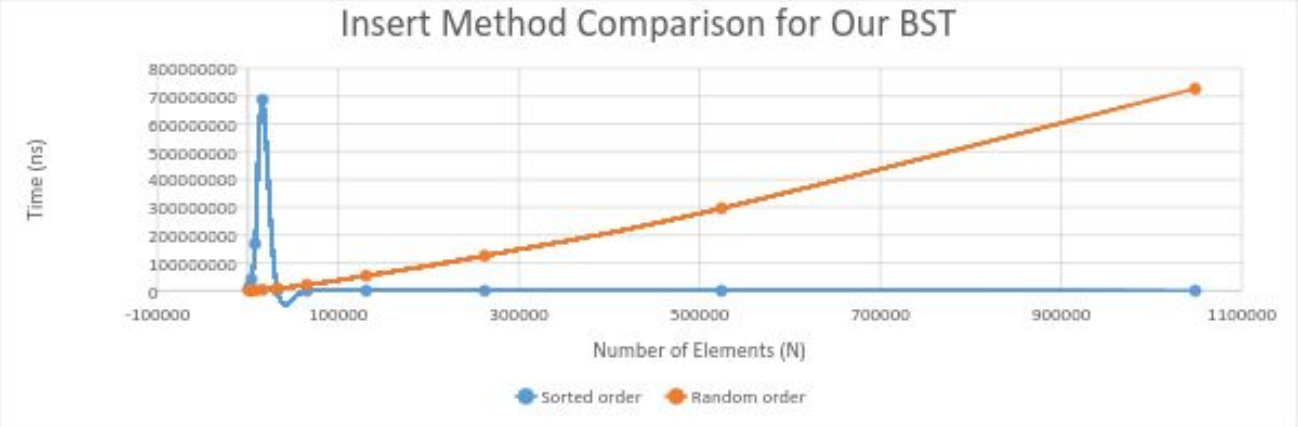
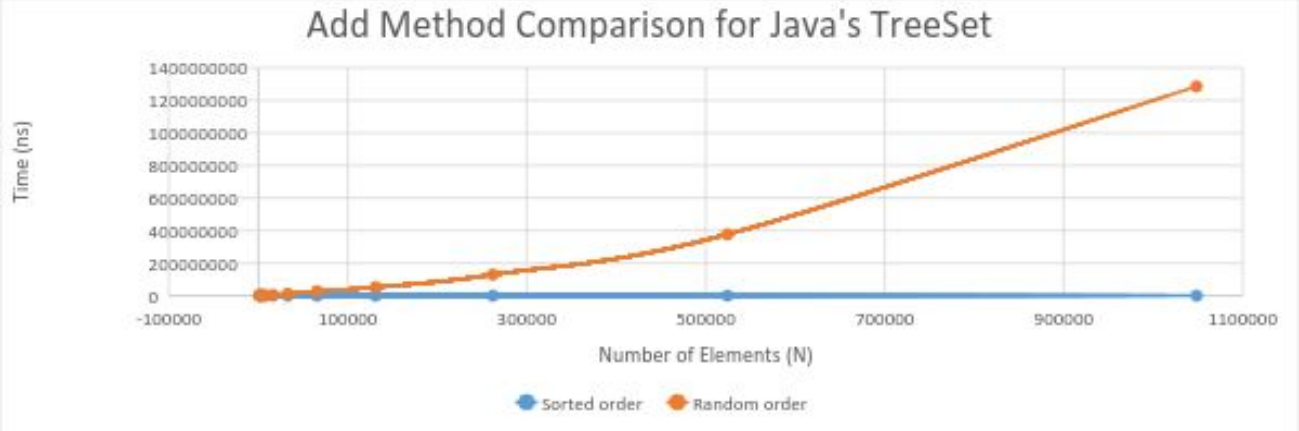
Tony Diep
3/02/17
CS 2420
Analysis Document

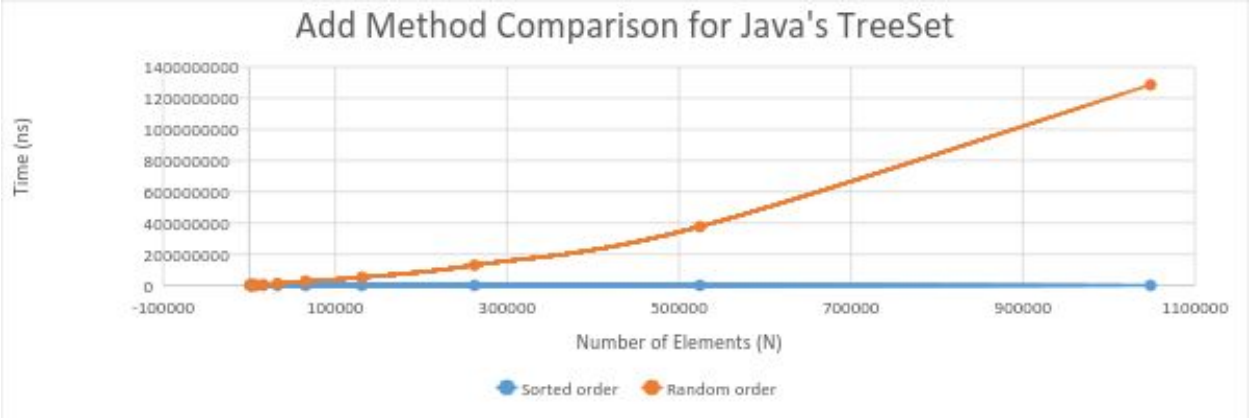
OVERVIEW OF PROJECT

Binary search trees (BST's) are efficient data structures because of fast insertion and searching time. They are in essence similar to a Linked List because they both contain nodes in which they contain data as well as another reference to neighboring nodes. Conceptually, we can think of BST's as similar to a Set (in particular, a sorted set) because of the similar properties of a Set. The BST (at least for our purposes of this project), disables duplicates since every element added to the tree will either be lesser or greater than the root node's element.

EXPERIMENTS AND WHAT THE GRAPHS TELL US







The two graphs above show a commonality between add, remove, and contains. Ideally, when we examine a balanced BST, the complexity should be O(log N), but for an unbalanced BST, it should be O(N). However, our graphs do not display the expected result we wanted. Conceptually, it should be expected that the BST containing random elements should take lesser time as opposed to the sorted. When adding elements in sequential order, it causes an unbalanced BST because the number of elements added to the BST will make the BST one sided. The line for random should have been logarithmic because the tree is balanced which takes O(log N) as opposed to adding sorted elements. So for sorted elements added to the BST, the complexity must be linear because the tree would be unbalanced.

Perhaps our graphs turned out parabolic for the random elements line because we are implemented a bit more iterative implementations than necessary. We may be implementing a couple of code that takes constant time. We are also surprised to see that even if we implemented Java's TreeSet, it still gave us incorrect results on the graph but we could maybe hypothesize that our timing code may be off.

THE CLASS HIERARCHY

Each class in this project has contribution as a family hierarchy. Firstly, the Node static helper class is the basic class that gets the Binary Search Tree class working. Although it is the smallest class in terms of hierarchy, it is an essential class to implement because it is essentially performing the essential work. A correctly functional Node class allows for the whole program to run correctly and efficiently. The BinarySearchTree is a class that represents a collection of Nodes which clearly depends on the Node class. The BinarySearchTree class can be used to represent a dictionary which is created in the SpellChecker class. The SpellChecker class depends on the dictionary, which is essentially the BinarySearchTree. The main program of the spellchecker, called SpellCheckerDemo, depends on the SpellChecker class to work, which is top of the class hierarchy. The Timing class is a supplementary class that tests for efficiency of the BinarySearchTrees.

The Node class "supports" the BST class by doing most of the work. For example, the Node classes determine where the data should be placed in the BST. If the current node's data is lesser than the root node's data, then it must be placed on the left hand side. Otherwise, the

current node's data should be placed on the right hand side of the BST. We are going to neglect duplicate data and therefore, neglecting any data that is equal to the root node's data because that is an arbitrary choice. The BinarySearchTree class supports the SortedSet interface because it implements the interface and uses the required methods. The BinarySearchTree conceptually is a sorted set because the tree contains smaller elements on the left and bigger elements on the right.

ITERATION VS RECURSION

BinarySearchTrees are naturally recursive because we can conceptually divide it into smaller pieces of itself. A BST basically consists of smaller BST's in itself. In other words, it contains subtrees in which they have a left subtree and a right subtree. We see that as we traverse down below the tree, the height becomes O(log N) (with the exception of a right or left heavy BST which has a height of O(N)). In order to traverse through all of the nodes in the tree, recursion is best used.

We have used a lot of recursion in this project. In the Node class, the three methods required to implement are all recursive since each node potentially has a left or right child. We can avoid the worst complexity of O(N^2) since without recursion, we'll certainly be using iterations. We used some recursion but also iteration in the BST class. When we use recursion in BST, we create driver methods to get the main recursive methods going. For methods such as addAll, removeAll, and containsAll, they require an iterative implementation because they all take a Collection as a parameter. The easiest and efficient implementation for going through the Collection is to use an iterative process (i.e. for each loop).

A good example of a method that can easily transition from iterative to recursive and vice versa is finding the minimum node. In terms of iterative, we need to start at the root node and start traversing the left side until we reach a null node, which implies that there are no more nodes to traverse on the left side. When we think recursively, we can easily construct a base case that if the current node still exists, then we want to traverse at the current node's left child to find the minimum node. The concept in terms of implementation is not vastly different.

SOFTWARE DEVELOPMENT LOG

We spent over 15 hours working on this assignment. We struggled initially to implement all of the methods in Node in a timely manner. We started on Sunday in that we completed test cases and the drawings but the implementation took a little long than desired. In order for us to do better, perhaps we should consider understanding conceptually how the data structures work. We failed to understand the BST conceptually in some parts and hindered us from finishing the code in a timely manner.

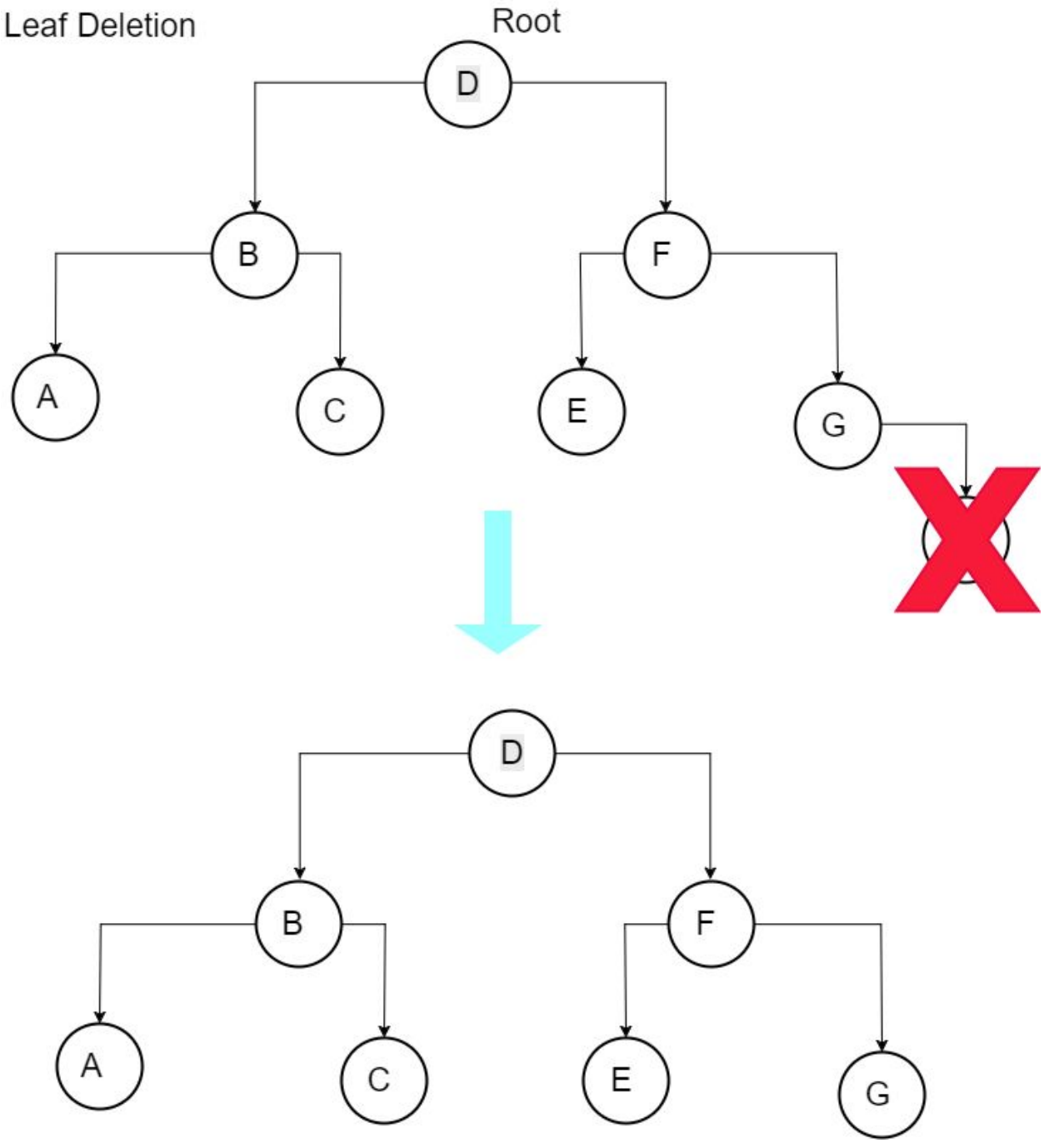
Although we struggled to finish the implementations on time, we had a good start to the assignment by first consolidating the test cases and drawings first. If we had not done the JUnits and drawings first, we would a lot more behind on this assignment. So we are behind but not as behind having not to do the JUnits and drawings initially. Nonetheless, we are able to have a good idea of what results we should expect when we implement the methods in Node and BST.

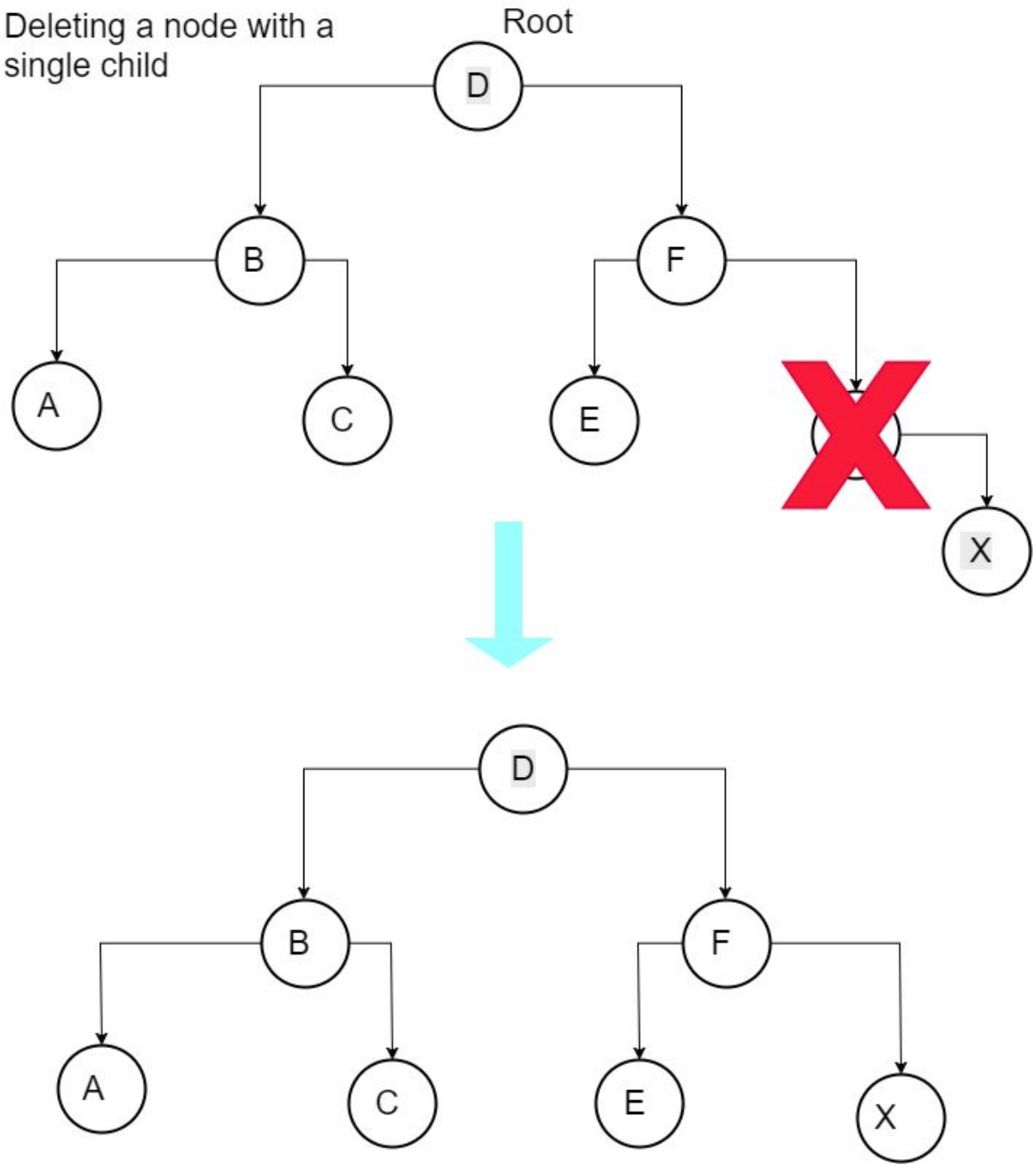
THOUGHT PROBLEMS

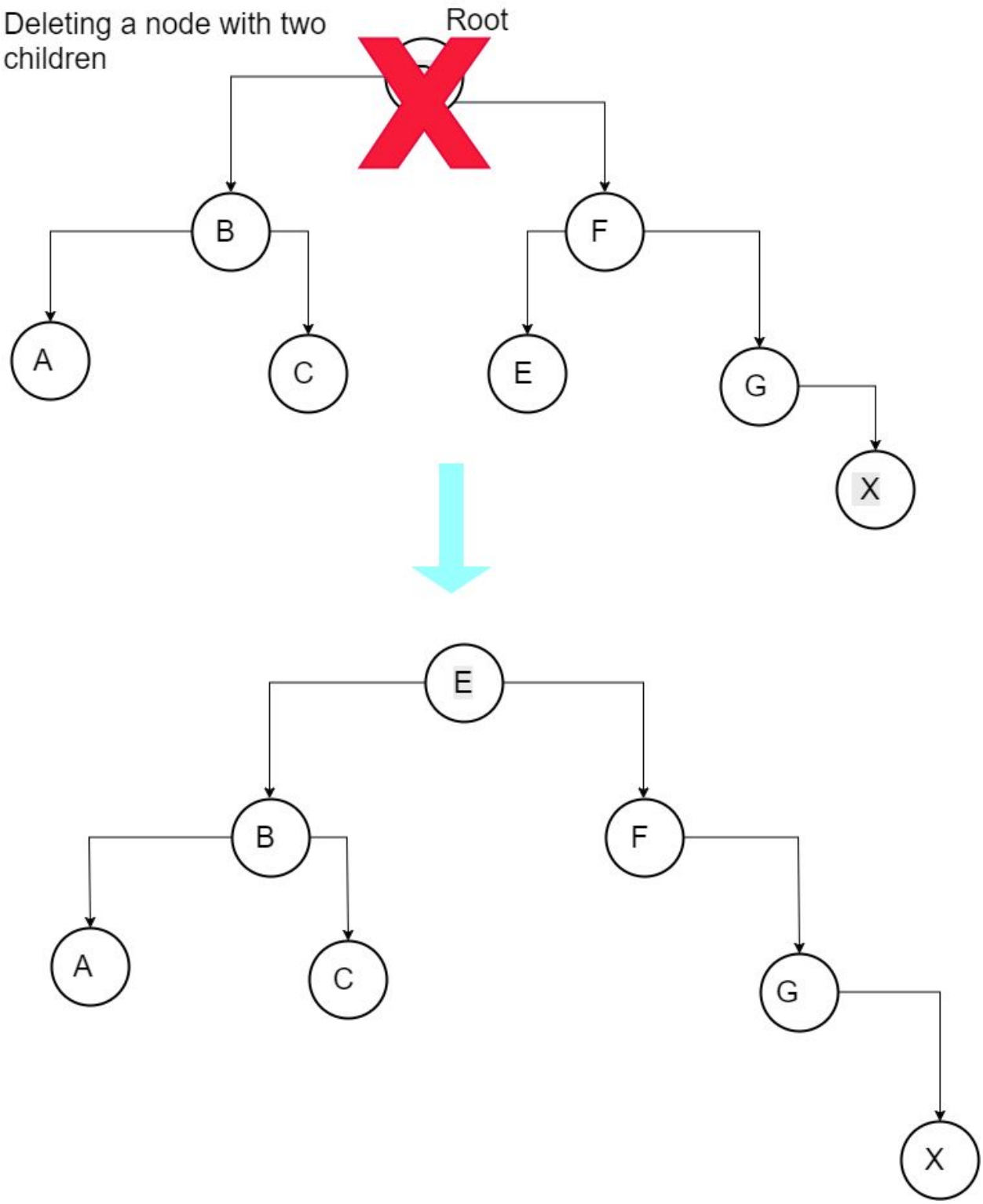
If the words are added to the dictionary BST in alphabetical order, we could get a left heavy or right heavy BST which does yield the complexity of O(N), which resembles similarly a Linked List data structure. Thus, we get an unbalanced tree. Perhaps we can try and add words randomly because in essence, the nodes containing each word will determine where in the tree should they be placed. We can likely get a balanced BST.

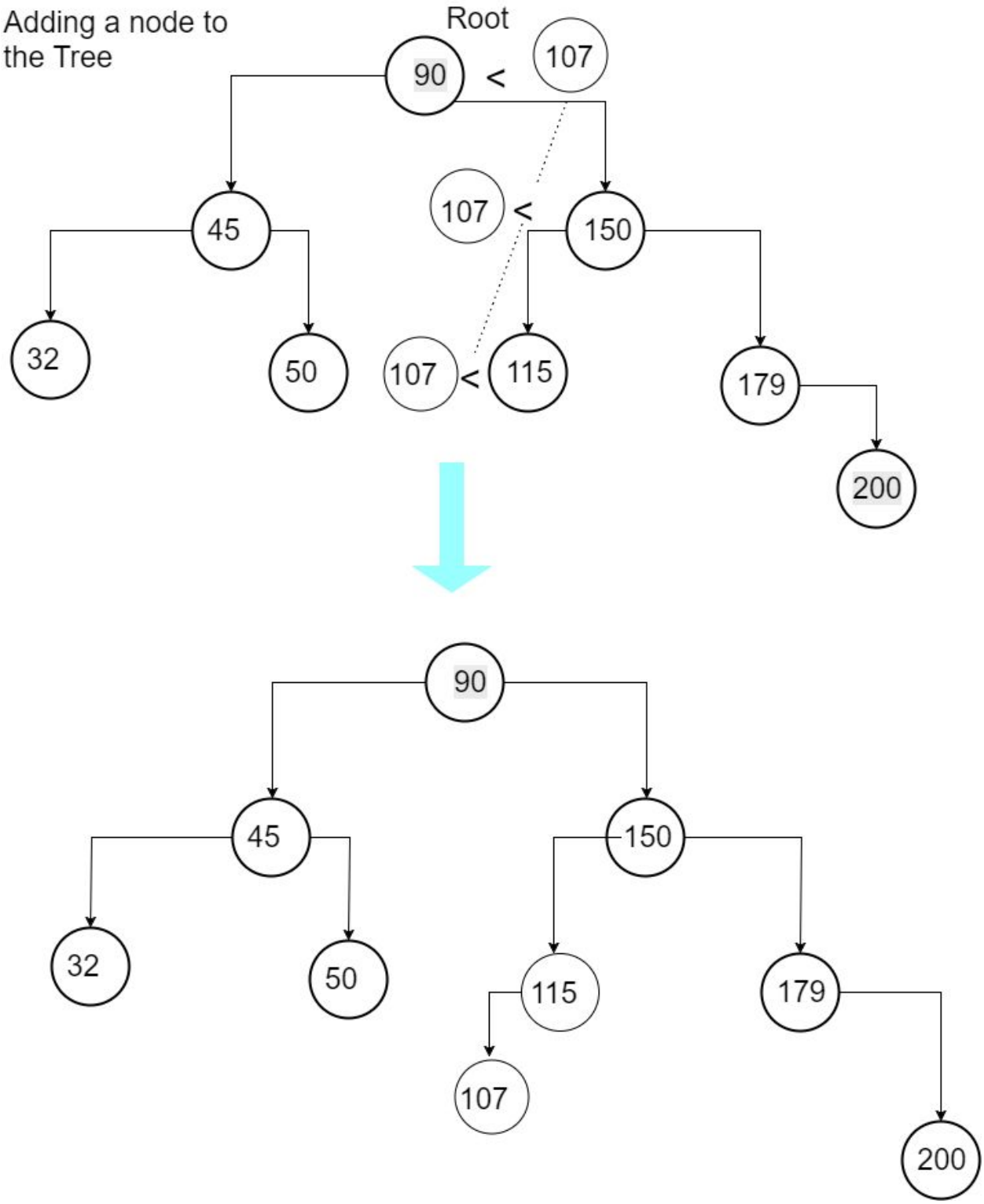
When running the spell checker application, we see that our tree is unbalanced. When adding the words in the text file “dictionary.txt”, we notice the file contains the words in alphabetical order. The alphabetization is problematic because as mentioned earlier, it is vulnerable to having a one-sided heavy BST. By adding words randomly, we can likely get an even distribution of some nodes placed on both sides of the tree. Because of this, we are likely to have the worst complexity of O(N^2). Searching for a particular word in the dictionary BST would certainly be a lot slower.

DRAWINGS


Leaf Deletion
Root
D
B
F
A
C
E
G
D
B
F
A
C
E
G


Deleting a node with a single child
Root
D
B
F
A
C
E
X
D
B
F
A
C
E
X


Deleting a node with two children
Root
B
F
A
C
E
G
X
E
B
F
A
C
G
X


Adding a node to the Tree
Root
90
107
<
45
107
<
150
32
50
107
<
115
179
200
90
45
150
32
50
115
179
107
200