John Mobarry

CMSC 204

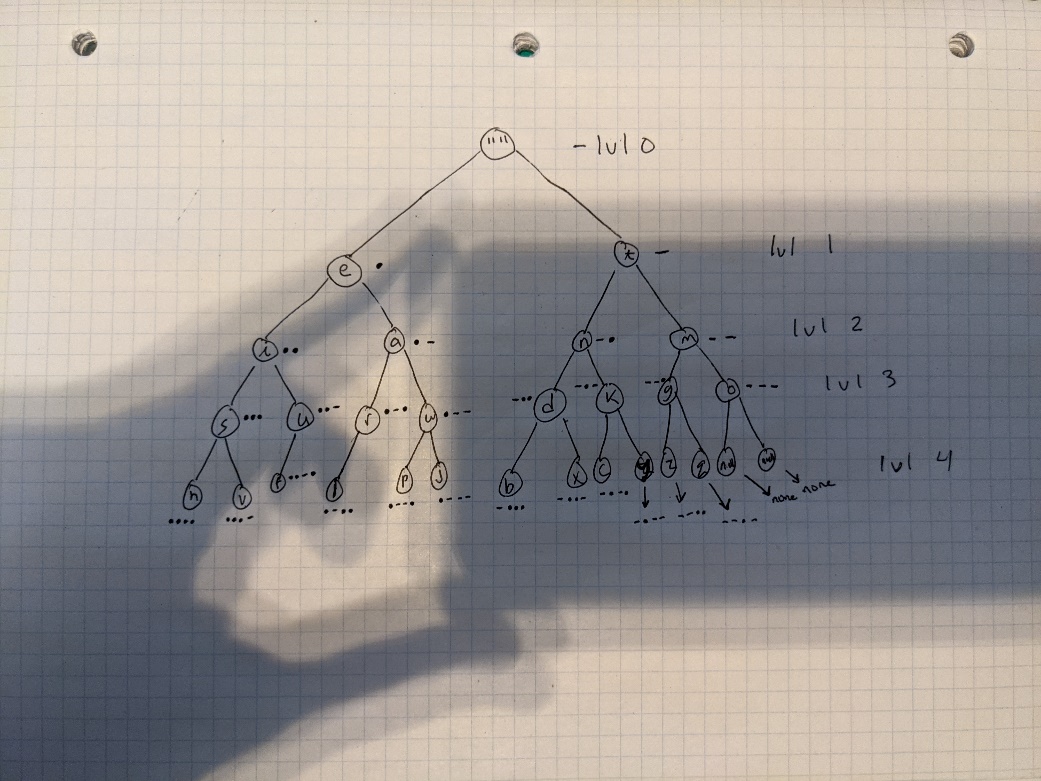
**Assignment #5 Reflection**

This assignment presented the student with an easy to conceptualize task that can be solved with one of the more difficult data structures we have learned this semester. Up until this point we had approached the Binary Search Tree in a conceptual sense where we were creating the trees on paper and understanding the theory behind search algorithms. This assignment tasked us with applying this knowledge in order to convert Strings of MorseCode into Strings of English. This goal was an excellent choice for a practical application because translating between MorseCode to English is a concept that can easily be understood but can be automated much easier by a computer program. Due to the straightforward nature of the problem, the student can then focus on how to apply algorithms to the problem instead of struggling over the steps required in order to implement by hand.

When I first sat down to complete this assignment, I attempted to conceptualize how the program structure at large would function upon completion. After 15-20 minutes I decided that the best thing to do in order to efficiently use my time was to approach the assignment document and locate the class that was the foundation for the rest of the program. The foundation class turned out to be the generic TreeNode<T> class as it is utilized to create the MorseCodeTree class that is then used in the MorseCodeConverter utility class. Without having a sound implementation of the TreeNode class then there will be no ability to progress forward in the implementation of the higher classes. The TreeNode class is like the other Node classes that we have implemented in previous assignments but modified to fit within the greater data structure of the binary search tree. This generic class has the standard characteristics of having data, a left child, and a right child in order to create cascading bifurcated branches to create a larger data structure where every node can have two child nodes. In order to get the class to function and provide everything that the MorseCodeTree will require I added getter and setter methods for data, leftChild, and RightChild accompanied with Boolean hasLeft and hasRight methods for debugging purposes. With the TreeNode class completed with the methods required to successfully debug downstream we proceed to the MorseCodeTree ADT class.

The MorseCodeTree ADT is an example of a class that was easy to visualize in terms of completed structure but more difficult to determine the algorithms required to generate the class.

The key to developing the class was understanding that there would be numerous implementations of recursive algorithms required in order to successfully create the MorseCodeTree class. My first step was to draw a tree that combined the two diagrams that were provided in the assignment in order to visualize the static tree in all its glory. Attached is my diagram that was created in order to help me visualize the tree:



With the finalized structure visualized it became easier to implement the recursive method addNode(), to implement the insert method(), the buildTree() method and the constructor that implements the buildTree() method.

The MorseCodeConverter utility class was a class that took a bit of debugging in order to get the class to function appropriately. I first tackled the convertToEnglish(String code) method due to the fact that if I could get this one to function it would just take a bit of modification to get the convertToEnglish(File codeFile) method to function properly. The key to getting this method to run was creating and Array of Strings called codes that would store the entered String code split by the delimiter of a space. Then I implemented a for loop to fetch the value of each element and concatenate them together into a large String to be returned. This passed the Junit testing so then I copy and pasted this method into the codeFile method and then placed the same within a while loop which sets the arrayList of strings equal to the data in the file delimited by space as long as there is a next line. Then it applies the same algorithm to the array of strings stored and returns the concatenated result String. To implement the printTree() method I created a new MorseCode tree and then used an enhanced for loop to iterate through the tree concatenating the elements. I had an issue with this method as it was always printing a null at the end of the information, so I had to use substring to print out to the length-1 of the String. I am not sure if this is an optimal implementation of this method but it was able to fulfill my needs.

This assignment surely resulted in a bit of white knuckling while I was debugging and resulted in a few instances of me questioning if I even understood what was being asked of me. But in the end I was able to chug through it using the debugger in order to get a functional program.