Program 3 – Morse Code Tree

By: Jeffrey Morgan

Assumptions:

Initial assumption was thinking this program was going to be easy. It was in the end, but I tried tackling the problem with the wrong type of tree. I initially started coding it as a Huffman Tree. But what I ran into was trouble loading the Internal nodes with data instead of the leafs. This ultimately caused my tree to be a much longer depth than intended, and it also caused my tree traversal to give incorrect values per the Morse table. After crashing on that Idea, I came into the realization I can just built this tree normally as a BST. I kept the weights from the Huffman tree I was initially building, but built the tree as a BST. Doing so I merely had to organize the inserts to make sure they were inserted in the correct order, and maintained the weights on the nodes as a means of comparison so they were placed in the correct positions.

Big-O:

Insertion is based on LogN. It splits down the tree with the same amount of comparisons. While inserting it was also inserting a K,V pair into the map. When it would go left for a comparison it would add a “\*” if it was greater than it would go right adding a “-“. So when building the tree instead of manually assigning the morse code value to ensure it was built correctly. It adds the code to each Instance of a Morse Letter.

The encode function took in only a string for the message to be encoded. Since during the insertion each KV pair is stored within the Tree in a map I only needed to make a foreach function that took each letter of the message and then spit out the maps->second whenever it found the character it was associated with. Since maps are based upon the red-black tree the find function helped eliminate overhead as it is a LogN based functionality.

The last major function is the Decode function. It takes a morse code string and for example “\*\*\*\*” which is the letter H. It will traverse down the left sub tree for every \* It sees and if it sees a “-“ it will go right. When it encounters a space in the string message as the next-character it will add that node it is sitting on to the string result. If it encounters a space and the next character is a space it will add a space to the actual result message incase there is more than once morseCode word that is needed to be decoded. This function also operates at LogN.

In summarization:

All these functions operate at LogN Space/Time, requiring very little overhead. By traversing the tree for finding nodes based on characters and codes you make it where you are not linearly(N) finding characters which would be the worst-case scenario for finding/inserting. Which should not happen in these functions.

References:

I used both the HuffmanTree/BinarySearchTree for inspiration on how to make the nodes for the tree, and the latter for the actual building of the tree. Instead of using a Huff\_Data as a Node I placed a Morse class inside of the BTNode and used that to built the tree. A little bit of a work-around, but this only adds a small amount of overhead rather than using the Morse class itself as a Node.