Koby Miller

Project 3 HuffmanProject

To start off, I will say that my code is wrote a little weird. I made a lot of different classes to keep things organized and make it less confusing for me. For example, I have two node classes. One was used for the list of symbols and the other was used for the HuffTrees. Also, it is still weird for me to not be coding the main(String[] args). To me it is like building the tools, but not being the one that uses them. So, I know how I would use them, but that’s not necessarily how everyone would use them. For example, I can’t use the buildTree() multiple times in a certain way (which will probably hurt me grade wise) but if I were to use this program, I would never need to use it twice. Anyways, here are the main methods we needed to implement and how I did.

getFrequences() – O(n) 2 for loops, but one is constant

First, I read the .txt file given char by char. I then create a node for each new character that is read. Each time it is read after it’s node has been made, a freq int is incremented. So once the file is read all the way through, there will be a list made with all the symbols used and how many each symbol was used. I originally had a method that then re-arrange this list to make the characters go in ascii value order. Then a string would be made while I read through the list and the string keeps track of the character and its frequency and that is returned. The method was causing problems and I don’t know why I couldn’t get it right. I mainly did it for fun. I ended up just searching through the list for the ascii value ranging from 32 to 126 and adding to a string every time a value was found.

buildTree() – O(n) 1 while loop  
This was based off of the method given too us online. I needed to make a second node class because I was confused with the code given. I understood everything it said and what it was doing, but I was confused with how we called things inside the class. How we would get certain variables or nodes. So, my node class takes account of the two different types of nodes: Internal and Leaf, and different variables like if has been visited or what it’s weight is. But the tree is built like you would think. It takes the two smallest values off of the heap, merges them into a tree, then puts that tree back into the heap. Once there is only one item in the heap left, it is the full Huffman tree and it is returned.

encodeFile() – O(nc) where c is the amount of characters in the file and n is the amount of nodes in a list. For each character read, you have a loop to find a node. (so maybe O(n^2) but really it is even more because the characters can become very large)

I began by reading the file character by character. Each time a character was read, it would check the list made from the Huffman tree to see if that character was in the list. If it was, it would add its code to a string. After the whole file was read through, it would return the string which was compiled of 0/1 codes.

decodeFilel() – O(n) 1 for loop

I would start with an empty string like most other methods. I would read the string of 0/1’s one at a time traverse through the list starting from the root. If a leaf was found, the character of that leaf would be added to the string and the traversal would then restart at the root. It would throw an error if there was an input other than 0/1 or if the last input didn’t land on a leaf.

traverseHuffmanTree() – O(n^2) 2 while loops (but really n can only be up to around 100 here)

So, I actually made another method called traverseTree because the way my encodeFile method worked, I needed to be able to traverse the tree. I didn’t know how the grade would be affected if I didn’t just make another method, so to be safe I just made it.

It starts at the root and checks the left node to see if it has been visited. If not, go there, if it has, check the right node to see if it has been visited. If not, go there. It does this until it reaches a leaf. It then says that leaf is visited. Now it restarts from the root and goes through again. If both the left and the right root have been visited, then you are at an internal node that can’t move so it is now set to visited. It runs through until the root has been considered visited. Each time it goes left a “0” is added to a string and if it goes right a “1” is added. Once it reaches a leaf, this string is now set to be the character’s code at that leaf.

I had other methods in the code too that help me throughout the writing of this program. Some methods were to print lists, print codes, add to a heap, add to a list. Another note (that again will probably hurt my grade), I could not figure out why, but when I go to make a second HuffmanEncoder object, it is like I didn’t make a new object at all and I am using the first one. This makes the methods return incorrectly or even make them not work. I had no clue how to fix this. Because only some methods can gather specific information that other methods need, you kind of have to use the methods in order. For example, you can’t make a tree before you make a heap, and you can’t make a heap before you know the frequencies. So the program has a sort of flow that you have to follow.

The data structures I used were: List, Heap, Tree.

I used a list instinctively because I feel comfortable with them, plus I was able to use some methods from past projects and just change them a little to help. But they were needed to keep track of the frequencies of the characters. I had never implemented a Huffman tree or a heap before so those were both learning experiences. I learned a lot about how they work and good tricks to implement them easily. I used a heap because I thought it would be cool to learn something new plus the website given for help used a heap to make the tree.