Lan Anh Do

ld9hu

29 November 2018

Post-Lab Report

In both encoding and decoding, I didn't make a Hoffman Tree class. Instead, I simply had a node class and the main method class that handled everything. By the time the nodes were removed off the heap to build the hoffman tree, there would only be one node that is the root of the entire tree in my main method. I realize in hindsight that a tree class would have simplified my code for insert, traversal, and generating prefixes.

For encoding, I read the file. I use the character as a key in a map and increased the integer value of it to count the frequency. The map will serve as an easy way to pair char with frequency.This step has worst-case running time of O(n) where n is the number of encoded characters. A map space complexity of `((sizeof(key)+sizeof(value))* map.size())+sizeof(map)` so my map has worst space complexity of at least 5n+48 bytes. I use the map to save the char and its paired frequency into a node insert it into the heap. Inserting into the heap is worst-case O(log n) because a heap is always balanced so we never have to go through every node like in a linked-list situation. The heap will have worst-space complexity of n*(sizeof(node)). My node has a char, int, 2 pointers, and string so its size is 25 bytes. When building the hoffman tree, I deletemin the first 2 nodes in the heap and make them the child of a temporary node. This will have worst-case runtime of O(n) because all nodes will have to be removed. When the temp node has been created, I recursively traverse through the tree until I hit a leaf-node. Once I do, I save the prefix code in the node. This builds the prefix for the temp and its children. I realize going over my code now that I did not need to store all my nodes in a vector. I only did this as a quick way to print the character and prefix code but it gave me another O(n) runtime and takes up at least 25n bytes of space. Finally, I used a map of character and prefix codes to encode the file because it can do the fastest lookup per char. This is an O(n) runtime worst-case and same 5n+48 bytes of space for the map.

For the decoding section, I read in the prefix code and characters and saved it in a map. A hashmap was the best way to pair the two together for later when I rebuild the hoffman tree. This is O(n) complexity and taking up at least 5n+48 bytes of space. Next, I built my tree. Since I didn't have a separate tree class, I just made a method and passed in the string of all the bits, an empty string to build my prefix, the map of characters and prefixes, and the root node. This would have been much easier with a tree class I see in hindsight. While I build the tree, first I make sure the entire length of bit string is longer than 0. Then I check if the prefix builder string is a valid prefix code. If it is, I would set the current node's prefix to that string and its char to the matching value in the map and return to the top of the tree to start over again. However, if the prefix-builder string isn't a prefix, I will make a new node as the child, add 1 or 0 to the prefix-builder, and start the recursion again on that node. This traverses the tree and builds prefix codes and has a worst-case runtime of O(n) because the resulting structure if just a binary tree whose shape is dependent on the string input. The worst-case space is 25*n bytes because my nodes are 25 bytes. Once the tree is built, I decode the string of 1's and 0's using basically the same method. No other data structures are used. I only pass in the root node of the tree and the string itself. If the first character is a one, I increase the count of how many steps taken and move call the recursive function on the right node and vice versa if the character is a zero. If I hit a leaf node, I print the node's char, take out the number of chars from the string that it took to get there, set the recursion to start over at the root node. There was no map necessary unlike in other steps because the tree served as the key of characters if we followed the bits. The complexity is O(n).