//Yujian Li(yl7kd) Section 102 Postlab10.pdf 04/22/16

Postlab10

This lab consisted of two parts: the huffman decoding process and the huffman encoding process. I first modified the heap code given to serve as the basis of the encoding process. When reading the characters from the normalx.txt, I created an array with the size of 129 elements which will cover all the English alphabets in ASCII value. Every time the char's int value matches with the index, I add one to the value on that index to store the frequency the character occurs. Then for all the character that occurs more than 0 times will be converted as a huffmannode and inserted into the heap tree structure that I built. In the huffmantree.cpp, I create a method that will pop the values inside heap and for a huffmantree and later I can use the recursive functions to print out the entire tree which contains the both the character and its code. During the print method, I also used the map data structure to store all the code of the characters (at the leaf of the tree) inside the tree. The key of the map is the int value of the characters and the value is the code. In order to print the all the encoded version, I can just print out the given code inside the map once the character insde the file matches the key of the map. The compression size can be obtained inside the method by take the length of the string code.

In the deconding section, I created a huffman tree structure again using recursion. Once the program starts to read the prefix code, if the code is a “0” and the left child is NULL, then the node will create its left child and if the code is a “1” and the right child is NULL, then the node will create its right child until both the code length reaches zero which mean the tree reaches its leaf and assign the character into the node's content. Then I restore the node for a new recursion process. When printing out the characters from the tree. I used the same method and recurse down the tree. If the code is “0” then it continues to go left until it reaches to the leaf then prints out the content and restore the node back to the beginning.

Time complexity:

In encoding process, the values are inserted into the array at a linear run time, which depends on the length of the file length n. (O(n)). Then the heap tree takes in all the value with a O(logn) runtime. For the huffmantree; in general it will take a linear time to construct and retrieve values inside the tree if the data is sorted. However, consider the worst case scenario, it may take O(nlogn) runtime to for a new tree and print out all the values inside the tree. In storing the code into the map, the map data structure will generally have a linear runtime for either inserting and retrieve values which make the later printing the encoded value a runtime of O(n^2logn).

In the decoding process. The values are read was at first read by the program through a linear runtime O(n) depends on the length of the file. Then the values are stored inside the tree through the recursion process. As mentioned before, the constructing process will take O(nlogn) runtime and so it is for printing all the decoding values.

Space complexity:

Encoding process:

Huffmannode structure is consisted of a character (1 bytes), a integer (4 bytes), a string(which is 1*n bytes depending on the size of the string), left and right pointers 8 for each. Which will take a total of 21+(1*n) bytes.

The int array with the size of 128 that stores all the values for the characters and frequencies takes a total of 128*4 =512 bytes

The heap is consisted of a vector that consisted of huffmannodes. So the total space it takes equals the size of the vector(128 in my code) times the space of each huffmannode = 128*(21+(1*n)).

Since we make the tree from the heap, it will take the same space as the heap.

The map that stores the code will take the number of huffmanodes n and times the length of each string s = n*s

Decoding process:

The two node structure take 2*(21+(1*n)) bytes in total.

The tree will take the space which depends on the total amount of prefix's length times node's size.