Hacettepe University

Department of Computer Engineering

BBM203 Software Laboratory I

Programming Assignment IV

Name and Surname: Abdullah Palaz

Student Number: 21993016

Subject: Trees in C++

Submission Date: 17.12.2020

Due Date: 01.01.2020 – 23:59

Advisor: Yunus Can Bilge

Problem Definition: It is expected to encode the given text with Huffman Encoding, then decode the code that has been created by encoding procedure. It is also expected to print the Huffman code of a given character, to the screen and print the Huffman Tree, again, to the screen.

Solution: According to the command line input, program checks a few if – else statements. With these statements, program determines whether the "encode", "decode", "show tree" or "Huffman code of a char" part will be run. Since there must be a connection between these runs, say you encoded a string then in the next run you need the tree (or something like that) to decode the code, when you run the encode command, program creates a two text files called "charHuffmanMap.txt" and "huffmanTree.txt" which contains all nodes in the Huffman Tree and huffmanTree itself. You will see more on that in the upcoming paragraphs. Simple explanation was that.

Data Structures:

As data structures, we have:

- a struct called heapNode
- a struct called comparator
- a vector<heapNode*> called heapVector
- a priority_queue<heapNode*, std::vector<heapNode*>, comparator> called heap
- a map<char, std::string>* called charHuffmanMap

heapNode is node element of the Huffman Tree, it has letter (char), frequency (unsigned int), leftChild and rightChild (heapNode*) as a member variable as well as a constructor.

comparator is used for comparing the heapNodes in heap (priority_queue) according to their frequencies, from less to more.

heapVector is does not store something at the end but is used in the process. While creating the heapNodes from string, program first stores those nodes in heapVector instead of heap directly. This happens because I could not find a way to traverse the heap (priority_queue). So that I first stored these nodes in heapVector and checked if they exist or not with heapVector. More explanation on this will be in the algorithm section.

heap is used for automatic sort whenever you push a new element to the data structure. In this instance, elements are heapNodes. In heap, heapNodes are sorted by their frequencies from less to more as mentioned above.

charHuffmanMap is basically holds the leaf nodes in the Huffman Tree (which are letters of the string that needs to be encoded) in <letter, huffmanCode> pairs (check heapNode's member variables). huffmanCode of a letter is determined by a function (named huffmanCodeFinder) which will be explained in algorithm section.

The "**charHuffmanMap.txt**" is coming to the game here. After creating the charHuffmanMap, I write that data to charHuffmanMap.txt for future use. The file has a format, for line:

letter – space – huffmanCode

To mention, "**huffmanTree.txt**" contains the tree. For middle nodes, it just contains their frequency but for letter (leaf) nodes it contains just letters, nothing more. When you enter "-l" code, you will see that tree in your screen. In nowhere else this file is being used.

Algorithms:

Encoding Algorithm:

Since the first (excluding program name as an argument) argument of the line, we enter that section of the code. Then we open the file with the name of second algorithm. This part is the same for decoding too. Since this part is same for both encoding and decoding, the program doesn't actually enter the "encoding part" till here. You can find where this is in the code by checking the code's comments. Now we are checking 3rd argument which is should be -encode this case. And here we are entering encoding part of the code.

For your ease to read, each code block has it's explanation UNDER it. Now the explanation. We first get the line in the file as "stringToEncode" (which is a std::string). Then for each char in the stringToEncode, we create a heapNode and put it to heapVector if a heapNode with that char does not exist. If a letter with that char does exist in heapVector, then we add 1 to it's frequency. In the end we have a heapVector which contains heapNodes with unique letters and their frequencies. Frequency = number of times a char appears in a string. Letter = check the data structures.

Now we are pushing all the heapNodes in the heapVector to the heap (which is a priority_queue). This helps to sort them by their frequencies, from less to more.

Now we need to create middle nodes. For that, while heap's size does not equal to 1 (which will be root element in the tree after that process), we take first two element from the heap (which will be the elements with least frequencies) and create a middle node (letter is '|') with the frequency of sum of these two nodes. And we then push that node to heap. In that way, the heap will be sorted always.

Now we have a Huffman Tree. It's time to find the Huffman codes of the letters (which are leafs of that tree). When we find the For that, we have a function with name and arguments:

```
void huffmanCodeFinder(struct heapNode* nodePtr, const std::string
huffmanCode)
```

This function is taking the root node of the Huffman Tree as nodePtr and an empty string as huffmanCode. The huffmanCode is modified during the process. The algorithm is:

If nodePtr is nullptr, then return

If nodePtr's letter does not equal to "|" then we add <letter, huffmanCode> pair to charHuffmanMap.

huffmanCodeFinder(nodePtr->leftChild, huffmanCode + "0");

huffmanCodeFinder(nodePtr->rightChild, huffmanCode + "1");

The function is recursive. If we are going to left then huffmanCode is added 0, if we are going to right then huffmanCode is added 1. If the node is a leaf node (which is determined by if that node's letter is '|' or not), then we add the pair to charHuffmanMap, which is a map.

Since we have the huffmanCodes of all letters, now we can print the encoded string, which is the total code.

For that, for each letter in the stringToEncode, we are searching the charHuffmanMap. We are using an iterator for searching. Iterator's first (iterator->first) is the letter and iterator's second (iterator->second) is the huffmanCode. If letter equals that char in the stringToEncode, then we print that letter's huffmanCode to screen. In total we have the full code.

The encoding part is over but there is two code blocks more in the code file as you can see. First block is used for writing this charHuffmanMap (<letter, huffmanCode> pairs) to a txt file named "charHuffmanMap.txt". We will use this file in -s command.

Second block is where we are writing the tree to a txt file named "huffmanTree.txt". This file will be used in -l command.

Decoding Algorithm:

We did open the file in the common part with the encoding part of the code. We get the code from that file and put it to stringToDecode.

Then we open charHuffmanMap.txt file. For each line of that file, we pushing the <letter, code> pairs to charHuffmanMap (a map, data structure declared in global).

Now it's time to decoding. We have a string called codeForOneChar. For each char is stringToDecode, first we are adding that char to codeForOneChar. Then if that codeForOneChar is equal to any code of the charHuffmanMap (data structure), then we print the letter with that code from charHuffmanMap (the data structure one).

And that's it, in the end we have the decoded string in the screen.

-s Command Algorithm:

We are taking the char in the algorithm. Then we open charHuffmanMap.txt file. Then for each line of the file:

If line's letter is equal to char, we print the huffmanCode of that char to screen.

That's it!

-l Command Algorithm:

Actually, printing the tree in this part is just printing the whole "huffmanTree.txt" file. Important part (writing the tree to txt file is the important one) was handled in encoding part. Here is how it was handled:

There is a function with name and parameters:

```
void printTreeToFile(struct heapNode* node, std::string prefix, std::ofstream
&huffmanTreeFile)
```

This function takes root node of the Huffman Tree as node in the parameters, prefix is an empty string, huffmanTreeFile is the txt file named “huffmanTree.txt”. Function writes the tree in a recursive way.

If node is nullptr then return

If node’s letter is ‘|’ (meaning middle node) then write \n and then write prefix<<“-“<<node’s frequency

If node’s letter is not ‘|’ then write \n and then prefix<<node’s letter (this letter is in quote marks “”)

The recursive calls’ prefixes for left child call:

old prefix’s length times “space” + “ left child is ”

for right child call:

old prefix’s length times “space” + “ right child is ”

The recursive calls are:

Left child call:

```
printTreeToFile(node->leftChild, std::string(prefix.length(), ' ')+" left
child is  ", huffmanTreeFile)
```

Right child call:

```
printTreeToFile(node->rightChild, std::string(prefix.length(), ' ')+" right
child is ", huffmanTreeFile)
```