

Data Structures and Algorithms

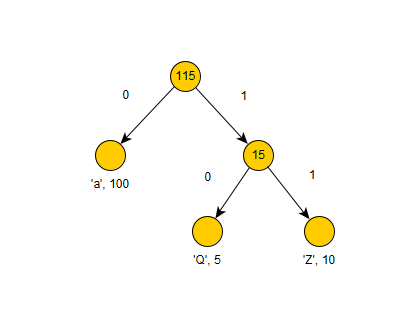
**Lab 8: Huffman.h**

**The Scenario**

This is it. The final lab of DSA. Despite all the hats you’ve worn in this class, all the adventures you’ve gone on, there is one common thread to each of these lab adventures. And that thread is that it’s always been you working on these labs, of course! You’ve worked through a variety of problems and I want to thank you for all of your hard work this month! For the final lab, you will be assuming the role of yourself—we will be looking at something called **Huffman Compression**.

**Huffman Compression** is a way of using Binary Search Trees organized by weight (in our case, frequency) as a means of compressing data. i.e. Let’s say you have a text document containing the entirety of J.R.R. Tolkien’s “The Hobbit”. You could certainly store it the uncompressed way with 8 bits per ASCII character in the document. But what if you want it to take up less space on your hard disk? How about compressing it for sending out via e-mail? Huffman compression is one solution that allows for lossless compression of files.

Take for example, a file filled with 115 characters: All of which are ‘a’, ‘Z’, and ‘Q’. There are 100 ‘a’s, 10 ‘Z’s, and 5 ‘Q’s. Each ASCII character normally takes up 8 bits of storage. The goal of Huffman Compression is to reduce this from 920 bits down to something less substantial. Below is the tree generated for the compression.



You may be wondering what each number is. Each leaf-node (node with no children) is the document’s character, followed by its frequency in the document. A 0 is assigned to each left path and a 1 is assigned to each right path from the root. This will be used to generate our new encoding table.

As you can see in the figure above, ‘a’ would have a code that consists of a single bit: 0, while the less-common ‘Q’ and ‘Z’ would both be comprised of 2-bit codes: 11 and 10 respectively. This reduces the total size of the document to 120 bits instead of the original 920 bits. This is the power of Huffman Compression!

For this lab, you’ll be generating a tree along with a list of the leaves, a frequency table consisting of the relative frequency of each character in a document, generating the encoding table, and doing other general tasks associated with Huffman Compression. In this lab, the comments in the code are a little bit more thorough than normal, so be sure to read them all very carefully!

**What To Do…**

Open Huffman.h. There will be instructions written in the comments on what is expected. Below is the gist of each function and variable…

***Variables:***

**mFileName** The name of the file for reading and writing.

**mFrequencyTable** Storesthe frequency of each character.

**mLeafList** Contains all of the leaf nodes—the characters.

**mRoot** The root of the Huffman tree! Set in GenerateTree instead of constructor.

**mEncodingTable** The encoding table generated from the tree.

***Functions:***

**Huffman Constructor** Takes a file name and sets it. mRoot is set in GenerateTree instead, however.

**GenerateFrequencyTable** Opens the file (defined in the variables!) and specifies it is in binary mode. Stores the count of the file in the final position of mFrequencyTable, and reads the file one byte at a time, incrementing each character’s position in mFrequencyTable. Don’t forget to close the file when you’re done!

**GenerateLeafList** Loops through mFrequencyTable, creating new huff nodes and putting them into mLeafList for each character with a frequency greater than 0.

**GenerateTree** Create a priority queue and use HuffCompare in it for the comparison crieteria. It will store HuffNode\*s. Next, add all the values from mLeafList in. Now comes the tree generating part…

While the queue has more than 1 node left, set two temporary pointers to the top two nodes. Pop them off of mLeafList and create a parent node for the two, setting the 1st node as left and the 2nd as the right. Then make the parent’s value -1 and set its frequency to the sum of its two children’s frequency. Set both node 1 and node 2’s parent to be the created parent node.Insert that new node into the priority queue. Once you finish, take a sip of coffee, tea, or water—you deserve it! Finally, Once there is only one node left in the queue, you can set that node to be mRoot.

**GenerateEncodingTable** Now that your tree is made, you will write the code to generate the encoding table. Cycle through each node in mLeafList and move up from each all the way to the root. Keep track of whether each node is a left or a right child of its parent and at the end, reverse the sequence of bits. (This will show the route you must take from mRoot to the leaf.) Remember, the encoding table’s position is based on the value of each leaf.

**ClearTree** Call the helper function on mRoot and then afterwards, set mRoot to null.

**ClearTree (Helper!)** Accepts a single HuffNode\*. Recursive helper function for ClearTree. Recurses through both the left and right branches of the Tree and cleans up all dynamically allocated memory.

**Compress** Takes a single parameter—the output File to write the data to. First, the function calls the functions you’ve written, creating the frequency table, leaf list, tree, and encoding table. Next, creates a Bit Output Stream (BitOStream) and supplies it the header, along with the file’s size. Next, opens the input file as an ifstream in binary mode. After this, does the actual compression—replaces every character in the original file with its bit-code from mEncodingTable. After all is said and done, closes both streams and Clears the Tree. If you used any dynamically allocated memory for this, deletes that as well.

**Decompress** Does the opposite of Compress! Accepts the file to write the uncompressed data to. mFileName will be the compressed file. Creates a BitIStream and reads the frequency table, then generates the leaft list, tree, and creates a stream for output in binary mode. Next, creates a char for writing and a temporary Boolean for reading into and traversing using. Next, create a node pointer to bookmark your place in the list—i.e. starting at mRoot. Go through the compressed file one bit at a time next, traversing through the tree. Once you reach a leaf, you know you’ve finished with the current bit and can write the value to the file. Next, close the streams and clear the tree.

**Tips, Tricks, and Resources**

* Functions/Data Members available in the priority queues class can be found on the Cplusplus.com documentation:
  + <http://www.cplusplus.com/reference/queue/priority_queue/>
* Don’t be scared to draw your trees out if needbe!
* Thank you for all of your hard work in lab this month!

**Plagiarism**

Plagiarism and Academic Dishonesty are considered a **very** serious offense in this class and can have a range of consequences including suspension, and in very serious cases, expulsion. If you either share your code or copy someone else’s code, you will be given a **0** on your lab and can face further disciplinary action.

In other words, don’t cheat please!