- 3460:316 PROJECT 2 : HUFFMAN CODE TREES -

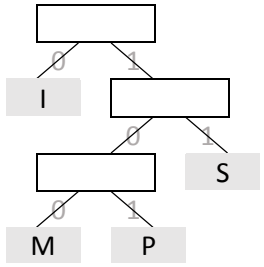
Text compression is an important text processing task in which a string defined over an alphabet (such as the ASCII or Unicode character sets) is efficiently encoded into a small binary string. This is useful when wanting to minimize the time to transmit text over a low-bandwidth channel or efficiently storing large collections of documents.

One popular text compression method is the *Huffman code*, which uses a variable-length encoding optimized for the specific string in question. The optimization uses character frequencies to save space by using short code-words for commonly occurring characters and long code-words to encode low-frequency characters. An example of a Huffman code is pictured below.

Input string: MISSISSIPPI

Character	I	M	P	S
Frequency	4	1	2	4
Encoding	0	100	101	11

Output string: 100011110111101011010



As you can see Huffman's algorithm is based on the construction of a binary tree representing the code. Each external node is associated with a specific character, and the code word for that character is defined by the sequence of bits associated with the nodes in the path from the root to the character. The code created is a *prefix code*, in that no code word is a prefix for any other code word. This simplifies the decoding of the output string to retrieve the original text.

You will write a program to create a Huffman coding tree based on an input string in order to encode that string.

Files

You are supplied with these completed files:

- `makefile` - use `make` on the linux system to compile the project using this makefile. The binary file will be named `huffman.out`.
- `Huffman.hpp` - This contains the header file for Huffman object as well as the `Encoded struct`.
- `Heap.hpp` - This contains the header file for a min-heap class that stores `HNode*`.
- `HNode.hpp` – This is the `struct` for the nodes to be used inside the heap.
- `test_huffman.cpp` – There are no test cases for this program. Instead your program will output the tree serialization (see below) and the encoded string. Based on your output you can draw the tree and manually check that the encoded string matches.

You must complete the project by completing/writing the following:

- `Huffman.cpp` - There are three functions to be implemented in this file. The `encode` function is already done, but you need to make `Heap.cpp` reflect all calls used within it.

- `Heap.cpp` – Your implementation of the min-heap class will be very similar to the examples we did during the lectures, with appropriate changes to handle `HNode` objects. You must implement the functions `enqueue`, `dequeue`, `fix_up`, `fix_down` and `clear`, and you must correctly update the `count` variable.
- `HNode.cpp` – For leaf nodes, the `value` member is the character and the `weight` member the frequency count. For internal nodes `value` is null and `weight` is the combined count of the nodes in the subtrees. There are two constructors. One creates a leaf node, the other combines two existing nodes to make an internal node. You must implement both.

General Information

- **Make any changes to the `.cpp` files and not the header files.**
- The heap is going to function as a priority queue without the overhead of creating a class around the heap. You are **NOT** allowed to use `std::priority_queue`!
- When encoding, use "0" for a left branch and "1" for a right. In case of ties give precedence to the character occurring first in the ASCII table.
- You ARE allowed to modify the heap code we did in class to implement your heap.
- You ARE allowed to use `std::vector` and `std::map`.

On Serializing Binary Trees

To represent a binary tree as a string, print the nodes in the order that they would appear in a preorder traversal. Print the character (the `value` but not the `weight`) for each leaf and a "*" for internal nodes. Represent a null pointer with a "/". For example the serialized representation for our tree above would be

*I//**M//P//S//

Rubric

- **[20%]** Your program must compile and run on our knuth linux server
- **[10%]** Your Heap must work correctly with HNode pointers
- **[20%]** The Huffman tree must be correct
- **[20%]** The encoded string output must match the given string and Huffman tree
- **[10%]** You must document all functions in your code (within reason)
- **[10%]** You must properly handle memory, memory leaks will cost you points

Submission Instructions

When done, zip the eight relevant files (`makefile`, `Huffman.hpp`, `Heap.hpp`, `HNode.hpp`, `test_huffman.cpp` and your newly-constructed `Huffman.cpp`, `Heap.cpp` and `HNode.cpp`) into one archive and submit it to Brightspace.

Last updated 10.12.2017 by T. O'Neil, based on a project by A. Deeter.

TO-DO CHECKLIST

Finish `HNode.cpp`

- ☐ Creating a leaf node
- ☐ Creating an internal node

Write `Heap.cpp`

- ☐ Implement `enqueue`
- ☐ Implement `dequeue`
- ☐ Implement `fix_up`
- ☐ Implement `fix_down`
- ☐ Implement `clear`

Finish `Huffman.cpp`

- ☐ Implement `create_codes`
- ☐ Implement `serialize_tree`
- ☐ Implement `encode_string`