CS3610 Project 3

Due: Nov. 2, Thursday, 11:59 pm

Having data compressed can save space on hard drives, as well as speed up file transfer online. In this assignment, you will implement a simple, yet elegant, data compression algorithm called Huffman coding `https://en.wikipedia.org/wiki/Huffman_coding`. Let us first walk through how it works.

Huffman Coding

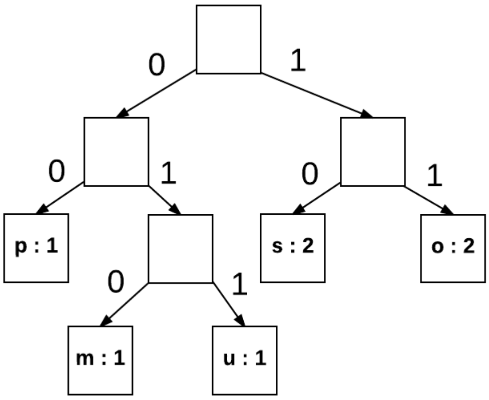
Assume that I want to store the message "opossum" on my computer. This string of characters will be represented on my hard drive as a series of bits (1's and 0's). Typically, an ASCII character takes up 8 bits of space on a computer. This means that "opossum" will cost 56 bits of storage. For the sake of argument, let us assume that this is a significant number.

One obvious way to reduce the number of bits in our message is to represent each character in the message with fewer bits. Of course, using smaller bit sequences will limit the number of unique characters we can encode. For example, a two bit sequence is only capable of representing 4 unique characters. Consequently, our characters will vary in length. This was expected though. So what are the real issues preventing us from using fewer bits? Well, I have two for you right here:

1. Let us assume that the characters in "opossum", 'p', 'm', 'u', 'o', and 's', are represented as 0, 1, 00, 01, and 10 respectively. What problems do you see with this? Well, it just so happens that our most common characters 'o' and 's' are also some of our largest characters. We could have saved ourselves a little more space if we instead represented 'o' and 's' with the single bits 0 and 1, which were naively used to denote 'p' and 'm'. So the question arises, how do we efficiently delegate bit sequences such that the most frequently occurring characters in our messages take up the least amount of space?

2. Let us assume that the characters in "opossum", 'p', 'm', 'u', 'o', and 's', are represented as 0, 1, 00, 01, and 10 respectively. Let us also assume that we have used these characters to encode a new message "0001" on our hard drive. I now ask you, what message did we just encode? Well, unfortunately, it is ambiguous. One possible decoding could be "pppm". Another could be "uo". So the question arises, how do we generate variable-length character codes such that our encoded messages will not be ambiguous? In other words, how do we develop a prefix coding system?

One way to resolve these issues is by constructing a Huffman tree, just like the one displayed right here:



A Huffman tree is a binary tree that encodes the characters of a message into a reduced bit representation by implicitly storing the component bits along the tree's edges. Typically, left edges denote 0 while all the right edges denote 1. Thus, to find the encoding of a specific character, we simply concatenate the bits found along the edges of the path between the root node and the character's node.

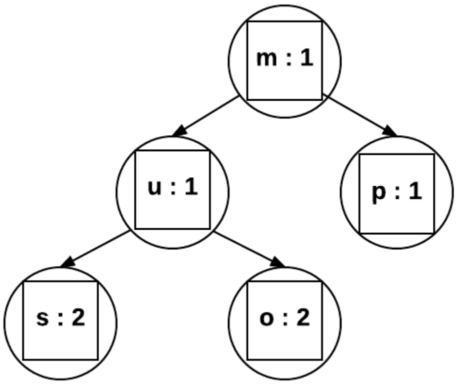
In the tree above, you can see that 'o' is represented as 11 because its path from the root first follows a right edge and then another right edge. Take note that all character nodes are leaf nodes and each path produces a unique string of bits. Also, you should be aware that the most common characters, such as 'o' and 's', appear closer to the top of the tree, which gives them the shortened encoding that we desire. Conversely, the lowest frequency characters reside near the bottom of the tree.

Using the Huffman tree above, "opossum" can now be written as 1100111010011010, which is just 16 bits as compared to the original 56. Better yet, no characters share the same prefix, which means this binary message can be decoded unambiguously. You can try it out for yourself by reading the bits left to right while following the recorded bit path in the tree starting from the root. As soon as you hit a leaf node, grab its character and continue reading the remaining bits starting from the root again. So this is wonderful, I now have a compressed message that was easy to encode and decode. However, the question remains, how do we construct the Huffman tree? Well here is the algorithm:

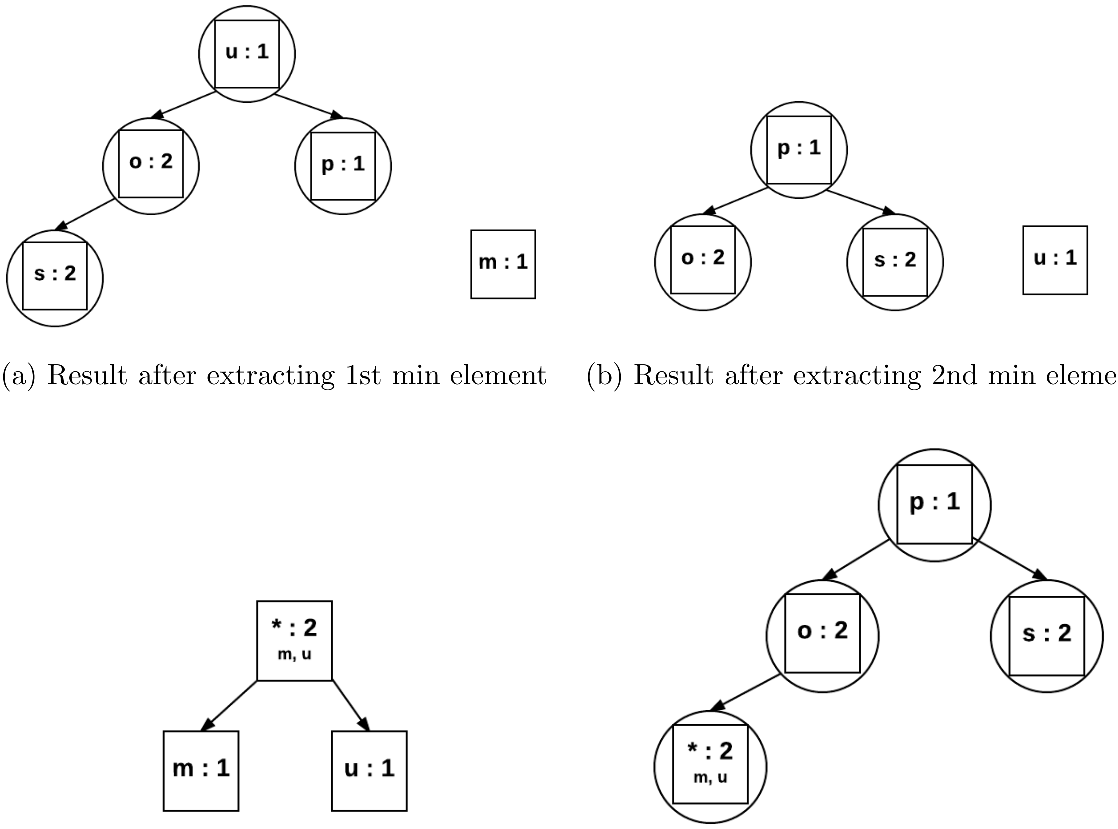
1. Identify the frequency of each character in a given message. The frequency of each character in "opossum" is listed alphabetically as follows:

   $$m = 1 \qquad o = 2 \qquad p = 1 \qquad s = 2 \qquad u = 1$$

2. Package each character-frequency pair into a binary leaf node and insert each node into a min heap using the frequency as the key.



3. Extract the two smallest elements from the min heap and respectively attach them to the left and right pointers of a new internal node. Set the key value of the new internal node to be equal with the sum of the frequencies of the extracted nodes. Insert this new internal node into the min heap.



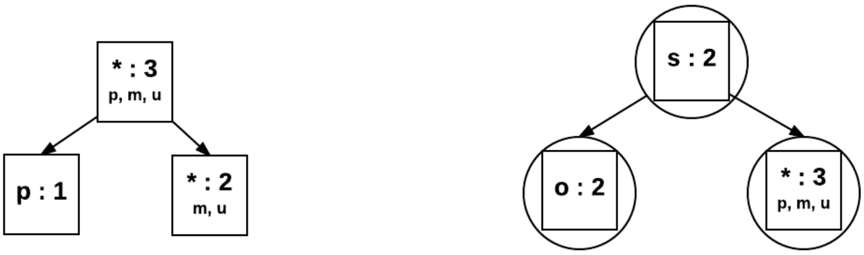
(a) Result after extracting 1st min element

(b) Result after extracting 2nd min element

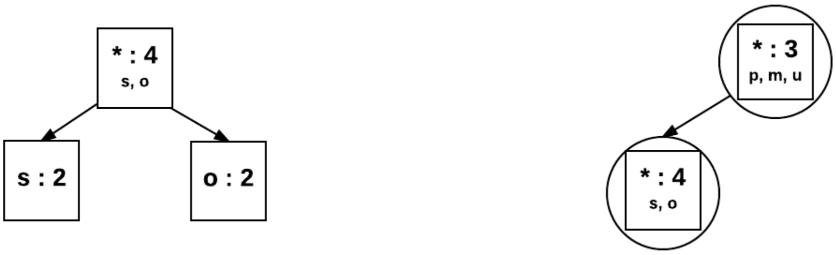
(c) New node from extracted mins

(d) New node inserted into heap

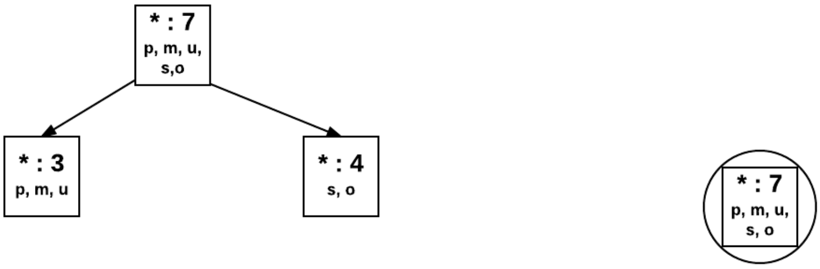
4. Repeat step 3 until the min heap has one element remaining.



(a) New node from extracted mins (b) New node inserted into heap

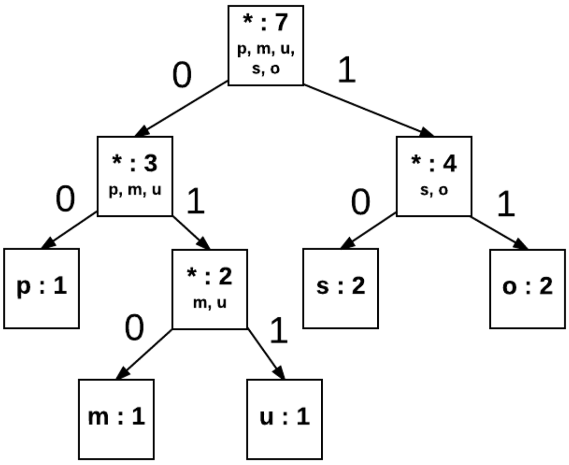


(a) New node from extracted mins (b) New node inserted into heap



(a) New node from extracted mins (b) New node inserted into heap

5. Draw out the newly constructed Huffman tree



Task

For this project, you have been provided an implementation of the Huffman tree construction algorithm (DO NOT modify this code. It is possible to construct multiple Huffman trees with the same message if numerous characters in the message share the same frequency value. The `construct` function that I set up ensures that your Huffman tree will produce results consistent with the test cases used for grading). This algorithm utilizes a templated min heap class that you must implement. You must also implement a function that prints the Huffman encoding of `message` using the Huffman tree constructed from the same message. Specifically, you must implement the following functions:

`void MinHeap::insert(const T data, const int key)` :
: Insert the provided user data into the min heap using the provided key to make comparisons with the other elements in the min heap.

  To ensure that your min heap produces consistent results, stop bubbling up a child node if it shares the same key value as its parent node.

`T MinHeap::extract_min()` :
: Remove from the min heap the element with the smallest key value and return its data.

  If you come across two sibling nodes that share the same key value while sifting down a parent node with a larger key value, then you should swap the parent node with the left child to ensure that your min heap produces consistent results.

`T MinHeap::peek() const` :
: Retrieve the minimum element in the min heap and return its data to the user. Do not remove this element from the min heap in this function.

`void MinHeap::size() const` :
: Return the size of the min heap

`void HuffmanTree::print() const` :
: Print the Huffman encoding of the member variable `message` assigned in the `construct` function.

To ensure that you always produce a consistent output, DO NOT modify the completed code in the `HuffmanTree` class. You may however add `print` helper functions if you feel it necessary.

In heap insert() and extract_min() functions, parents and children may swap their values. You can consider using swap() function available under STL for this purpose.

Input

Input is read from the keyboard. The first line of the input will be an integer $t > 0$ that indicates the number of test cases. Each test case will contain a message on a single line to be processed by the Huffman tree `construct` function. Each message will contain at least 2 characters.

Output

For each test case, print `Test Case:` followed by the test case number on one line. On another line, print the Huffman encoding of the input message. Separate the individual character encodings by a space.

Sample Test Cases

Use input redirection to redirect commands written in a file to the standard input, e.g. `$ ./a.out < input1.dat`.

Input 1

```
3
opossum
hello world
message
```

Output 1

```
Test Case:  1
11 00 11 10 10 011 010
Test Case:  2
010 011 10 10 00 1100 1101 00 1110 10 1111
Test Case:  3
011 10 11 11 010 00 10
```

Timing Analysis

At the top of your main file, in comments, write the time complexity of constructing a Huffman tree with a min heap in terms of the number of characters in the input message, which you can denote as n. Also consider the time complexity of constructing a Huffman tree without a min heap. Specifically, what running time can you expect if you use a linear search to find minimum frequencies. Write these time complexities using Big-O notation.

Turn In

Submit your source code through blackboard. If you have multiple files, package them into a zip file.

Grading:

Total: 100 pts.

- **10**/100 - Code style, commenting, general readability.

- **05**/100 - Compiles.
- **05**/100 - Follows provided input and output format.
- **75**/100 - Successful implementation of the min heap and Huffman print functions.
- **05**/100 - Correct timing analysis.