UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
SCHOOL OF COMPUTER SCIENCE & APPLIED MATHEMATICS

COMS1017A/COMS1021A
Intro. to Data Structures & Algorithms
Project 3: Huffman Coding

Dr Richard Klein

Due Date: Thursday 5 November 2021, 08:00

1 Description

The American Standard Code for Information Exchange (ASCII) represents characters by a string of 7 bits. The letter A is represented by the bit string 1000001 (65), B by 1000010 (66) and so on. There is room for 128 characters including lower-case and special characters. Over time, ASCII has been superseded by the newer UTF-8 character encoding. ASCII characters were included in UTF-8 as the first 128 characters, which means that UTF-8 is backwards compatible with ASCII.

Data compression is an important technique used in data storage and transmission over communication networks. There are 2 basic types of compression: lossy and lossless. This involves how well the data can be decoded. Lossless compression means that absolutely no information is lost in the coding/decoding process. For example, files that are compressed with zip compression are lossless. You would use this type of compression when errors in decoding are unacceptable – eBook/text formats for example. The file you compress is exactly what you get when you unzip it. Lossy compression methods, on the other hand, are allowed to discard some information in order to make the file smaller. Many video (MPEG, OGG, H.264/H.265[1]) and audio (MP3) compression formats are lossy. They rely on the fact that our brains can make up for the lost information or that there are parts of the data that we cannot sense. MP3 for example, drops many of the high frequencies that a layperson wouldn't notice anyway. The more frequencies it drops, the less information we need to store and the smaller file becomes. On the other hand WAV and FLAC (open source) are a lossless audio compression formats. In the context of Audio/Visual compression, we often speak in terms of a *codec*, which stands for **co**der-**dec**oder – or **co**mpressor-**dec**ompressor (depending on who you ask).

Huffman coding is an *entropy encoding algorithm* used for lossless data compression. Think of entropy as a measure of randomness or uncertainty. Huffman encoding replaces the codes of characters that appear most frequently with shorter codes. Huffman codes have the nice property that, if the weights are sorted, the time it takes to encode or decode a message is linear in the length of the message to be encoded. In general, however, Huffman encoding takes $O(n \log n)$.

The key to Huffman coding is the so-called prefix constraint, i.e. the code for any character may not be the prefix of any other code. For example, if the string `001` is the code for some character, then no other code starts with `001`. A Huffman Code is an optimal prefix code that guarantees unique decodability of a file compressed using the code. The code was devised by David A. Huffman at MIT in the early 1950s. Huffman coding is a technique for assigning binary sequences to elements of an alphabet. The goal of an optimal code is to assign the minimum number of bits to each symbol in the alphabet, given some frequency distribution.

This results in a *Variable Length Code*, where characters that appear often use fewer bits than characters

[1] mostly

that rarely appear at all. For example, if we know that the letter `e` appears more often than the letter `z` we should try to use fewer bits to represent `e` than we do `z`. ASCII and UTF-8 are examples of *Fixed Length Codes* as every letter uses the same number of bits regardless of how many actually occur in a text – 7 or 8 respectively.

In this project, you will have to implement a system that allows encoding/decoding of messages into and from Huffman code.

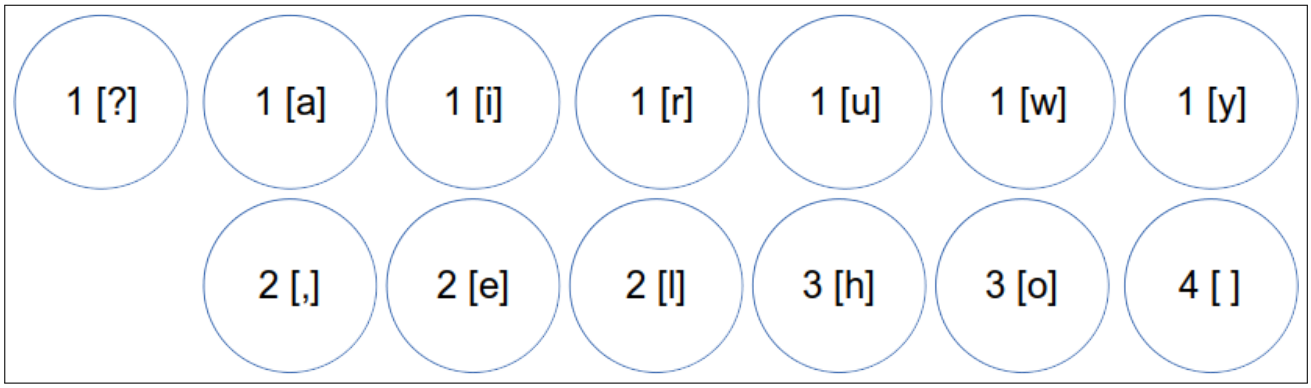
2 Huffman Coding

Huffman coding works by using binary trees. First you need the frequency at which each letter/symbol appears. This is easy to do in linear time. Suppose we have the following sentence: `hi, hello, how are you?` We have the following frequencies:

Character	Frequency	Character	Frequency
?	1	,	2
a	1	e	2
i	1	l	2
r	1	h	3
u	1	o	3
w	1	space	4
y	1		

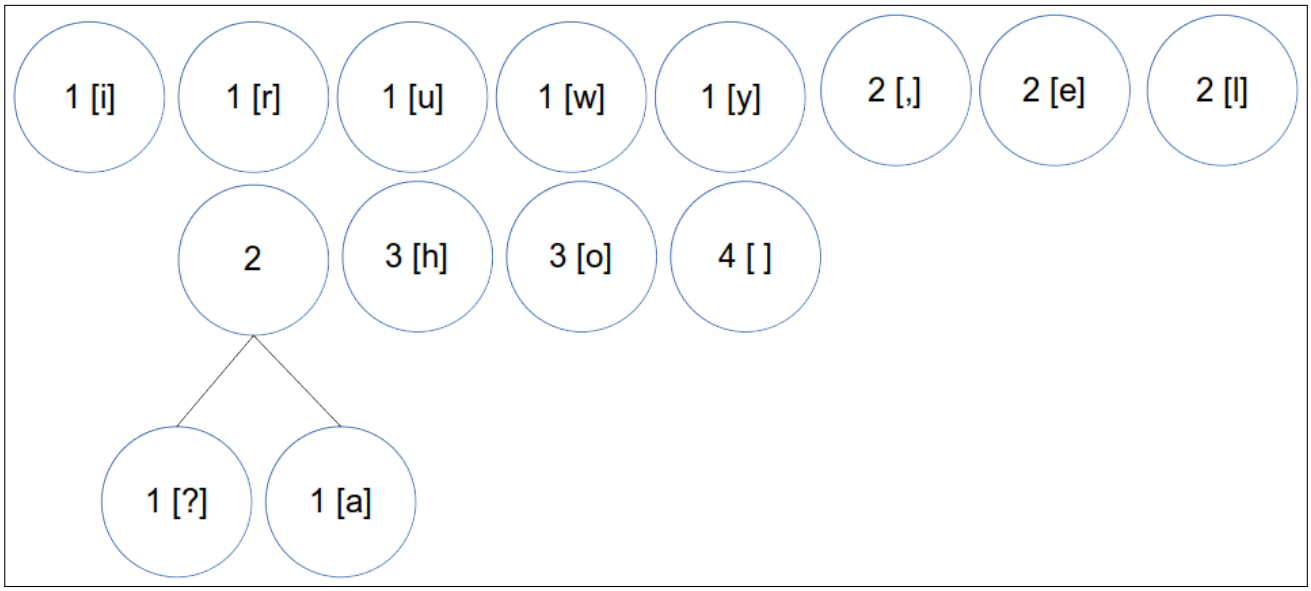
Table 1: Frequency Distribution

We create a node for each unique character, and associate the frequency with each. These should be placed in a min-priority queue.



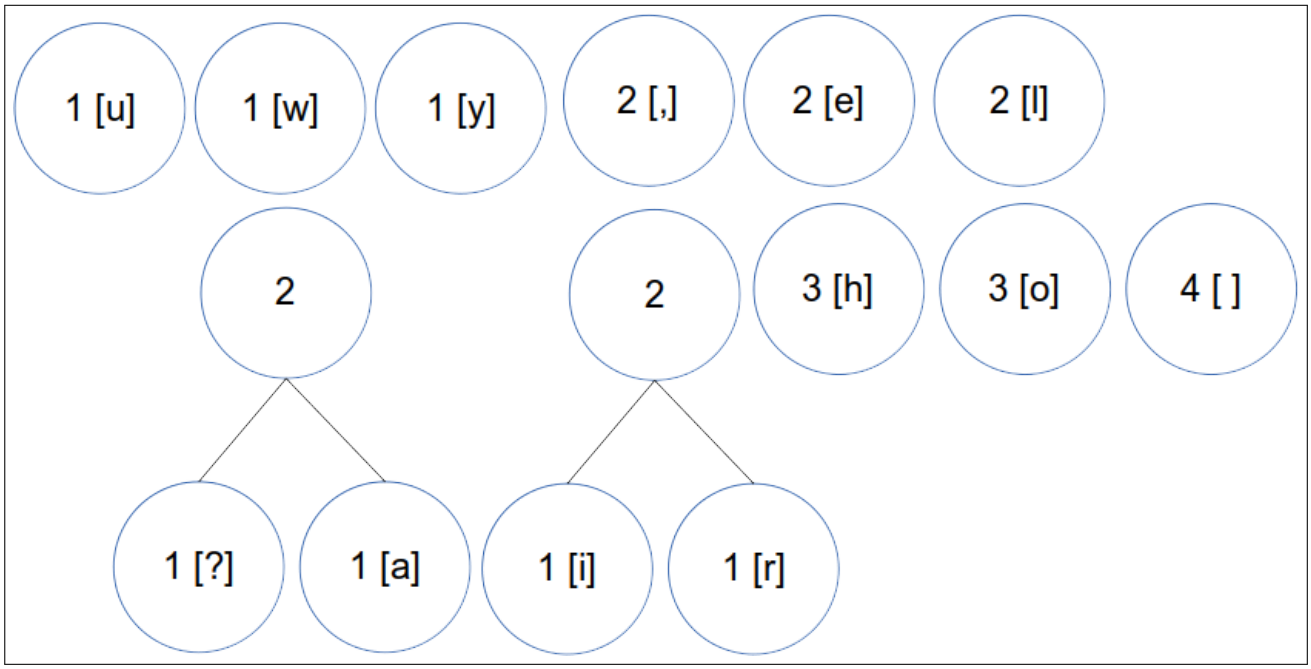
We continue, by taking the 2 nodes with the smallest frequencies, and joining them as children of a new node. The new node should contain the sum of their frequencies. In the example above, we remove the first 2 nodes from the priority queue, these correspond to `i` and `w`. The sum of their frequencies is therefore 2. The new tree that has been created, is then placed back into the priority queue.

Step 1:

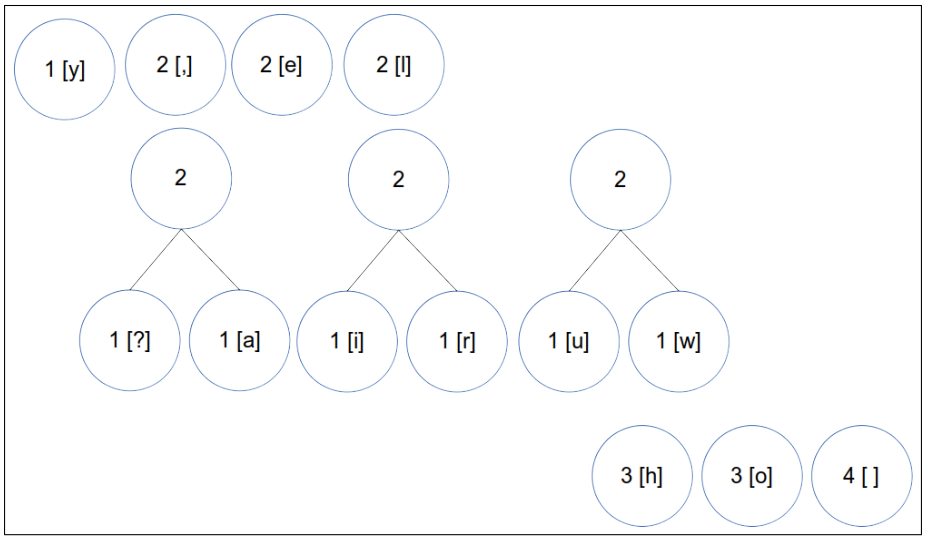


This process is repeated:

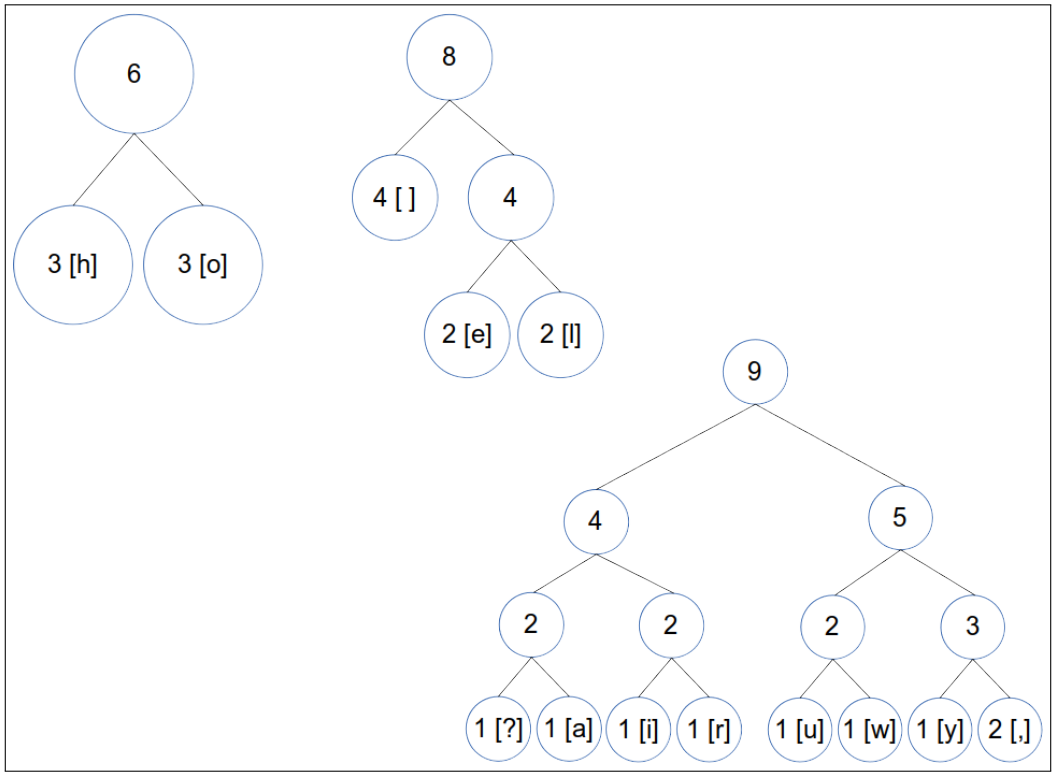
Step 2:



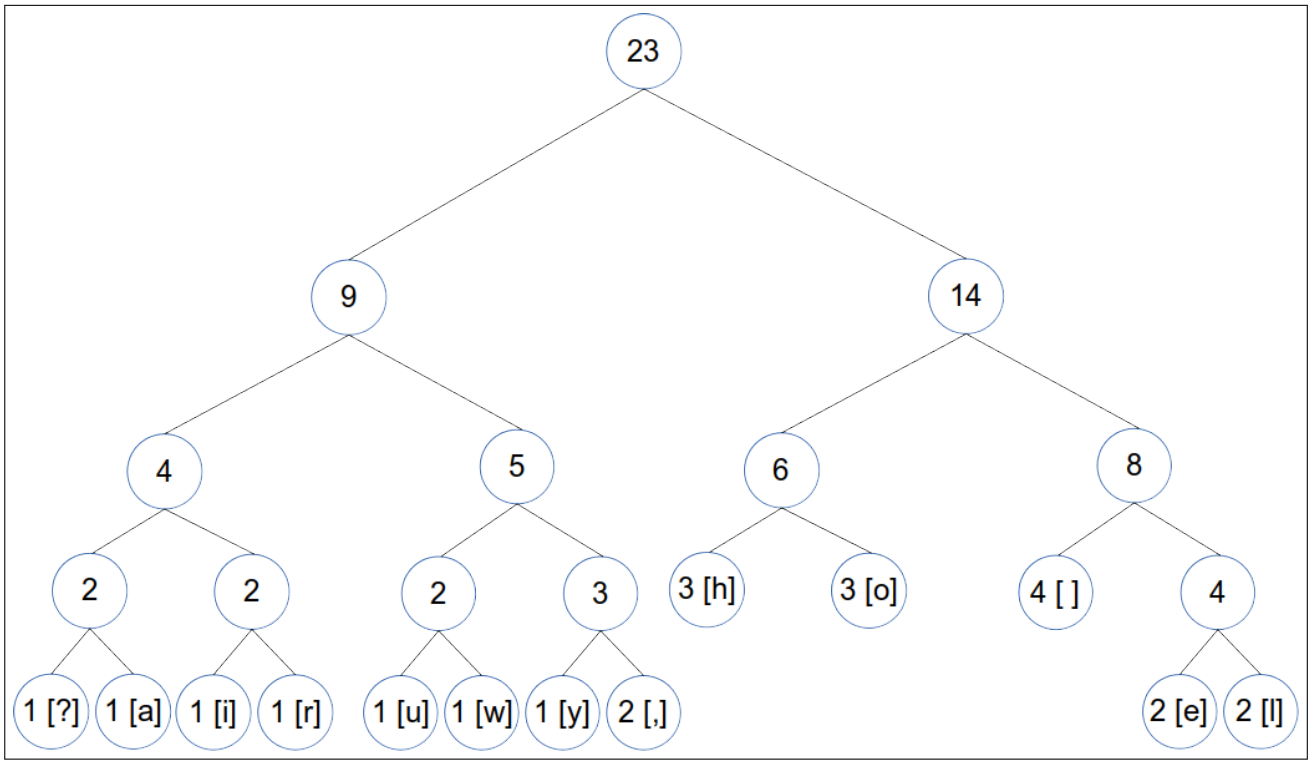
Step 3:



Step 10:



The process repeats until there is just one tree left:



To encode or decode something using the tree, we say that for each left path we follow we get a 0 and for each right path we get a 1. For example, `0010` means that we go left, left, right, left. This becomes the new code for character `i`. Character `h` on the other hand occurs more frequently and is therefore higher up in the tree at `100`.

In a fixed length encoding, we would have needed at least 4 bits in order to represent the 13 different characters, making the sentence a total of 92 bits (4 bits $\times$ 23 characters). This is shown in Table 2 below. If we used ASCII or UTF-8 encodings we do even worse with 161 and 184 bits respectively.

Character	Frequency	Code	Bits	Character	Frequency	Code	Bits
h	3	0000	12	r	1	0111	4
i	1	0001	4	w	1	1000	4
e	2	0010	8	a	1	1001	4
l	2	0011	8	y	1	1010	4
o	3	0100	12	u	1	1011	4
,	2	0101	8	?	1	1100	4
space	4	0110	16				

Table 2: Fixed Length Encoding, 92 Bits

In the variable length encoding given by the tree above the bits required to represent the string drop by 13 bits down to 79:

Character	Frequency	Code	Bits	Character	Frequency	Code	Bits
h	3	`100`	9	r	1	`0011`	4
i	1	`0010`	4	w	1	`0101`	4
e	2	`1110`	8	a	1	`0001`	4
l	2	`1111`	8	y	1	`0110`	4
o	2	`101`	6	u	1	`0100`	4
,	2	`0111`	8	?	1	`0000`	4
space	4	`110`	12				

Table 3: Fixed Length Encoding, 79 Bits

Here is another example: `she sells sea shells by the sea shore`
In a fixed length code, the 11 unique characters would require 4 bits each. This means the sentence would need 148 bits at least. In ASCII or UTF-8, we'd need 259 or 296 bits respectively.

Character	Frequency	Code	Bits	Character	Frequency	Code	Bits
s	8	`10`	16	y	1	`11100`	5
h	4	`010`	12	t	1	`111101`	6
e	7	`00`	14	o	1	`111011`	6
l	4	`011`	12	r	1	`111100`	6
space	7	`110`	21				
a	2	`11111`	10				
b	1	`111010`	6				

Table 4: Variable Length Encoding, 114 Bits

Encoding is done by building the code column of the table above. This can be done easily by using a Depth First Traversal. The characters are then simply replaced by their relevant code in the output. For example, the first 6 characters of the sentence would result in the following binary stream: `011001101110110`. To decode, one simply follows the tree left or right depending on whether one reads a `0` or a `1`. For the above encoding, the first number is `0` which means we go left. Then we read a `1` so we go right. At this point we have reached a leaf, so we output the character `s` and return to the root. We then read a `1` so we go right. We read a `0` and go left, followed by another `0` so we go left again. We are at a leaf again, so just output the character and go back to the root. This process repeats until the text is decoded.

3 Deliverables

You need to write 2 programs. The first program must accept a sentence on `stdin` and output the characters and frequencies for the Huffman tree, the encoded sentence, the number of bits used originally and the number of bits (8 bits per character) in the compressed stream.

The second program must accept the character frequencies and reconstruct the Huffman tree. It should then read the encoded string and return the decoded output.

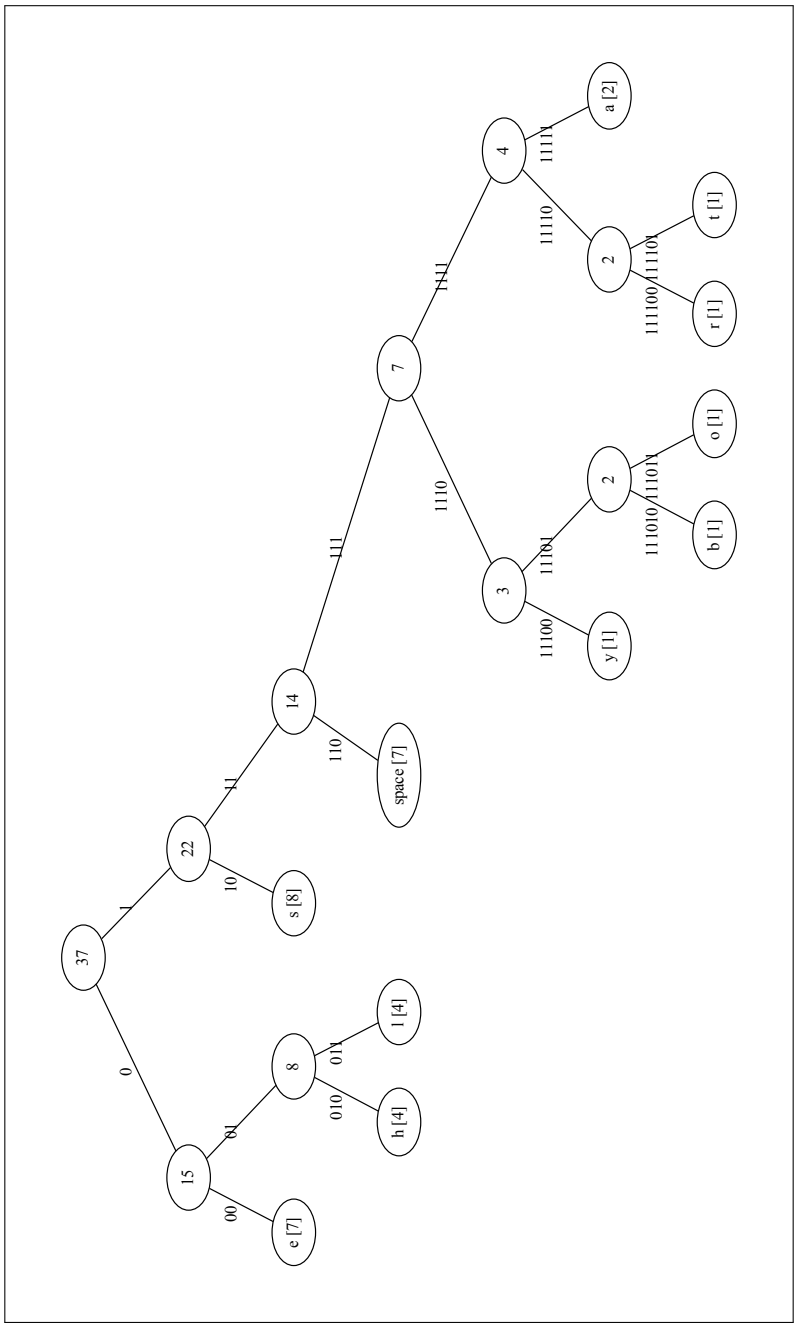


Figure 1: Huffman Code Tree for `she sells sea shells by the sea shore`

3.1 Encoder

The first program should accept a string from `stdin`. It should then calculate the frequency of each letter and build the binary tree necessary to perform Huffman encoding. When calculating the priority of a leaf, it should be done by the frequency, ties should then be broken in increasing order of the relevant ASCII/UTF-8 code. When the frequencies have been calculated, print the priority queue characters, separated by spaces. On the next line, print the frequencies. Print them in increasing order of priority. On the next line, you should print out the encoded string. Following that, print out the number of bits used if characters are represented in 8-bit UTF-8 (compatible with ASCII). The next line should contain the number of bits used in the encoded string.

The input string will only ever contain lower case letters and a spaces. No other ASCII characters will be given.

Based on the `she sells sea shells by the sea shore` example above, here is some sample input and output:

```
Sample Input:
    she sells sea shells by the sea shore

Sample Output:
    b o r t y a h l   e s
    1 1 1 1 1 2 4 4 7 7 8
    1001000110100001101110110100011111110100100001101110110111010111001101111010100
    Total Bits (Original):296
    Total Bits (Coded):114
```

3.2 Decoder

The second program should read the first two lines of text from the first programs output. It first reads a line with all the characters in increasing order or priority. The next line has the respective frequencies for these characters. Once these lines have been read, the Huffman Tree can be reconstructed by using the `exact same` process from the encoder. Next your program should read the line containing the encoded string. Your program needs to decode the string by using the Huffman tree and output the decoded text followed by a new line. The program can ignore all other input.

```
Sample Input:
    b o r t y a h l   e s
    1 1 1 1 1 2 4 4 7 7 8
    1001000110100001101110110100011111110100100001101110110111010111001101111010100
    Total Bits (Original):296
    Total Bits (Coded):114

Sample Output:
    she sells sea shells by the sea shore
```

4 Linux Pipes

The Linux operating system provides a mechanism for programs to communicate using pipes. Imagine a pipe consists of 2 queues. One from A to B and another from B to A. We can connect `stdout` of one program to `stdin` of another by using the pipe operator (|) in the terminal. So if you have both programs coded and working correctly, you should be able to do this:

```
echo she sells sea shells by the sea shore | ./encode | ./decode
```

This calls the `echo` command, which just outputs `she sells sea shells by the sea shore` to `stdout`. We then pipe that into `stdin` of ./encode. We take the `stdout` of encode and pipe that into `decode`. If you have got everything right, decode should output the original text. When you're done with the coding part of this assignment, give this all a try!

5 Code

You are given code for the tree structures. Familiarise yourself with the code so that you understand how it works. You may use any built in data structures from the C++ STL library, particularly, you might consider using the STL's `Priority Queue` (which is a heap) and `Map` (which is a balanced binary search tree).

6 Grades & Submission

There is one submission. You should create two files: `encode.cpp` and `decode.cpp`. Your submission should be a zip file containing only these two files. They will be compiled on the marker with the following commands:

```
unzip your_submission.zip
g++ -g -std=c++11 -Wall -pedantic -Werror=vla -o encoder encode.cpp
g++ -g -std=c++11 -Wall -pedantic -Werror=vla -o decoder decode.cpp
```

Your code is graded with *neither* I/O matching nor unit tests. Another program will read your program's output and check the relevant reconstructions.

The marker will check for a valid frequency table. The order of the characters in the frequency table does not matter as long as the characters are printed on the first line (separated by spaces) and the frequencies are printed on the second line (also separated by spaces).

The system then checks that you have the correct number of "Original Bits" and "Coded Bits". After that it checks that you have the correct correct Huffman Enoding. Finally it will check that your decoder is able to reconstruct the original input, using the output from your own encoder.

Because of the way the marker is structured, **it does not matter what order you merge trees as long as you implement a valid Huffman Coding**: the left and right subtrees of any node can be swapped, and any two nodes with the same frequency valid can be swapped and the encoding will remain valid.

7 Extra Challenge - Binary File Compression

As an extra challenge, adapt your programs above so that they read bytes from a file and save the resulting encoding to a binary file. Be sure to write the resulting strings of eight 0's and 1's as bytes, not as characters otherwise each 0 or 1 will be an 8 bit character. How small can you compress different files?

It is noteworthy, that this coding scheme is used in both JPEG and ZIP, as well as a myriad of other file formats.

Try to compress this PDF file. How does the filesize compare to the original file and how does it compare to the size of the file if you zip it?