ECE368 Project 2: Huffman Coding

In this project, you will utilize your knowledge about queues and trees to design our own simple version of a file compression and decompression utility (similar to `zip` and `unzip`). You will base your utilities on the widely used algorithmic technique of Huffman coding. In case you did not know it, Huffman coding is used in JPEG compression as well as in MP3 audio compression. Let us see how the procedure of coding works.

ASCII coding

Many programming languages use ASCII (which stands for American Standard Code for Information Interchange) coding to represent characters. In ASCII coding, every character is encoded (represented) with the same number of bits (8-bits) per character. Since there are 256 different values that can be represented with 8-bits, there are potentially 256 different characters in the ASCII character set. For your reference, the ASCII character table is available at `http://www.asciitable.com/`. Let us now look at a simple example of ASCII encoding of characters. We'll look at how the string "`go go gophers`" is encoded in ASCII. Using ASCII encoding (8 bits per character) the 13-character string "`go go gophers`" requires 13 * 8 = 104 bits. The table below shows how the coding works.

Character	ASCII code	8-bit binary value
g	103	1100111
o	111	1101111
p	112	1110000
h	104	1101000
e	101	1100101
r	114	1110010
s	115	1110011
Space	32	1000000

The given string could be written (coded numerically) as `103 111 32 103 111 32 103 111 112 104 101 114 115`. Although not easily readable by humans, this would be written as the following stream of bits (the spaces would not be written, just the 0's and 1's)

1100111 1101111 1100000 1100111 1101111 1000000 1100111 1101111 1110000 1101000 1100101 1110010 1110011

From ASCII coding towards Huffman coding

Next, let us see how we might save bits using a simpler coding scheme. Since there are only 8 different characters in "`go go gophers`", it is possible to use only 3-bits to encode the 8 different characters. We might, for example, use the coding shown in the table below (keep in mind that other 3-bit encodings are also possible).

Character	Code Value	3-bit binary value
g	0	000

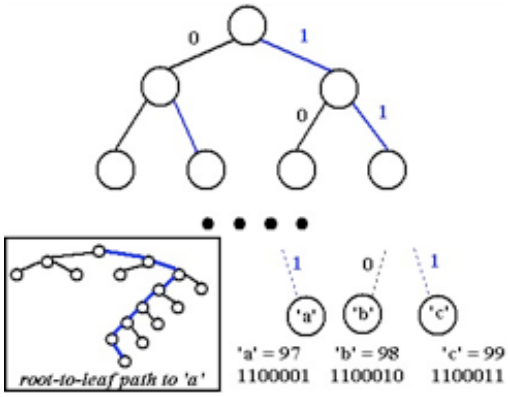
o	1	001
p	2	010
h	3	011
e	4	100
r	5	101
s	6	110
Space	7	111

Now the string "`go go gophers`" would be encoded as `0 1 7 0 1 7 0 1 2 3 4 5 6` or, as bits: `000 001 111 000 001 111 000 001 010 011 100 101 110 111`. As you can see, by using three bits per character instead of eight bits per character that ASCII uses, the string "`go go gophers`" uses a total of 39 bits instead of 104 bits.

However, even in this improved coding scheme, we used the same number of bits to represent each character, irrespective of how often the character appears in our string. Even more bits can be saved if we use fewer than three bits to encode characters like `g`, `o`, and `space` that occur frequently and more than three bits to encode characters like `e, p, h, r,` and `s` that occur less frequently in "`go go gophers`". *This is the basic idea behind Huffman coding: to use fewer bits for more frequently occurring characters*. We'll see how this is done using a tree data structure that stores the characters as its leaf nodes, and whose root-to-leaf paths provide the bit sequence used to encode the characters.

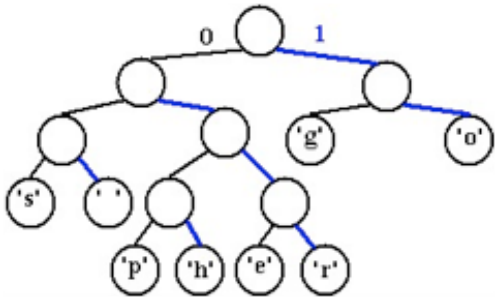
Towards a Coding Tree

Using a binary tree for coding, all characters are stored at the leaves of a tree. In the diagram below, the tree (which happens to be a complete binary tree) has eight levels meaning that the root-to-leaf path always has seven edges. A left-edge (black in the diagram) is numbered `0` and a right-edge (blue in the diagram) is numbered `1`. The ASCII code for any character/leaf is obtained by following the root-to-leaf path and concatenating the `0's` and `1's`. For example, the character '`a`', which has ASCII value `97` (`1100001` in binary), is shown with root-to-leaf path of *right-right-left-left-left-left-right*.



Clearly, the specific structure of the tree determines the coding of any leaf node using the 0/1 edge convention described. If we use a different tree, we get a different coding. As an example, the tree below on the right yields the coding shown on the left.

Character	Binary code
'g'	10
'o'	11
'p'	0100
'h'	0101
'e'	0110
'r'	0111
's'	000
' '	001



Using this coding, "`go go gophers`" is encoded (spaces wouldn't appear in the bit-stream) as: `10 11 001 10 11 001 10 11 0100 0101 0110 0111 000.` This is a total of 37 bits, which saves two bits over the improved encoding in which each of the 8 characters has a 3-bit encoding! The bits are saved by coding frequently occurring characters like '`g`' and '`o`' with fewer bits (here two bits) than characters that occur less frequently like '`p`', '`h`', '`e`', and '`r`'.

The same character encoding induced by the tree can be used to decode a stream of bits. For example, you can try to decode the following bit-stream using the above tree; the answer with an explanation follows:

```
01010110011100100001000101011001110110001101101100000010101011001110110
```

To decode the stream, start at the root of the tree, and follow a left-branch if the next bit in the stream is a 0, and a right branch if the next bit in the stream is a 1. When you reach a leaf, write the character stored at the leaf, and start again at the top of the tree. To start, the bits are `010101100111`. This yields *left-right-left-right* to the letter '`h`', followed (starting again at the root) with *left-right-right-left* to the letter '`e`', followed by *left-right-right-right* to the letter '`r`'. Continuing thus yields a decoded string "`her sphere goes here`".

Prefix codes

When all characters are stored in leaves, and every interior (non-leaf) node has two children, the coding induced by the 0/1 convention outlined above satisfies a very important property called the *prefix property* which states that no bit-sequence encoding of a character is the prefix of the bit-sequence encoding of any other character. This makes it possible to decode a bitstream using the coding tree by following root-to-leaf paths. The tree shown above for "`go go gophers`" satisfies this prefix property and is an optimal tree. That means there are no other trees with the same characters that result in fewer total bits to encode the string "`go go gophers`". There are other trees that use 37 bits; for example you can simply swap any sibling nodes and get a different encoding that uses the same number of bits. Next, we look at an algorithm for constructing such an optimal tree. This algorithm is called Huffman coding, and was invented by David A. Huffman in 1952 when he was a Ph.D. student at MIT.

Huffman Coding

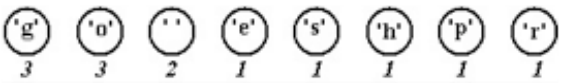
In the previous section we saw examples of how a stream of bits can be generated from an encoding. We also saw

how the tree can be used to decode a stream of bits. We'll discuss how to construct the tree here using Huffman's algorithm.

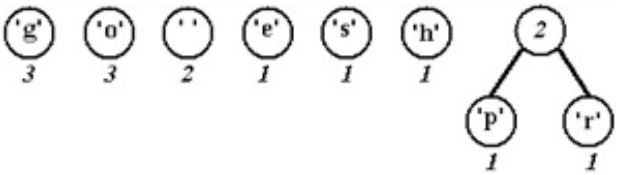
We'll assume that each character has a weight associated with it that is equal to the number of times the character occurs in a file. For example, in the string "`go go gophers`, the characters '`g`' and '`o`' have weight 3, the space has weight 2, and the other characters have weight 1. When compressing a file we'll need to first read the file and calculate these weights. Assume that all the character weights have been calculated. Huffman's algorithm assumes that we're building a single tree from a group (or forest) of trees. Initially, all the trees have a single node containing a character and the character's weight. Trees are combined by picking two trees and making a new tree from the two trees. This decreases the number of trees by one at each step since two trees are combined into one tree. The algorithm is as follows:

1. Begin with a forest of trees. All trees have just one node, with the weight of the tree equal to the weight of the character in the node. Characters that occur most frequently have the highest weights. Characters that occur least frequently have the smallest weights.
2. Repeat this step until there is only one tree:
   Choose two trees with the smallest weights; call these trees T_1 and T_2. Create a new tree whose root has a weight equal to the sum of the weights $T_1 + T_2$ and whose left sub-tree is T_1 and whose right sub-tree is T_2.
3. The single tree left after the previous step is an optimal encoding tree.

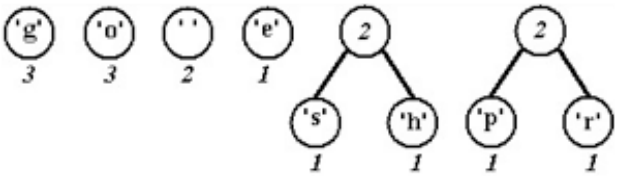
We'll use the string "`go go gophers`" as an example. Initially we have the forest shown below. The nodes are shown with a weight that represents the number of times the node's character occurs in the given input string/file.



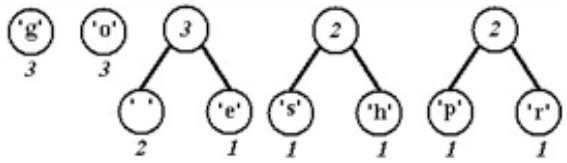
We pick two minimal nodes. There are five nodes with the minimal weight of one, it doesn't matter which two we pick. In a program, the deterministic aspects of the program will dictate which two are chosen, e.g., the first two in an array, or the elements returned by a priority queue implementation. We create a new tree whose root is weighted by the sum of the weights chosen. We now have a forest of seven trees as shown here:



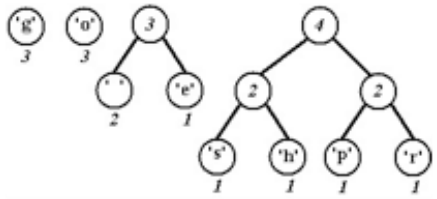
Choosing two minimal trees yields another tree with weight two as shown below. There are now six trees in the forest of trees that will eventually build an encoding tree.



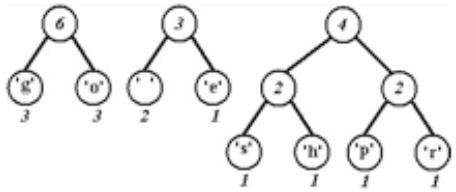
Again we must choose the two trees of minimal weight. The lowest weight is the 'e'-node/tree with weight equal to one. There are three trees with weight two, we can choose any of these to create a new tree whose weight will be three.



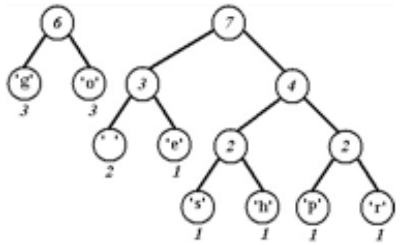
Now there are two trees with weight equal to two. These are joined into a new tree whose weight is four. There are four trees left, one whose weight is four and three with a weight of three.



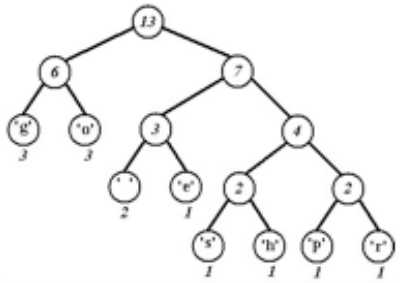
Two minimal (three weight) trees are joined into a tree whose weight is six. In the diagram below we choose the 'g' and 'o' trees (we could have just as well chosen the 'g' tree and the space-'e' tree or the 'o' tree and the space-'e' tree.) There are three trees left.



The minimal trees have weights of three and four; these are joined into a tree with weight seven.



Finally, the last two trees are joined into a final tree whose weight is thirteen, the sum of the two weights six and seven. Note that this tree is different from the tree we used to illustrate Huffman coding above, and the bit patterns for each character are different, but the total number of bits used to encode "`go go gophers`" is the same.



The character encoding induced by the above tree is shown below where again, 0 is used for left edges and 1 for right edges.

Character	Binary value
'g'	00

'o'	01
'p'	1110
'h'	1101
'e'	101
'r'	1111
's'	1100
' '	100

The string "`go go gophers`" would be encoded as (with spaces used for easier reading, the spaces wouldn't appear in the real encoding): `00 01 100 00 01 100 00 01 1110 1101 101 1111 1100`. Once again, 37 bits are used to encode "`go go gophers`". There are several trees that yield an optimal 37-bit encoding of "`go go gophers`". The tree that actually results from a programmed implementation of Huffman's algorithm will be the same each time the program is run for the same weights (assuming no randomness is used in creating the tree).

Implementing/Programming Huffman Coding

In this section we'll see the basic programming steps in implementing Huffman coding. There are two parts to an implementation: a compression program and a decompression program. You need both to have a useful compression utility. We'll assume these are separate programs, but they actually have many common functions. We'll call the program that reads a regular file and produces a compressed file the *compression* or *huffing* program. The program that does the reverse, producing a regular file from a compressed file, will be called the *decompression* or *unhuffing* program.

The Compression or Huffing Program

To compress a file (sequence of characters) you need a table of bit encodings, i.e., a table giving a sequence of bits used to encode each character. This table is constructed from a coding tree using root-to-leaf paths to generate the bit sequence that encodes each character.

Assuming you can write a specific number of bits at a time to a file, a compressed file is made using the following top-level steps. These steps will be developed further into sub-steps, and you'll eventually implement a program based on these ideas and sub-steps.

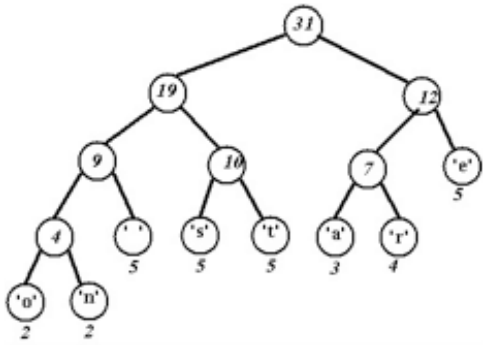
1. Build a table of per-character encodings. The table may be given to you, e.g., an ASCII table, or you may build the table from a Huffman coding tree.
2. Read the file to be compressed (the plain file) and process one character at a time. To process each character find the bit sequence that encodes the character using the table built in the previous step and write this bit sequence to the compressed file.

As an example, we'll use the table below on the left, which is generated from the tree on the right. For now, ignore the weights on the nodes; we'll use those when we discuss how the tree is created.

Another Huffman Tree/Table Example

Character	Binary value
'a'	100

'r'	101
'e'	11
'n'	0001
't'	011
's'	010
'o'	0000
' '	001



To compress the string/file "`streets are stone stars are not`", we read one character at a time and write the sequence of bits that encodes each character. To encode "streets are" we would write the following bits: `010011101111101101000110010111.` The bits would be written in the order *010*, *011*, *101*, *11*, *11*, *011*, *010*, *001*, *100*, *101*, *11*. That's the compression program. Two things are missing from the compressed file: (1) some information (called the header) must be written at the beginning of the compressed file that will allow it to be decompressed; (2) some information must be written at the end of the file that will be used by the decompression program to tell when the compressed bit sequence is over (this is the bit sequence for the pseudo-eof character described later).

Building the Table for Compression/Huffing

To build a table of optimal per-character bit sequences you'll need to build a Huffman coding tree using the Huffman algorithm described earlier. The table is generated by following every root-to-leaf path of the Huffman tree and recording the 0/1 edges followed. These paths represent the optimal encoding bit sequences for each character.

There are three steps in creating the table:

1. Count the number of times every character occurs. Use these counts to create an initial forest of one-node trees. Each node has a character and a weight equal to the number of times the character occurs.
2. Use the greedy Huffman algorithm to build a single tree. The final tree will be used in the next step.
3. Follow every root-to-leaf path of the tree creating a table of bit sequence encodings for every character/leaf.

Header Information

You must store some initial information in the compressed file that will be used by the decompression/unhuffing program. Basically, you must store the tree that is used to compress the original file. This is because the decompression program needs this exact same tree in order to decode the data. There are several alternatives for storing the tree. Two of these are outlined here:

- Store the character counts at the beginning of the file. You can store counts for every character, or counts for only the non-zero characters. If you do the latter, you must include some method for indicating the character, e.g., store character/count pairs.

- You can store the tree at the beginning of the file. One method for doing this is to do a pre-order traversal, writing each node visited. You must differentiate leaf nodes from internal/non-leaf nodes. One way to do this is to write a single bit for each node, say 1 for leaf and 0 for non-leaf. For leaf nodes, you will also need to

write the character stored. For non-leaf nodes there's no information that needs to be written, just the bit that indicates there's an internal node.

The pseudo-EOF character

Operating systems typically require that disk files have sizes that are multiples of some architecture/operating system specific unit, e.g., a byte or word. On many systems all file sizes are multiples of 8 or 16 bits so that it isn't possible to have a 122-bit file. So, if all file output is done in eight-bit chunks and your program writes exactly 61 bits explicitly, then 3 extra bits will be written so that the number of bits written is a multiple of eight. Your decompressing/unhuff program must have some mechanism to account for these extra or "padding" bits since these bits do not represent compressed information. Your decompression/unhuff program cannot simply read bits until there are no more left since your program might then read the extra padding bits written due to buffering. This means that when reading a compressed file, you **CANNOT** use code like this (this assumes that the function `readbits(x)` reads `x` bits from the file).

```
int bits;
while ((bits = readbits(1)) != -1)
{
    // process bits
}
```

To avoid this problem, you can use a pseudo-EOF character and write a loop that stops when the pseudo-EOF character is read in (in compressed form). The code below illustrates how reading a compressed file works using a pseudo-EOF character:

```
int bits;
while (true)
{
    if ((bits = readbits(1)) == -1)
    {
        printf("Should not happen! trouble reading bits");
    }
    else
    {
        // use the zero/one value of the bit read
        // to traverse Huffman coding tree
        // if a leaf is reached, decode the character and print UNLESS
        // the character is pseudo-EOF, then decompression done

        if ( (bits & 1) == 0) // read a 0, go left  in tree
        else                  // read a 1, go right in tree

        if (at leaf-node in tree)
        {
            if (leaf-node stores pseudo-eof char)
                break;   // out of loop
            else
                write character stored in leaf-node
        }
    }
}
```

When a compressed file is written the last bits written should be the bits that correspond to the pseudo-EOF character. You will have to write these bits explicitly. These bits will be recognized by the unhuff program and used in

the decompressio. This means that your decompression program will never actually run out of bits if it's processing a properly compressed file (you may need to think about this to really believe it). In other words, when decompressing you will read bits, traverse a tree, and eventually find a leaf-node representing some character. When the pseudo-EOF leaf is found, the program can terminate because all decompression is done. If reading a bit fails because there are no more bits (the bit reading function returns false) the compressed file is not well formed. It is obvious from the above that every time a file is compressed the count of the number of times the pseudo-EOF character occurs should be one --- this should be done explicitly in the code that determines frequency counts. In other words, a pseudo-char EOF with number of occurrences (count) of 1 must be explicitly created and used in creating the tree used for compression.

WHAT YOU NEED TO DO?

Your task is to write two programs, one titled "`huff.c`" and the other titled "`unhuff.c`". The program huff.c should accept one argument, which will be the name of an input file. The program should compress the input file using Huffman coding and write out the compressed output to a file that has the same name as the input file with a ".huff" appended to it. For example, if "example.txt" is the name of an input file, the output file should be "example.txt.huff". The program unhuff.c should accept one argument, which will be the name of an input file. The program should de-compress the input file and write out the de-compressed output to a file that has the same name as the input file with an ".unhuff" appended to it. For example, if "example.txt.huff" is the name of an input file, the output file should be "example.txt.huff.unhuff".

WHAT CAN YOU ASSUME?

You can assume that the input file to huff.c will be a pure ASCII text file. Note that this assumption is not strictly required, but is only to make your job of writing the program a little easier.

WHAT YOU NEED TO SUBMIT?

1. You will need to submit all your source code (all the ".c" and ".h" files). Code should be well commented and easy to understand.
2. You will need to submit a 2-page report in PDF format that describes your solution and results. If you need any special flags included for compilation, mention that in your report.

HOW TO SUBMIT?

1. Create a directory named "ece368-project2". Store all your source code and project report in this directory.
2. Make sure you remove any executables, core dump files, etc. At the time of submission, the directory should only contain ".c", ".h", and "report.pdf" files.
3. On shay, cd to the parent of this "ece368-project2" directory.
4. Type: **`turnin -c ee368 -p proj2 ece368-project2/`**
5. You can check if your files have been submitted by typing **`turnin -v -c ee368 -p proj2 ece368-project2/`**

HOW WILL WE GRADE YOUR SUBMISSION?

We will test your huff.c and unhuff.c by compiling them with the following command:

```
gcc -Werror -Wall -O3 huff.c -o huff
```

(If you need to use additional libraries and hence compiler flags, mention that in your report, and we will compile your code with the specified compiler directives added to the compilation instruction above.)

We will then run your compression program on several input files of varying sizes, and then decompress these compressed files. 60 points of your grade will depend on whether the final decompressed file is exactly the same as the original file.

We will also compute two performance metrics for your code: (a) the compression ratio (this is just the size of your original file divided by the size of the compressed file), and (b) the time taken for your huff.c and unhuff.c to execute. 20 points of your grade for this project will depend on how well you do in terms of the two metrics. This part of the grading will be normalized (top 25% of the submissions get 20 points, the next 25% get 15 points, and so on).

The final 20 points of your grade will depend on the quality of your report.