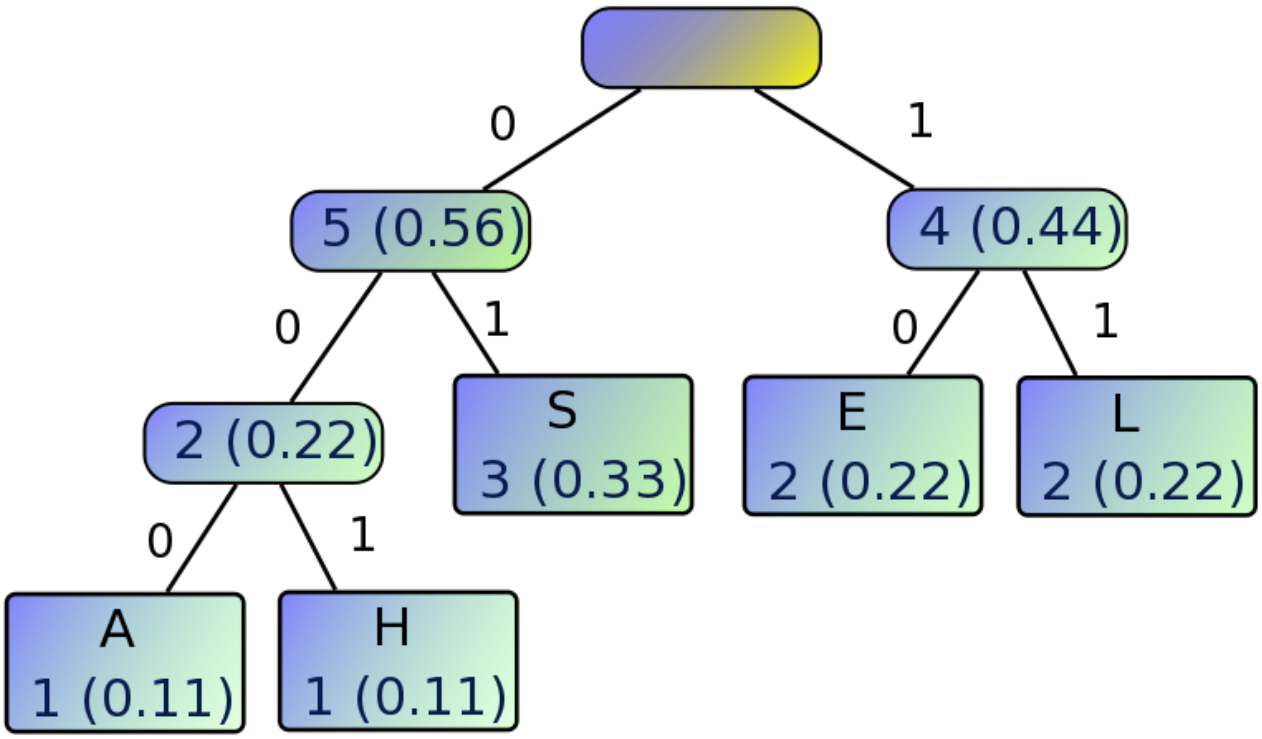
CSC 192 Assignment 5

Due: Dec. 4th @ 11:59 PM

Question1)
Mark 1:
Suppose that the following declarations are in effect:

```
struct point { int x, y; };
struct rectangle { struct point upper_left , lower_right; };
struct rectangle *p;
```

Assume that we want p to point to a rectangle structure, whose upper left corner is at (10,25) and whose lower right corner is at (20,15). Write a series of statements (in a .txt file) that allocate such a structure and initialize it as indicated.



Question2)
Mark 9:
In this question, your program will compress data using a technique called *Huffman coding*. Your program will build Huffman trees as seen above, uses these trees to assign codes to each character, and then use these codes to compress some text.

Huffman Coding

Huffman coding is a method of assigning codes to each symbol in a document based on the symbol's frequency (how many times it appears in the document). The more frequent the symbol, the lesser the length of its code. The main idea is to assign a code for each character that requires less than the 8 bits needed by the ASCII encoding. In this assignment, the documents will be plain ASCII text and the symbols will be ASCII characters but in general the symbols could be part of any alphabet whose size is $|A|$.

The algorithm works in several steps:

1. Read through the input data, tabulating the frequency (number of occurrences), c_f, of each character in the input, and counting the total number of characters, c_{total}. For each character, calculate the *probability* of the character as

$$c_p = \frac{c_f}{c_{total}}$$

For example, if the input were the string `hello world` without a newline character, then the frequency of `l` would be 3, and its probability would be $\frac{3}{11}$ whereas the frequency of `h` would be 1 and its probability would be $\frac{1}{11}$.

2. For every unique character in the document, construct a Huffman Tree node and insert it into a heap-based priority queue. The Huffman Tree node contains a unique character from the input and the probability of the character (c_p). Insert the node into the priority queue with priority $1 - c_p$. In other words, the less (likely/frequent) the character is, the higher priority it has in the queue[1]. The nodes added here will form the leaf nodes of the Huffman tree.

3. Construct the Huffman Tree. While there are two or more nodes in the priority queue, repeat the following:
   (a) Remove the two nodes with highest priorities (lowest probabilities) from the priority queue: denote these n_1 and n_2, with character probabilities c_{p_1} and c_{p_2}.
   (b) Create an internal Huffman tree node n' with probability equal to the sum of the probabilities of each of these two nodes, $c'_p = c_{p_1} + c_{p_2}$. Modify your data structures to represent that n_1 and n_2 are children of n'.
   (c) Insert n' into the priority queue with priority $1 - c'_p$.

4. The last node to remove from the priority queue is the root of the Huffman tree.

5. Assign the codes to each character. The code is a bit string of variable length which depends on the character's location in the tree. First note that the Huffman Tree is a binary tree, so each internal node has 2 children. The code is obtained by the path taken to get to the leaf corresponding to the character it is representing: for instance, if the path is: { follow the left child, follow the left child, follow the right child, follow the left child } then its code would be `0010`. Starting with an empty code: every time a left child is followed, concatenate `0` to the code and every time the right child is followed, concatenate `1` to the code. Look at the diagram on the first page to see an example of the codes assigned to each letter (*e.g.* the code for `H` is `001` and the code for E is `10`).

6. Pass through the input document again. For each character, in sequence: print its Huffman Code.

The output produced is a Huffman-compressed version of the input. For example, the Huffman Tree depicted on the front page of the assignment could be one constructed from the input `SEASHELLS` (without a newline). The Huffman-compressed phrase is just the codes for each character in the input, in sequence:

`01100000100110111101`

The size of this compressed text is 20 bits as opposed to $9 \cdot 8 = 72$ bits using ASCII encoding, resulting in a 72.222% compression rate. The Huffman Tree is needed to uncompress the data, so if this type of compression were used in practice the Huffman Tree and other meta-data would also have to be encoded along with the compressed output to form a fully compressed file. For this assignment, your program only prints out the Huffman-compressed bit sequence as seen above. *Note that due to ties in character frequencies and the arbitrary order that children nodes assigned to internal nodes, the Huffman Tree for a particular input is not necessarily unique.*

The input will be read using file I/O. The input file name will be sent as the only command-line argument to your program. If your executable is named `huffman` then your program will be run like so:

```
./huffman file.txt
```

In C the way to access a command-line argument is by defining main this way:

```c
int main(int argc, char * argv[]) {
  // argc is the size of the argv array
  // argv[0] is the command that was executed
  // argv[1] is the first command-line argument
  // argv[2] is the second command-line argument
  return 0;
}
```

[1]Note: in practice, one could use simply use the $c_{total} - c_f$ as the priority to avoid problems with numerical precision. Later steps would have to be modified accordingly.

The input file will contain valid ASCII characters (in the range $\{0, 1, \cdots, 127\}$), *i.e.* they will be arbitrary text files, possibly containing special characters. Your program will never be tested on invalid input and you can assume that the input will always have length $n \geq 1$ and the number of unique characters $k \geq 2$. You can be certain that the input received will be much larger than simple phrases, so you should test your solution on larger test files. The output, sent to the standard output stream, should be a sequence of bits using characters `1` and `0` as shown in the example in the previous paragraph (in particular, *do not* pack the bits back into bytes!). You may include any amount of whitespace in your output to help visualize the data as it will all be ignored by the verifier. However, any output other than characters `1`, `0`, tabs, spaces, or newlines will be considered incorrect.

Efficiency

It is important that your solution be efficient as outlined here in order to recieve full credit

Important note: the efficiency requirements below assume that k is unbounded, *e.g.* that the algorithm must handle a maximum number of unique symbols which is not known in advance. This will not be true in practice *for this assignment* since your program will be run on ASCII input (therefore the number of unique characters $k \leq 127$) but your program should be easy to extend to the general case of an unbounded alphabet size and the use of a hash table for storing the codes without affecting the asymptotic time complexity of the algorithm.

Following these efficiency requirements means implementing a heap-based priority queue and a Huffman Tree data structure and then implementing the algorithm above which use these data structures.

```
(Note: A tree is heap-ordered if the key in each node is larger
than or equal to the keys in all of that node's children(if any).
You did this in Assignment3)
```

Assuming that the size of the input (number of characters) is n and the number of unique characters is $k \leq n$:

- Note: an array of size $|A| = 128$ is acceptable for this assignment. Since the alphabet size is bounded by a constant (127) then for this assignment the space required is constant. In the general case, this array could be very easily replaced by a hash table or other map-based data structure depending on whether space savings or time savings is more important.
- Step 1 of the algorithm should run in $O(n)$ time.
- Step 2 of the algorithm should run in $O(k \log k)$ time.
- Step 3 of the algorithm should run in $O(k \log k)$ time.
- Step 4 of the algorithm should run in $O(1)$ time.
- Step 5 of the algorithm should run in $O(k)$ time.
- Step 6 of the algorithm should run in $O(n)$ time.

Design and Advice

You are free to design your data structures however you wish *as long as your solution respects the efficiency requirements detailed above*. Justify all of your design choices in your documentation.

Design and test your data structures before attempting to implement the Huffman compression algorithm. Practice good programming style by structuring and organizing your code appropriately.
It will help if you think of your heap based priority queue having the following minimum functions:

```
void initializePriorityQueue(int sizeOfQueue);

//after this function is called, your heap based priority queue condition must be satisfied.
//see above note for the condition
void insertInPriorityQueue(int v);

//after this function is called, your heap based priority queue condition must be satisfied.
//see above note for the condition
int getMaximumPriorityItem();
```

Optional: Huffman Compression Quality

Just how good is Huffman compression? Suppose the original number of bits required to represent the input was b_{org} and the number of bits in the compressed output is b_{comp} then the *compression rate* is:

$$r = \frac{b_{org} - b_{comp}}{b_{org}}$$

You can compute these values in case you are curious about how well this compression technique does in practice, but it is not required for the assignment. Remember that one ASCII character takes up 8 bits.

Calculate the value of r for an assortment of random text files. Do you notice a relationship between n, k, and r?

What about using Huffman codes for substrings (words) rather than characters? Would that work better or worse in general? What kind of documents would it work better on and what kind of documents would it do worse on? Would it be harder or easier to implement than character-based Huffman compression?

For more information on the properties of Huffman encoding, see `http://en.wikipedia.org/wiki/Huffman_coding`.

```
 What to submit for this assignment?

1) For question1, q1.txt

2) For question2:
-- huffman.c
-- your test cases (input files)
-- and a 1 page document (pdf) that explains how your code satisfies the
   requirements for efficiency.
```