CMPT 225 Assignment 4: Huffman Compression

Preamble

- Due Wednesday, March 20 at 3pm.
- Please, read the entire assignment first before starting it! And let's start our assignment as early as possible!
- This assignment may be done in pairs. The pair may contain students only from the D1 section.

Objectives

In Assignment 4, our objectives are:

- To practice going through the steps of the software development process.
- To design and implement a software solution to a given problem and to do so by following an algorithm and by satisfying a set of requirements.
- To design and implement various ADT classes.
- To practice manipulating input/output files (text and binary) and streams in C++.

Problem Statement

Many of you may have used WinZip, tar, gzip or some other file compression utility to compress files so that they can be transferred via e-mail, or ftp'ed more efficiently. There are many different ways of performing file compression.

In this assignment, you will build a compression/decompression application that uses Huffman's Algorithm, which was first published in 1952.

Compression/Decompression Scheme with Prefix Codes

Suppose you have a text file containing only the phrase:

```
that's the way it is
```

When stored as a regular text file, it takes a total of 20 bytes, one byte for each character. Doing a frequency analysis shows that 't' and the ASCII space are the most frequent:

Character	Frequency	Character	Frequency
t	4	s	2
<space>	4	'	1
a	2	e	1
h	2	w	1
i	2	y	1

Table 1 - Frequency Table

A full binary tree is constructed with characters at the leaves. The tree is organized so the most frequent characters appear on the upper levels of the tree while the least frequent appear on the lower levels. Huffman's Algorithm for constructing an optimal tree is presented later.

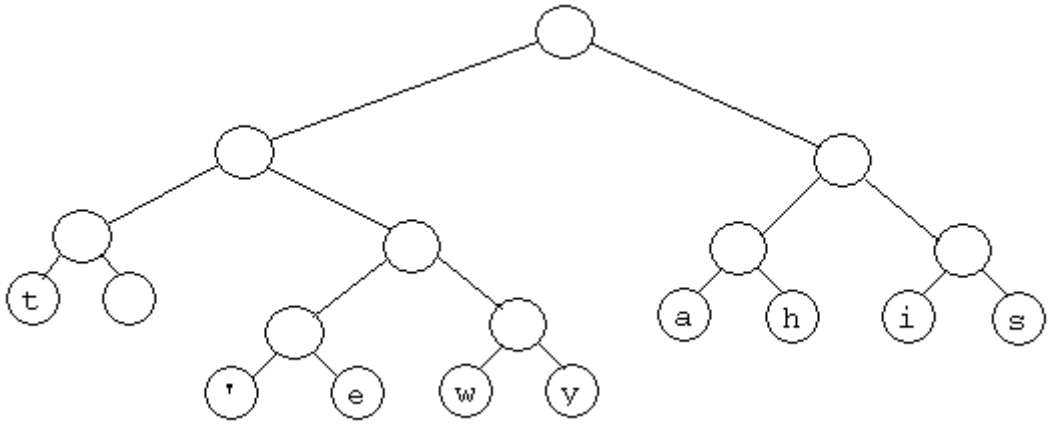

Figure 1 - Huffman Tree

The tree can be viewed as a **decision tree**. Follow the path from the root to each leaf: 0 means go left; 1 means go right. For example, to derive the a special code for 'h', start at the root node, go to the right child (`1`), then the left child (`10`) and finally to the right child (`101`).

Thus you can derive the special code for each character of the input:

Character	Code	Character	Code
t	000	s	111
<space>	001	'	0100
a	100	e	0101
h	101	w	0110
i	110	y	0111

Table 2 - Coding Table

This is a **Huffman code**. It has the special property that no Huffman code is a prefix of another Huffman code. For example, the Huffman code for the character 'a' (`100`) never occurs as the first three digits of any other Huffman code in the table. This is a consequence of having all the characters at the leaf nodes.

The original phrase can now be written as a sequence of binary:

which would occupy 64 bits or 8 bytes - quite a savings over the original 20 bytes! The savings comes about because the codes used to represent the characters are shorter than the standard 8-bit ASCII code. However, the compressed file will end up being larger than 8 bytes because the Huffman tree (or some means of reproducing it) must also be saved with the file. For small files like this 20-byte example, the size of what must be saved in order to reproduce the Huffman codes can be quite significant when compared to the size of the data!

How would you decode a string such as the one above? Read the string one bit at a time, tracing down the Huffman tree until you reach a leaf node. Remember: a `0` indicates a move to the left child and a `1` indicates a move to the right child. Once you reach a leaf node you have found your first character. Continue with the next bit in the string from the root node of the tree, and trace down again until you reach a leaf node - this node contains your next character. And so on.

Huffman's Algorithm

Now to the process of building the Huffman tree.

Set Up

Start by counting the frequency of each character of the input file. For each character in the file, build a tree having exactly one node containing that character. Assign to each tree a *weight* equal to its frequency. You will have *n* trees.

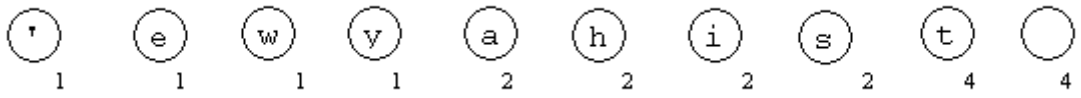

Figure 2 - Set Up for Huffman's Algorithm

Building the Tree

At each step, pick the two trees that have the lowest weights and join them into one, by making their roots siblings of a new larger tree. The weight of the new tree equals the sum of the weights of its children.

In this case four of the trees have a weight of 1 - we will arbitrarily work through the above list of trees from left to right when more than two trees have the same weight:

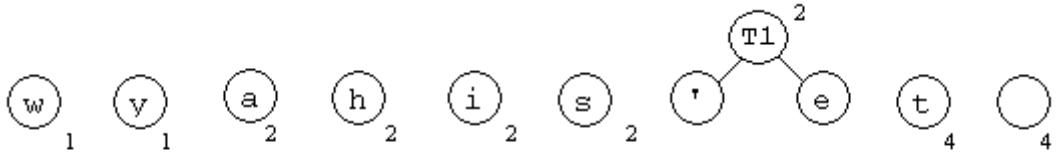

Figure 3a - Huffman's Algorithm after the 1st join

The next three steps consist of building a tree with 'w' and 'y' as its children, another with 'a' and 'h' as its children and a third with 'i' and 's' as its children:

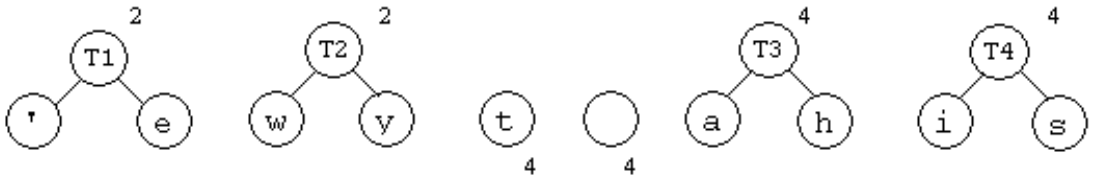

Figure 3b - Huffman's Algorithm after the 4th join

There are four trees of weight 4 and two trees of weight 2. In the next step, the latter two trees are joined to create a tree of weight 4.

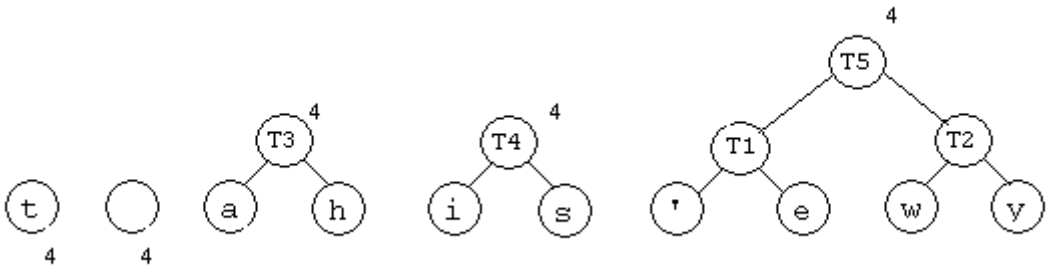

Figure 3c - Huffman's Algorithm after the 5th join

All trees now have weight 4. The next two steps consist of building a tree with t and <space> as its children and another with T3 and T4 as its children:

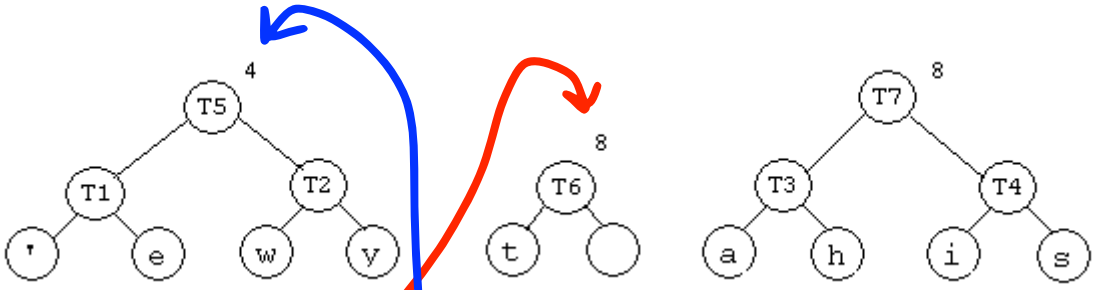

Figure 3d - Huffman's Algorithm after the 7th join

Now combine T6 and T5. We arbitrarily make the choice to place the heavier subtree on the left:

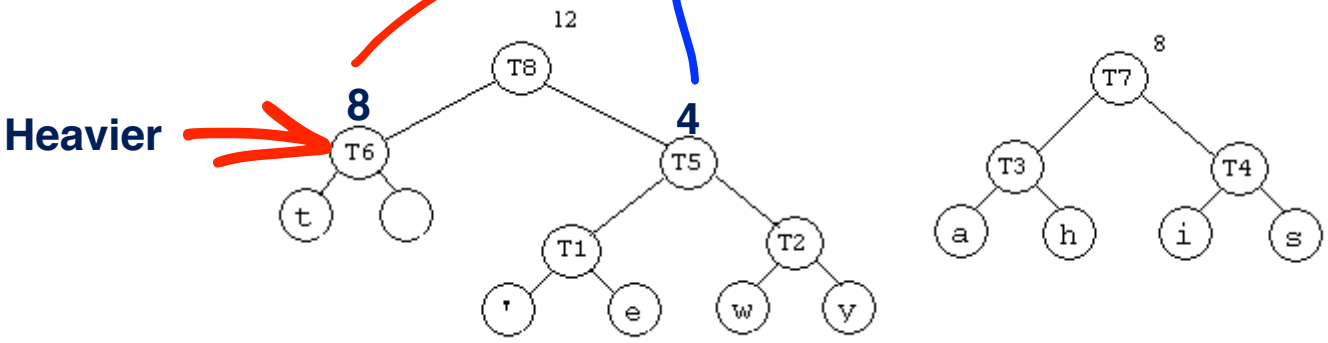

Figure 3e - Huffman's Algorithm after the 8th join

And finally combine trees T8 and T7, once again placing the heavier subtree on the left, to produce the following tree:

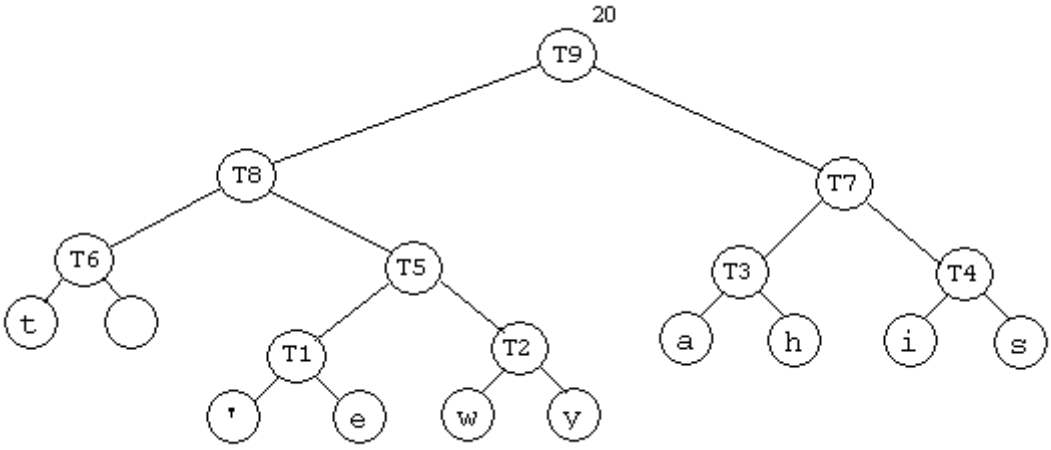

Figure 3f - Huffman's Algorithm after the last join

Requirements

- This assignment is a little different than the others. For starters, there is no base code and no makefile. You have an empty palette to start with, and your design skills will play a bigger role than usual. You are, however, expected to develop your solution to this problem statement using object oriented design and programming which means, amongst other things, that you cannot use C++ strut mechanism.
- Your implementation may ***not*** make use of the STL container classes and algorithms.
- Your compression/decompression application will make use of the Huffman compression/decompression scheme described above.
- Your application will supply a command line interface as follows:

Compressing a file

To compress a file, the user must enter:

```
uname@hostname:~$./huff -c source.txt compressedSource.huff
```

Your application will:

- Warn and terminate if the user types an invalid command.
- Open the source file (e.g., "source.txt") for reading.
- Create and open a destination file (e.g., "compressedSource.huff") for writing.
- Build the frequency table using the source file.
- Build the Huffman tree based on the frequency table.
- Write enough information to the destination file so that you will be able to decompress its content.
- Keep track of number of bytes written to the destination file.
- Write the compressed version of the source file to the destination file. For each character x in the source file:
  - determine x's Huffman code; and
  - write this Huffman code to the destination file.

- Display the size of the source file (file to be compressed) and the size of the destination file (compressed file) in bytes on the computer monitor screen as follows (the figures are fictitious):

```
uname@hostname:~$./huff -c source.txt compressedSource.huff
source.txt -> 7839 bytes
compressedSource.huff -> 453 bytes
```

- Issue a warning if the size of the compressed file > the size of the source file (file to be compressed) as follows (the figures are fictitious):

```
uname@hostname:~$./huff -c source.txt compressedSource.huff
source.txt -> 39 bytes
compressedSource.huff -> 453 bytes
*** Size of compressed file > size of source file ***
```

- Close the source and destination files.

Decompressing a file

To decompress a file, the user must enter:

```
uname@hostname:~$./huff -d compressedSource.huff sourceCopy.txt
```

Your application will:

- Warn and terminate if the user types an invalid command.
- Open the compressed source file (e.g., "compressedSource.huff") for reading.
- Create and open the destination file (e.g., "sourceCopy.txt") for writing.
- Read information that was written to the compressed source file and reconstruct whatever object(s) you will need in order to decompress the content of the compressed source file.
- Decompress the content of the compressed source file and write it to the destination file. For each Huffman code in the source file:
  - get its corresponding character from the Huffman tree; and
  - write the character to the destination file.
- Close the source and destination files.

- Even though you must follow the above algorithm, you are free to specify which information you must save into the compressed file in order to figure out which ASCII character each of the Huffman code represent.

- You are also free to design and implement the classes you'll need to complete your Huffman compression/decompression application. Some suggested classes are given below.

Suggested Classes

- BitStream class or (InBitStream and OutBitStream classes) - provides operations necessary to read a bit from a file and to write a bit or sequence of bits to a file.

  When you compress a file, you read each character in the file, determine its Huffman code and write this code to the compressed file. With reference to the example presented in the Huffman Compression/Decompression Scheme section above, if the first character in the file to be compressed is a 't', then you want to write the following sequence of bits to the compressed file: 000. Similarly when decompressing a file, you want to read successive bits from the compressed file (e.g., 000) and determine the corresponding character, (e.g., 't').

  In C++, there is no direct mechanism for reading and writing single bits. Therefore create two new objects: one to write bits to an output file stream and one to read bits from an input file stream. This class should encapsulate the operations necessary to read or write a bit or sequence of bits to/from a file stream.

- FrequencyCounter Class - to build and manipulate a frequency table that stores the frequency with which a char appears in a file.

  In order to build the Huffman tree, you need to determine the frequency with which a char (8 bits) appears in the file to be compressed. There are a total of 256 possibilities.

- PriorityQueue class - to help build the Huffman tree.

  When you build the Huffman tree, you repeatedly choose the two subtrees having the smallest weight and combine them to create a new tree. A priority queue of subtrees keyed by weight might be useful for this purpose.

  Note: It is possible to implement Huffman's algorithm using a pair of queues. If you feel this may be a simpler approach, you can read about its implementation.

- Huffman tree class - provides operations to build/rebuild the Huffman tree, to compute the Huffman code of a given character, and to obtain the character given a Huffman code.

  This class is responsible for ensuring that the appropriate information (needed to decompress a compressed file) is written to the compressed file (when compressing). It is also responsible for gathering information necessary to decompress a file from the compressed file (when decompressing).

  Think carefully about what kind of information you will need in order to decompress a compressed file. Will you need information in order to rebuild the huffman tree? To reconstruct the coding table? Whatever information you select, it must requires a minimum amount of space. There is little point coming up with an elaborate compression scheme if the additional information written to the file (so that it can be decompressed) takes up an inordinate amount of space! The simplest approach is to write each character found in the original file along with its frequency count, i.e., the frequency table. This is enough information to allow you to re-construct the Huffman tree and determine the Huffman code for each character. This is not necessarily the most efficient approach - feel free to investigate alternatives!

  This class is also responsible for determining the Huffman code given a character and vice versa. When a file is compressed, you write the Huffman codes to the compressed file using a BitStream object. When you decompress a file, you have to know when to stop reading bits from the file. One approach is to write the number of characters in the original file to the compressed file.

Reading/Writing bits from/to a file stream

In C++, there is no direct mechanism for reading and writing a single bit to a file stream. But you can read or write 8 bits at a time, i.e., one char. (See istream::read() and ostream::write() for details).

To convert individual bits into sequences of 8 bits, use the bitwise operators: |, &, ^, <<, >>. For example, consider the process of writing bits to an output stream. Start with a 1-byte buffer that consists of the null character (i.e., all of the bits are zero): `00000000`. Suppose you want to write the following sequence of bits: `100011110111` to the output stream. Using bitwise or, you can repeatedly add one bit to the buffer until the buffer is full. In the following diagram, the first row shows the individual bits that are written to the buffer and the second row shows the state of the buffer after the bit has been written.

```
  1          0          0          0          1          1          1          1
 10000000 10000000 10000000 10000000 10001000 10001100 10001110 10001111
```

Note that the state of the buffer is changed only when a 1 is written.

After adding 8 bits, the buffer is full. It can be written to the output file stream (see ostream::write() for details) and then cleared so that the remaining bits can be written to it:

```
                    0          1          1          1
                    00000000 01000000 01100000 01110000
```

Now that there are no more bits to be written, the buffer can be flushed, that is, written to the output file stream. There is nothing you can do about the additional 4 bits (all zeros) that are also written to the stream.

Design Document

You must document the design of your Huffman compression/decompression application. This design document called "HuffmanDesign.pdf" must contain:

- A simple UML class diagram of your classes.
- For each class, a point-form description of what it does.
- A drawing showing the format of the compressed files your application produces. More specifically, your drawing should show (i.e., label) exactly what type of information your application writes into the compressed files, i.e., information required to reconstruct the Huffman tree, and actual compressed data.

We shall examplify a design document in class on Friday March 8.

How to proceed

Implement your assignment in an incremental fashion: one class at a time. Test each class and then move on to the next.

Suggestion:

- Start by implementing the class that writes and reads bits to an output file and from an input file, respectively. You may want to test both operations together, i.e., read bits from a source file and write them to a destination file. Both files should be identical once your test driver has executed.
- You can then move onto to the class that reads an input file and creates its frequency table.
- You can also design and implement your minimum PriorityQueue class and test it thoroughly.
- Then, you are ready to deal with the class that is responsible for the creation of the Huffman tree.
- You can finish with the creation of the main function of your application.

Compiling and Testing

Makefile

You need to create your own makefile and submit it as part of this assignement (see Submission section below). Make sure it builds the executable "huff".

Test files

Here are some test files you can use to test the compression function of your Huffman compression/decompression application.

- emptyFile.txt
- oneCharFile.txt
- Muppets.txt
- MontyPythonandtheHolyGrail.txt
- picard.jpg

Ensure that you have tested your application with other files as well, such as executable files (*.class files), small files, etc. If well implemented, your compression application should be able to compress any types of file. (Why?).

Assumption: You can assume that when we test the decompression feature of your Huffman compression/decompression application, we shall only use files that have been compressed by your Huffman compression/decompression application.

Marking Scheme

When marking Assignment 4, we shall consider some or all of the following criteria:

- Solving a problem: Does your solution compress files and allow us to decompress them?
- Design and implementation of your solution: Does it satisfy the requirements stated in this assignment?
- Design document: Does it accurately described the design and implementation of your solution?
- Coding style: Has Good Programming Style been used?

Remember ...

You can use any C++ IDE you wish to do your assignments in this course as long as the code you submit compiles and executes on the CSIL computers. Why? Because your assignments will be marked using this platform (Linux and C++ - version running on CSIL computers). So, the suggestion is to compile and execute your code using your makefile (g++) at the Linux command line in CSIL before you submit your assignment to CourSys.

Submission

- Assignment 4 is due Wednesday, March 20 at 3pm.
- In the interest of fairness to your classmates, and expedient marking, late assignments will only be marked for feedback, and therefore will receive 0 marks.
- Since this assignment can be done in pairs, you are required to form groups of 2 on CourSys in order to submit your work - even if you have done this assignment on your own. In this case, you must form a group of 1 on CourSys.
- Submit a zip file (not a "huff" file :) named Huffman.zip to CourSys. This file must contain the following:
    - All you classes, i.e., all *.h and *.cpp files (or only *.h if you have templatized some of your classes).
    - The file containing the "main" function.
    - Your makefile file. Make sure it builds the executable "huff".
    - Your design document called "HuffmanDesign.pdf".
