Computer Science 245

Project #3

The Zip File

`README` Due: Friday, November 5th, 11:59 PM

`unzip` Due: Wednesday, November 10th, 11:59 PM

Final Project Due: Friday, November 12th, 11:59 PM

50 Points

Objective

The purpose of this assignment is to give students experience implementing a class to perform file compression and uncompression using the Huffman encoding algorithm. Students will gain experience developing a complete implementation for constructing and processing a tree using an array of structs.

Assignment Summary

Construct a program that takes as input a text file from the command line and produces an encoding table that could be used to compress this file using the Huffman Encoding algorithm.

You will need to develop and implement a class to represent the Huffman Tree ADT that includes the following operations :

- **insert** – inserts a character and its associated weight into a node of the Huffman tree
- **inTree** – returns a boolean value indicating whether a given character is stored in the tree
- **GetFrequency** – returns the weight of a given character
- **GetCode** – returns the compressed code of a given character in the tree using recursion
- **PrintTable** – displays the Huffman encoding table used in the compression process
- **numNodes** – returns the number of nodes currently in your Huffman tree

You can and should have additional private operations within your class that are called by the above operations. You may or may not want to have a public operation for merging two nodes (depending on how you design this portion of the algorithm).

The easiest data structure to represent this tree is an array or vector of structs where each struct includes the information about a character needed by the Encoding Table.

Your compression program should print the total bits used in the compressed file along with the compression ratio (as a percentage).

You are also required to write a short program which will successfully uncompress your zip file.

Program Files

You may copy over several files for use with this program by typing :

```
cp /pub/digh/CSC245/Prog3/* .
```

This will copy over several files for you including :

- `Zip*` – An executable version of the client software for compression that takes the source file as input from the command line; all compressed files are given a `.zip` extension
- `Unzip*` – An executable version of the client software for inflating a compressed file that takes the zipped filename as input from the command line (informs user if input file does not have a `.zip` extension); all uncompressed files remove the `.zip` extension
- `sample1` – A sample input file to be compressed based on our class example
- `letters.cpp` – A nice little C++ program to help you get started. This program reads in, stores, and prints out the frequency of all the characters in an input file.
- `huffman.h` – A complete sample specification file to represent a Huffman Tree ADT; feel free to modify/embellish
- `huffman.cpp` – An incomplete implementation file based on `huffman.h`

Name your compression program `zip.cpp`, your specification file `huffman.h`, your implementation file `huffman.cpp`, and your uncompress program `unzip.cpp`. **You should create an object file for your Huffman class before compiling your client file**.

Notes

1. The filename of the input file must come from the command line. If the filename entered by the user does not exist, inform them and terminate program. When running your uncompress program, you should also inform them if the source filename does contain the `.zip` extension.
2. The compress program may include the option `--t`. If it does, the program should display the Huffman encoding table. The user may also include the option `--help` for more information on using the command. Any other options or command formats should be flagged as invalid and reported to the user.

3. Your summary of the total bits used in the zip file must include a grand total of the bits used along with a compression ratio expressed as a percentage where

$$\text{Compression Ratio} = 1 - \frac{\text{\# Bits in Compressed File}}{(\text{\# of Chars in Source File}) * 8}$$

Shown next is an example using the `myText` input file.

```
File Successfully Compressed To 146 Bits (68.53% Less).
```

4. Your zip file will always have the same format : a header followed by the compressed data in binary. The header is a text-based portion which consists of an integer representing the number of characters in the source file on a single line, followed by a line for each character and its corresponding compressed string. Each line will have an integer representing the ASCII value of the character, and then a string representing that character's compressed string. This information is followed by the compressed data in binary. For example,

```
7
10 10001
32 00
97 101
101 11
105 01
115 10000
116 1001
00101101101101101101101101101010000001111111111111111111111111111110000000101
01010101010101010101000000100001000010000000000010011001100110011000110001
```

5. Upon uncompressing, inform the user with a message indicating that the zipped file has been successfully inflated. For example,

```
File Successfully Inflated Back To Original.
```

6. The function for printing your encoding table will be very helpful to you in debugging this program. You may want to use a special character such as `('\0')` to help identify your total nodes. Shown below is a sample encoding table for the `mytext` input file. Each total is printed as T1, T2, etc. T1 is certainly not a character, but is used in placed of the null character. When printing these out, just print a counter value right after a `'T'`.

```
        ++++  ENCODING TABLE  ++++

Index   Char    Weight  Parent  ChildType
0       nl      1       7       1
1       sp      13      10      0
2       a       10      9       1
3       e       15      11      1
4       i       12      10      1
```

```
5       s       3       7       0
6       t       4       8       1
7       T1      4       8       0
8       T2      8       9       0
9       T3      18      11      0
10      T4      25      12      0
11      T5      33      12      1
12      T6      58      0       N/A
```

You may obtain a slightly different listing of child type values depending on the order you used to insert your two children from the parent. However, the length of the trie path used to store each character in the compressed file will not vary.

Analysis & Design

You are required to complete an object-oriented design discussion with this assignment. You should show a short, sample text file for the string `"MERCER SOCCER"` followed by a newline character, and how it will look zipped. Explain to the reader the compression algorithm being used. You should also discuss how you plan on uncompressing the file. Mention that a `Huffman` class is being used, and briefly mention the attributes (private variables) and operations (public methods) of it.

Finishing Up

We need to create a typescript file. So, at your prompt in your program directory, type `script`. Now, perform each of the following steps:

1. display your client program by typing `cat -n zip.cpp`
2. display your `Huffman` specification by typing `cat huffman.h`
3. display your `Huffman` implementation by typing `cat huffman.cpp`
4. provide a sample compilation of your zip program by typing `c++ zip.cpp huffman.o`
5. rename your executable file as `myZip` by typing `mv a.out myZip`
6. provide a sample compilation of your unzip program by typing `c++ unzip.cpp`
7. rename your executable file as `myUnzip` by typing `mv a.out myUnzip`
8. display any input test case
9. provide a sample run of both programs
10. exit typescript by typing `exit`

Finishing Up

You now need to move your code into your `scratch` directory where you replace `last_fm` with your cobra username.

```
cp README /scratch/csc245/last_fm/Prog3
cp typescript /scratch/csc245/last_fm/Prog3
cp *.cpp /scratch/csc245/last_fm/Prog3
cp *.h /scratch/csc245/last_fm/Prog3
cp sample1 /scratch/csc245/last_fm/Prog3
```