CS 15 Project 3: zap

Contents

Overview

Introduction

Compression is the process of reducing the number of bits needed to represent some data. We rely on it all the time: when we stream music and movies, talk to someone on Zoom, or even access a web page. It's likely that you've compressed data before into a `.zip` file—maybe to send that data to someone via email, or maybe just to save some space on your hard drive.

Today, you will be writing your own compression program called `zap`! Your program will be used specifically for compressing and decompressing text files using the Huffman coding algorithm.

Running zap

`zap` can be run in just two ways: you can *zap* (compress) a file, or *unzap* (decompress) a file. To zap, from the command line run:

```
./zap zap inputFile outputFile
```

This will take the ASCII text file named `inputFile`, compress it using the Huffman coding algorithm, and store the result in a file named `outputFile`. It will also print to `cout` a message that looks as follows:

```
Success! Encoded given text using N bits.
```

where `N` is the size in bits of the encoded text.

To unzap, simply run:

```
./zap unzap inputFile outputFile
```

This will take the previously zapped file `inputFile`, decompress it, and store the resulting ASCII text in `outputFile`. It will not print anything to `cout`.

And that's it! There are no other ways to run `zap`, and unlike previous assignments, there is no command loop for this program.

An unzapped file's contents should be identical to the originally zapped file, e.g., if you run

```
./zap zap shakespeare.txt shake.zap
./zap unzap shake.zap new_shakespeare.txt
```

then the files `shakespeare.txt` and `new_shakespeare.txt` should be identical—you can use `diff` to confirm this.

If you run `zap` incorrectly, either by using the wrong number of command line arguments or by providing a first argument other than "zap" or "unzap", then `zap` will print the following usage message to `cerr`:
`Usage: ./zap [zap | unzap] inputFile outputFile`
then terminate with exit code `1`.

If you provide the name of an input file that cannot be opened, then your `encoder` function, or one of its helpers, should throw a `runtime_error` with the error message:
`Unable to open file FILENAME`
where `FILENAME` is the invalid input file name, and the error message **does not** end with a newline. The `encoder` function is described in detail below.

Program structure

Your program should include *at least* the following files:

- `HuffmanCoder.h` and `HuffmanCoder.cpp`. These files respectively define the interface and implementation of the `HuffmanCoder` class, which comprises the main logic of the Huffman coding algorithm. The class should have the following two public functions:

  ```
  void encoder(string input_file, string output_file)
  ```

  `encoder` takes a text file named `input_file`, compresses its text, and stores the result in a file named `output_file`.

  ```
    void decoder(string input_file, string output_file)
  ```

  `decoder` takes a zapped file named `input_file`, decompresses it, and stores the resulting text in a file named `output_file`.

  You can optionally add a public constructor and destructor to the `HuffmanCoder` class, but you are not required to do so. Any other functions that you add to this class **must be private**.

- `main.cpp`. This file includes your `main` function. It should process the command line arguments (ensuring they are valid), and appropriately call either the `encoder` or `decoder` function of the `HuffmanCoder` class.
- `HuffmanTreeNode.h`, `BinaryIO.h`, and their corresponding `.o` files: these files are provided by us, and will be discussed in greater detail later in the spec.

You can choose to add any other classes/files to your program as you see fit.

Starter Files

To copy the starter files, run the following command on the server:

```
/comp/15/files/proj3/setup
```

Note that you should **not** run `cp`.

Encoder

There are two primary components of `zap`: the *encoder*, which encodes a given ASCII text file into compressed binary code, and the *decoder*, which takes the compressed binary code and turns it back into text.

We begin by discussing the encoder, the more involved of these components. The encoder comprises the following stages:

(1) Counting character frequencies in the given text.

(2) Building the Huffman tree.

(3) Using the tree to generate character encodings.

(4) Using the generated encodings to encode the given text in binary.

(5) Serializing the Huffman tree.

(6) Saving the serialized tree and encoded text in a file.

Each of these stages is discussed in greater detail below. How you implement this functionality is up to you. At a minimum, each stage should be placed in its own function—those functions may in turn be broken down into smaller functions. They may also be placed in or make use of other classes if you so choose.

Counting Character Frequencies

Recall that the Huffman algorithm compresses data by encoding more frequently occurring characters using fewer bits. Thus, the first thing you must do is count the number of occurrences of each character in the given text. This count should include whitespace and punctuation characters, and it should separately count different casings of a letter. For example, given the following two-line text:

```
Apple apple.
Banana?
```

The counts you come up with should be as follows:

```
'A': 1
'p': 4
'l': 2
'e': 2
' ': 1
'\n': 1
'a': 4
'B': 1
'n': 2
'?': 1
```

Notice that the counts include whitespace characters.

How you store/represent these counts is up to you. You are welcome to use C++'s `std::map` or `std::unordered_map` libraries. Alternatively, given that there are just 256 possible values of a `char`, you may find it easy to represent counts using a more familiar data structure.

Building the Huffman Tree

Once you have character counts, you can build your Huffman code tree. Recall that this is a binary tree where:

- Leaves correspond to characters.
- On the path from the root to a leaf, each left turn denotes a '0' in a character's code, and each right turn denotes a '1'.
- Characters that occur less frequently should be further from the root, and characters that occur more frequently should be closer to the root.

Review the lecture on Huffman coding for a refresher.

You should build the tree using the `HuffmanTreeNode` class which we define for you in the provided files `HuffmanTreeNode.h` and `HuffmanTreeNode.o`. Take a look around the `.h` file. The `HuffmanTreeNode` class includes the following:

- A `char` member variable named `val`, which contains the character stored in this node. Because internal nodes in a Huffman tree do not store characters, you can assign this field to the null character '\0' for internal nodes.

- An `int` member variable named `freq`, which stores the character's frequency for leaf nodes, or the sum of the children frequencies for internal nodes.
- Two `HuffmanTreeNode` pointer fields `left` and `right`, for pointing to the children nodes.
- Two constructors which assign the corresponding member variables.
- A helper function `isLeaf()`.
- Some getter and setter functions for the member variables.

You'll also notice that `HuffmanTreeNode.h` includes the declaration for a *second* class named `NodeComparator`. This class defines a comparator function for comparing `HuffmanTreeNode*` instances, will allow you to use `HuffmanTreeNode*`'s within C++'s `std::priority_queue` data structure. It simply compares the `freq` fields of the two provided `HuffmanTreeNode`s.

You should use an `std::priority_queue` of `HuffmanTreeNode*`'s to build your Huffman tree. You can initialize a min-heap priority queue as follows:

```
priority_queue<HuffmanTreeNode*, vector<HuffmanTreeNode*>,
    NodeComparator> my_pq;
```

This creates a priority queue of `HuffmanTreeNode` pointers, which will be represented under the hood using a `vector`, and which uses the comparator function we defined. Using the priority queue is easy—you can review the `std::priority_queue` documentation here: https://cplusplus.com/reference/queue/priority_queue/. You should also look over the provided starter file `minpq_example.cpp` for an example of a C++ program that uses a min priority queue of `HuffmanTreeNode*`s.

With your priority queue initialized, building the Huffman tree is straightforward:

1. Create one `HuffmanTreeNode` for each character in the text, which stores the character and its frequency.
2. Push all nodes into the priority queue.

3. Get the two minimum frequency nodes from the priority queue (with the provided comparator function, all you need to do is call `top()` and `pop()` twice on the priority queue). Join the two nodes with a new `HuffmanTreeNode` parent that stores the sum of the frequencies of the children. Push the parent onto the priority queue.

4. Repeat step 3 until one node remains on the priority queue. That node is the root of your Huffman tree.

Generate Character Codes

With your Huffman tree built, you can now use it to generate the binary codes for each character. These codes can be represented simply as strings of 0s and 1s. Recall: the code for a character is represented by the path from the root to that character, where each left turn denotes a '0' and each right turn denotes a '1'. For example, given the following tree:

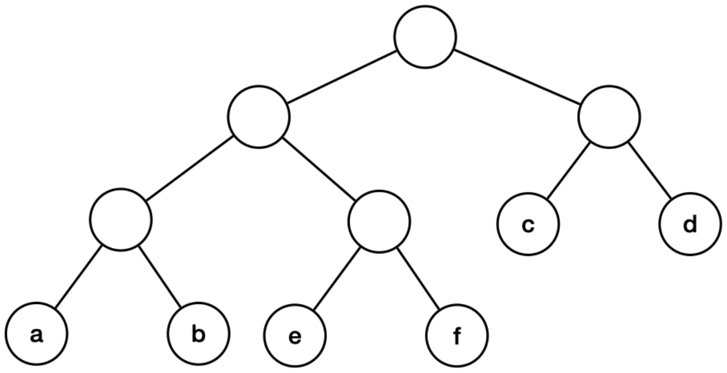

Figure 1: Huffman tree example.

You should generate the following codes:

```
a: 000
b: 001
e: 010
f: 011
c: 10
d: 11
```

Once again, how you store character codes is up to you—you will likely want to use the same method that you used to store character frequencies.

Encoding the Text

With the character codes, you are now ready to encode the given text into binary. All you need to do is iterate over the original text, look up each character's code, and append that code to the final encoded binary string. For example, given the character codes above, the encoding for the text "cafe" would be the binary string "10000011010".

Serialize the Tree

Whoever wants to later decode the Huffman encoded text will need access to the original Huffman tree. That means that we need a way to save the tree in a file such that it can later be reconstructed. This process of storing a program object for later reconstruction is known as *serialization*.

In our case, we will store the Huffman tree as a string. We can take advantage of the fact that a Huffman tree is a *full* binary tree: a binary tree in which every node has either 0 or 2 children (you can think about why this is the case—if a Huffman tree were not full, we could come up with a more efficient encoding).

A convenient property of full binary trees is that they can be uniquely represented using a variation of a preorder traversal of the tree: first store the current node, then recursively store the left subtree, then recursively store the right subtree. We must also distinguish between internal nodes and leaf nodes. Use the following approach for serializing your tree nodes:

- **Internal nodes**: these nodes do not store a character. Serialize an internal node as just the character `'I'`, followed by the serialized left subtree, then the serialized right subtree.
- **Leaf nodes**: these nodes do store a character. You should represent them using the character `'L'` followed by the character stored within that node.

For example, given the Huffman tree in Figure 1, the serialized version of the tree would be:
`"IIILaLbILeLfILcLd"`
Make sure you understand how the above serialization was carried out before implementing this step. With this serialized tree, we will later be able to reconstruct the tree when it is read from a file. Note that we did not store character frequencies in the serialized tree. That is because the frequencies

are not needed during decoding, they are only needed when first building the tree—so we can safely throw them away.

Saving to File

We now have the encoded text and the serialized Huffman tree which can be used for decoding. All that we need to do now is save these to a file. However, we don't want to just save the binary string to a file. Each '0' and '1' in the binary string is an ASCII character, so it takes up 8 bits of memory—8 times as large as a single 0 or 1 bit!

Writing actual bits to memory is a more involved process. Luckily, we have taken care of the details for you with the `BinaryIO` class. With your starter files, you should have received `BinaryIO.h` and `BinaryIO.o`, which respectively contain the interface and compiled implementation of the class. Take a look at the two public functions, `writeFile` and `readFile`, in `BinaryIO.h`. For this stage, all you need to do is provide the target filename, encoded binary string, and serialized tree to `writeFile`, and it will create the binary file for you.

You are now down with the encoding process! Print a message to the user that says:

```
Success! Encoded given text using N bits.
```

where `N` is the size of encoded text. **Note:** the printed `N` *does not* include the size of the serialized tree—it is just the size of the encoded text. Make sure to recycle any heap memory before your program terminates.

If you are curious to compare the size of your compressed data to the original data, in terminal you can use `ls -l` to view the size in bytes of all files in your current working directory. Alternatively, you can use `ls -lh` to view the sizes rounded to the nearest Kilobyte, Megabyte, etc.

Decoder

Decoding is the more straightforward component of Huffman coding. It involves decompressing a previously zapped file, and saving the resulting ASCII text to an output file. More precisely, it comprises:

1. Reading a zap file to get the serialized tree and encoded binary string.
2. Deserializing the tree.
3. Using the tree to decode the binary string into ASCII text
4. Saving the ASCII text to an output file.

Reading the zap file

You should once again use the provided `BinaryIO` class, this time to read from previously saved binary file. Simply call the `readFile` function, providing the `filename` to read from. This function will return a `vector<string>`, containing both the serialized tree and the binary string encoding. See the documentation in `BinaryIO.h` for more on the format of this vector.

Deserializing the tree

You must now *deserialize* the tree: convert it from its serialized string format back into a tree data structure composed of `HuffmanTreeNode`s. Use your knowledge of preorder traversals to do this: First create a `HuffmanTreeNode` for the current node you are reading in from the serialized string. Then, if this is a non-leaf node, you should next recursively deserialize the node's left subtree, then its right subtree. Remember the properties of the serialized tree: ‘`I`’s denote internal nodes (which do not contain characters), and ‘`L`’s denote a leaf, and are followed by the character contained within that leaf.

After deserializing, you should once again have the root `HuffmanTreeNode` of a valid Huffman tree.

Decoding

Using this Huffman tree, you can now decode the encoded binary string. Recall the algorithm for doing this: starting at the root of the tree, read in bits one at a time. For each 0, go to the left child, and for each 1, go to the right child. Once you reach a leaf node, append the character in that leaf node to the overall decoded message. Then, go back to the root and repeat this process until all bits from the encoding have been processed.

For example, given the tree in Figure 1, and the encoded message `01100010010`, you should get the decoded message “face”. Try it by hand and make sure that this is what you get!

The last bit you read from the binary encoding should lead you to a leaf—that is, you should never finish reading in bits while in the middle of the tree. For example, given the tree in Figure 1, if you were given the encoded message ‘00101’, something must be wrong!

Should this happen, your `decoder` function, or one of its helpers, should throw a `runtime_error` with the error message:
`Encoding did not match Huffman tree.`
This error message **should not** terminate with a newline.

Saving to file

You should now have the decoded text. Save this text to the output file named on the command line, and you're done! Make sure to recycle any heap memory before your program terminates.

Reference Implementation and Testing

To assist you with implementing and testing your program, we have provided you with `the_zap`, a compiled reference implementation. You should play around with this implementation to get a feel for how it works, and use it to test your own implementation once it is built.

It is important to note one thing: There is an ambiguity in the Huffman coding algorithm which may affect your testing. Specifically, when building the Huffman tree, the algorithm dictates that when there are more than two minimum frequency nodes, the tie can be broken arbitrarily: *any* two minimum nodes may be picked next. As a result, there may be multiple valid Huffman trees, and hence multiple valid Huffman encodings for a particular text. How an encoding ultimately is chosen may rely on unimportant factors, like the order in which `HuffmanTreeNode`s are added to the priority queue.

What this means for you: It is possible that providing the same input text to your program and to the reference implementation will result in two different `zap` files. Hence, it will not always be possible to use `diff` testing with `zap` files.

Luckily, one property will always hold true: *any* valid Huffman encoding will *always be the same size.* Therefore, though the `zap` file created by your implementation may differ from the reference, the *size* of the encodings should be the same, and thus the message printed to `cout` regarding encoding size should always be the same. For example, if the reference prints to `cout`:

```
Success! Encoded given text using 25053897 bits.
```

then your program's output should print the exact same thing. So, you should rely on redirecting the output from `cout` and `cerr` to specific files, and `diff` test those files against the reference's output.

There are other ways to test your program as well. For example, any `zap` file created by your implementation should be unzap-able by the reference implementation, and vice versa. Try it out: `zap` a file with your implementation, and `unzap` with the reference. Is the resulting text the same as the original? If not, something has gone wrong.

You can also test your own implementation's encoder against its decoder, for example:

```
./zap zap A.txt B.zap
./zap unzap B.zap C.txt
diff A.txt C.txt
```

Calling `diff` above should result in no output—`A.txt` and `C.txt` should contain identical contents. If they don't, you have some debugging to do.

As always, beyond `diff` testing, you should unit test your program early and often. Because you will likely have many private functions to test, you may find it useful to temporarily make these functions public for testing purposes—just be certain to make them private again before your final submission.

Finally, make sure to submit all testing files you create, whether for unit or diff testing. This includes:

- Any new input text files you create to pass to `./zap zap` in addition to the provided text files.
- Any output files you create from redirecting `cout` when running `./zap zap` to compare with the reference solution.
- Any output files you create from decoding a `zap` file that you encoded. These should be identical to whatever the input text file you used.

Make sure you submit the output files even for your tests on the provided text files as these will be used as evidence of your diff testing on this assignment. You should describe these files in README section G.

Testing Files

To test your solution, we have provided you with some text files. You can copy them from `/comp/15/files/proj3-test-files/`. You should use them to test if your compression size matches the reference solution. The files are:

- `banana.txt` contains the text in the example for the `count_freqs` function in Phase One. This file is 6 bytes.

- `hi.txt` contains the text in the example for the `count_freqs` function in Phase One. This file is 8 bytes.
- `banana_apple.txt` contains the text in the example in Counting Character Frequencies. This file is 20 bytes.
- `sentences.txt` contains a few sentences from a novel. This file is 225 bytes.
- `all_conll_english.txt` contains the 2003 CoNLL English dataset used commonly in Natural Language Processing. This file is about 5 MB.
- `works_of_shakespeare.txt` contains the works of William Shakespeare. This file is about 5 MB.
- `ecoli.coli` contains the E.coli genome made up of the 4 nucleotide bases of DNA. This file is about 5 MB.

Makefile and README

Makefile

You must write and submit a `Makefile` along with your code. When we run both `make` and `make zap`, your `Makefile` should build your program and produce an executable named `zap` that we can run.

README

In addition to your program and testing files, you should submit a `README` file that includes the following sections:

A. The title of the assignment and your name.

B. The purpose of the program.

C. Acknowledgments for any help you received.

D. A list of the files that you provided and a short description of what each file is and its purpose.

E. Instructions on how to compile and run your program.

F. An outline of the data structures and algorithms used. There should be a number of data structures for this assignment. Discuss each one, and why it was the correct choice for the task at hand. Additionally, pick a couple algorithms that you relied on to complete this assignment.

G. Details and an explanation of how you tested the various parts of assignment and the program as a whole. You may reference the testing files that you submitted to aid in your explanation.

I. Please let us know approximately how many hours you spent working on this project. This should include both weeks one and two.

Each of the sections should be clearly delineated and begin with a heading that describes the content of the section.

Submitting your work

Phase One

You will need to submit two files for phase one:

- `phaseOne.h`
- `phaseOne.cpp`

`phaseOne.h` should contain the declarations of three functions, and `phaseOne.cpp` should contain their definitions. These functions are described below.

Note: You should not declare or define any classes for this phase! Though we are used to splitting up interface/implementation files for classes, in this case you are only declaring and defining *functions*, not classes.

The functions in these files should be the following:

- `void count_freqs(std::istream &text)`
  This function takes an istream reference `text` as input (remember, an `istream` could be various kinds of streams: an `ifstream`, an `istringstream`, `cin`...). It should read in the entire stream and count the number of occurrences of every character. Finally, it should print to `cout` the number of occurrences of every character. Each character should be printed on its own line using the format:
  `CHAR: NUM`
  For example, if `text` contains the string `BaNana`, your function should print to `cout`:

  ```
  B: 1
  a: 3
  N: 1
  n: 1
  ```

  The function can print the character counts in any order. Note that your function should count non-alphanumeric characters as well. This could make the format of the output looks strange, especially in the presence of whitespace characters, but that is okay. For example, if `text` contains:

```
hi  hi
hi
```

Your function should print to `cout`:

```
h:  3
i:  3
 :  1

:  1
```

Notice that the newline's frequency is reported across two lines.

- `std::string serialize_tree(HuffmanTreeNode *root)`

  This function takes a pointer to the root of a Huffman tree as input, and it returns the serialized version of the tree. The tree should be serialized according to the format described earlier.

- `HuffmanTreeNode *deserialize_tree(std::string serial_tree)`

  This function goes in the other direction: given a serialized tree as input, it reconstructs the Huffman tree and returns a pointer to its root.

You may declare/define additional helper functions in phase one, but you are not required to. You should test your functions thoroughly. How you do so is up to you—you can use the `unit_test` framework, or you can write your own `main()` function (but you should do so in a separate file, since our autograder will use its own `main`s).

Phase One README

Your `README` does not need to contain everything we normally ask for. For this phase, you can just write a very brief summary with any information you'd like the grader to see (e.g., if there is a bug you have not been able to fix).

Provide

The provide command for phase one is:

```
provide comp15 proj3_zap_phase1 README phaseOne.h phaseOne.cpp
    [...any testing files...]
```

Phase Two

When you begin working on your final submission, you will find the functions from phase one useful. For the final submission, you *do not* need to keep these functions within the `phaseOne` files, and you are welcome to change other properties of the functions as well (their names, type signatures, etc.). At a minimum, you will not want your `count_freqs` function to print anything to `cout`. But you will want to reuse the logic within the functions.

For this part you will submit all of the files required to compile your `zap` program, including a `Makefile`. Be sure to include any files you used for testing, including test data that you created. The `provide` template is:

```
provide comp15 proj3_zap README Makefile unit_tests.h main.cpp
    HuffmanCoder.h HuffmanCoder.cpp
```

Note: We cannot give you a complete `provide` command because we do not know what files you will have. So, make sure to submit everything we need to run your program. Maybe copy the files to another directory, type `make` to try building your program, do a `make clean` and then provide everything in the directory. We should be able to use `make` or `make zap` to build your program.

Helpful Tips

- At various points in your program, you'll need to read characters one at a time from a stream. Most likely, you **do not** want to use the `>>` operator to do this. This operator skips over whitespace characters—but in order to correctly compress and decompress text, whitespace characters should not be skipped. Instead try using the `get()` function, which can read *any* single character from a stream. Here is the documentation for `istream`'s `get()` function: https://cplusplus.com/reference/istream/istream/get/

- We will only test your program on ASCII text. Recall that modern ASCII encodings use 8 bits to represent up to 256 characters. Moreover, under the hood, C++'s `chars` are represented using the character's ASCII integer. That means you can use a `char` in contexts where an integer is expected, you can cast `char`s to `int`s, or you can use the `int()` constructor to get a `char`'s ASCII number—e.g., for a `char c`, the expression `int(c)` will give you the character's ASCII number. You may find it useful to leverage this knowledge in your implementation.