lab_huffman

Hazardous Huffman Codes

Due: Mar 03, 23:59 PM

Doxygen | Lab handout

Assignment Description

In this lab, you will be exploring a different tree application (Huffman Trees), which allow for efficient lossless compression of files. There are a lot of files in this lab, but you will only be modifying `huffman_tree.cpp`.

Lab Insight

Huffman encoding is a fundamental compression algorithms for data. Compressing data is a very powerful tool that can represent a given set of information in less space, thus allowing the data to be transferred more efficiently. Different types of compression can be seen within images formats like JPG(lossy) or PNG(lossless). It can also be seen in ZIP files for compressing multiple files. The concept of encoding data can be seen in future courses CS 438, Communication Networks, dealing with transferring large amounts of data, and CS 461, Computer Security, which deals with encoding data for a layer of privacy.

Getting Set Up

From your CS 225 git directory, run the following on EWS:

```
git fetch release
git merge release/lab_huffman -m "Merging initial lab_huffman files"
```

If you're on your own machine, you may need to run:

```
git fetch release
git merge --allow-unrelated-histories release/lab_huffman -m "Merging initial lab_huffman
files"
```

Upon a successful merge, your lab_huffman files are now in your `lab_huffman` directory.

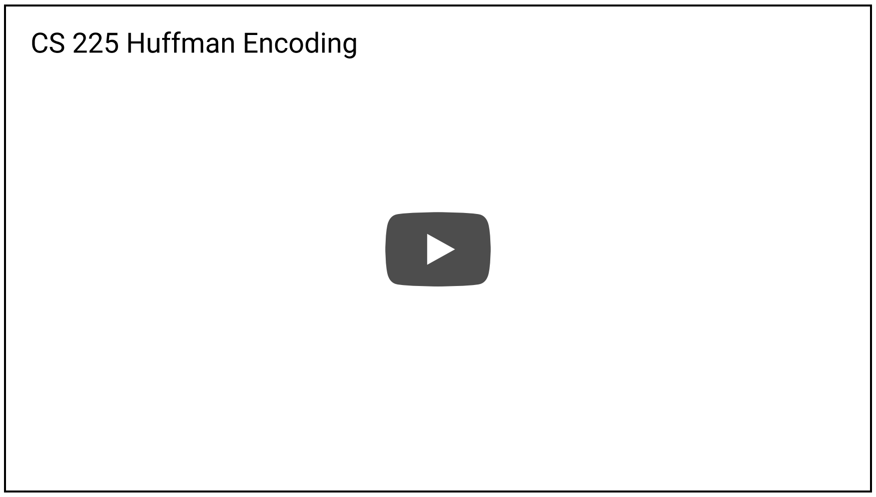
The code for this activity resides in the `lab_huffman/` directory. Get there by typing this in your working directory:

```
cd lab_huffman/
```

Video Intro

The following is meant to help you understand the task for this lab. It is strongly recommended that you watch the video to understand the motivation for why we're talking about Huffman Encoding as well as how the algorithm works.

There is a video introduction for this lab! If you are interested in seeing a step-by-step execution of the Huffman Tree algorithms, please watch it:



The Huffman Encoding

In 1951, while taking an Information Theory class as a student at MIT, David A. Huffman and his classmates were given a choice by the professor Robert M. Fano: they can either take the final exam, or if they want to opt out of it they need to find the most efficient binary code. Huffman took the road less traveled and the rest they say is history.

Put simply, Huffman encoding takes in a text input and generates a binary code (a string of 0's and 1's) that represents that text. Let's look at an example: Input message: "feed me more food"

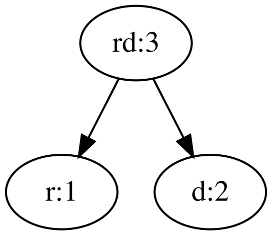
Building the Huffman tree

Input: "feed me more food"

Step 1: Calculate frequency of every character in the text, and order by increasing frequency. Store in a queue.

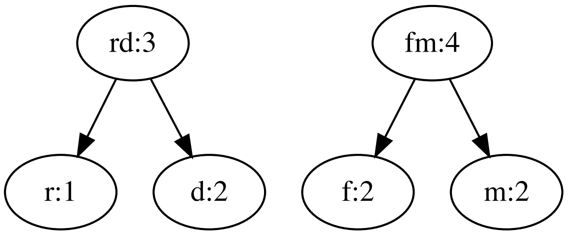
```
r : 1 | d : 2 | f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4
```

Step 2: Build the tree from the bottom up. Start by taking the two least frequent characters and merging them (create a parent node for them). Store the merged characters in a new queue:



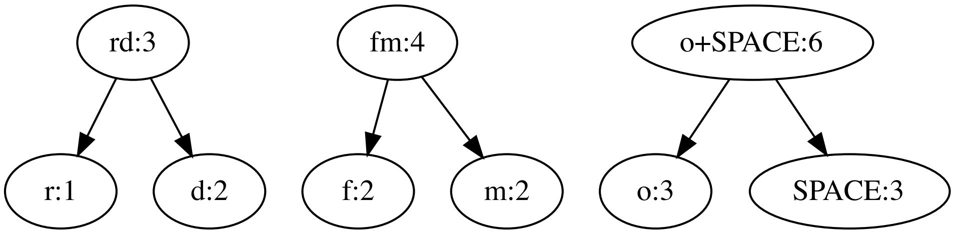
```
SINGLE: f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

MERGED: rd : 3
```

Step 3: Repeat Step 2 this time also considering the elements in the new queue. 'f' and 'm' this time are the two elements with the least frequency, so we merge them:

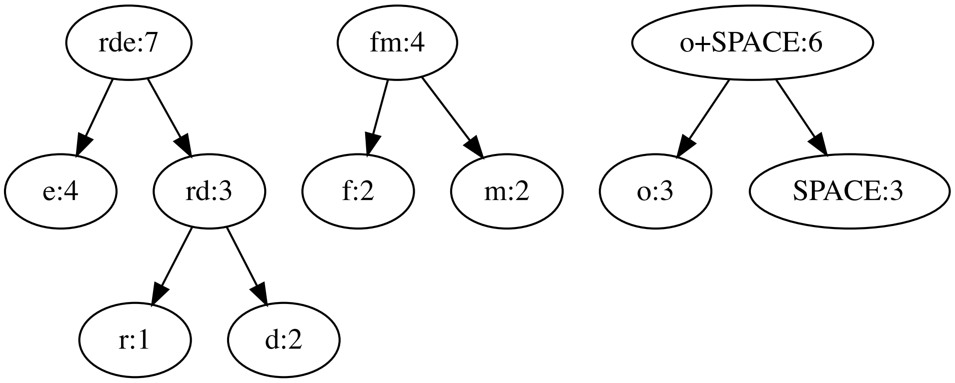


```
SINGLE: o : 3 | 'SPACE' : 3 | e : 4

MERGED: rd : 3 | fm : 4
```

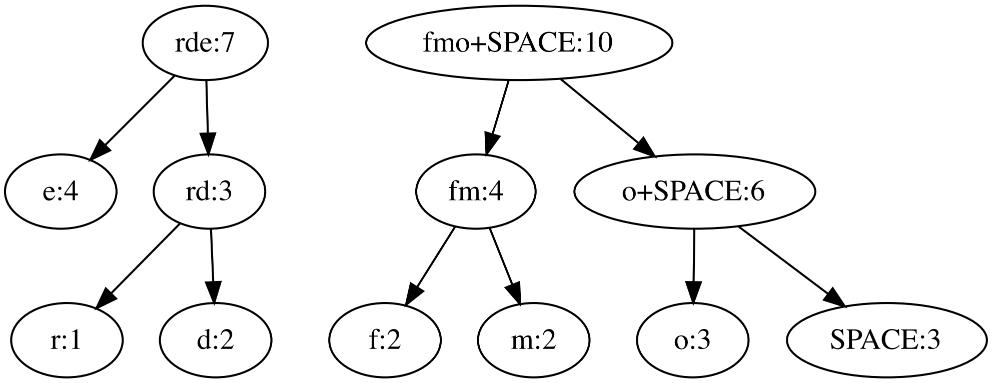
Step 4: Repeat Step 3 until there are no more elements in the SINGLE queue, and only one element in the MERGED queue:



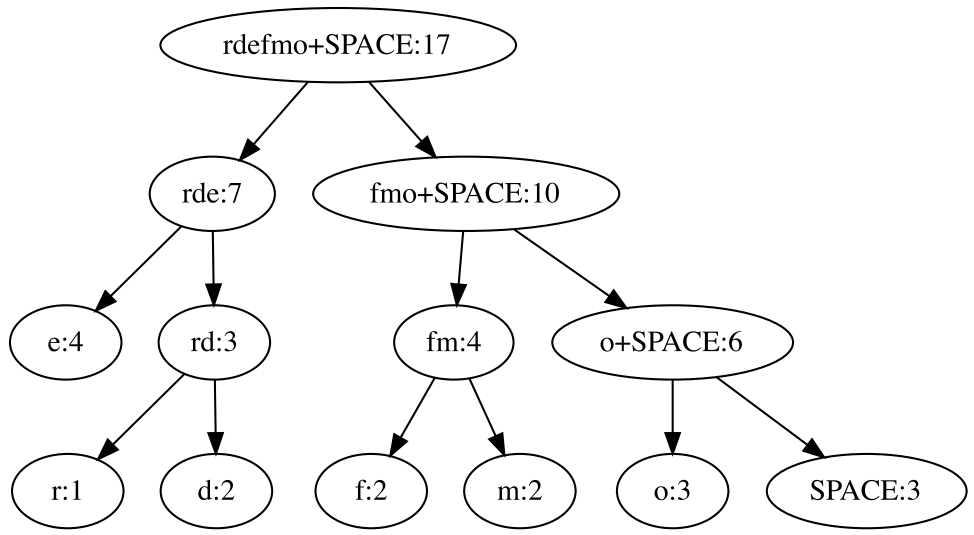
```
SINGLE: e : 4

MERGED: rd : 3 | fm : 4 | o+SPACE : 6
```



```
SINGLE:

MERGED: fm : 4 | o+SPACE : 6 | rde: 7
```



```
SINGLE:

MERGED: rde: 7 | fmo+SPACE: 10
```



```
SINGLE:

MERGED: rdefmo+SPACE: 17
```

From Text to Binary

Now that we built our Huffman tree, its time to see how to encode our original message “feed me more food” into binary code.

Step 1: Label the branches of the Huffman tree with a ‘0’ or ‘1’. BE CONSISTENT: in this example we chose to label all left branches with ‘0’ and all right branches with ‘1’.

Step 2: Taking one character at a time from our message, traverse the Huffman tree to find the leaf node for that character. The binary code for the character is the string of 0’s and 1’s in the path from the root to the leaf node for that character. For example: ‘f’ has the binary code: `100`

So our message “feed me more food” becomes `1000000011111101001111011100100011110011011 0011`

Efficiency of Huffman Encoding Notice that in our Huffman tree, the more frequent a character is, the closer it is to the root, and as a result the shorter its binary code is. Can you see how this will result in compressing the encoded text?

From Binary Code to Text

We can also decode strings of 0’s and 1’s into text using our Huffman tree. What word does the code `01000011` translate to?

What About the Rest of the Alphabet?

Notice that in our example above, the Huffman tree that we built does not have all the alphabet’s letters; so while we can encode our message and some other words like “door” or “deer”, it won’t help us if we need to send a message containing a letter that’s not in the tree. For our Huffman encoding to be applicable in the real world we need to build a huffman tree that contains all the letters of the alphabet; which means instead of using “feed me more food” to build our tree, we should use a text document that contains all letters of the alphabet to build our Huffman tree. As a fun example, here is the Huffman tree that results when we use the text of the Declaration of Independence to build it.

Here is the Doxygen generated list of files and their uses.

Static Keyword The static keyword means that the variable or function is shared by all instances of the class. This means that if a static function is used that inside the function, no references to the functions member variables may be used (no access to `this` pointer). Static functions can be beneficial when it is inconvient to make a new instance of a class, but it would be nice to use the member function. For example, if you’re inside a member function and want to call a static function of that class you can do `myStaticHelper(args)` the same way you’d call another member function.

Implement `buildTree()` and `removeSmallest()`

Your first task will be to implement the `buildTree()` function on a `HuffmanTree`. This function builds a `HuffmanTree` based on a collection of sorted `Frequency` objects. Please see the Doxygen for `buildTree()` for details on the algorithm. You also will probably want to consult the list of constructors for `TreeNodes`.

You should implement `removeSmallest()` first as it will help you in writing `buildTree()`!

Tie Breaking To facilitate grading, make sure that when building internal nodes, the left child has the smallest frequency.

In `removeSmallest()`, **break ties by taking the front of the `singleQueue`**!

Implement `decode()`

Your next task will be using an existing `HuffmanTree` to decode a given binary file. You should start at the root and traverse the tree using the description given in the Doxygen. Here is the Doxygen for `decode()`.

You will probably find the Doxygen for `BinaryFileReader` useful here.

We’re using a standard `stringstream` here to build up our output. To append characters to it, use the following syntax:

```
ss << myChar;
```

Implement `writeTree()` and `readTree()`

Finally, you will write a function used for writing `HuffmanTrees` to files in an efficient way, and a function to read this efficiently stored file-based representation of a `HuffmanTree`.

Here is the Doxygen for `writeTree()` and the Doxygen for `readTree()`.

You will probably find the Doxygen for `BinaryFileWriter` useful here.

Testing Your Code!

We’ve provided you with a collection of data files to help you explore Huffman encoding. Run the following command to download and extract the files. They will be in a newly-created `data` directory.

```
wget
https://courses.engr.illinois.edu/cs225/sp2019/assets/assignments/labs/huffman/lab_huffman_da
 && tar -xf lab_huffman_data.tar && rm lab_huffman_data.tar
```

When you run `make`, two programs should be generated: `encoder` and `decoder`, with the following usages:

```
$ ./encoder
Usage:
        ./encoder input output treefile
                input: file to be encoded
                output: encoded output
                treefile: compressed huffman tree for decoding

$ ./decoder
Usage:
        ./decoder input treefile output
                input: file to be decoded
                treefile: compressed huffman tree to use for decoding
                output: decompressed file
```

Use your `encoder` to encode a file in the data directory, and then use your compressed file an the huffman tree it built to `decode` it again using the decoder. If `diff`-ing the files produces no output, your `HuffmanTree` should be working!

When testing, try using small files at first such as `data/small.txt`. Open it up and look inside. Imagine what the tree should look like, and see what's happening when you run your code.

Now try running your code:

```
$ ./encoder data/small.txt output.dat treefile.tree
Printing generated HuffmanTree...
                                  ______________ 28 _____________
                   ______________/                               \_______________
          ______ 11 _____                                                 ______ 17 _____
   ______/               \______                                   ______/
\______
s:5                           __ 6 __                         __ 8 __
__ 9 __
                           __/       \__                   __/       \__
__/       \__
                        y:3              3              l:4             i:4              4
:5
                                       /   \                                           /
\
                                    h:1     t:2                                      2
2
                                                                                    / \
/ \
                                                                                 \n:1 r:1
o:1 a:1
Saving HuffmanTree to file...
```

Differing Output It is possible to get different output than this tree and still pass catch. **Use the provided test cases on catch to see if your code is passing.**

You can also test under catch as usual by running:

```
make test && ./test
```

Submitting Your Work

The following files are used to grade this assignment:

- `huffman_tree.cpp`
- `huffman_tree.h`
- `partners.txt`

All other files including any testing files you have added will not be used for grading.

Good luck!

Guide: How to submit CS 225 work using git