lab_huffman Hazardous Huffman Codes

Due: **Sunday, April 3 at 11:59 PM**

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`.

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: Huffman Encoding Video.

Checking Out Your Code

To check out your files for this lab, run the following from your `cs225` directory:

```
svn up
cd lab_huffman
make data
```

This will create a new folder in your working directory called `lab_huffman` and grab the data text files we will be dealing with.

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

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 have provided you with a set of data files in the `data` directory you checked out. 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 monad. **Use the provided test cases on monad to see if your code is passing.**

You can also test under monad as usual by running:

```
./monad lab_huffman
```

Grading Information

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.