Assignment 4a: Building a Huffman Tree

Data Structures, CSCI-UA 102, Section 7
Instructor: Max Sklar

Build a Huffman Tree from a list of symbol frequencies or a corpus of text.

Overview

The task is to send a message that represents a sequence of symbols using a sequence of bits. In a *fixed-length encoding* (like ASCII), each symbol is encoded with the same number of bits. If there are n symbols, $\lceil log_2(n) \rceil$ bits are needed. There are a couple of drawbacks to fixed-length encoding:

1. Not all encodings will represent a symbol unless n is a power of 2.
2. Each symbol uses the same number of bits even though some appear more frequently than others (and often dramatically so).

In this two-part assignment, you will instead implement an efficient scheme for compressing a text message into a *variable-length encoding* called a *Huffman code*. Your system will assigns each symbol a unique sequence of bits. To avoid any ambiguity, no symbol may have a bit sequence that is the prefix for another symbol. The huffman algorithm finds the optimal encoding rules given the relative frequencies of each symbol.

You will be given a set of symbols and their corresponding freqencies. From there, you will build a special binary tree called a Huffman Tree where the path to each leaf represents a different symbol.

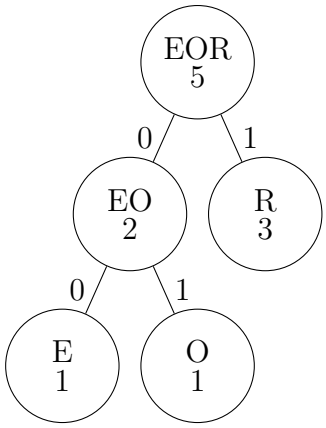
Your function will run in one of two modes. In one mode, you will print a table of the bit encoding for each symbol. In specification mode, you will print a special traveral of the Huffman Tree that will allow us to store the tree and reconstruct it later.

Huffman Tree

A Huffman Tree stores elements based on their frequency such that the more common symbols have shorter paths. For example, take the word "ERROR". The symbol frequency input will look like this:

```
E 1 R 3 O 1
```

Here is a corresponding Huffman Tree:



The leaf nodes are the symbols with their frequencies, and the internal nodes are the combinations of its children's symbols with the sum of their frequencies. Notice the labels on the paths from node to node: each left child is a 0 and each right child is a 1. This is the basis for the encoding.

Character	Encoding	Frequency
E	00	1
O	01	1
R	1	3

Using this encoding, we can represent the word "ERROR" with only 7 bits (0011011). This is far fewer than ASCII (40), and even in the most efficient fixed length encoding of 2 bits per character, we would still be using 10 bits.

Implementation

The class you must complete is `HuffmanTree.java`. When its main function runs, it will take a list of character frequencies and a mode as input, and produce a huffman tree. In order to do this, the following classes are provided in full:

- `BinaryHeap.java`: This is an implementation of the Binary Heap / Priority Queue that you'll need in order to find the correct Huffman Nodes to combine and create branching nodes.
- `StdinToString.java`: This is a convenience utility that you'll call from your main function in order to take input from `stdin` and turn it into a string.
- `FrequencyCounter.java`: This class will not be used in building your huffman tree, but it is used to turn any file into a list of character frequencies, which can then be fed into your HuffmanTree code.

The `HuffmanTree` class will contain a private class called `HuffmanNode`. You will be responsible for building the constructors, and implementing the compareTo() function.

```
private static class HuffmanNode implements Comparable<HuffmanNode> {
    final public String symbols;
    final public double frequency;
    final public HuffmanNode left, right;

    //public HuffmanNode(String symbol, double frequency) {}
    //public HuffmanNode(HuffmanNode left, HuffmanNode right) {}
    //public int compareTo(HuffmanNode o) {}

    public String toString() {
      return "<" + symbols + ", " + frequency + ">";
    }
  }
```

The rest of the `HuffmanTree class` will also have methods that need to be filled in.

```
public class HuffmanTree {

   HuffmanNode root;

   private static class HuffmanNode implements Comparable<HuffmanNode> {...}

   // public HuffmanTree(HuffmanNode huff)
   // public void printLegend()
   // public static BinaryHeap freqToHeap(String frequencyStr)
   // public static HuffmanTree createFromHeap(BinaryHeap b)
   public static void main(String[] args) {...}
   public static String convertSymbolToChar(String symbol) {..}
}
```

There are two static methods implemented to help you get started.

1. `main` Read the main function carefully. Note that it takes 2 types of input. The first is the "mode" which comes in through the command line to tell your class how to print out the tree. The second comes from standard input (using the StdinToString helper function), which is passed in differently (see below)

2. `convertSymbolToChar` Your output will include characters that should for the most part be the same as the symbols you import. However, there are some exceptions:

   - space: This will be written verbatim as "space" in your input file to distinguish it from the spaces used as separators.
   - eom: For part 2, we'll need an "eom" or "end of message" symbol that represents the end of a given message. In a variable length encoding, there is no way to calculate when the message ends given the number of characters, so this sentinel value is sometimes neccesary. To represent this in our output, we will use the backslash as an escape character and make it "$\backslash e$"

- Pipe (|): This symbol is going to have a special meaning in our tree specifier, so it needs to be printed differently when it is used. We will again use the backslash as an escape character to make it "\|".
- Backslash (\): Well, now that backslash is being used as an escape character, we need to use it to represent itself as "\\". Note that Java also uses the backslash as an escape character, so the code contains four backslashes in a row!

Reading from Standard Input

The frequencies will be read into HuffmanTree using the *standard input* or *stdin* stream. This is a lot like reading in a file, except the data can be sourced and redirected from any applicable source.

For our test case, we will use the given file `sample_frequencies.txt`. If you want to print this out on the command terminal, you can type:

```
cat sample_legend.txt
```

In order to pipe it through to your HuffmanTree java process, you can type:

```
cat sample_frequencies.txt | java HuffmanTree
```

Keep in mind that on the windows command prompt, it is type instead of cat:

```
type sample_frequencies.txt
type sample_frequencies.txt | java HuffmanTree
```

Your IDE will also have a way to send data from a file through to standard input, usually under "run configurations". If you do this correctly, and with the given `sample_frequencies.txt` file, your `StdinToString.read()` call should return the string

```
"A 20 E 24 G 3 H 4 I 17 L 6 N 5 O 10 S 8 V 1 W 2"
```

Methods to be Implemented

Here is informtion about the methods that you need to implement.

HuffmanNode

- **public HuffmanNode(String letter, Double frequency)**
  Constructor that creates a new HuffmanNode.
- **public HuffmanNode(HuffmanNode left, HuffmanNode right)**
  Constructor that creates a new HuffmanNode from its two children.
- **public int compareTo(HuffmanNode hn)**
  This allows creation of a heap of HuffmanNodes where frequency determines ordering.
- **public String toString()**
  This is given. It returns a string of the form `"<"+letter+", "+frequency+">"`.
  This will be helpful when debugging.

HuffmanTree

- **public static BinaryHeap freqToHeap(String legend)**
  Takes a String of frequency data (as in `sample_frequencies.txt`) as input. Symbols and frequencies are separated with single spaces. Symbols will be single characters for the most part. The only exception to that is the frequency for the space character will be denoted by "space", and another special symbol "eom" will be used to denote the end of the message. `sample_frequencies.txt` does not contain either of these, but `FrequencyCounter.java` will include it when counting symbols in a file (such as `just_to_say.txt`.

- **public static HuffmanTree createFromHeap(BinaryHeap b)**
  Implements the Huffman algorithm (below). When only one element remains in the heap, it called extractMin to return the root node.

- **public HuffmanTree(HuffmanNode huff)**
  Constructor that sets `this.root`. This is used after the tree is fully constructed.

- **public void printLegend()**
  This prints out a list of each symbol and its corresponding bits. Each line of the output represents a single symbol, and the two columns are separated by a tab. The following line of code would print out a symbol and its bits correctly:

  ```
  System.out.println(convertSymbolToChar(symbol) + "\t" + bits);
  ```

  The implementation of this method will require a tree traversal, starting from the left side (bit 0) and then the right side (bit 1). Most implementations work by storing the bits seen so far at each point in the traversal, so that when you reach a leaf you know what to print out. The correct legend for `sample_frequencies.txt` is given in `sample_frequencies_correct_legend.txt`.

- **public void printTreeSpec()**
  This prints out a string that is meant to go in a file so that it can be saved and later reconstructed into the original huffman tree. The frequencies are not saved, because those are irrelevant to the encodings - only enough information to organize the characters into a tree.

  The tree specification is a string of symbols and also pipes (|). In part 2, you will pass this string to a method that will reconstruct the huffman tree by pushing every symbol onto a stack, and combining the top two simples into an internal node every time a pipe is encountered. When the string is finished being read, the top two nodes are combined until there is only one node on the stack (the additional pipes at the end to get the stack to one element do not need to be included.)

  Building the specification will also require a post-order tree traversal. At every leaf, print the symbol using `convertSymbolToChar`. At every internal node, print a pipe (|) unless it's on the rightmost path of the tree - in which case it should not be included because the pipes at the end of the sequence are inferred.

The correct specification for `sample_frequencies.txt` is given in

`sample_frequencies_correct_spec.txt`.

- **public static void main(String[] args)**
  This is given. and controls the program flow and puts together all the components you have built above.

  1. It finds the `mode` (either "legend" or "spec" from the command line.
  2. It calls `StdinToString.read()` to find the string containing all of the symbols and frequencies.
  3. Calls `freqToHeap()` on the frequency string to get a BinaryHeap.
  4. Calls `createFromHeap()` to build the HuffmanTree and return the root node.
  5. Calls the HuffmanTree constructor to finalize the tree.
  6. Calls either `htree.printLegend()` to print binary encodings or `htree.printTreeSpec()` to print the tree specification.

The Algorithm

The input is a list of characters and their corresponding frequencies. The output is a Huffman Tree, build using a Binary Heap.

1. Create a single HuffmanNode for each letter and its frequency, and insert each of these into a new BinaryHeap. Call the BinaryHeap constructor that takes an array of Comparables.
2. While the Binary Heap has more than one element:
   (a) Remove the two nodes with the minimum frequency.
   (b) Create a new HuffmanNode with those minimum frequency nodes as children (using the HuffmanNode constructor with left and right nodes as parameters) and insert that node back into the BinaryHeap.
3. The BinaryHeap's only element will be the root of the Huffman Tree. Pass this node into the HuffmanTree constructor and return the result.

Testing

You can store your output to a file in the following way:

```
cat sample_frequencies.txt | java HuffmanTree legend > my_legend.txt
```

Then you can compare it with the correct one by calling

```
cmp sample_frequencies_correct_legend.txt my_legend.txt
```

Submission Requirements

Submit the following files:

1. HuffmanNode.java
2. HuffmanTree.java
3. BinaryHeap.java (with any of your adjustments if you choose to do so)

README

Included in the `readme.txt` file should be the following:

RUNBOOK: Information on how to compile and run your code in the command line.

TIME SPENT: An estimate of the time spent working on this project.

NOTES: Include the difficulties and roadblocks encountered while working on this project, including notable bugs that needed to be overcome. Are there any quirks in your solution that we should be aware of? Was the outcome what you expected?

RESOURCES AND ACKNOWLEDGEMENTS: Disclose the sources you consulted for help on this project. Possible human sources include: NYU Staff (the instructor, TA, tutor), other students, and anyone who helped you with the project. Sources that you consulted, including code that you found online, in the book, and output from LLMs. Describe the nature of your LLM usage.

There will never be a grading penalty for a complete and accurate accounting of resources and acknowledgments.