Assignment 5a: Building a Huffman Tree

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

Build a Huffman Tree from a list of symbols and their frequencies or a corpus of text.

Overview

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 to represent each one. This approach has a couple of drawbacks:

1. Not all encodings will represent a symbol unless n is a power of 2.
2. Every symbol uses the same number of bits, even though some occur much more frequently than others.

In this two-part assignment, you will instead implement an efficient scheme for compressing a text message using a *variable-length encoding* called a *Huffman code*. Huffman coding assigns each symbol a unique bit sequence such that no code is a prefix of another—this is known as a *prefix code*. The algorithm uses the relative frequencies of symbols to generate the most efficient prefix code possible.

You'll be given a list of symbols and their corresponding frequencies. From this, you'll construct a binary tree known as a *Huffman Tree*, where each leaf represents a symbol and its position in the tree determines its encoding.

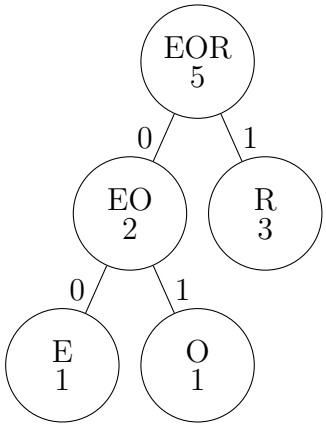
Your program will run in one of two modes. In one mode, it will print a table showing the binary encoding for each symbol. In the other (called specification mode), it will output a special traversal of the Huffman Tree, which can later be used to reconstruct the tree exactly.

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 min-heap that you'll need in order to find the correct Huffman Nodes to combine and create branching nodes.
- `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 contains a private class called `HuffmanNode`. You will be responsible for building the constructors, and implementing the compareTo() function. HuffmanNodes should be compared by frequency. However, because sometimes there are ties, ties should be broken by comparing the symbols String character-by-character (using the ascii code). Without that, the tree will still be optimal, but it will not match the test cases.

```
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 + ">";
    }
  }
```

There is another helper class inside `HuffmanTree` called `HuffmanHeapBuilder`. This is a class that's used initially by the main function in `HuffmanTree` and it should keep track of all of the symbols and frequencies received in an array before passing that array into a BinaryHeap. All of this should be done in the first phase of the main function where the frequency file is being ingested. Two functions need to be implemented in `HuffmanHeapBuilder`:

1. `void insertSymbol(String symbol, int frequency)`: Your main function just ingested a new symbol and its frequency from the input file. Keep track of it in an array inside `HuffmanHeapBuilder`, resizing if necessary!

2. `BinaryHeap<HuffmanNode> heapify()` The frequency file has finished loading. Now take all of the Huffman nodes that have been created, and make a `BinaryHeap` out of them. You could have been creating a `BinaryHeap` all along and updating it with the `insertSymbol` function, but by waiting until the end you can take advantage of the $O(n)$ buildHeap algorithm (using percolate down) implemented in `BinaryHeap.java`.

The rest of the `HuffmanTree` also has methods that need to be implemented.

```
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) {...}
}
```

Methods to be Implemented

HuffmanNode

- **public HuffmanNode(String symbol, double frequency)**
  Constructor that creates a leaf node from a single symbol and its frequency. Both `left` and `right` should be set to `null`.

- **public HuffmanNode(HuffmanNode left, HuffmanNode right)**
  Constructor that creates an internal node by combining two child nodes. The `symbols` field should be the concatenation of the symbols from the left and right nodes. The `frequency` field should be the sum of the two frequencies. The `left` and `right` fields should be assigned accordingly.

- **public int compareTo(HuffmanNode hn)**
  This method allows `HuffmanNode` objects to be inserted into a priority queue (min-heap) based on frequency. You must first compare the nodes by frequency. If the frequencies are equal, break ties by comparing the `symbols` strings in ASCII order (character by character). This ensures a consistent tree structure across implementations, which is important for automated testing.

- **public String toString()**
  This method is provided for you. It returns a string in the format `"<" + symbols + ", " + frequency + ">"`. This is useful for debugging and visualization.

HuffmanHeapBuilder

- **public void insertSymbol(String symbol, int frequency)**
  This method takes a symbol and its frequency as input. You should create a `HuffmanNode` for each symbol and store it in an internal array. You will not insert the nodes into a heap right away; instead, hold them in an array until all the symbols have been ingested. If more symbols are added than the initial capacity allows, you'll need to resize the array dynamically.

- **public BinaryHeap<HuffmanNode> heapify()**
  Once all symbols have been inserted, this method builds a min-heap from the stored nodes. Use the `BinaryHeap` constructor that takes an array and performs the $O(n)$ build-heap operation (also called "heapify" or "percolate down") to efficiently produce a min-heap of `HuffmanNode` objects.

HuffmanTree

- **public static HuffmanTree createFromHeap(BinaryHeap b)**
  Implements the Huffman algorithm (below). When only one element remains in the heap, it calls 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(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 method prints a representation of the Huffman tree to efficiently store it in a file. The printed format is called the "tree specification", which can later be used to reconstruct the original Huffman tree structure.

  The specification contains only symbols and pipe characters (`|`), and does not include frequencies. Symbols represent leaf nodes, and each pipe (`|`) represents an internal node formed by combining the two most recent nodes.

  To build the specification, perform a *post-order traversal* of the Huffman tree. For each leaf node, print its symbol using `convertSymbolToChar`. For each internal node, print a pipe character (`|`) **except** when the node lies on the rightmost path of the tree. The final trailing pipes are unnecessary and will be inferred later.

  The correct output for `sample_frequencies.txt` is provided in

  `sample_frequencies_correct_spec.txt`.

- **public static void main(String[] args)**
  This method is provided for you. It controls the overall program flow and integrates the components you've implemented.

  1. It reads the `mode` (either "legend" or "spec") from the command line. If no argument is given, the default is "spec".
  2. It reads symbol-frequency pairs from standard input using a buffered reader. Each line should contain a symbol and a frequency, separated by a tab.
  3. For each line, it calls `heapBuilder.insertSymbol()` to store the data.
  4. After all input has been read, it calls `heapBuilder.heapify()` to produce a min-heap of Huffman nodes.
  5. It calls `createFromHeap()` to build the Huffman tree from the heap.
  6. It constructs a `HuffmanTree` object using the root node.
  7. Depending on the mode, it calls either `tree.printLegend()` or `tree.printTreeSpec()` to print the output.

The Algorithm

The input is a list of characters and their corresponding frequencies. The output is a Huffman Tree, built 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

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
```

If you use an IDE, it will 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 main function will successfully read the file, and send each symbol and frequency to the `HuffmanHeapBuilder` class.

Testing Output

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
```

Remember you can use `fc` instead of `cmp` on windows machines.

You can also build a Huffman Tree using any document and calculating its character frequencies. Provided is a short poem called "This Is Just To Say" by William Carlos Williams. You can use the provided `FrequencyCounter` class to get a symbol frequency file from the text:

```
cat just_to_say.txt | java FrequencyCounter
```

You can then store it to a file and run it through your implementation of the `HuffmanTree` algorithm. Then you can compare it with the given legend:

```
cat just_to_say.txt | java FrequencyCounter > just_to_say_freq.txt
cat just_to_say_freq.txt | java HuffmanTree legend > my_just_to_say_legend.txt
cmp my_just_to_say_legend.txt just_to_say_legend.txt
```

You can also generate a legend all in one step by chaining 2 pipe commands! Try created a tree specification as well:

```
cat just_to_say.txt | java FrequencyCounter | java HuffmanTree legend
cat just_to_say.txt | java FrequencyCounter | java HuffmanTree
```

Submission

The entire submission should be a zip file containing all relevant files and folders and should be uploaded to Brightspace by the due date. It should be zipped up in file with your netid as the name. This includes:

- Your Java code, which should include one or more Java files and may or may not be organized into packages.
- Your `readme.txt` file (described below).

README

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

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.