Lab 11 Huffman Tree

Goal

In this lab you will use a binary tree to create a Huffman code for compressing data.

Resources

- Chapter 23: Trees
- `Encode.jar`—The working application
- Lab Manual Appendix—An Animation Framework

In `javadoc` directory

- *`BinaryTreeInterface.html`*—API documentation for the binary tree ADT
- *`HuffmanTree.html`*—API documentation for the class `HuffmanTree`
- *`HuffmanTreeInterface.html`*—API documentation for the Huffman tree ADT
- *`TreeInterface.html`*—API documentation for the tree ADT
- *`Code.html`*—API documentation for the class `Code`, which is a buffer containing the coded message
- *`Message.html`*—API documentation for the class `Message`, which is a buffer containing a message of type `Character`
- *`SymbolFrequencyPacket.html`*—API documentation for the class `SymbolFrequencyPacket`, which is the data in a node of a Huffman tree

Java Files

- *`Code.java`*
- *`EncodeActionThread.java`*
- *`EncodeApplication.java`*
- *`FindDefaultDirectory.java`*
- *`Message.java`*

There are other files used to animate the application. For a full description, see Appendix: An Animation Framework of this manual.

In `TreePackage` directory

- *`BinaryNode.java`*
- *`BinaryTree.java`*
- *`BinaryTreeInterface.java`*
- *`HuffmanTree.java`*
- *`HuffmanTreeInterface.java`*
- *`SymbolFrequencyPacket.java`*
- *`TreeInterface.java`*

Input Files

- *`message1.txt`*—A two line message to encode
- *`message2.txt`*—The same message as the first, but spaces have been added
- *`message3.txt`*—A four line message to encode
- *`message4.txt`*—A longer message to encode

Encoding the Message

Step 1. Read the file HuffmanTreeInterface.java inside the TreePackage folder and the method headers in Code.java.

Step 2. In the method `encodeCharacter()` in `EncodeActionThread`, add statements that will take a single character and encode it. You will add a "0" or "1" to the codeBuffer for each left or right branch, respectively, that was taken. Refer to the pre-lab presentation.

Step 3. After a character is added to the buffer, do an animation pause.

*Checkpoint: The application should run by typing **java EncodeApplication**. Use message1.txt for the text file. Step until the final code tree is displayed. Step once more. The first character in the message should be red. A "1" should appear in red in the code. The 22 (frequency of the right child of the root) in the code tree should appear in red to indicate the right branch was taken. Continue stepping and verify that the correct code is produced.*

Run the application for message2, message3, and message4. To change the input file, click Reset, type the file name and hit Enter. Record the final code tree for each on paper. Decode the results by hand and verify the results.

The trees for these texts may appear a bit strange since the space character does not draw. When highlighted, it will not show up either.

The tree for message4 will have nodes that overlap each other towards the bottom. This can be alleviated a bit by having the application use a bigger window. (Change the value of DISPLAY_WIDTH to be larger.) Since there are 10 levels to the tree, this will only be a minor improvement.

Post-Lab Follow-Ups

1. Modify the code to use a `List` instead of an array to hold the forest of Huffman trees. Use an iterator to find and remove the two smallest frequency trees.

2. Develop a format for representing a Huffman tree using a string of characters that is suitable to be written to a file. Add two methods to HuffmanTree. The method `writeTree()` will return a string that is the representation of the tree. The static method `parseTree(String)` will return a HuffmanTree equivalent to the representation in the String. Throw an exception if the string is not in the correct format. You might find it convenient to have some pattern of characters that marks the beginning and end of the representation as well.

3. Develop and implement a method that will decode a message using a `HuffmanTree`.