OO Software Engineering Assignment 1.

Engr 9859, 2013

Due Sept 23 @ 11:00PM.

For each question you will be marked on programming style as well as correctness. To see my opinion about what constitutes good programming style see

http://www.engr.mun.ca/~theo/Courses/ds/pub/style.pdf

In short:

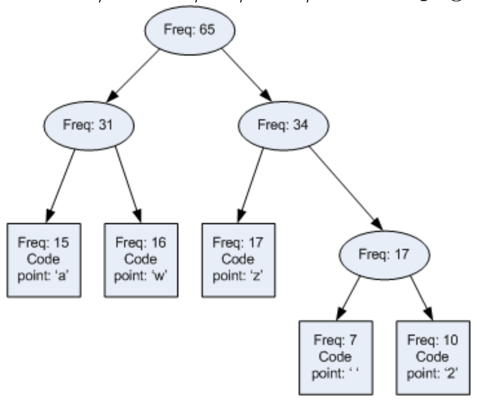
- All .java files must be professionally commented; in particular, each file should contain a comment header that gives your name and student number, and each class and subroutine should have a comment at the start of it. I encourage you to use the "javadoc" conventions for comments.
- If you use code, a data structure, or an algorithm you did not write or invent, give a citation for the source of the code, data structure or algorithm.
- Code and comments must be consistently indented; tab stops should be set every 4 characters.
- Names must be chosen carefully and spelled correctly. (Use names starting with lower case letters for variables and methods; use names starting with upper case letters for classes and interfaces.)
- Use subroutines appropriately to avoid redundant coding.
- Keep control structures and data structures simple.

All classes must be tested by you prior to being submitted. You are welcome to share test code with each other.

The assignment is to be done alone. Each file should contain the following declaration in comments near the top. "This file was prepared by [your name here]. It was completed by me alone.". If you obtained help in doing the assignment, do not include this declaration, but rather an explanation of the nature of any help that you received in doing the assignment.

Q0. This question continues the work on Huffman coding that was started in assignment 0. The idea is to build a tree structure representing the frequencies of various symbols. In this question, you will design and implement the classes that we will need to represent trees. In the next question, you will implement the algorithm to build the trees. A **tree** looks like this:

Folders/courses/cef/2013/huffman.png



You can see there are two kinds of **nodes** in this tree: **leaf** nodes (drawn as squares) and **branch** nodes (drawn as ovals). Branch nodes are said to have two **child** nodes. One node in the tree does not have a parent; it is called the **root**; all others have one **parent**.

A tree represents a binary code as follows. Each leaf node can be represented by a sequence of directions, left or right, from the root (top node). We take left=true and right=false, so the code in the example is

Code point	Code
'a'	[true, true]
'w'	[true, false]
'z'	[false,true]
' '	[false, false, true]
'2'	[false, false, false]

We will start with an interface

```
package huffman ;
public interface TreeNode {
    int getFrequency() ;

    BranchNode getParent() ;

    void setParent( BranchNode parent ) ;

    char decode( BooleanSource boolSource ) ;

    void encode( BooleanSink boolSink ) ;
}
```

You are to design and code two classes that implement this interface: The first represents leaf nodes and is called LeafNode. The second represents branch nodes and is called BranchNode.

The constructors are as follows

public LeafNode(**char** codePoint, **int** frequency) {...}
public BranchNode(TreeNode left, TreeNode right) { ... }

The left and right parameters represent the two "children" of the branch node — and so must not be **null**. When a branch node is constructed, it should set the parent of its two children to be itself. The following should be invariants

- Each branch node will have two children.
- Each tree node will have 0 or 1 parents.
- If a tree node n has a parent, it is a BranchNode and n should be a child of its parent.
- If a tree node has no parent, getParent should return **null**.

The two classes implement the methods of the TreeNode interface as follows.

- getFrequency. The leaf node returns the frequency given in its constructor argument. The branch node returns the sum of the frequencies of its two children.
- setParent. Changes the parent.
- getParent. The argument to the most recent call to setParent. If setParent has never been called, then getParent returns **null**, indicating the node is a root..
- decode. The boolSource parameter represents a sequence of boolean values. decode uses this sequence to go down the tree to find a symbol. For example if the source is

  [false, false, true, true, false]

  then the result of decode from the root of the example tree would be 3. Furthermore the first 3 items of the source would be removed so the source would then be

  [true, false]

  You can implement decode as follows: A leaf node returns its symbol. A branch node removes one boolean from the sequence and then requests either its left or right child to decode.
- encode. The boolSink parameter represents a sequence of boolean values which can be added to (on the right). When encode is called, the sequence of directions from the root of the tree down to this node is added to the sequence. For example, if we start with an empty sequence and call encode

on the leaf node for code point 2 in the example tree, then first false and then true will be added to the sequence to give

[false, true]

You can implement encode as follows: A node with no parent does nothing. A node with a parent requests its parent to encode itself, and then appends a true or false to the sequence depending on whether it is the left child or the right child of its parent.

In addition to the methods in the TreeNode interface, BranchNode should implement the following methods

public TreeNode getLeft() {...}
public TreeNode getRight() {...}

Which return references to the two children.

I will supply you with 'mock' implementations[1] of BooleanSource and BooleanSink and a few JUnit test cases.

Submit LeafNode.java and BranchNode.java.

Q1 Making a tree. Design and implement a class Tree in package huffman. Tree should have a constructor

public Tree(**int**[] frequencies, **char**[] symbols)

The frequencies array is a sequence of symbol frequencies. The symbols array indicates the corresponding symbol. I.e. frequencies[i] will be the frequency of character symbol[i], for each i. As a precondition, you should assume the two arrays have the same length and that all frequencies are non-negative. The frequencies and symbol arrays could be as follows:

symbols	frequencies
'z'	17
'w'	16
'a'	15
'2'	10
' '	7

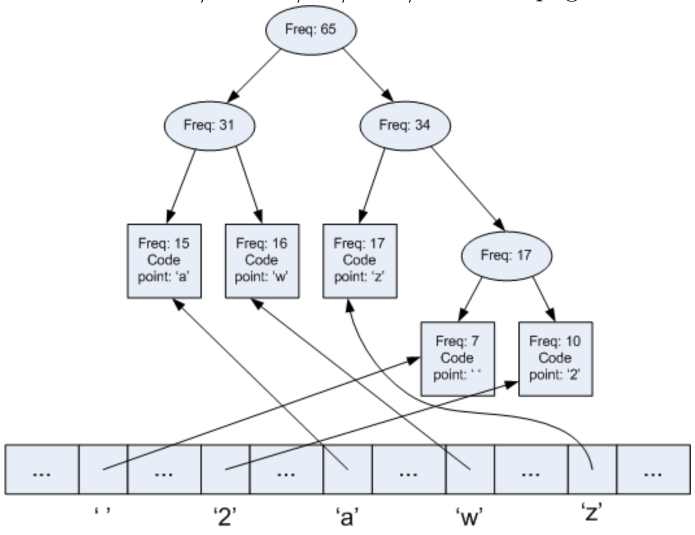
Frequencies array will be ordered from largest to smallest.

This constructor should:

- build an array of LeafNodes of length equal to 1 plus the maximum value in the symbol array converted to an int. (In the example the maximum character is 'd', so the size of the array would be (int)'d' + 1.Each LeafNode will have a frequency from the frequency array and a code point equal to the position in the array. In the example, item (int)'d' of the array would be a reference to a LeafNode that has 'd' as its code point and 1 as its

[1] A **mock** class is an implementation of an interface used in testing other classes.

Folders/courses/cef/2013/huffman1.png



frequency. All other items of the array should be null. When the frequency is 0, no leaf node should be created.

- The constructor should then build a tree over these leaves according to Huffman coding (see below). The constructor should store references to the root node of the tree and to the array in private fields of Tree for later use.

The Tree class should also have methods

public void encode(**char** symbol, BooleanSink sink)

and

public int decode(BooleanSource source)

The encode method will find the LeafNode corresponding to the symbol and then append the symbol's code to the sink object. As a precondition, the symbol must be one that was in the constructor's symbols argument. I.e., it must be one that has a LeafNode in the array.

The decode method will decode the next code point from the BooleanSource using the tree and return the code point. You may assume, as a precondition, that the source can supply enough bits.

Huffman coding.

You can find many discussions of Huffman's method of building the tree either on the web or in the library. For example the Wikipedia has a discussion. Be sure to cite any sources you use. In short here is the method:

Keep a sequence of nodes ordered by frequency. Initially the

> sequence contains just the leaf nodes. When the sequence has only one node in it, we stop; that node is the root. As long as there are two or more nodes in the sequence, we remove the two with the lowest frequencies, build a branch node above them, and then add the branch node to the sequence.

You are welcome to use classes and interfaces from the Java standard library. For example, you might use the java.util.ArrayList class, java.util.LinkedList, or the java.util.PriorityQueue classes. Using the java.util.PriorityQueue class gives an easy way to achieve $O(N \log N)$ time to build the tree, where N is the length of the frequency array. A way to achieve $O(N)$ time is use two FIFO (first in, first out) queues, one for leaves and one for branches;[2] since branch nodes are built in order of increasing frequency, you can always add new branch nodes to the back of the branch node queue.

Huffman coding guarantees that the code is a most efficient code possible in the sense that it minimizes

$$\sum_{i \in \{0,..N\}} \mathit{freq}(i) \times |\mathit{code}(i)|$$

where $\mathit{freq}(i)$ is the frequency of code point i and $\mathit{code}(i)$ is the length of the code word assigned to code point i.

Submit `Tree.java`.

[2] java.util.LinkedList implements a FIFO queue for the branches.