Programming a Huffman Encoder and Decoder

We're going to break this assignment into parts. There are 8 parts. Below is a schedule for when each part is due as well as a solution to the first part.

Set Up the Huffman Encoder Class	October 26
Create Symbol Table	October 28
Create Node List	October 31
Create Huffman Tree	Nov 2
Create Huffman Code Map	Nov 4
Write Serialized Code Map to File (used by decoder)	Nov 7
Encode File with Code Map	Nov 9
Decode File with Serialized Code Map	Nov 11

Step 1 : Set Up the Huffman Encoder Class

Create a **HuffmanEncoder** class. There is only one method in the class that we need to call, so there is no need to create instances of this **HuffmanEncoder**. The name should be short and simple, yet descriptive. We'll call our public function **encode()**. The driver will call **encode()** as follows.

```
public class Driver {
        public static void main(String[] args)
        {
                HuffmanEncoder.encode("original.txt", "encoding.txt", "code.serialized");
                System.out.println("Finished encoding file.");
        }
}
```

In order to allow **encode()** to be called like this, **all** methods in **HuffmanEncoder** must be static.

Now lets set up the HuffmanEncoder class. I've created a public static method named **encode()** and included below it comments detailing the steps in the encoding process. These comments are formed by decomposing the algorithm discussed in class.

For each step of the algorithm described in the comments we create a function name that describes the step. We include the name at the beginning of each comment. We'll use the following names for our functions.

After deconstructing the algorithm and adding the comments to the **HuffmanEncoder** class, the class should look like the one below.

```
public class HuffmanEncoder {

	public static void encode(String inFileName, String outFileName, String codeFile)
	{
	}

	/************************* createSymbolTable() ************************
	* Construct a HashMap (char symbol, int frequency) of key,value pairs.
	* For each symbols in a file, if the symbol has not been seen before
	* add it to the HashMap and set its frequency to 1, otherwise increment
	* the symbols frequency in the HashMap.
	**********************************************************************/
	/************************** createNodeList() **************************
	* Create a List of BinaryNodes where each Node contains a symbol and a
	* frequency field.  For each (symbol, frequency) pair in the HashMap add a
	* new BinaryNode to the List while keeping the list sorted in increasing order
	* according to frequency.
	**********************************************************************/
	/************************** createHuffmanTree() ***********************
	* While the List has more than 1 BinaryNode, create a new BinaryNode
	* with frequency equal to the sum of the frequencies of the top two Nodes in
	* the list. Set the left child equal to the top Node and remove the top Node from
	* the list. Then set the right child equal to the new top Node and remove the top
	* Node from the list. Insert the new Node into the sorted List while
	* maintaining its sortedness.
	**********************************************************************/
	/************************ createHuffmanCodeMap() **********************
	* Traverse the Huffman Tree using a Depth First Search while maintaining a
	* String that represents the path from the root. If descending a left branch
	* concatenate a '0' to the string, if descending a right branch concatenate
	* a '1' to the string, and if ascending remove the last character. When you
	* visit a leaf, add (symbol,code) to the HashMap where symbol is the
	* symbol stored in the leaf Node and code is the String that was constructed
	* while traversing the Tree.
	**********************************************************************/
	/***************** writeSerializedHuffmanCodeToFile() *****************
	* Write the Huffman code HashMap as a serialized object to an output file
	**********************************************************************/
	/*************************** encodeFile() *****************************
	* While reading each symbol of the input file, output (in binary) the code
	* associated with the symbol that is stored the code HashMap.
	**********************************************************************/
}
```

Each of the comments describes a function that must be performed in the process of encoding a file with the file's Huffman code. In **encode(),** determine what each function needs as arguments and determine what each function returns. Lets discuss the parameters and return types for each function.

We know **createSymbolTable()** reads a file and creates a Map that maps the symbols in the file to their frequency so we need to pass to the function the name of the file. Whitespace is better represented in this map as an integer because when we debug it we can print (and see) the integer value of every symbol but not the character itself. The frequency is an integer. Therefore the map should map integers to integers.

CreateNodeList() takes the symbol table as an argument and returns a linked list of binary nodes that is in sorted order according to frequency. Since the Java LinkedList class has no sorting mechanism, we'll create our own linked list class that extends LinkedList and call it **HuffmanLinkedList**. In our class we'll overwrite the add() method so that when nodes are added they're added in sorted order.

We need to pass the linked list to **createHuffmanTree()** since it creates a Huffman tree by repeatedly combining the top two nodes of the linked list into a new node and reinserting the new node into the list. The remaining element in the list is a binary node that is also a binary Huffman tree. We'll return that remaining element in the list so that it can be used to create the Huffman code map. Since some of the binary nodes in the tree need to contain the symbol and frequency we'll create a **HuffmanData** class to hold this data and store **HuffmanData** objects in each of the binary nodes. Therefore the node we return will be a **BinaryNode<HuffmanData>** object.

CreateHuffmanCodeMap() takes the root of the Huffman tree as an argument and returns a Map object that maps the symbols (integers) to Strings of 0's and 1's.

WriteSerializedHuffmanCodeToFile() takes both the name of the file we want to write the serialized code map to, as well as the Huffman code map.

To encode the file in **encodeFile()**, we need the input file and output file names as well as the Huffman code map.

The **HuffmanEncoder** class's **encode()** method at this point looks like this.

```
public class HuffmanEncoder {

	public static void
	encode(String inFileName, String outFileName, String codeFile)
	{
		Map<Integer, Integer> symbolTable = createSymbolTable(inFileName);
		HuffmanLinkedList nodeList = createNodeList(symbolTable);
		BinaryNode<HuffmanData> rootNode = createHuffmanTree(nodeList);
		Map<Integer, String> huffmanCodeMap = createHuffmanCodeMap(rootNode);
		writeSerializedHuffmanCodeToFile(codeFile, huffmanCodeMap);
		encodeFile(inFileName, outFileName, huffmanCodeMap);
		return;
	}
	...
}
```

At this point, the **encode()** method is complete.

Now we create functions in the **HuffmanEncoder** class for each of the functions that are called in **encode()**. The code below does not have any comments in it, but you should keep the comments from above in your code. You will use them when writing the body of each of your functions. In addition, they will help future maintainers of your code.

```
public class HuffmanEncoder {

	public static void
	encode(String inFileName, String outFileName, String codeFile)
	{
		Map<Integer, Integer> symbolTable = createSymbolTable(inFileName);
		HuffmanLinkedList nodeList = createNodeList(symbolTable);
		BinaryNode<HuffmanData> rootNode = createHuffmanTree(nodeList);
		Map<Integer, String> huffmanCodeMap = createHuffmanCodeMap(rootNode);
		writeSerializedHuffmanCodeToFile(codeFile, huffmanCodeMap);
		encodeFile(inFileName, outFileName, huffmanCodeMap);
		return;
	}

	private static Map<Integer, Integer>
	createSymbolTable(String file) { }

	private static HuffmanLinkedList
	createNodeList(Map<Integer, Integer> map) { }

	private static BinaryNode<HuffmanData>
	createHuffmanTree(HuffmanLinkedList nodeList) { }

	private static Map<Integer, String>
	createHuffmanCodeMap(BinaryNode<HuffmanData> rootNode) { }

	private static void
	setMapWithDepthFirstSearch(Map<Integer, String> huffmanCodeMap,
		BinaryNode<HuffmanData> curNode, String code) { }

	private static void
	writeSerializedHuffmanCodeToFile(String fileName,
		Map<Integer, String> huffmanCodeMap) { }

	private static void
	encodeFile(String inFileName, String outFileName, Map<Integer, String> huffmanCodeMap) { }
}
```

As we develop the body of our functions we want to be able to test our code. As it stands, this class will not compile for 4 reasons:

1. **HuffmanData**, **HuffmanLinkedList** and **BinaryNode** are refrenced but do not yet exist in our namespace.
2. We haven't specified any import statements in HuffmanEncoder that will link the standard libraries to our code.
3. Some of our functions return objects that have yet to be instantiated.

To fix these issues, we first need to create classes for **HuffmanData** and **HuffmanLinkedList**. You need only include the constructors in each class to get a project that will compile. Over time we'll add functionality to them.

The **HuffmanLinkedList's** constructor should simply extend the LinkedList class and call the LinkedList's constructor.

```
import java.util.LinkedList;

@SuppressWarnings("serial")
public class HuffmanLinkedList extends LinkedList <BinaryNode<HuffmanData>>{

	public HuffmanLinkedList() {
		super();
	}
}
```

The **HuffmanData** class is going to store symbols and frequency so we might as well set this up now. Lets add private fields, getter methods and override the toString() method.

```
public class HuffmanData {
	private int symbol;
	private int frequency;

	public HuffmanData(int symbol, int frequency) {
		this.symbol = symbol;
		this.frequency = frequency;
	}

	public int getSymbol() {
		return symbol;
	}

	public int getFrequency() {
		return frequency;
	}

	@Override
	public String toString() {
		return symbol + "-" + frequency;
	}
}
```

Next we need to import the necessary libraries using import statements in **HuffmanEncode**r. At this point we only need the Map class to get our class to compile.

```
import java.util.Map;

public class HuffmanEncoder {

        ...
```

Now, lets add the **BinaryNode** class. The **BinaryNode** class implements the **BinaryNodeInterface** so we need to include both of these class files. I've uploaded them to GitHub. Include these in your project.

Last, we need to supply the **HuffmanEncoder** functions with return statements. If a functions return void, we simply add **return;** in the function body. On the other hand, if they return an object we add **return null;** to the body.

At this point the project should compile and run. When running, it should print out *Finished encoding file*.

Next Step:

Fill in the body of the createSymbolTable() function.

Due Wednesday, October 28

On Wednesday, we'll discuss a solution to this problem.