CSE 143, Spring 2014
Programming Assignment #8: Huffman Coding (40 points)
Due Thursday, June 5th, 2014, 11:30 PM

This program provides practice with binary trees and priority queues. Turn in files named `HuffmanTree.java`, `secretmessage.short`, and `secretmessage.code` from the Homework section of the web site. You will need support files `MakeCode.java`, `Encode.java`, `Decode.java`, `Bit*.java`, and input files from the course web page.

Huffman Coding:

Huffman coding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing text data to make a file occupy a smaller number of bytes. This relatively simple compression algorithm is powerful enough that variations of it are still used today in computer networks, fax machines, modems, HDTV, and other areas.

Normally text data is stored in a standard format of 8 bits per character, commonly using an encoding called ASCII that maps every character to a binary integer value from 0-255. The idea of Huffman coding is to abandon the rigid 8-bits-per-character requirement and use different-length binary encodings for different characters. The advantage of doing this is that if a character occurs frequently in the file, such as the letter `'e'`, it could be given a shorter encoding (fewer bits), making the file smaller. The tradeoff is that some characters may need to use encodings that are longer than 8 bits, but this is reserved for characters that occur infrequently, so the extra cost is worth it.

The table below compares ASCII values of various characters to possible Huffman encodings for the text of Shakespeare's *Hamlet*. Frequent characters such as space and `'e'` have short encodings, while rarer ones like `'z'` have longer ones.

Character	ASCII value	ASCII (binary)	Huffman (binary)
`' '`	32	00100000	10
`'a'`	97	01100001	0001
`'b'`	98	01100010	0111010
`'c'`	99	01100011	001100
`'e'`	101	01100101	1100
`'z'`	122	01111010	00100011010

The steps involved in Huffman coding a given text source file into a destination compressed file are the following:

1. Examine the source file's contents and count the number of occurrences of each character.
2. Place each character and its frequency (count of occurrences) into a sorted "priority" queue.
3. Convert the contents of this priority queue into a binary tree with a particular structure.
4. Traverse the tree to discover the binary encodings of each character.
5. Re-examine the source file's contents, and for each character, output the encoded binary version of that character to the destination file (done for you by `Encode.java`).

Making a Code:

For example, suppose we have a file named `example.txt` with the following contents:

```
ab ab cab
```

byte	1	2	3	4	5	6	7	8	9
char	`'a'`	`'b'`	`' '`	`'a'`	`'b'`	`' '`	`'c'`	`'a'`	`'b'`
ASCII	97	98	32	97	98	32	99	97	98
binary	01100001	01100010	00100000	01100001	01100010	00100000	01100011	01100001	01100010

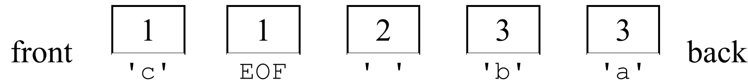
In Step 1 of Huffman's algorithm, a count of each character is computed. This is done by the `MakeCode.java` program. This program prompts the user for a file and computes the frequency of each character in it. These counts are stored in an integer array with 256 values representing each of the ASCII values:

index	0	1	...	32	...	97	98	99	100	...
value	0	0		2		3	3	1	0	

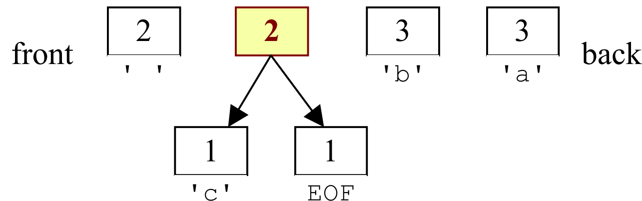
The `MakeCode` program passes this array of counts into the constructor of your `HuffmanTree` class. Your program should not depend on the array having 256 values but should instead use the length field of the array to know how many different possible characters there are.

Step 2 of the algorithm places these counts into binary tree nodes, each storing a character and a count of its occurrences. The nodes are put into a priority queue, which keeps them in sorted order with smaller counts at the front of the queue.

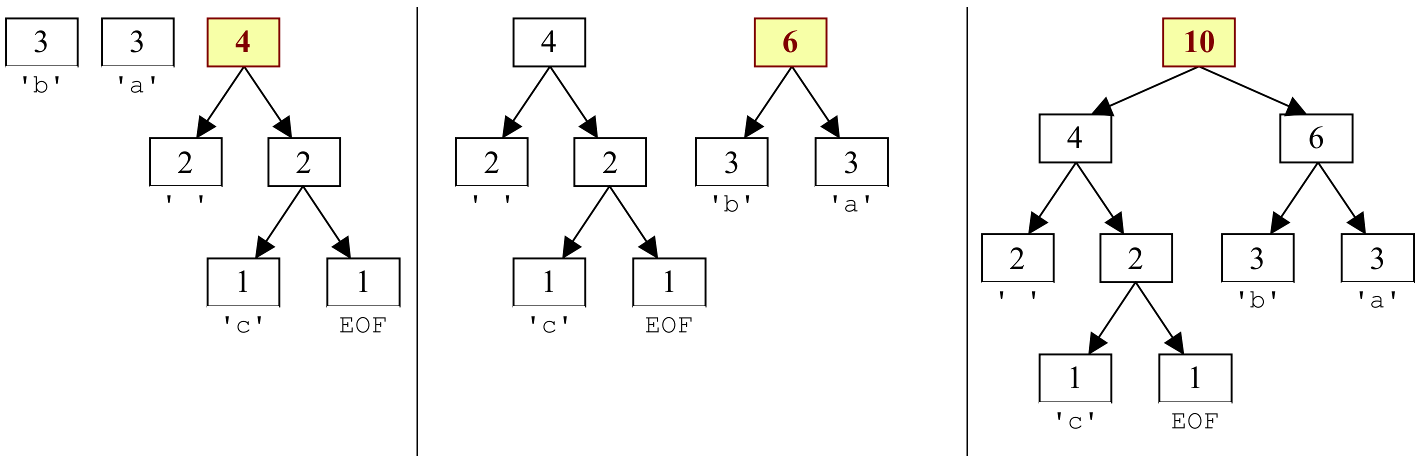
You will need to introduce an "end of file" (EOF) character that will be used to tell our Huffman encoding and decoding code that a file is ending. The value of this character will be one greater than the value of the highest character in the frequency array passed to the constructor and will have a frequency of 1 since it only appears once at the end of each file. This character will not be part of the frequency array so you will have to manually add it to the priority queue. (The priority queue is somewhat arbitrary in how it breaks ties, such as `'c'` being before EOF and `'b'` being before `'a'`).



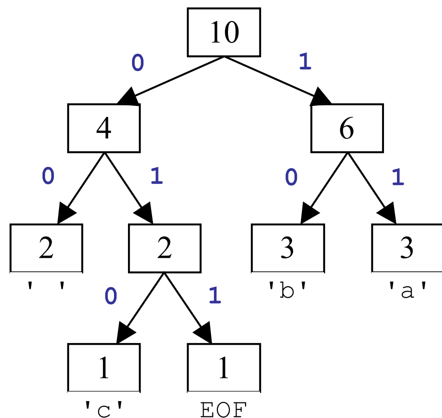
Now the algorithm repeatedly removes the two nodes from the front of the queue (the two with the smallest frequencies) and joins them into a new node whose frequency is their sum. The two nodes are placed as children of the new node; the first removed becomes the left child, and the second the right. The new node is re-inserted into the queue in sorted order:



This process is repeated until the queue contains only one binary tree node with all the others as its children. This will be the root of our finished Huffman tree. The following diagram shows this process:



Notice that the nodes with low frequencies end up far down in the tree, and nodes with high frequencies end up near the root of the tree. This structure can be used to create an efficient encoding. The Huffman code is derived from this tree by thinking of each left branch as a bit value of 0 and each right branch as a bit value of 1:




```
32
00
99
010
256
011
98
10
97
11
```

`example.code`
(output from `MakeCode.java`)

The code for each character can be determined by traversing the tree. To reach 'a' we go right twice from the root, so the code for 'a' is 11. The box at the right above shows the output produced when `MakeCode.java` is run with the `example.txt` file from page 1. This calls your `write` method which should write out each character's code. As with 20 Questions, we will use a standard format for Huffman counts files. The output should contain a sequence of line pairs where each pair represents one leaf of the tree. The first line of each pair should contain the ASCII value of the character stored in that leaf. The second line should have the code (0s and 1s) for the character with that ASCII value. The codes should appear in "traversal order" – the order that any standard traversal would visit the leaves in.

Encoding a File:

`Encode.java` uses a counts file created by `MakeCode.java` and the original text file to produce a binary file that is the compressed version of the original. This is a complete, working program that doesn't need your `HuffmanTree` class.

Using this class, the text `ab ab cab` would be encoded as:

char	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	EOF
binary	11	10	00	11	10	00	010	11	10	011

The encoded contents of the file are `1110001110000101110011`, which is 22 bits, or just under 3 bytes, compared to the original file which was 9 bytes. (Many Huffman-encoded text files compress to about half their original size.)

byte	1	2	3
char	a b a	b c a	b EOF
binary	11 10 00 11	10 00 010 1	1 10 011 00

Since the character encodings have different lengths, often the length of a Huffman-encoded file does not come out to an exact multiple of 8 bits. Files are stored as sequences of whole bytes, so in cases like this the remaining digits of the last bit are filled with 0s. You do not need to worry about this in the assignment; it is part of the underlying file system.

It might worry you that the characters are stored without any delimiters between them, since their encodings can be different lengths and characters can cross byte boundaries, as with 'a' at the end of the second byte above. But this will not cause problems in decoding the compressed file, because Huffman encodings have a prefix property where no character's encoding can ever occur as the start of another's encoding. This is important for decoding the file.

Decoding a File:

`Decode.java` uses your `HuffmanTree` class to convert a compressed file back to a standard ASCII file. The first step in decoding a file is to build up a `HuffmanTree` from a code file. This will be done by a constructor that takes in a `Scanner` representing a tree stored in standard format. In this case, the frequencies are irrelevant because the tree was already built using the constructor that takes in counts so you should set all the frequencies to some standard value such as 0 or -1.

Remember that the standard code file format is a series of pairs of lines where the first line has an integer representing the character's ASCII value and the second line has the code to use for that character. You might be tempted to call `nextInt()` to read the integer and `nextLine()` to read the code, but remember that mixing token-based reading and line-based reading is not simple. Here is an alternative that uses a method called `parseInt` in the `Integer` class that allows you to use two successive calls on `nextLine()`:

```
int n = Integer.parseInt(input.nextLine());
String code = input.nextLine();
```

To decode a compressed file, you will need to read it bit by bit. We have supplied you with a `BitInputStream` class that will help you do this. To read a bit from an input file, call the following `BitInputStream` method:

```
public int readBit()
```

The `decode` method will do the reverse of the encoding process. It will need to read sequences of bits that represent encoded characters and figure out what the original characters were. Start at the top of the tree and read bits from the input stream, going left or right depending on whether the stream returns a 0 or 1. A leaf node indicates the end of an encoded sequence. When you reach a leaf node, write the integer code for that character to the output file by calling the following method from the `PrintStream` class:

```
public void write(int b)
```

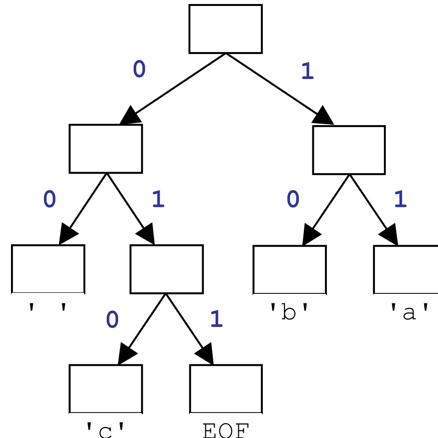
After writing a character's integer (the value between 0 and 255 stored in the leaf), go back to the top and start over, reading more bits and descending the tree until reaching a leaf. This process continues until you reach the EOF character. The EOF character should not be written to the `PrintStream` because it wasn't part of the original file.

If you fail to recognize the EOF character, you might end up accidentally reading past the end of the bit stream. When that happens, the `readBit` method returns a value of -1. So if you see a value of -1 appearing, it's because you've read too far in the bit stream.

Suppose we are asked to decode a file containing the following bits:

```
1011010001101011011
```

Using the Huffman tree, we walk from the root until we find characters, then we output them and go back to the root.



- First we read a `1` (right), then a `0` (left). We reach `'b'` and output `b`. Back to the root. `1011010001101011011`
- We read a `1` (right), then a `1` (right). We reach `'a'` and output `a`. `1011010001101011011`
- We read a `0` (left), then a `1` (right), then a `0` (left). We reach `'c'` and output `c`. `1011010001101011011`
- We read a `0` (left), then a `0` (left). We reach `' '` and output a space. `1011010001101011011`
- We read a `1` (right), then a `1` (right). We reach `'a'` and output `a`. `1011010001101011011`
- We read a `0` (left), then a `1` (right), then a `0` (left). We reach `'c'` and output `c`. `1011010001101011011`
- We read a `1` (right), then a `1` (right). We reach `'a'` and output `a`. `1011010001101011011`
- We read a `0` (left), then a `1` (right), then a 1 (right). We reach the EOF and stop. `1011010001101011011`

The overall decoded text is `bac aca`.

Implementation Details:

In this assignment you will create a class `HuffmanTree` to represent the overall tree of character frequencies drawn on the previous pages. You will also create a private inner class `HuffmanNode` to store information about one character. The contents of the `HuffmanNode` class are up to you, but it should not perform a large share of the overall algorithm.

Your `HuffmanTree` class must have the following public constructor and methods:

`public HuffmanTree(int[] counts)`
This constructor is passed an array of frequencies where `count[i]` is the count of the character with ASCII value i. Use these counts to build your Huffman tree using a priority queue (`PriorityQueue`) as previously described.

`public void write(PrintStream output)`
Write the tree to the given output stream in the standard format described at the top of page 3.

`public HuffmanTree(Scanner input)`
This constructor reconstructs a tree from a code file. Assume that the `Scanner` contains a tree stored in standard format.

`public void decode(BitInputStream input, PrintStream output, int eof)`
This method should read individual bits from the input stream and should write the corresponding characters to the output. It should stop reading when it encounters a character with value equal to the eof parameter. Assume that the input stream contains a legal encoding of characters for this tree's Huffman code.

You may have additional methods, so long as they are `private`.

Methods that traverse your tree should be implemented recursively whenever practical. You will have to be careful if you use recursion in your decode method. Java has a limit on the stack depth you can use. For a file like `hamlet.txt`, there are hundreds of thousands of characters to decode. That means it would not be appropriate to write code that requires a stack that is hundreds of thousands of levels deep. You might be forced to write some or all of this using loops to make sure that you don't exceed the stack depth.

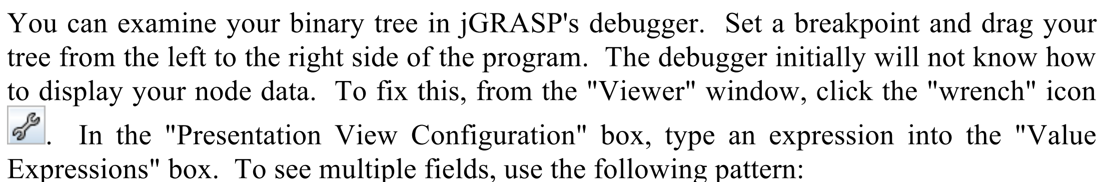
We will not test decompressing a completely empty file.

Development Strategy and Hints:

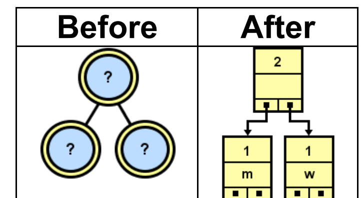
We suggest that you first focus on building your Huffman tree properly from the given array of character counts. Then work on writing the encodings to a file and lastly work on decompressing a file.

For your nodes to be able to be stored in a priority queue, the queue needs to know how to sort them. Therefore your node class must implement the `Comparable<E>` interface as discussed in lecture and section. Nodes should be compared by character frequency, where a character that occurs fewer times is "less than" one that occurs more often. If two nodes have the same number of occurrences, they are considered "equal" in this context.

Consider writing a `toString` method in your `HuffmanNode` so you can easily print nodes or priority queues of them. Note that if you `println` a priority queue, it does not necessarily show its elements in the order they would be returned.



You can examine your binary tree in jGRASP's debugger. Set a breakpoint and drag your tree from the left to the right side of the program. The debugger initially will not know how to display your node data. To fix this, from the "Viewer" window, click the "wrench" icon. In the "Presentation View Configuration" box, type an expression into the "Value Expressions" box. To see multiple fields, use the following pattern:



`_node_.`**`field1`**`#_node_.`**`field2`**

It can be difficult to tell whether a file was compressed properly. If you open a Huffman-compressed binary file in a text editor, the appearance will be gibberish (because the text editor will try to interpret the bytes as ASCII encodings, which is not the way the data is stored). While developing your program, it can be helpful to write out each 0 or 1 as an entire character (byte) rather than as a bit. This defeats the purpose of compression, because the "compressed" file is actually larger than the original, but it can help you see whether the 0s and 1s are what you expect.

To write out your 0s and 1s as entire bytes instead of as bits, you can simply set the `DEBUG` flag to true at the top of the `Encode` client. Note that you will not be able to decode a file written in debug mode with the `Decode` client.

We suggest you start with a very small input file such as the example shown in this document (`example.txt`), and work your way up to larger files once that works.

Creative Aspect (`secretmessage.short` and `secretmessage.code`):

Along with your program you should turn in files named `secretmessage.short` and `secretmessage.code` that represent a "secret" compressed message from you to your TA, and its code file. The message can be anything you want, as long as it is not offensive. Your TA will decompress your message with your tree and read it while grading.

Style Guidelines and Grading:

Part of your grade will come from appropriately utilizing binary trees and recursion to implement your Huffman tree as described previously. We will also grade on the elegance of your recursive algorithms; don't create special cases in your recursive code if they are not necessary or repeat cases already handled. Redundancy is another major grading focus; some methods are similar in behavior or based off of each other's behavior. You should avoid repeated logic as much as possible. Your class may have other methods besides those specified, but any other methods you add should be `private`.

You should follow good general style guidelines such as: making fields `private` and avoiding unnecessary fields; declaring collection variables using interface types; appropriately using control structures like loops and `if/else`; properly using indentation, good variable names and types; and not having any lines of code longer than 100 characters.

Comment your code descriptively in your own words at the top of your class, each method, and on complex sections of your code. Comments should explain each method's behavior, parameters, return, pre/post-conditions, and exceptions. For reference, our `HuffmanTree` class is around 115 lines long (70 "substantive") including comments and blank lines.