Huffman Coding

1. Read the "Intro to Compression and Huffman Coding" article (yes, all of it – maybe twice) in the "info" folder. This *startlingly good* article explains character encoding and compression algorithm basics, in addition to the Huffman coding algorithm you will use. From here you have two options:

    a. Given what you learned from the article, write a program that is able to compress and decompress text files using the Huffman encoding scheme. Starting with a blank page and ending with a functioning program can be really satisfying. But wait! Before you start, make sure you read #2 below.

    b. If you need a little more direction, continue reading; a suggested API is provided. There are many ways this program can be written, feel free to make changes to the API as you see fit.

2. There are more resources provided that will help:

    a. The "Huffman Hints" document in Canvas has some hints from me in no particular order.

    b. The video link on Canvas is a nice demonstration of the Huffman tree building process. It's strongly suggested you check it out, whether or not you're using the starter code.

    c. The provided BitInputStream/BitOutputStream helper classes are used to read and write individual *bits* (i.e. 0 or 1, not a `char`/`int`/String) to and from a file.

        i. For the recklessly curious, some info on bit manipulation can be found along with the bit I/O classes. This info isn't required to complete this project. For an **(Advanced)** style challenge, write the bit I/O classes yourself.

    d. The provided TreePrinter class will print a "tree-like" diagram, when supplied the overall root.

The Assignment (Break up into stages: 1: build tree 2: write key file 3: encode)

Phase I: Build Tree

Begin by defining a class **HuffmanTree.java** that represents a Huffman encoding tree. Add the following methods (more methods will be added later):

Method	Description
`HuffmanTree(int[] count)`	This method will construct the Huffman tree using the given array of character frequencies, where `count[i]` is the number of occurrences of the character with integer (decimal) value `i`. For example, `count[32]` represents the number of spaces.
`void write(String fileName)`	This method should write your encoding tree to the given file in a standard format, using the naming conventions provided later.

The array passed to HuffmanTree's constructor should have exactly 256 elements, but your program should not depend on this. Instead, use the `length` field of the array to know how many there are. You should use a priority queue to build up the tree as described in the how-to article. First you will add a leaf node for each character that has a frequency greater than 0 (we don't include characters not in the input source in our tree). These should be added in increasing character (decimal) order (character 0, character 1, and so on).

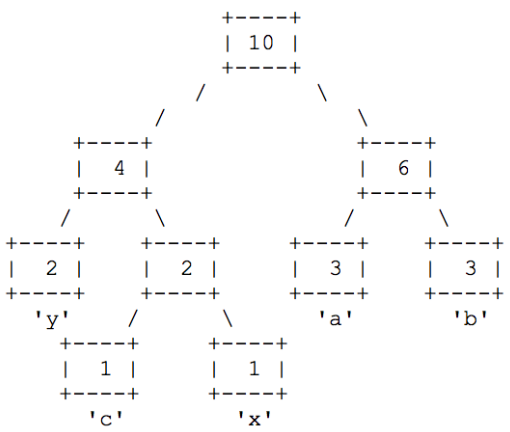
You will need to define a Node class. You should decide what data fields are appropriate to include in the node class. It's suggested you put this class in the same file as HuffmanTree, but at the top level (not as a private *inner* class) for easier testing. **Note:** Node should not be `public`, as you can only have one public top-level class per file.

When building the Huffman tree using a `java.util.PriorityQueue<E>`, you'll be adding values of type Node. This means that your node class will have to implement the `Comparable<T>` interface, the contract for types that are "order-able". It should use the frequency (weight) of the sub-tree to determine its ordering relative to other sub-trees, with lower frequencies considered "less" than higher frequencies.

The provided TreePrinter class will print your Huffman tree in a nice way, when debugging. When overriding the `toString` method of your node class, recall that only the frequencies matter for interior nodes, while the leaf nodes will store actual characters.

Phase II: Write Code Key File

As with the Twenty Questions program, we will use a standard format for outputting the Huffman tree. The output should contain a sequence of line pairs, one for each leaf of the tree. The first line of each pair should have the integer value of the character stored in that leaf (rather than the corresponding character, which may be something like a newline character and therefore not visible). The second line should have the code (a string of 0's and 1's) for the character with this integer value.



For the example above (five letters a, b, c, x, y with frequencies 3, 3, 1, 1, 2, respectively), the file output would be as follows (note: the letter 'a' has integer value 97, character 'y' is 121):

```
121
00
99
010
120
011
97
10
98
11
```

As mentioned in the intro document, Huffman coding works best if one character is designated as "end of file," meaning that every file is guaranteed to end with such a character and it will be used for no other purpose. Some operating systems have such a character, but if we want to write a general-purpose program, we should do something that is not specific to any one operating system.

In addition to encoding the actual characters that appear in the file, we will create a code for a fictitious end-of-file character that will be used only by the Huffman encoding and decoding programs. The pseudo-EOF character should be one higher than the value of the highest character in the frequency array passed to the constructor (256 in this case, as ASCII values of 0-255 are assumed). It will always have a frequency of one because it appears exactly once at the end of each file to be encoded. You will have to manually add this character to your priority queue because it will not be included as part of the frequency array.

The output listed above does not include the pseudo-EOF character. When you include the pseudo-EOF character with a frequency of one, the output becomes:

```
121
00
256
010
99
0110
120
0111
97
10
98
11
```

Recall that a Huffman solution is not unique. You can obtain any one of several different equivalent trees depending upon how certain decisions are made. However, if you use Java's PriorityQueue class, thus letting it handle the decision of which node to choose when two nodes have the same frequency, the tree should at least be consistent.

Phase III: Encode File

There are two main parts to this assignment. The first is the HuffmanTree class, used to build a variable bit length Huffman encoding tree. This class is also responsible for producing the "code file", a file that contains the codes used to compress some input source. The second is a class that will handle encoding and decoding input text given a particular Huffman coding tree. To encode/decode from a file, you will first have to first add some new methods to your HuffmanTree class:

Method	Description
`HuffmanTree(String codeFile)`	This constructor will reconstruct the tree from a file. You can assume that the file contains a tree stored in standard format. Note that when reconstructing the tree from the file, the character frequencies no longer matter.
`void decode(BitInputStream in, String outFile)`	This method will decode the stream of bits (0's and 1's) supplied and output the corresponding characters to the supplied file, given the naming conventions later in this document.

Previously you wrote a constructor that took an array of frequencies and constructed an appropriate tree given those frequencies. This new constructor is passed a file representing the *product* of your `write` method from before. For this second part, the frequencies are irrelevant because the tree has already been constructed once; however, you are likely using the same node class as before. You can set all the frequencies to some standard value like 0 or -1 for this part. *Follow your heart.*

Remember that the agreed-upon format for the code file was a series of pairs of lines where the first line has an integer representing the character's integer 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 can be problematic. Here is an alternative that uses the `Integer.parseInt` method, that allows you to use two calls on `nextLine`:

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

Phase IV: Decode File

For the decoding part, you read a BitInputStream. This utility class that works in conjunction with another class called BitOutputStream. They each have a very simple interface and allow you to write and read compact sequences of individual bits.

The only non-constructor method you'll need to use from BitInputStream is the following:

```
public int readBit() //returns a bit from the file, as an integer
```

The `decode` method is doing the reverse of the encoding process. It is reading (variable-length) sequences of bits that represent encoded characters and it is figuring out what the original characters must have been. Your method should start at the top of your tree and should read bits from the input stream, going left

or right depending upon whether you get a 0 or 1 from the stream. When you hit a leaf node, you know you've found the end of an encoded sequence. At that point, you should write the character to the output file.

Once you've written this character, you go back to the top of your tree and start over, reading more bits and descending the tree until you hit a leaf again. At that point you write again, go back to the top of the tree, read more bits and descend until you hit a leaf, then write the leaf, go back to the top of the tree, and so on.

Remember that we introduced a pseudo-EOF character with a special value (256, given the standard ASCII alphabet with decimal values 0-255). The encode program will write exactly one occurrence of this character at the end of the file. At some point your decoding method will come across this EOF character. At that point, it should stop decoding. It should not write this integer to the file because it isn't part of the original file.

If you fail to recognize the pseudo-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. If you see a value of -1 appearing, it's because you've read too far in the bit stream.

Be careful of using recursion in the `decode` method; Java has a limit on stack depth. For a long file, 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 thousands of levels deep. Use loops to make sure that you don't get a StackOverflowException, which could cause sadness or frustration or both.

You are given some data files for this assignment, including: "happy hip hop.txt" (a text file containing the example string from the how to), "short.txt", and "Hamlet.txt". The files "happy hip hop.txt" and "short.txt" are small files suitable for preliminary testing. The file "Hamlet.txt" contains the full text of Shakespeare's play *Hamlet* and should be used to test your compression when you believe your program is working (compression works better on larger data sets). Feel free to make more as you see fit.

The table below describes the naming conventions to use for the files involved in this assignment:

Extension	Example	Description
`.txt`	`Hamlet.txt`	Original text file
`.code`	`Hamlet.code`	List of character codes to use
`.short`	`Hamlet.short`	Compressed file (binary, not human readable) Should take up less disk space than the .txt file
`.new`	`Hamlet.new`	Decompressed file (should match the original)

It can be challenging to debug the decoding part of the assignment because the encoded files are not readable in a normal text editor and because the character boundaries are not obvious. If you absolutely MUST view your encoded files, you will have to use a hex editor. However, it is suggested that you log things to the console for debugging, rather than browsing the binaries of your encoded files.

Next, define a class **HuffmanCompressor.java** that will function as the client of your HuffmanTree class. Add the following static methods:

Method	Description
`void compress(String filename)`	This method should read `fileName` and generate a character frequency table (array), used to construct a Huffman tree. It should then write the encoding tree to file, and generate the compressed file, using the naming conventions seen previously.
`void expand(String codeFile, String fileName)`	This method should re-build the tree from the supplied code file. It should then write the decoded (expanded) output to the supplied file.

Remember that when generating the frequency table, you can assume that all characters in the file will be part of the standard 256-character ASCII alphabet. The `compress` method should use the BitOutputStream helper class to write individual bits to file; it has just one public bit-writing method, `void write(int b)`, which will write the integer provided as a bit to disk. If you want to see what is being output before you use BitOutputStream you could use a PrintWriter object and output to a temp file or just output to the console.

The `expand` method should use HuffmanTree's `decode` method to parse the stream of bits in the compressed file, and write the decoded text to the provided file. The expanded file should exactly match the original.

(Advanced) Java file compression

Idea curtesy Vivan Shah, Liberty student 2019

Write a program to recursively navigate your CS 3 project folders (e.g. on your H: drive or on your flash drive) and locate the largest .java file. Ensure that this file has a `main` method, such that you can see some simple output when it runs. Run your Huffman compression program on this file, then decompress it, then run the decompressed file to see if it compiles and produces the same output. The compiler won't cut you any slack on missing information or incorrect symbols!

java.io.File objects have a `listFiles` method that will return an array of all the files in its directory (if the file is a directory).

(Advanced) Burrows-Wheeler compression

Huffman coding (or variations of it) are used in conjunction with other techniques in many common compression techniques, for example PKZIP, JPEG, and MP3. The Burrows-Wheeler compression algorithm has even better performance than PKZIP or gzip, despite not being much harder to implement.

Write a program that implements Burrows-Wheel data compression (use your Huffman classes when applicable):

http://www.cs.princeton.edu/courses/archive/spring17/cos226/assignments/burrows.html