CSE 143: Computer Programming II — Winter 2021

Take-home Assessment 8: Huffman Coding — *due March 12, 2021 11:59pm*

This assignment will assess your mastery of the following objectives:

- Implement a well-designed Java class to meet a given specification.
- Implement, manipulate, and traverse a binary tree.
- Implement the `Comparable` interface
- Follow prescribed conventions for code quality, documentation, and readability.

Background: ASCII and Binary Representation

(You do not need to fully understand this section to complete the assessment.)

We discovered on an earlier assignment that every `char` has an equivalent `int` value between 0 and 255. (Recall, `(int)'a' = 97`, `(int)'z' = 122`, etc.) In this assignment, we go a step further: computers actually see integers as *binary* (1's and 0's). Binary is a lot like the normal numbers we use, except it's based around 2 instead of 10. For example, "1204" is written that way, because

$$1204 = \boxed{1} \cdot 10^3 + \boxed{2} \cdot 10^2 + \boxed{0} \cdot 10^1 + \boxed{4} \cdot 10^0$$

Similarly, because

$$97 = \boxed{0} \cdot 2^7 + \boxed{1} \cdot 2^6 + \boxed{1} \cdot 2^5 + \boxed{0} \cdot 2^4 + \boxed{0} \cdot 2^3 + \boxed{0} \cdot 2^2 + \boxed{0} \cdot 2^1 + \boxed{1} \cdot 2^0$$

97 is "`01100001`" in binary! In fact, all *char*'s can be written using exactly 8 binary digits (bits). This encoding, which is used pretty universally by computers, is called ASCII.

Background: Huffman Encoding

Huffman encoding is an algorithm devised by David A. Huffman of MIT in 1952 for *compressing* data to make a file occupy a smaller amount of space. Unbelievably, this algorithm is still used today in a variety of very important areas. For example, mp3s and jpgs both use Huffman Coding. The general idea behind Huffman Coding is the following:

What if we used fewer than 8 bits for characters that are more common?

Consider the following (simple) example. Imagine we have the following data:

```
bbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
cccaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ddaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

How many bits to store with ASCII Coding?

First, we find the ASCII code for each letter:

`a`	→	`01100001`,	`b`	→	`01100010`,
`c`	→	`01100011`,	`d`	→	`01100100`

Since each line has 80 letters, and each letter code is 8 bits, the number of bits required is:

$$\boxed{80 \cdot 8 \cdot 4 = 2560}$$

How many bits to store with Huffman Coding?

Since `a` is most frequent, we use a short code for it, then we use the next longest code for `b`, etc:

`a`	→	`1`,	`b`	→	`00`,
`c`	→	`011`,	`d`	→	`010`

This data has 229 `a`'s, 4 `b`'s, 3 `c`'s, and 2 `d`'s. Since we need one bit for `a`, two for `b`, and three for `c` and `d`, the total count of bits is:

$$\boxed{229 \cdot 1 + 4 \cdot 2 + 3 \cdot 3 + 2 \cdot 3 = 255}$$

Woah! By changing the coding, we compressed the data by a factor of 10!!

In this assignment, you will create classes HuffmanCode and HuffmanNode to be used in compression of data You are provided with a client HuffmanCompressor.java that handles user interaction and calls your huffman code methods to compress and decompress a given file.

Program Behavior: Building a Huffman Code

In the previous example, we magically arrived at the special binary sequences to be used as codes. In this section, we explain the algorithm to create these special binary sequences.

Throughout this section, we will use the following example:

simple-spec-example.txt

```
aba ab cabbb
```

Step 1: Count the Frequencies of the Characters

The file simple-spec-example.txt has the characters 'a', 'b', 'c', and ' '. Counting up the frequencies, we get the following:

Character	Count
' '	2
'a'	4
'b'	5
'c'	1

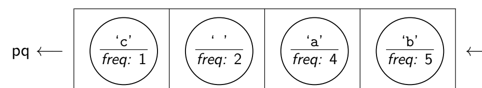
Step 2: Create a Priority Queue Ordered By Frequency

Since our ultimate goal is to create a code based on frequencies, we need to use a data structure that helps us keep track of the order of the letters based on frequencies. We will use a *priority queue* for this. A *priority queue* is a queue that is ordered by *priorities* (in this case frequencies) instead of FIFO order. In other words, we can ask the priority queue to insert a new element (add(element)) and we can ask it to remove the highest priority element (remove()). We will use the PriorityQueue<E> class in Java for the implementation of priority queues, but we will still use the Queue<E> interface for variables.

In Java, if you print out a priority queue, the elements *will not* appear in priority order. This is EXPECTED behavior, and can't be easily changed.

For our huffman code, we want to remove the node with *lowest* frequency first. (Intuitively, the reason is that the things we remove first will end up having the longest codes).

We begin by creating a node for each letter in our text:



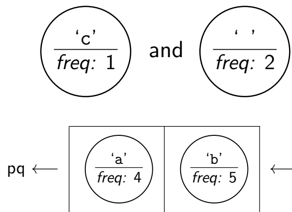
Continued on next page...

Step 3: Combine the Nodes Until Only One is Left

Now that we have a priority queue of the nodes, we want to put them together into a tree. To do this, we repeatedly remove the smallest two (the ones at the front of the priority queue) and create a new node with them as children. (The new node does not have a character associated with it.) Note that the priority queue is *arbitrary* in how it breaks ties; you should just take nodes in the order the priority queue provides them.

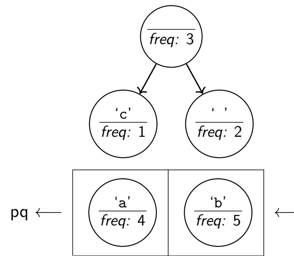
(3a) Remove Two Smallest

First, remove the smallest two nodes from the priority queue:



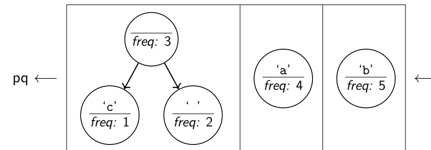
(3b) Combine Them Together

Then, create a new node. The frequency is the sum of two nodes:



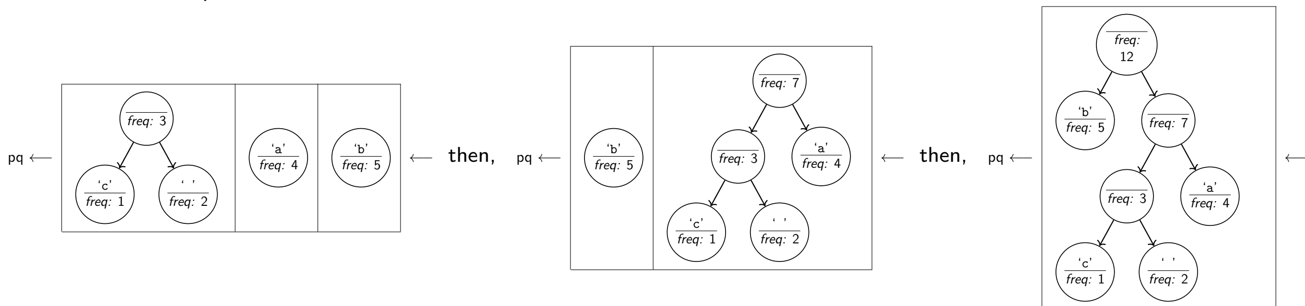
(3c) Add Back To Priority Queue

Now that we have a “new node”, add it back to the priority queue:



We repeat this process until there is only one node left in the priority queue.

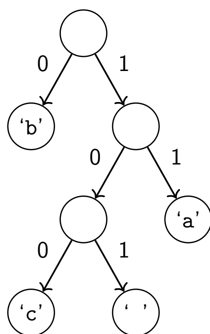
Here are the remaining steps (each time, we remove the two smallest frequencies, combine them, and put the result back):



Now that we only have one node left, we can use the tree we constructed to create the huffman codes!

Step 4: Read Off The Huffman Codes

At this point, the frequencies of the letters have already been taken into account; so, we no longer even look at them. Thus, the tree we've constructed looks like the following:



simple-spec-example.code

```
98
0
99
100
32
101
97
11
```

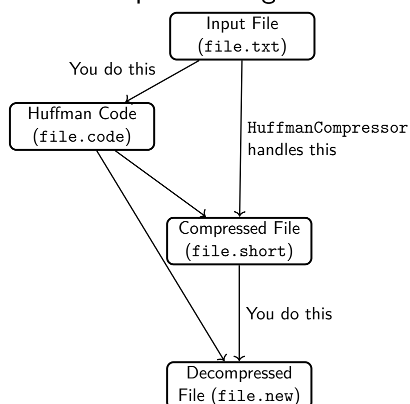
To figure out the huffman code for a letter, we traverse the tree from the root to the node with the letter in it. When we go left, we print a 0 and if we go right, we print a 1. For example, ‘c’ would be 100, because we go right, then left, then left to reach it.

Just like in 20 questions, we will output the tree in “standard format”. Notice that the only actual

information is in the leaves of the tree. So, the code file, which you can get by asking the main to "make a code", will consist of line pairs: the first line will be the ASCII value of the character in the leaf and the second line will be the huffman code for that character. For example, the output of the tree we just constructed would look like the above. The leaves should appear in the order of a **pre-order** traversal.

Program Behavior: Huffman Compression and Decompression

Now that we know how to construct a huffman code, we are ready to understand the huffman compression and decompression algorithms. Here is an overview of how they work:



Compression Algorithm

(1) `HuffmanCompressor` generates the frequencies in a given file.

(2) `HuffmanCode` creates the huffman code from the frequencies.

(3) `HuffmanCode` writes out the huffman codes to a `.code` file.

(4) `HuffmanCompressor` writes a `.short` file.

Decompression Algorithm

(1) `HuffmanCode` reads in a `.code` file in standard format.

(2) `HuffmanCode` writes out a `.new` file.

`HuffmanCompressor` is a client program that we will provide to you. In addition to handling user interactions, it implements some of the steps of the compression and decompression algorithms that we are not asking you to deal with. Specifically, `HuffmanCompressor` handles:

- computing character frequencies for a given input file (these frequencies are passed to the first `HuffmanCode` constructor below)
- compressing a text file using a given Huffman tree
- producing a `BitInputStream` from a given compressed input file (this stream is passed to the `translate` method)

You do not need to implement any of the above behavior. You **only** need to implement the behavior and methods described below.

`HuffmanNode`

The contents of the `HuffmanNode` class are up to you. Though we have studied trees of `int`s, you *should* create nodes specific to solving this problem. Your `HuffmanNode` should must be a separate public class and should have at least one constructor used by your tree. The fields in your `HuffmanNode` *must* be public. `HuffmanNode` should not contain *any actual huffman coding logic*. It should only represent a single node of the tree.

Continued on next page...

`HuffmanCode`

This class represents a huffman code for a particular message. It keeps track of a binary tree constructed using the huffman algorithm.

Your `HuffmanCode` class should have the following constructors:

`public` **`HuffmanCode`**`(int[] frequencies)`
This constructor should initialize a new `HuffmanCode` object using the algorithm described for making a code from an array of frequencies. `frequencies` is an array of frequencies where `frequences[i]` is the count of the character with ASCII value `i`. Make sure to use a `PriorityQueue` to build the huffman code.

`public` **`HuffmanCode`**`(Scanner input)`
This constructor should initialize a new `HuffmanCode` object by reading in a previously constructed code from a `.code` file. You may assume the `Scanner` is not `null` and is always contains data encoded in legal, valid standard format.

Your `HuffmanCode` class should also implement the following public methods:

`public void` **`save`**`(PrintStream output)`
This method should store the current huffman codes to the given output stream in the standard format described above.

`public void` **`translate`**`(BitInputStream input, PrintStream output)`
This method should read individual bits from the input stream and write the corresponding characters to the output. It should stop reading when the `BitInputStream` is empty. You may assume that the input stream contains a legal encoding of characters for this tree's huffman code. **See below for the methods that a `BitInputStream` has.**

`BitInputStream`

The provided `BitInputStream` class reads data bit by bit. This will be useful for the `translate` method in `HuffmanCode`. `BitInputStream` has the following methods:

`public int` **`nextBit`**`()`
This method returns the next bit in the input stream. If there is no such bit, then it throws a `NoSuchElementException`.

`public boolean` **`hasNextBit`**`()`
This method returns true if the input stream has at least one more bit and false otherwise.

The interface for `BitInputStream` looks very much like a `Scanner`, and it should be used similiarly.

Implementation Guidelines

Your program should exactly reproduce the format and general behavior demonstrated in the Ed tests. Note that this assignment has two mostly separate parts: creating a huffman code and decompressing a message using your huffman code. Of the four methods to implement, two are relevant to each part.

Creating A Huffman Code

You will write methods to (1) create a huffman code from an array of frequencies and (2) write out the code you've created in standard format.

Frequency Array Constructor

You should use the algorithm described in the "Making a Huffman Code" section to implement this constructor. You will need to use `PriorityQueue<E>` in the `java.util` package.

The only difference between a priority queue and a standard queue is that it uses the natural ordering of the objects to decide which object to dequeue first, with objects considered "less" returned first. You will be putting subtrees into your priority queue, which means you'll be adding values of type `HuffmanNode`.

This means that your `HuffmanNode` class will have to implement the `Comparable<E>` interface. It should use the frequency of the subtree to determine its ordering relative to other subtrees, with lower frequencies considered "less" than higher frequencies. If two frequencies are equal, the nodes are too.

Remember that, in order to make our code more flexible we should be declaring variables with their interface types when possible. This means you should define your `PriorityQueue` variables with the `Queue` interface.

The Huffman solution is not unique. You can obtain any one of several different equivalent trees depending upon how certain decisions are made. If you implement it as we have specified, then you should get exactly the same tree for any particular implementation of `PriorityQueue`. Make sure that you use the built-in `PriorityQueue` class and that when you are combining pairs of values taken from the priority queue, you make the first value removed from the queue the left subtree and you make the second value removed the right subtree.

Decompressing A Message

You will write methods to (1) read in a `.code` file created with `save()` and (2) translate a compressed file back into a decompressed file.

`Scanner` Constructor

This constructor will be given a `Scanner` that contains data produced by `save()`. In other words, the input for this constructor is the output you produced into a `.code` file. The goal of this constructor is to re-create the huffman tree from your output. Note that the frequencies are irrelevant for this part, because the tree has already been constructed; so, you should set all the frequencies to some standard value (such as 0 or -1) when creating `HuffmanNode`s in this constructor.

See lecture slides for example of this algorithm

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

`translate`

This method takes in a `BitInputStream` representing a previously compressed message and outputs the original decompressed message. `BitInputStream` can be used in a very similar way to a `Scanner`; see the description of its methods on page 4.

This method reads sequences of bits that represent encoded characters to figure out what the original characters must have been. The algorithm works as follows:

- Begin at the top of the tree
- For each bit read in from the `BitInputStream`, if it's a 0, go left, and if it's a 1, go right.
- Eventually, we will hit a leaf node. Once we do, write out the integer code for that character to the output using the following `PrintStream` method:

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

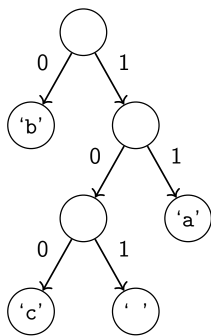
- Then, go back to the top of the tree, and do the process over again.

Be sure to use `write` instead of `print`!

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 large message, there are hundreds of thousands of characters to decode. 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.

Translate Example

Suppose we have the following `.code` and `.short` files:



simple-spec-example.short
```
1101110111010110011000
```

Read 1, go right. Read 1, go right. 'a' is a leaf. Output 'a'. (Input: 110101110101100)
Read 0, go left. 'b' is a leaf. Output 'b'. (Input: 110101110101100)
Read 1, go right. Read 0, go left. Read 1, go right. ' ' is a leaf. Output ' '. (Input: 110101110101100)
Read 1, go right. Read 1, go right. 'a' is a leaf. Output 'a'. (Input: 110101110101100)
Read 0, go left. 'b' is a leaf. Output 'b'. (Input: 110101110101100)
Read 1, go right. Read 0, go left. Read 1, go right. ' ' is a leaf. Output ' '. (Input: 110101110101100)
Read 1, go right. Read 0, go left. Read 0, go right. 'c' is a leaf. Output 'c'. (Input: 110101110101100)

So, the decompressed text is "`ab ab c`".

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.

You will need to *upload* your `secretmessage.short` and `secretmessage.code` files to Ed by either dragging and dropping them into the window or using the "Upload" button. Copying/pasting the file into the Ed editor will **NOT** correctly copy the compressed text.

Code Quality Guidelines

In addition to producing the behavior described above, your code should be well-written and meet all expectations described in the grading guidelines, Code Quality Guide, and Commenting Guide. For this assessment, pay particular attention to the following elements:

`x = change(x)`

An important concept introduced in lecture was called `x = change(x)`. This idea is related to proper design of recursive methods that manipulate the structure of a binary tree. You should follow this pattern where necessary when modifying your trees.

Avoid Redundancy

Create "helper" method(s) to capture repeated code. As long as all extra methods you create are `private` (so outside code cannot call them), you can have additional methods in your class beyond those specified here. If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code.

Generic Structures

You should always use generic structures. If you make a mistake in specifying type parameters, the Java compiler may warn you that you have "unchecked or unsafe operations" in your program. If you use jGRASP, you may want to change your settings to see which line the warning refers to. Go to `Settings/Compiler Settings/Workspace/Flags/Args` and then uncheck the box next to "`Compile`" and type in: `-Xlint:unchecked`

Data Fields

Properly encapsulate your objects by making data fields in your `HuffmanCode` class `private`. (Fields in your `HuffmanNode` class should be `public` following the pattern from class.) Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place. Fields should always be initialized inside a constructor or method, never at declaration.

Commenting

Each method should have a header comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implemenation details.

Running and Submitting

If you believe your behavior is correct, you can submit your work by clicking the "Mark" button in the Ed assessment. You will see the results of some automated tests along with tentative grades. **These grades are not final until you have received feedback from your TA.**

You may submit your work as often as you like until the deadline; we will always grade your most recent submission. Note the due date and time carefully—**work submitted after the due time will not be accepted**.

Getting Help

If you find you are struggling with this assessment, make use of all the course resources that are available to you, such as:

- Reviewing relevant examples from class
- Reading the textbook
- Visiting office hours
- Posting a question on the message board

Collaboration Policy

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own**. In particular, you should **NEVER** look at a solution to this assessment from another source (a classmate, a former student, an online repository, etc.). Please review the full policy in the syllabus for more details and ask the course staff if you are unclear on whether or not a resource is OK to use.

Reflection

In addition to your code, you must submit answers to short reflection questions. These questions will help you think about what you learned, what you struggled with, and how you can improve next time. The questions are given in the file `HuffmanReflection.txt` in the Ed assessment; type your responses directly into that file.