Homework 5: Priority Queue and Huffman

Due by 11:59PM EDT on Sunday, October 13, 2019

2 Required Problems: Priority Queue and Huffman (95 points total), and Style and Tests (5 points)

Setup and Logistics

We have put together several stub files — **click here to download them (/~cis121/current/hw/5-huffman/hw5-stub-files.zip)**.

4 files to submit: `Huffman.java` , `HuffmanTest.java` , `BinaryMinHeapImpl.java` , `BinaryMinHeapImplTest.java`
Please do not submit any additional files.

Motivation – Compression Algorithms

The goal of a compression algorithm is to take a sequence of bytes and transform it into a different sequence of fewer bytes, such that the original can be recovered. Because compression algorithms reduce the size of a file, they allow quicker transmission of files over the network, benefiting everyone on that link. Regarding bandwidth, please see this xkcd link (https://what-if.xkcd.com/31/).

Compression algorithms can be lossless (like ZIP) or lossy (like JPEG). Lossy compression is useful for images, music, and video because multimedia is an emergent property of its file formats; in other words, we don't care if there are slight imperfections in a JPEG image. On the other hand, text-based files must be compressed without any data loss; what good is a source code file if it cannot be compiled because a few of its bits switched from 1 to 0?

In this assignment, you will implement a *lossless* compression algorithm, namely Huffman Coding, that is used as part of larger compression schemes, such as ZIP. In order to implement it, you'll also be implementing a binary min heap which will be used as a priority queue.

NOTE: The `BinaryMinHeap` interface cannot be modified in any way, nor can the method headers in `BinaryMinHeapImpl` or `Huffman.java` . You also cannot modify the provided `MinPQ.java` . Failure to abide by this will prevent your code from compiling and will lose you credit on the assignment.

Part 1: Binary Min Heap

Files to submit: `BinaryMinHeapImpl.java` , `BinaryMinHeapImplTest.java`

In order to implement Huffman's algorithm, you'll need to implement a min-heap. We have provided an interface that you must implement, and we have also provided you with the corresponding class stub file. Please make sure you do not modify the `BinaryMinHeap.java` interface at all! Also, do not modify any of the method headers in the `BinaryMinHeapImpl.java` file. As always, you may add package private helper methods in the Impl file. Note, you do not need to worry about your `BinaryMinHeapImpl.java` working with other implementations.

Please make sure you read the interface thoroughly, as it has details about the required runtimes for all the methods. Also, you do NOT have to implement the `decreaseKey` method–that will be used for a later homework assignment.

Some notes regarding implementation:

- We strongly recommend using ArrayLists rather than arrays to avoid resizing the array and dealing with casting issues.
- TreeSets do not support null values and requires entries to extend Comparable, for that reason, use a HashSet instead.
- You should implement min-heapify, but do not need to support an $O(n)$ build-heap operation.
- You should not implement your heap as a sorted array or linked list.

Regarding `MinPQ`

The `MinPQ` class is useful when you are trying to store objects that both 1) contain the information you want to keep track of and 2) are also the thing you want to compare on. For this assignment, the `MinPQ` should be a priority queue of nodes (your inner class nodes), where the node contains both information about the key (i.e. how the nodes should be ordered) and any other values/additional fields that you may want to add.

- Don't let the type `<E, E>` within `MinPQ` confuse you - when looking into the actual implementation of `MinPQ` it can get confusing, but just think of `MinPQ` as being some queue which contains these above nodes - this is why the `MinPQ` class is just of one "type" `E`
- This class is just emulating the behavior of Java's PriorityQueue - which you can read up on here (https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html).

Note: some runtimes will not match what you learned in lecture. We'll improve on some of these runtimes later on in the course when you revisit your implementation in the graphs programming assignment.

Part 2: Huffman coding

Files to submit: `Huffman.java`, `HuffmanTest.java`

The compression algorithm you'll implement is known as Huffman coding. The idea behind Huffman coding, and an idea common to a lot of concepts and techniques in computer science, is that common activities should have a lower cost than rarer activities. For example, we encode ASCII characters with eight bits each, but we see the character `e` much more often than the character `x`, which we see much more often than the `BEL` character (ASCII code 7 (http://www.asciitable.com)). Therefore, why not represent those more frequent characters with fewer bits and those rarer characters with more bits?

First, assume the existence of an alphabet, whose composing characters abide by some positive-valued probability mass function. In other words, each character in the alphabet has a probability of showing up, these probabilities are each greater than zero, and these probabilities sum to one. Your Huffman implementation will permit alphabet specification by either passing in a map from `char`s to `int`s or by providing a `String` from which the alphabet and probability mass function will be determined. Here, each `int` represents its corresponding `char`'s count, not its probability.

Why is this the case? We use `int`s instead of `double`s or `float`s because of the inherent imprecision of the floating-point standard; it is impossible to accurately represent a number as simple as $\frac{1}{3}$. Instead, we determine the alphabet's probability mass function by taking each `Character`'s corresponding count and dividing it by the sum of all `Character`s' counts.

Here is a (simple) example alphabet, according to *our* specification:

a	b	c
1	2	2

Here, the character `a` shows up $\frac{1}{5}$ of the time, and the characters `b` and `c` each show up $\frac{2}{5}$ of the time.

Huffman coding uses a binary tree to encode character information, where every leaf represents a character in the alphabet and where each edge represents the state of a bit in the compression. Starting from the tree's root, traveling to a node's left child indicates appending a `0`-valued bit to the compressed representation of a character, and traveling to a node's right child indicates appending a `1`-valued bit to the compressed representation of a character. Note that because of this, Huffman coding requires the alphabet to have no fewer than two characters.

The generation of the Huffman tree is the crux of the algorithm. It uses a priority queue, which is an abstract data type that keeps track of the smallest (i.e. highest-priority, hence the name) element in a collection of elements. Naturally, a priority queue can be implemented by a binary min-heap, and so we have provided a `MinPQ.java` class which does exactly that: it wraps around your implementation of `BinaryMinHeap.java` and provides functions similar to those provided by `java.util.PriorityQueue`.

Using the priority queue, one can implement Huffman's algorithm as follows:

Create a lone leaf node for each character in the alphabet (i.e. you should have n discrete leaf nodes). Add each node into a priority queue, and while the priority queue has more than one node in it, follow the following process:

1. Remove the two nodes of lowest frequency from the priority queue.
2. Create a new node. Assign its left child to be the first removed node (the one with lower frequency) and its right child to be the second removed node (the one with relative higher frequency). **Note**: While this directional invariant is arbitrary, failure to abide by this standard will result in lost points, as our unit tests will expect this invariant.
3. Assign the new node's frequency to be the sum of its left child's frequency and its right child's frequency. This new node is an internal node and no longer corresponds to a character in the alphabet.
4. Add the new node to the priority queue.

When there is only one node left in the priority queue, it is our final Huffman tree.

> **Note**: You may NOT use `java.util.PriorityQueue` as the priority queue for this part of the assignment. Instead, you should either use the provided `MinPQ.java` or your implementation of `BinaryMinHeap.java`

We have provided five method and constructor stubs for you. **Do not** modify the headers of these methods and constructors, only the bodies. Failure to abide by this will prevent your code from compiling, and you will lose credit on the assignment.

Of note regarding the implementation:

- Note that in the lecture notes, the tree generation algorithm was $O(n \log n)$ when using a binary min heap. In this assignment, because the `add` method in your `BinaryMinHeap.java` implementation is $O(n)$, you won't be required to meet this runtime.
- Decompression is straightforward; given a sequence of `0`s and `1`s, start from the root and go left or right as determined. When you get to a leaf, you've translated a character; record that, and go back to the root.
- Compressing is less straightforward because you cannot use the `0`s and `1`s to find a character; you want to use the character to find the `0`s and `1`s. Any characters not in your alphabet are not compressible.

- Both `compress` and `decompress` methods should return the empty string `""` on an empty string input.
- Your `compress` and `decompress` methods should be as asymptotically efficient as possible. Part of this includes using the `StringBuilder` class to create your compressed and decompressed output instead of using `String` concatenation; the former can concatenate two `String` s in $\Theta(1)$, whereas the latter can only concatenate two `String` s in $\Theta(n)$. We will stress test your code against massive inputs to make sure you used `StringBuilder` .
- The `expectedEncodingLength` method is nothing more than the expectation of a discrete random variable. We are using this method not to test your knowledge of basic probability, but instead to test the optimality of the encoding that your algorithm generates. You will need to compute the weighted sum of the lengths of the encodings of each character, where the weights are the probabilities of the characters.
- You will need to use an inner class for the Huffman tree. You can make it package private so that you can test it. This class will need to implement the `Comparable<T>` interface so you can use the `MinPQ` (or `BinaryMinHeap` should you choose not to use the provided `MinPQ` class) to store the incomplete Huffman trees. You are strongly encouraged to override the `toString` method to print a tree representation for ease of debugging.
- Because we will be unit testing your output, you must implement your `compareTo` function of the Huffman tree nodes with the following procedure for breaking frequency ties: if two leaf nodes have the same frequency, the node with the “smaller” character (use ASCII value to represent “smaller”) should be the “smaller” node. If the two nodes being compared aren’t both leaf nodes, the node that was created *first* should be the “smaller” node. (How might you go about implementing this?)
- Our implementation uses `String` s of `0` s and `1` s to represent a stream of bits. This is convenient for a program written in Java, but it is not what would happen in the real world. If we were to implement this algorithm in production software, we would post-process each compressed answer by converting each character into its corresponding bit.
- We will compute a compression ratio that quantifies how well our compression performs. It should return the average compression ratio for all strings you have compressed over the lifetime of the object instance. We define compression ratio as the length of all outputs from compression over the length of all inputs to the compression. This length should be calculated in terms of “bits”. We recognize that our output, although a binary string, isn’t actually a sequence of bits. However, you can pretend that it is just some binary representation of the original input. You do not have to explicitly turn the input into binary, but treat the “length” as the number of bits. You can treat the input length as the number of characters times the size of a Java `char` , which is 16 bits. The ratio is maintained and modified as you continue compressing strings using the same Huffman instance, since it is an aggregate ratio.

Visualizer

For this homework assignment, we are providing a visualizer to help you visualize Huffman trees and test the correctness of your Huffman implementation.

First, you need to add the `GraphJar.jar` file to your build path. To do so, right click on your project in Eclipse → Build Path → Configure Build Path… → Click on `Add External JARs...` on the right pane → select `GraphJar.jar → Open → Apply and Close` .

You can then run the Huffman visualizer by right clicking on `HuffmanVisualizer.java` → Run as → Java Application

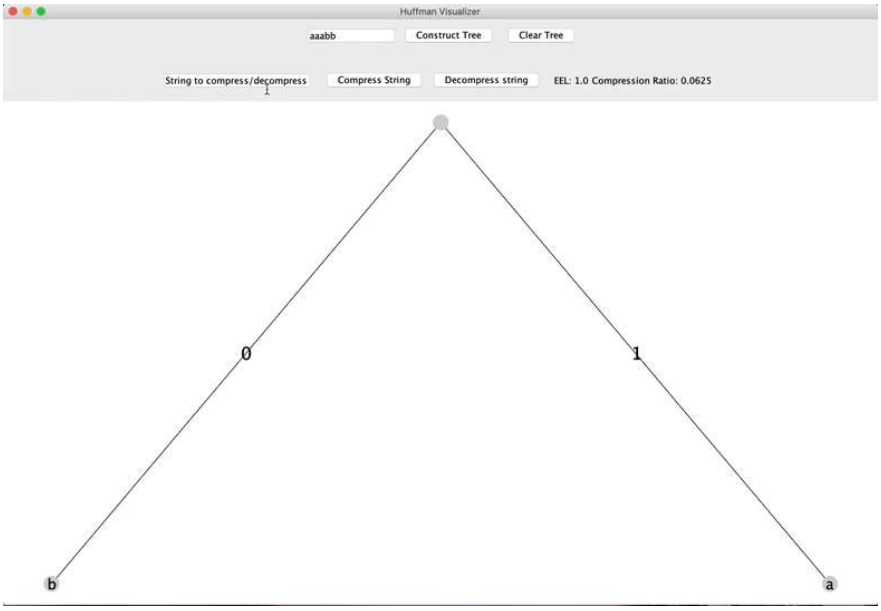
In the topmost field, enter the alphabet seed, or the same string that the Huffman constructor takes in. You can then press `Construct Tree` to build the Huffman tree. Note: this operation uses the implementation of your Huffman constructor. You then should see your Huffman tree:

Note: For long strings, the resulting tree might have overlapping nodes. You can click and drag the nodes to move them around.

You can clear a tree and all of the variables by clicking the `Clear Tree` button.

In addition, the visualizer will display the Expected Encoding Length in the top right corner.

The visualizer supports compressing and decompressing strings. The visualizer may not act as intended with numerical characters, so please stick to non-numerical characters. You can do so by entering a string to either compress or decompress in the second row and clicking the compress or decompress buttons. The output will be displayed. In addition, the compression ratio will be displayed in the top right corner.



Style & Tests (5 points)

The above parts together are worth a total of 95 points. The remaining 5 points are awarded for code coverage. Style is graded on a subtractive basis with deductions at a maximum of 5 points, and you will be graded according to the CIS 121 style guide (/~cis121/current/java_style_guide.html). Gradescope will give you your grade on this section immediately upon submission.

> **IMPORTANT**: Please **DO NOT** use any external files (.txt, etc.) to write your JUnit tests for this assignment. Our code coverage tool will fail, and you will not be able to submit your code due to failing test cases.

You will need to write comprehensive unit test cases for each of the classes, inner classes, and methods you implement (including helper methods!) in order to ensure correctness. Make sure you consider edge cases and exceptional cases in addition to typical use cases. Do not throw exceptions not specified in the documentation! Use multiple methods instead of cramming a bunch of asserts into a single test method. Your test cases will be auto-graded for code coverage. You do not need to test any files we give you! Be sure to read the testing guide (/~cis121/current/testing_guide.html) for some pointers on what level of testing we expect from you. Finally, due to the Code Coverage tools we use, **please be sure to use JUnit 4** as it is the only version supported.

Note: You will not be able to write JUnit test cases for any private methods. Instead, make them package-private by leaving off any privacy modifier (i.e., do not write `public` or `private`).

This assignment was developed by the CIS 121 Staff.
Last updated on Thu, Oct 3 at 05:30 PM.