# Huffman Encoding

CS 223

## 1 Introduction

Huffman encoding is a form of data compression which is very useful for text. Normally, (English) text is encoded so that each letter occupies the same amount of space – typically a byte. Huffman encoding works by observing that some letters occur more frequently than others (e.g. E is very common, Z is very uncommon.) Using this, it constructs variable-length encodings so that frequently occurring letters occupy less space, individually, than their infrequent counterparts.

For this project, you'll be implementing a priority queue based on a min heap, and using that as the basis of a Huffman encoder. Your encoder will work on strings, and will return strings of both the encoded Huffman tree and the text itself, along with bitstrings for each character. I am providing you with skeleton code and a test suite. More on those in Section 2. Details of how your priority queue should work are covered in Section 3, with the Huffman encoder itself covered in Section 4. Section 5 covers the use of the provided test suite, and Section 6 covers how to turn in your project.

## 2 Provided Code

This project is accompanied by a file named `huffman-handout.zip`. Inside, you will find five Java source files that will help you with your project. (You'll also find five other files, which can be safely ignored.) The Java source files are:

**HuffmanEncoder.java** – Skeleton you'll fill in and then submit

**PriorityQueue.java** – A Java Interface for a Priority Queue ADT

**HuffmanDecoder.java** – A bare-bones Huffman decoder, used by the Huffman tests

**HuffmanEncoderTests.java** – JUnit tests for your Huffman encoder class

**MinHeapPriorityQueueTests.java** – JUnit tests for your Priority Queue class

To get started, I recommend adding all five Java files to IntelliJ (or your development environment of choice). IntelliJ will report errors with the test files as it does not include JUnit in the set of available libraries by default. To fix this, expand the imports at the top of the file, hover over the import for `org.junit.Test`, and select "Add JUnit4 to classpath". You'll be prompted to install the library. Click OK and give it a moment to install.

You should only make changes you want graded in `HuffmanEncoder.java`. (You are welcome to add tests, but we won't see them.) I have provided a pair of simple `main` methods which may help for testing. The JUnit tests are more extensive, and more indicative of what we'll run for grading.

# 3 Priority Queue Implementation

There is a skeleton Priority Queue implementation at the bottom of `HuffmanEncoder.java`. I have provided you with some helper definitions and methods which you may use. Your primary responsibility for this part of the project is to implement the `insert` and `delNext` methods so as to produce a min-heap-based priority queue. (That is, a PQ in which the item with the highest priority is the one with the lowest value.)

## 3.1 Implementing the Data Structure

Your implementation must work for all Comparable types and implement the PriorityQueue interface I provided. (The existing class definition takes care of this.) Further, you must maintain a proper, array-based heap as described in class. Implementations that store values in arbitrary order and then search for the smallest at runtime will be considered wrong. You may implement separate sift/swim methods, or leave the functionality embedded in the insert and delete methods.

## 3.2 Arrays of Generics

Java does not like to have arrays of generic types. To get around this, I have provided you with a properly declared array (named `heap`), and set up the constructor to initialize it to hold 1 element. I have also provided a `resize` method which will double the size of the array for you. Your `insert` method will need to check whether the array must be resized, and, if so, invoke `resize` as needed. The array should otherwise behave exactly like a normal Java array.

## 3.3 The `toString` Method

I've included an implementation of the `toString` method in the Priority Queue class. This will return a string version of the contents of the array, which is useful for debugging and is used in one of the tests I provided for you. If you use my array you will not need to modify this method.

# 4 Huffman Encoding

There is a skeleton Huffman Encoder at the top of `HuffmanEncoder.java`. I included a `Node` implementation which is Comparable and provides a useful toString method, for use with your Priority Queue.

A `HuffmanEncoder` object will receive the string it is to encode via its `encode` method. Once a string is encoded, the Encoder provides the encoded Huffman tree, the bitstrings for each character, and the encoded text via method calls. Comments in the skeleton provide information about what each of the required functions should do.

You are welcome to add additional methods to your `HuffmanEncoder`, and arrange its internal data storage however makes sense to you. Careful use of a `StringBuilder` and `String`s should make it straightforward to generate the tree encoding and the bitstrings. Please note that your implementation might produce a different tree, and therefore encoding, than appears in the slides. This is to be expected, and is fine so long as a) it decodes properly and b) you end up with the right string length for the encoded text.

## 5 Testing

I have provided you with JUnit test suites for both your Priority Queue and Huffman Encoder. IntelliJ understands how tests work (once you add JUnit, anyway), and can manage running the tests for you. You can run all of the tests in a given file at once, or run them individually. There are 13 tests in total. Your submission must pass all 13. Autolab is set up with an addition four tests which you must also pass. It will provides hints as to what went wrong if there are problems.

Because there are potentially multiple correct Huffman encodings of a given string, the tests for your Huffman Encoder rely on a Decoder I've provided you. That way, I can test whether your encoded tree makes sense, provides the bitstrings you got, successfully decodes the encoded text, and gives an encoding of the correct length. You're free to ignore the details of the HuffmanDecoder, but perusing it might give you some ideas for implementing pieces of your Encoder.

## 6 Submitting your solution

You will be submitted exactly one file for this project: `HuffmanEncoder.java`. Once you have implemented this project to your satisfaction, submit the file, uncompressed, to Autolab. Your submission will be passed through automatic testing, so you should get feedback on functionality quickly.