Compression

For this homework assignment you will implement and experiment with methods for data compression, using two lossless data compression algorithms: Huffman coding and LZW compression. The primary goal of the assignment is to implement the backend for a program called `tez` (two-eleven zip) to compress and decompress data files using a variety of compression algorithms.

Goals

- Apply data structures and algorithms from 15-211 -- priority queues and Huffman trees -- to a real-world problem.
- Gain an understanding of (basic versions of) major data compression algorithms, including Huffman coding and LZW.
- Gain experience working with more realistic problems that are larger and open-ended, with black-box specifications and requirements.

Deliverables

PriorityQueue (20 points)

Implement the java.util.PriorityQueue API, except for a few methods and features like fail-fast iterators.

Huffman (20 points)

Generate an object-oriented abstraction of the concept of Huffman codes. We have created a simple Huffman compressor that uses this.

LZW (20 points)

Implement the LZW algorithm taught in lecture. This is similar to one of the components in DEFLATE compression, used by gzip and zip, two extremely popular compression methods. It is also used in GIF image compression. Details can be found in the lecture notes.

Theory Questions (30 points)

Turn in a printed, typeset copy of your theory questions in recitation on Wednesday, 18 October.

Testing and Style (30 points)

We understand that a lot of your testing will be to compress and decompress files on your hard-drive. Frontdesk does not allow filesystem access, and so you will be unable to run those tests on Frontdesk. Your non-internal unit tests should focus on small easy-to-debug tests cases.

Testing Code

Writing tests is good!

By now you should realize that testing is highly beneficial. Please use the .tests package for your JUnit tests, as in previous assignments. Be sure to test major compress/decompress functionality, as well as unit tests for individual parts like the PQ and building the HuffmanTree.

Important: Again, you will *not* be able to access the file system on FrontDesk. FrontDesk will throw an exception any time you access the file system. You will need to generate `byte[]` arrays for your non-internal tests, instead of using the file system. Two useful methods for getting arrays are `String.getBytes("ASCII")` and `Random.nextBytes`.

Sample files

To help you ensure that your file format is correct, we have created a repository of example files and their compressed versions.

You should be able to uncompress all of the files that we compressed. For Huffman-based formats, the bytes that constitute your files can be different from supplied files (because of how one creates Huffman trees). However, the file length should be the exactly same (try `wc -b` or `ls -la`).

This is a good time to point out that good code coverage does not neccesarily equal good grades. The few hand cases, if written correctly can achieve full code coverage of the compression stage. They are, however, insufficient to thoroughly test your code. You should perform local tests with the provided files. You should put try-catch blocks around all tests that access files to handle the IOExceptions that will occur on frontdesk.

Priority Queue (20 points)

Your first task in this assignment is going to be to implement a priority queue. You will be using the Queue/AbstractQueue in Java 5, much like the API from the Snake assignment. There isn't much to say about this API, other than to read the javadocs and PLEASE TEST.

As far as implementation goes, you are free to implement a priority queue in any (efficient) way you want. This means that offer and poll must be O(log(n)) time. We recommend using a binary heap.

You *may not* use the Java implementation of priority queue, nor may you hack a priority queue using some sort method (not log n time).

Note that for the iterator, you do not have to return the elements in any specific order. This would make your life painful, and we have no desire to do that. Similar to Java's own implementation of `PriorityQueue`'s Iterator, just return all elements in the queue in any order.

Bits and Compressors

I'm More Than Just a Number!

Up until now you have been dealing with integers which represent the natural numbers. However, there is another, (slightly more painful) world where we use integers as messengers of a sequence of bits. Compression is often done in terms of a stream of bits.

We define a `BitReader` as an interface which allows reading bits from some stream. It has methods for reading one bit, reading a few bits, reading a byte, reading a few bytes, and reading an integer. It also supports getting the length of the stream and resetting to the beginning. `InputStreamBitReader` is a concrete implementation of `BitReader` that supports reading from a file or a few other types of streams (including a stream backed by a byte array).

A `BitWriter` which reads files in the same format as `BitReader`, is defined with a concrete implementation in `OutputStreamBitWriter`.

In general, we will tell you the exact format in which to write the files. Your files must be *bit for bit* compatible with our files. You should be able to read our files, and we yours. Huffman-compressed files may not necessarily be exactly the same since there are multiple, valid Huffman trees, but they should be compatible. LZW-compressed files, on the other hand, should be identical.

WARNING: do *not* try to convert bytes into strings. Ever. It might work when you are working with ASCII. But, due to the beast known as encoding, it will not work on binary files. In this assignment, you are dealing with *bytes* not *strings*

The byte primitive type in Java is signed (sadly). This may be slightly confusing if you are using the debugger and see the value -32 or something. Do not be alarmed. The byte encoding is still the exact same, so you needn't worry about this. If you want to get a readable output from Java, say `Integer.toString (byteValue & 0xff)`. In general you code should work even if you never knew this.

When testing code, you may have to view binary files. The easiest platform to do this on is Linux, where there are lots of quality hex editors. In emacs, use the Hexl mode (`M-x hexl-mode`). In vi, execute `:%xxd` to change the file to hex and `:%xxd -r` to change it back. There is also the `hexdump` utility (try with the `-C` flag). GUIs also exist such at `ghex2` for GNOME. For Windows and Mac, there are quite a few shareware/freeware hex editors, use Google.

Compressor

We also define an interface called `Compressor`, which represents the abstract concept of

compression. In mathematical terms, a compressor is a function which takes a bit stream *s*

and maps it to another stream in an injective manner. The function has the property

that when *s* has redundant data, then

Please note that your compressors *must* be stateless. This means that we first should be able to construct a new instance, c, of your compressor. We then should be able to call c.compress(x) and c.decompress(y) multiple times on multiple bit streams, without having to reconstruct another compressor object.

Huffman Code (20 points)

`HuffmanCode.java` implements a generic Huffman backend. The details of this API are described in the javadocs; this document gives a high-level overview. Feel free to make use of anything you want from `java.util` and `java.lang`, but nothing else.

Compression

First we consider the process of compression.

Here is a brief description of the compression sequence, much of which has already been implemented for you. The compression sequence starts in a class implementing the `Compressor` interface. First it reads in the files and creates a list of frequencies for each data item. It passes these frequencies to the `HuffmanCode` constructor that takes a `Map`. Then the compressor calls the `writeHeader` method which emits the tree recursively, using `writer.writeByte(byte)` to emit each data item. The compressor then reads each symbol in the file and emits the code word by using `encode`

Expansion

We now consider expansion, the reverse of compression.

First expand reads the header from the file using the `HuffmanCode` constructor that takes a `BitReader`. The Constructor reads in the header, calling the `readByte` method of the `BitReader` whenever it sees a leaf in the file header. Then expand reads in each Huffman code and outputs the symbol.

The Huffman Compressor

Once you complete `HuffmanCode.java`, the provided implementation for `HuffmanCompressor.java` should work. Note the file format we use when compressing/decompressing files. We first encode the Huffman header (table for the tree outputted via `writeHeader`). Then we encode the number of symbols in the file, which corresponds to the number of times we need to call `encode`/`decode` (which for `HuffmanCompressor` is just the number of bytes in the file).

The implication with this file format is that when we have a character set of just one letter, we can define its Huffman code to be the empty string, rather than arbitrarily defining it to be 0 or 1. This actually makes it simpler because we don't have to deal with the special case (at least we shouldn't) when the root is a leaf node. Also, a file of all 'a's compresses better under this technique. Please test for this and make sure you get the appropriate output.

HuffmanCodeVisualizer

Like the TreeVisualizer for expression trees, there is a visualizer for Huffman trees that you may find interesting, but do not need to use.

LZW (20 Points)

LZW is a more sophisticated method of compression that relies on patterns in the files instead of just the frequency of the bytes. It can provide drastic improvements in compression ratios over the Huffman algorithm. When implemented with a trie, it also runs extremely quickly. You only need to read the file in once, and compress as you go.

In order for us to test that you are encoding and decoding correctly, you need to follow a few LZW conventions:

- Your program should detect invalid LZW codes and throw an exception (see javadocs). One such condition would be when you are reading in the codes in decompress and you see a code that isn't in the dictionary. If that code is just the next code you were going to insert into the dictionary, everything is fine. If the code, however, is greater than the *nextCode*, you need to throw an `IOException()`. This may seem like a royal pain, however not doing so has caused grave security issues in the real world.
- You must encode the initial dictionary in the following way. The dictionary will contain 256 entries, one for each ASCII byte value. The codes will be 32 bit zero-extended integers, one for each byte. So for example, you will encode the byte value 01100001 (which corresponds to the letter a) as 00000000000000000000000001100001. So just create 32 bit integer values from 0 to 255 and map them directly to the unsigned bytes (0000000, 00000001,...,11111111) they represent. While this may seem obvious, there are other ways to do the initial dictionary that are valid for LZW, but will cause you to fail our tests.
- You must read and write LZW codes using a variable-width encoding compatible with our conventions. To simplify this task, use the `LempelZivCodeReader` and `LempelZivCodeWriter` classes.

Be sure to call `flush` at the end of the compression and decompression methods. Otherwise, file output will be corrupt along with other potentially bad things.

FAQ

Pay special attention to this section. This is a collection of common questions students had in previous semesters.

- Q: How many elements should my PQ take before it runs out of memory
  - A: At least 100k items (ints).
- Q: I'm running out of heap space on some of the larger files. Is there any way I can fix this?
  - A: First make sure your code is correct :). For some of the larger files (like theorygirl.mp3) you may need to increase the JVM heap size. You can do this by using the JVM arguments -Xms512M -Xmx512M.
- Q: I've written good unit tests, and matched all the staff files exactly. So why am I still failing?
  - A: Make sure your compressors are stateless. This is mentioned in a previous section of the docs.

Handing it in

As always, hand in just the files you modified. If you hand in files that we give you, other than the ones with RuntimeExceptions where you implemented things, FrontDesk will *ignore them*. Also, please do not submit tons of random compressed files, filter them out before you zip.

As always, you can, and should, just export your project from Eclipse as an archive file. Then just submit that as usual.