CS211 Spring 2007
Assignment 3 — Huffman Coding
Due 14 March 2007, 11:59:59pm

1 General Instructions

1.1 Purpose

This assignment will involve Java interfaces, recursion, binary trees, file reading and writing, bitstring manipulation, priority queues, and the `Iterable` and `Comparable` interfaces. The main part of the assignment will be to implement a popular compression/decompression scheme based on Huffman trees.

This assignment involves a substantial amount of code, some of which we supply, but a lot of which you must write. Get started early! Read through the whole assignment before starting. In particular, there are some instructions in Section 3.5 regarding the organization of your submission.

In A1, we were rather explicit in the design of the programs to help you get started. In A2, we left a little more to your discretion. In this assignment, since we are taking a quantum jump in the amount of code you have to understand and write, we will again be fairly explicit in describing what you have to do.

1.2 Grading Criteria

As before, solutions will be graded on both correctness and style. For correctness, at the very least your program must compile without errors or warnings. For style, we like concise, clear, easy-to-read code. Use mnemonic variable names. Use proper indentation. Comment where necessary for clarification, but do not overcomment. Be concise; a two-page solution to a problem that can be done in a few lines will lose points, even if correct.

1.3 Partners

As in A2, you will be allowed to work in pairs. Please follow the instructions on the website for selecting a partner. You must create a group in CMS for you and your partner.

2 Strong Induction

Purpose: Understanding strong induction
Estimated time: 30 minutes
Points: 15

A binary tree is *complete* if every node has either 0 or 2 children. (It does not need to be balanced.) A node with no children is called a *leaf* and a node with two children is called an *internal node*. Prove that every nonempty complete binary tree with n leaves has exactly $2n - 1$ nodes in all.

3 Huffman Coding

Purpose: Using Java interfaces, recursion, binary trees, file reading and writing, bitstring manipulation, priority queues, the `Iterable` and `Comparable` interfaces
Estimated time: 25 hours
Points: 80

In this problem you will implement a data compression mechanism called *Huffman coding*. This is a useful mechanism for compressing data based on the frequencies of bytes in a file. The idea is that bytes

that occur very frequently are encoded as short bitstrings, while those that occur infrequently are encoded as long bitstrings.

A *bitstring* is just a sequence of bits, e.g. 10010110. Normally, printable characters in a file are encoded as *bytes*, or bitstrings of length 8. This is called ASCII code. For example, the ASCII code for the character 'a' is the bitstring 01100001, which is 97 in binary.

However, we are not forced to use a fixed-length encoding. If we choose a different encoding in which the codes can have different lengths, then we can assign shorter strings to the more frequent characters, thereby saving space.

To illustrate, let's do this for bitstrings of length 2. Suppose we have a file that is exactly 2000 bits long, so it can be viewed as a sequence of 1000 2-bit bitstrings. Suppose we scan the file and find that the four 2-bit bitstrings 00, 01, 10, 11 occur in the file with the frequencies as shown in Fig. 1.

00:	900
01:	90
10:	9
11:	1

Figure 1 Frequencies of 2-bit bitstrings in a file

00	→	0
01	→	10
10	→	110
11	→	111

Figure 2 A self-delimiting code

A good encoding for this set of frequencies would be the variable-length code shown in Fig. 2.

This particular encoding has the nice property that it is *self-delimiting*, which means that no codeword is a prefix of any other codeword. This property allows us to place the codewords end-to-end in a file without having to mark their endpoints, and still be able to recover the original text (see Section 3.3 below). For example, if the file begins with

00 00 00 10 10 11 01 00 00 00 00 01 ...

then it would be encoded as

0 0 0 110 110 111 10 0 0 0 0 10 ...

(We have inserted spaces just to show the original 2-bit bitstrings and the codewords, but the spaces are not really there in the file.) Given the correspondence of Fig. 2, the two sequences determine each other uniquely.

How much space have we saved? The original file was 2000 bits long. The length of the encoded file is

$$900 \cdot 1 + 90 \cdot 2 + 9 \cdot 3 + 3 \quad = \quad 1110$$

bits. We have compressed the file by almost half.

Tree Representation of Self-Delimiting Codes

A convenient representation of a self-delimiting code is a complete binary tree with the original uncompressed bitstrings at the leaves. The path from the root down to the leaf containing x, represented as a bitstring (left=0, right=1) is the codeword for x. For example, the code of Fig. 2 would be represented by the tree in Fig. 3. The tree in Fig. 4 shows the *identity code*, in which each word is represented as itself. This code doesn't achieve any compression.

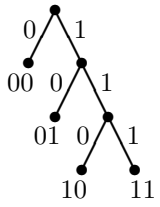

Figure 3

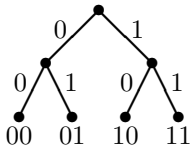

Figure 4

Huffman Trees

A *Huffman tree* is a tree built from known frequencies for bitstrings of some fixed constant length k. It tries to code frequently occurring bitstrings of length k as short bitstrings and infrequently occurring bitstrings of length k as long bitstrings. The tree of Fig. 3 is a Huffman tree for $k = 2$ constructed from the frequencies of Fig. 1. In this assignment, you will implement Huffman trees for $k = 8$ (bytes).

To build a Huffman tree, suppose we know the number of occurrences of each k-bit bitstring in the file. We make a separate tree node for each k-bit bitstring containing that bitstring as the *key* along with its frequency; these will be the leaves of our Huffman tree when we are done. So we start with a forest of 2^k single-node trees.

Now we build the Huffman tree bottom up using a *greedy algorithm*. We first put all these tree nodes into a *priority queue*. A priority queue is an ADT that supports the following operations:

1. Insert an element.
2. Extract the minimum element.

In our case, *minimum* will mean minimum frequency. Now we repeatedly do the following, as long as the priority queue contains at least two elements:

1. Extract the two elements with the lowest frequency x, y from the priority queue.
2. Make a new tree node with children x and y and frequency the sum of the frequencies of x and y. Call this new node z.
3. Insert z back into the priority queue.

We keep doing this until there is exactly one element remaining, which will be the root of the Huffman tree. This has to happen eventually, since the size of the priority queue decreases by one in each stage.

For example, suppose we wish to form a Huffman tree from the frequencies as shown in Fig. 1. We start with the four tree nodes $(00, 900)$, $(01, 90)$, $(10, 9)$, $(11, 1)$. In the first step, the priority queue will extract the two lowest-frequency elements $(10, 9)$ and $(11, 1)$, form a new tree node $(u, 10)$ with children $(10, 9)$ and $(11, 1)$, then insert this back into the priority queue. Here u is a new key that we assign. After this step, the priority queue will contain $(00, 900)$, $(01, 90)$, $(u, 10)$. Doing this again, after the next step the priority queue will contain $(00, 900)$ and $(v, 100)$, where v is a new node with children $(u, 10)$ and $(01, 90)$. Finally, the last two nodes are combined, yielding a node $(w, 1000)$ which is the root of the Huffman tree. The resulting Huffman tree looks like Fig. 3.

3.1 What To Do

In this problem you will write a *Huffman codec*. The term *codec* is short for a compression/decompression (or coding/decoding) scheme. Your codec will implement the `a3.Codec` interface, which means it must supply two methods:

```
void compress(File input, File output) throws IOException;
void decompress(File input, File output) throws IOException;
```

Implementing this interface allows your codec to be plugged into a client program that wants to do compression and decompression. We have supplied such a program, which we will discuss later (Section 3.4).

In the following two sections, we will go over carefully what you have to do for compression and decompression.

3.2 Compression

3.2.1 Constructing the Huffman Tree

To construct the Huffman tree from the uncompressed `inputFile`, you first need to calculate the byte frequencies in the file. Open the file for reading by calling

```
FileInputStream inStream = new FileInputStream(inputFile);
```

You can then read bytes sequentially from the file by calling `inStream.read()` successively. This method returns an `int` containing the next byte in the file in the low-order 8 bits. For example, if the input byte was 01100001 (binary) representing the ASCII character 'a', the `int` returned by `inStream.read()` would be 97. You can use this `int` as an index into an array of length $256 = 2^8$, the total number of possible bytes. Go through the file once and record the byte frequencies in an `int` array of length 256.

Now construct the Huffman tree as described above. You should use a `java.util.PriorityQueue` of `TreeNode`s for this. First construct a new priority queue by calling

```
PriorityQueue<TreeNode> pq = new PriorityQueue<TreeNode>();
```

then insert a new `TreeNode` for each byte and its frequency. These will be the leaves of the soon-to-be-built Huffman tree.

The minimum element of `pq` is extracted by calling `pq.poll()`. But how does it know what *minimum* means? You have to tell it. You do this by making the `TreeNode` class implement the `java.util.Comparable` interface. To implement this interface, you must define a method `int compareTo(Object obj)`. The return value of this method should be

negative	if `this` is less than `obj`,
0	if `this` is equal to `obj`, and
positive	if `this` is greater than `obj`.

Your `compareTo` method can do this by casting `obj` to a `TreeNode` (checking for a class cast error, of course), comparing the frequencies, and returning the appropriate `int`. The class `PriorityQueue` does not know (or care) what `compareTo` does, it just calls it blindly to determine the minimum `TreeNode` to extract. So it can work with all kinds of different data objects, as long as they are `Comparable`. Isn't abstraction a beautiful thing?

Your `TreeNode` class should contain `int` fields `key` for the key (which for leaves will be the uncompressed byte) and `frequency` for the frequency, `TreeNode` fields `left` and `right` for the children, and a `compareTo` method. You may also include a `toString` method for debugging purposes if you wish, as well as any other needed fields or helper methods. When constructing the tree, you should give each newly constructed internal node a `key` consisting of a number assigned sequentially, starting at 256. All the leaves will already have `key`s less than 256. By Problem 2, the root of the tree should have `key` value $2 \cdot 256 - 2 = 510$. You might want to include this test as a sanity check.

Constructing the Byte-to-Codeword Map

To do the coding, we will need an efficient way to map each incoming byte to its corresponding codeword. We could search the tree every time for the byte, but this would be very inefficient. We will instead do a little preprocessing to construct an array `codes` of length 256 whose kth element is the codeword for k. Then while we are coding, we will be able to map incoming bytes to codewords by just pulling the appropriate codeword out of the array.

We have provided a class `BitVector` to represent codewords. This class gives an efficient reresentation of a bitstring with interface similar to `ArrayList` or `Vector`; Javadoc documentation is provided.

You will construct the codewords by walking the tree recursively, filling in the `codes` array when you encounter a leaf. This should be done with a recursive method `createCodes(TreeNode n, BitVector bv)`. The method should just assign `bv` to the key of `n` if `n` is a leaf. Otherwise, if `n` is an internal node, it should clone `bv`, add `true` to one copy and call recursively on the right subtree, and add `false` to the other copy and call recursively on the left subtree. If you do this right, it is only a few lines of code.

Saving the Huffman Tree

The compressed file will contain the Huffman tree followed by the sequence of codewords corresponding to the bytes of the input file. You must first write the Huffman tree to the output file.

Open the output file for writing as a `BitStream` with

```
BitStream outStream = new BitStream(outFile, "w");
```

The `"w"` means "write". The class `BitStream` is a class for reading and writing streams of bits as opposed to bytes. We have provided this class and Javadoc documentation for it.

Write the keys in the tree in *preorder* to the output file. Write just the keys, nothing else. Use the `outStream.writeBits` method to write each key. You are writing an integer, but you only want to write 9 bits of it. (All keys are between 0 and 511, which only requires 9 bits. We are trying to save space, remember?) The `outStream.writeBits` method allows you to specify the number of bits to write. Again, this can be done recursively with only a few lines of code.

When decompressing, we will be able to reconstruct the tree from the preorder sequence of keys.

Compressing the Input File

Now we are ready to compress the input file. Close the input stream `inStream` and reopen it so we can read the bytes again from the beginning. The output stream should still be open. Read the bytes sequentially from the input stream. For each byte, get its codeword, then write the codeword to the output bitstream.

When you are done, make sure you close the input and output streams. This is especially important for the output stream, because bits are buffered, and not all of them may have been written to the output file yet. Moreover, since the number of bits is not necessarily a multiple of 8, the actual bit count must be saved in the output file. The `BitStream.close()` method takes care of all this for you, but you have to remember to call it when you are done.

3.3 Decompression

Reconstructing the Huffman Tree

Before we can decompress, we must reconstruct the Huffman tree that was saved in the compressed file as a preorder sequence of keys. In the decompression phase, the frequencies are not used.

Open the input file as a `BitStream` for reading. Read in the sequence of 9-bit bitsequences that were the keys of the Huffman tree in preorder into an `Iterable<Integer>` such as `ArrayList<Integer>` or `Vector<Integer>`.

Now we will reconstruct the tree recursively. The recursive procedure should take an `Iterator` and return a `TreeNode`, which will be the root of the tree constructed from the keys in the `Iterator`. The recursive procedure will be called with the initial `Iterator` obtained from the `Iterable` object containing the keys that you read in. The recursive procedure does the following:

1. Get the next key from the `Iterator`.
2. Make a new `TreeNode n` with that key. This is the root of the subtree we are currently visiting.
3. If we are not at a leaf, do two recursive calls to construct the left and right subtrees and store them as the children of `n`. We can tell whether we are at a leaf just from the key. How?
4. Return `n`.

Decompressing the Input File

After reading in the Huffman tree, the remainder of the input file contains the codewords corresponding to the bytes of the uncompressed file. Now we want to decompress these codewords and map them back to the original bytes.

Open the output file as a `FileOutputStream`. We can write bytes to it using the `write(int)` method of `FileOutputStream`.

Set a pointer to the root of the Huffman tree. Read the bits of the input file one at a time and follow the path in the tree as determined by those bits (0=left, 1=right). When we arrive at a leaf, that leaf contains the uncompressed byte corresponding to the codeword we have just seen. Write that byte to the output stream and reset the pointer to the root of the tree for the next codeword. Note that this works since the code is self-delimiting: we know when we are at the end of a codeword, since that occurs exactly when we are at a leaf of the tree. Beautiful!

When done, remember to close the input and output streams.

3.4 Getting Started

The file `a3.jar` contains a lot of code for your use, including the `BitVector` and `BitStream` classes. There is full Javadoc documentation on the website. The source code is also available in the file `a3source.zip` if you want to take a look at it. This code is all in the package `a3`, so you have to import classes from this package.

We have provided a client program `Zip` that is very much like a simplified version of WinZip (Windows) or tar (Unix). It is an archive program that can zip several compressed files together into an archive, then extract and uncompress them again later. It has a nice graphical user interface (gui). You will be working with a gui something like this in A4 and A5, so if you want to get a head start, take a look at the source to see how it is built.

We have also provided two simple plug-in codecs, neither one of which does any compression. One of the codecs is just the trivial identity codec that maps the input file to the output file verbatim. The other is a "rot13" codec that exchanges 'a' and 'n', 'b' and 'o', etc. Both of these codecs are self-inverses, so the `compress` and `decompress` methods are identical.

To run the `Zip` program with a specified `Codec codec`, call `new Zip(codec).run()`. For example, you can run it with the `Rot13` codec by calling `new Zip(new Rot13()).run()`. Note that a file compressed with one codec will produce garbage when decompressed with a different codec!

With your Huffman codec, you should see about 25 to 40% compression on text files.

We have supplied a text file `text.txt`. Compress this with your Huffman codec and submit the compressed file `compressed.txt`. Do not use our archive program for this, just give us the raw output of your `compress` method. You will have to write a little driver to call your `compress` method on these files, but don't submit that.

3.5 What to Submit

Submit files `TreeNode.java`, `HuffmanTree.java`, and `Huffman.java`, each containing a class of the same name. The class `TreeNode` should contain your definition of a tree node. The class `HuffmanTree` should encapsulate a Huffman tree; it should have two constructors, one for building a tree from an array of frequency counts and one for building a tree from an `Iterable<Integer>` of keys in preorder. The code for constructing the byte-to-codeword mapping should also go in this class; the array representing the mapping should be private and accessed from the outside through a getter method. The class `Huffman` should be your codec. Finally, submit the file `compressed.txt` that you have compressed from the supplied text file `text.txt`.

4 Submitting Your Work

Points: 5

Submit the following files in CMS:

- Exercise 2: `Induction.txt`
- Exercise 3: `TreeNode.java`, `HuffmanTree.java`, `Huffman.java`, `compressed.txt`

Do not submit any other files.