CSCI 230 Data Structures & Algorithms

Project 3 Huffman coding[1]

Crafton Hills College
Total Points: 80

BE SURE TO READ AND UNDERSTAND THE PROJECT INSTRUCTIONS BEFORE YOU START ANY CODING.

Overview

For this assignment you'll implement several classes to build what are conceptually two programs: one to compress (huff) and the other to uncompress (unhuff) files that are compressed by the first program. However, there is really just a single program with the choice of reading a compressed file or a normal file specified by choosing a menu-option in the GUI-based program.

The `Huff` class is a simple main that launches a GUI with a connected model that you'll write. This will eventually be the single application/GUI you use to compress/huff files or decompress/unhuff files.

Most of the work will be in the classes you implement that correspond to interfaces you're given as part of this assignment. You're writing code based on the greedy Huffman algorithm in the detailed explanation of Huffman Coding that is provided with this project (*Project 3 READ THIS Huffman Coding Explanation.pdf*). Be sure to read and understand it before beginning this project.

The resulting program will be a complete and useful compression program although not, perhaps, as powerful as standard programs like compress or zip which use slightly different algorithms permitting a higher degree of compression than Huffman coding.

Programming Advice

Because the compression scheme involves reading and writing in a bits-at-a-time manner as opposed to a char-at-a-time manner, the program can be hard to debug. In order to facilitate the design/code/debug cycle, you should take care to develop the program in an incremental fashion. If you try to write the whole program at once, you probably will not get a completely working program!

Design, develop, and test so that you have a working program at each step.

Build a program by adding working and tested pieces.

[1] Source: http://www.cs.duke.edu/csed/poop/huff/fall06/

THE HUFFMAN COMPRESSION PROGRAM

You'll implement several classes to create the program. You're given several interfaces as part of the code framework. The implementation details of the classes and methods are completely up to you, but a good design will receive maximal points.

To compress you should use the Huffman coding algorithm. The algorithm has four steps. You should understand each of these steps before starting to code. The first three steps are the basis for Part I, which you should design, implement, and test before proceeding to the next part and step four.

First, the process of Huffman compression is discussed, then some details about the program follow.

1. To compress a file, count how many times every character occurs in a file. These counts are used to build weighted nodes that will be leaves in the Huffman tree. The word character is used, but we mean 8-bit chunk and this chunk-size could change. Do not use any variables of type char in your program. Use only ints. Use an `ICharCounter` class to do the counting.
2. From these counts build the Huffman tree. First create one node per character, weighted with the number of times the character occurs, and insert each node into a priority queue. Then choose two minimal nodes, join these nodes together as children of a newly created node, and insert the newly created node into the priority queue. The new node is weighted with the sum of the two minimal nodes taken from the priority queue. Continue this process until only one node is left in the priority queue. This is the root of the Huffman tree. Use an `ITreeMaker` class to create the tree.
3. Create a table or map of characters (8-bit chunks) to codings. The table of encodings is formed by traversing the path from the root of the Huffman tree to each leaf, each root-to-leaf path creates an encoding for the value stored in the leaf. When going left in the tree append a zero to the path; when going right append a one. All characters/encoding bit pairs may be stored in some kind of table or map to facilitate easy retrieval later. The table can be an array of the appropriate size (roughly 256, but be careful of PSEUDO_EOF) or you can use a `Map` subclass. Use an `IHuffEncoder` class to create the table from the tree.
4. Finally, read the input file a second time. For each character/8-bit chunk read, write the encoding of the character (obtained from the map of encodings) to the compressed file.

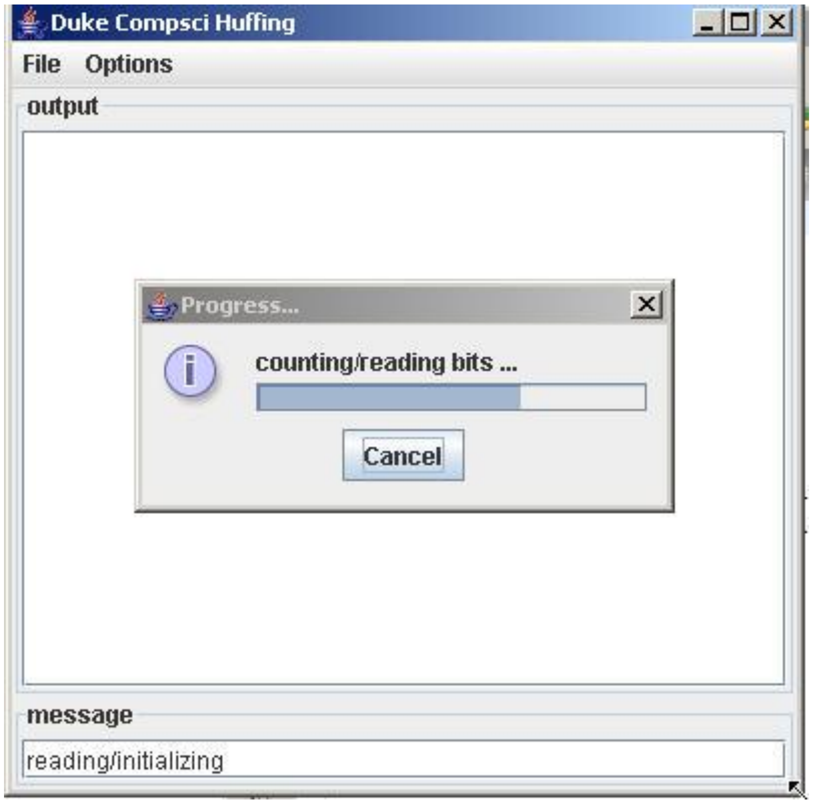
To uncompress the file later, you must recreate the same Huffman tree that was used to compress (so the codes you send will match). This tree might be stored directly in the compressed file (e.g., using a preorder traversal), or it might be created from character counts stored in the compressed file. In either case, this information must be coded and transmitted along with the compressed data (the tree/count data will be stored first in the compressed file, to be read by `unhuff`. There's more information below on storing/reading information to re-create the tree.

Part I

For part one you should implement a class that implements the `IHuffModel` interface so that all methods except write work. You don't actually turn in Part I separately, but it's a very good idea to make this part work before proceeding to Part II and completion of the entire program.

Specifically, for part one your `IHuffModel` implementation class must be connected to the GUI/Viewer and must implement three of the methods from the `IHuffModel` interface:

- Implement `initialize`.
  - This involves reading a stream and updating local state. You'll need to store counts of characters in some local state (use the `ICharCounter` interface).
  - The GUI should display something similar to the screen shot below when the reading starts and when it is nearly finished --- this will happen automatically if you read from the `InputStream` passed to `initialize`.

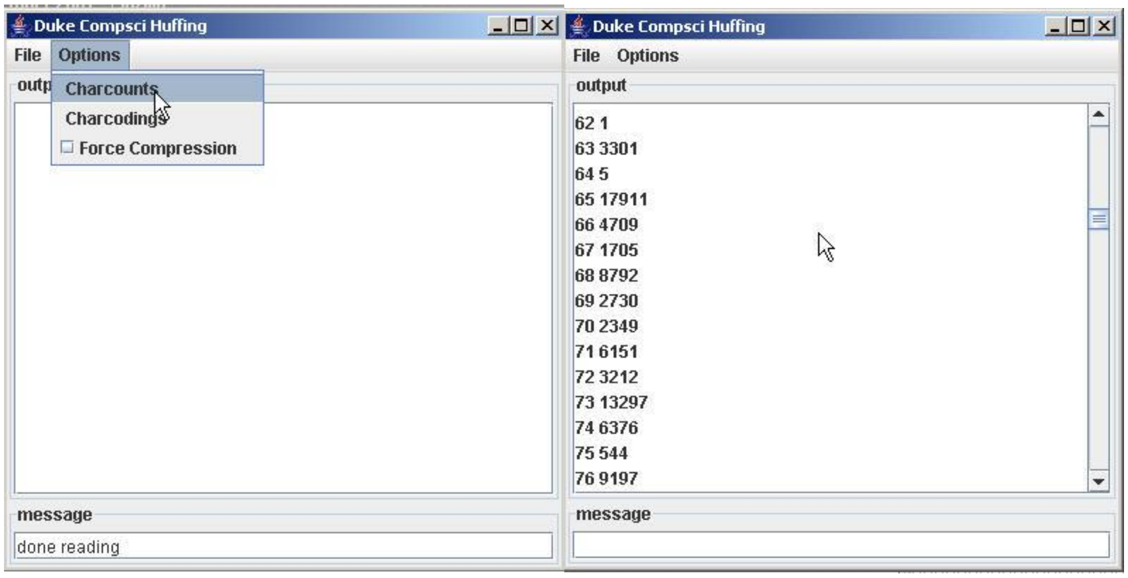


counting/reading bits ...

reading/initializing

- Implement `showCounts`.
  - This should call the view's `update` method, displaying per-character counts via the view/display. The counts represent the data read during initialization. Here's a screenshot. Recall that you pass a `Collection`, e.g., an `ArrayList` to the `HuffViewer.update` method, where the list contains strings that will be displayed.

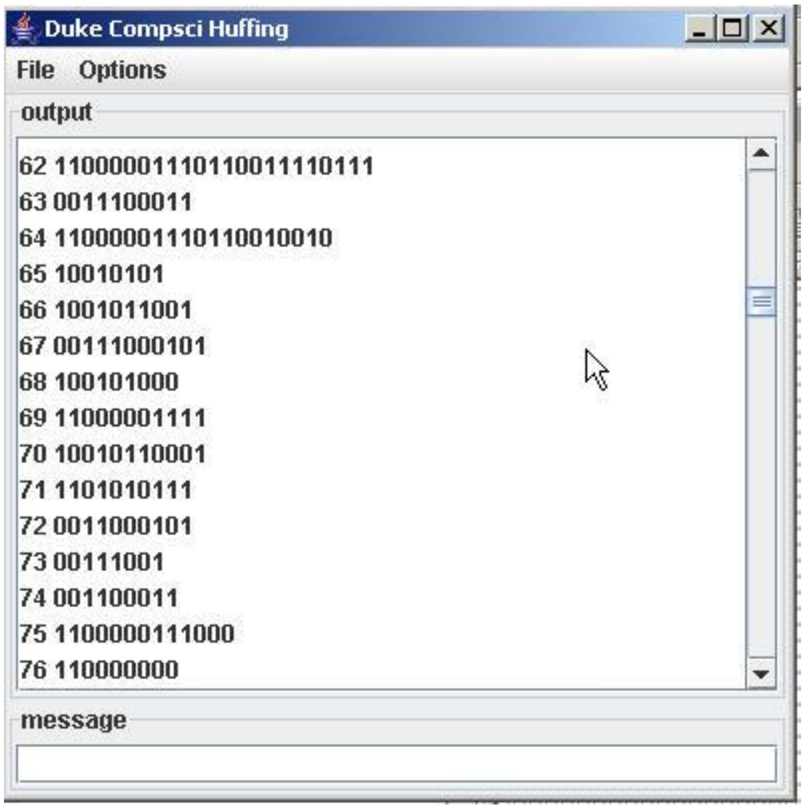
The strings should contain all int values with associated counts for chars/ints in the alphabet (e.g., 0-255). The screenshot below on the left shows how to choose "show character counts" from the *Options* menu; the screen on the right shows character counts for the file `kjv10.txt`, note that there are 17,911 occurrences of an uppercase 'A' whose unicode/ASCII value is 65.



```
62 1
63 3301
64 5
65 17911
66 4709
67 1705
68 8792
69 2730
70 2349
71 6151
72 3212
73 13297
74 6376
75 544
76 9197
```

done reading

- Implement `showCodings`.
    - Displaying per-character encodings via view/display. Each line shown gives a character/chunk number and the encoding for that value based on the huffman tree that comes from the counts. You'll need to make a tree and a map/table (see below) to get these encodings. In the screen shot you can see the encoding for 'A' is smaller in length then encoding for 'F' (ASCII/unicode 70) which makes sense given the number of occurrences shown above.



```
62 11000001110110011110111
63 0011100011
64 11000001110110010010
65 10010101
66 1001011001
67 00111000101
68 100101000
69 11000001111
70 10010110001
71 1101010111
72 0011000101
73 00111001
74 001100011
75 1100000111000
76 110000000
```

- Implement `setViewer`.
    - You must implement this method, or the viewer will fail to attach itself to the model. You'll need to store a viewer in the model's private state.

For more information about Part I code and design, see Part 1 Coding and Algorithmic Details.

Part II Writing and Reading Compressed Files

To complete the program, you'll need to write a compressed file based on the encodings and you'll need to read a compressed file and uncompress it.

When you uncompress, the uncompression program/methods must be able to recreate the Huffman tree which was used to compress the file originally, then use this tree to recreate the original (uncompressed) file. Using the interfaces described in Part I, and here, will help ensure you don't duplicate too much code in writing the compress/uncompress classes and code.

Your compression and uncompression methods work together. A file compressed using someone else's compression code probably can't be uncompressed by your code unless you agree on several standards that aren't part of this assignment.

Writing a Compressed File

There are three steps in writing a compressed file from the information your code stored/maintained in Part I (the counts and encodings).

1. Write a *magic number* at the beginning of the compressed file. This is explained more fully below, but your code uses the magic number to determine if a compressed file was created by your program when uncompressing.
2. Write information after the magic number that allows the Huffman tree to be recreated. The simplest thing to do here is write `ALPH_SIZE` counts as int values, but you can also write the tree. This is described more fully below.
3. Write the bits needed to encode each character of the input file. For example, if the coding for 'a' is "01011" then your code will have to write 5 bits, in the order 0, 1, 0, 1, 1 every time the program is compressing/encoding the chunk 'a'.

MAGIC NUMBERS

One easy way to ensure that both programs work in tandem is to write a magic number at the beginning of a compressed file. This could be any number of bits that are specifically written as the first N bits of a huffman-compressed file (for some N). The corresponding decompression program first reads N bits, if the bits do not represent the "magic number" then the compressed file is not properly formed. You can read/write bits using the classes declared in the `BitInputStream` and `BitOutputStream` classes.

For example, in my working program I have the following lines in different parts of my class that implements the `IHuffHeader` interface.

```
 // write out the magic number

out.write(BITS_PER_INT, MAGIC_NUMBER);
```

then in another part of the class (in another method)

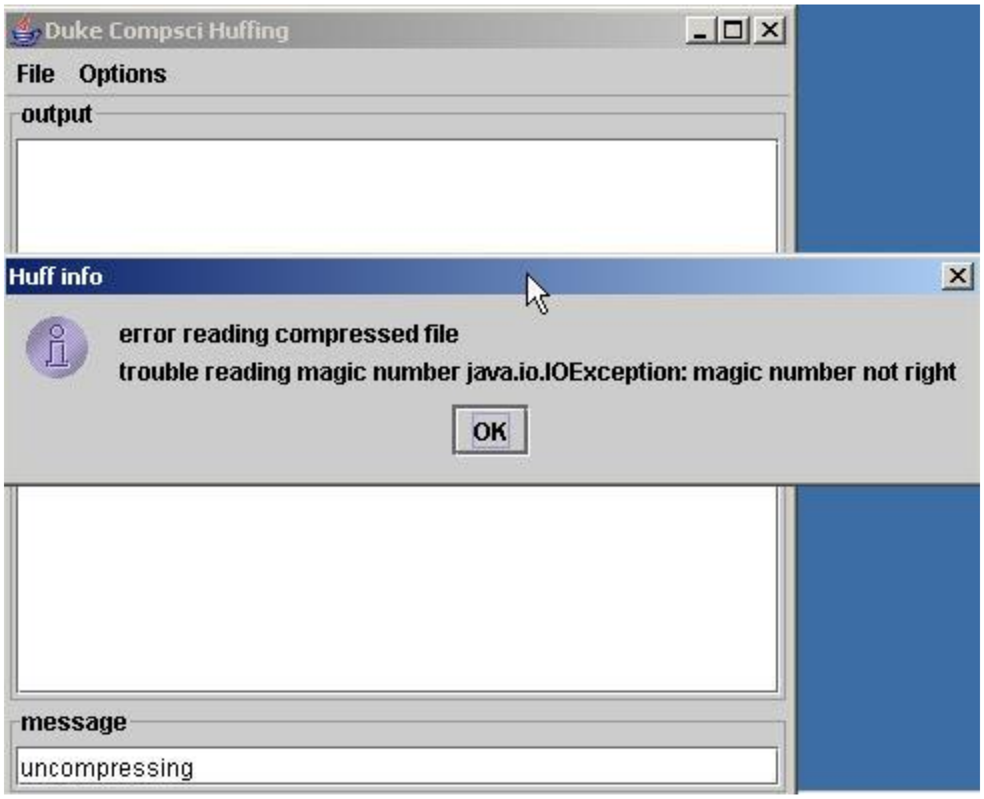
```
int magic = in.read(BITS_PER_INT);

if (magic != MAGIC_NUMBER){

    throw new IOException("magic number not right");

}
```

In general, a file with the wrong magic number should not generate an error, but should notify the user. For example, in my program the exception above ultimately causes the user to see what's shown below. This is because the exception is caught and the viewer's `showError` method called appropriately. Your code should at least print a message, and ideally generate an error dialog as shown.



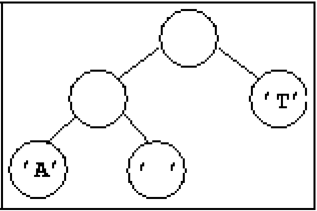
error reading compressed file
trouble reading magic number java.io.IOException: magic number not right

uncompressing

Storing the Huffman Tree

For decompression to work with Huffman coding, information must be stored in the compressed file which allows the Huffman tree to be re-created so that decompression can take place. There are many options here. You can store all codes and lengths as normal (32 bit) int values or you can try to be inventive and save space. For example, it is possible to store just chunk/character counts and recreate the codes from the counts (i.e., store 256 counts, one for each 8-bit character). It's also possible to store code-lengths and codes using bit-at-a-time operations. Any solution to storing information in the compressed file is acceptable, but full credit requires some attempt to save space/storage. Space-saving techniques are defined as those using less space than simply storing 256 counts as 32 bit ints. One useful technique is to write the tree to the file using a preorder traversal. You can use a 0 or 1 bit to differentiate between internal nodes and leaves, for example. The leaves must store character values (in the general case using 9-bits because of the pseudo-eof character).

For example, the sequence of 0's and 1's below represents the tree on the right (if you write the 0's and 1's the spaces wouldn't appear, the spaces are only to make the bits more readable to humans.)

`0 0 1 001100001 1 000100000 1 001110100`	

The first 0 indicates a non-leaf, the second 0 is the left child of the root, a non-leaf. The next 1 is a leaf, it is followed by 9 bits that represent 97 (001100001 is 97 in binary), the ASCII code for 'a'. Then there's a 1 for the right child of the left child of the root, it stores 32 (000100000 is 32 in binary), the ASCII value of a space. The next 1 indicates the right child of the root is a leaf, it stores the ASCII value for a 't' which is 116 (001110100 is 116 in binary).

Your program can write these bits using a standard pre-order traversal. You can then read them by reading a bit, then recursively reading left/right subtrees if the bit is a zero (think about the 20-questions/animal program).

Even storing only non-zero counts qualifies as space-savings. To store non-zero counts, however, you'll need to store the character/chunk being counted and this might not save space. In my non-saving-space code, my header is written by the following code. Note that `BITS_PER_INT` is 32 in Java.

```
for(int k=0; k < ALPH_SIZE; k++){
    out.write(BITS_PER_INT, myCounter.getCount(k));
}
```

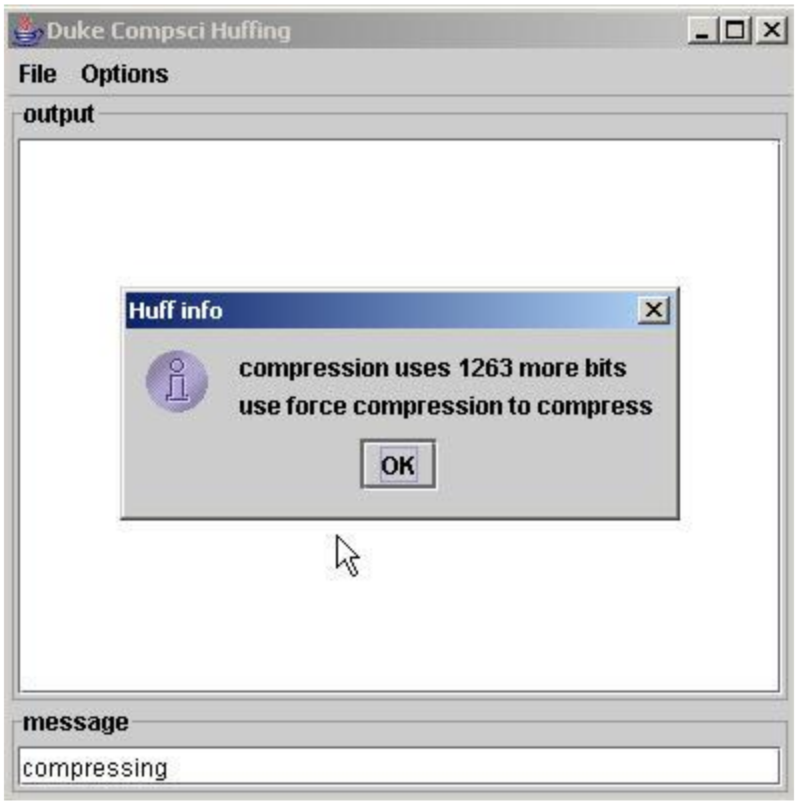
This header is then read as follows, this doesn't do much, but shows how reading/writing the header are related.

```
for(int k=0; k < ALPH_SIZE; k++){
    int bits = in.read(BITS_PER_INT);
    charcounter.set(k,bits);
}
```

In my code to read/write the header as a tree, the resulting header is much smaller. Both reading/writing are done in classes that implement `IHuffHeader`.

Forcing Compression

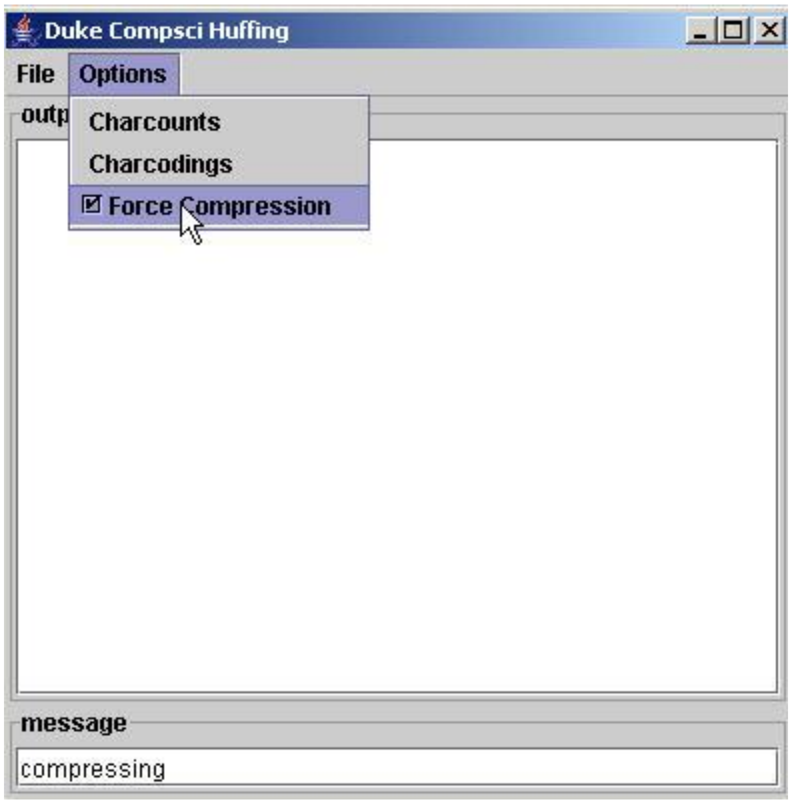
If compressing a file results in a file larger than the file being compressed (this is always possible) then no compressed file should be created and a message should be shown indicating that this is the case. Here's a screen shot from what happens in my program.



compression uses 1263 more bits
use force compression to compress

compressing

If the user forces compression using the Options menu as shown below then compression occurs even if the compressed file is bigger.



compressing

Determining If Compression Happens

To determine if compression results in a smaller file, you'll need to calculate the number of characters/chunks in the original file (your program will compute this by determining character/chunk counts). The size of the compressed file can be calculated from the same counts using the size of each character's encoded number of bits. You must also remember to calculate the file-header information stored in the compressed program. To be more precise, if there are 52 A's, and each A requires 4 bits to encode, then the A's contribute 52*4 = 108 bits to the compressed file. You'll need to make calculations like this for all characters.

The `IHuffHeader` interface specifies a method `headerSize` to help with keeping the code for headers in one place.

Other Requirements

1. Object-oriented design principles, including inheritance, data encapsulation, and information hiding, should be used.
2. Code should be well-documented, including all classes and class members. Follow the style guidelines for the Javadoc tool (http://www.oracle.com/technetwork/articles/java/index-137868.html).
3. All source code and runnable jar file must be submitted properly on Blackboard. (IMPORTANT: if I am not able to read your source files and/or run your executable, your project will not be graded. It is your responsibility to make sure all files are submitted correctly before the deadline.)

Creating a New Runnable JAR File

To create a new runnable JAR file in the Eclipse workbench:

- From the menu bar's **File** menu, select **Export**.
- Expand the Java node and select **Runnable JAR** file. Click Next.
- In the Runnable JAR File Specification page, select a 'Java Application' launch configuration to use to create a runnable JAR.
- In the **Export destination** field, either type or click **Browse** to select a location for the JAR file.
- Select an appropriate library handling strategy. Choose “Package required libraries into generated JAR”.
- Be sure to test the runnable JAR file by opening it.

What you need to submit for the project:

- A report containing the following items:
  - Your name
  - Program Design Explanation
    - How is your priority queue implemented? Why did you choose such implementation?
    - What data structure did you use to store the encodings? Why did you choose such data structure?
    - How is the Huffman tree stored in your compressed file? Why did you make this choice?
  - A brief discussion of your project experience
    - Did you enjoy this project? What problems did you encounter?
    - What did you get out from the project?
    - How did you find the project (too easy, easy, just right, difficult, too difficult)?
    - What type of help/references did you use in your project (e.g. book, web sites, classmates, tutors)? List their names.
    - If you work with a partner, describe the roles of each partner.

If you work in a group, only one group member submits the project (code, report, executable, etc.).

Grading Criteria

- Satisfaction of project requirements (60 points)
  - compression of any text file
  - compression of any file (including binary files)
  - decompression
  - robustness (does unhuff program crash on non-huffed files?)
  - char count/char codings (Part I)
  - program design
- Report (10 points)
- Documentation and coding style (10 points)
  - Provide sufficient documentation in the source code. **Each class and method should be documented clearly.**
  - Your name should be present in each user-defined class.
  - Use descriptive identifiers.
  - Use proper spacing and indentation (refer to the textbook's program style).

APPENDIX

Part I Coding and Algorithmic Details

There are many details that you will need to think about as you code these programs. Some of these are discussed here with some alternatives proposed.

Note, **do not use any variables of type char**! You should use int variables when you think you might need a char everywhere in your program. The only time you might want to use a char variable is to print for debugging purposes -- you can cast an int to a printable char as shown in the code fragment below.

```
int k = 'A';
System.out.println(char(k));
```

Pseudo-EOF character

The operating system will buffer output, i.e., output to disk actually occurs when some internal buffer is full. In particular, it is not possible to write just one single bit to a file, all output is actually done in "chunks", e.g., it might be done in eight-bit chunks. In any case, when you write 3 bits, then 2 bits, then 10 bits, all the bits are eventually written, but you cannot be sure precisely when they're written during the execution of your program. Also, because of buffering, if all output is done in eight-bit chunks and your program writes exactly 61 bits explicitly, then 3 extra bits will be written so that the number of bits written is a multiple of eight. Because of the potential for the existence of these "extra" bits when reading one bit at a time, you cannot simply read bits until there are no more left since your program might then read the extra bits written due to buffering. This means that when reading a compressed file, you CANNOT use code like this (only because the number of bits isn't a multiple of 8 in a compressed file).

```
int bits;
while ((bits = input.read(1)) != -1)
{
    // process bits
}
```

To avoid this problem, you can use a pseudo-EOF character and write a loop that stops when the pseudo-EOF character is read in (in compressed form). The code below is pseudo-code for reading a compressed file using such a technique.

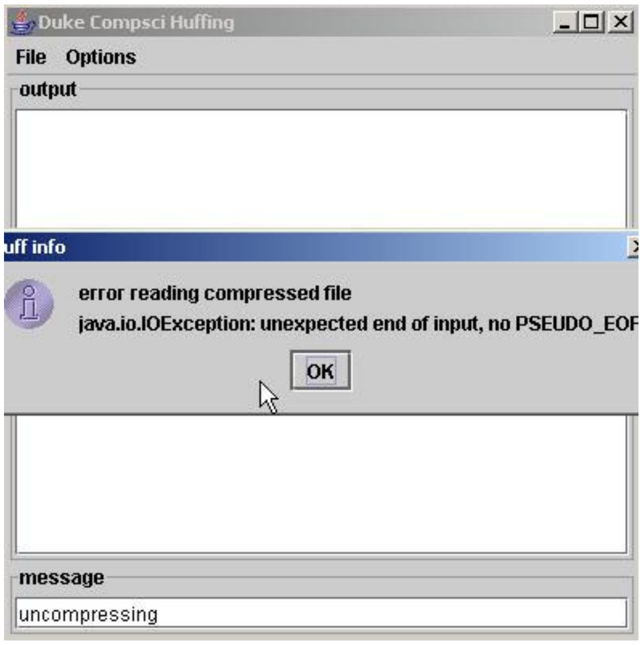
```
int bits;
while (true)
{
    if ((bits = input.read(1)) == -1)
```

```
            {
                System.err.println("should not happen! trouble reading bits");
            }
            else
            {
                // use the zero/one value of the bit read
                // to traverse Huffman coding tree
                // if a leaf is reached, decode the character and print UNLESS
                // the character is pseudo-EOF, then decompression done

                if ( (bits & 1) == 0) // read a 0, go left  in tree
                else                  // read a 1, go right in tree

                if (at leaf-node in tree)
                {
                    if (leaf-node stores pseudo-eof char)
                        break;    // out of loop
                    else
                        write character stored in leaf-node
                }
            }
        }
```

When a compressed file is written the last bits written should be the bits that correspond to the pseudo-EOF char. You will have to write these bits explicitly. These bits will be recognized uncompressing used in the decompression process. This means that your decompression program will never actually run out of bits if it's processing a properly compressed file (you may need to think about this to really believe it). In other words, when decompressing you will read bits, traverse a tree, and eventually find a leaf-node representing some character. When the pseudo-EOF leaf is found, the program can terminate because all decompression is done. If reading a bit fails because there are no more bits (the bit-reading method returns -1) the compressed file is not well formed. **Your program should cope with files that are not well-formed, be sure to test for this**, i.e., test unhuff with plain (uncompressed) files.



error reading compressed file
java.io.IOException: unexpected end of input, no PSEUDO_EOF

uncompressing

In Huffman trees/tables you use in your programs, the pseudo-EOF character/chunk always has a count of one --- this should be done explicitly in the code that determines frequency counts. In other words, a pseudo-char EOF with number of occurrences (count) of 1 must be explicitly created.

In the file `IHuffConstants` the number of characters counted is specified by `ALPH_SIZE` which has value 256. Although only 256 values can be represented by 8 bits, these values are between 0 and 255, inclusive. One character is used as the pseudo-EOF character -- it must be a value not-representable with 8-bits, the smallest such value is 256 which requires 9 bits to represent. However, ideally your program should be able to work with n-bit chunks, not just 8-bit chunks.

Priority Queues

The easiest way to build the Huffman tree is to use a priority queue since it supports `peek` and `remove` operations which are used in creating the Huffman tree. It always returns the minimum count element/node (or subtree). When more than one character/node has the same count, it picks one to return (it always picks them in the same order, but it does so in no particular order). It turns out this is not a problem for the decompressor as long as both programs use the same implementation of a priority queue.

You're given a `TreeNode` that implements `Comparable`. You can use this class in storing weighted character/chunk objects in a priority queue to make a Huffman tree.

Creating a Table from a Huffman-tree

To create a table or map of coded bit values for each character you'll need to traverse the Huffman tree (e.g., inorder, preorder, etc.) making an entry in the table each time you reach a leaf. For example, if you reach a leaf that stores the character 'C', following a path left-left-right-right-left, then an entry in the 'C'-th location of the map should be set to 00110. You'll need to make a decision about how to store the bit patterns in the map. At least two methods are possible for implementing what could be a class *BitPattern*:

- Use a String. This makes it easy to add a character (using +) to a string during tree traversal and makes it possible to use `String` as `BitPattern`. Your program may be slow because appending characters to a string (in creating the bit pattern) and accessing characters in a string (in writing 0's or 1's when compressing) is slower than the next approach.
- Alternatively you can store an integer for the bitwise coding of a character. You need to store the length of the code too so your code will be able to differentiate between 01001 and 00101. However, using an int restricts root-to-leaf paths to be at most 32 edges long since an int holds 32 bits. In a pathological file, a Huffman tree could have a root-to-leaf path of over 100. **Because of this problem, you should use strings to store paths rather than ints. A slow correct program is better than a fast incorrect program.**

This means you'll need to follow every root-to-leaf path in the Huffman tree, building the root-to-leaf path during the traversal. When you reach a leaf, the path is that leaf value's encoding. One way to do this is with a method that takes a `TreeNode` parameter and a `String` that represents the path to the node. Initially the string is empty "" and the node is the global root. When your code traverses left, a `"0"` is added to the path, and similarly a `"1"` is added when going right.

```
...
...
   recurse(root.left, path + "0");
   recurse(root.right, path + "1");
```

Implementing and Debugging

It's a good idea to create several classes to help manage the complexity in these programs. Because the same data structures need to be used to ensure that a file compressed using your huff algorithm can be decompressed, you should be able to share several parts of the implementation. You can use classes to exploit this similarity. For example, in writing `huff` you can implement several classes, usually a class will take care of one part of the Huffman algorithm. These classes should also provide support for the decompression program `unhuff`.

Some ideas for classes are given below. You may decide to combine some of these into one class, or you may decide to implement more classes. These are a start and some suggestions, not requirements.

Counting Characters

A class could be responsible for creating the initial counts of how many times each character occurs. You will be required to design, implement, and test the class in isolation from the rest of the `huff` program. Doing this for each of several classes will help both in developing `huff` and in re-using code in writing the uncompress program `unhuff`.

You should use the ICharCounter interface for doing this.

For example, one possibility for using a character counting class is shown below:

```
IcharCounter cc = new SimpleCounter();
BitInputStream bit = new BitInputStream(...);
cc.countAll(bit);
for (int k = 0; k < ALPH_SIZE; k++) {
    int occs = cc.getCount(k);
    if (occs > 0) {
        System.out.println(char(k)+ " occurs "+ occs);
    }
}
```

Here the method `cc.countAll` reads and counts all the characters in a file. The method `getCount` returns the number of occurrences of a given character. You might also use *ICharCounter* to read compressed-file headers too in `unhuff`.

You'll need at least `ALPH_SIZE` counters (typically this is 256, see IHuffConstants). There are more than 256 possible characters if you include the pseudo-EOF character --- so you'll probably need `ALPH_SIZE+1` counters.

Mapping characters to Codes

A class that represents the map of character and encoding bit pattern pairs. You can use one of the Map classes or simply an array of ints.

Since the values stored in the map are constrained to be between 0 and 256, inclusive, you can simply store bitpattern/strings in an array and index them directly. There's nothing at all inferior about this approach compared to using a map. In fact, using an array is in many ways simpler, is as fast as it can be, and is easy to implement (so it's a good choice!).

Debugging Code

Designing debugging functions as part of the original program will make the program development go more quickly since you will be able to verify that pieces of the program, or certain classes, work properly. Building in the debugging scaffolding from the start will make it much easier to test and develop your program. When testing, use small examples of test files maybe even as simple as "go go gophers" that help you verify that your program and classes are functioning as intended.

You might want to write encoding bits out first as strings or printable int values rather than as raw bits of zeros and ones which won't be readable except to other computer programs. A *Compress* class, for example, could support *printAscii* functions and *printBits* to print in human readable or machine readable formats.

We cannot stress enough how important it is to develop your program a few steps at a time. At each step, you should have a functioning program, although it may not do everything the first time it's run. By developing in stages, you may find it easier to isolate bugs and you will be more likely to get a program working faster. In other words, *do not write hundreds of lines of code before compiling and testing*

Using *BitInputStream*

In order to read and write in a bit-at-a-time manner, two classes are provided BitInputStream and BitOutputStream.

Bit read/write subprograms

To see how the *read* routine works, note that the code segment below is functionally equivalent to the Unix command `cat foo` --- it reads `BITS_PER_WORD`bits at a time (which is 8 bits as defined in IHuffConstants) and echoes what is read.

```
    int inbits;
    BitInputStream bits = new BitInputStream(new
FileInputStream("data/poe.txt"));
    while ((inbits = bits.readb(BITS_PER_WORD)) != -1) {
        System.out.println(inbits);    // put writes one character
    }
```
Note that executing the Java statement `System.out.print('7')` results in 16 bits being written because a Java char uses 16 bits (the 16 bits correspond to the character '7'). Executing `System.out.println(7)`. results in 32 bits being written because a Java int uses 32 bits. Executing `obs.write(3,7)` results in 3 bits being written (to the BitOutputStream *obs*) --- all the bits are 1 because the number 7 is represented in base two by 000111.

When using `write` to write a specified number of bits, some bits may not be written immediately because of some buffering that takes place. To ensure that all bits are written, the last bits must be explicitly flushed. The function `flush` **must** be called either explicitly or by calling `close`.

Although *read* can be called to read a single bit at a time (by setting the parameter to 1), the return value from the method is an int. You'll need to be able to access just one bit of this int (`inbits` in code above). In order to access just the right-most bit a bitwise and & can be used:

```
    int inbits;
    BitInputStream bits = new BitInputStream(new
FileInputStream("data/poe.txt"));
    inbits = bits.read(1);
    if ((inbits & 1) == 1)
        // do stuff because the bit read was 1
    else
        // do stuff because the bit read was 0
```
Alternatively, you can mod by 2, e.g., `inbits % 2` and check to see if the remainder is 0 or 1 to determine if the right-most bit is 0 or 1. Using bitwise-and is faster than using mod, but this speed is minor compared to what you'll spend reading the file.

InputStream objects

In Java, it's simple to construct one input stream from another. The Viewer/GUI code that drives the model will send an `InputStream` object to the model that represents a file to be read or written. The client/model code you write will need to wrap this stream (or a File in the case of writing) in an appropriate `BitInputStream` (or output) object.

```
   // create BitOutputStream for a file
   FileInputStream fis = new FileInputStream(file);
   BitOutputStream bitout = new BitOutputStream(fis);

   // or
   BitOutputStream bitout = new BitOutputStream(new FileInputStream(file));
```

Of course exceptions may need to be caught or rethrown.

HUFF FAQ

1. **Deleting a file from a Java program**

   To unlink a file in java, create a File object with the right name, path, then call the `delete` method on the object.

2. **Writing characters when unhuffing**

   To write characters/8-bit chunks to an uncompressed file when unhuffing, use a `BitOutputStream` object and the `write` method.

   ```
   bitout.write(BITS_PER_WORD, value);
   ```

   where `value` is an int/8-bit chunk stored in a leaf of a Huffman tree, for example -- the leaf your code finds during uncompression.

3. **Are files the same?**

   There's a program `Diff.java` that will report of two files are byte-wise identical.

4. **pseudo-eof**

   You must create a node, with count/weight 1, that contains `PSEUDO_EOF` (see IHuffConstants.java) as the info field of the node, and add this node to the collection of nodes used to make the Huffman tree. You **must** create this node explicitly, it's not created automatically from your counts.

   Once created, it will become part of the tree and thus have a zero/one encoding derived from the root-to-leaf path.

   You must write this encoding *after* compressing all the "real" characters (when you re-read the file being compressed). You must write the encoding explicitly.

   Your unhuff program will know what the encoding of `PSEUDO_EOF` is -- see the assignment write-up for how to use the pseudo-eof value when uncompressing.