File Compression

For this assignment you'll implement several classes to build a file compression program! Most of the work will be in the classes you implement that correspond to interfaces you are given as part of this assignment. You're writing code based on the Huffman algorithm discussed in class and in the textbook. 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.

Review the slides from February 14th for a visual walkthrough on the process of compression and decompression.

Starter Files&*Readme template*

Topics: Huffman Coding Tree, I/O Streams, Testing, Recursion, Data Compression, Variable-length Coding.

Due Date: February 26 @ 11:59pm

NOTE
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 testing/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!**

1. Design, test, and develop (in that order) so that you have a working program at each step.
2. Build a program by adding working and tested pieces.
3. Create simple examples and work them out thoroughly. Be prepared to draw out trees and write down bits!

Background & Objective

In this assignment, it is **critical** to fully understand the assignment before you try to code anything. Take time to read through this writeup and the several interfaces/files completely before starting.

To compress you should use the **Huffman Compression algorithm** which has numerous steps. **You should understand each of these steps before starting to code.** To repeat: do not start coding until you read all of this Background and then all of Part I. You should understand all of the implementation details explained in Part I before you write your test cases, and you should write test cases for the methods in Part I before you implement them. Keep in mind that you will implement the first three steps of Huffman coding in Part I, and then move on to writing and uncompressing in Part II.

By the end of the assignment, your program should successfully process a text file and compress it using the Huffman Compression algorithm. It should also be able to process a compressed file and uncompress it back into a text file.

Data Reading & Writing Details

Before we begin coding, there are many details that you will need to think about. It also may be helpful to refer back to this section during project implementation.

> ### *Variable Types*
>
> **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);
> ```

> ### *Using BitInputStream & BitOutputStream*
>
> 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` — 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("poe.txt"));
> > while ((inbits = bits.read(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("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
```

InputStream objects

In Java, it's simple to construct one input stream from another. The 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);
BitInputStream bitout = new BitInputStream(fis);
// or
BitInputStream bitout = new BitInputStream(new FileInputStream(file));
```

Of course exceptions may need to be caught or rethrown.

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 can not 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).

```
// DO NOT USE
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 a 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 as 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 (consider the example below). 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 uncompress with plain (uncompressed) files.**

The last few bytes of a file with extra bits written at the end to get to a multiple of eight, highlighted in red: 10011001 00010011 11011000

The last few bytes of the same file with a pseudo-EOF (1010) added to the end, and then some extra bits due to buffering: 10011001 00010011 11011101 000000000

Because we can scan for the pseudo-EOF, we'll know to stop reading once we catch it. This way we won't read the remaining 0s as compressed text. 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 `IHuffConstants` interface 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. Your program should be able to work with n-bit chunks, not just 8-bit chunks.

Part 1: Huffman

Relevant Files: `ICharCounter`, `CharCounter.java`, `CharCounterTest.java`, `ITreeMaker`, `IHuffConstants`, `Huff.java`, `HuffTest.java`

The first part of the assignment is to complete a compression program using the Huffman coding algorithm. This compression has four essential steps: counting character frequencies, constructing the Huffman tree, mapping and storing the Huffman encodings, and finally compressing documents. You should understand each of these steps before starting to code. Keep in mind that you will implement the first three steps of Huffman coding in Part I, and then move on to writing and uncompressing in Part II.

Step 1: CharCounter

In order to perform Huffman compression, we must 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.**

The CharCounter class should 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 Huffman Encoding program (name this file `CharCounterTest.java`). The class must implement the `ICharCounter` and `IHuffConstants interfaces`. In this step, implement all the methods outlined in the ICharCounter class (implementation details are provided in the interface). You can use the pseudo-code from the textbook as a reference to get started.

EXAMPLE TEST FOR `CountAll()`
You can use a string to test countAll()! Convert the String to a ByteArrayInputStream like this:

```
ICharCounter cc = new CharCounter();
InputStream ins = new ByteArrayInputStream("teststring".getBytes("UTF-8"));
int actualSize = cc.countAll(ins);
assertEquals(10, actualSize, "should be 10 chars read in 'teststring'");
assertEquals(3, cc.getCount('t'), "'t' appears three times");
```

Step 2: Huffman Tree

Now that we have the character counts, we are ready to build our 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.

You will use an ITreeMaker class to create the tree called `Huff.java`. Note the method `makeHuffTree()` in `Huff.java` can make use of the buildTree method available in the textbook; feel free to reference this code directly!

HUFFMAN TIP
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 can use the built-in java PriorityQueue. Understand the java files provided to you including `HuffTree.java`, `IHuffBaseNode.java`, `HuffLeafNode.java`, and `HuffInternalNode.java`. Read the `buildTree()` method here and put it inside your Huff class, make it non static.

Step 3: Mapping Characters (8 Bit Chunks) to Encodings

Now that we have Huffman tree built, we need to store the Huffman encodings so that we can refer to them later during the file compression in Part 2. Create a table or map of characters (8-bit chunks) to encodings. 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.

You will use the `IHuffEncoder` interface to create the table from the tree. Thus, `Huff.java` must implement `IHuffEncoder.java`. Note that the `showCounts()` method should return the map of all characters to their counts created by `CharCounter.java`.

CHARACTERS (8 BIT CHUNKS) TO ENCODINGS TIP

To create a map of coded bit values for each character you'll need to use a **preorder traversal** of the Huffman tree, adding 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:

1. 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.
2. 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 `IHuffBaseNode` 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");
```

Part 2: Writing and Reading Compressed Files

Relevant Files: `ICharCounter`, `CharCounter.java`, `CharCounterTest.java`, `ITreeMaker`, `IHuffConstants`, `Huff.java`, `HuffTest.java`

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. Modify your `Huff.java` and have it implement `IHuffModel` and

`IHuffHeader` interfaces.

When you uncompress, the uncompression program/methods must be able to recreate the Huffman tree which was used to compress the file originally (so the codes you send will match), then use this tree to recreate the original uncompressed file. This means that information on the tree construction will be stored directly in the compressed file (e.g., using a preorder traversal). 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 during uncompression. There's more information below on storing/reading information to re-create the tree. 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 will probably be uncompressed by your code, if both solutions are correct.

WHY RECREATE THE HUFFMAN TREE FROM SCRATCH WHEN UNCOMPRESSING?

There are many ways to construct (correct) Huffman tree encodings, even with idential characters and character counts. In order to ensure that the sender and receiver encodings match, we must include information on **how** the Huffman tree was built so that when we need to uncompress a file, we can correctly decode the file.

Step 1: Compress

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 the **header** at the beginning of the compressed file (see `IHuffHeader.java` for implementation details). The **header** contains the magic number and all information needed to reconstruct a tree. Both of these are explained in detail below.
2. Write the bits needed to encode each character of the input file.
3. Write the PSEUDO-EOF once the end of the file is reached.

BIT WRITING EXAMPLE

If the coding for the character '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'.

TESTING TIP

You can use a ByteArrayOutputStream to test `IHuffHeader` methods like this:

```
//create the ByteArrayOutputStream
ByteArrayOutputStream out = new ByteArrayOutputStream();
//construct a  BitOutputStream from out
//check the size of the header that was written where h if your Huff object
assertEquals(118, h.writeHeader(new BitOutputStream(out)));
out.close(); //do not forget to close the stream
```

Magic Number

Your code uses the magic number to determine if a compressed file was created by your program when uncompressing. 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 magic number is defined in the `IHuffConstants` interface. 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 methods 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) when uncompressing

```
// check the magic number to validate the compressed file
int magic = in.read(BITS_PER_INT);
if (magic != MAGIC_NUMBER){
  throw new IOException("magic number not right");
}
```

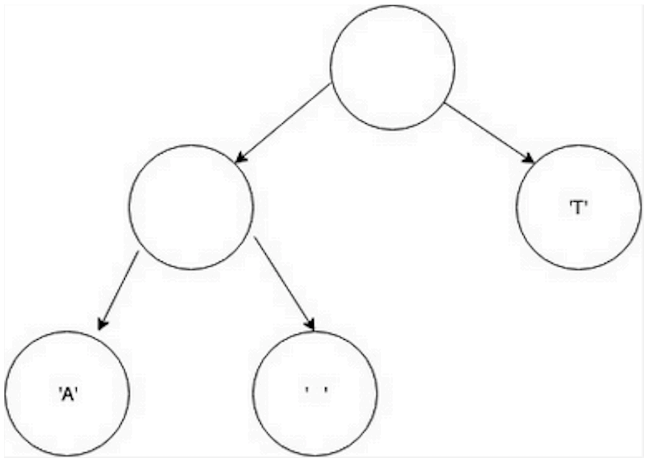
Storing the Huffman Tree

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. The technique that will be **required for this assignment** 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).

EXAMPLE FOR HUFF TREE STORAGE

As mentioned above, we follow a pre order traversal of the tree. Whenever we reach a non-leaf node, we write 0. Whenever we reach a leaf node, we write 1 followed by the 9 bit encoding stored inside the leaf. **The sequence of 0's and 1's below represents the tree below (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 001000001 1 000100000 1 001010100



To explain the following encoding in detail, the first `0` indicates a non-leaf (since we're at the root), the second `0` is the left child of the root, a non-leaf. The next 1 is a leaf and it is followed by 9 bits that represent 65 (`001000001` is 65 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 84 (`001010100` is 84 in binary).

In order to write the sequence, you should perform a standard pre-order traversal.

Step 2: Uncompress

Now, we are ready to uncompress a compressed file. Similarly to compress, there are three steps to uncompress a compressed file.

1. Check to see if the first N bits at the beginning of the compressed file match the magic number.
2. Use the encoded huffman tree (bits written after the magic number) to fully reconstruct the encoded Huffman tree. Both reading/writing are done in classes that implement `IHuffHeader`.
3. Decode the encoded file. In order to to do this, you can read bit by bit. Then using your constructed Huff tree, traverse to the left/right subtrees if the bit is a zero or not. Once you reach a leaf node, write the respective node element and start the process again!

Step 3: 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 the write method should return the expected size of the compressed file. If the user forces compression using the 'force' argument, then compression occurs even if the compressed file is bigger.

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 = 208 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. `Huff.java` will implement the methods from the `IHuffHeader` interface.

Part 3: Submit to Gradescope

Submit all of your classes (`CharCounter.java`, `CharCounterTest.java`, `Huff.java`, and `HuffTest.java`), the `readme.txt` file, and all relevant testing files to Gradescope. Submissions without a test class will result in a grade of zero on the assignment.

> **SUBMITTING TESTING FILES**
> When submitting files for your test cases, upload them at the top level of the submission outside of any directories. If you use any directories in your local submission, make sure to change your filepaths when you go to submit (e.g. `simple_file.txt` instead of `/Users/sharry/Documents/hw3/simple_file.txt`). Ask in OH if you need help with this.

Make sure that you follow all of the CIT 5940 testing guidelines so that the autograder runs properly on your submission. Submissions without a test class will result in a grade of zero on the assignment.

- Achieve high code coverage
- Write unit tests for everything
  - Prefer many unit tests with just one or two assertions
  - Some longer tests with many assertions may be appropriate, but these aren't as useful for debugging
- Submit all files required for testing, including any text or data files you may have generated.

Don't forget to polish, comment, and take pride in your code! You worked hard to finish this assignment, so make sure that the presentation of your code reflects that.

> **STYLE CHECKING HERE IS A LINK TO INSTRUCTIONS FOR INSTALLING AN AUTOMATED STYLE CHECKER FOR ECLIPSE AND FOR INTELLIJ. THIS STYLE CHECKER USES EXACTLY THE SAME GUIDELINES THAT YOU'LL BE GRADED ON; THAT IS, IF YOU PASS THESE STYLE CHECKS IN YOUR IDE, YOU'LL GET FULL CREDIT ON THE STYLE PORTION OF YOUR GRADE!**

Grading

Your grade will consist of points you earn for submitting the homework on time and for the final product you submit.

The assignment is worth 205 points.

Description	Points
Problem Coverage (correctness)	135

Description	Points
Testing (correctness + 80% code coverage)	50
Documentation and Style	15
Readme	5

Testing is worth a lot of points! Here are a few additional guidelines:

- All test cases must contain an assertion. There will be a flat 10 point penalty if any test is missing an assertion statement.
- All of your tests must pass. Remove failing tests. There will be a flat 10 point penalty if any test fails.
- You will earn full credit for meeting the previous two criteria and achieving at least 80% code coverage.

This assignment was originally adapted for Penn by Eric Fouh. This version of the assignment was updated by Daniel Paik & Harry Smith, 2023.