Project: Huffman Coding

(Last Submission Date: Thursday October, 6th at 11:59pm)

Goal: Practice implementation of an algorithm using techniques learned in CS3xx courses.

Activity: Given a file encoded using Huffman encoding and a Huffman tree, decode the files.

Teamwork: Pairs within the same section of CS483 are permitted however… the assignment is slightly different for pairs [see below]. *One set of code should be produced, but both students must create a submission and upload that submission to Blackboard for credit.*

Details

Motivation

We have covered Huffman coding in class, but it is time to take that knowledge and "use it" in a real-world setting. It is very common in both academia and industry to encounter the *specification* for some data structure (in the traditional sense of "structured data", such as a file, message, or data packet) that you want to use. However, it is infrequent that this is the structure you want the data to remain in.

For example, an internet data packet contains a lot of information, and you might want to extract the destination of the packet so that you can forward it on. Similarly, you could encounter an encoded file (perhaps using Huffman encoding) which you'd like to decode.

We're going to practice this skill while reinforcing your knowledge of Huffman coding. To do this you're going to need to remember some skills from your previous CS3xx classes, such as file I/O, byte and bit manipulation, and other such tools. I'm providing some guidance in this document, and a few restrictions on what you can do, but otherwise this is a "make it work" situation.

Review of File I/O in Java

File I/O is an important aspect of computer science, we rarely run a program from start to finish without needed to save *something* somewhere, and the most common form of "saving" information is to write it to a file which has a life beyond that of the program. For this assignment, I am allowing the following tools for Java file I/O from the package **`java.io`**:

- **`FileInputStream`**
  - Anything named **`----InputStream`** in Java is for **reading *bytes* from** a file.
  - Documentation: https://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html
- **`FileOutputStream`**
  - Anything named **`----OutputStream`** in Java is for **writing *bytes* to** a file.
  - Documentation: https://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html
- **`FileReader`**
  - Anything named **`----Reader`** in Java is for **reading *characters* form** a file.
  - Documentation: https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html
- **`FileWriter`**
  - Anything named **`----Writer`** in Java is for **writing *characters* to** a file.
  - Documentation: https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html

Java (like most languages) does not allow you to read/write single *bits* to a file (only *bytes*). Therefore I have also included a few files to help with reading/writing bits:

- **`BitInputStream`**
  - This is a wrapper around an **`InputStream`** (which works with *bytes*) to allow both *byte*-reading and *bit*-reading. Sample usage:

```
//initialization
BitInputStream bis = new BitInputStream(new FileInputStream(fileName));

//reading one BYTE
byte oneByte = (byte) bis.readNext(); //returns an int, downcast to byte
//there is also a hasNext() method for loop control

//reading multiple BYTES into an array
byte[] multipleBytes = new byte[numberOfBytesYouWant];
int count = bis.read(multipleBytes);
//count now contains the number of bytes read (or -1 for errors)

//reading BITS, once in bit-mode you shouldn't do any more byte reading!
bis.startBitMode();

//reading one BIT
int bit = bis.readNext();
//bit will be the int 1 or the int 0 (if a bit could be read) or -1 if not
//hasNext() will still work for loop control

bis.close(); //important!
```

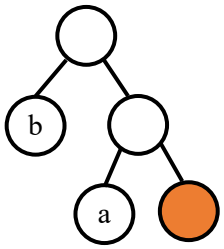
- **`BitOutputStream`**
  - This is a wrapper around an **`OutputStream`** (which works with *bytes*) to allow both *byte*-writing and *bit*-writing. Sample usage:

```
//initialization
BitOutputStream bos = new BitOutputStream(new FileOutputStream(fileName));

//writing one BYTE
bos.writeByte(0); //accepts ints between 0 and 255

//writing BITS, once in bit-mode you shouldn't do any more byte writing!
bos.startBitMode();

//writing one BIT
bos.writeBit(0); //accepts ints between 0 and 1

//writing multiple BITS
String s = "100"; //put your bits in a string
bos.writeBits(s); //and it will write them in order (e.g 1, then 0, then 0)

bis.close(); //important!
```

Encoded File Format

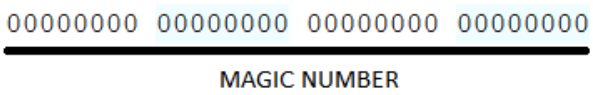
What does an encoded file look like? There is a test-files directory with this project with six compressed files (`in1.txt.comp` through `in6.txt.comp`) and one decompressed file (`in1.txt`) to explain the format.

File `in1.txt` contains the text "ab" and so we need to represent the following characters in the Huffman tree: 'a', 'b', and the end-of-file. We need an end-of-file representation because we might not end on even multiples of 8, as we will see later. The Huffman tree that was built for this file looks like the picture below, where the orange-filled circle is the end-of-file character:



This means the encoding for the text ("ab") will be 10 ("a"), then 0 ("b"), then 11 (e-o-f). However, we will need more than just 10011 in our file or we won't have enough information to decode.

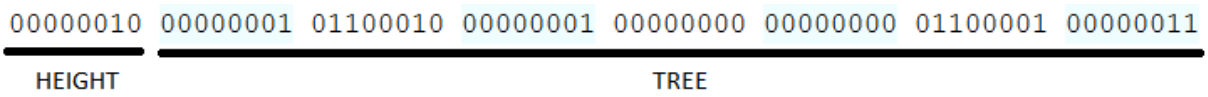
First, we need some indicator that this is, in fact, a valid binary file with our Huffman encoding. Most binary files contain what is called a "magic number" which kicks off the binary and indicates that the binary file is of a certain type. For example, Java class file magic number is "1100 1010 1111 1110 1011 1010 1011 1110" which in hex is "CAFE BABE". Our magic number is going to be four bytes of 0:



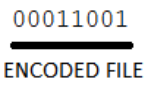
Second, we need to represent the tree. Normally, one wouldn't include this *in* the file, it would be stored in the *program* doing the compression/decompression, or sent as a *separate* file alongside multiple files compressed with the same tree. For this homework, however, we're including the tree in the file. The tree is stored as *perfect* binary tree of some height in standard array format (*level* order). This is not an efficient tree encoding, but that's not the point of this assignment.

In the encoded file, the height of the perfect tree is given immediately after the magic number. Followed immediately by the tree array. Non-existent nodes are stored as the null *character* (0 on the ASCII chart), note that this is not the same thing as "null" that you are used to in Java, this the *character* null. Internal nodes are stored as the ASCII character for the start of a header (1 on the ASCII chart). The end-of-file is stored as the end-of-text character (3 on the ASCII chart). All other characters are stored as *single* bytes, so only single-byte characters are allowed (i.e. ASCII characters).

So our tree above will look like this:

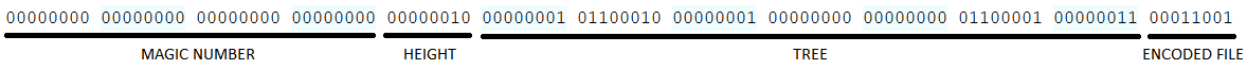


Finally, after the tree, we have the actual encoding for “ab” (10011):



Why is it backward? Because the BitInputStream and BitOutputStream treat each byte like a “stack” of bits with the lowest *digit* being the *bottom* of the stack. So if we wanted to store the bits “1000000011000000” we would see “00000001 00000011” in a hex editor because we did the following: push 1 onto the bottom of the stack `(-------1)`, push seven 0s `(00000001)`, write the byte, push 1 onto the bottom of the stack `(-------1)`, push another 1 on the stack `(------11)`, push six 0s `(00000011)`, write the byte.

So the entire file for in1.txt.comp looks like this:



Common Question: Will ETX (end-of-text) always be encoded as 11 (it's ASCII value)?

No. That just happens to be the case in the example above. It is encoded by whatever position it is in the tree. For example, if the tree looks like this:

```
00000001 (header), 01100001 (a), 00000001 (header), 00000000 (null),
00000000 (null), 00000011 (ETX), 01100010 (b)
```

Then ETX is encoded as "10".

And if the tree looks like this:

```
00000001 (header), 00000001 (header), 00000011 (ETX), 01100001 (a),
01100010 (b), 00000000 (null), 00000000 (null)
```

Then ETX is encoded as "1".

Your Turn

So what are you doing? If you are working on your own, you are just decoding the files. If you are working in a pair, then you are also writing a program to encode text files into the given format above. These are your restrictions:

You may only have the following imports:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;
```

Your code must compile without warnings or errors with the following command in your user directory:

```
javac *.java
```

Your program must run with the following command (for encoding and decoding):

```
Usage: java Huffman <inputFile>
```

If the files starts with the magic number, perform decoding and write the file to the same file name but add "`.txt`" to the end. For example, `in.txt.comp` becomes `in.txt.comp.txt`. If you are working in pairs and are doing encoding, then if the file does not start with the magic number, output to the same file name but add "`.comp`" to the end. For example, `in.txt` becomes `in.txt.comp`. Do not try to "optimize" these file names, your code will be graded for correctness by a *computer* and if it can't find the file you were supposed to output, then it will give you a 0. The TAs are NOT going to fix this for you.

> ***Common Question: Where should files be read from and/or written to?***
>
> We can provide the following for arg[0]:
>
> - the name of a file (this means it expects to read/write it in the current directory)
> - the relative path to a file (that path says where you read from, and when you write, you write to that same location)
> - the absolute path to a file (that path says where you read from, and when you write, you write to the same location)
>
> All of these things are achievable by simply appending the correct post-fix (extension) ".txt" to the string given to you in main and writing to that file (as directed above).

Note: the restrictions and commands above mean that you cannot use packages, add additional imports, hardcode which files to read/write, etc. Doing any of those things will most likely result in a 0 on the entire assignment.

Additional Restriction: This project will be graded by JUnit tests. JUnit tests are just Java code that runs your Java code. Therefore, if you call `System.exit()` in your code, you will kill the test cases and you will not be able to get a score. The GTAs will not fix this for you, so do not use `System.exit()` to end your program (you can simply return from `main()` to end a program).

Grading Rubric

No credit will be given for non-submitted assignments, assignments submitted after the last submission date, non-compiling assignments, or non-independent work ("pairs" are still required to work independently of other students/pairs).

Otherwise your score will be as follows:

For individual students:

- Full credit:
  - 10pts – you can decode files
- Partial credit:
  - 1pt – Correct file generated (correctly named file exists)
  - Up to 3pts – Manual inspection indicating a valid attempt at doing the project

For pairs of students:

- Full credit:
  - 5pts – you can decode files (0pts if you can't)
  - 5pts – you can encode files (0pts if you can't)
- Partial credit:
  - 1pt – Correct files generated (correctly named file exists)
  - Up to 3pts – Manual inspection indicating a valid attempt at doing the project

I don't care who is "responsible" for what part of the project. If you're working in a pair, both parts need to work. You may NOT start working as a pair and then split to try to get credit for only the first half. All cases of similar code being submitted by two students *not claiming to be on a team* will be considered an honor code violation.

5% will be subtracted if you don't submit in the correct zip➔folder➔files format listed in the submission instructions (because it will slow down grading for the entire class).

Extra credit for early submissions: 1% extra credit rewarded for every 24 hours your submission made before the last submission date (up to 5% extra credit). Your latest submission before the due time will be used for grading and extra credit checking. You CANNOT choose which one counts.

Submission

- Make a backup copy of your user folder (in case you mess this up), then remove all test files, jar files, class files, output folders, etc. You're just submitting your java files and the readme.txt.
- ***EVERYONE:***
  - Complete the readme.txt file in the user folder (instructions are in the file).
- ***INDIVIDUALS:***
  - Zip your user folder (***not just the files***) and name the zip `username.zip`.
  - The submitted file should look something like this for INDIVIDUALS:

```
krusselc.zip --> krusselc --> BitInputStream.java
                              BitOutputStream.java
                              Huffman.java
                              readme.txt
                              ...
```

  That's a zip file containing a folder containing your code.

- ***TEAMS (next page):***

- ***TEAMS:***
  - Zip your user folder (***not just the files***) and name the zip `team-username1.zip`.
  - The submitted file should look something like this for EACH TEAM MEMBER:

```
team-krusselc.zip --> krusselc --> BitInputStream.java
                                   BitOutputStream.java
                                   Huffman.java
                                   readme.txt
                                   ...
```

    That's a zip file which starts with the word "team-" containing a folder for YOU containing your team's code. BOTH STUDENTS need to create SEPARATE zip submissions in this format. So the *second* person on the team creates a *separate* file that would look something like this:

```
team-otherusername.zip --> otherusername --> BitInputStream.java
                                             BitOutputStream.java
                                             Huffman.java
                                             readme.txt
                                             ...
```

    The code and readme.txt files are the same for both people, only the folder and zip are different.
- ***EVERYONE:***
  - Go to the Homework 5 submission link on Blackboard and submit your assignment. You can resubmit as many times as you'd like, only the last submission will be graded.
  - CHECK WHAT YOU UPLOADED. Wrong submissions will receive a 0.

Source of any ascii art you may find: https://www.asciiart.eu