CSC 225 - SUMMER 2019
ALGORITHMS AND DATA STRUCTURES I
PROGRAMMING ASSIGNMENT 2
UNIVERSITY OF VICTORIA

Due: Sunday, July 7th, 2019 before 11:55pm. **Late assignments will not be accepted.**

1 Assignment Overview

This assignment covers a basic lossless data compression technique, which will require the application of various data structures, especially trees. To receive full marks, you will need to write a Huffman Coding algorithm (using a tree representation to ensure efficiency).

In a lossless data compression system, there are two separate but equally important components. The **encoder** takes an arbitrary stream of data (such as a file) and produces an encoded representation (which, ideally, requires far less memory). The **decoder** reads an encoded representation and recovers the original data stream. Accordingly, this assignment has two parts: your submission will include both an encoder and a decoder, using a simplified compressed file format (for which some Java classes have been provided to handle reading and writing data).

As with all assignments in this course, your priority should be to produce **working code**. Some partial marks are available for code that does not perfectly implement the specification (or achieve certain running time thresholds), but, in general, your code will receive no marks unless it can correctly compress and decompress data (with no errors or output differences whatsoever).

This assignment will be evaluated by a combination of automated testing and human inspection. Unlike assignment 1, there will not be in-person demos. For the decoder component, you will be expected to achieve an efficient running time to receive full marks; since there are no in-person demos, you are advised to write explanatory comments in your code to ensure that the running time is not misinterpreted by the evaluator.

2 Compression Scheme

This assignment uses a lossless compression scheme for arbitrary data (text, images, binary data, etc.) using a prefix-free code. As we have seen in the lectures, prefix-free codes have several advantages for compression, specifically the ability to easily encode symbols into bitstrings of diverse lengths. Although the best prefix-free code for a particular set of symbols can be produced by Huffman Coding, the format used by this assignment can theoretically accommodate any kind of prefix-free code (even if it was not generated by the Huffman Coding algorithm).

The encoding program, which will be implemented in `HuffEncoder.java`, takes two command line arguments: an input filename and a compressed output filename. Since the file format is unique to this course, we will adopt a convention of naming the compressed files with the '`.225`' extension

(although this is just a naming convention, so there is nothing stopping you from using other extensions). For example, to encode the contents of '`some_file.txt`' and store the compressed output to '`compressed_file.225`', the following command could be used.

```
java HuffEncoder some_file.txt compressed_file.225
```

The decoding program, which will be implemented in `HuffDecoder.java`, also takes two arguments: an input compressed filename and the name of a file to store the decompressed result. For example, to decode the contents of '`compressed_file.225`' and store the result in '`decompressed.txt`', the following command could be used.

```
java HuffDecoder compressed_file.225 decompressed.txt
```

2.1 File Format

Each compressed file has two sections, in the order below.

- A **symbol table** containing a mapping of symbols (which are sequences of zero or more bytes of data) to their bit encodings (which are one or more bits in length).
- A **bit stream** consisting of a long string of bits constructed from the bit strings in the symbol table.

The compressed files store these two sections in a binary representation that you are not expected to understand or use directly. Instead, two classes `HuffFileReader` and `HuffFileWriter` have been provided. These classes define a higher level interface for reading and writing the compressed data files. You are not permitted to modify these classes in any way (to ensure compatibility of your code with the automated testing infrastructure), but you are allowed to create subclasses if you want to change the functionality (just be sure to submit all of the code).

Both the reader and writer classes use the auxiliary class `HuffFileSymbol` to represent entries in the symbol table. Each `HuffFileSymbol` object contains a symbol (as a byte array) and a bit sequence (as an array of integers). For your convenience, the provided interfaces store bit sequences as integer arrays (which is not very space efficient).

The reader and writer objects are designed solely to help you read through the file in order. They do not store the different parts in memory (since the files could become very large). As a result, you are responsible for reading/writing the symbol table entries before handling the bit stream.

Specific information about the methods in the `HuffFileReader` and `HuffFileWriter` classes can be found in the comments in each source file. A sketch of the normal use of each class is given below.

A normal use of `HuffFileReader` might follow the pattern below.

- To open a particular file for reading, construct the object with '`new HuffFileReader(filename)`'.
- To read each entry of the symbol table (which may be in any order, depending on how the file was encoded), call `readSymbol()` until that method returns `null`. When `readSymbol()` returns `null`, the symbol table has been completely read.
- Read bits from the bit stream, one at a time, using `readStreamBit()`, decoding the bits as needed. The `readStreamBit()` function cannot be used until after the entire symbol table has been read (and `readSymbol()` has returned `null`). Each call to `readStreamBit()` will return an `int` value: If a bit was successfully read, the value will be 0 or 1. If end-of-file was encountered, the return value will be -1.

- Call `close()` when finished to close the file.

A diagnostic program, `PrintHuffFile.java`, has been provided to print a text representation of the encoded files to the console. You may want to look at the source code for `PrintHuffFile.java` for examples of the use of the `HuffFileReader` class.

A normal use of `HuffFileWriter` might follow the pattern below.

- To open a particular file for writing, construct the object with '`new HuffFileWriter(filename)`'. If the file already exists, its contents will be erased before writing.
- To write each entry of the symbol table, call `writeSymbol()`.
- Once each symbol entry has been written, call `finalizeSymbols()` to end the symbol table. After calling `finalizeSymbols()`, no further symbol table entries may be added.
- To write the bits of the encoded bit stream, call `writeStreamBit()`.
- Call `close()` to finalize and close the encoded file.

3 Specification

The following Java source files have been provided.

- `HuffDecoder.java`
- `HuffEncoder.java`
- `HuffFileException.java`
- `HuffFileReader.java`
- `HuffFileSymbol.java`
- `HuffFileWriter.java`
- `PrintHuffFile.java` (for your own testing only)

Your task is to complete the implementations of the `HuffEncoder` and `HuffDecoder` classes. You are not permitted to modify the other code (to ensure compatibility between your version and the version used during marking).

Your `HuffDecoder` implementation must be able to correctly decode all valid encoded files (including files not produced by your encoder). Several sample encoded files have been posted to conneX to assist you in testing your implementation. Note that a 'correct decoding' must be identical to the original input file that was compressed; even the smallest difference (such as one missing or incorrect character) will result in the decoding being considered incorrect.

Your `HuffEncoder` implementation must be able to encode **any data** (not just text). You are expected to build a prefix coding scheme in which all symbols are **one byte** in length. Do not develop a scheme to encode multi-byte symbols. You will not receive any marks unless your encoding is a prefix-free code (and is also encoded correctly using the format described in Section 2.1).

Recall that it is easy to produce a prefix-free code for any set of symbols. For example, for the symbols 'a', 'b', 'c', we could use the mapping below.

Symbol	Encoding
a	10
b	110
c	1110

For full marks, your encoding should be as space-efficient as possible in all cases (while still using single-character symbols). This will require you to use Huffman Coding. You will receive partial marks for sub-optimal encodings as long as they are prefix-free codes and the data is in the correct format.

Important Distinction: Your encoder **must** use single-character symbols. However, the file format supports multi-byte symbols, and your decoder must support symbols of any length. Several samples using unusual symbol sizes (or a mix of symbol sizes) have been posted for your review. You may want to use `PrintHuffFile` to inspect their contents.

The provided versions of `HuffEncoder` and `HuffDecoder` have a small amount of starter code that you may want to examine before starting your solution. You do not have to use the starter code provided (as long as your program has the same command line interface).

3.1 Using Outside Code

You are encouraged to use the features of the Java Standard Library (including any of the data structures it provides) in your code. If you use a standard library data structure, make sure you are aware of the running times of the operations you use, since that information will be important for determining the running time of your program.

There should be no need to use large volumes of code from other sources (such as outside libraries or the internet) in this assignment. If you believe that your implementation requires an outside library, talk to your instructor.

If you find a small snippet of code on the internet that you want to use (for example, in a StackOverflow thread), put a comment in your code indicating the source (including a complete URL). Remember that using code from an outside source without citation is plagiarism.

You are encouraged to discuss the assignment with your peers, and even to share implementation advice or algorithm ideas. However, you are not permitted to use any code from another CSC 225 student under any circumstances, nor are you permitted to share your code in any way with any other student (or the internet). Sharing your code with others before marking is completed, or using another student's code for assistance in any way (even if you do not directly copy it) is plagiarism.

3.2 Why is the compression ratio so bad?

You may notice that, for some input files, your compressed data is actually larger than the original data (which would obviously be considered terrible performance for a compression scheme). Although a Huffman Code on a given set of symbols (and frequencies) will produce an optimal bitstream, there are several other variables that affect real compression performance. The most significant one is the symbol size (or, more generally, the logic used to select which sequences become symbols). Since this assignment forces the use of single-byte symbols, your code has no control over this aspect.

The bitstream produced by a Huffman Code may be optimal (among prefix-free codes), but the symbols themselves (and their encodings) must also be stored if the bitstream is to be decoded. The format used by this assignment has a very simplistic way of storing the symbol table (to make it easier for you to write your code), and therefore has very high overhead compared to most

compression schemes used in practice. The majority of lossless compression schemes for general data use Huffman Coding as part of the compression process, in addition to other techniques to improve efficiency. Most schemes used in practice also have more advanced algorithms to find the best set of symbols for a given input file, and may change the symbol table over the course of the encoding to take advantage of changes in symbol frequency.

The Department of Computer Science occasionally offers a fourth year course in compression techniques, cross-listed with the graduate level compression course (CSC 563), which covers various compression schemes (including lossy techniques, like JPEG) in more depth.

4 Evaluation

Submit all `.java` files needed to compile your assignment electronically via conneX. Your code must compile and run correctly in the Linux environment in ECS 242. If your code does not compile as submitted, you will receive a mark of zero.

This assignment is worth 9% of your final grade and will be marked out of 18, with 9 marks each for the encoding and decoding programs. Evaluation will be based primarily on correctness, with some marks for efficiency of the decoder component. Although there are no formal running time requirements for the encoding component, the encoder is still expected to be fast enough in practice to allow testing.

The marks are distributed among the components of the assignment as follows.

Marks	Component
5	The `HuffDecoder` implementation functions correctly on a variety of input files. Your implementation will only be tested on correctly encoded input files (so you do not need to include any error handling logic for invalid input files). Note that the decoder is expected to work with input files that use multi-byte symbols (not just one byte symbols), unlike the encoder.
4	The `HuffDecoder` implementation achieves a worst case running time of $O(s + n)$ on an input file with s entries in the symbol table and n bits in the bit stream. To receive any partial marks for this item, the running time must be $o(n \log_2 s)$.
3	The `HuffEncoder` implementation generates valid compressed output files (using a prefix-free code). It is not necessary to use a Huffman code to receive these marks. No marks will be given for the encoder unless the encoding is a prefix-free code.
6	The `HuffEncoder` implementation generates valid compressed output files in which the encoding is a Huffman code.

You are permitted to delete and resubmit your assignment as many times as you want before the due date, but no submissions or resubmissions will be accepted after the due date has passed. You will receive a mark of zero if you have not officially submitted your assignment (and received a confirmation email) before the due date.

Your implementation must be based on the provided classes, and may not modify the core interfaces or libraries (including the `HuffFileReader` and `HuffFileWriter` classes). You may use additional files if needed by your solution. Ensure that each submitted file contains a comment with your name and student number.

Ensure that all code files needed to compile and run your code on the Linux environment in ECS 242 are submitted. Only the files that you submit through conneX will be marked. The best way

to make sure your submission is correct is to download it from conneX after submitting and test it. You are not permitted to revise your submission after the due date, and late submissions will not be accepted, so you should ensure that you have submitted the correct version of your code before the due date. conneX will allow you to change your submission before the due date if you notice a mistake. After submitting your assignment, conneX will automatically send you a confirmation email. **If you do not receive such an email, you did not submit the assignment.** If you have problems with the submission process, send an email to the instructor **before** the due date.