Project 3: Huffman Encoding

CS 245: Summer 2023

Due: Sunday July 30 by 11:59pm PST on Github

Overview

In the early computer days, memory was expensive and storing large files was discouraged. As a result, algorithms that could drastically reduce file sizes and then expand them again without losing any information were really valuable. Even though memory is cheap these days, smaller devices such as your phone still have less space to store large files. This makes compression algorithms still useful. In this assignment, we will focus on a text compression algorithm known as **Huffman Encoding**.

Your computer translates all data, whether it be a word document, photograph or video into a series of 1s and 0s that are abstracted away from the end user. For example, any integer is represented as a sequence of 32 1s and 0s. When storing strings, your machine translates each character into a 8-bit sequence (known as a byte). For example, the string “abc” is stored underneath the hood as “011000010110001001100011”.

Utilizing 8-bits for each character can be quite memory-wasteful. Suppose the string is “aaabbc”. The character ‘a’ occurs the most frequently, and ‘b’ the second most frequently. Each instance of ‘a’ and ‘b’ still needs 8 bits, while we utilize 8 more bits for ‘c’. What we’d like to do is develop a methodology where frequently occurring characters have a shorter, more compact representation. On the other hand, having longer representations for infrequent symbols is tolerable since they don’t appear that often. We call each bit sequence representing a character a **codeword.**

The problem of coming with variable length encodings is that it can be hard to determine where one codeword ends and the next one starts. Suppose we let ‘a’ be represented as ‘0’, ‘b’ as ‘1’, and ‘c’ as ‘10’. If we ever came across a sequence of bits of “10”, we would not know whether this encoded “ab” or the single letter “c”.

David Huffman developed an algorithm (aka Huffman Encoding) that would allow us to generate codewords where one codeword would never be a prefix of another codeword. For example, the encoding of ‘a’ will never be a prefix for the encoding ‘b’.

Setting Up

Navigate to your local copy of the Assignments repo. Download the starter code for the assignment by running the following command: **git pull origin main.** Make a copy of the project3 folder and move it to your own private repo that you should have setup by now. **For this assignment, you do not need any of the jar files from Canvas.**

Huffman Encoding Algorithm

In this section, we describe how the encoding/compression phase of the algorithm works.

Building the Frequency Table

The Huffman compression algorithm begins by reading in the input file and determining the frequency of each character.

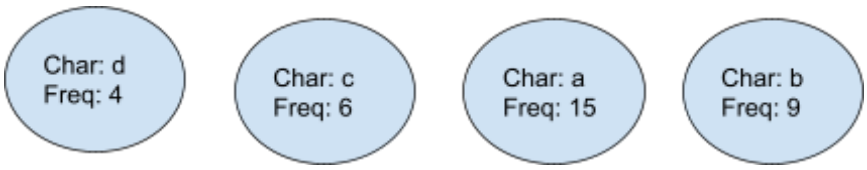
Building the Huffman Tree

Suppose in the input file, we have the following character frequencies:

Character	Frequency
a	15
b	9
c	6
d	4

We create singleton tree nodes for each of these characters and put them in a priority queue.

Original Priority Queue:

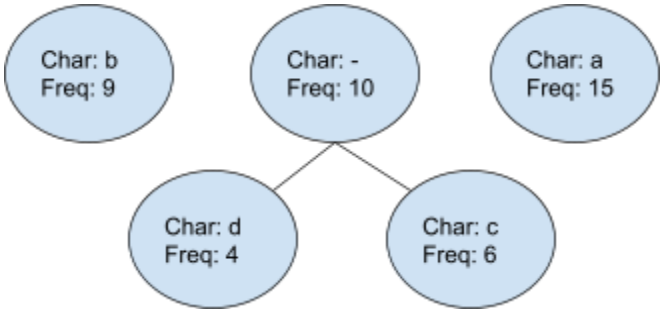


We then perform the following steps:

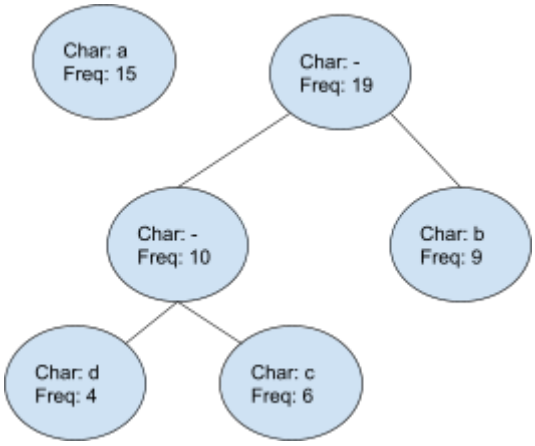
1. Remove the T1 and T2, the two nodes with the lowest frequency.
2. Create a new node T3 where T1 and T2 are the left and left child. Which one is left and which one is right does not matter.

3. Make the frequency of T3 the sum of the frequency of T1 and T2. Make the character of the T3 any value (it doesn't matter).
4. Put T3 back into the priority queue.
5. Repeat steps 1-4 until there is only one node left in the Priority Queue.

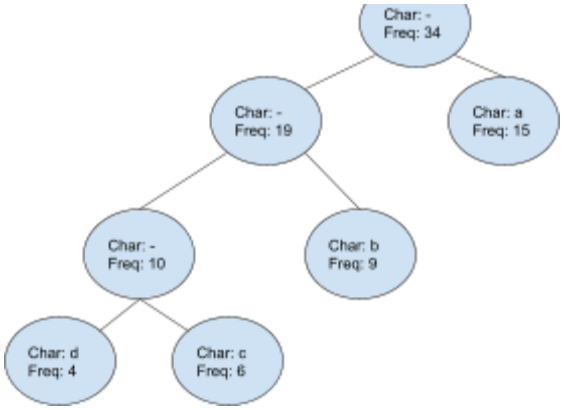
In this example, the nodes for 'd' and the 'c' were the lowest in frequency. So, we remove them from the Priority Queue and created a new node T3 whose frequency is 6+4 = 10. In this example, we make the char associated with T3 '-'.



When we perform steps 1-4 for a second time, the Priority Queue now looks like



The final state after all the nodes have been merged looks like:



Extracting Huffman Codes

After we construct the tree, we now need to determine the Huffman codes of each character. We start at the root with an empty encoding string of “”. Whenever we traverse the left side of a tree, we append 0 to the encoding and 1 whenever we go down the right side. Whenever we hit a leaf node, we store the path we traversed to get to that leaf from the root.

Following this procedure and using the final tree shown above, we derive the following Huffman codes:

Character	Code
a	1
b	01
c	001
d	000

Compressing the File

For each character in the input file, we look up its codeword and write the codeword as a sequence of 1 and 0 bits to an output file.

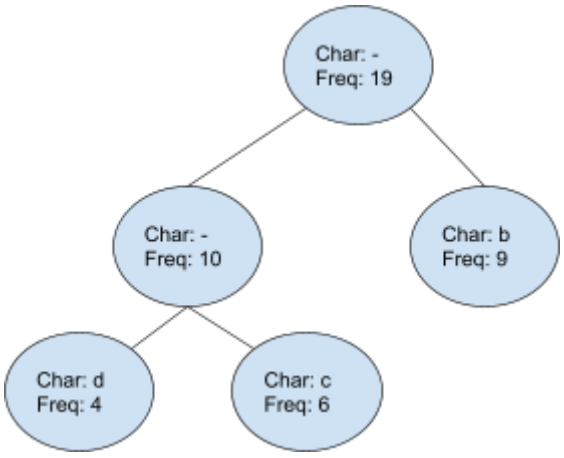
Serializing the Tree

We haven’t talked about decoding yet (see below for more details), but in general you won’t have immediate access to the Huffman tree we created above. What we can do is store a representation of the tree in the output file as a header. In other words, the output file we create

during the encoding/compression phase has a structure that looks something like that shown below:

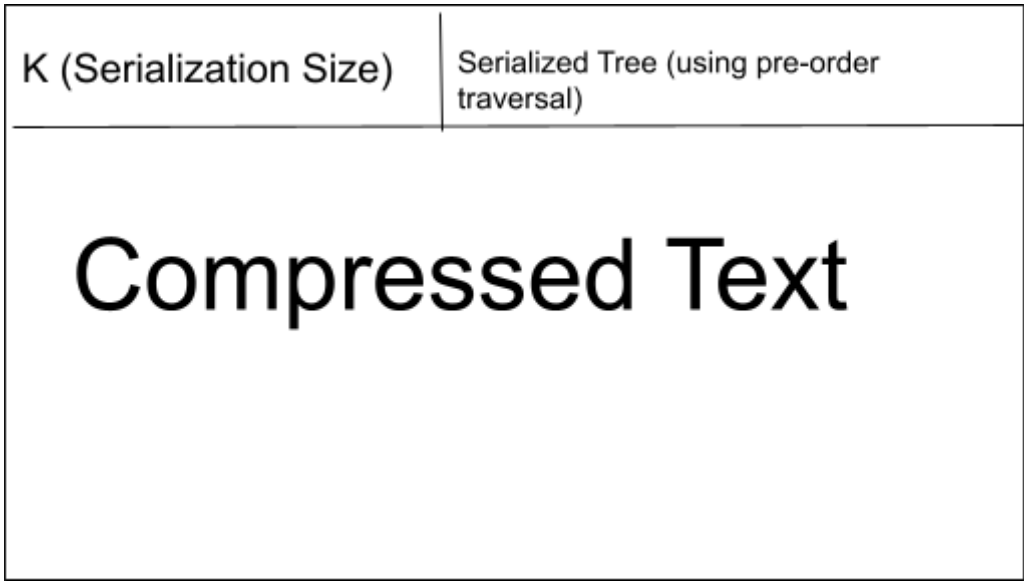
How do we generate the serialization of the tree? We perform a **pre-order** traversal on the Huffman tree we created above. For each internal (non-leaf) node, we write a **0** bit to the output file. For each leaf node, we write a **1 bit followed by the 8-bit representation** of the character associated with the leaf node.

Suppose our Huffman tree looked like this:



The serialization of this tree would then be: 00 101100100 101100011 101100010, where each sequence in red is the 8-bit representation of ‘d’, ‘c’ and ‘b’ respectively (spaces added for clarity). The two zeros in purple denote the internal node (root and root.left in this example), and the 1s highlighted in green indicate a leaf.

Also as part of the meta data, we also store an integer **K (as a sequence of 1 and 0 bits**) that indicates how many bits are required to represent the tree's serialization. The size required can be calculated as **K = (8 + 1) * num_leaves + num_internal_nodes.** In the example above, we needed 29 bits to store the serialization of our Huffman tree. The contents of our compressed file can now be more accurately represented as below



Implementing Compression

Huffman Encoding Implementation

In **Encoder/HuffmanEncoder.java,** implement the Huffman encoding algorithm that generates and compresses a text file. You will notice that we have not provided any skeleton code, so how you implement the Huffman encoder is totally up to you. We strongly recommend being liberal with the use of helper functions for this part. Put all your orchestration logic (including tree serialization, see below) in the **encodeFile** method that is defined in **HuffmanEncoder.java.** The encodeFile method takes two string arguments: the first argument represents the path to the file to be compressed. The second argument is the path of the compressed outputfile. **Our testing script calls this method, so do not modify this method signature in any way.**

Tree Serialization Implementation

In **Utils/TreeSerializer.java**, we defined a function called **writeHeader** that takes in two parameters: A BufferedBitWriter instance and the root of the Huffman tree. This function writes out the meta- data (see above) at the start of the compressed file. This meta-data is used during the decompression phase. Your job is to fill in the helper methods that **writeHeader** depends on. The functions you need to implement are:

- **int getNumInternalNodes(HuffmanNode root**): Return the number of non-leaf nodes in the Huffman tree
- **int getNumLeaves(HuffmanNode root)**: Return the number of leaf nodes in the Huffman tree
- **void serializeTree(BufferedBitWriter bw, HuffmanNode root)**: Write the serialization of the Huffman tree. See below for more helpful hints.

You do not need to modify writeHeader in any way!

Hints for the serializeTree Method

1. Be sure to serialize the tree using a **pre-order traversal.**
2. If we are at an internal node, write out the bit of 0.
3. If we are at a leaf node, write out the bit of 1 followed by the 8-bit representation of the character.

Suppose we have a letter stored in the variable Z. You can generate a bit string of length 8 by using the following code:

```
int letter = (int) z;
String bitString = Integer.toBinaryString(letter);
bitString = String.format("%8s", bitString).replaceAll(" ", "0");
```

For each ‘0’ in the bitString, use the BufferedBitWriter to write the 0-bit. For each ‘1’ in the bitString, use the BufferedBitWriter to write the 1-bit.

Creating the Output File

When creating your output file, **first call the writeHeader() method from the TreeSerlalizer class before writing your compressed text**. Remember, we want our compressed file to look like:

We have provided some helper files that will be useful during the encoding process. You do not need to modify these files in any way.

- **Utils/HuffmanNode.java**: Represents a node in the Huffman tree

- **Utils/PQ.java**: Priority Queue implementation from class

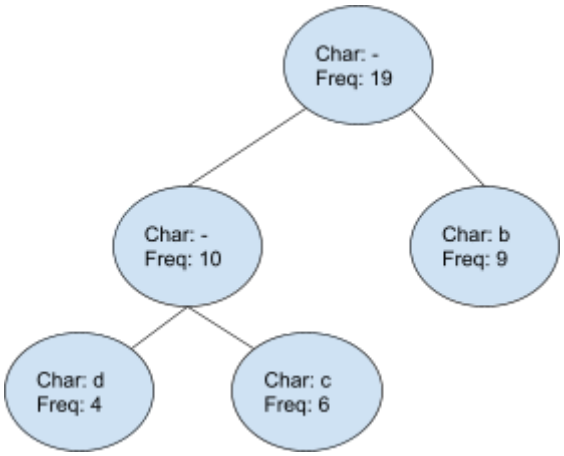
- **Utils/TextReader.java**: This file contains a function called **readNextChar()** that reads the next character in the input file. This function returns the ASCII values of the characters, so you will need to cast the result to a char data-type to get the actual character. If there are no more characters to be read, the TextReader returns an integer less than 0.

- **Utils/BufferedBitWriter.java**: Can be used to write bits to the output file. **Be sure to call the .close() method once you are done with a BufferedBitWriter instance**. Otherwise, your compressed file will not be saved correctly.

Huffman Decoding

Recreating the Tree

Recall that the first 32 bits in the compressed file represent an integer K, where K indicates how many bits were used to serialize the tree. We then read the next K bits (i.e. bits [33, 32 + K]) and recreate the Huffman tree using a pre-order traversal (remember we used a pre-order traversal to serialize the tree). During this process, we just care about recreating the structure of the tree. Suppose the tree we serialized during the compression phase was



The meta data in our compressed file would consist of the sequence of bits:
0…..00011101 00 101100100 101100011 101100010

The first 32 bits(highlighted in blue) is the binary representation of 29. This represents how many bits were used to serialize the tree. We then read the next 29 bits to re-create the structure of the tree. The sequences in red are the 8-bit binary representations of ‘d’, ‘c’ and ‘b’ respectively. When we rebuild the tree from the serialization, we don’t really care about the frequency attribute of each node. The frequency is not used in the decoding phase of the algorithm. We just need the structure of the tree to be right so that we obtain the right codewords for each character.

Implementing TreeDeserializer

In **Utils/TreeDeserializer.java**, we implemented a function called **readHeader** that takes in a BufferedBitReader instance. Your job is to implement the functions that readHeader depends on. **Do not modify readHeader in any way!** You have to implement the following functions:

- **int getSerializedSize(BufferedBitReader br)**: Return the integer that is represented by the first 32 bits in the compressed file. This value is then stored in the **bitsToRead** attribute of the BufferedBitReader class.

- **HuffmanNode deserializeTree(BufferedBitReader br)**: Return the root of the Huffman tree after reading the appropriate number of bits (aka deserializing the tree). Hint: You will need to use **bitsToRead** to determine when you have finished reading the tree serialization bits.

Decoding the Text

After we recreate the tree, we can then read the rest of the compressed file to recover the original text. We read the rest of the file (aka the content after the meta data) bit by bit. Whenever we read a 0 bit, we traverse down the left side of the tree. When we read a 1 bit, we go down the right side. When we hit a leaf node, we output the character stored at that leaf node. You repeat this process until you read all the bits in the file.

Implementing Decoding

Implement the Huffman decompression algorithm in **Decoder/HuffmanDecoder.java.** Put all your orchestration logic in the **decodeFile** method that is defined in **HuffmanDecoder.java. Do not modify this method signature in any way.**

We have provided some helper files you that will be useful during the decoding process

- **Utils/BufferedBitReader.java**: This file reads the bits one by one until there are no more bits to be read. It returns a -1 when all the bits have been read.

- **Utils/TextWriter.java**: This class has a function called writeChar that is used to write characters to an output file. Be sure to pass in the ASCII values of the characters to this function. Be sure to call the **.close()** method once you are done with a TextWriter instance. Otherwise, your uncompressed file will not be saved correctly.

Testing

For this project, we have provided integration tests that your encoding and decoding algorithms are able successfully compress and then decompress a file. You can find these tests in the **Tester.java** file in the **Utils** folder. That being said, you will be responsible for ensuring that the different components you implement in this project work correctly. We recommend that you write unit test cases as you go along. Debugging everything all at once at the end will be a major hassle! You do not have to submit the test cases you write.

We have given you a few sample input files you can use to verify that your compression and decompression algorithm work correctly in tandem. We also have two hidden files that we will not be releasing.

Grading

Your grade will be calculated using two criteria: whether your encoding is the optimal one and whether you can compress and decompress files correctly. We have setup an autograder on Gradescope you can use to make sure your code works correctly. Put only the following files in a folder called **huffman**:

1. HuffmanEncoder.java
2. HuffmanDecoder.java
3. TreeSerializer.java
4. TreeDeserializer.java
5. Any other non-testing .java files you may have created that we didn't provide for you.

Zip up the huffman folder and upload it to the Project3 assignment on Gradescope. You are responsible for ensuring the autograder can process your submission. If your code does not compile and run on the autograder, the maximum score you will earn on this project is 50%.

Compression Correctness:

Original Text Name	Point Worth
constitution.txt	5
lotf.txt	5
warandpeace.txt	5
Hidden File 1	5
Hidden File 2	5

The number of points you earn under this category will be dependent on what percentage of the compressed text our decoding algorithm can successfully recover when reading the compressed files your code generated. For example, if we can recover 50% of the characters of all the texts, you will receive 50% on the compression phase of the assignment.

Decompression Correctness:

Original Text Name	Decompressing Own Compressed File	Decompressing Our Text File	Point Worth
constitution.txt	2.5	2.5	5
lotf.txt	2.5	2.5	5
warandpeace.txt	2.5	2.5	5
Hidden File 1	2.5	2.5	5
Hidden File 2	2.5	2.5	5

The number of points you earn under this category will be dependent on what percentage of the compressed text you can successfully uncompress. For example, if you can recover 50% of the characters of all the texts, you will receive 50% on the compression phase of the assignment.

We will be passing in compressed texts that both your and our implementation generated. Your decoding algorithm should be able to decode any file that has the structure we mentioned in the Huffman Encoding section above.

Size Correctness

Original Text Name	My Compressed Size	Point Worth
constitution.txt	~25,432 bytes	3
lotf.txt	~322 bytes	3
warandpeace.txt	~1,811,854 bytes	3
Hidden File 1	~673,683 bytes	3
Hidden File 2	~82,876	3

We have provided the size (in bytes) of the compressed version of the files we were able to generate. Since the Huffman encoding algorithm is deterministic, you will be able to obtain the same file size if your implementation is correct. For every 2% larger your compressed file is than mine, you will lose 20% for that case. If our compressed file is 100 bytes and yours is 106, you'd get a 40% for that test case (i.e. 1.2 / 3).

Total Points available: 25 (compression + 25 (decoding) + 15 (size) = 65

Submission

Submit all your code to your own private Github repo.

Code Credit

Code for BufferedBitWriter and BufferedBitReader developed by Professor Scot Drysdale.