

**DATA STRUCTURE PROJECT REPORT**

## **“HUFFMAN TEXT COMPRESSION CODING”**

SUBMITTED TO: DR. JAWWAD SHAMSI

DATED: 08-1-2022

TEAM MEMBERS

* JUNAID SAYANI (21K-3401) (LEADER)
* FATIMA SALEEM (21K-3178)
* SHAFAQ SALEEM (21K-3198)

## **INTRODUCTION AND MOTIVATION**

The project we decided to pursue for DATA STRUCTURES COURSE was “HUFFMAN ENCODING AND DECODING”.

We use different software in our daily life yet never seem to be mesmerized by how it actually works on the back end. I have always been curious about how Zip and Windows Rar (win-rar) works: how it compresses my files and bring it back to the original format. The logic building was simple however it was more than that; the different time complexities through different data structures and the use of Greedy Algorithm (HUFFMAN ENCODING ALGORITHM) intrigued us, so we decided to dig deeper in it and to our astonishment we found the use of different combinations of data structure to build up an efficient compression algorithm and hence decided to take up on this mantle to build it ourselves.

## **BACKGROUND**

For over the years, different type of compression algorithms has been created and the most frequently used algorithm is Huffman Encoding Algorithm.

There are mainly 2 types of Compressions

1. Lossy Compression
2. Lossless Compression

In Lossy Compression, the ratio of compression is usually 70-90 percent i.e., it can range from 20:1 to 100:1. However in Lossy Compression, the original file is almost completely lost

In Lossless compression, the ratio of compression varies upon the size of the file, uniqueness of the characters, frequency ranges of the characters etc., but on an average case the compression ratio is from 30-60 percent, approximately restricted at 20:1 as in Lossless compression the original file can be recreated and no important data is lost.

We decided to build Lossless compression through Huffman Encoding with use of some data structures, such as Linked List, Priority Queues in Min Heap implementation, and Binary Trees. We managed and reduced our time complexity by trying different data structures and picked out the best possible combination, our data set was of 10-12 files all of them worked perfectly. We have covered all the possible case scenario files for e.g., a large file with less repetitions, a large file with more repetitions and less unique characters, a medium sized file with less repetitions, a medium file with more repetitions, and 2 small files. We noticed that in small files the text compression was limited to only 1-2 percent but if the uniqueness of characters was low, it went up to 10 percent. The rest files were mostly on Average Case Scenarios, the reasons and complete walk through of our code is further explained on the following page.

## **METHODOLOGY**

Huffman Encoding:

In order to read from the original file, it was needed to store the unique characters along with their frequencies (the number of times they occur in the original file). For this, we needed two different arrays; one array to store characters and the other array to store their frequencies. But since the number of unique characters was not defined, Dynamic Memory Allocation was the suitable option, but to resize the two arrays, it would cost us O(N) time. So to reduce time complexity, we decided to use non-vanilla implementation of Doubly Linked List to dynamically grow as we encounter a unique character. The traversal of linked list and to check for unique characters cost us O(N/2).

The two arrays of characters and frequencies were passed in our Heap and created a Min Heap so that the minimum value is at the top (root) that is because we are using priority queue where the priority of an element is the opposite of its value, i.e., the minimal frequency of character will have the highest priority, so that when Huffman Binary Tree is created, minimum is directly extracted from Min Heap and heapify the Heap. While building Huffman Tree, two nodes with minimum frequencies were extracted. The two frequencies were then added to form an internal node; all the characters are the leaf node. This process was repeated until all the characters were in Huffman Tree where character with minimum frequency was at the bottom of the tree. Minimum character was placed at the end of the tree so that we can afford longer code for less frequent character and smaller codes for more frequent characters.

We used binary tree as it has only 2 children: right child or left child. Binary value 0 was assigned to left edge while binary value 1 was assigned to right edge and traversed the tree from root to leaf node to produce code for each character.

We made use of Bit Buffers to write to the file, that is we made use of binary bit shifting to compress the file to merge different code sizes into Byte chunk sized data and wrote it to the file, we wrote a header line in the compressed file with the characters alongside their frequencies and separating them using section separator operator so that we can use it while decoding.

Collision: There can be many possibilities in decoding the code back to its original character for e.g., the code for 'a' is 010 and for letter 'p' is 010100 and for 'b' is 100 so if the code in compressed file is 010100 we do not know if the character of original file was, 'p' or 'ab' thus we had to resolve this.

Collision Resolution: We made use of Prefix Rule that is, the code for any other character can never be the prefix or included in the code of any other character, that is, all characters are in as Leaf Nodes and that paths can sure cross but characters cannot follow on from edges of other character leafs.

Decoding:

At first, the compressed file is opened and characters along with their code are retrieved. Then HUFFMAN TREE was rebuilt in the same manner i.e., binary value 0 was assigned to left edge and binary value 1 was assigned to right edge of the tree.

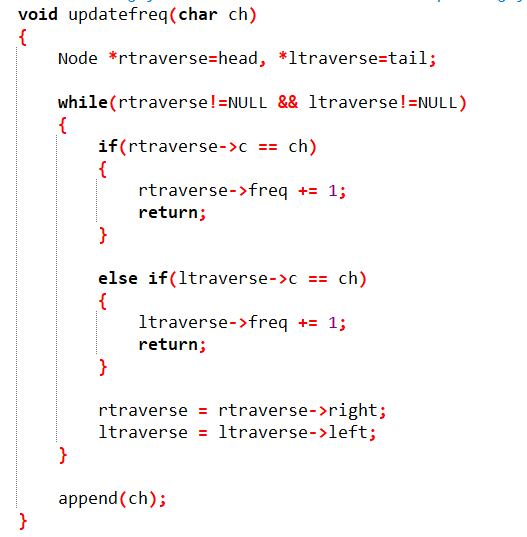
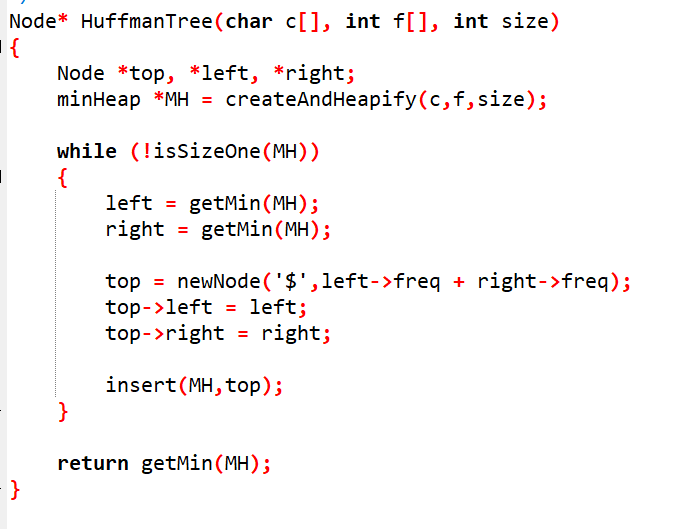
Leaf node was formed and the character was stored in it. This process was repeated for all the characters and thus HUFFMAN TREE was rebuilt.

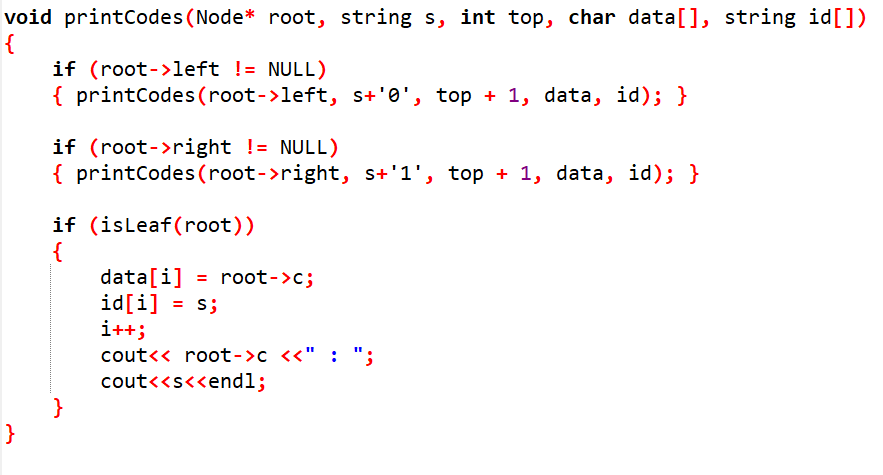
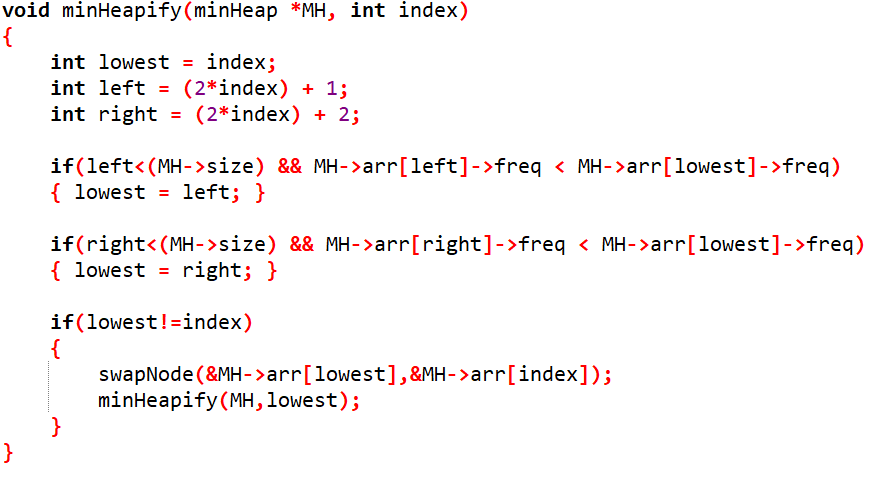
The remaining file was read bit by bit and the tree was traversed using 1/0 values. The corresponding character was written into the decompressed file as soon as leaf node was encountered in the tree. This process is repeated until the compressed file was completely read.

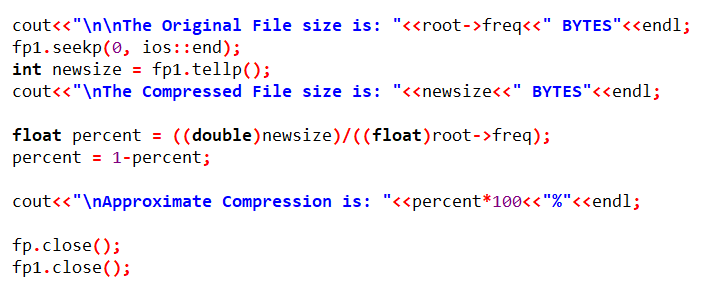
In this manner, we recovered all of the characters from our input file into a newly decompressed file with no data loss

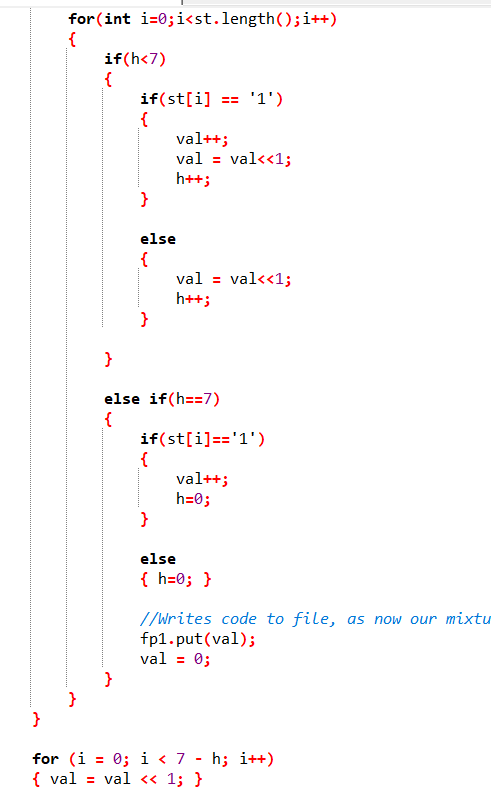
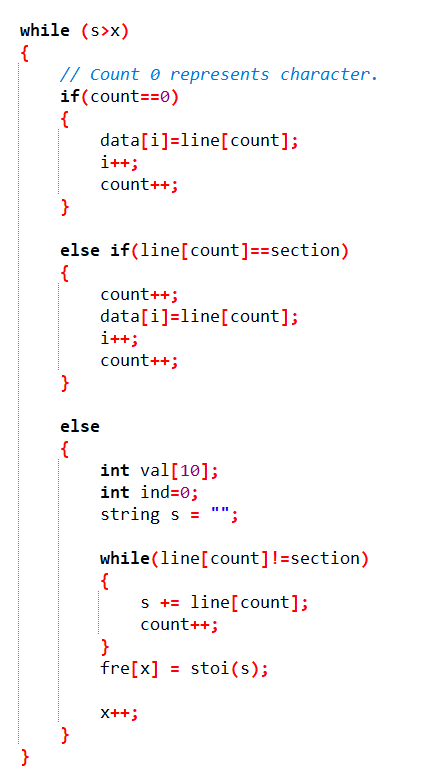
Following the steps above, we accomplished compression large text files and then decompressed them easily without data loss or quality loss.

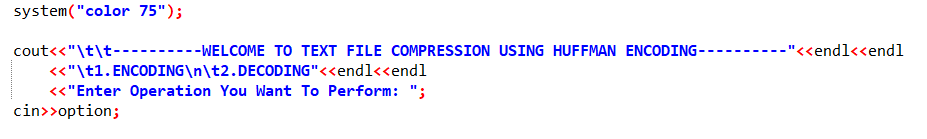
## **CODE SNIPPETS**

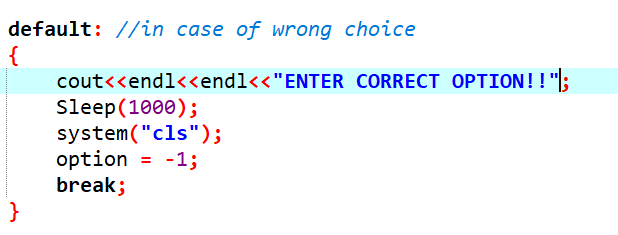
 

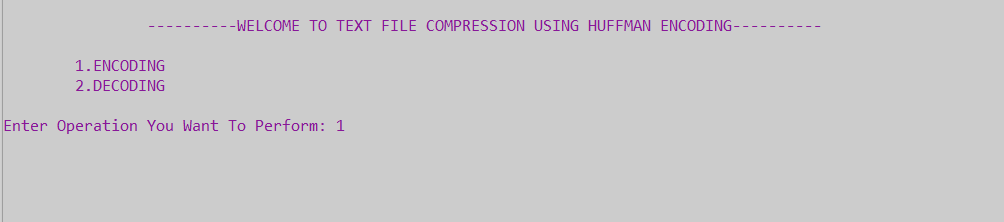


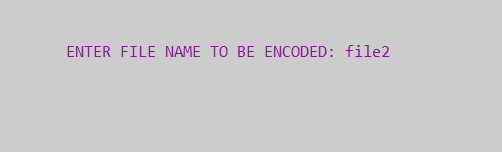
 

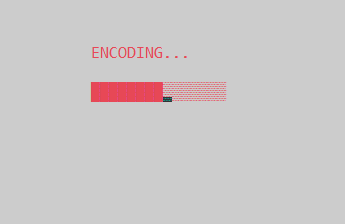


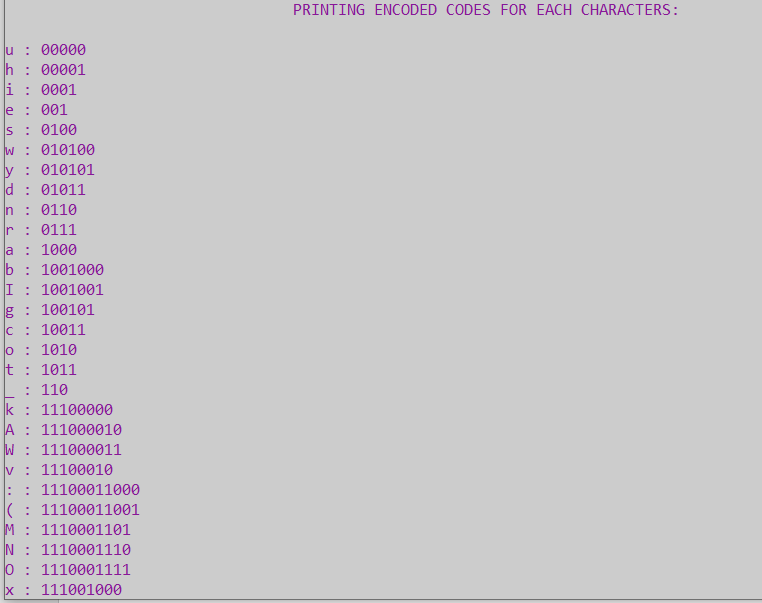


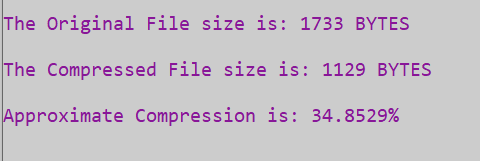
## **RESULTS**

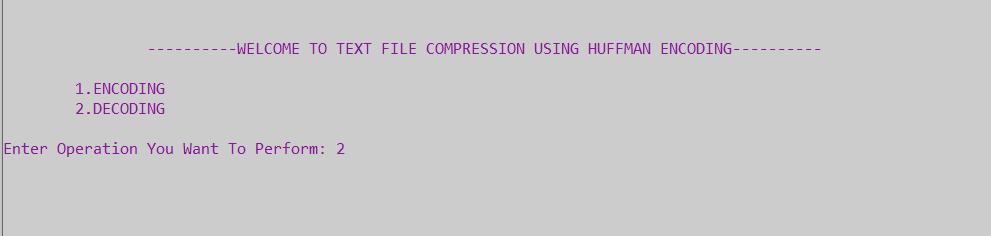


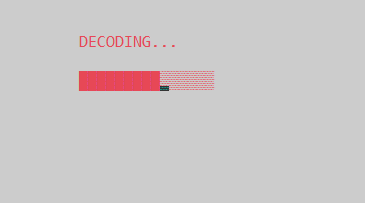


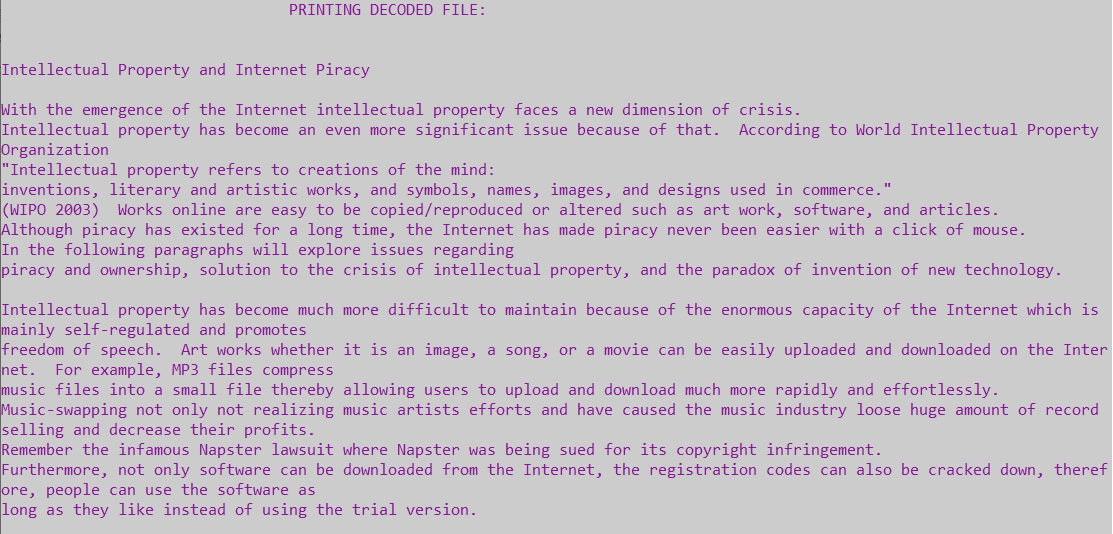




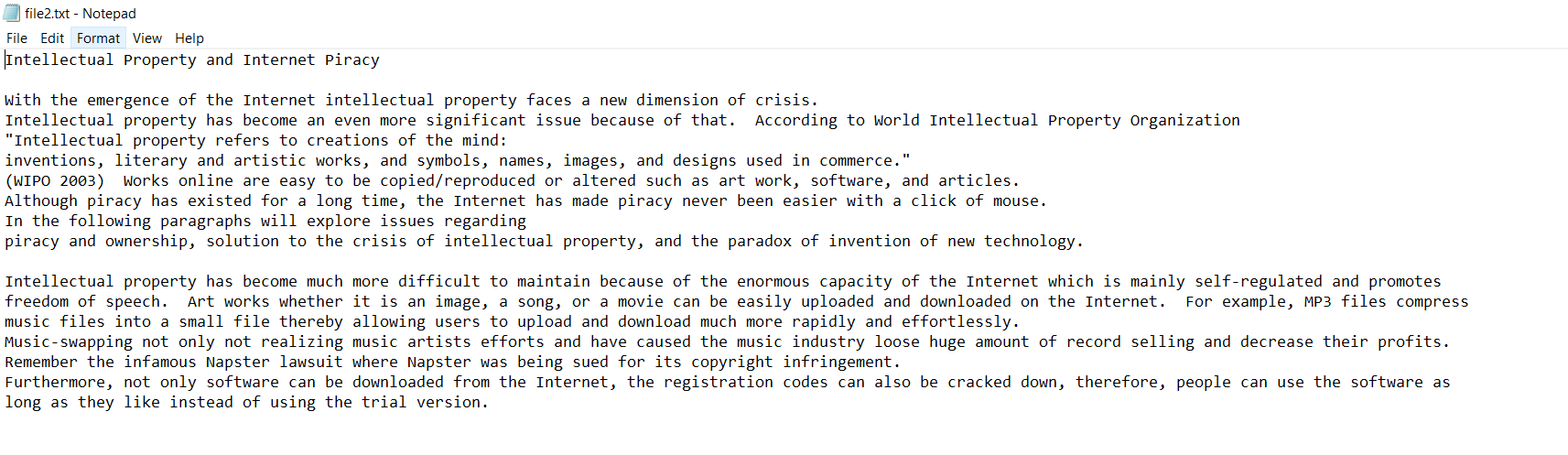








Original File: (file2)



## **ANNEXURES**

|  |  |  |  |
| --- | --- | --- | --- |
| **ORIGINAL FILE SIZE** | **COMPRESSED FILE SIZE** | **COMPRESSION %** | **COMPRESSING TIME** |
| 76 bytes | 58 bytes | 26% (many unique characters) | 0.6 seconds |
| 1733 bytes | 1129 bytes | 34.8% (average case scenario) | 1.15 seconds |
| 445 bytes | 371 bytes | 16.79% (many unique characters) | 1.15 seconds |
| 8691 bytes | 5049 bytes | 41.9% (average case scenario) | 1.17 seconds |
| 5533 bytes | 4476 bytes | 18.98% (many unique characters) | 1.15 seconds |
| 67 kb | 48 kb | 28.3% (average case scenario) | 1.77 seconds |
| 70 kb | 24 kb | 65.7% (less unique character) | 1.78 seconds |
| 58 kb | 25 kb | 56.21 % (less unique character) | 1.77 seconds |

## **Conclusion**

Our project is basically based upon the HUFFMAN ENCODING ALGORITHM or GREEDY ALGORITHM for compression and decompression of text files. We used different data structures for example, Linked List, Priority Queues in Min Heap implementation, and Binary Trees, in order to accomplish this task.

For retaining the uniqueness of the text, the Prefix rule was used; that is, the code generated for each character is unique and none of the code was prefixed to other.

Two arrays were created i.e., character array and frequency array. Character array stored unique character while frequency array stored frequency of that character. These two arrays were passed to Min Heap where it stored the minimum element at the top and the minimum elements were then used to build HUFFMAN TREE. While building tree, two nodes with minimum frequency was extracted and their frequencies were added together to form an internal node. Binary value 0 was assigned to left sub-tree while binary value 1 was assigned to right sub-tree and traversed the tree from root to leaf node to produce code for each character. Tree was traversed and the code generated was stored in Compressed File.

For decoding, code was read from the Compressed File and then HUFFMAN TREE was rebuilt in a manner where left edge was assigned 0 and right edge was assigned 1. The remaining file is now read bit by bit, and the corresponding 0/1 bit in the tree is traversed. The corresponding character is written into the decompressed file as soon as a leaf node is encountered in the tree. This is repeated until compressed file is completed read. In this manner, we recovered all of the characters from our input file into a newly decompressed file with no data loss.

Time Complexity: O(N\*logN) where N is the number of unique characters as an priority queue data structure takes O(logN) time per insertion.

### References:

* <https://nirav.com.np/2019/02/14/writing-huffman-compression-in-cpp.html>
* https://www.codesdope.com/course/algorithms-huffman-codes/#:~:text=Huffman%20code%20is%20a%20data,characters%20appearing%20in%20a%20file.