**INTRODUCTION**

Huffman tree is a type of binary tree that encodes a text to compress it but can also decode the text to decompress it. Since the computer stores its data in the form of bits, some files might contain millions of bits, which takes up more disk space, and this does not ease the transmission of data or files from network to network. So, Huffman coding is a solution to the large size file problem because it reduces the size of a given file or data by traversing the original data and summing up the frequencies of the encoded bits for compression. Indeed, Huffman tree is very efficient because it is used for lossless data compression; although it compresses the original file to a small-sized file, all the data are recovered.

**BACKGROUND**

The Huffman technique was a solution to the problem of finding an efficient binary code. In the quest to conquer a challenge, David A. Huffman and his classmates in 1951 strived and overcame the problem with the birth of the Huffman approach. The idea embraced the use of a binary tree such that data frequencies were sorted to produce the desired output (Huffman, 1952).

Huffman encoding and decoding is a technique used as a compressor for sourced data and also, the lossless transmission between the transmitter and receiver. As a data compressor, it makes use of the Binary Tree concept as its backbone. By forming binary trees, this data compression technique encodes message bits where inputs are given as symbols with their corresponding probabilities. Here, each frequency recorded is assigned as nodes that represent data in a tree structure.

Furthermore, in using the binary tree structure is created based on the frequency of the symbols which are later encoded and are equally decodable. Decoding information is made possible by tracing from the crown of the tree to the preferred symbol. With Huffman coding, any code allocated to a symbol is not a prefix of the code assigned to the other symbol (Vidhyaa et al., 2016). For this reason, the Huffman code is termed a prefix type code which further denotes that its decoding process is presented without difficulty.

The Huffman logic is said to have an advantage over other algorithms due to it being practical, static, its execution speed, closeness to an optimum and its depiction using a “look-up” table. The Huffman algorithm, as being practical presses the fact that compression executed is precise, with a realistic time interval. Huffman’s static nature argues that the algorithm can be generated once where various images with the same statistical features will be produced. Again, its execution speed provides an apparent output that the data obtained never gets missing. Huffman’s closeness to optimum vouches that for it being a data compression technique that is lossless, data decompressed results perfectly matches the initial one. In making minimal changes, the incorporation of a “look-up” table makes it simple since each pixel is typically coded in the form of one or successive further bits (Radhakrishnan et al., 2016).

**APPROACH**

The algorithm we developed to help achieve our goal represented by a flow chart,

Algorithm Huffman ():

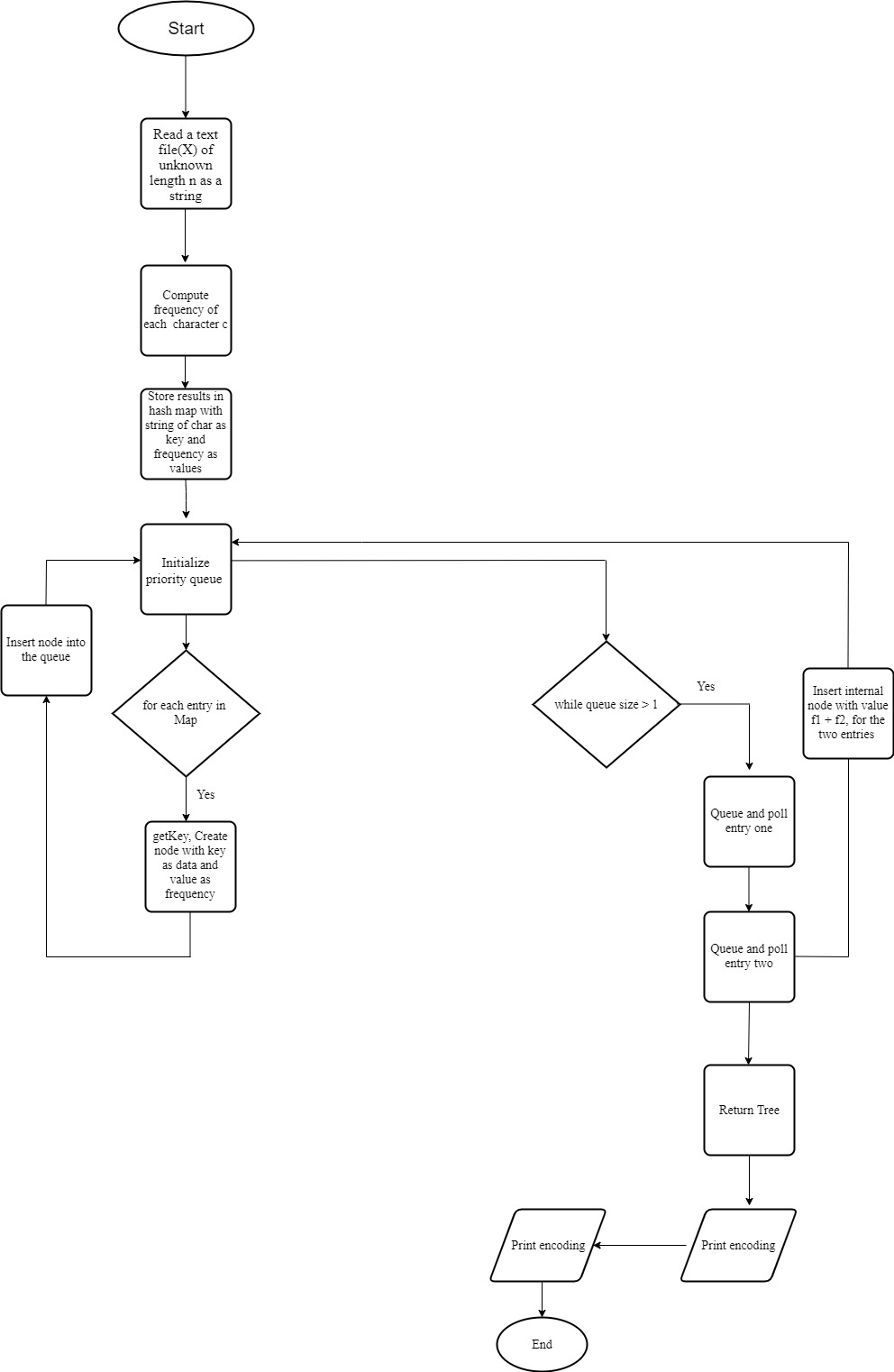
Input: Read a text file(X) of unknown length n as a string,

Output1: Encoded tree for X

Output2: Decoded tree for X

On a high level, we first read a text file as an input string. We create a hashmap storing the frequency of each occurring string character as the value and the character itself as the key. We then create a priority queue(reverse) and for each entry in the map create a single-binary node from it. We then traverse the queue until it is empty. Every time we traverse the queue, we remove two nodes with the highest priority and create an internal node, with the sum being equal to the frequencies of the extracted nodes. We then create a binary tree with its children as the nodes removed, and the internal node as its parent. We then insert the created internal node back into the queue. Then the last node remaining after traversing until the queue is empty becomes the root of the Huffman tree.

 A flow chart to show you how Huffman algorithm works;



Here is how we are implementing our Huffman tree, we first created an interface called Project Huffman to enable present a robust algorithm, here is what the interface does,

public interface **ProjectHuffman {**

public **HashMap<String,Integer>  readfile**(String inputFile) throws IOException;

public **void huffmanTree ()** throws IOException;

public **void printEncoded** (String inputFile) throws IOException;

public **void printDecoded** ();

**}**

**Key functionality one;**

**Algorithm** Hash<String,Integer> readfile(X):

**Input:** String file of length n with distinct characters

**Output:** Returns HashMap with characters as key and its corresponding value is the total frequency of the character in the file.

Initialize a FileReader and wrap a BufferedReader around the variable

  Initialize a HashMap

while line in file is not empty **do**

**if** (line is not a space)

split line into an array of words

**for each** in character in words **do**

**if** (Map has character)

**replace** (character and increment frequency**)**

**else**

**put** (character with frequency of one)

**return** HashMap

**Key functionality two;**

**Algorithm** void huffmanTree():

  Initialize a PriorityQueue of type huffmanNode

**while** treeQueue.size() > 1 **do**

huffmanNode left = treeQueue.**poll**() with left having key T1 and value f1.

huffmanNode right = TreeQueue.**poll**() with right having key T2 and value f2.

create **internalNode** of type **huffmanNode**

set frequency of internalNode as f1 +f2.

set leftChild and rightChild as left and right respectively.

set root as internalNode

add interNalNode to treeQueue i.e treeQueue.**add** (internalNode)

**Algorithm** void printEncoding():

    Load encoded text file

Initialize a FileReader and wrap a BufferedReader around the encoded file.

while line in file is not empty **do**

split line into an array of encoded characters

**for each** in item in encoded characters **do**

**print** out the items to console

Key functionality four:

**Algorithm** void printDecoding():

  Load encoded text file

Initialize a FileReader and wrap a BufferedReader around the encoded file.

while line in file is not empty **do**

**if** (line is not a space)

split line into an array of encoded characters

**for each** in item in encoded characters **do**

**for each** element in Map **do**

**if** element equals item at index **do**

**get** corresponding and **add** to decode string

**print** the decoded message

Reason for using priority queue is that that provides fast access to the minimum element so instead of sorted left → right, it is sorted top → bottom based on its implementation using a min heap it has a runtime of O(NlogN) in the worst-case scenario.  Also, the priority queue was used such that nodes with the lowest frequency is extracted first. Reasons for using a binary tree, for a binary tree we know that each node will have at most two children so traversing the tree is easy since we can encode each left traversal as 0 and a right traversal as a 1 easily which will be difficult for a general tree. Also, traversing the tree gives a Runtime of O(logN) in the worst-case scenario where the character to encode is in the innermost left or right child Reasons for using a HashMap is a java-based representation of the Hash-table which implements the Map ADT with the data unsynchronized and allowing null entries. We used a HashMap because we wanted to store each character and its corresponding total frequency. The map helped us store the characters as key and the total frequency as the value. Also, the get, put and replace operations of the map runs in constant which is very efficient for our implementation of Huffman encoding.

**RESULTS**

* The code was tested on a file of length 200. With the results, we observed that the code performed its function as intended. Encoding and decoding was successful. Limitations arise when we have a text file with over 1000 lines. Though some methods throw exceptions, they are not caught. Again, the Huffman class is not generic and therefore only works for string text files. Finally, we observe some code duplications when reading and writing to a text file. We believe this could have been improved by writing a text file for it.

**CONCLUSION**

* In this project, we encoded a text entered as a string, then later decoded the encoded data to get back the entered string. To accomplish that, we used three data structures; Priority queues, hashmap, and binary trees. The HashMap was used to store key-value pairs of each character entered and its frequency. A priority queue was then used to compare the entered frequency nodes, add the two lowest frequencies, then put the sum back in the queue in an ordered sequence. Finally, we created a binary tree with its root being the sum of the least frequencies polled, and the children being the polled frequencies. We learnt that Huffman encoding and decoding is by far the best technique to compress and decompress files.

**References**

V. G. Vidhyaa, S. A. Rajalakshmi, R. Raghavan, G. S. V. V. Gopal and R. Gandhiraj, "Huffman encoding and decoding algorithm using IJulia," *2016 International Conference on Communication and Signal Processing (ICCSP)*, Melmaruvathur, 2016, pp. 0587-0591. doi: 10.1109/ICCSP.2016.7754207

J. Radhakrishnan, S. Sarayu, K. G. Kurian, D. Alluri and R. Gandhiraj, "Huffman coding and decoding using Android," *2016 International Conference on Communication and Signal Processing (ICCSP)*, Melmaruvathur, 2016, pp. 0361-0365. doi: 10.1109/ICCSP.2016.7754156

David A. Huffman. A method for the construction of minimum redundancy codes, proceedings of the institute of radio engineers, 40(9):1098-1101, September 1952.