* **Approach**: Presents your high-level solution architecture and describes your approach to solving your problem. This includes a breakdown of your problem into key modules/pieces of functionality, a specification of the algorithm for solving each key piece (you may use structured pseudocode or a flowchart for this), an explanation of the data structures chosen, and why, and a specification of the classes comprising your implementation (you may use UML diagrams here)

The algorithm we developed to help achieve our goal,

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

* Compute the frequency f(c) of each character c of X.
* Store in a hash map M with character c as key and f(c) as values
* Initialize a priority queue Q.
* for each entry in M do getKey and Create a single-node binary tree T storing c with value f(c).
* Insert T into Q.
* while Q.size() > 1 do Entry e1 = Q.poll() with e1 having key T1 and value f1.
* Entry e2 = Q.poll() with e2 having key T2 and value f2.
* Create a new binary tree T with left subtree T1 and right subtree T2.
* Insert T into Q with value f1 + f2.
* Entry e = Q.poll() with e having tree T as its key.
* return tree T
* print encoding
* print decoding

On high level, we first read a text file as an input string. We create a hash map 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 highest priority and create an internal node, with the sum being equal to the removed nodes. We then create a binary tree with its children as the nodes removed and the internal node as parent. We then insert the created internal node back into the queue. Then the last node remaining after traversing till the queue is empty becomes the root of the Huffman tree.

Here is how we are implementing our Huffman tree, we first created an interface called Project Huffman to enables 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)

Key functionality three:

**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 printDencoding():

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's 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 Hashtable which implements the Map ADT with the data unsynchronized and allowing null entries. We used the 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.

Moving on three classes were used to achieve our goal, the projectHuffman class, the Huffman class which implements project Huffman and the test class which helps us to present our results.