# HUFFMAN ALGORITHM

## **Reading and sorting data** ( Line 49-151 | Main.cpp )

- Create an “ifstream” object, read the entire line and store the data in a character array
- If file has been read successfully, then delete the duplicate characters from the file using the algorithm of your choosing
- Sort the array in ascending order
- Use two nested for loops, both the original and the unique characters array to find out the frequencies of each of each character and store them in an array “frequencies”

## **Building Huffman Tree** ( Line 34-74 | UnoptimizedTree.cpp/OptimizedTree.cpp )

- Pass the size of the unique characters array to the object of the tree so that it can make its dynamically allocated data members in its parametrized constructor
- Pass the array of unique characters and their respective frequencies to the “buildHuffmanTree” function
- Create leaf nodes for each of the unique characters, where each node contains the character, the frequency, the left & right pointers
- **Unoptimized1:** Declare a simple pre-defined queue.
  Push every leaf node into the queue, the node that was pushed first will be at the top of the queue.
  **Unoptimized 2:** Declare a pre-defined max-priority queue, using vector & condition notation. Set the criteria in “condition” (in header file) so that it makes max-priority queue. Push every leaf node into the queue, they are organised in descending order with respect to their frequencies, i.e., the highest frequency node is at the top.
  **Optimized:** Declare a pre-defined min-priority queue, using vector and condition notation.

Set the criteria in “condition” (in header file) so that it makes min-priority queue. Push every leaf node into the queue, they are organised in ascending order with respect to their frequencies, i.e., the least frequency node is at the top

- Make left pointer point to the top node of queue and then pop queue. Make right pointer point to the new top node of queue and then pop queue. Declare memory for a new “parent” node which will have character ‘~’ and its frequency will be the sum of the two left & right nodes. Push that new “parent” node to the queue. This reduces the size of the queue by 1, as you are popping two nodes and pushing one at each iteration.
  Keep doing this until you are only left with one node in the queue. Now, simply make the private data member “root” point to that one node which is connected to all the other nodes.

**The only difference between an optimized tree & an unoptimized tree is that when you push the new “parent” node into the respective queues, the simple queue will place it at the end whereas the max-priority queue will place it in descending order according to its frequency, and the min-priority queue will place it in ascending order according to its frequency. This small condition drastically changes the shape of the trees with the min-priority queue making it optimal and the simple queue/max-priority queue lagging.**

## Generating Codes ( Line 79-103 | UnoptimizedTree.cpp/OptimizedTree.cpp )

- A recursive “calculateCodes” function is implemented and an empty string “code” is passed to it in its perimeters
- Follow two rules here:
  - Going left, add a 0 to “code”
  - Going right, add a 1 to the “code”
- When a leaf node is encountered, meaning a node that has any character except ‘~’, search for the character of that leaf node through the “nodes” array and save the code string in the same index as the one where the character has been found on the “nodes” array. Then, going either right or left, the “Root” pointer becomes NULL and the function returns.
  For example, if you find ‘a‘ at the leaf node, traverse the “nodes” array and find the node where that character lies. Suppose you found ‘a’ at “nodes[5].character”, now you save the “code” for ‘a’ at that same index “codes[5]”.

- Then, trace the path to the next leaf node, and this goes on & on until all the codes have been generated.

**Until now, the leaf nodes have been stored in "nodes", the codes in "codes" and the "root" has been pointed to the top of Huffman Tree.**

## Encoding & Decoding ( Line 123-186 | UnoptimizedTree.cpp/OptimizedTree.cpp )

- **Encoding**
- Receive the char array of "data" as the sole perimeter in the "encodeDecode" function.
- Read the characters of "data" index by index, search for the code of each character in "codes" array and concatenate it to the "encodedData" string.
  Keep doing this until all characters of "data" have been read and encoded.
- **Decoding**
- Make a "tempNode" and point it to the "root" of Huffman Tree.
- For decoding, traverse the "encodedData" string variable
- Stopping condition for the for loop is the size of "encodedData"
- If you encounter a 0, go left. If you encounter a 1, go right and keep doing this until the condition of leaf node becomes true ( left & right are NULL ). Once you reach that condition, add the character of that leaf node to the "decodedData" string variable.

## COMPRESSION RATIO ( Line 190-211 | UnoptimizedTree.cpp/OptimizedTree.cpp )

- Formula = 8 / ( (Summation(frequency*codelength) / no of chars )
- Find codelength for each character by using "codes[i].size()" and then multiply it by its respective frequency "nodes[i].frequency".
  Do this for all the nodes and codes in the corresponding arrays and then add them all
  Divide the sum by the total frequency (number of characters in file)
  Finally, divide 8 by the result of the previous calculations to get the compression ratio