

## Data Structures: Project Report

This project is implemented using OOP and Data Structures concepts. Project is mainly divided into two main parts and further divided into several parts which will be defined further. The main focus was to implement Huffman Tree of the data generated in the file and generate the unique codes for each character. The second task was to generate a Huffman tree of given data in an optimal state to decrease the compression ratio. All the work was done dynamically and efficiently keeping in view time and space complexity.

Following are the tasks which were implemented:

### **Task 1:**

The first task is to build a Huffman Tree and print the unique codes for every character obtained from the file. This task was further divided into below defined steps:

- **Computing Frequencies:** Firstly, we have implemented functions which reads the given file and compute the frequencies for each character obtained from the file.

**Note:** We have used classes to obtain/return multiple data types instead of using vectors.

Functions used:

1. ***CharFreqarrLen evaluateFreq (String):*** This function will return the object of class CharFreqarrLen which is having length of the unique characters and an array of objects of class CharFreq as data members, CharFreq class have unique character with its frequency as data member.

Hepler functions and classes:

2. ***bool checkChar (char \*, char):*** For checking character present in array.

3. **class CharFreq:** Contains unique character with its frequency.
4. **Class CharFreqarrLen:** Contains array of objects of CharFreq and length of unique characters found in the file/input.

- **Queue Implementation:** The computed frequencies and characters are then converted into data members of class huffmannode and then we declared a class Queue to store these objects in queue in ascending order. The queues were templated. For sorting of the queue, we used bubble sort.

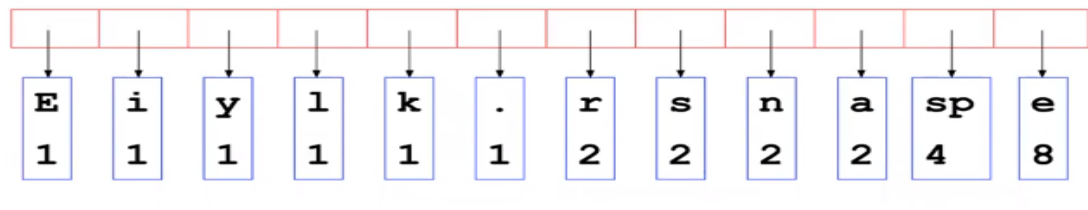
Functions and classes used:

1. **class Queue:** Templated queue class was used to generate Huffman tree.
2. **Class huffmanNode:** Contains character with its frequency, a left and right pointer to huffmanNode were also used as data members.
3. **void sortQueue (Queue<huffmanNode \*> &, int):** Used to sort the queue in ascending order, used Bubble Sort technique.

- **Building Huffman Tree:** After enqueueing these objects we dequeued these objects and made tree by using one of the algorithms used for making Huffman tree. Finally, the codes generated from the Huffman Tree were printed and compression ratio was computed from the generated codes.

Algorithm for building Huffman Tree:

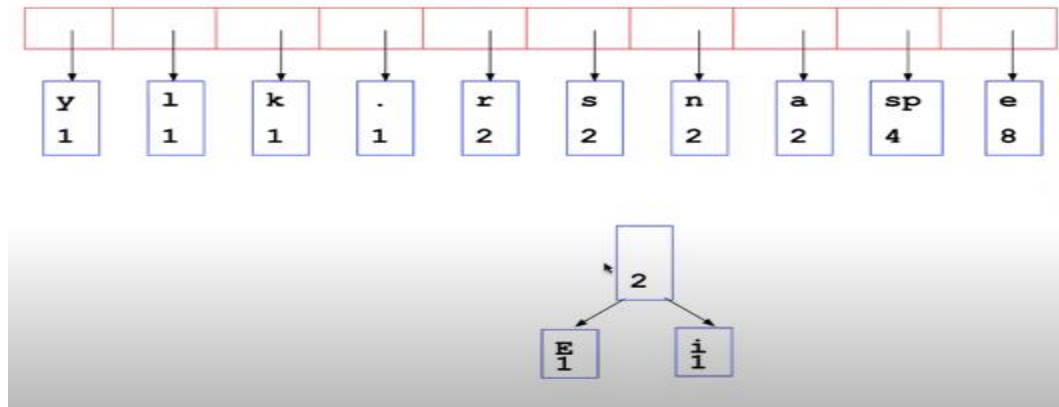
- We generated huffmanNode objects and stored them into queue in ascending order.



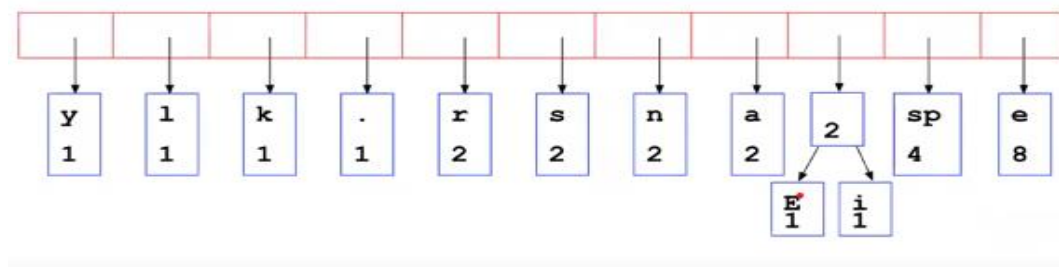
- By the help of queue, we dequeued first two elements/huffmanNodes (having lowest frequencies) and made

another huffmanNode object as newnode. We set the frequency of newnode by adding frequencies of dequeued nodes and made dequeued nodes as right and left children of newnode.

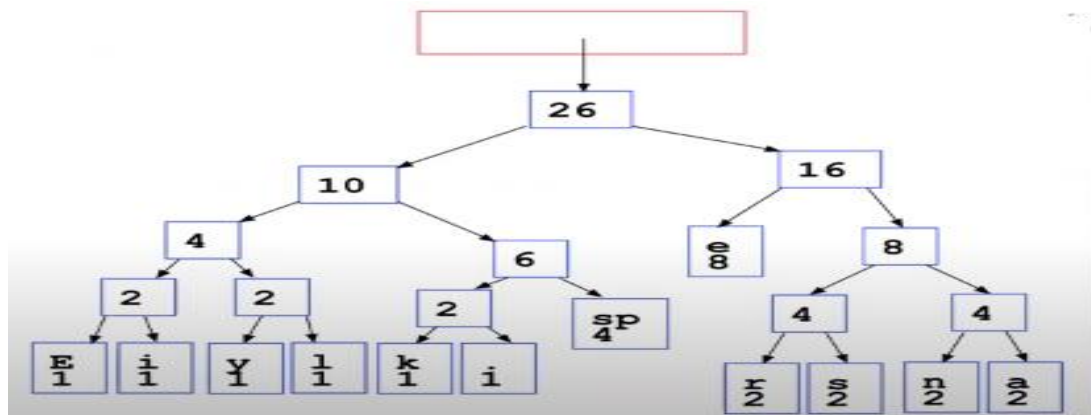
Note: We have made first dequeued node right child of the new node and second node as the left child of the new node as we were capable to implement desired task asked in assignment, this is opposite in below example picture.



- Then the next was to enqueue that newly made node in sorted order.



- We repeated this process until queue becomes empty and finally we got the Huffman tree.



### Functions Used:

1. ***huffmanTree(string)***: This function takes the string as argument and compute frequencies of unique characters by using above defined functions. Then it will make objects of huffmanNode. Then it will apply the above defined algorithm and generate Huffman Tree. Then we traversed the tree and printed the unique codes for every unique character. Finally, we computed the compression ratio by given formula in Project statement.

### Helper Functions and Classes:

2. ***class Queue***: Templated Queue class.
3. ***Class huffmanNode***: Contains character with its frequency, a left and right pointer to huffmanNode were also used as data members
4. ***CharFreqarrLen evaluateFreq(string)***: To evaluate frequencies of every unique character.
5. ***huffmanNode::enqueue(huffmanNode\*)***: To enqueue nodes.
6. ***huffmanNode\* huffmanNode::dequeue()***: To dequeue nodes.
7. ***void sortQueue(Queue<huffmanNode \*> &, int)***: To sort queue in ascending order using bubble sort technique.
8. ***void printCode(huffmanNode\*, string)***: Used to traverse the tree and print the codes generated for every character recursively.
9. ***float Sum\_FiLi(huffmanNode\*, float, string)***: Function to return the sum of  $(F_i * L_i)$  where  $F_i$  is the frequency of the character and  $L_i$  is length of the code generated for that character (as defined in Project statement).
10. ***bool checkBalance(huffmanNode\*)***: To check if the tree is balanced or not.

## Task 2:

The second task was to build an optimized version of the Huffman Tree and print the new unique codes generated for every character that were obtained from the file. The task was further divided in the following steps:

- **Optimized Huffman Tree**: After executing the simple Hoffman Tree we further optimized it by applying AVL tree concepts. We performed rotations on the unbalanced nodes of the already generated Huffman Tree to increase the compression rate even more as well as decreasing the bits.

Note: Sometimes the simple Huffman tree is already generated in optimal state so there will be a message printed in the program if such scenario occurs, moreover there will be no or slight difference in compression ratio.

Functions used:

1. ***balHuffmanTree(string)***: This function is same as the *huffmanTree(string)*, the difference is that it just balances the nodes before enqueueing them into the queue. That's how a balanced/optimized tree is generated.

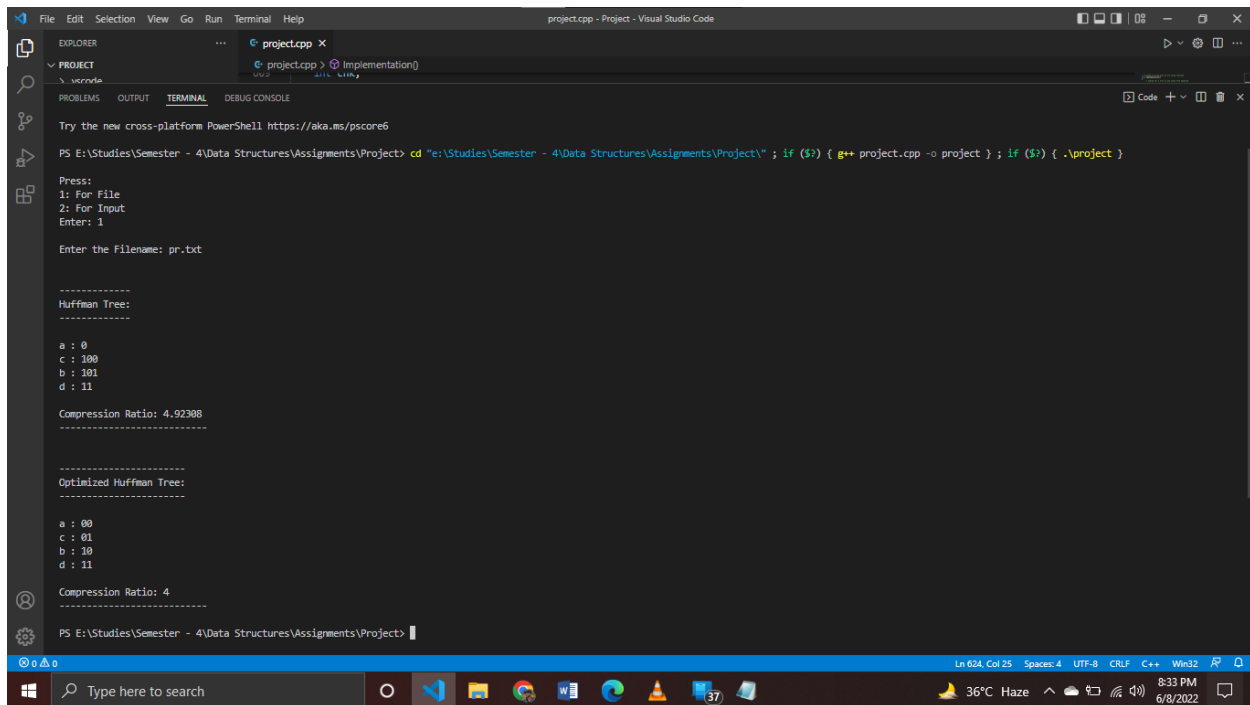
Helper Functions and Classes:

1. ***class Queue***: Templatized Queue class.
2. ***Class huffmanNode***: Contains character with its frequency, a left and right pointer to *huffmanNode* were also used as data members
3. ***CharFreqarrLen evaluateFreq(string)***: To evaluate frequencies of every unique character.
4. ***huffmanNode\* balance(huffmanNode\*)***: This function was used to balance the nodes, by applying AVL tree methods and techniques.
5. ***Int height(huffmanNode\*)***: To compute the height of the node.
6. ***int bfactor(huffmanNode\* )***: To compute the balance factor of the node.
7. ***huffmanNode\* rrRotation(huffmanNode\*)***: Right-Right rotation/anticlockwise rotation, which is applied on the edge below a node having balance factor -2.
8. ***huffmanNode\* llRotation(huffmanNode\*)***: Left-Left rotation/clockwise rotation, which is applied on the edge below a node having balance factor 2.
9. ***huffmanNode\* rlRotation(huffmanNode\*)***: Right-Right rotation/double rotation, first does *rrRotation* and then *llRotation*.
10. ***huffmanNode\* lrRotation(huffmanNode\*)***: Right-Right rotation/ double rotation, first does *llRotation* and then *rrRotation*.
11. ***huffmanNode::enqueue(huffmanNode\*)***: To enqueue nodes.
12. ***huffmanNode\* huffmanNode::dequeue()***: To dequeue nodes.
13. ***void sortQueue(Queue<huffmanNode \*> &, int)***: To sort queue in ascending order using bubble sort technique.
14. ***void printCode(huffmanNode\*, string)***: Used to traverse the tree and print the codes generated for every character recursively.
15. ***float Sum\_FiLi(huffmanNode\*, float, string)***: Function to return the sum of  $(F_i * L_i)$  where  $F_i$  is the frequency of the character and  $L_i$  is length of the code generated for that character (as defined in Project statement).
16. ***bool checkBalance(huffmanNode\*)***: To check if the tree is balanced or not.

## Conclusion:

The project main focus was to build a Huffman coding scheme and show the results, moreover the second task was to optimize the tree to reduce bits. Program was implemented in a way that it will perform both of the tasks efficiently on every character with various frequencies except character '\$'. While debugger, decompressor or dehuffing program is not implemented as that is not asked in project.

## A sample output of the code:



```
project.cpp - Project - Visual Studio Code
EXPLORER
PROJECT
S: source
project.cpp x
project.cpp > Implementation()
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Try the new cross-platform PowerShell https://aka.ms/powershell
PS E:\Studies\Semester - 4\Data Structures\Assignments\Project> cd "E:\Studies\Semester - 4\Data Structures\Assignments\Project\" ; if ($?) { g++ project.cpp -o project } ; if ($?) { .\project }
Press:
1: For File
2: For Input
Enter: 1
Enter the Filename: pr.txt
-----
Huffman Tree:
-----
a : 0
c : 100
b : 101
d : 11
Compression Ratio: 4.92308
-----
Optimized Huffman Tree:
-----
a : 00
c : 01
b : 10
d : 11
Compression Ratio: 4
-----
PS E:\Studies\Semester - 4\Data Structures\Assignments\Project>
```

Firstly a choice was given to user to input filename or to give keyboard input, then unique codes for characters based on Simple Huffman Tree was printed with compression ratio, then codes for optimized Huffman Tree was printed with decreased compression ratio and less number of bits for characters.

We have taken help from internet in understanding the implementation of the program, whereas code in the program is not copied from internet.

## **References:**

<https://youtu.be/PIV1BLrPQnc>

[https://youtu.be/co4\\_ahEDCho](https://youtu.be/co4_ahEDCho)

<https://engineering.purdue.edu/ece264/17au/hw/HW13?alt=huffman>

<https://www.programiz.com/dsa/huffman-coding>

<https://www.gatevidyalay.com/huffman-coding-huffman-encoding/>

<https://www.javatpoint.com/avl-tree>

<https://www.programiz.com/dsa/avl-tree>