CO658 Data Structures & Algorithms

Amir Mohamed

21905318

| **Module code and title:** | CO658 Data Structures & Algorithms | **Module leader:** | Guy Walker |
| --- | --- | --- | --- |
| **Assignment No. and type:** | CW 1 | **Assessment weighting:** | 100% |
| **Submission time and date:** | 26/05/2023 23:59 | **Target feedback time and date:** | Submission Date + 3 Weeks |

Contents

[Assignment Domain Selection: 2](#_Toc135825453)

[UML Diagrams: 3](#_Toc135825454)

[Task 2’s UML: 5](#_Toc135825455)

[Tasks Implementations and Console Screen Shots: 6](#_Toc135825456)

[Task1 Implementation Explanation: 6](#_Toc135825457)

[Task1 Console Screen Shot: 7](#_Toc135825458)

[Task 2 Implementation Explanation: 8](#_Toc135825459)

[Task 2 Console Screen Shot: 9](#_Toc135825460)

[Critical Justification for Using Data Structures in Huffman Coding Compression: 10](#_Toc135825461)

[References: 11](#_Toc135825462)

**Git:** [**https://github.com/Amir94Mohamed/CO658-CW1**](https://github.com/Amir94Mohamed/CO658-CW1)

Task 3 is available on GitHub in the main.cpp file of each task. All CPP files should have comments to explain.

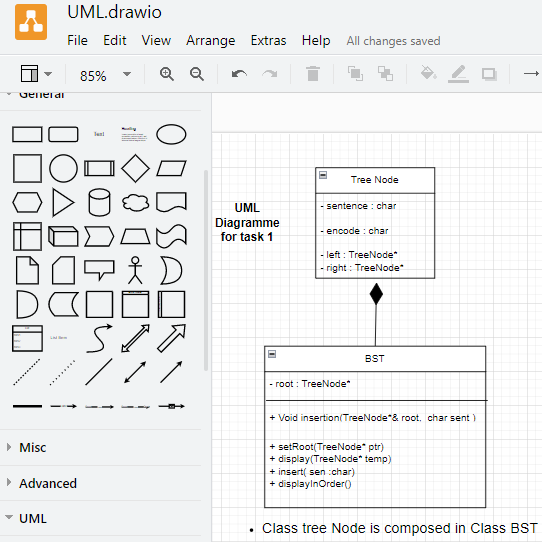
# Assignment Domain Selection:

For this assignment, the chosen domain is data compression basically Building a Data Compression Method for Text Files using Huffman coding compression techniques, specifically focusing on compressing text files. The goal is to convey the same information while utilizing a smaller amount of space. To achieve this, an understanding of how computers internally represent text is necessary.

Computers store information as a sequence of bytes, where each byte is an eight-bit value ranging from 0 to 255. To represent English text, a technique is needed to assign each character, including letters, punctuation symbols, special characters, and others, to an eight-bit sequence (a value from 0 to 255). This mapping is achieved through ASCII encoding, which utilizes 128 out of the 256 possible byte values.

In the implementation of this data-driven application, the requirements have been fulfilled by incorporating at least three different data structures: stacks, nodes, linked lists, and heap trees. These data structures are essential components of the Huffman coding compression algorithm, which is employed to compress text files. Additionally, the implementation also incorporates the concept of Object-Oriented Programming (OOP) to encapsulate data structures and algorithms into classes, promoting modularity and code reusability. Additionally, File handling has also been done.

UML Diagrams:Task 1 s’ UML:



## Task 2’s UML:

Tasks Implementations and Console Screen Shots:

## Task1 Implementation Explanation:

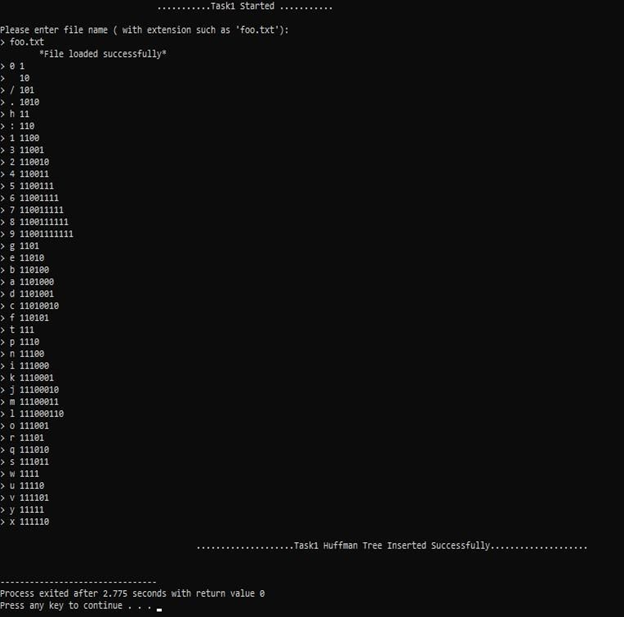
Firstly, the user will be asked to enter file name. Then if the file doesn’t exist message of **file not loaded** will be displayed on screen otherwise **file loaded successfully.**

After that file be read by character by character and store in string. After that from **for loop** character by character from string will be sent to Tree function where at a time Encoding will be calculated and will be placed with character node.

This tree function is almost like binary search tree function, but it includes some checks and constraints on which basis insertion is of **Huffman Coding** type and in these checks Encoding part is also done accordingly.

Now afterwards complete insertion characters along encoded bits will be displayed through **Pre Order display** function. And task1 ends successfully.

## Task1 Console Screen Shot:



## Task 2 Implementation Explanation:

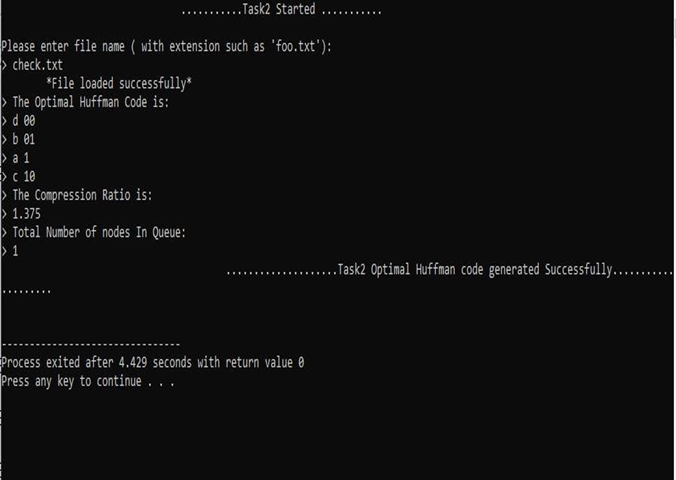
Firstly, the user will be asked to enter the file name which is foo.txt. Then if file doesn’t exist message of **file not loaded** will be displayed on screen otherwise **file loaded successfully.**

After that file is read character by character and all of the characters are stored in string. From there we iterate through the string character by character and insert it in the queue.

The data is stored in ascending order. After that Huffman function is called. Using this we will make a tree. In the Huffman function we use recursion. In it the function we use the a function which takes the top two nodes and creates a new node which has those two nodes as children and deletes these two top nodes and inserts the newly created node at its particular location. The function is called recursively until there is only one node left in the tree. After that we call a function encoding which encodes each character according to Huffman logic.

At last we also calculate the compression ratio. We iterate the tree using In-order logic. In this we multiply the frequency of character with the length of encoded character. Then we add it to a global variable. At last this sum of all the character’s frequency and their encoding length is divided by overall frequency. Thus, we get our compression ratio of optimal Huffman code.

## Task 2 Console Screen Shot:



Critical Justification for Using Data Structures in Huffman Coding Compression:

The implementation of Huffman coding compression algorithm relies on several essential data structures, including **stacks**, **nodes**, **linked lists**, and **heap trees**. These data structures play vital roles in achieving efficient compression and optimal encoding and decoding of characters.

**Stacks** are utilized in Huffman coding to store intermediate results during encoding and decoding. They enable the algorithm to traverse the Huffman tree efficiently in a depth-first manner, ensuring accurate encoding and decoding of characters. Additionally, stacks allow for optimal memory utilization by storing only necessary information at each step, reducing overall space complexity.

**Nodes** form the building blocks of the Huffman tree and carry information about character frequencies and their binary representations. They enable the construction of the Huffman tree through merging, sorting, and traversal operations. Nodes facilitate the mapping of characters to their compressed binary representations, resulting in optimal compression ratios.

**Linked** **lists** are employed to store and manipulate the frequency count of characters. They allow efficient insertion and deletion operations necessary for updating frequencies after character occurrences. Linked lists maintain an ordered list of characters based on frequencies, enabling faster access and manipulation of frequencies.

Heap trees, specifically binary min heaps, aid in selecting characters with the lowest frequency. The heap tree organizes nodes based on their frequencies, ensuring that nodes with the lowest frequency are readily accessible. This facilitates the selection and merging of nodes during Huffman tree construction, leading to optimal compression. Heap trees allow for constant time extraction of the minimum frequency node, reducing time complexity.

By leveraging these data structures, the Huffman coding compression algorithm achieves efficient compression, high compression ratios, and a balance between time and space complexities. The strategic use of stacks, nodes, linked lists, and heap trees optimizes the encoding and decoding processes, resulting in an effective compression algorithm.

References:(No date) *Huffman coding example | time complexity - gate vidyalay*. Available at: <https://www.gatevidyalay.com/huffman-coding-huffman-encoding/> (Accessed: 23 May 2023).

*Huffman coding algorithm* (no date) *Studytonight.com*. Available at: <https://www.studytonight.com/data-structures/huffman-coding> (Accessed: 24 May 2023).