# Text Compression Algorithms

**AQA A-level Computer Science NEA project report**

**A project which investigates two lossless compression algorithms being Huffman coding and Dictionary Data Compression.**

Name: **Jai Soni**

Candidate Number: **1284**

Centre Number: **37101**

Contents

[Text Compression Algorithms 1](#_Toc195254660)

[Analysis – Introduction 4](#_Toc195254661)

[Analysis – Problem 4](#_Toc195254662)

[Analysis – Interview process 4](#_Toc195254663)

[First Interview 4](#_Toc195254664)

[Research – Post First Interview (Huffman Coding and Shannon-Fano) 5](#_Toc195254665)

[Research – Post First Interview (Dictionary Data Compression) 8](#_Toc195254666)

[Second Interview 9](#_Toc195254667)

[Research – Post Second Interview (Huffman Coding) 10](#_Toc195254668)

[Third Interview 15](#_Toc195254669)

[Analysis – Objectives 16](#_Toc195254670)

[Documented Design 18](#_Toc195254671)

[User Interface 18](#_Toc195254672)

[Documented Design – Huffman Coding Algorithm 21](#_Toc195254673)

[Huffman Coding – Initial variables 21](#_Toc195254674)

[Huffman Coding - Pseudocode and Prototyping 22](#_Toc195254675)

[Documented Design - Dictionary Data Compression 35](#_Toc195254676)

[Dictionary Data Compression - Initial Variables 35](#_Toc195254677)

[Dictionary Data Compression - Pseudocode 36](#_Toc195254678)

[Documented Design – Files 40](#_Toc195254679)

[Files- Pseudocode and Prototyping 40](#_Toc195254680)

[Huffman Coding files 43](#_Toc195254681)

[Dictionary Data Compression Files 44](#_Toc195254682)

[Documented Design – Statistics 44](#_Toc195254683)

[Documented Design – Classes and methods 46](#_Toc195254684)

[Structures 46](#_Toc195254685)

[File Handling Class 47](#_Toc195254686)

[Dictionary Data Compression Class 49](#_Toc195254687)

[Statistics Class 55](#_Toc195254688)

[Huffman Coding Class 57](#_Toc195254689)

[Technical solution 67](#_Toc195254690)

[Testing 87](#_Toc195254691)

[Introduction To Testing 87](#_Toc195254692)

[Explanations To Tests 88](#_Toc195254693)

[Evaluation 97](#_Toc195254694)

[Statistical Analysis 97](#_Toc195254695)

[How well objectives have been met 99](#_Toc195254696)

[Supervisor’s evaluation 101](#_Toc195254697)

[Additional Features 102](#_Toc195254698)

[Research Links 103](#_Toc195254699)

# Analysis – Introduction

I am investigating how effective text compression algorithms are. Compression is the process of reducing a file size for a piece of text. An algorithm is a finite set of instructions used to solve a problem. There are two types of compression, lossy and lossless. Lossy is when the file size is massively reduced however the quality of the text is reduced which means that the compressed version cannot be turned back into the original. Lossless is when the file size reduction is much smaller in comparison to lossy however the compressed text can be turned back into the original text. I can conclude that my project will only contain lossless text compression algorithms so the compressed text can be turned back into the original. Some examples of lossless algorithms would be Huffman coding, RLE, Dictionary data compression and Shannon-Fano algorithm. Algorithms such as RLE would not be suitable for my project as it is likely that the “compressed” file size would be roughly double of the original.

A day-to-day use of compression algorithms would be with the amazon kindle. These devices typically have a low storage capacity, for example an 8GB kindle would have 6GB of usable storage. Using compression, the number of books which can be stored on a kindle can be increased. The use of compression could also reduce the amount of storage needed to be included with the device leading to a lower hardware cost in terms of manufacturing. Another example would be emailing attachments. When attaching a large file, it may take a while for it to be added to the email. Using compression, the time taken to add the file would be reduced as the file is smaller.

## Analysis – Problem

My supervisor for my project will be my friend Martin who is a book enthusiast and someone who finds compression interesting. He will also give me some listings of books to compress so then I have a range of samples to use for my statistical analysis when I complete the project.

## Analysis – Interview process

### First Interview

Me: What pieces of text would you like me to compress?

Martin: You can test a range of text however I would like the books An Inspector Calls to also appear among these tests. It does not need to be main test that occurs. I would also like you to test empty files as well to ensure that the program cannot be crashed that way as well. I would also like you to try some small files (less than 1MB) and also some larger files around 1MB of text. This would then showcase the project compressing a variation of text.

Martin: Do you know which lossless algorithms you are going to investigate?

Me: No not yet. I am going to do some thorough research and then come back to you once I have decided on the algorithms which I will investigate.

Martin: That is fine.

Now I am going to do some research on some lossless text compression algorithms

### Research – Post First Interview (Huffman Coding and Shannon-Fano)

Initially, I decided to look at the Huffman coding algorithm.

I used Wikipedia for the history of the algorithm - [Huffman coding - Wikipedia](https://en.wikipedia.org/wiki/Huffman_coding)

Huffman Coding is a greedy algorithm developed by David Huffman. Huffman created the algorithm whilst he was at MIT when he was assigned a term paper on finding a way of producing the most optimal binary codes when compressing a piece of text. He outdid his professor’s method at the time being Robert Fano (he worked with Claude Shannon) which was the Shannon-Fano algorithm.

The main difference between the Shannon-Fano algorithm and Huffman coding are that the Shannon-Fano algorithm does not always produce the most optimal codes (sometimes it does) and it is generally less efficient than Huffman coding. However, Shannon-Fano algorithm is a simpler algorithm to implement in comparison to Huffman coding.

I used Wikipedia to research how this algorithm works. - https://en.wikipedia.org/wiki/Shannon%E2%80%93Fano\_coding

How the Shannon – Fano algorithm works:

1. Creates a list of frequencies of characters in the text and sorts the list from lowest to highest frequency
2. The list is then split into two with having both sides of the list have their total frequencies as similar as possible
3. Assign a 0 to the left part of the list and 1 to the right part of the list
4. Repeat the 2nd and 3rd steps until all the characters are separated in their own groups

Here is an example of the Shannon-Fano algorithm below for the text SEE SEA SHELLS

A white background with many lines

Description automatically generated with medium confidence

A screenshot of a white screen

Description automatically generated

Here is a clearer version of the table shown in the image below

|  |  |
| --- | --- |
| Symbol | Binary code |
| E | 01 |
| S | 00 |
| L | 110 |
| SPACE | 10 |
| H | 1111 |
| A | 1110 |

The encoded text for SEE SEA SHELLS would be: 00 01 01 10 00 01 1110 10 00 1111 01 110 110 00

The compressed text has a total of 34 bits whereas the original text would have 112 bits (14\*8)

How Huffman coding Algorithm works

1. Sort the frequencies of characters from lowest to highest (add all of these to a priority queue)
2. Set the LP and RP of each character to -1
3. Remove the first two items from the priority queue (The smallest items)
4. Then create a new node with the LP of the new character being the node which contains the smallest frequency and then the RP with the node which contains the second smallest frequency. Add the new node to the priority queue.
5. Repeat step 3 and 4 until there is only one item left in the priority queue
6. Then traverse the tree. Traversing the tree to the right would result adding 1 to the binary code whereas to the left you would add 0 to the binary code

**A screenshot of a computer

Description automatically generated**

|  |  |
| --- | --- |
| character | Binary code |
| S | 01 |
| A | 000 |
| H | 001 |
| E | 10 |
| SPACE | 110 |
| L | 111 |

Here is an image of a Huffman tree generated online. This Huffman tree uses a linear priority queue like mine will

[Huffman Tree - Computer Science Field Guide](https://www.csfieldguide.org.nz/en/interactives/huffman-tree/)

The compressed text would be: 01 10 10 110 01 10 000 110 01 001 10 111 111 01 which is 34 bits whereas the original is 112 bits which is (14\*8). The depth of the tree is 3 as a maximum of 3 bits are used to represent a single character in the text.

In this case Huffman coding and Shannon-Fano algorithm both compress the text by the same amount however Huffman Coding is a more efficient algorithm. This is also why I disregarded the Shannon-Fano algorithm as it has been proven that Huffman Coding is more effective.

### Research – Post First Interview (Dictionary Data Compression)

I also decided to sketch the process for dictionary data compression. I used the same piece of text being “SEE SEA SHELLS”. Each unique word is given a unique number.

A white paper with writing on it

Description automatically generated

As you can see, Dictionary Data Compression is a much simpler algorithm used to compress a piece of text in comparison to Huffman coding. An issue which could occur could be the program crashing when a character which is not represented in 7-bit ASCII. For example, there are two speech marks a left one and a right one and only one is represented in ASCII. I have therefore decided that my code would therefore be using Unicode which means that it can accommodate all characters on the QWERTY keyboard. If I was to optimise my algorithm, I could sort the words from highest to lowest frequencies and therefore the words with the higher frequencies should be assigned a smaller code to represent the word.

Words such as **Books**  and **books**  would be classed as 2 separate entries despite them being the same word as my program will be case-sensitive. Another example would be **Shapes** and **Shape** which are very similar words but yet again 2 different entries in the dictionary.

A limitation of my dictionary data compression algorithm is that for texts with a large variety of words but a relatively low frequency of each of those words, the file size created could be larger than the original. This is also factoring that the dictionary itself would also need to be stored. Another limitation could be that the algorithm could be quite slow as there are at least 2 for loops needed for majority of the implementation.

If a number representation of a word made the representation of the word larger, I would just use the actual word itself when compressing the text.

For example, if I had the number **5678** which represents the word **got**, the word representation is smaller than the numeric representation therefore I would use that instead.

I would be storing the dictionary in a text file. Each unique word would be stored on a single line. This will make it easier to add the words to a list. The dictionary would be stored in a separate file to the compressed text.

Decompressing the text will involve looping through each of the compressed lines of text and then matching each code to each of the words in the text.

### Second Interview

Martin: Have you decided which compression algorithms you are going to investigate?

Me: I have decided to investigate Huffman Coding and Dictionary data compression. I can predict that some aspects of each of these algorithms will have a time complexity of O(n²) as I would have to loop through each line of text and each character in each of those lines.

Martin: Why have you specifically chosen these algorithms?

Me: I decided to use these algorithms because they are implemented in completely different ways. If I was to choose Arithmetic coding or Shannon-Fano algorithm for example, I believe it would be too similar of an algorithm to compare to Huffman coding as they are both involve probability/frequency-based methods to construct a tree-like structure. I believe Dictionary data compression and Huffman coding are different enough for me to explore lossless text compression algorithms. RLE is also unsuitable due to the very high chance that it would make the file size significantly bigger.

Martin: I would agree that they are suitable algorithms to investigate. A minimum requirement of both algorithms is that they can successfully compress and decompress accurately without any loss of data.

Martin: Do you fully understand each aspect of the project?

Me: I fully understand Dictionary data compression as it is a simpler algorithm compared to Huffman coding so I will come back to you again once I have conducted some further research.

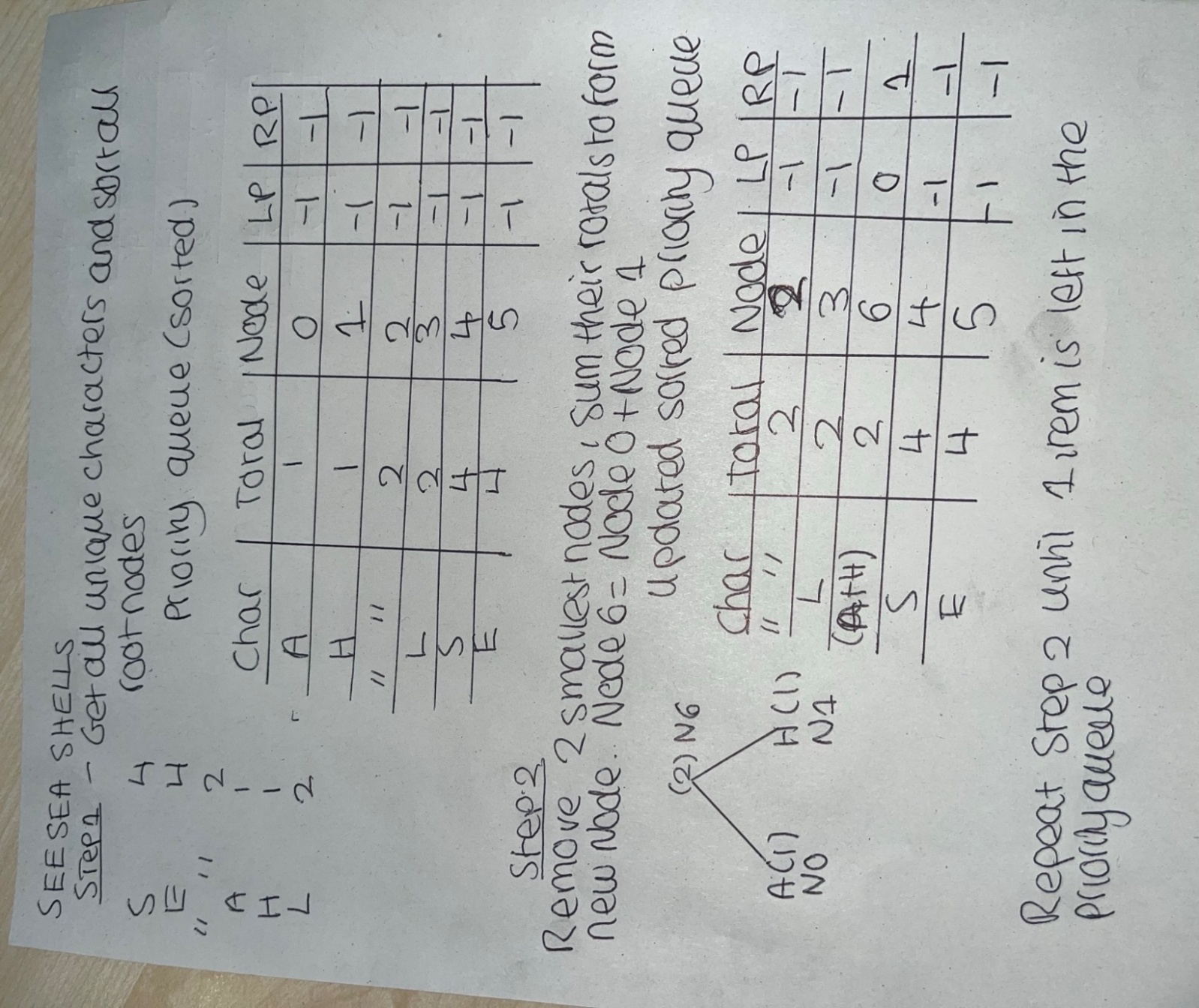
Martin: I agree. Before diving into the project, you must understand all key aspects on how it works.

### Research – Post Second Interview (Huffman Coding)

Here are some key terms for Huffman coding.

|  |  |
| --- | --- |
| **Term** | **Definition** |
| Tree | A connected undirected graph with no loops |
| Root Nodes | When either LP and RP = -1 |
| Rooted Tree | A tree where one node is a root node, and the other node is pointed in a different direction to the root node |
| binary tree | A rooted tree where each parent node has a maximum of 2 child nodes. |
| Huffman Tree | A binary tree |
| Variable-length | Means that the lengths of the binary codes are not fixed. The more frequent characters present in the text will have a shorter binary code. This will reduce the number of bits used to store the file. |
| Priority queue | Where elements in the queue are placed in order from highest to lowest priority. In this case the nodes which have the lower total will have a higher priority as these will be the nodes to be removed first |
| Recursion | When a function calls itself. Recursion will be used in the pre-order traversal when compressing and decompressing the text. Using recursion will be faster than using FOR loops |
| Preorder Traversal | If the LP and RP = -1 then display the binary code, otherwise recursively traverse the left subtree (child node) and then the right sub tree (child node) |
| Maximum depth of Huffman tree | The maximum depth of the tree is determined by the maximum number of bits to represent a single character. |
| Number of nodes in a Huffman tree | The number of nodes is 2X-1 where X is the number of unique characters in the text. A piece of text with 10 unique characters will have a maximum of 19 nodes as 2(10)-1 is 19. |
| Greedy algorithm | A greedy algorithm is an algorithm which chooses the most optimal choice in the hope of producing the most optimal solution. Huffman coding is a greedy algorithm as it produces the optimal codes for each character in a piece of text. |

I decided to sketch the process of building a Huffman tree using a very small piece of text being “SEE SEA SHELLS”. This allowed me to get a full understanding of the steps I would take.



A paper with writing on it

Description automatically generated

**A white paper with black writing on it

Description automatically generated**

**A white board with writing on it

Description automatically generated**

A white board with writing on it

Description automatically generated

As you can see, my final Huffman tree matches the one generated online which was shown earlier on page 7

To retrieve the decompressed text (the original text), I would store the tree generated when compressing the text into a text file. This text file would then be opened, and I would use that tree to then decompress the text.

### Third Interview

Me: I have researched both algorithms fully and believe I have sufficient knowledge to complete my project.

Martin: Ok that is good.

Me: What are the minimum requirements that you would expect of the project?

Martin: I would say being able to compress and decompress text using both algorithms and then being able to store the compressed text or the decompressed text in a suitable file and then showcasing the performance of both algorithms. I don’t have any benchmarks for these algorithms as the whole point of this project is to investigate how well they compress different pieces of text.

Martin: What other features would you like to include?

Me: Allowing users to view the frequency distribution of characters/words in the text as well as the key. These are so I can verify whether my project was successful in compressing and then decompressing. I will also have my project display the compressed text and the decompressed text as well.

Martin: I agree that they are suitable additions to the project.

Martin: What type of files will your program work with?

Me: Text files and binary files which can be accessed via notepad.

Martin: Will your program have a graphical display?

Me: No, I am going to use the console application for my user interface as creating a nice user interface is not the priority on the project. I will make sure that the user interface is easy to use and robust so will therefore deal with any invalid inputs.

Martin: That is fine, but could you clarify what invalid inputs would consist of.

Me: It would consist of inputting file paths which don’t exist, pieces of text which are clearly not file paths so therefore a file-path which doesn’t exist, making sure that the file inputted is the correct type for example binary files cannot be inputted for Dictionary data compression. I will also make sure that the only responses will be valid are the ones which are listed on the console display to the user. To do this, I will number each of the different menus

Martin: Thank you for the clarification.

Now that I have finished speaking to my supervisor, I can now create the objectives for the project.

## Analysis – Objectives

**Huffman coding objectives**

1. Create a robust Huffman Coding menu: The program cannot be broken by any invalid inputs. Invalid inputs would be invalid file paths (when requested) such as inputting binary files when a text file is inputted and not selecting a valid digit option from any of the menus. Choosing an invalid response will result in the user being requested to input another response. This will be repeated until they input a correct response.
2. Allow users to view the compressed text when using the Huffman coding algorithm. The user must also be able to store the compressed text in a suitable binary file. A valid binary file must not end in .txt and must also be accessible on the device which it is being stored on. Any invalid binary file should result in the user being requested to input another file path. When the compressed text is stored, a file called TreeSave must also appear in a suitable folder. Both the TreeSave file and the file which holds the compressed text must be accessible to the user.
3. Allow users to decompress the text using the Huffman coding algorithm when they input a binary file. If the user inputs a valid binary file, but it is not the file which corresponds to the tree, the user will be directed back to the main menu. If the binary file correctly corresponds to the tree, then the user can view the decompressed text. Then the user can then save the decompressed text in a suitable text file. A suitable text file must end in .txt and must also be accessible on the device which it is being stored on. The user must also be able to access the file which holds the decompressed text. The decompressed text must match the original text with 100% accuracy, and this will be tested by viewing the original text’s frequency distribution and the decompressed text’s frequency distribution.

**Dictionary Data Compression Objectives**

1. Create a robust Dictionary Data Compression menu: The program cannot be broken by any invalid inputs. Invalid inputs would be invalid file paths (when requested) and not selecting a valid digit option from any of the menus. Choosing an invalid response will result in the user being requested to input another response. This will be repeated until they input a correct response.
2. Allow users to view the compressed text for dictionary data compression. Users should be able to store the compressed text in a suitable text file. A suitable text file must end in .txt and it must be on a valid location on the device. The dictionary should also be stored in a suitable file. The dictionary and the compressed text must be accessible to the user.
3. Allow users to decompress the text. Then the user can then save the decompressed text in a suitable text file. A suitable text file must end in .txt and must also be accessible on the device which it is being stored on. The user must also be able to access the file which holds the decompressed text. The decompressed text must match the original text with 100% accuracy. This will be tested by making sure that the key and the frequency distribution of the original text is equivalent to the decompressed text’s frequency distribution and key.

**Statistics objectives**

1. Create a robust Statistics menu: The program cannot be broken by any invalid inputs. Invalid inputs would be invalid file paths (when requested) and not selecting a valid digit option from any of the menus. Choosing an invalid response will result in the user being requested to input another response. This will be repeated until they input a correct response.
2. Allow users to view some statistics basing the performance on both algorithms. This will include the file size percentage reduction as well as the time taken to compress the piece of text. This will be checked manually by totalling the size of the compressed text with its dictionary/tree using a calculator.

**All these objectives should be completed by the end of the project.**

# Documented Design

## User Interface

As explained earlier, my user interface will work via the console application. All inputs will be stored as a string variable to handle all exceptions, for example the options are

numbered so if the user inputted “one” or “hello” they would be classed as invalidated. If I had the options stored as integers, if a non-integer response was inputted, the program would crash.

A way that I would validate the options used in my user interface would be using a WHILE LOOP. As I am going to number each of the options below, WHILE the user does not input one of the valid options, they will be shown that menu again. This same logic will be used throughout the program when implementing the user interface for each section of my project.

A screen shot of a computer

Description automatically generated

A black background with white text

Description automatically generated

If the user inputs 1, they will be directed to the Huffman coding menu

**A black background with white text

Description automatically generated**

If the user inputs 2 they will be directed to the Dictionary data compression menu

**A black background with white text

Description automatically generated**

If the user inputs 3, they are directed to the statistics menu

Below is some further clarification on how my user-interface would work

**A screenshot of a computer program

Description automatically generated**

This will repeatedly show the main menu to the user until they exit the program. There will be two separate displays for the Huffman coding menu (one for text files and the other one for binary files). The statistics and the dictionary data compression will only be using text files.

**A screen shot of a computer program

Description automatically generated**

If the user views the key, then they can compress the text. The user can then choose to save their compressed text to a binary file. Obviously if the user wanted to view the frequency distribution, the frequency distribution would be displayed to the user. The user also can return back to main menu

If the user inputs a binary file they can decompress the text and save it to a text file if they want to. Otherwise they will be returned back to the main menu

**A screenshot of a computer program

Description automatically generated**

User can view the compressed text and then choose whether they want it saved to a text file. They can also do the same for decompressing the text. Like Huffman coding the user can view the frequency distribution, view the key and return back to the main menu

**A screenshot of a computer

Description automatically generated**

# Documented Design – Huffman Coding Algorithm

Initially I decided to work on the Huffman Coding algorithm as I knew it would be more complicated than dictionary data compression. Once I can compress and decompress text using both algorithms, then I will implement file handling.

## Huffman Coding – Initial variables

These are variables which I know that I would have to use when implementing the Huffman coding algorithm

|  |  |  |
| --- | --- | --- |
| Variable Name | Type | Purpose |
| Unique | A structure which contains   * Character (String) * CharacterFrequency (Integer) * CodeProduced (Boolean)   It will be an array with a fixed size of 1000 elements | Will hold all unique characters in the text. Will hold the frequency of each character, CodeProduced determines if the characters binary code has been displayed to the screen or not. |
| UniqueSize | Integer | Holds the size of the array. No point looping through empty spaces in the array and only looping through the number of unique characters the text contains |
| Node | A structure which contains   * RP (Integer) * LP (Integer) * Total (Integer) * Character (String) * BinaryCode (String) * IsProcessed (Boolean)   It will be an array with a fixed size of 1000 elements | Contains the pointers of each node in the tree as well as the total for the node. Will contain the character with its obtained binary code (If it doesn’t have one, I will leave them as empty strings). When building the tree, I need to check whether a node has already been looked at when constructing the tree. If it hasn’t then it can be used as a child node. That is what IsProcessed is used for. |
| TreeSize | Integer | Holds the size of the array. No point looping through empty spaces in the array and only looping through the number nodes which the tree has |
| PriorityQueue | List Of Integer | Will hold all the node totals. This will be sorted from highest priority to lowest priority with highest priority being nodes with the lowest total. |
| Lines | List Of String | Will hold all the lines which have been retrieved from the text file. |

## Huffman Coding - Pseudocode and Prototyping

Collecting all the unique characters in the text

A screen shot of a computer code

Description automatically generated

The purpose of this to collect all of the unique characters in the text as well as the frequency of each character.

This pseudocode loops through all the lines of the original text and then loops through each character in each line. If a character already exists in the array known as unique (), then increase the frequency of that character by one. If it is a new character, then add it to the array and increase the uniquesize variable by one to keep track of how many unique characters there are.

Using the example which I used earlier being the text SEE SEA SHELLS I will show what the expected result would be in the table below

Expected outcome

|  |  |
| --- | --- |
| Character | Character Frequency |
| S | 4 |
| E | 4 |
| Space | 2 |
| A | 1 |
| H | 1 |
| L | 2 |

This would be the expected outcome because, S is the first character which has been added to the array. The next character which appears is then E and then so on.

Console outcome

A screen shot of a computer screen

Description automatically generated

The pseudocode performs as expected with the first character being added to the array and if it re-appears then the frequency of the character is increased. This occurs for all the characters in the text

Sorting Algorithm

Now I have gotten all the frequencies for each character, I need to sort my priority queue from highest to lowest frequency (smallest to largest) Initially, I decided to use a bubble sort however even when modifying it to improve the efficiency, it was still taking too long when sorting a large set of frequencies.

The bubble sort implemented below checks to see if a swap has been made if one has then it is not sorted, doesn’t need to check the left furthest item from each pass as that item is already sorted.

Module program

Sub main()

Dim priorityqueue As New List(Of Integer)

Dim stopwatch As New Stopwatch

For a = 0 To 9999

Randomize() 'adds a random number to the unsorted queue

priorityqueue.Add(Int(Rnd() \* 1000))

Next

stopwatch.Start()

BubbleSort(priorityqueue)

stopwatch.Stop()

Console.WriteLine(stopwatch.Elapsed)

End Sub

Sub BubbleSort(ByRef priorityqueue As List(Of Integer))

Dim sorted As Boolean

Dim placeholder As Integer

Dim x As Integer = 0

Do

sorted = True

For i = 0 To priorityqueue.Count - 2 - x

'each time the last term is always sorted so no need to check it again

If priorityqueue(i) > priorityqueue(i + 1) Then

placeholder = priorityqueue(i + 1)

priorityqueue(i + 1) = priorityqueue(i)

priorityqueue(i) = placeholder

sorted = False

End If

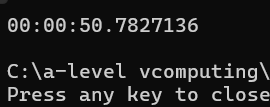
Next

x = x + 1

Loop Until sorted = True

End Sub

End Module

A black background with white text

Description automatically generated

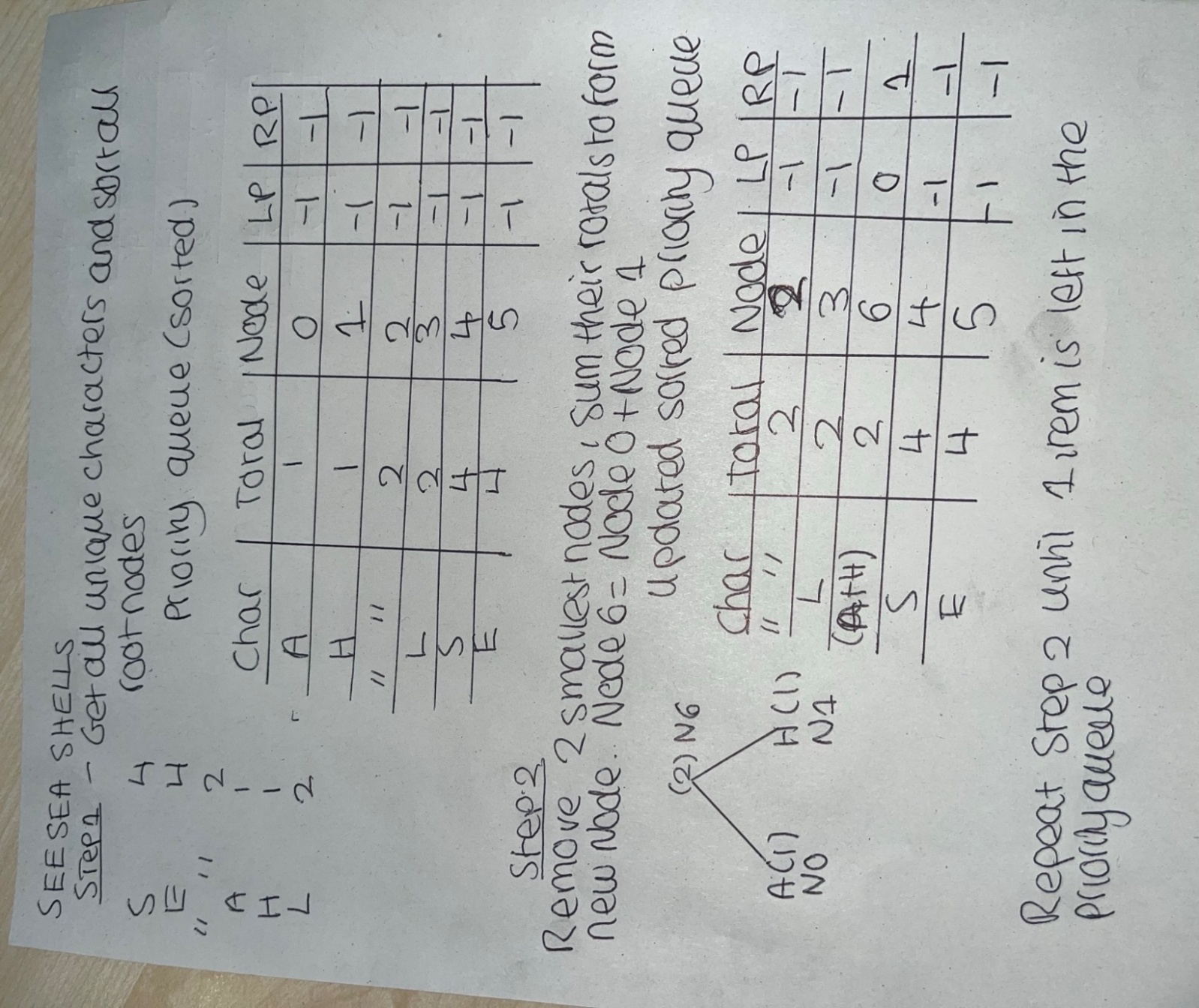
I ran my bubble sort algorithm with 10000 numbers to sort and took roughly 50 seconds. This is inefficient when sorting lists with a great range of characters. I researched multiple sorting algorithms such as merge sort but concluded with using a quick sort algorithm using Hoare’s partition. This algorithm is recursive. I will also be using this algorithm in dictionary data compression too when I sort the words from highest to lowest frequency.

Website I used to research Hoare’s partition algorithm. I used the pseudocode from here to implement the algorithm - [Hoare's Partition Algorithm - GeeksforGeeks](https://www.geeksforgeeks.org/hoare-s-partition-algorithm/)

https://www.geeksforgeeks.org/quick-sort-algorithm/

Constructing the Huffman Tree

I have copied and pasted the sketch I created of a Huffman Tree which I showed earlier down below.



A paper with writing on it

Description automatically generated

**A white paper with black writing on it

Description automatically generated**

Here are the steps that I would need to complete to construct the Huffman Tree

1. Sort the priority queue which contains the totals of nodes which are yet to be added to the tree. This will be sorted from lowest to highest frequency as the lower total will need to be assigned to a node earlier in the construction of the tree
2. Remove the two smallest nodes (This would mean I need to remove 2 items from the priority queue). This would be the first 2 items in the queue as it is a FIFO structure as it is First In First Out
3. Form the new node. I will list some steps below how to form the new node.
4. I then need to add the new node total onto the priority queue.
5. Recursively call the Huffman Tree subroutine until the size of the priority queue is equal to 1. This means that steps 1,2,3 will be repeated.

How to form a new node

* As the two items removed on the priority queue are two integer values which are the frequencies of the nodes, I need a way of matching the frequency back to the corresponding node in the tree. To do this I will use a FOR loop to navigate through the tree’s contents. To make sure that I do not use the same node twice, I will need a Boolean variable to determine if the node has been used or not. This is contained in the node structure which will be explained more in detail on page 45. I also need to make sure that when matching the two frequencies to nodes, that a maximum of 2 nodes is selected as those 2 nodes will be the children nodes for the new node. Yet again I will be using 2 Boolean variables to do this.
* Once the two nodes have been found, I need to set the LP of the new node as the node which had the lower frequency and then the RP of the new node as the node which had the higher frequency of the two removed nodes
* I also need to add the totals of the two removed nodes to form the total of the new node

A screenshot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generated

The console output matches what my sketch demonstrates with all the nodes in the correct place and the tree’s construction is shown to have been completed once there is only one item in the queue remains.

Traversing the Huffman Tree – Assigning Binary Codes

A drawing of a diagram

AI-generated content may be incorrect.

How a pre-order traversal works using the sketch of a tree shown to the left using nodes between A and F.

* A is the node at the top of the tree so this will be the first node to be looked at
* The left subtree of A is traversed so you then arrive at node B
* Then traverse the left subtree of node B so you then arrive at node D
* As node D has no children nodes, the right subtree of node B is then traversed so you arrive at E
* As node E has no children nodes, the right sub-tree of A is traversed
* This means that you arrive at node C. Then node C’s right subtree is traversed where you arrive at node F

**A → B → D → E → C → F** is the order that the tree above is traversed

As you can see all of the nodes in the tree are explored when traversing the tree.

A screenshot of a computer code

Description automatically generated

Above is the general pseudocode for a preorder traversal. As you can see it is recursive as the subroutine calls itself. The preorder traversal will traverse the tree until it reaches a root-node. Every time that the tree traverses the tree, a binary digit is added to the tree. Every time the right side is traversed, a 1 is added otherwise a 0 is added to the string which will hold the binary code.

A binary code will only be produced if

* The LP and/or RP is equal to -1. This means that the current node is a node location of a character in the text.
* If the binary code for a particular character has not been produced yet (I would be using unique().codeproduced which is a Boolean variable to determine this. This will be explained in further detail on page 45)
* A Boolean variable to make sure that a binary code for one character is also not given to another character by making sure that only one character is assigned a binary code for every traverse of the Huffman tree.

A black background with white text

AI-generated content may be incorrect.

Without this Boolean variable, some of the characters have the same binary codes and this is shown using the image above.

The expected binary codes for the text SEE SEA SHELLS would be

|  |  |
| --- | --- |
| Character | Binary code |
| A | 000 |
| H | 001 |
| S | 01 |
| E | 10 |
| SPACE | 110 |
| L | 111 |

A black background with white text

AI-generated content may be incorrect.

As you can see with the pseudocode implemented, these binary codes match the expected results

Creating Binary Lines and splitting into bytes

The image below shows the encoded text in binary. This image was shown earlier in the analysis section.

A white board with writing on it

Description automatically generated

A white background with black text

Description automatically generated

To convert all of the text into binary, I will need 3 FOR loops

1. The first loop will be looping through each line of text
2. The second loop will be looping through each character in each line of text
3. The third loop will be looping through the tree to check to see if the character in the tree matches the character which is currently be looked at as I have a copy of the character which a binary code represents stored in the tree. The corresponding binary code for that character will be added to the compressed text

The next part is also quite straight forward where I would split each binary string into 8-bit bytes and then write the specified character to a binary file.

The image below shows how I would convert the binary text into bytes

A paper with numbers and digits

Description automatically generated

This works by making sure that the length of each byte is exactly 8 bits.

Converting from Binary back to the Original Text

I will save and retrieve the Huffman Tree from a text file. The file will also contain the length of each binary lines. This will help me keep track of how many characters will be added to the original line. Below are the steps needed to complete this task

1. Convert the characters stored in the Huffman file back into a single line of binary. In order to do this, I would use a StringBuilder which is much faster than using regular strings. I would then convert it back int a string once this bit is completed.
2. While the length of the individual binary line doesn’t match the length of the binary line stored add bits from the binary line which contains all the bits. This will then form a list of binary lines. This list is then used in step 4 and broken down even further making it easier to manage the bits
3. Repeat step 2 until all binary lines have been reformed.
4. Then add each bit in a single binary line into a list.
5. Call a subroutine called PreorderTraversalDecoding which is recursive. This will traverse the tree until a LP or RP of -1 is found. It will then check through the stored tree to see which character the current binary codes correlates to and then add the character to the new string of text. Remove the number of bits used to represent the character from the list containing all the bits.

Completing this task will be much easier to accomplish if I use 3 separate subroutines.

* One subroutine will retrieve all the characters stored in the binary file and then convert them back into binary whilst making sure the length of the total text matches the original. (Steps 1,2 and 3)
* One subroutine which will add each binary digit from a single encoded line of text into a new list. For example, 10111 would have the list

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 |
| Binary digit | 1 | 0 | 1 | 1 | 1 |

This will make it easier to then keep track of which bits have been converted back into binary as I would then remove them from the list. This list behaves like a queue as it removes the bits which were added to the list first. This is a FIFO (First in first out data structure) I will then call the preorder traversal subroutine

If A was equal to 10 and B was equal to 111 then if the current code is 10 then remove the bits 10. If the length of the line is equal to 0, I would process a new line and would do so until there are no encoded lines of binary text remaining. This would then use a WHILE loop. (Step 4)

|  |  |  |  |
| --- | --- | --- | --- |
| Index | 0 | 1 | 2 |
| Binary Digit | 1 | 1 | 1 |

* The second subroutine will recursively traverse the tree and see if the current code obtained from traversing the tree is equal to a binary code stored in the tree. If it is then adding the character equivalent to the list of original text. This will also remove binaryLines(0) when binaryline(0) length = 0. (Step 5)

To complete steps 1,2,3 I am going to have to:

* convert each character back into a decimal number and then convert each decimal number back into binary. I also need to make sure that when the final decimal number is being converted back into binary, that its length is 8 bits. All the bits accumulated from the entire text will be stored in a single list.
* Then I am going to have to add N number of bits. N is the number in index 0 of a list which will contain the length of each line in binary. When N bits have been added to the binary line string, the binary line length will be removed from the list and the bits which have already been used will also be removed from the list containing all the binary bits in the text. This will be completed when the list which holds all of the lengths of binary lines is empty.

Here is some pseudocode for the steps 1,2,3

A white background with black text

AI-generated content may be incorrect.

In this pseudocode, SingleLineInBinary would be a list of string, BinaryLineLength would be a list of integer and FullBinary will be a string

Down below, I have included an image of pseudocode used for steps 4 and 5

A screenshot of a computer code

Description automatically generated

# Documented Design - Dictionary Data Compression

## Dictionary Data Compression - Initial Variables

These will be variables which I will use when implementing dictionary data compression.

|  |  |  |
| --- | --- | --- |
| Variable Name | Data Type | Purpose |
| WordBank | List of string | Holds all the unique words used |
| WordFrequencies | List of integer | Holds the frequencies of all unique words. |
| EncodedLines | List of string | Holds all of the encoded lines of text |
| Lines | List of string | Holds all of the lines of original text |
| OriginalLines | List of string | When converting from compressed text to decompressed text, this will be used to hold the original text. |

The **WordBank** and **WordFrequencies** variables will typically be accessed parallel to one another. For example, for the text **AQA Computer Science**

|  |  |  |
| --- | --- | --- |
| **Index** | **WordBank** | **WordFrequencies** |
| 0 | AQA | 1 |
| 1 | Space | 2 |
| 2 | Computer | 1 |
| 3 | Science | 1 |

This will allow me to modify the WordFrequencies and WordBank lists in the same FOR Loop. This will reduce the number of loops needed.

## Dictionary Data Compression - Pseudocode

Collecting all the unique words

Here are the steps I would need to collect all the unique words in the text

* Loop through each line in the text
* Loop through each character in each individual line
* If the current character is not a piece of punctuation which is not used to join words together (so it could be a digit or a letter) then add the character to a string which is currently holding the word.
* If the current character is a piece of punctuation which is not used to join words together (Two examples of punctuation which joins words together is **‘** and **-**). Then check the piece of punctuation (the current character) and the word which has been accumulated to the current list of words. If the current list of words does not contain the word and/or the character, then add them to the word bank. If the word and/or the character already exists in the word bank, I would need to increase the frequency by 1.

To do this, I will probably create 2 different subroutines

1 subroutine will check if the word/character is already in the word bank and then the other one would be which loops through all the lines and each character in each line accumulating the word. This would reduce repeated code as I would need to check if a word exists in the word bank twice. In total, this will require 3 FOR loops between the two subroutines.

I would then sort the WordBank list and the WordFrequencies list from highest to lowest frequency. This ensures that words with the greater frequency are represented using smaller numbers. Like in Huffman Coding, I would be using the quick sort algorithm using Hoare’s partition. This is a quick and efficient way of sorting the frequencies of all the words in the text as for pieces of text with several thousands of words, using a bubble sort would be extremely slow. However, I will be sorting the list from highest to lowest frequency whereas Huffman was from lowest to highest frequency. The quicksort algorithm will use both the WordBank and WordFrequecies lists

Compressing the text

To compress the text, I will be using 2 different subroutines

1 subroutine will loop through each line of text and loop through each character and will check to see if the character that is currently being looked at is a piece of punctuation which does not join 2 words together. (This was explained earlier on page 36)

The other subroutine will be used to match the corresponding code to each word in the text. This will use a single FOR loop to accomplish this. The FOR loop will be looping through the entire word-bank which was constructed earlier. When the corresponding location of the word is found, I also need to determine whether the “compressed” numerical code is smaller than the original. If the numerical code is bigger then, I would just use the actual word instead. To determine this, I will use an IF statement

Here is some pseudocode below for the following subroutines

A screenshot of a computer code

AI-generated content may be incorrect.

If the number representation of a word is longer than the actual word/symbol itself then use the actual word/symbol instead.

This will increase the percentage of which the text is compressed instead of encoding each word as a number. Adding a space between each encoded word makes it easier when distinguishing when a new word comes along in the text.

For the Text SEE SEA SHELLS the expected result would be:

**Space** being 0

**SEE SEA SHELLS** being allocated 1,2 or 3 in any order as the key is corresponding to from highest to lowest frequency and all of these words have the same frequency

The higher the frequency, the lower the number used to represent that word/symbol

A screenshot of a computer

AI-generated content may be incorrect.

A black background with white text

AI-generated content may be incorrect.

Here is the key for the text SEE SEA SHELLS

A screen shot of a computer program

AI-generated content may be incorrect.

As expected, **Space** was allocated 0 and then **SEE** was allocated 3, **SEA** was allocated 1 and **SHELLS** was allocated 2.

Decompressing Text

The key for the text will be stored in a text file. (This is explained in further detail on page 44) It would hold each word used.

SEE SEA SHELLS SEA SHELLS SEE SHELLS SEE SEA SEE SEA SEE SEE SEA SHELLS

would only have SEE SEA “ “ and SHELLS stored in the dictionary text file as they are the only unique words used

I would have to read from a file and then store each word in a list. Each index in the array will correspond to a word.

This will need 3 FOR loops.

* One loop will go through each line of text
* The 2nd line will go through each character of text. I will need to determine if the phrase being added is a number or the actual word itself. To do this, I will check if the current character is a non-numeric character and not space. I will then use a WHILE loop and then keep adding characters until the character space appears in the line of compressed text. If the character is a digit, I will also keep adding characters until there is a space and then use the 3rd FOR loop which is explained below.
* The 3rd line will re-assign the original word and therefore replace the compressed text.

Here is some pseudocode below.

A screenshot of a computer program

Description automatically generated

# Documented Design – Files

## Files- Pseudocode and Prototyping

File handling is quite an important aspect in this project making sure that my program cannot be broken from user input.

My program should also allow users to create new file regardless of whether it is in a new folder or in a current folder. Here is how I would create a new folder using **file.create** and **directory.createdirectory**. Here is the link on how I did this below. <https://learn.microsoft.com/en-us/dotnet/visual-basic/developing-apps/programming/drives-directories-files/how-to-create-a-file>

A screenshot of a computer code

Description automatically generated

I also must be able to differentiate between a binary file and a Huffman file. My pseudocode above will check to see if the last 4 characters in a file path are .txt, if they are not then it is assumed to be a binary file.

When the user inputs a file path, I need to make sure that the file exists. I also need to check if a file “will exist” if they choose to save the compressed file. I will be using **try catch exception** in vb.net which checks to see if the file path is valid. This should work for both binary files and text files. For all sections of the program (Huffman coding, Dictionary data compression and the statistics), I need to make sure that the file is not empty. When I load the text from the file, I will count the number of lines of text in the file. If the count of lines in the text is equal to 0 then an appropriate response will be displayed to the user.

A screenshot of a computer

AI-generated content may be incorrect.

In order to open a file, I have decided to use FileOpen and FileClose methods.

When inputting an empty file, the user is redirected to the main menu again.

Sub OpenFile()

* This code would keep asking for a new file path to be entered until a valid one is inputted.
* Once a valid file path is inputted, I decided to display the contents of the file to the user

Console.WriteLine("Input filepath")

Dim filepath As String = Console.ReadLine

Try

FileOpen(1, filepath, OpenMode.Input)

While Not EOF(1)

Console.WriteLine(LineInput(1))

End While

FileClose(1)

Catch ex As Exception

OpenFile()

End Try

End Sub

A computer screen with white text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

When Huffman coding or dictionary data compression takes place, I will have to create two files to store the sufficient data to decompress the text. In the case of Huffman coding, it would be the tree and for dictionary data compression it would be the dictionary of all the words used.

Unfortunately, I could not use binary serialisation when storing the tree/dictionary. A screenshot of a computer

AI-generated content may be incorrect.

This would mean that I could not store the tree/dictionary into a binary file. If this was possible, the overall percentage file size reduction would have been increased. As the data cannot be stored in a binary file, I will have to store it in a text file instead.

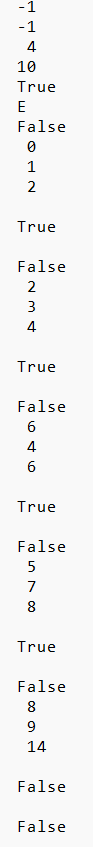
Then I will assign a folder for both files to appear in being C:/CourseworkTesting

## Huffman Coding files



Here is an example of data written to a binary file. This represents SEE SEA SHELLS. As you can see this is a much more compact version of the text.

A close up of a text

Description automatically generated

These two images hold the information needed to decompress the binary data shown above back into the text SEE SEA SHELLS.

The file path for the tree would be C:/CourseworkTesting\TreeSave.txt

What the file contains

* The file-path of the binary file used to store the compressed text; in this case it is C:\NEA/Huffman. When testing the project I will be using the **C:\CourseworkTesting/**  folder.
* The length of each compressed line in binary. In this case, the length of the encoded line in binary is 34 bits. (In this case there was only a single line so having multiple lines would mean that I would also need to save the length of the other lines as well)
* An empty string is added to file so then I can distinguish when to reconstruct the contents of the Huffman tree
* The value of the LP and the RP (in that order)
* The total frequency of characters for a particular node
* The binary code which represents a particular character for example 001 would represent H
* Whether the binary code has been already used when constructing the tree. (for all the nodes in the tree, this should be set to true)
* The character which the binary code represents
* Whether if the binary code has been outputted to the user when they want to view the binary codes. For all the nodes which contains characters this would be set to true otherwise it would be set to false.

I also must be able to write to binary files in vb.net. To do this, I will be using **filestream** and **binary writer.**

Here is how I would write to a binary file - https://www.tutorialspoint.com/vb.net/vb.net\_binary\_files.htm#:~:text=The%20BinaryWriter%20class%20is%20used,methods%20of%20the%20BinaryWriter%20class.

## Dictionary Data Compression Files

This is the compressed text of SEE SEA SHELLS stored in a text file. Each encoded word (the number) is separated by a space.

A number with a number on it

Description automatically generated with medium confidence

A white background with black text

Description automatically generated

This image holds the file path of where the compressed text. (C:NEA/Dictionary.txt)

The dictionary also stores the key allowing the text to be decompressed

The file path which stores the dictionary will be called C:/CourseworkTesting\Dictionary.txt

The first line read after the file path is “space” which has a compressed text of 0, SHELLS is 1, SEE is 2 and SEA is 3

# Documented Design – Statistics

The statistics class will work by having objects of the Huffman Coding class and the Dictionary Data Compression passed through as parameters in one of its subroutines. This will allow me to make use of both algorithms subroutines without needing to inherit from them directly.

Here are the things which should be shown when the user wants to view the statistics

* The file size of the original text (in bytes)
* The size of the binary file with the size of the tree used to store the text (in bytes)
* The time taken for the Huffman coding algorithm to compress the text (In bytes)
* The size of the text file when the text has been compressed using the dictionary data compression algorithm. It will also add the size of the dictionary too
* The time taken for the Dictionary Data Compression algorithm to compress the text
* The percentage file size reduction for Huffman coding
* The percentage file size reduction for Dictionary Data Compression

Here is the link to the website to extract the size of a given file: <https://learn.microsoft.com/en-us/dotnet/api/system.io.fileinfo.length?view=net-9.0>

This would involve using the data type “long” to calculate the size of the file. I will be using this to calculate the size of the original file. I will not be using it to calculate the size of the compressed text and the compressed texts corresponding dictionary/tree as I would then have to create the file for the compressed text and the dictionary/tree. **These will be rough estimates**

To calculate the size of the compressed text using Dictionary Data Compression

* Add the length of each item in the key (each character will represent one byte)
* Add the length of the file path used to store the compressed text (in this case I would just use the original file path used as a rough estimate)
* The length of each compressed line of text. (Each character in a line will represent one byte)

To calculate the size of the compressed text using Huffman Coding

* Count the number of bytes used to store the compressed text. When I convert each of the binary lines into 8-bit bytes, I will just keep a counter of the number of bytes formed. This will be returned using a function in the Huffman Coding class
* Convert each section of the tree into strings and then adding the length of each of these sections to the number of bytes.
* Also add the length of each compressed line in binary bits. These should be stored as strings but then have the length multiplied by 4 because integers are stored as 32 bits in vb.net whereas a single character in a string would be 8 bits.

To calculate the time taken to compress the text, I will be using a built-in stopwatch in vb.net.

Here is the formula that I will use to calculate the percentage file size reduction. I will be using this formula for both the Huffman coding file and the dictionary data compression file. The greater the number produced using this formula, the greater the file size reduction for a piece of text.

**(1 – (size of the compressed text/ size of original file)) \* 100**

However, to do this I would need to convert the file size to an integer using **Cint** as it is currently using the **long** datatype.

# Documented Design – Classes and methods

The classes I have used are called File Handling, Huffman Coding, Dictionary Data Compression and Statistics. The Huffman Coding, Dictionary data compression and statistics class will all inherit the file handling class. I decided to use classes as it organises where my subroutines are located for each aspect of my project.

## Structures

I have used two structures; one called characters, and another one called node.

As I have used structures, I must set a maximum size of each array. I decided to set it at a maximum size of 1001 items. This means a maximum number of 1001 nodes can be stored in the tree and maximum number of 1001 unique characters can be stored.

Characters structure

|  |  |  |
| --- | --- | --- |
| **Name** | **Data type** | **Purpose** |
| CharacterFrequency | Integer | Holds the frequency of all the characters used in the text |
| Character | String | Holds a unique character |
| CodeProduced | Boolean | Checks to see if the binary equivalent of a character has been produced or not |

This structure is responsible for holding the characters used in the text for the Huffman coding algorithm. I have a Boolean variable to check if the code for the character has already been displayed to the user.

Node structure

|  |  |  |
| --- | --- | --- |
| **Name** | **Data Type** | **Purpose** |
| LP | Integer | Holds the left pointer (child node location) |
| Total | Integer | Holds the sum of characters contained in that node. |
| RP | Integer | Holds the right pointer (child node location) |
| IsProcessed | Boolean | Checks to see if a node has already been used as either a RP or a LP in the tree when constructing it. |
| Character | String | Holds the character which the node represents |
| BinaryCode | String | Holds the binary code which represents the character |

This structure is responsible for holding the Huffman tree and it used throughout the process of the Huffman coding algorithm.

## File Handling Class

The purpose of this class is to have all the code which involves handling files which are used in all the program’s aspects. This will include handling incorrect files and making sure that a file path is inputted until a correct one is

**There are no global variables used in this class**

**This class does make use of the structures**

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Purpose** | **Parameters** |
| DetermineTextFile  Function which returns a Boolean variable | Checks to see if a file path is a text file or a binary file. This is accomplished by checking to see if the last 4 characters of a file path is .txt. Returns true if it is a text file otherwise false  This function is recursive as if the user does not input a file path with a length > than 4 then call the function again | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Filename  (byref) | String | Holds file path that user has inputted | |
| LoadingText  Function which returns a Boolean variable | Loads all the text from a text file. Keeps asking the user for a text file path until a valid one is inputted. Returns true if file path exists otherwise returns false. | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Filename | String | Holds file path that user has inputted | | Lines (byref) | List of string | Holds all the lines of text from the file | |
| CreateDirectoryAndFile  Subroutine | Will create a new directory and/ or a text file depending on the file path the user has inputted. Subroutine is recursive as if the file still doesn’t exist then it is called again. | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Filepath  (byref) | String | Holds the file path that user has inputted | | Huffman | Bool-ean | A checker if the user is inputting a binary file or a text file. True means binary file and false means text file. | |
| Resavingtoatextfile  Subroutine | Will store the uncompressed text into a new text file. This is used when they are decompressing the text. I will use this during both Huffman coding and dictionary data compression algorithms. | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Textfile  path | String | Holds the file path that user has inputted | | Original  Lines | String | Holds the original lines of text | |

## Dictionary Data Compression Class

The purpose of this class is to have all of the subroutines for dictionary data compression which makes it easier to manage any of the features.

**There are no global variables used in this class**

**This class does not make use of structures**

**This class inherits the file handling class**

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Purpose** | **Parameters** |
| GettingAllUniqueWords  Subroutine | Gets all the unique words stored in the text file. This uses 2 FOR Loops to navigate through each line of text  Calls CheckingWordExists | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Word  Bank  (byref) | List of  String | Holds all of the unique words in the text | | Word  Frequen  cies (byref) | List of  Integ  er | Holds frequen  cy of all words in the text | | Lines | List of string | Holds all of the lines of text from file | |
| CompressText  Subroutine | Compresses the text by assigning each unique word to the number assigned in the array. This uses 2 FOR loops  Calls WordChecker | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Encode  d Lines  (byref) | List of  string | Holds the compres  sed lines of text | | Lines | List of string | Holds the text stored in the file | | Word  Bank | List of string | Holds all of the unique words in the text | |
| CheckingWordExists  Subroutine | Checks to see if a word has already been stored in the dictionary. If it hasn’t then add it to the dictionary. | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Previou  slystore  dword | Bool  ean | Used to see if a word already exists | | Word  Bank  (byref) | List of string | Holds all of the unique words  in the text | | Word  Freque  ncies  (byref) | List of integer | Holds the frequen  cies of all of the words in the text | | Lines | List of string | Holds all of the lines of text from file | | Word | String | Holds the word which is being checke against the dictiona  ry | |
| DictionaryDataCompresion  Intotextfile  Subroutine | Will store the compressed text into a file of the users choosing. Calls CreateDirectoryAndFile, Compress text and StoringKey subroutines | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Lines | List of string | Holds all of the lines of text from file | | Word  Bank  (byref) | List of string | Holds all of the unique words  in the text | | Filepath | String | Holds file path that user has inputted | | Encode  dlines  (byref) | List of string | Holds the compre  ssed lines of text | |
| ViewText  Subroutine | Displays either the compressed or decompressed lines of text to the user | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Output  Lines | List of string | Used to display either the compres  sed text or the decomp  ressed text | |
| StoringKey  Subroutine | Stores the key into a text file  Calls CreateDirectoryAndFile subroutine | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Word  Bank | List of string | Holds all of the unique words  in the text | | User  Path | String | Holds file path that user has inputted | |
| WordChecker  Subroutine | Adds the number equivalent of the word to a string called line | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Word | String | Holds the word which is being  Checked against the word bank | | Word bank | List of string | The list of words which the word is being compare  d to | | Line (byref) | String | Holds the compres  sed line of text | |
| DictionaryDataCompress  ionKey  subroutine | Displays the key for the text. | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Word  Bank | List of string | Holds all of the unique words in the text | |
| FrequencyDistribution  Subroutine | Displays the frequency of each word in the piece of text | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Word  Bank | List of string | Holds all of the unique words in the text | | Word  Freque  cies | List of  Integer | Holds the frequen  cies of all of the words in the text | |
| DecompressText  Subroutine | Turns the compressed text back into the decompressed text | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Word  Bank | List of string | Holds all of the unique words in the text | | Encode  dlines | List of string | Holds the compre  ssed lines of text | | Original  Lines | List of string | Used to hold the decomp  ressed text | |
| DictionaryDataCompress  ionUserInterface  Subroutine | User interface which gives user the option to view the key, store the compressed text into a text file, encode and decode the text, view the frequency distribution of words, return back to main menu | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Diction  ary | Dictio  nary  Data  Compr  ession | Used to call the user interface | | Done once  (byref) | Boolea  n | Makes sure that the GetAll  Unique  Words  Subrouti  ne is only called once | | Lines  (byref) | List of string | Holds all of the lines of text from the text file | | Filepath | String | File path of text that user inputted | |
| NumberOfBytes  Function which returns an integer value | Calculates the number of bytes the compressed text uses | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Encode  d lines | List of string | Holds the compre  ssed lines of text | |
| Partion  Function which returns an integer value | Calculates the pivot point. If an item is in the wrong order, they are swapped with each other | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Word  Freque  cies – byref | List of  Integer | Holds the frequen  cies of all of the words | | LP | Integer | Start of the word  Frequen  cies array | | RP | Integer | End of the word  Frequen  cies array | | Word  Bank – byref | List of string | Holds all of the unique words in the text | |
| Quicksort  Subroutine | Used to sort the list from highest to lowest frequency. Calls the partion function to determine the pivot point. Quicksort is also done recursively calling the quicksort subroutine | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Word  Freque  ncies – byref | List of integer | Holds the frequen  cies of all of the words in the text | | LP | Integer | Start of the word  Frequen  cies array | | RP | Integer | End of the word  Frequen  cies array | | Word  Bank – byref | List of string | Holds all of the unique words in the text | |

## Statistics Class

The purpose of this class is to handle the statistics part of the project by using encapsulation to access the Huffman coding subroutines and the dictionary data compression subroutines. This will involve zero creation of files to calculate the file size reduction

**This class has no global variables used in this class**

**This class does not inherit any classes**

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Purpose** | **Parameters** |
| AllSubroutines  Subroutine | Holds all of the subroutines used to compress the piece of text for both algorithms. Displays the initial file size and the compressed file size for both algorithms as well as the time taken for both algorithms to compress the text | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Huffman | Huffman  Coding | Used to access Huffman Coding subrout  ines | | Dictionary | Dictionary  Data  Compres  sion | Used to access  Dictiona  ry data  compres  sion sub  routines | | Original  Filepath | String | Filepath  That user  inputted | | Lines  Byref | List of string | Holds all of the lines of text in the original file | |
| Percentages  Subroutine | Displays the percentage file size reduction of both Huffman coding and dictionary data compression | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Dictionary  SizeIn  Bytes | Integer | Holds the size of compres  sed file using dictionary | | Original  Filepath  Size | Long | Size of original file path | | Huffman  SizeIn  Bytes | Integer | Size of compres  sed file using Huffman coding | | Treesize  Inbytes | Long | Size of tree | |
| StatisticsUI  Subroutine | User interface which allows user to return back to the main menu or continue through to the statistics page | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Huffman | Huffman  Coding | Used to access Huffman  Coding  Sub  Routines | | Dictionary | Dictionary  Data  Compress  ion | Used to access Diction  ary sub  routines | | Original  Filepath | String | File path that user inputted | | Lines | List of string | Holds the lines of text from the file | |
| Calculating  TreeSize  Function which returns an integer value | Calculates the size of the tree manually instead of needing to create a file to then calculate the file size. | |  |  |  | | --- | --- | --- | | **Name** | **Data**  **Type** | **Purpose** | | Tree | Node | contains the entire Huffman Tree | | TreeSize | Integer | Used to determine the number of FOR loops as I would need to loop through the entire tree to calculate the tree size | | EachLine  InBinary | List of string | This length of each line will also be added as it would have also been stored alongside the tree when compressing the text. | |

## Huffman Coding Class

The purpose of this class is to have all the Huffman coding subroutines which makes it easy to access and understand each subroutine using in constructing the tree, compressing and decompressing the text.

**This class has no global variables**

**This class inherits the File Handling class**

**This class makes use of the Characters structure and the node structure**

The characters structure is used under the name **Unique** and the node structure is used under the name **Tree**

|  |  |  |
| --- | --- | --- |
| **Methods Name** | **Purpose** | **Parameters** |
| SetUpArrays  Subroutine | Sets up the tree and unique arrays | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Tree()  byref | Node | Holds the tree | | Unique()  Byref | Chara  cters | Holds the unique characters | | Treesize | Integer | Holds size of tree | | Unique  Size | Integer | Holds size of unique | |
| CollectingAllUnique  Characters  Subroutine | Retrieves all of the unique characters and collects the frequency of each character | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Unique()  Byref | Chara  cters | Holds all of the unique characters | | Unique  Size  Byref | Integer | Holds the size of unique | | Lines | List of string | Holds the unique lines of text from the file | |
| SortingFrequency  Subroutine | Adds frequencies of characters to priority queue as well as sorting the queue  Calls the quick sort subroutine | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Unique() | Chara  cters | Holds all of the unique characters | | Unique  Size | Integer | Holds the size of unique | | Priority  Queue  (byref) | List of integer | Holds the frequencies of all of the nodes | |
| Partion  Function which returns an integer value | Determines the partion index.  If pivot point not met, if the two values stored in the array are in the wrong order then they are swapped. | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Priority  Queue  (byref) | List of integer | Holds the frequencies of all of the nodes | | LP | Integer | The minimum index value to be searched | | RP | Integer | The maximum index value to be searched | |
| Quicksort  Subroutine | Calls Partion function and recursively calls quicksort to sort the priority queue | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Priority  Queue  (byref) | List of integer | Holds the frequencies of all of the nodes | | LP | Integer | The minimum index value to be searched | | RP | Integer | The maximum index value to be searched | |
| CreatingRootNodes  Subroutine | Creates the root nodes of the tree setting the LP and RP of all of the items in the priority queue to be -1 | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Priority  Queue | List of integer | Holds the total of each node in the queue | | Tree()  Byref | Node | Holds all of the totals of nodes which are yet to be added to the tree | | Treesize  Byref | Integer | Holds the size of the tree | |
| HuffmanTree  Subroutine calls it self so it is recursive | Creates the Huffman tree. Recursive call is made until the size of the priority queue is equal to 1 as no more nodes need merging together.  Calls the RemoveHighest  Priority function twice when getting the items in the queue with the smallest frequencies | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Priority  Queue  (byref) | List of integer | Holds all of the totals of nodes which are yet to be added to the tree | | Tree()  Byref | Node | Holds the all of the nodes in the tree | | TreeSize  Byref | Integer | Holds the size of the tree | |
| RemoveHighestPriority  Function which returns an integer value | Calls the quicksort subroutine and returns the value of the first item in the priority queue which is the smallest value | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Priority  Queue  Byref | List of integer | Holds the total of nodes yet to be added to the tree | |
| ResettingCodeProduced  Subroutine | Resets the Huffman tree as well as the unique() variable  Value of Treesize and UniqueSize is set to 0 | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Unique()  Byref | Chara  cters | Holds all of the unique characters | | Unique  Size  Byref | Integer | Holds the size of unique | | Tree()  Byref | Node | Holds the all of the nodes in the tree | | TreeSize  Byref | Integer | Holds the size of the tree | |
| FrequencyDistribution  Subroutine | Displays the frequency of each of the characters in the text | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Unique() | Chara  cters | Holds all of the unique characters | | Unique  Size | Integer | Holds the size of unique | |
| PreorderTraversal  Encoding  Subroutine | Assigns the binary codes to each of the characters  Subroutine is called recursively when traversing the left subtree and the right subtree | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Unique() | Chara  cters | Holds all of the unique characters | | Unique  Size | Integer | Holds the size of unique | | Node Index | Integer | Index of node which is currently being looked at | | Current  Code | String | Holds the current binary code obtained when traversing the tree | | Tree()  Byref | Node | Holds the all of the nodes in the tree | | TreeSize | Integer | Holds the size of the tree | | Statistic | Boolean | If value is set to false then display the binary codes obtained otherwise don’t | |
| Preorder Traversal  Decoding  Subroutine | Converts the binary codes back into their original characters  Subroutine is called recursively when traversing the left subtree and the right subtree | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Node Index | Integer | Index of node which is currently being looked at | | Current  Code | String | Holds the current binary code obtained when traversing the tree | | Tree()  Byref | Node | Holds the all of the nodes in the tree | | EachLine  InBinary  Byref | List of string | Holds all the compres  sed lines of text in binary | | BinaryLine  Byref | List of string | Holds each individual bit in an individual index. Holds one lines worth of bits | | Original  Lines  Byref | List of string | Holds the decomp  ressed text | | FullLine | String | Holds an individual line of decompre  ssed text | |
| RetreiveBinaryLines  Subroutine | Retrieves the binary equivalent of each line of text | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Filename | String | Holds the file path of the binary file | | Each  Linein  Binary  Byref | List of string | Holds all the compres  sed lines of text in binary | | Each  Binary  Line  Length  (byref) | List of integer | Holds the length of each binary line | |
| ConvertFromBinaryTo  Text  Subroutine | Converts each line in binary back into each line of the original text by forming 2 lists which holds all of the bits.  Calls the preorder traversal decoding subroutine for each line in the text. This is to ensure that the correct number of characters are added to the decompressed text. | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Tree()  Byref | Node | Holds the all of the nodes in the tree | | TreeSize | Integer | Holds the size of the tree | | Each  Linein  Binary  Byref | List of string | Holds all the compres  sed lines of text in binary | | BinaryLine | List of string | Holds each individual bit in an individual index. Holds one lines worth of bits | | Original  Lines  Byref | List of string | Holds the decomp  ressed text | | FullLine | String | Holds an individual line of decompre  ssed text | |
| ConvertAllTextToBinary  Subroutine | Converts all text to binary by using 3 for loops | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Tree()  Byref | Node | Holds the all of the nodes in the tree | | TreeSize | Integer | Holds the size of the tree | | Each  Linein  Binary  Byref | List of string | Holds all the compres  sed lines of text in binary | | Lines | List of string | Holds the lines of text from the text file | | Statistics | Boolean | If set to true then output the binary line otherwise don’t | |
| SplitIntoBytes  Function which returns an integer value | Returns the number of bytes the compressed text contains. Number of bytes is only returned when statistics is set to true otherwise the code is directed to the WritingToBinaryFile subroutine | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Tree()  Byref | Node | Holds the all of the nodes in the tree | | TreeSize | Integer | Holds the size of the tree | | Each  Linein  Binary  Byref | List of string | Holds all the compres  sed lines of text in binary | | Statistics | Boolean | If set to true then call WritingTo  Binary  File  Sub otherwise  Don’t. | |
| WritingToBinaryFile  Subroutine | Writes the binary data to a binary file. Subroutine calls itself if an invalid binary file path is inputted. Once Binary data is stored, SaveHuffmanTree subroutine is called | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Tree() | Node | Holds the all of the nodes in the tree | | TreeSize | Integer | Holds the size of the tree | | Each  Linein  Binary | List of string | Holds all the compres  sed lines of text in binary | | Binary  Bytes | List of string | Holds all the strings of binary bytes | |
| SaveHuffmanTree  Subroutine | Stores the current state of the Huffman tree into a text file along with the length of each line in binary so each line can be retrieved. This uses FileOpen() and FileClose() | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Tree() | Node | Holds the all of the nodes in the tree | | TreeSize | Integer | Holds the size of the tree | | Eachline  Inbinary | List of string | Holds each compres  sed  Line of binary | | Filename | String | File path of the binary file | |
| RetrieveHuffmanTree  Function which returns a Boolean variable | Returns true if the binary file inputted correlates to the information stored in the Huffman tree otherwise returns false. If false is returned user is sent back to the main menu. If true then Huffman tree is retrieved. An example of a value being returned false would be if a valid binary file has been inputted but the tree which is currently being stored does not correspond to that particular binary file. | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Tree()  (byref) | Node | Holds the all of the nodes in the tree | | TreeSize  (byref) | Integer | Holds the size of the tree | | Eachline  Inbinary  (byref) | List of string | Holds each compres  sed  Line of binary | | Filename  (byref) | String | File path of the binary file | | Each  Binary  Line  Length  (byref) | List of integer | Holds the length of each of the binary lines stored | |
| ResetDecoding  Subroutine | Clears all of the compressed text and resets the size of the tree | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Eachline  Inbinary  (byref) | List of string | Holds each compres  sed line of text | | Eachline  Inbinary  Length  (byref) | List of integer | Holds the length of each line in binary | | Treesize  (byref) | Integer | Any item in the tree will be over  written | |
| HuffmanCoding  UserInterface  Subroutine | User interface which allows users to compress text files and decompress binary files, view the frequency distribution of characters, view the key used to decode the text, store compressed text into a binary file and decompressed text into a text file | |  |  |  | | --- | --- | --- | | **Name** | **Data Type** | **Purpose** | | Huffman | Huffman  Coding | Used to access the user interface | | Lines  (byref) | List of string | Holds all of the lines of text from the file | | Filename | String | The file path that the user has used | |

# Technical solution

Imports System.ComponentModel.Design

Imports System.IO

Imports System.Net

Imports System.Net.Security

Imports System.Runtime.InteropServices

Imports System.Text

Imports System.Xml

Module Program

'C:\CourseworkTesting/

'C:\NEA/Testing.txt

Structure characters

Dim CharacterFrequency As Integer 'the frequency of the character

Dim Character As String

Dim CodeProduced As Boolean 'determines whether the code has been displayed or not

End Structure

Structure Node

Dim LP As Integer 'Left Pointer (child node with the smaller total)

Dim RP As Integer 'Right Pointer (child node with the larger total)

Dim total As Integer 'Sum of leaf nodes // the value which the parent node will contain

Dim IsProcessed As Boolean 'Checks to see if the node has already been viewed in the tree during construction

Dim character As String

Dim BinaryCode As String 'The binary code which is produced to the screen

Dim CharacterProduced As Boolean

End Structure

Sub Main(args As String())

Dim DictionaryDoneOnce As Boolean = False 'just to prevent the dictionary being built multiple times

Dim Filehandler As New FileHandling

Dim choice As String = ""

Console.WriteLine("Welcome")

Console.WriteLine("To access the menu type the number of the task you want to complete and then press enter")

Console.WriteLine("All Files must have a valid file path otherwise you will be instructed to input another one")

Console.WriteLine("Text files will end in .txt whereas binary files will not")

Console.WriteLine()

Console.WriteLine()

Do

Dim lines As New List(Of String)

Dim huffman As New HuffmanCoding

Dim Dictionary As New DictionaryDataCompression

Dim Statistics As New Statistics

Dim statistical As Boolean = False 'prevents the UI being presented when using statistics

choice = ""

While choice <> "1" And choice <> "2" And choice <> "3" And choice <> "4"

'displays all of the valid options and requests the user for an input

Console.WriteLine("1:Huffman Coding Menu")

Console.WriteLine("2:Dictionary Data Compression Menu")

Console.WriteLine("3:Statistics")

Console.WriteLine("4:Exit")

choice = Console.ReadLine

Console.WriteLine()

Console.WriteLine()

End While

If choice <> "4" Then 'if they have not exited the menu then require the user to input a filepath

Dim filename As String = ""

Do

Console.WriteLine("Input filepath")

filename = Console.ReadLine

If Not (choice = "1") Then

'ensures that for dictionary data compression and the statistics page that a valid text file path is inputted and not a binary file

While Filehandler.DetermineTextFile(filename) = False

Console.WriteLine("Input filepath")

filename = Console.ReadLine

End While

Else

'checks to see if a binary file is a valid file path

Dim ValidHuffman As Boolean = False

Do

Try

FileOpen(1, filename, OpenMode.Binary.Input)

FileClose(1)

ValidHuffman = True

Catch ex As Exception

Console.WriteLine("Input filepath")

filename = Console.ReadLine

End Try

Loop Until ValidHuffman = True

End If

'requests for userinput until a valid file path is inputted

Loop Until Filehandler.LoadingText(filename, lines) = True And filename.Length > 4

Console.WriteLine()

If lines.Count > 0 Then 'checks to see if the file is empty

'if the file is not empty then for the appropriate option the user will be directed to that user interface

If choice = "1" Then

huffman.HuffmanCodingUserInterface(huffman, lines, filename)

ElseIf choice = "2" Then

Dictionary.DictionaryDataCompressionUserInterface(Dictionary, DictionaryDoneOnce, lines, filename)

DictionaryDoneOnce = False

ElseIf choice = "3" Then

Statistics.StatisticsUI(huffman, Dictionary, filename, lines)

End If

Else

Console.WriteLine("Empty file")

Console.WriteLine()

End If

End If

Loop Until choice = "4"

Console.WriteLine("Bye")

End Sub

Public Class FileHandling

Function DetermineTextFile(ByRef filename As String) As Boolean

Dim ValidTxt As String = ".txt"

If filename.Length <= 4 Then 'if the filename has an invalid length

Console.WriteLine("Input file path")

filename = Console.ReadLine()

Return DetermineTextFile(filename)

Else

Dim Ans As String = ""

For x = filename.Length - 1 To 0 Step -1

If Ans.Length <> 4 Then 'collects the last 4 characters from the text file to see if it .txt

Ans = filename.Substring(x, 1) & Ans

End If

Next

Return Ans = ValidTxt ' Return result directly

End If

End Function

Function LoadingText(filename As String, ByRef lines As List(Of String)) As Boolean

Try

FileOpen(1, filename, OpenMode.Input)

' Read each line from the file and add it to the lines array

Dim line As String

Dim i As Integer = 0

Do While Not EOF(1)

line = LineInput(1)

'if a line is an empty string then just add a space instead

If line = "" Then

lines.Add(" ")

Else

lines.Add(line)

End If

Loop

FileClose(1)

Catch ex As Exception

Return False

End Try

Return True

End Function

Sub CreateDirectoryAndFile(ByRef filepath As String, Huffman As Boolean)

If Huffman = True Then

While DetermineTextFile(filepath) = True

Console.WriteLine("Input file path")

filepath = Console.ReadLine

End While

ElseIf Huffman = False Then

While DetermineTextFile(filepath) = False

Console.WriteLine("Input file path")

filepath = Console.ReadLine

End While

End If

Dim Direct As String = ""

Dim x As Integer = 0

While filepath.Substring(x, 1) <> "/"

Direct = Direct & filepath.Substring(x, 1)

x = x + 1

End While

'the directory that the user has inputted has been stored in the 'Direct' variable

If Directory.Exists(Direct) = False Then 'if the directory does not exist

Directory.CreateDirectory(Direct)

End If

If File.Exists(filepath) = False Then

Using fs As FileStream = File.Create(filepath)

End Using

End If

Try

If DetermineTextFile(filepath) = False Then

FileOpen(2, filepath, OpenMode.Binary)

Else

FileOpen(2, filepath, OpenMode.Input)

End If

FileClose(2)

Catch ex As Exception

CreateDirectoryAndFile("", Huffman) 'if the file does not exist then recall the subroutine

End Try

End Sub

Sub resavingToATextFile(textfilepath As String, originallines As List(Of String))

'allows the original lines formed from the decoded text into a text file

CreateDirectoryAndFile(textfilepath, False) 'if the directory is not present or the file isnt then create a new one

Console.WriteLine()

FileOpen(4, textfilepath, OpenMode.Output)

For h = 0 To originallines.Count - 1

PrintLine(4, originallines(h))

Next

FileClose(4)

End Sub

End Class

Public Class HuffmanCoding

Inherits FileHandling

Function partion(priorityqueue As List(Of Integer), LP As Integer, RP As Integer) As Integer

Dim pivot As Integer = priorityqueue(LP)

Dim i As Integer = LP - 1

Dim j As Integer = RP + 1

While i < j 'repeat this code until it reaches the pivot point

Do

i = i + 1

Loop Until priorityqueue(i) >= pivot

Do

j = j - 1

Loop Until priorityqueue(j) <= pivot

If i >= j Then

Return j

Else

'swaps the elements around

Dim placeholder As Integer = priorityqueue(i)

priorityqueue(i) = priorityqueue(j)

priorityqueue(j) = placeholder

End If

End While

Return j

End Function

Sub quicksort(ByRef priorityqueue As List(Of Integer), LP As Integer, RP As Integer)

If LP < RP Then

Dim PartionIndex As Integer = partion(priorityqueue, LP, RP)

quicksort(priorityqueue, LP, PartionIndex)

quicksort(priorityqueue, PartionIndex + 1, RP)

End If

End Sub

Sub SetUpArrays(ByRef tree() As Node, ByRef unique() As characters, treesize As Integer, uniquesize As Integer)

tree(treesize) = New Node

unique(uniquesize) = New characters

End Sub

Public Sub CollectingUniqueCharacters(ByRef unique() As characters, ByRef uniqueSize As Integer, lines As List(Of String))

Dim found As Boolean 'variable to check if character exists or not

Dim character As String = ""

For i = 0 To lines.Count - 1 'looping through each line

For x = 0 To lines(i).Length - 1 'looping through each character

character = lines(i).Substring(x, 1)

found = False

For k = 0 To uniqueSize - 1 'loops through the list of known characters

If unique(k).Character = character Then

unique(k).CharacterFrequency = unique(k).CharacterFrequency + 1 'if found increase the counter by 1

found = True

End If

Next

If found = False Then 'if the character has not been found

'set up the new character

unique(uniqueSize).Character = character

unique(uniqueSize).CharacterFrequency = 1

uniqueSize = uniqueSize + 1

End If

Next

Next

End Sub

Public Sub SortingOutFrequency(unique() As characters, uniquesize As Integer, ByRef priorityqueue As List(Of Integer))

For x = 0 To uniquesize - 1

priorityqueue.Add(unique(x).CharacterFrequency)

Next

quicksort(priorityqueue, 0, priorityqueue.Count - 1)

End Sub

Public Sub CreatingRootNodes(priorityqueue As List(Of Integer), ByRef tree() As Node, ByRef treeSize As Integer)

'for all of the characters, they are rootnodes so the LP and RP are -1 and the total is their frequency

For i = 0 To priorityqueue.Count - 1

Dim newNode As New Node()

newNode.RP = -1

newNode.LP = -1

newNode.total = priorityqueue(i)

tree(treeSize) = newNode

treeSize = treeSize + 1 'increase the size of the array by one

Next

End Sub

Public Function RemoveHighestPriority(ByRef priorityqueue As List(Of Integer)) As Integer

quicksort(priorityqueue, 0, priorityqueue.Count - 1) 'sorts the list

Dim minvalue As Integer = priorityqueue(0)

priorityqueue.RemoveAt(0)

'removes the smallest item in the priority queue

Return minvalue

End Function

Public Sub ResettingCodeProduced(ByRef unique() As characters, ByRef uniquesize As Integer, ByRef tree() As Node, ByRef treesize As Integer)

'basically without this, if you tried to display the huffman codes again, it would just display empty strings

For t = 0 To uniquesize - 1

unique(t).CodeProduced = False

Next

For i = 0 To treesize - 1

tree(i).CharacterProduced = False

tree(i).BinaryCode = ""

Next

treesize = 0

uniquesize = 0

End Sub

Public Sub FrequencyDistribution(unique() As characters, uniquesize As Integer)

For x = 0 To uniquesize - 1

Console.WriteLine(unique(x).Character & " " & unique(x).CharacterFrequency)

Next

End Sub

Public Sub HuffmanTree(ByRef priorityqueue As List(Of Integer), ByRef tree() As Node, ByRef treeSize As Integer)

Dim node1Freq As Integer = RemoveHighestPriority(priorityqueue) 'gets smallest element

Dim node2Freq As Integer = RemoveHighestPriority(priorityqueue) 'gets second smallest element

Dim node1Found As Boolean = False 'ensures that only two nodes are taken

Dim node2found As Boolean = False 'ensures that only two nodes are taken

Dim node1Index As Integer = -1

Dim node2Index As Integer = -1

For t = 0 To treeSize - 1

If tree(t).total = node1Freq And tree(t).IsProcessed = False And node1Found = False Then

node1Index = t

tree(t).IsProcessed = True 'makes sure that specific node is not looked at again

node1Found = True

End If

If tree(t).total = node2Freq And tree(t).IsProcessed = False And node2found = False Then

node2Index = t

tree(t).IsProcessed = True 'makes sure that specific node is not looked at again

node2found = True

End If

Next

'create a new node

Dim newNode As New Node()

'LP is the smaller node out of the two

newNode.LP = node1Index

newNode.RP = node2Index

'NewNode total is the sum of the two nodes

newNode.total = node1Freq + node2Freq

tree(treeSize) = newNode

treeSize = treeSize + 1 'increase the size of the tree by one

priorityqueue.Add(newNode.total) 'add the new node to the priority queue

If priorityqueue.Count <> 1 Then

HuffmanTree(priorityqueue, tree, treeSize)

End If

End Sub

Public Sub PreorderTraversalEncoding(nodeIndex As Integer, currentCode As String, ByRef tree() As Node, ByRef unique() As characters, uniqueSize As Integer, statistics As Boolean)

Dim NodeFound As Boolean = False 'ensures that only one character is being produced as characters with same frequency would also be outputted

If tree(nodeIndex).LP = -1 And tree(nodeIndex).RP = -1 Then

For x = 0 To uniqueSize - 1

If unique(x).CharacterFrequency = tree(nodeIndex).total And unique(x).CodeProduced = False And NodeFound = False Then

'if it is a chharacter then output it

unique(x).CodeProduced = True 'prevents a characters binary code being produced multiple times

tree(nodeIndex).character = unique(x).Character

tree(nodeIndex).BinaryCode = currentCode

NodeFound = True

If statistics = False Then

'when this code is running through the statistics, the binary codes do not need to be displayed

Console.WriteLine(unique(x).Character & " : Binary code = " & currentCode)

End If

End If

Next

Else

If tree(nodeIndex).LP <> -1 Then

'Traverse the left node

PreorderTraversalEncoding(tree(nodeIndex).LP, currentCode & "0", tree, unique, uniqueSize, statistics)

End If

If tree(nodeIndex).RP <> -1 Then

'Traverse the right node

PreorderTraversalEncoding(tree(nodeIndex).RP, currentCode & "1", tree, unique, uniqueSize, statistics)

End If

End If

End Sub

Sub PreorderTraversalDecoding(nodeIndex As Integer, currentCode As String, ByRef tree() As Node, ByRef EachLineInBinary As List(Of String), ByRef binaryline As List(Of String), ByRef OriginalLines As List(Of String), ByRef fullline As String)

If tree(nodeIndex).LP = -1 And tree(nodeIndex).RP = -1 Then

If binaryline.Count <> 0 And EachLineInBinary.Count <> 0 Then

Dim PlaceholderCode As String = "" 'remove n characters from binaryline

If binaryline.Count >= currentCode.Length Then 'makes sure that the length is valid

For u = 0 To currentCode.Length - 1

PlaceholderCode = PlaceholderCode & binaryline(u)

Next

End If

If PlaceholderCode = currentCode And currentCode <> "" Then 'if they are equal to each other

fullline = fullline & tree(nodeIndex).character 'As the two codes are equal append the character equivilant to the output line

'if A = 01 and the currentcode and the placeholder code is equal to 01 then the character A would be added to the fullline

For i = 0 To currentCode.Length - 1

binaryline.RemoveAt(0) 'remove the character from the line

Next

If binaryline.Count = 0 Then

EachLineInBinary.RemoveAt(0)

OriginalLines.Add(fullline)

'fullline is a line from the decoded text.

Console.WriteLine(fullline) 'outputs the line

fullline = ""

End If

End If

currentCode = "" 'reset the code

End If

Else

If tree(nodeIndex).LP <> -1 Then

'Traverse the left node

PreorderTraversalDecoding(tree(nodeIndex).LP, currentCode & "0", tree, EachLineInBinary, binaryline, OriginalLines, fullline)

End If

If tree(nodeIndex).RP <> -1 Then

'Traverse the right node

PreorderTraversalDecoding(tree(nodeIndex).RP, currentCode & "1", tree, EachLineInBinary, binaryline, OriginalLines, fullline)

End If

End If

End Sub

Public Sub ConvertFromBinaryToText(ByRef EachLineInBinary As List(Of String), ByRef tree() As Node, treeSize As Integer, binaryline As List(Of String), ByRef OriginalLines As List(Of String), fullline As String)

While EachLineInBinary.Count > 0

For h = 0 To EachLineInBinary(0).Length - 1

binaryline.Add(EachLineInBinary(0).Substring(h, 1)) 'adding each binary bit to an array

'each bit in the binary line will have its own index in the array

Next

While EachLineInBinary.Count > 0 AndAlso binaryline.Count > 0 'while there are still lines which need to be converted back into text

PreorderTraversalDecoding(treeSize - 1, "", tree, EachLineInBinary, binaryline, OriginalLines, fullline)

End While

End While

End Sub

Public Sub RetrieveBinaryLines(filename As String, ByRef eachlineinbinary As List(Of String), ByRef eachlineinbinarylength As List(Of Integer))

Dim totallength As Integer = 0

For y = 0 To eachlineinbinarylength.Count - 1 'sum of all total bits

totallength = totallength + eachlineinbinarylength(y)

Next

If eachlineinbinary.Count = 0 Then ' Adds an empty string to the eachlineinbinary array

eachlineinbinary.Add("")

End If

Dim finalStringBuilder As New StringBuilder()

Using fileStreaming As New FileStream(filename, FileMode.Open, FileAccess.Read) ' Open the binary file

Using reader As New BinaryReader(fileStreaming)

Dim fileLength As Long = fileStreaming.Length

For x = 0 To fileLength - 1

Dim byteStore As Byte = reader.ReadByte()

Dim binaryCode As String = Convert.ToString(byteStore, 2)

If x < fileLength - 1 Then

While binaryCode.Length <> 8 'ensures it is an 8 bit binary code

binaryCode = "0" & binaryCode

End While

finalStringBuilder.Append(binaryCode)

Else

' For the last byte, ensure the length matches the total length

While finalStringBuilder.Length + binaryCode.Length <> totallength And binaryCode.Length <> 8

binaryCode = "0" & binaryCode

End While

finalStringBuilder.Append(binaryCode) ' Add to StringBuilder

End If

Next

End Using

End Using

Dim finalString As String = finalStringBuilder.ToString() ' Convert StringBuilder to string

Dim nextBit As Integer = 0 ' Bit of binary code being read

While eachlineinbinarylength.Count > 0

If eachlineinbinarylength(0) = finalString.Length Then ' Just add the entire thing if the lengths are equal (There is only a single line of text)

eachlineinbinary(eachlineinbinary.Count - 1) = eachlineinbinary(eachlineinbinary.Count - 1) & finalString

ElseIf eachlineinbinarylength(0) < finalString.Length Then ' Looping until the line ends

Dim placeholder As New StringBuilder() ' Used to store the line

For h = 0 To eachlineinbinarylength(0) - 1

placeholder.Append(finalString.Substring(nextBit + h, 1)) ' Adds the bit to the placeholder

Next

nextBit = nextBit + eachlineinbinarylength(0) ' Update the next bit to be read

eachlineinbinary(eachlineinbinary.Count - 1) = eachlineinbinary(eachlineinbinary.Count - 1) & placeholder.ToString()

End If

eachlineinbinary.Add("")

eachlineinbinarylength.RemoveAt(0) ' Remove the line length from the array

End While

eachlineinbinary.RemoveAt(eachlineinbinary.Count - 1) ' Remove the last item in the array which is an empty string

End Sub

Public Sub ConvertAllTextToBinary(lines As List(Of String), tree() As Node, treeSize As Integer, ByRef EachLineInBinary As List(Of String), statistics As Boolean)

If EachLineInBinary.Count = 0 Then 'as the arraylist is empty, add an empty string

EachLineInBinary.Add("")

End If

For x = 0 To lines.Count - 1 'looping through each line

For i = 0 To lines(x).Length - 1 'looping through each character

For h = 0 To treeSize - 1 'looping through the tree

If tree(h).character = lines(x).Substring(i, 1) Then 'if the tree character matches the character in the line

'add the binary equivalent to the eachlineinbinary array

EachLineInBinary(EachLineInBinary.Count - 1) = EachLineInBinary(EachLineInBinary.Count - 1) & tree(h).BinaryCode

End If

Next

Next

EachLineInBinary.Add("")

Next

EachLineInBinary.RemoveAt(EachLineInBinary.Count - 1) 'removes the final empty string

If statistics = False Then

For a = 0 To EachLineInBinary.Count - 1

Console.WriteLine(EachLineInBinary(a))

Next

Console.WriteLine()

End If

End Sub

Public Function SplitIntoBytes(ByRef EachLineInBinary As List(Of String), tree() As Node, treesize As Integer, statistics As Boolean) As Integer

Dim BinaryBytes As New List(Of String)

BinaryBytes.Add("") ' Start with an empty string to collect bits

For x = 0 To EachLineInBinary.Count - 1 ' Loop through each encoded line

For a = 0 To EachLineInBinary(x).Length - 1 ' Loop through each bit in the line

If BinaryBytes(BinaryBytes.Count - 1).Length = 8 Then ' If we have 8 bits, create a new string

BinaryBytes.Add("")

End If

BinaryBytes(BinaryBytes.Count - 1) = BinaryBytes(BinaryBytes.Count - 1) & EachLineInBinary(x).Substring(a, 1) ' Append bit to the current byte

Next

Next

If statistics = False Then

WritingToBinaryFile(BinaryBytes, treesize, tree, EachLineInBinary)

End If

'if the code is running through the huffman coding UI Then get user to input file

Return BinaryBytes.Count - 1 'return the number of bytes. This is required in the statistics class

End Function

Public Sub WritingToBinaryFile(binarybytes As List(Of String), treesize As Integer, tree() As Node, eachlineinbinary As List(Of String))

Console.WriteLine()

Console.WriteLine("Input filepath of binary file:")

Dim filepath As String = Console.ReadLine

While DetermineTextFile(filepath) = True Or filepath.Length <= 4

Console.WriteLine("Input filepath of binary file:")

filepath = Console.ReadLine

End While

CreateDirectoryAndFile(filepath, True)

Using fs As New FileStream(filepath, FileMode.Create)

Using writer As New BinaryWriter(fs)

For i = 0 To binarybytes.Count - 1

Dim ByteString As String = binarybytes(i)

' Only write complete bytes

Dim byteValue As Byte = Convert.ToByte(ByteString, 2) ' Convert binary string to byte

writer.Write(byteValue) ' Write the byte to the file

Next

End Using

End Using

SaveHuffmanTree(treesize, tree, eachlineinbinary, filepath)

End Sub

Public Sub SaveHuffmanTree(treesize As Integer, tree() As Node, eachlineinbinary As List(Of String), filename As String)

CreateDirectoryAndFile("C:\CourseworkTesting/TreeSave.txt", False)

FileOpen(5, "C:\CourseworkTesting/TreeSave.txt", OpenMode.Output)

PrintLine(5, filename) 'for when it is opened

'adds the length of eachlineinbinary to the file

For h = 0 To eachlineinbinary.Count - 1

PrintLine(5, eachlineinbinary(h).Length)

Next

PrintLine(5, " ")

For x = 0 To treesize - 1

'sends the current state of the tree into a text file

PrintLine(5, tree(x).LP)

PrintLine(5, tree(x).RP)

PrintLine(5, tree(x).total)

PrintLine(5, tree(x).BinaryCode)

PrintLine(5, tree(x).IsProcessed)

PrintLine(5, tree(x).character)

PrintLine(5, tree(x).CharacterProduced)

Next

FileClose(5)

End Sub

Public Function RetrieveHuffmanTree(ByRef treesize As Integer, ByRef tree() As Node, ByRef eachlineinbinary As List(Of String), ByRef filename As String, ByRef eachlineinbinarylength As List(Of Integer)) As Boolean

FileOpen(5, "C:\CourseworkTesting/TreeSave.txt", OpenMode.Input)

Dim TextFileName As String = ""

If Not EOF(5) Then

TextFileName = LineInput(5)

End If

If filename = TextFileName And DetermineTextFile(TextFileName) = False Then 'if the filepath matches the file path in the text file and is a binary file

Dim linelength As String = ""

If Not EOF(5) Then

Do

linelength = LineInput(5)

If linelength <> "" And linelength <> " " Then 'if the line is not empty then it will be a number

eachlineinbinarylength.Add(CInt(linelength)) 'add the linelength to list

End If

Loop Until linelength = " " Or EOF(5)

'loops until all eachlineinbinarylength is retreived or file ends

End If

While Not EOF(5) 'retrieves each part of the tree

tree(treesize).LP = LineInput(5)

tree(treesize).RP = LineInput(5)

tree(treesize).total = LineInput(5)

tree(treesize).BinaryCode = LineInput(5)

tree(treesize).IsProcessed = LineInput(5)

tree(treesize).character = LineInput(5)

tree(treesize).CharacterProduced = LineInput(5)

treesize = treesize + 1

End While

filename = ""

filename = TextFileName

FileClose(5)

Return True

Else

Console.WriteLine("Invalid file path")

'if they have not inputted the same filepath as the one in the text file

'the user will be returned to the main menu

FileClose(5)

Return False

End If

End Function

Public Sub ResetDecoding(ByRef eachlineinbinary As List(Of String), ByRef eachlineinbinarylength As List(Of Integer), ByRef treesize As Integer)

eachlineinbinary.Clear()

eachlineinbinarylength.Clear()

treesize = 0

End Sub

Public Sub HuffmanCodingUserInterface(huffman As HuffmanCoding, ByRef lines As List(Of String), filename As String)

Dim tree(1000) As Node

Dim unique(1000) As characters

Dim treeSize As Integer = 0 'size of the tree

Dim uniqueSize As Integer = 0 'Number of unique characters

Dim priorityQueue As New List(Of Integer)

Dim EachLineInBinary As New List(Of String) 'each line in binary

Dim BinaryLine As New List(Of String)

Dim eachlineinbinarylength As New List(Of Integer)

Dim choice As String = ""

Dim ExitHuffman As Boolean = False 'used to exit the program if the file is not a text file

Dim Statistics As Boolean = False

Dim OriginalLines As New List(Of String)

Do

choice = ""

If DetermineTextFile(filename) = True Then 'if it is a valid filepath for a text file

While choice <> "1" And choice <> "2" And choice <> "3"

'loops until user inputs 1,2 or 3

Console.WriteLine("1:View the key")

Console.WriteLine("2:View Character Frequency Distribition")

Console.WriteLine("3:Return to main menu")

choice = Console.ReadLine

Console.WriteLine()

End While

SetUpArrays(tree, unique, treeSize, uniqueSize)

ResettingCodeProduced(unique, uniqueSize, tree, treeSize)

CollectingUniqueCharacters(unique, uniqueSize, lines)

SortingOutFrequency(unique, uniqueSize, priorityQueue)

CreatingRootNodes(priorityQueue, tree, treeSize)

'if there is only a single character in the text

If priorityQueue.Count = 1 Then

tree(0).BinaryCode = "0"

tree(0).character = unique(0).Character

tree(0).total = unique(0).CharacterFrequency

tree(0).LP = -1

tree(0).RP = -1

Else

'As the tree needs more than one item in the priority queue construct the tree using the huffmantree subroutine

HuffmanTree(priorityQueue, tree, treeSize)

End If

Console.WriteLine()

If choice = "1" Then

Dim nodeindex As Integer = treeSize - 1

Dim currentcode As String = ""

'traverses the tree

PreorderTraversalEncoding(nodeindex, currentcode, tree, unique, uniqueSize, Statistics)

Dim returnback As String = ""

Console.WriteLine()

While returnback <> "1" And returnback <> "2"

Console.WriteLine("1:Compress text and save to binary file")

Console.WriteLine("2:Return to Huffman Coding Menu")

returnback = Console.ReadLine

Console.WriteLine()

End While

If returnback = "1" Then

ConvertAllTextToBinary(lines, tree, treeSize, EachLineInBinary, Statistics)

SplitIntoBytes(EachLineInBinary, tree, treeSize, Statistics)

ExitHuffman = True

Console.WriteLine()

End If

ResettingCodeProduced(unique, uniqueSize, tree, treeSize)

ElseIf choice = "2" Then

FrequencyDistribution(unique, uniqueSize)

End If

Else

Dim DecodeOption As String = ""

While DecodeOption <> "1" And DecodeOption <> "2"

'loops until user inputs 1 or 2

Console.WriteLine("1:Decode binary file")

Console.WriteLine("2:Exit")

DecodeOption = Console.ReadLine

End While

If DecodeOption = "1" Then

If RetrieveHuffmanTree(treeSize, tree, EachLineInBinary, filename, eachlineinbinarylength) = True Then

RetrieveBinaryLines(filename, EachLineInBinary, eachlineinbinarylength) 'retrives all of the binary lines

ConvertFromBinaryToText(EachLineInBinary, tree, treeSize, BinaryLine, OriginalLines, "")

Console.WriteLine()

Dim SaveToFile As String = ""

While SaveToFile <> "1" And SaveToFile <> "2"

Console.WriteLine("1:Save to text file")

Console.WriteLine("2:Exit")

SaveToFile = Console.ReadLine

End While

If SaveToFile = "1" Then

resavingToATextFile("", OriginalLines)

End If

End If

End If

ExitHuffman = True

End If

ResetDecoding(EachLineInBinary, eachlineinbinarylength, treeSize)

Console.WriteLine()

Loop Until choice = "3" Or ExitHuffman = True

End Sub

End Class

Public Class Statistics

Public Function CalculatingTreeSize(tree() As Node, treesize As Integer, eachlineinbinary As List(Of String)) As Integer

Dim ByteCount As Integer = 0

For x = 0 To treesize - 1

ByteCount = ByteCount + ((tree(x).LP).ToString).Length

ByteCount = ByteCount + ((tree(x).RP).ToString).Length

If tree(x).BinaryCode <> "" Then

ByteCount = ByteCount + (tree(x).BinaryCode).Length

Else

ByteCount = ByteCount + 1

End If

If tree(x).character <> "" Then

ByteCount = ByteCount + (tree(x).character).Length

Else

ByteCount = ByteCount + 1

End If

ByteCount = ByteCount + ((tree(x).total).ToString).Length

ByteCount = ByteCount + ((tree(x).CharacterProduced).ToString).Length

ByteCount = ByteCount + ((tree(x).IsProcessed).ToString).Length

Next

For a = 0 To eachlineinbinary.Count - 1

Dim Linelength As String = eachlineinbinary(a).Length

ByteCount = ByteCount + (Linelength.Length \* 4)

Next

Return ByteCount

End Function

Public Sub StatisticsUI(Huffman As HuffmanCoding, Dictionary As DictionaryDataCompression, OriginalFilePath As String, lines As List(Of String))

Dim choice As String = ""

While choice <> "1" And choice <> "2"

'ensures a valid choice is entered

Console.WriteLine("1:Continue")

Console.WriteLine("2:Exit")

choice = Console.ReadLine

End While

If choice = "1" Then

AllSubroutines(Huffman, Dictionary, OriginalFilePath, lines)

End If

End Sub

Public Sub AllSubroutines(Huffman As HuffmanCoding, Dictionary As DictionaryDataCompression, OriginalFilePath As String, ByRef lines As List(Of String))

'General variables used

Dim stopwatch As New Stopwatch

Dim originalfile As New FileInfo(OriginalFilePath)

Dim originalfilepathsize As Long = originalfile.Length

Console.WriteLine("Original file path size " & originalfilepathsize)

'Variables/Functions/Subroutines for Huffman coding

Dim tree(1000) As Node

Dim unique(1000) As characters

Dim treeSize As Integer = 0

Dim uniqueSize As Integer = 0

Dim priorityQueue As New List(Of Integer)

Dim EachLineInBinary As New List(Of String)

Dim BinaryLine As New List(Of String)

Dim DecodeBoolean As Boolean = False

Dim eachlineinbinarylength As New List(Of Integer)

Dim DoneOnce As Boolean = False

Dim stats As Boolean = True

Dim originalLines As New List(Of String)

stopwatch.Start()

Huffman.SetUpArrays(tree, unique, treeSize, uniqueSize)

Huffman.ResettingCodeProduced(unique, uniqueSize, tree, treeSize)

Huffman.CollectingUniqueCharacters(unique, uniqueSize, lines)

Huffman.SortingOutFrequency(unique, uniqueSize, priorityQueue)

Huffman.CreatingRootNodes(priorityQueue, tree, treeSize)

While priorityQueue.Count > 1

Huffman.HuffmanTree(priorityQueue, tree, treeSize)

End While

'starts from the top of the tree

Huffman.PreorderTraversalEncoding(treeSize - 1, "", tree, unique, uniqueSize, stats)

Huffman.ConvertAllTextToBinary(lines, tree, treeSize, EachLineInBinary, stats)

Dim HuffmanSizeInbytes As Integer = Huffman.SplitIntoBytes(EachLineInBinary, tree, treeSize, stats)

Dim TreeSizeInBytes As Integer = CalculatingTreeSize(tree, treeSize, EachLineInBinary)

Console.WriteLine("Huffman file size in bytes " & (HuffmanSizeInbytes + TreeSizeInBytes))

stopwatch.Stop()

Console.WriteLine(stopwatch.Elapsed)

stopwatch.Restart()

'Variables/Functions/Subroutines for Dictionary Data Compression

Dim WordBank As New List(Of String)

Dim WordFrequencies As New List(Of Integer)

Dim encodedlines As New List(Of String)

Dictionary.GettingAllNewUniqueWords(WordBank, WordFrequencies, lines)

Dictionary.CompressText(encodedlines, lines, WordBank, WordFrequencies)

Dim DictionarySizeInBytes As Integer = Dictionary.NumberOfBytes(encodedlines)

Dim KeySizeInBytes As Integer = Dictionary.NumberOfBytes(WordBank)

KeySizeInBytes = KeySizeInBytes + (OriginalFilePath.Length)

DictionarySizeInBytes = DictionarySizeInBytes + KeySizeInBytes

Console.WriteLine("Dictionary Data Compression File Size in bytes " & DictionarySizeInBytes)

stopwatch.Stop()

Console.WriteLine(stopwatch.Elapsed)

percentages(DictionarySizeInBytes, originalfilepathsize, HuffmanSizeInbytes, TreeSizeInBytes)

End Sub

Public Sub percentages(DictionarySizeInBytes As Integer, OriginalFilePathsize As Long, HuffmanSizeInBytes As Integer, treesizeinbytes As Integer)

Console.WriteLine()

Dim DictionaryPercent As Double = (1 - (DictionarySizeInBytes / (CInt(OriginalFilePathsize)))) \* 100

Dim HuffmanPercent As Double = (1 - ((HuffmanSizeInBytes + (treesizeinbytes)) / OriginalFilePathsize)) \* 100

Console.WriteLine("Huffman percentage reducton " & HuffmanPercent)

Console.WriteLine("Dictionary percentage reduction " & DictionaryPercent)

Console.WriteLine()

End Sub

End Class

Public Class DictionaryDataCompression

Inherits FileHandling

Function partion(ByRef WordFrequencies As List(Of Integer), LP As Integer, RP As Integer, ByRef wordbank As List(Of String)) As Integer

Dim pivot As Integer = WordFrequencies(LP)

Dim i As Integer = LP - 1

Dim j As Integer = RP + 1

While i < j 'repeat this code until it reaches the pivot point

Do

i = i + 1

Loop Until WordFrequencies(i) <= pivot

Do

j = j - 1

Loop Until WordFrequencies(j) >= pivot

If i >= j Then

Return j

Else

'swaps items around

Dim placeholder As Integer = WordFrequencies(i)

WordFrequencies(i) = WordFrequencies(j)

WordFrequencies(j) = placeholder

Dim wordholder As String = wordbank(i)

wordbank(i) = wordbank(j)

wordbank(j) = wordholder

End If

End While

Return j

End Function

Public Sub quicksort(ByRef WordFrequencies As List(Of Integer), LP As Integer, RP As Integer, ByRef wordbank As List(Of String))

If LP < RP Then

Dim PartionIndex As Integer = partion(WordFrequencies, LP, RP, wordbank)

quicksort(WordFrequencies, LP, PartionIndex, wordbank)

quicksort(WordFrequencies, PartionIndex + 1, RP, wordbank)

End If

End Sub

Public Sub GettingAllNewUniqueWords(ByRef wordbank As List(Of String), ByRef wordfrequencies As List(Of Integer), lines As List(Of String))

Dim word As String = ""

Dim PreviouslyStoredWord As Boolean = False 'checks to see if the word has already been stored in the wordbank

Dim PreviouslyStoredCharacter As Boolean = False 'checks to see if the character has already been added in the wordbank

For i = 0 To lines.Count - 1 'looping each line

For x = 0 To lines(i).Length - 1 'looping for each word in each line

If Char.IsLetterOrDigit(lines(i).Substring(x, 1)) Then

word = word & lines(i).Substring(x, 1)

ElseIf x < lines(i).Length - 2 And (lines(i).Substring(x, 1) = "-" Or lines(i).Substring(x, 1) = "—") Then

If Char.IsLetter(lines(i).Substring(x + 1, 1)) Then 'ensures that a letter is the next character in the string

word = word & lines(i).Substring(x, 1)

End If

ElseIf lines(i).Substring(x, 1) = "'" Or lines(i).Substring(x, 1) = "’" Then

word = word & lines(i).Substring(x, 1)

Else

PreviouslyStoredWord = False

CheckingWordExists(PreviouslyStoredWord, wordbank, wordfrequencies, lines, word)

CheckingWordExists(PreviouslyStoredWord, wordbank, wordfrequencies, lines, lines(i).Substring(x, 1))

word = ""

End If

Next

CheckingWordExists(PreviouslyStoredWord, wordbank, wordfrequencies, lines, word)

If i <> lines.Count - 1 Then 'if it is not the final line set word to ""

word = ""

End If

Next

End Sub

Public Sub CompressText(ByRef EncodedLines As List(Of String), lines As List(Of String), wordbank As List(Of String), wordfreequencies As List(Of Integer))

Dim word As String = ""

Dim line As String = ""

Dim CurrentCharacter As String = ""

For i = 0 To lines.Count - 1

line = ""

For x = 0 To lines(i).Length - 1

If Char.IsLetterOrDigit(lines(i).Substring(x, 1)) Then

word = word & lines(i).Substring(x, 1)

ElseIf x < lines(i).Length - 2 And (lines(i).Substring(x, 1) = "-" Or lines(i).Substring(x, 1) = "—") Then

If Char.IsLetter(lines(i).Substring(x + 1, 1)) Then

word = word & lines(i).Substring(x, 1)

End If

ElseIf lines(i).Substring(x, 1) = "'" Or lines(i).Substring(x, 1) = "’" Then

word = word & lines(i).Substring(x, 1)

Else

CurrentCharacter = lines(i).Substring(x, 1)

'adds number equivilant of the word and the character to a string

WordChecker(word, wordbank, line)

WordChecker(CurrentCharacter, wordbank, line)

word = ""

End If

Next

CurrentCharacter = ""

WordChecker(word, wordbank, line)

WordChecker(CurrentCharacter, wordbank, line)

'add compressed lines to array

word = ""

EncodedLines.Add(line)

Next

End Sub

Function NumberOfBytes(EncodedLines As List(Of String)) As Integer

Dim ByteCount As Integer = 0

For x = 0 To EncodedLines.Count - 1

ByteCount = ByteCount + EncodedLines(x).Length

Next

Return ByteCount

End Function

Public Sub CheckingWordExists(PreviouslyStoredWord As Boolean, ByRef wordbank As List(Of String), ByRef wordfrequencies As List(Of Integer), lines As List(Of String), word As String)

If word <> "" Then 'if the word isnt just empty

PreviouslyStoredWord = False

For x = 0 To wordbank.Count - 1 'looping through all of the current words stored

If word = wordbank(x) Then 'if equal to a word then increase the count of that word by one

PreviouslyStoredWord = True 'previously stored word is not set to true

wordfrequencies(x) = wordfrequencies(x) + 1

End If

Next

'if the word has not been stored already then set up a new word

If PreviouslyStoredWord = False Then

wordbank.Add(word)

wordfrequencies.Add(1)

End If

End If

End Sub

Public Sub DictionaryDataCompressionIntoTextFile(lines As List(Of String), ByRef wordbank As List(Of String), filepath As String, ByRef encodedlines As List(Of String), wordfrequencies As List(Of Integer))

Console.WriteLine("Input file path")

filepath = Console.ReadLine

CreateDirectoryAndFile(filepath, False)

If encodedlines.Count = 0 Then

CompressText(encodedlines, lines, wordbank, wordfrequencies)

End If

FileOpen(2, filepath, OpenMode.Output) 'opens the file

For x = 0 To encodedlines.Count - 1

'loops through all of the encoded lines And adds them to the text file

PrintLine(2, encodedlines(x))

Next

FileClose(2)

'removes all of the encoded lines from the array

StoringKey(wordbank, filepath)

End Sub

Public Sub ViewText(Outputlines As List(Of String))

'loops through each of the encodedlines and then outputs them

For x = 0 To Outputlines.Count - 1

Console.WriteLine(Outputlines(x))

Next

End Sub

Public Sub StoringKey(wordbank As List(Of String), userpath As String)

Dim filepath As String = "C:\CourseworkTesting/Dictionary.txt"

CreateDirectoryAndFile(filepath, False)

FileOpen(5, filepath, OpenMode.Output) 'opens the file

PrintLine(5, userpath)

For x = 0 To wordbank.Count - 1

'loops through all of the encoded lines and adds them to the text file

PrintLine(5, wordbank(x))

Next

FileClose(5)

End Sub

Sub WordChecker(word As String, wordbank As List(Of String), ByRef line As String)

If word <> "" Then 'if word is valid then

For x = 0 To wordbank.Count - 1 'looping through each word

If wordbank(x) = word And ((x).ToString).Length <= word.Length Then

line = line & x & " " 'adds corresponding number to the code

ElseIf wordbank(x) = word And ((x).ToString).Length > word.Length Then

line = line & word & " "

End If

Next

End If

End Sub

Public Sub DictionaryDataCompressionKey(wordbank As List(Of String))

Console.WriteLine()

Console.WriteLine("Key")

Console.WriteLine()

'loops through each word in the wordbank

For x = 0 To wordbank.Count - 1

If ((x).ToString).Length > wordbank(x).Length Then

Console.WriteLine(wordbank(x) & " " & wordbank(x))

Else

Console.WriteLine(x & " " & wordbank(x))

End If

Next

End Sub

Public Sub FrequencyDistribution(wordbank As List(Of String), wordfrequencies As List(Of Integer))

For x = 0 To wordbank.Count - 1

Console.WriteLine(wordbank(x) & " " & wordfrequencies(x))

Next

End Sub

Public Sub DecompressText(encodedlines As List(Of String), wordbank As List(Of String), ByRef originallines As List(Of String))

Dim num As String = ""

Dim SingleLine As String = ""

For x = 0 To encodedlines.Count - 1 'looping through all of the encoded lines

For a = 0 To encodedlines(x).Length - 1 'looping through each character of the encoded line

If Not (Char.IsDigit(encodedlines(x).Substring(a, 1))) And encodedlines(x).Substring(a, 1) <> " " Then

Dim word As String = ""

While Not (Char.IsDigit(encodedlines(x).Substring(a, 1))) And encodedlines(x).Substring(a, 1) <> " " And a < encodedlines(x).Length

word = word & encodedlines(x).Substring(a, 1)

a = a + 1

End While

For h = 0 To wordbank.Count - 1

If word = wordbank(h) Then

SingleLine = SingleLine & word

End If

Next

ElseIf encodedlines(x).Substring(a, 1) = " " Then

a = a + 1

Else

While encodedlines(x).Substring(a, 1) <> " " And a < encodedlines(x).Length

num = num & encodedlines(x).Substring(a, 1) 'adds character to the string of numbers

a = a + 1

End While

For i = 0 To wordbank.Count - 1

If i = CInt(num) Then

SingleLine = SingleLine & wordbank(i) 'adds the word to the line

End If

Next

num = ""

End If

Next

originallines.Add(SingleLine)

SingleLine = ""

Next

End Sub

Public Sub DictionaryDataCompressionUserInterface(dictionary As DictionaryDataCompression, ByRef doneonce As Boolean, ByRef lines As List(Of String), filepath As String)

Dim WordBank As New List(Of String) 'wordbank is only used in DictionaryDataCompression Class

Dim WordFrequencies As New List(Of Integer)

Dim encodedlines As New List(Of String)

Dim originallines As New List(Of String)

Dim stats As Boolean = False

Dim exitUI As Boolean = False

If doneonce = False Then 'ensures that the words are only collected once

CreateDirectoryAndFile("C:\CourseworkTesting/Dictionary.txt", False)

FileOpen(5, "C:\CourseworkTesting/Dictionary.txt", OpenMode.Input)

Dim RetreivedFilepath As String = ""

If Not EOF(5) Then 'if there is text in the text file

RetreivedFilepath = LineInput(5)

End If

If RetreivedFilepath <> filepath Then 'if they are not decoding from the text file

FileClose(5)

GettingAllNewUniqueWords(WordBank, WordFrequencies, lines)

quicksort(WordFrequencies, 0, WordBank.Count - 1, WordBank)

Else

While Not EOF(5)

WordBank.Add(LineInput(5)) 'adding the word to the file

WordFrequencies.Add(0)

End While

FileClose(5)

FileOpen(2, filepath, OpenMode.Input) 'opening the filepath of the text file which the user inputted

While Not EOF(2)

encodedlines.Add(LineInput(2))

End While

FileClose(2)

DecompressText(encodedlines, WordBank, originallines)

GettingAllNewUniqueWords(WordBank, WordFrequencies, originallines)

originallines.Clear()

End If

doneonce = True

End If

Dim choice As String = ""

Do

Console.WriteLine()

choice = ""

While choice <> "1" And choice <> "2" And choice <> "3" And choice <> "4" And choice <> "5"

Console.WriteLine("1:View the key")

Console.WriteLine("2:View Compressed text")

Console.WriteLine("3:View Decompressed Text")

Console.WriteLine("4:View Frequency distribution")

Console.WriteLine("5:Return to main menu")

choice = Console.ReadLine

Console.WriteLine()

End While

If choice = "1" Then

DictionaryDataCompressionKey(WordBank)

ElseIf choice = "2" Then

Console.WriteLine()

If encodedlines.Count = 0 Then

CompressText(encodedlines, lines, WordBank, WordFrequencies)

End If

ViewText(encodedlines)

Dim ReturnBack As String = ""

While ReturnBack <> "1" And ReturnBack <> "2"

Console.WriteLine()

Console.WriteLine("1:Save Compressed Text")

Console.WriteLine("2:Return back")

ReturnBack = Console.ReadLine

End While

If ReturnBack = "1" Then

DictionaryDataCompressionIntoTextFile(lines, WordBank, "", encodedlines, WordFrequencies)

Console.WriteLine("Completed")

exitUI = True

End If

ElseIf choice = "3" Then

If encodedlines.Count = 0 Then

CompressText(encodedlines, lines, WordBank, WordFrequencies)

End If

DecompressText(encodedlines, WordBank, originallines)

ViewText(originallines)

Dim Userinput As String = ""

While Userinput <> "1" And Userinput <> "2"

Console.WriteLine()

Console.WriteLine()

Console.WriteLine("1:Save decompressed text to text file")

Console.WriteLine("2:Return to Dictionary data compression menu")

Userinput = Console.ReadLine

Console.WriteLine()

End While

If Userinput = "1" Then

resavingToATextFile(Userinput, originallines)

exitUI = True 'so user is sent back to the main menu

End If

ElseIf choice = "4" Then

FrequencyDistribution(WordBank, WordFrequencies)

End If

Console.WriteLine()

Loop Until choice = "5" Or exitUI = True

WordBank.Clear()

WordFrequencies.Clear()

encodedlines.Clear()

End Sub

End Class

End Module

# Testing

## Introduction To Testing

The testing for my project has been done in a video format.

All the files and sub folders used in the testing have been stored in the folder **C:\CourseworkTesting/**

All the tests have been carried out with the file **C:\CourseworkTesting/Mechanisms.txt**. I decided to use this file as it would be easy enough to see if the program successfully compressed and decompressed text as there a limited number of unique words/characters in the text is which means that the key would be relatively small. If I used a file which had a very long key, it would take me a while to check if the program was successful in completing the objectives that I set out in my analysis. I also did a very quick partial demonstration using the **C:\CourseworkTesting/AnInspectorCalls.txt** file by showing the program can handle the use of more sophisticated punctuation and a greater range of vocabulary by showcasing the key using both algorithms. The other files of text do not class as full tests at all, but they were used as some more statistical samples for the program when I discuss the effectiveness of the program in the evaluation section of the report.

If I was to use the program on another device, I would have to modify the location of where the tree and the dictionary would be stored. This is because they are currently stored in C drive in the folder which I have specified for them to be stored in.

**https://www.youtube.com/watch?v=I1vu9qxyGUc**

## Explanations To Tests

**Huffman Coding Tests**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objective** | **Purpose of test** | **Expected Result** | **PASS**  **/FAIL** | **Time Stamp** |
| **1** | The program cannot be broken by any invalid inputs such as inputting invalid options in the menus and inputting any invalid file paths. | **onew503495890** is an invalid response for the main menu as it is not any of the options being 1 to 4 so should ask the user for another response  **\*)(F£894905fiodvjfdgjk** is also an invalid response for the main menu as it is not any of the options being 1 to 4 so should ask the user for another response  Inputting **1** is a valid response so the user will be requested to input a file path  When inputting **C:\CourseworkTesting/**  **Empty.txt** or any empty file a response saying that is an empty file should be displayed to the user and the user is re-directed to the main menu  When **C:\CourseworkTesting**  **/Mechanisms.txt** is inputted, the user should be displayed with the user interface for Huffman coding where the user can compress the text as this is a valid text file.  When **C:\CourseworkTesting**  **/HuffmanFiles/**  **MechanismsHuffman** is inputted, a separate user interface should be displayed asking the user if they want to decompress the text as you can only decompress the text when inputting a binary file. | PASS | 00:50 – 04:41 |
| **2** | Checks to see if the program will allow users to compress the text accurately and store it in a suitable binary file | **File-path handling**  **one** – Invalid binary file  **C:\CourseworkTesting/**  **Huffman.txt** – Invalid binary file as a valid binary file cannot end in .txt as that would result in a text file being produced.  **C:\CourseworkTesting\BinaryFiles/Mechanisms**  **Huffman** – Valid Binary file as it does not end in .txt and the file is stored on a suitable drive. This should result in a new folder called BinaryFiles being created and inside that folder that is where the MechanismsHuffman file will be stored.  Once the binary file has been produced, a text file will also be produced called TreeSave. The file path of this would be **C:\CourseworkTesting/**  **TreeSave.txt**  Both the binary file and the tree must be accessible by the user.  **Accuracy test – Binary codes produced correctly**  For this demonstration, I decided to use a simple piece of text used earlier being SEE SEA SHELLS.  **A screenshot of a computer  Description automatically generated**  If I was to traverse the tree from the image above  A would be 000, S would be 01, H would be 001, E would be 10, L would be 111 and space would be 110. As expected my console output of the binary codes here are the same binary codes achieved if I was to traverse the Huffman tree that was generated online  A black background with white text  AI-generated content may be incorrect.  Here is a sketch of the Huffman Tree for SEE SEA SHELLS above which match the binary codes produced by the online Huffman Tree generator and my programs output.  The binary codes for the text **C:\CourseworkTesting/**  **Mechanisms.txt** should also be displayed to the user.  The binary codes for the text **C:\CourseworkTesting/**  **AnInspectorCalls.txt** should also be displayed to the user. The key in the text will be considerably bigger and should showcase the program handling much larger keys with a greater range of punctuation | PASS | **01:56 – 03:07**  (compression check)  **01:47 – 01:56**  **AND**  **09:04 – 09:35**  (binary code) |
| **3** | Checks to see if the program will allow users to decompress the text and store it in a suitable text file | **C:\CourseworkTesting\**  **HuffmanFiles/Huffman**  **Decompressed.txt** – would be a valid file path so would create a new file called HuffmanDecompressed in the same HuffmanFiles folder used earlier. The text should be the exact same as the original text. A quick check is that both the original **C:\CourseworkTesting/**  **Mechanisms.txt** file and the **C:\CourseworkTesting\**  **HuffmanFiles/Huffman**  **Decompressed.txt**  file should have the same size in KB both being 867KB.  **Accuracy Test – Frequency distribution**  The frequency distribution of the **C:\CourseworkTesting**  **/Mechanisms.txt** file and the **C:\CourseworkTesting\**  **HuffmanFiles/Huffman**  **Decompressed.txt** files should be equivalent to one another | PASS | **03:08 – 04:22**  (de-compress text)  **04:23- 05:02** (freq check) |

**Dictionary Data Compression Tests**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objective** | **Purpose of test** | **Expected Result** | **PASS**  **/FAIL** | **Time Stamp** |
| **1** | The program cannot be broken by any invalid inputs such as inputting invalid options in the menus and inputting any invalid file paths. While an invalid response is inputted, the user is requested to input another response.  This must reject the use of any binary files AND must only accept valid text files. | When inputting **C:\CourseworkTesting**  **/Empty.txt**  a response saying that is an empty file should be displayed to the user. (This one was done at the same time it was tested for Huffman Coding)  **C:\CourseworkTesting\**  **HuffmanFiles**  **/MechanismsHuffman** is an invalid file path as it is a binary file and only text files can be used for Dictionary data compression | PASS | 05:02 – 10:20 |
| **2.** | To check if the program can compress the text using the Dictionary data compression algorithm and whether they can store the text into a text file. | **File-path inputs**  **C:\CourseworkTesting\**  **DictionaryData**  **Compression/**  **CompressedMechanisms.txt** is a valid response. This should create the new folder DictionaryDataCompression and inside that folder, there should be a text file called CompressedMechanisms  Once the text file has been produced, another text file will also be produced called Dictionary. The file path of this would be **C:\CourseworkTesting/**  **Dictionary.txt**  Both the text files must be accessible by the user. | PASS | 05:52 – 06:52 |
| **3.** |  | Should allow users to decompress the text and then store the text in a suitable text file.  **C:\CourseworkTesting\**  **DictionaryData**  **Compression/DecompressedMechanisms**  **Dictionary.txt** is a valid file path and will create the file DecompressedMechanismsDictionary Text file in the DictionaryDataCompression folder.  The text should be the exact same as the original text. A quick check is that both the original **C:\CourseworkTesting/**  **Mechanisms.txt** file and the **C:\CourseworkTesting\**  **DictionaryData**  **Compression/DecompressedMechanisms**  **Dictionary.txt** should have the same size in KB both being 867KB.  **Accuracy check – key**  The key produced for the **C:\CourseworkTesting/**  **Mechanisms.txt** and the **C:\CourseworkTesting\**  **DictionaryData**  **Compression/**  **DecompressedMechanisms**  **Dictionary.txt** should be exactly the same  When inputting the file **C:\CourseworkTesting/**  **AnInspectorCalls.txt** ¸the key should also be accessible and displayed to the user. Not all the items in the key will be numbers as words where the number representation of the word is bigger is not used. Instead, the actual word is used.  **Accuracy Test – Frequency distribution**  The frequency distribution of words produced for the **C:\CourseworkTesting/**  **Mechanisms.txt** and the **C:\CourseworkTesting\**  **DictionaryData**  **Compression/**  **DecompressedMechanisms**  **Dictionary.txt** should be exactly the same | PASS | **06:53 – 08:59** (decompress text)  **08:23 – 08:59** (key and word frequencies check) |

**Statistics Tests**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objective** | **Purpose of test** | **Expected Result** | **PASS**  **/FAIL** | **Time Stamp** |
| **1** | The program cannot be broken by any invalid inputs such as inputting invalid options in the menus and inputting any invalid file paths. While an invalid response is inputted, the user is requested to input another response.  This must reject the use of any binary files AND must only accept valid text files. | When inputting **C:\CourseworkTesting**  **/Empty.txt** a response saying that is an empty file should be displayed to the user. (This one was done at the same time it was tested for Huffman Coding)  **C:\CourseworkTesting**  **\HuffmanFiles**  **/MechanismsHuffman** is an invalid file path as it is a binary file and only text files can be used for the statistics.  **Iwque3048324** is also an invalid file path as it has no file path and is just some random text  When having the option to continue to the statistics the input **890348jsdfjskru348034** would be invalid as the option is between 1 and 2.  When having the option to continue to the statistics the input **1** would be valid and would show the user the statistics. **2** would also be valid but will direct the user back to the main menu | PASS | 10:21 – 14:38 |
| **2.** | The program should display the file size of the original text as well as the file size of the compressed text using both algorithms and the time taken for both algorithms to compress the text | The file size percentage reduction will be calculated using the compressed texts size and then displayed to the user. A manual check will be performed using a calculator by adding the size of the compressed text with the size of its corresponding key/dictionary. The percentage file size reductions were unknown, but this explains how I would achieve the file size reduction using the manual test and then compare this to the percentage reduction that my program calculates.  In the case of the file: **C:\CourseworkTesting**  **/Mechanisms.txt** the file size reduction was 26.8% using Huffman coding and then 63.3% using dictionary data compression.  Adding the size of the **C:\CourseworkTesting**  **/Dictionary.txt** as well as the size of the **C:\CourseworkTesting\**  **DictionaryData**  **Compression**  **/CompressedMechanisms.txt** and then dividing this total by the size of the **C:\CourseworkTesting**  **/Mechanisms.txt** and then doing 1 – this answer and then \* 100 should give 63.3%  Adding the size of the **C:\CourseworkTesting**  **\HuffmanFiles/**  **MechanismsHuffman** and the size of the **C:\CourseworkTesting**  **/TreeSave.txt** and then dividing it by the size of the **C:\CourseworkTesting**  **/Mechanisms.txt** and then doing 1- this answer and then \* 100 should give an answer of 26.8% | PASS | 10:58 |

In conclusion all the tests have been a success. As expected, the user can successfully compress and decompress text using both algorithms and can also view some statistics on the effectiveness and speed of both algorithms.

# Evaluation

## Statistical Analysis

Here is a table below of how effective the Huffman Coding algorithm was at compressing text.

|  |  |  |  |
| --- | --- | --- | --- |
| **File path** | **File Size of original text in KB** | **File size percentage reduction (3 significant figures)** | **Time taken to compress text (3 significant figures) in seconds** |
| C:\CourseworkTesting/  Mechanims.txt | 867 | 26.8% | 1.13 |
| C:\CourseworkTesting/  Small.txt | 1 | -195% | 0.000486 |
| C:\CourseworkTesting/  Sample.txt | 996 | 42.2% | 2.56 |
| C:\CourseworkTesting/  AnInspectorCalls.txt | 117 | 25.5% | 0.41 |

Excluding the anomaly being **C:\CourseworkTesting/Small.txt** due to the size of the tree making the file considerably bigger and it would completely skew the results, Huffman Coding had an average file size reduction of **31.5%** (26.8+42.2+25.5)/3 to 3 significant figures

As the file size of file increases, the time taken to compress the text also increases. For files which are incredibly small, Huffman coding is not effective due to it making the file significantly bigger due to the size of the tree. For files which are much larger, the file size reduction is much more significant which makes it highly effective.

Here is a table below of how effective the Dictionary Data Compression Algorithm was at compressing text

|  |  |  |  |
| --- | --- | --- | --- |
| **File path** | **File Size of original text in KB** | **File size percentage reduction (3 significant figures)** | **Time taken to compress text (3 significant figures)** |
| C:\CourseworkTesting/  Mechanims.txt | 867 | 63.3% | 0.133 |
| C:\CourseworkTesting/  Small.txt | 1 | 20.8% | 0.000138 |
| C:\CourseworkTesting/  Sample.txt | 996 | -2.25% | 6.61 |
| C:\CourseworkTesting/  AnInspectorCalls.txt | 117 | -21.0% | 5.94 |

There were no anomalies for Dictionary data compression

Dictionary Data Compression had an average file size reduction of **15.2%** (63.3+20.8-2.25-21.0)/4 to 3 significant figures

Unlike Huffman Coding, the time increase was not entirely dependent on the size of the file as there is another factor also affecting the time. As the size of the dictionary increases, the time taken to compress the text increases as there would be more checks between each of the words in the dictionary. This would mean that if there was a file which had a significant number of unique words and had a large amount of text, it would likely take a while. Dictionary data compression is effective at compressing text with a great amount of repetition but is not great when compressing text which a great range of vocabulary.

In conclusion, I would say that Huffman coding is the more effective text compression algorithm as on average it had a greater file size percentage reduction (excluding the anomaly). Furthermore, it also would be able to handle larger files much easier as shown with the An Inspector Calls book and the Sample file. Also a vast majority of files, do not have a great amount of repetition where the Dictionary data compression does relatively well hence making it not as practical as Huffman Coding which would be more effective in more scenarios. Dictionary Data Compression would become too slow when compressing much larger files where words have a relatively low frequency but there are a large proportion of words.

## How well objectives have been met

Here are my objectives copied and pasted from my analysis

**Huffman coding objectives**

1. Create a robust Huffman Coding menu: The program cannot be broken by any invalid inputs. Invalid inputs would be invalid file paths (when requested) such as inputting binary files when a text file is inputted and not selecting a valid digit option from any of the menus. Choosing an invalid response will result in the user being requested to input another response. This will be repeated until they input a correct response.
2. Allow users to view the compressed text when using the Huffman coding algorithm. The user must also be able to store the compressed text in a suitable binary file. A valid binary file must not end in .txt and must also be accessible on the device which it is being stored on. Any invalid binary file should result in the user being requested to input another file path. When the compressed text is stored, a file called TreeSave must also appear in a suitable folder. Both the TreeSave file and the file which holds the compressed text must be accessible to the user.
3. Allow users to decompress the text using the Huffman coding algorithm when they input a binary file. If the user inputs a valid binary file, but it is not the file which corresponds to the tree, the user will be directed back to the main menu. If the binary file correctly corresponds to the tree, then the user can view the decompressed text. Then the user can then save the decompressed text in a suitable text file. A suitable text file must end in .txt and must also be accessible on the device which it is being stored on. The user must also be able to access the file which holds the decompressed text. The decompressed text must match the original text with 100% accuracy, and this will be tested by viewing the original text’s frequency distribution and the decompressed text’s frequency distribution.

**Dictionary Data Compression Objectives**

1. Create a robust Dictionary Data Compression menu: The program cannot be broken by any invalid inputs. Invalid inputs would be invalid file paths (when requested) and not selecting a valid digit option from any of the menus. Choosing an invalid response will result in the user being requested to input another response. This will be repeated until they input a correct response.
2. Allow users to view the compressed text for dictionary data compression. Users should be able to store the compressed text in a suitable text file. A suitable text file must end in .txt and it must be on a valid location on the device. The dictionary should also be stored in a suitable file. The dictionary and the compressed text must be accessible to the user.
3. Allow users to decompress the text. Then the user can then save the decompressed text in a suitable text file. A suitable text file must end in .txt and must also be accessible on the device which it is being stored on. The user must also be able to access the file which holds the decompressed text. The decompressed text must match the original text with 100% accuracy. This will be tested by making sure that the key and the frequency distribution of the original text is equivalent to the decompressed text’s frequency distribution and key.

**Statistics objectives**

1. Create a robust Statistics menu: The program cannot be broken by any invalid inputs. Invalid inputs would be invalid file paths (when requested) and not selecting a valid digit option from any of the menus. Choosing an invalid response will result in the user being requested to input another response. This will be repeated until they input a correct response.
2. Allow users to view some statistics basing the performance on both algorithms. This will include the file size percentage reduction as well as the time taken to compress the piece of text. This will be checked manually by totalling the size of the compressed text with its dictionary/tree using a calculator.

Within the time I have had to complete the project I have managed to create a program which will:

* Successfully compresses and decompresses text using the Huffman Coding algorithm where the decompressed text is equal to the original text
* Successfully compresses and decompresses text using Dictionary Data Compression where the decompressed text is equal to the original text

Allow users to store and view the compressed text and decompressed text in notepad. Users to create new files and new folders when they are saving text. It also creates a suitable file for storing the dictionary and the tree for both algorithms.

* Allows the users to view the key and frequency distribution of word/characters in the text.
* Allow users to view some statistics on how well they compress text including percentage file size reduction and time taken to compress the text
* Handles any invalid inputs such as incorrect file paths and incorrect responses in each of the individual menus in the program.

**Optimisations for the project**

* Even though a quicksort being relatively efficient for sorting the frequencies of words/characters in the text, I could use a binary heap which is much faster. I would use this for both dictionary data compression and Huffman coding.
* Despite my program successfully compressing and decompressing text, sometimes the “compressed” file was bigger and, in some cases, significantly bigger. Unfortunately, this was due to the inability of being able to use binary serialization. I resorted to storing the tree/dictionary in a text file. This was quite in-effective for some pieces of text as integer values were stored as 4 bytes whereas they would have been stored as 1 using serialisation. For files which have a relatively small key and a large proportion of text, both algorithms will make the compressed file smaller.
* Although the variables in my program being effective in the program, there are a lot of data structures particularly the Dictionary data compression part. I could perhaps create a new structure or even a class which would manage all these structures in a single variable. This would enhance the readability of my code when looked at by another person.
* Finding a way of improving the calculation of the percentage of file size reduction. Currently, I have created a rough estimate and improving the calculation would make my program more accurate.

Seeing with what my program accomplishes and looking back on my original objectives, **I would say that I have met all the requirements of my project** however there are optimisations for this project.

## Supervisor’s evaluation

Me: Martin, are you satisfied with the project’s outcome?

Martin: Overall, I am satisfied with the project. It successfully compresses and decompresses text, and the compressed and decompressed text can be stored in suitable locations. The user-interface is robust just as expected. I would say that the project meets all the initial objectives. However, I do have some improvements which I would like to discuss with you.

Me: What improvements would you like to discuss?

Martin: The program uses a console display so a solution would be to make the project graphical as at times it sorts of felt a bit clunky in some places and unclear to how the user was supposed to use the program. I do recall you stating that graphics were not the goal of the project, but it would also make it clearer on how to navigate the program and therefore enhancing the user experience.

Me: Thank you for the feedback. As for the graphics of the project, if I was to implement this, I would also add the additional feature of being able to display the tree whilst it is being built so a full step by step process of creating the Huffman Tree is visualised to the user. I would also create the visualisation of the dictionary data compression too. My project would then also have the purpose of teaching users about text compression and how it works.

Martin: I was also expecting the file size reduction for both algorithms to be bigger. Can you explain why sometimes the files for the compressed text were made bigger.

Me: Due to the inability of using binary serialisation which made the size of the tree/dictionary’s size to be quite significant. I had to use text files as an alternative which reduced the compression percentage.

Martin: Ok, thank you for the clarity.

Overall, I am happy with my supervisor’s evaluation on the project. Their criticisms on the project were what I expected as I agree that there were some limitations in my projects design. However, as my supervisor is **overall satisfied** with my project, I can **re-confirm that my project has met all the intended objectives**.

## Additional Features

Here are some additional features that I would like to add to my project if I had more time available.

* Making the project graphical (as stated earlier) and allowing the users to see a full visualisation of the Huffman tree being built as well as the dictionary for dictionary data compression
* I could create a database using SQL. This is so I could hold the tree/dictionary for multiple pieces of text. Currently, the project can only hold the tree for the last binary file saved and the dictionary for the last piece of text saved using dictionary data compression.
* Another improvement would be finding a way so I can access files directly from applications such as Microsoft word instead or with addition to notepad. This would enhance the scalability of my project as typically most people use applications such as word for creating text-based documents.
* I could also implement more lossless compression algorithms furthering my investigation. This would take a lot more time and research, but it would be interesting to see the investigation of other text compression algorithms.
* I could expand the projects scope by compressing other forms of media such as photos or even videos. This would possibly require more complex algorithms to compress the data.

# Research Links

* **Huffman Coding Algorithm History** - <https://en.wikipedia.org/wiki/Huffman_coding>
* **Hoare’s partitioning algorithm** - <https://www.geeksforgeeks.org/hoare-s-partition-algorithm/>
* **Quick sort algorithm** - <https://www.geeksforgeeks.org/quick-sort-algorithm/>
* **Writing to a binary file -** <https://www.tutorialspoint.com/vb.net/vb.net_binary_files.htm#:~:text=The%20BinaryWriter%20class%20is%20used,methods%20of%20the%20BinaryWriter%20class>.
* **Huffman Tree Generator-**<https://www.csfieldguide.org.nz/en/interactives/huffman-tree/>
* **Huffman Coding and Shannon-Fano algorithm Generators-** <https://planetcalc.com/2481/>
* **Reading and writing to text files** - <https://learn.microsoft.com/en-us/dotnet/api/microsoft.visualbasic.filesystem.fileopen?view=net-9.0>
* **Creating files and directories in vb.net -** <https://learn.microsoft.com/en-us/dotnet/visual-basic/developing-apps/programming/drives-directories-files/how-to-create-a-file>
* **Binary Serialisation -** <https://learn.microsoft.com/en-us/dotnet/standard/serialization/>
* **Extract Size of a given file -** <https://learn.microsoft.com/en-us/dotnet/api/system.io.fileinfo.length?view=net-9.0>
* **How to use string builder -** <https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/strings/how-to-create-strings-using-a-stringbuilder>
* **Shannon-Fano algorithm -** <https://en.wikipedia.org/wiki/Shannon%E2%80%93Fano_coding>