CSCI 312 Huffman Encoding Project: Design Document

Lucas Furlong

Davenport University

CSCI 312 Huffman Encoding Project: Design Document

**Overview**

This program will take an input and output filename from a user and compress it. The user can also choose to decompress the file using the same inputs. This document will go over the components of the program, the processing of the input, and the outputs from the program. This document will also include the testing of the program.

**BinaryTree Component**

|  |
| --- |
| BinaryTree |
| -c: integer  -root, current: BinaryTreeNode<T>  -codes: List<Tuple<T, string>> |
| +CompareTo(object): integer  +Insert(BinaryTreeNode<T>, Relative enum): boolean  +Insert(T, Relative enum): boolean  +BuildTable(string, BinaryTreeNode<T>)  +InOrderList(BinaryTreeNode<T>)  +getCodes(BinaryTree<T>): List<Tuple<T, string>> |

**Fields**

***c***

This field represents the count of the tree.

***root***

This field represents the root node of the current.

***current***

This field represents the placeholder for the current node of the tree.

***codes***

This field represents a universal variable that holds a list of the encoding codes for the characters.

**Methods**

***CompareTo***

This method compares objects being passed in to BinaryTree root values.

***Insert(node)***

This method inserts into the tree based on a BinaryTreeNode.

***Insert(data)***

This method inserts into the tree based on data being passed in.

***BuildTable***

This method builds the encoding table.

***InOrderList***

This method lists the tree out using in order traversal.

***getCodes***

This method gets the codes from the list that contains tuples to be passed to another class.

**BinaryTreeNode Component**

|  |
| --- |
| BinaryTreeNode |
| -v: T  -left, right: BinaryTreeNode<T>  -coding: string |
| +isLeaf(): boolean |

**Fields**

***v***

This field represents the value of the BinaryTreeNode.

***left***

This field represents the left child of a BinaryTreeNode.

***right***

This field represents the right child of a BinaryTreeNode.

***coding***

This field represents the encoding value for the current BinaryTreeNode.

**Methods**

***isLeaf***

This method checks to see if the current node is a leaf node.

**CharacterFrequency Component**

|  |
| --- |
| CharacterFrequency |
| -\_ch: char  -\_freq: int |
| +Increment()  +GetHashCode(): int  +Equals(object): boolean  +ToString(): string  +CompareTo(object): int |

**Fields**

***\_ch***

This field represents the character stored in the object.

***\_freq***

This field represents the character frequency stored in the object.

**Methods**

***Increment***

This method increments the frequency.

***GetHashCode***

This method gets the hashcode of the character.

***Equals***

This method checks if an instance of an obj is equal to a character frequency object.

***ToString***

This method overrides the ToString method to write out the object to a designed string.

***CompareTo***

This method compares a passed in object to character frequencies.

**CompressionResult Component**

|  |
| --- |
| CompressionResult |
| -\_original\_file\_size: long  -\_compressed\_file\_size: long |
| +ToString(): string |

**Fields**

***\_original\_file\_size***

This field represents the original file size.

***\_compressed\_file\_size***

This field represents the compressed file size.

**Methods**

***ToString***

This method returns an overridden string that writes out the original and compressed file sizes to a custom string.

**Encoding Component**

|  |
| --- |
| CompressionResult |
| -c: char  -code: string |
| +ToString(): string  +Equals(object): boolean  +GetHashCode(): int |

This component is not utilized. However, if code it changed to use it later, it is still important to show what the component would look like.

**SortedLinkedList Component**

|  |
| --- |
| SortedLinkedList : LinkedList<T> |
| +Add(T) |

**Methods**

***Add***

This method utilizes an algorithm to add an element in ascending order based on whatever type of that element is.

**Main Program Methods**

**TurnBitOn**

This method utilizes bitwise operators to turn a specific bit on in a byte.

**IsBitOn**

This method checks to see if a certain bit is on in a byte.

**CountCharacters**

This method counts the characters in a file and then sets them to an array.

**BuildOrderedList**

This method creates a list of BinaryTree roots which orders the list based on the frequencies of the characters.

**BuildTree**

This method creates a BinaryTree utilizing the ordered list from the previous method.

**MakeTable**

This method calls the methods from the BinaryTree class to get the encoding table from the tree. This method also writes the table to a file.

**Compress**

This method compressed the file utilizing the bit methods and a compression algorithm.

**BuildTreeFromFile**

This method builds a binary tree from the table file.

**Decompress**

This method decompresses the file utilizing the tree from the previous method.

**Input**

This section will cover the inputs that are permitted by the user and processed by the program.

**Arguments**

The four arguments that can be passed into the program consist of the program mode, the input file, the output file, and the table file.

***Program Mode***

The program mode is the first argument in the set of four arguments that you pass in. It can only be set to “C” or “D.” C means that you are compressing the file and D means that you are decompressing the file.

***Input, Output, and Table file***

The three types of files that can be passed in are the input, output, and table file. These are the next three arguments that can be passed into the command line. The input file must contain data. All three files must be text files.

**Processing**

This section will discuss the processing of each of the program modes. In doing this, it will discuss the methods that are utilized during the processing of each program mode.

**Compression: Methods**

This section will cover the compression program mode.

***CountCharacters***

This method counts the input file byte by byte and adds each character from the file into an array of Character frequency objects.

***BuildOrderedList***

This method instantiates a SortedLinkedList of BinaryTree objects which contain CharacterFrequency object. It then iterates through the previous character frequency array and for each character, it makes a new binary tree root, inserts it as a root, and adds it to the list.

***BuildTree***

This method builds a binary tree from the previous list. It removes the first two nodes of the list, combines the frequency of the characters contained in the nodes, makes a new node with that frequency, and then sets first two nodes and children of that node. It splits them based on the frequency. The lower frequency will go the left child and the higher to the right child. After combining, it starts at the beginning and does the whole loop over until there is one root node

***MakeTable***

This method utilizes the getCodes method from the BinaryTree class to receive the codes for each character in the tree. That method calls the BuildTable method from the same class. The BuildTable method traverses through the tree and adds the code to each character in the tree. After receiving the encoding table, the program then writes the table to a file. It writes the total amount of characters at the beginning utilizing a 4-byte integer and then writes the characters and their frequencies in this format. (Character)(number of numbers in the encoding values written as a byte)(Encoding value for that character).

***Compress***

This method takes the encoding table, the input file, and the custom output filename. First it opens the input file to read and creates a file to write to using the output file name. It initializes a bit position integer, a current code string, a byte, and reads the first byte from the file. Now, while the end of file is not reached, it checks the current byte and sets it to the current code string with the bit values from the encoding table. It then iterates through each number in the current code string and checks if it is a 1 or 0. If it is a 1, then you turn on that bit using the bit position integer. If it is a 0 then decrease the bit position integer. After the end of the 8 bits have been reached, it checks the final position and then writes a new byte to the output file. It does this iteration for each character in the input file.

**Decompression: Methods**

This section will cover the decompression program mode.

***BuildTreeFromFile***

This method takes the table file and build an encoding tree from the file. First it initializes a tree and then counts the first four bytes, so it knows the file length and sets it to a variable. It then initializes two new BinaryTreeNodes, one being the node and one being the root. It then sets the root to the node and enters while loop which checks for end of file. This then initializes a variable which is set to a new CharacterFrequency object and makes the first character read the character passed into the variable. The next byte read will then be the length of the number of numbers to read, so the program knows how many spaces to go through before it hits the next character. The program then enters a for loop which checks if the next byte is a 1 or 0. If it is a 1 then you traverse left. If the node to the left does not exist, then create one with a BinaryTreeNode object. If it does, then traverse to the left down the tree. This is the same for 0’s but it goes to the right. After reading through the entire file. The program should then set the file length to the root node value. It then returns that entire tree of BinaryTreeNodes as a BinaryTree object.

***Decompress***

This method decompresses the compressed files using the tree from the previous method. First it opens the file to read and creates the file to write to. It reads the first byte and the sets it to a variable. The program then initializes the character counter, the bit position, sets the file length to a variable, and then sets the current node to the root node. The program then enters a while loop which checks for the end of the file and enters a while loop which checks if the bit position is more or equal to 0. The program then asks if the character counter is less than the file length. If it is, then check to see if the current bit is on of that current byte. If it is, then traverse left down the tree. If it traversed to a leaf node, then write that character in the leaf node to the decompressed file, subtract the bit position by one, add to the character counter, and set the current node to the root node. If it is not a leaf node, then subtract from the bit position. This is the same if the current bit is off, but instead of traversing left, it traverses right down the tree. After 0 in the bit position has been reached and it has traversed the tree, read the next byte, and reset the bit position variable. If the character counter is at the file length, then break out of the first while loop.

**Output**

This section will cover the output that the program will return when certain modes are executed

**Compression**

After the user passes in the four arguments for compression, the input file, output file, and the table file, the program will return all those files. The input file will return as normal, the output file will return as the compressed file, and the table file will return with the encoding table of the characters.

**Decompression**

After the user passes in the four arguments for decompression, the compressed input file, the output file, and the corresponding table file, the program will return all those files. The compressed input file will return as normal, the output file will return as the decompressed output file, and the table file will return as the normal file the user passes in.

**Testing**

This section will cover the testing scenarios of the program, describe what each does, what is should return, and what it does return.

|  |  |  |
| --- | --- | --- |
| Scenario | Description | Pass/Fail |
| 1 | Test for four arguments | PASS |
| 2 | Test for empty input file | PASS |
| 3 | Test for incorrect program mode argument | PASS |
| 4 | Test if last three arguments are text files | PASS |

**Scenario One: Test for four arguments**

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/Output |
| 1 | Open command line and prepare to run program |  |
| 2 | Type .exe file and pass in less than or more than 4 arguments |  |
| 3 | The program should return an error message that says, “Please pass in program mode (C for Compression and D for Decompression), input filename, custom output filename, and custom table name... EX: C read.txt write.txt table.txt” and “Table filename must be saved to decompress file” |  |
|  |  |  |
|  |  |  |
|  |  |  |
| EXPECTED OUTPUT | | Error message from step #3 |
| ACTUAL OUTPUT | | Error message from step #3 |
| RESULTS | | PASS |

**Scenario Two: Test for empty input file**

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/Output |
| 1 | Open command line and prepare to run |  |
| 2 | Type .exe file and pass in input file with no data |  |
| 3 | The program should exit and return an error message that says “Input file must contain data” |  |
|  |  |  |
|  |  |  |
|  |  |  |
| EXPECTED OUTPUT | | Error message from step #3 |
| ACTUAL OUTPUT | | Error message from step #3 |
| RESULTS | | PASS |

**Scenario Three: Test for incorrect program mode argument**

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/Output |
| 1 | Open command line and prepare to run |  |
| 2 | Type .exe file and pass in argument for program mode that is something other than “C” or “D” |  |
| 3 | The program should exit and return an error message that says “Program mode must be C or D (C is Compression and D is Decompression” |  |
|  |  |  |
|  |  |  |
|  |  |  |
| EXPECTED OUTPUT | | Error message from step #3 |
| ACTUAL OUTPUT | | Error message from step #3 |
| RESULTS | | PASS |

**Scenario Four: Test if last three arguments are text files**

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/Output |
| 1 | Open command line and prepare to run |  |
| 2 | Type .exe file and pass in arguments for any of the last three that are not .txt files |  |
| 3 | The program should exit and return an error message that says “The input file, output file, and/or the table file must be a .txt file” |  |
|  |  |  |
|  |  |  |
|  |  |  |
| EXPECTED OUTPUT | | Error message from step #3 |
| ACTUAL OUTPUT | | Error message from step #3 |
| RESULTS | | PASS |