**OVERVIEW**

This program will read from a text file and create an output file with the unique characters, ASCII value, and frequency of character. The program will also use a Huffman Encoding algorithm to compress the input file and write to another output file. Finally, the program will create another output file that will have the decoded original information from the input file provided. A binary tree structure will be used to encode and decode the input file, and encoded file.

**INSTRUCTIONS**

The user will specify the location of the input file they would like to encode and decode in the program arguments of the program as well as an output file location. If no input file or output file location is specified, the default text file included with the program is wap.txt and Count.txt located in the programs source folder. Once program parameters are set the user can now run the program which will produce two output documents called decoded.txt and encoded.txt. Encoded.txt will provide the user with a text file of binary numbers that have been encoded using the Huffman Encoding algorithm. Decoded.txt will contain the original input file contents from wap.txt or another specified file path provided by the user.

**SCOPE**

This program will read an input file character by character and produce three output text files. The output text files that will be created are Count.txt, encoded.txt, and decoded.txt. Count.txt will contain the unique characters found in the input file, their ASCII value, and the frequency they appear in the input text file. Encoded.txt will represent characters and their frequency converted to binary using the Huffman Encoding algorithm. Decoded.txt will contain the same information found in the input file.

**PROCESSING**

The process requires an input file to be located within a specified location found in the arguments of the project, and an output file location as well. The process will also create two output files encoded.txt and decoded.txt.

This process will create a linked list of character frequency objects. After the linked list of character frequency objects is created the process will require a binary tree to encode and decode.

The overall processing of the file will be as follows. First, a character will be read from the file. Next, the character will then be located in the array and the count for that character will be incremented. This process will be repeated until each character in the input file is counted.

Processing Outline for Count.java

1. If file does not exist

a. Display message error message

b. Exit

2. If file cannot be read from the given path

a. Display message #2

b. Exit

3. Initialize Linked list, buffered reader, and local variables

4. Read characters in file one by one

1. While buffered reader is not -1
   1. Character equals character input
   2. Call method to handle new character from reader

5. Create file instance to write output to

1. If arguments length equals 2 open files in argument one
2. Else create the output file in specified location

7. Create file output stream and set file output to writable

8. For each character instance in total unique characters

1. Call sort linked list method
2. Write character frequency objects to file

9. Close the output file and buffered reader

10. Create Huffman tree and pass character frequency objects through it

a. Create output file encoded.txt to send compressed code to

b. Create file output stream and pass output file into it

11. While buffered reader is not -1

a. Take character from reader and pass it into get code method in Huffman class and write to file

12. Close file output stream

13. Create new output file decompressed code

14. Create buffered reader instance

15. Get tree root of binary tree of character frequency objects

16.While buffered reader is not -1

a. If character is 0 get left

b. Else if character is 1 get right

c. If at a leaf get the character and write to file and set current node back to the root

17. Close file stream and buffered reader

18. updateCharFrequency method returns int

a. Create instance of character frequency object

b. Iterate through each instance of character frequency objects in linked list

c. If character frequency instance equals any in linked list call increment method

d. If character does not exist, add it to the linked list of objects and increment frequency by one

19. sortLinkedList method

a. Calls collections sort method

b. implements a compare method which compares character frequency objects by frequency

Processing outline for Huffman.java

1. Huffman constructor

2. Initialize data structures

a. Linked list character frequency

b. Binary tree root node

c. Linked list Huffman codes

d. String code

3. Constructor that takes linked list of frequencies

a. Frequencies

b. Create tree

4. Method to create a list of char frequency Binary tree nodes returns nodes

a. Linked list binary tree nodes

b. For CharFreq frequencies

i. Create binary tree node

ii. Place into list of nodes

5. Method to create tree

a. Linked list binary tree node character frequency nodes

b.While nodes size is greater than 1

i. Sort binary tree node of character frequencies by frequency

ii. Get first two nodes in list

iii. Create new node combining two nodes taken and set left and right

iv. Add new node back to the list of nodes

v. Set root node to first entry of nodes list

vi. Call generate codes method

6. Method to get root tree node

7. Generate code method

a. If node is null return

b. If left and right node are null and character is c, code equals s

c. Generate code by getting left node string s, 0, and character c

d. Generate code by getting right node, string s, 1, and character c

8. Method to get code

a. For HuffmanCode codes

i. If HuffmaCode character equals character

a. Code equals Huffman code

b. return code

9. Method \_get code

a. GenerateCode with root, Blank, character

b. Return code

10. Method to generate all codes

a. Create a list of all codes

b. For character frequencies cf frequencies

i. Get code for specific character

ii. Add code to linked list

**DATA**

***Input File***

Input file required to be a text file that is read character by character and any of file formats will not be handled by the program.

***Output Files***

The count.txt output file will be a text file created or updated by the program which will contain the character, its ASCII value, and its frequency.

For example:

a (97) 12

b (98) 6

c (99) 4

Encoded.txt output file will be created or updated by the program which will contain the new binary representation of the Huffman Encoded characters

For Example:

The string “Hello” would become:

11011100101001

Decoded.txt output file will be created or updated by the program which will contain the original message that matches the input file text.

For Example:

The Huffman Encoded 11011100101001 binary would become:

Hello

**User Messages**

These messages are displayed to the user.

|  |  |
| --- | --- |
| **Message Number** | **Message Text** |
| 1 | Argument file path (wap.txt) does not exist. Specify file location in the arguments of the project and re-run. |
| 2 | File name (wap.txt) cannot be read from. |

***Data Structures***

A linked list of character frequency objects. Size of linked list is determined by number of unique characters found in input file.

Character frequency object instances are stored within the linked list will be initialized as follows.

 The unique character found in the input file

 The frequency the character occurs within the input file set to zero when initialized

Linked list binary tree will contain characters and their frequencies.

**COMPONENTS**

A character frequency class is used to hold characters and their frequency in the input file. Below is the UML diagram which demonstrates its structure.

|  |
| --- |
| CharFreq |
| -character: char  -frequency: int |
| +getCharacter(): char  +setCharacter(int character: char)  +getFrequency(): int  +setFrequency(in frequency: int)  +increment()  +equals(): bool  +toString(): string |

A binary tree (BTreeNode) class will be used to set tree nodes.

|  |
| --- |
| BTreeNode |
| -left: T  -right: T  -el: T |
| +getElement(): T  +setElement(in el: T)  +getLeft(): T  +setLeft(in n: T)  +getRight(): T  +setRight(in n: T)  +isLeaf(): boolean |

***Properties***:

**CharFreq:**

**character:** Represents the character

**frequency:** Represents the frequency of the character in the input file, initialized as 0.

**BTreeNode:**

**left:** Represents the left node in the binary tree

**right:** Represents the right node in the binary tree

**el:** Represents the current element in the binary tree

***Methods*:**

**CharFreq:**

**CharFreq ():** The default constructor will initialize the properties as follows.

Character;

frequency = 0;

**getCharacter()**: Returns the value of **character**. **setCharacter(char character)**: Sets the value of **character.**

**getFrequency()**: Returns the value of **frequency**.

**setFrequency(int frequency)**: This method assigns a value to **frequency**. This method will only assign the value if the new value is greater than or equal to zero.

Method will set the value of frequency.

**Increment()**: This method will increment the frequency by 1 when called.

**Equals(CharFreq objectToCompare)**: This method will return true if two character frequency objects contain the same value in **character.**

**ToString()**: This method will return a string in the format of character (ASCII Code of character) and frequency. This format is defined in the Output File section of the DATA section.

**BTreeNode:**

**BTreeNode():** Constructor that initializes left and right to null and sets el

**getElement():** Returns the value of el

**setElement(T el):** Sets value of el

**getLeft():** Returns the value of left

**setLeft(BTreeNode<T> n):** Sets the value of left

**getRight():** Returns the value of right

**setRight(BTreeNode<T> n):** Sets the value of right

**isLeaf():** Checks for current node to be a leaf

**Count:**

**updateCharFrequency(LinkedList<CharFreq> charFrequencyList, char character, int totalUniquechar):** Returns total unique characters, updates linked list of character frequency objects with new instances found in input file

**sortLinkedList(LinkedList<CharFreq> charFrequency):** Sorts linked list of character frequency objects

**Huffman:**

**HuffmanCode():** Constructor that initiates character and code variables

**getListOfCharFreqBTreeNodes():** Creates list of character frequency binary tree nodes

**createTree():** Method that creates a Huffman tree

**generateCode(BTreeNode<CharFreq> node,String s,char c):** Generates code

**getCode(char c):** Method generates codes

**\_getCode(char c):** Method to generate code calling generateCode method

**generateAllCodes():** Method that generates all code and stores code in linked list

**TESTING**

The table below presents an overview of the testing scenarios and the status of running the scenarios.

|  |  |  |
| --- | --- | --- |
| Scenario | Description | Pass/Fail |
| 1 | Test for non-existent input file | PASS |
| 2 | Test for accurate Count.txt output file | PASS |
| 3 | Test for accurate encoded.txt output file | PASS |
| 4 | Test for accurate decoded.txt output file | PASS |

Each scenario is presented along with the steps required to execute the scenario.

*Scenario #1-* Test for non-existent input file

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/ Output |
| 1 | Remove wap.txt from source folder |  |
| 2 | Run program |  |
| 3 | Check console for error message | Error Message One |
| EXPECTED OUTPUT | | Error Message one |
| ACTUAL OUTPUT | | Error message one |
| RESULTS | | PASS |

*Scenario #2-* Test for accurate Count.txt output file

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/Output |
| 1 | Run program |  |
| 2 | Check source folder for Count.txt | Count.txt |
| 3 | Open Count.txt and check file for accuracy of format and information | Character(ASCII Value) Frequency |
| EXPECTED OUTPUT | | Count.txt file  Accurate character frequencies and ASCII value |
| ACTUAL OUTPUT | | Count.txt file  Accurate character frequencies and ASCII value |
| RESULTS | | PASS |

*Scenario #3-* Test for accurate encoded.txt output file

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/ Output |
| 1 | Run program |  |
| 2 | Check program folder for encoded.txt |  |
| 3 | Open encoded.txt file |  |
| 4 | Check encoded.txt to have been converted to binary |  |
| EXPECTED OUTPUT | | Text file with a string of binary |
| ACTUAL OUTPUT | | Text file with a string of binary |
| RESULTS | | PASS |

*Scenario #4-* Test for accurate decoded.txt output file

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/ Output |
| 1 | Run program |  |
| 2 | Check program folder for decoded.txt |  |
| 3 | Open decoded.txt file  Open wap.txt |  |
| 4 | Compare decoded.txt output file to wap.txt input file for consistency |  |
| EXPECTED OUTPUT | | Decoded.txt file  Consistent input information from wap.txt that matches decoded.txt output file |
| ACTUAL OUTPUT | | Decoded.txt file  Consistent input information from wap.txt that matches decoded.txt output file |
| RESULTS | | PASS |

**Sample Output for Test Scenario #2**

Output from the Count.txt file

h(104) 1

e(101) 1

l(108) 1

l(108) 1

o(111) 1

**Sample Output for Test Scenario #3**

Output from the encoded.txt file

11011100101001

**Sample Output for Test Scenario #4**

Output from the decoded.txt file

hello