Huffman Encoding Project

Purpose

The purpose of this program is to be able to compress an ASCII file and to decompress a compressed ASCII file

Scope

This program reads in files from a file, create a Huffman encoding tree, an encoding table, and then compress the file based on the table.

The program will also read a compressed file and recreate the Huffman encoding tree, and then decompress the file into the original file.

Inputs:

In the program the program will ask the users whether they want to decode or encode. After entering what the user wants they then are asked to input the absolute file path that they want to compress or decompress.

Output:

The program will output 2 files when it compresses and 1 file it decompresses. The two files for the compress is the file they inputted with _Encode.txt which holds the Frequency of characters. The second is a binary file with the name of the original file name with _EncodedText.bin added at the end which holds all the encoded data. The decompression outputs a file with the original file name with _Decode.txt at the end which contains the decoded text or the contents of the original file.

Encoding Processes:

The processes of encoding the file is this:

Read characters from the File

Build the Huffman encoding tree

Build the Encoding Table

Encode the File

Each part is broken up into more detail in the next section where we discuss the pseudocode for each section

Reading characters from the File:

The goal of this function is to create an ordered linked list of Character Frequencies from the file given. It takes in a string filePath which is the absolute path of the file the user inputs earlier in the program

Create a character Frequency Linked List

While not at end of file

    Read Character

    If Character is not in the list

        Add it to the list

    Else

        Increase the frequency of the character

Order the linked list in ascending order by the frequency the characters appeared in

Building the Encoding Tree:

The goal of this function is to create a LinkedList of BinaryTrees that hold Character Frequencies. The function takes in a LinkedList of Character Frequencies and outputs a LinkedList of BinaryTrees of Character Frequencies. Pseudocode is as follows:

Turn all the LinkedListNodes of CFs into BinaryTree roots and add them back into a linked list

While the LinkedList of BinaryTrees of Character Frequencies count is greater than 1

    Take out the first two elements

    Sum the two frequencies together

    Create a new Character Frequency with the character set to null and the frequency the sum

    Set this new Character Frequency as the BinaryTree root

    Add the first element taken out as the left child

    Add the second element taken out as the right child

    Put the new tree back into the list in ascending order by the frequency

Creating the Encoding table:

To create the encoding table we are going to call the Encode method from the BinaryTree class which uses an in order traversal to traverse through the tree. The Pseudocodes is as follows:

If TreeNode is not null

    Add a 0 to the encoding string

    Go right

If TreeNode is a leaf node

Create an encoding object with the character as a parameter and the encoding

Add the encoding object to the linked list of encoding

string as another parameter

Add a 1 to the encoding string

Go left

If the encoding length is greater than 1

Remove the last character from the encoding string

Else

Remove the last character from the encoding string

Encoding the File:

The goal of this function is to actually encode the file from the encoding table given. The function takes in a LinkedList of encodings, a filepath to the file it's writing to, and a StreamReader to read the original file. The pseudocode is as follows:

Open the file in a BinaryWriter

Read the first character from the file

While not at the end of the file

Create a new encoding with the character read for the character and null for the encoding string

Search through the encoding linked list and get the encoding string

While we haven't reach the end of the encoding length

If the character at the encoding position is == "1"

Turn on the bit

Go to the next bit

If the next bit is 0

Write the byte to the file

Reset the byte

Set the bit position to 7

If the character at the encoding positon is == 0

Go the next bit

If the next bit is 0

Write the byte to the file

Reset the byte

Set the bit position to 7

Read the next character

Write the byte

Decoding process:

The processes for decoding the file is as follows:

Open the _Encoded.txt file

Read a line

Split the string based on a “,” and add them to a Linked list of character frequencies

Rebuild the tree from the Linked list

Create the _Decode.txt file

Open the _EncodeText.bin in a binary Reader

Open the _Decode.txt in a StreamWriter

Read a byte

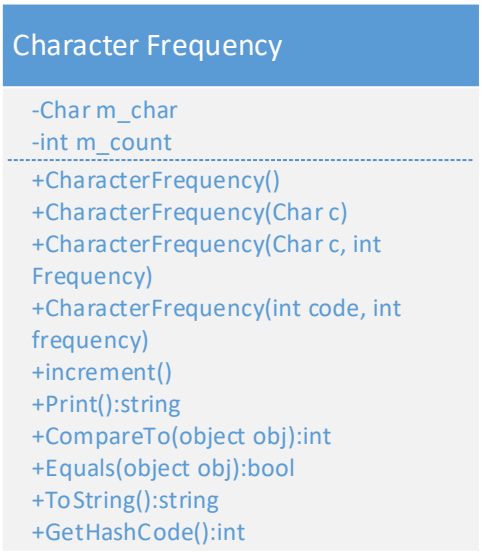
While were not the end of the file

Classes

This program utilizes the following classes:

Character Frequency:

The Character Frequency class is used to keep a count of all the characters that have appeared in the ASCII file that is going to be compressed.



The two data contained have the following constrains: the m_char is to be an ASCII value, the m_count can only be 0 or greater. It has fours constructors that take in different parameters. The default constructor will default the m_char to null or \0 and the m_count to 0. The constructor which takes in a Char will set the default value to the character given and set the count to 0. For both Constructors that take two parameters both set the Frequency if the frequency received is not less than 0. The constructor that takes in an int can only have a number value that is on the ASCII table.

The increment method is used when you find a repeated character in the file, it'll add 1 to the m_count.

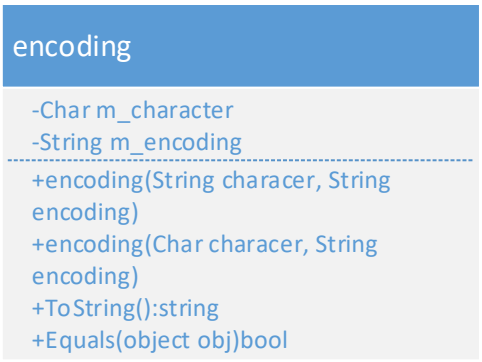
The print() method is used when we are writing the character frequencies onto a file in order to recreate the tree for decompressing. It prints out in the following format: The numeric ASCII value of the character, a comma to be used a delimiter, and the number of times the character appeared, then a new line. Or (int) m_char, m_count\n.

The toString() method is overridden in the character frequency class for when we're reading data from the tree or if we needed to print data from the list. Its written in this format: The character, a comma, the count, and a new line.

The CompareTo, Equals, and GetHashCode are not used in this project but are generic functions overridden if we need to use the character frequency to be ordered in something like a SortedLinkedList.

Encoding:

The encoding class is used to store what the encoded value for the character when we're compressing the file. It's created while we are traversing through our tree.



The two variables contained in the encoding class have the following constrains: m_character should only be an ASCII character and m_encoding should be a string that only contains either "1" or "0"s.

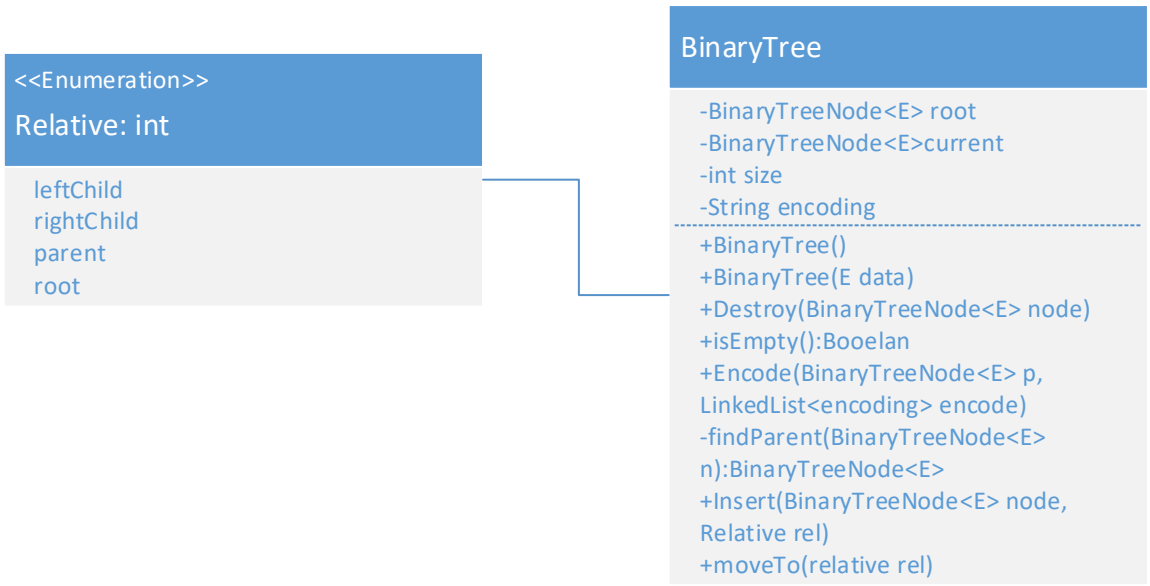
The encoding constructors both take in two parameters. The first encoding constructor is used when the encoding table is being built from the tree since the tree would only let me use the toString method from the character frequency so I have it take the character from the first position from the character array (string) and set it as that.

The encoding class has a toString method that is overridden that is not used but puts out the data in the following format: the character, a comma, the encoding values for the character.

The Equals method is overridden and is used when we are compressing the file in order to find the encoding data for the character. It's your typical Equals method where it looks to see if the m_character is equal to a given object.

Binary Tree:

The BinaryTree class is used to create the Huffman Encoding tree to create the encoding table which will help compress and decompress the file being used.



The class comes with an enum which tells the tree what is relative to it like the left child or the right child.

The BinaryTree constructor that takes in 0 parameters is used in this program to build the tree, the other constructor is not used at all.

Destroy is a method that is used when we want to destroy a node from the tree given that the node exists for this program we don't need to destroy any tree nodes

IsEmpty is used to tell whether or not the tree has data or not, for this program it is not used

The Encode method is used to actually encode the tree.

This method uses an in order traversal method to generate the encoding table

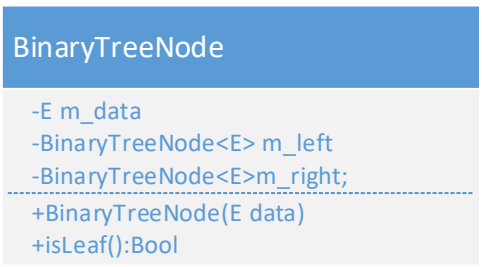
The findParent method is not used in the program but is used to find the parent node of any given BinaryTreeNode

The insert method is used to help build the Huffman Encoding tree. It takes a BinaryTreeNode and where it should go in relative to it and adds it to the tree.

The moveTo method is not used in this project but would be used to move to a given position of a binary tree like its left or right child, or to the root node.

BinaryTreeNode:

The BinaryTreeNode class is used to store data and be the building blocks for the BinaryTree class to build the Huffman Encoding tree. It's a generic class that can take in data it is told to take in



The isLeaf method is used in our decoding and encoding method to determine whether the current node doesn't have any children connected to it.

Testing:

For testing where this program can crash there are a few conditions: The user enters in the wrong absolute path or the file doesn't exist. When decompressing the _encode or _encodeText files don't exist or the user attempts to compress an empty file.

To test if the program doesn't crash when a user attempts to compress an empty file I created a file called empty and attempted to compress it. In my compression logic I check to see if the file is empty and if it is it won't compress. This logic worked and the program told the user it can't compress an empty file which indicates a passing test.

To test if the program doesn't crash when a user attempts to decompress a file that doesn't have its _encode or _encodeText files I used the empty text file again and attempted to decompress it. There is logic that checks whether each file exists and if it doesn't it won't decompress. When I told it to decompress the empty file it told me that it can't decompress a file when the encode file is missing which indicates a passing test.

To test if the program crashes if the user enters in the wrong file path I loaded up my program and attempted to compress. I told it to compress text.txt without a path which is not an absolute path and text.txt doesn't exists. The program told me that it can't open the file and to reenter another command. I also tested this with the decompress side which gave out the same results.

To test to see if the program builds a Huffman tree correctly I created a file and entered in text with the following frequencies. 25 A's 25 B's 10 C's 10 D's and 15 E's. I then hand built what the tree should look like from my algorithm and then checked to see what the tree built from the debugging menu. A comparison between the two showed that it matched which means that it builds the tree correctly.

To test to see if the program actually compresses the file. I created a file with the following contents in it: "This is a test document to make sure that my Huffman encoding is working correctly, please ignore

the contents" (without quotes). I compressed the file from my program then I a simple comparison between the .bin file to the original text file sizes from the file properties. The original file was 114 bytes while the .bin file was only 67 byes which indicates that it compress the file.

To test to see if the program decompresses correctly I took the same file used in my compression test and decompressed it. I then checked the _Decode.txt which is the output results of decompression and checked to see if it was the same size, and if the contents matched. After the check both matched indicating the program decompresses a compressed file correctly.