# Overview

The purpose of this application is to compress and decompress text file using the Huffman Encoding process. The program will read through the file and count the characters, then use this information to create a binary tree and a character encoding table. Using the table and while reading through the input file again the program will take the path for each character and write that to a byte when said character appears in the input file, writing the byte to the output file when all bits have been assigned. The table will then be written to the beginning of the output file for later use in the decompression process. This will lead to a compression rate of approximately 59.6%.

For decompression, the program will take the table that was appended to the start of the compressed file and recreate the binary tree. It will then read each byte from the compressed file and traverse through the tree using the values of each bit, writing to the decompressed file every time it reaches a character in the tree. The user will run this program from the command prompt and provide both the file name and whether they want the file to be compressed or decompressed.

# Assumptions

* Input file is a text file (.txt)
* User runs program on a Windows computer

# User input

The accepted input of this program follows the format below:

“Compression –c {text file}”

or

“Compression -d {bin file}”

Running this from the command prompt means you must specify the program you are running and supply the arguments on the same line. The first augment determines whether you are compressing or decompressing a file, with {-c} and {-d} representing the options respectively. The second argument specifies the path and file name the program will be using.

# Classes

# BinaryTree



The binary tree is the main meat of this program. It is used in both the compression and decompression stages. The tree is made of multiple nodes, one that is the root node that all other nodes branch from, with each node having a left child and right child. Using these properties allows the program to navigate through the tree to either find nodes or write out the path to each node.

# TreeNode



Tree nodes are what fill the binary trees. Each contain an element and have two children, a left and a right. In this program, the CompareTo method is required to compare the tree nodes and create an ordered linked list of tree nodes.

# BinaryPath



The binary path object is used to make the character encoding table. Each object has an element, and the path to said element from within a binary tree. These are used in the main program file and the binary tree class to create a list of BinaryPaths to use as the encoding table.

# OrderedLinkedList



OrderedLinkedList is a custom implementation of the LinkedList data structure that focuses on sorting the contents within the LinkedList. This is used during the compression process to keep an ordered list of tree nodes of character frequencies. It extends IEnumerable to allow the program to iterate through each item in the list.

# CharacterFrequency



CharacterFrequency is an object that contains a character and an associated Int that states the frequency of said character. The method increment is used to increase the frequency by one in case the frequency is be determined after the CharacterFrequency object has been made. It extends IComparable allowing for CharacterFrequencies to compare their frequencies for sorting purposes.

# Compression Process/Huffman Encoding

First in the process, after checking to be sure the input file exist, is reading and counting the characters in the input text file. The program reads each character individually until the end of the file and totals up the frequency of each character found in the file. After wards, all the characters that did not show up in the input file are discarded and the remaining are placed into nodes and added to an OrderedLinkedList. Once the ordered list is made the process of making the binary tree begins. To maintain order of the characters (ordered by frequencies, least to greatest) the tree is constructed within the ordered list and later added to a BinaryTree as the root node. The first two characters from the list are removed from the list, their frequencies as combined and used as the frequency of a new CharacterFrequency object. The two least characters are set as the children of the new combined CharacterFrequency object (least on the left, greatest on the right)., and the new object is then added back into the ordered list. This is looped until there is only one node left in the list, this node is then set as the root of a new binary tree.

Next the tree is used to make the encoding table (a LinkedList of BinaryPaths). The program then reads through the entire tree and finds the path for each leaf node (nodes with no children) using 0 and 1 as directions (0 means go to the right, 1 mean go to the left), then adds the path and character to the encoding table. After the table is created, the number of characters in the input file is written to the output file as well as encoding table itself to be used in the decompression process. The program then starts a loop of rereading the input file until the end, each character read is searched for in the encoding table and its path is written to a byte. Once all bits within a byte are set, or the file has reached its last character, it is then written using a BinaryWriter to the output binary file. The program then alerts the user that their file is compressed to the file “{input file name}-compressed.bin”, and stops.

# Compression Diagram



# Decompression Process

First the program checks to see if the path provided leads to a file, if not display an error to the user and end the program. Otherwise the next step of the decompression process is getting the both the encoding table from the compressed file and the number of characters expected to be in the output file. Using a BinaryReader the table is taken from the input file, split using a series of regex statements, and converted into a LinkedList of BinaryPaths to be used as the encoding table. From the table the tree is recreated, using the bits of each path as directions to move in the tree (0 is left, 1 is right) nodes are created and added to the tree as the program reads through the table, ending with a binary tree that is identical to the binary tree used during the compression process.

Finally, the decompression process really begins. The program begins a loop to read through the compressed file byte by byte after the point where the table and character number where written. Each bit in the byte is used as directions in the tree, 0 moves to the right and 1 moves to the left, and when a leaf node is hit that character is written to the decompressed output file, a counter is advanced by one, and the program moves the current node back to the root node. This is continued until either there are no more bytes in the compressed file to read or the counter is equal to the number of characters that were in the original decompressed file. The program then outputs to the screen that the file is decompressed to “{input file name}-decompressed}.txt, the program then stops.

# Decompression Diagram



# Regex

Regex statements are used during the decompression process to get the table and number of characters to write from the input compressed file. Regex.Split() function was used in this program, because you can have conditional statements in regex statements and we needed to be able to change what character we split on incase that character showed up in the file.

For splitting the table string: “(?(?<=;)\0|;)”

Check if the character before the current one was a semi colon (;), if so don’t match anything, otherwise match semi colons.

For splitting the total characters from the file: “(?(?<=\d)~{3}|\0)”

Check if the character before the current one is a number, if so match 3 tilde (~~~), otherwise don’t match anything

Both use the use of positive lookbehinds, where the regex checks the character that came before the current character and sees if it matches the supplied statement, and changes what character it matches based upon the results.

# Recursive Method and File Checking

The program only needs one file path in its second argument, as the output file name is generated based upon the input file name. We remove everything from the path they user provides to us except the file name (even the extension is removed), we append at the end either “-compressed” for the compression process or “-decompressed” for the decompression process (which can lead to files having both “-compressed” and “-decompress”).

Later when the program is creating the file that it will write to it first it must first run a recursive method to check if the file already exists as we do not want to overwrite files that we do not have permission to overwrite. We pass in the path we want to check first, which it just the output file name we generated before, and it checks if it exists in the exe directory. If it does then we prompt the user if they would like us to overwrite the file in this location, if yes then we overwrite the file otherwise we prompt them with a file browser to choose which location they would like us to save this file to. If they click the cancel button informs them that the program will close if they do not choose a location and is this okay, closing the program if they agree or reopening the file browser if they say no. Once they choose a location we run the same method again on the location they selected to see if the file exists there, running through the exact same steps until either; a. user provides a location without a file to overwrite b. user says overwriting file is okay c. user aggress to close program.

The method returns a path that the file can be written to, but there is one last try catch to check if the user does not have permission to access the folder they provided. If they do not, they the method is run again, otherwise the program continues.