Algorithm Description: Compression Using Huff.c

In order to implement Huffman coding compression, my algorithm implements four separate structures: Node, StackNode, Stack, and BitFile. These structures, as seen at the top of my huff.c source file are used to represent a linked list, or Stack, of StackNodes containing a pointer to the next node as well as a pointer to a binary tree of type Node. This is effectively a linked list of binary trees ultimately converges to a single StackNode that contains a pointer to a combined tree containing all of the characters from the file on leaf nodes; more specifically, the Huffman coding tree that will be used to compress the file. To generate the Huffman coding tree, the following five functions were implemented:

- `void weighCharacters (FILE * fp, int * asciiArray)`
  - Analyzes a file by retrieving individual characters and incrementing an index associated with a specific ASCII value, thus assigning weights to each ASCII character within the file.
- `Stack * genTrees (int * asciiArray)`
  - Generates a Stack of StackNodes that contain single node trees based on the passed in asciiArray. This is done by analyzing all of the non-zero, or weighted, elements of the asciiArray and creating a StackNode that pushed on to the front of the Stack to be returned. The final return value of this function is a Stack that contains a node for each individual character within the file which each contain the letter as well as the weight associated with that specific letter in the file.
- `void findTwoSmallest (Stack * stack, StackNode * * smallest, Stack * * secondsmallest)`
  - Given a Stack of at least size 2, traverses the Stack to determine which two StackNodes contain the smallest weighted trees. This serves as a utility for combineAndRemoveOld by determining the two smallest weighted trees within the Stack such that they can be combined and pushed onto the front of the Stack.
- `StackNode * findPrev (Stack * stack, StackNode * curr)`
  - Given a Stack and a current StackNode, determines the node immediately before the current node as to correct pointers within combineAndRemoveOld. The implementation of this function was a design decision that resulted from the choice of not implementing a doubly-linked list as to decrease memory complexity associated with the algorithm.
- `void combineAndRemoveOld (Stack * stack, StackNode * node1, StackNode * node1)`
  - Given a Stack and the two smallest nodes determined by findTwoSmallest, removes those two nodes, fixes the links around them, combines the two nodes into the left and right of a new tree node, and pushes the new node on the front of the Stack, effectively reducing the size of the stack by one as well as increasing the depth of the tree. Taking these two smallest nodes allows for the characters that are used the least to be put at the bottom of the tree and the "highest priority" characters to be placed near the top of the tree. When called in a loop, this function will reduce the size of the stack to one StackNode that contains a pointer to the Huffman coding tree for the file that is to be used for compression.

After the tree has been determined by the algorithm, it has two tasks remaining: writing the tree in a compact, but recoverable, fashion in the form of a header terminated by a new line character and writing the compressed data to the new file with the ".huff" suffix appended to it. To do so, I utilize four additional functions, with the incorporation of a new structure, BitFile, to allow for reading and writing individual bits to a file by maintaining a working "byte" attribute as well as an "offset" that determines where the next bit goes in the byte. This structure is accompanied by two functions that do the work associated with it, BitFile_writeBit and BitFile_writeByte, which function exactly as their names imply.

Header Design: It is worth noting that a design decision was made to generate a header using post-order traversal of the Huffman coding tree with a bit '1' placed upon arrival at a leaf node followed by the character associated with that node. Additionally, when visiting a non-leaf node, a bit '0' is placed into the header. The header's end is denoted by a single extra '0' bit followed by the length of the file as 8 bit values per digit of length, finally denoting the end of the header to the decompression program by addition of a new line character.

The file composition functions are as follows:

- `void treeToHeaderString (char * * headerstring, Node * huffTree)`
  - Utilizes post-order recursive traversal of the tree to place a '1' followed by the character in question to be added as a node to the tree as a leaf node and a '0' to signify a non-leaf node. The resulting string is placed at the address referenced by headerstring.
- `BitFile * writeHeaderToFile (FILE * fp, char * headerstring, int filelen)`
  - Given the string generated by treeToHeaderString, writes the header bit-by-bit into the file pointed to by fp. This is done by using BitFile_writeBit to write the 0's and 1's within the header as well as the BitFile_writeByte function to write entire characters into the header when a '1' is placed. The header is then terminated by a single extra zero followed by the length of the file as 8 bit values per digit of the length, finally denoting the end of the header to the decompression program by adding a new line character
- `void determineLetterCode (char * bitpattern, char letter, Node * huffTree, int ind, char * * f)`
  - Determines recursively the encoded value of a given letter within the file and assigns its value as a string that is returned in bitpattern. For instance, if a letter was to be encoded as 1101, bitpattern would exit the function pointing to "1101".
- `void writeDataToFile (BitFile * compfile, FILE * fp, Node * huffTree)`
  - Reads the original file one character at a time and determines its bit pattern using the determineLetterCode function. Once the bit pattern is attained, it writes each character that was returned in the string as a bit to the compressed file, compfile, by utilization of the BitFile_writeBit function. When this function exits, the compressed file has been completed and is ready for decompression.

Algorithm Description: Decompression Using Unhuff.c

Decompression of a file compressed by my compression algorithm occurs through the use of a similar structure as that of which encoded the file but with some small, but critical, differences. Once again, the algorithm utilizes a linked list, of type Stack, for which each node, of type StackNode, contains a binary tree, of type HuffNode. The implementation of Stack in this algorithm is a much more classical implementation of a Stack, and thusly has the primitives Stack_popFront, which removes the front of the stack and returns the value, in this case the tree of type HuffNode, to the caller, as well as Stack_pushFront, which takes a value, once again a tree of type HuffNode, and places it on the top, or front, of the Stack. Again, the same BitFile structure is utilized as was mentioned in the previous algorithm description, but instead of writing a passed bit and byte, it instead reads a bit from the file with BitFile_nextBit or a byte using BitFile_nextByte.

The implementation of these structures to allow for decompression was broken down for purposes of decompression into three functions, as follows:

- `void Stack_popPopCombinePush (Stack * stack)`
  - Using the classical Stack primitives previously discussed, removes the first two nodes from the Stack, combines them, and pushes them back onto the stack. This is done through two successive calls of Stack_popFront followed by the creation of a new HuffNode with left and right values of the two popped HuffNodes and then pushing that new tree to the front of the stack.
- `HuffNode * HuffTree_readBinaryHeader (FILE * fp)`
  - By reading the file pointed to by fp bit-by-bit using a BitFile, analyzes the header sent in from the file. Just as was discussed previously with the post-order traversal to compress the tree, the implementation of reading the header creates a new node from the byte following any ‘1’ read from the file. This happens until a ‘0’ is read, in which case the first two (the smallest weighted nodes, as determined through compression) nodes are removed, combined, and the result pushed back onto the stack by using Stack_popPopCombinePush on the stack. When there is only one node left in the stack, this means that the header has been completely read, thus justifying the extra ‘0’ bit written to the file when composing the header in the compression algorithm. The ultimately combined Huffman coding tree is then returned from the function.
- `Unsigned char * Huffman_applyTree (FILE * fp, HuffNode * tree, size_t * len_ptr)`
  - Receiving the Huffman coding tree from HuffTree_readBinaryHeader, this function applies it to the compressed contents of the file. With the file marker currently pointing to the length of the file written at the end of the header, the function reads the length of the file up to the new line character as to determine the length of a result buffer to later be written into a decoded file. Using a static implementation of the BitFile functionality, the function reads individual bits to traverse the tree down to a leaf node. When the function reaches a leaf node, this means that a character has been decoded and is thusly written into the result buffer. When the file is done being read, the return buffer is sent out of the function into the main function, at this point being the complete decoded text of the compressed file.

Upon the retrieval of the output buffer from Huffman_applyTree, the output is then written to the “.unhuff” appended decompressed file output from the function, back in the original state that it was prior to compression.

Results

To show the functionality of this algorithm, I chose to apply it to the poem “Alone” by Edgar Allen Poe, a file with many lines and a variety of characters that extend the bounds of the basic alphabet. The results, yielding a satisfactory compression ratio and no file corruption, were as follows:

-bash-4.1$ cat inputs/alone.txt
From childhood's hour I have not been
As others were -- I have not seen
As others saw -- I could not bring
My passions from a common spring --
From the same source I have not taken
My sorrow -- I could not awaken
My heart to joy at the same tone --
And all I lov'd -- I lov'd alone --
Then -- in my childhood -- in the dawn
Of a most stormy life -- was drawn
From ev'ry depth of good and ill
The mystery which binds me still --
From the torrent, or the fountain --
From the red cliff of the mountain --
From the sun that 'round me roll'd
In its autumn tint of gold --
From the lightning in the sky
As it pass'd me flying by --
From the thunder, and the storm --
And the cloud that took the form
(When the rest of Heaven was blue)
Of a demon in my view --

-bash-4.1$ ./compress inputs/alone.txt

At this point, the compressed file is now stored at inputs/alone.txt.huff.

In order to compare the size of the file, we utilize the “wc” function in Bash with the “-c” flag, which returns the number of bytes within a file.

-bash-4.1$ wc -c inputs/alone.txt
755 inputs/alone.txt

-bash-4.1$ wc -c inputs/alone.txt.huff
483 inputs/alone.txt.huff

The two above values indicate that the original file was 755 bytes whereas the new file is 483 bytes. This means we have a compression ratio of 36% for this file!

-bash-4.1$ ./decompress inputs/alone.txt.huff & cat inputs/alone.txt.huff.unhuff
From childhood's hour I have not been
As others were -- I have not seen
As others saw -- I could not bring
My passions from a common spring --
From the same source I have not taken
My sorrow -- I could not awaken
My heart to joy at the same tone --
And all I lov'd -- I lov'd alone --
Then -- in my childhood -- in the dawn
Of a most stormy life -- was drawn
From ev'ry depth of good and ill
The mystery which binds me still --
From the torrent, or the fountain --
From the red cliff of the mountain --
From the sun that 'round me roll'd
In its autumn tint of gold --
From the lightning in the sky
As it pass'd me flying by --
From the thunder, and the storm --
And the cloud that took the form
(When the rest of Heaven was blue)
Of a demon in my view --