Ashwin Haridas, ah1058 (Section 06)
Ritin Nair, rrn32 (Section 04)

Assignment 1: File Compressor

Purpose:

Have a user provide flags, a path or file, and optionally a codebook depending on the flag to build a codebook from a file, compress a file, or decompress a file.

Build Instructions:

Using the provided Makefile, simply running “make” or “make all” will produce the “fileCompressor” executable. Running “make clean” will delete the executable as well as other object files created.

Input:

The format of the inputs will be <flags><path or file><codebook>. The flags can be “-b” for building a codebook from a file, “-c” to compress a file, and “-d” to decompress a file. Having a “-R” flag before any of the above flags will result in the program carrying out these operations on all the files in the given path by recursively descending through the subdirectories and files contained in the path. Providing the inputs out of order or giving an invalid flag, path, or file will all result in an error.

To build a codebook from a file the inputs would be “-b”, a path or file name, and optionally a “-R” flag to carry out this operation on all the files within a directory recursively. When carrying out this command the user does not need to provide a codebook as an input. When compressing a file the user must provide a “-c” flag, a path or file to compress, a codebook to base the compression on, and optionally a “-R” flag. To decompress a file the user must provide a “-d” flag, a path or compressed file with the “.hcz” extension, a codebook to base the decompression on, and optionally a “-R” flag.

Output:

The output from running this program with the build flag will be a text file named “HuffmanCodebook”. This file will contain an escape character on the first line, and each following line will be in the format <huffman encoded bitstring><tab><token><newline>. The output from running this program with the compress flag will be a text file with the name of the file to be compressed, but with the extension “.hcz”. This file will contain a binary encoding of the text from the original file, with the encoding being based on the HuffmanCodebook the user provides in the command line. The output of running this program with the decompress flag will be a text file of ASCII characters based on decoding the encoded file the user provides using the HuffmanCodebook.

Header File:

This project uses a self defined header file, “fileCompressor.h” which uses important system libraries.

Struct:

This project uses a self defined generic struct, “node”, which is used for our AVL Tree, MinHeap, and Huffman Tree implementations. We found using the generic node very time and space efficient because we would always store data in a single struct rather than passing it between multiple structs.

The struct includes:

- char* token
- struct node* left
- struct node* right
- char* encoding
- int freq
- int height

This structure allows us to have the properties of an AVL Tree, MinHeap, and Huffman Tree all in one.

Library Files:

The project incorporates a multitude of library files, for organizational and efficiency purposes, that include important functions that our main program uses. These are:

- AVL.c
    - add
        - This function adds a given node into an AVL Tree. If rebalancing is needed, the function calls the appropriate rotate function.
    - Rotating functions
    - AVLSearch
        - Returns a node given a token. This is useful for finding the encoding of a function for decompressing purposes.
- huffman.c
    - buildHuffmanFromEncoding
        - This function creates and builds a Huffman Tree given a node with its encoding and token, which was read in from the Huffman Codebook. This allows us to have access to all tokens and its respective encoding.
    - buildHuffmanTree
        - This function uses the well-known Huffman Tree algorithm to construct a Huffman Tree given a MinHeap of nodes.
    - encode

        - This function traverses through the Huffman Tree, noting if the traversal is going left (by storing a 0) or right (by storing a 1). Once the function reaches a leaf node (a node with no children), the leaf node stores the correct encoding constructed.

- minheap.c
    - createHeap
        - This function uses pre-order traversal to go through the nodes of the AVL and move them to an array of nodes. It returns the index of the array that the node is assigned to. This is useful because having an array structure allows us to convert these nodes into a heap.
    - heapify
        - Maintains the min-heap structure by checking if a parent is smaller than its two children. If this is not the case, then the parent is swapped with its smallest child and heapify is called again to update the heap structure. This is useful for converting the array into a min-heap structure.
    - buildHeap
        - Calls heapify on all elements of the heap that have children, making sure that they are indeed valid parents. This is useful because it creates the min-heap structure from an unsorted array of nodes.
    - extractMin
        - Extracts the minimum element from the heap, which is the root. It then updates the size of the heap and calls heapify to maintain the min-heap structure.
    - insertIntoHeap
        - Performs an addition of an extra node at the bottom of the min-heap. This function then sifts up from the point of insertion to ensure the min-heap structure is maintained.
- IO.c
    - decompressFile
        - Given the compressed file, this function reads a 0 or a 1 (since a compressed file only consists of 0s and 1s) and traverses the given Huffman Tree until we reach a leaf node. We then write the token of the leaf node to the decompressed file.
    - buildHuffmanFromFile
        - Given the Huffman Codebook, we read a token and its encoding, create a node that contains both of these data, and insert it into a Huffman Tree.
    - compressFile

        - Given an uncompressed file, this function reads up until a control character, which is the delimiter for our tokens. We then traverse through our AVL tree (which contains tokens and their respective encodings) and write to the compressed file the encoding of the token we just read.
    - buildAVLFromFile
        - Similar to buildHuffmanFromFile, we read a token up to a control character, and insert the token into the AVL Tree. If a token appears more than once, the node in the AVL Tree has its frequency attribute incremented.
    - writeHuffmanCodebook
        - This function traverses through the Huffman Tree and writes to the Huffman Codebook a token and its encoding.
    - buildAVLFromHuffman
        - This function reads through the Huffman Codebook, and inserts a token and its encoding into the AVL Tree.

Analysis of Main Functions:

Build:

- To build a huffman codebook we first create a file in the path to write the encoded bytestrings and their corresponding ASCII values to. Next, we read the file provided in the command line with tokens being made from delimiters such as spaces and control characters. These tokens are inserted into an AVL of nodes consisting of the string, it's frequency, the node's height, and pointers to its left and right child. The AVL is organized lexicographically, and repeated tokens result in the node containing that token having its frequency incremented. A min-heap is then made from the nodes in the AVL based on frequency. This min-heap is used for huffman encoding to create an encoded bytestring for each token. These byte strings, their corresponding tokens, and an escape character are all written to the Huffman codebook. Special replacements were used to denote tokens representing spaces or control characters as writing these to the codebook directly would not make it easily legible.
- Time efficiency:
    - AVL -
        - Inserting into the AVL results in a worst case time efficiency of log(1) + log(2) + … log(n) because the deepest leaf node will be at height log(n) in the worst case scenario for an AVL because the tree is relatively balanced. This simplifies to log(n!) which is approximately nlogn, giving us a time efficiency of O(nlogn). All rotations are performed in constant time, or O(1) time complexity.

        - Using an AVL was more efficient in terms of time complexity compared to other options such as adding tokens to an array or a BST. These alternative options would take O(n^2) in the worst case scenario, derived from adding tokens taking 1 + 2 + … + n operations, which simplifies to O(n^2). As a result of these options being slower, we instead decided to use an AVL which can store all the tokens in O(nlogn).
    - Heap -
        - Building a heap can be done in linear time, O(n).
        - Inserting a node into a heap and having to perform sift up operations for this node is done in O(logn) because in the worst case the node must be moved up the entire "height" of the heap.
        - Heapify is done in O(logn) in the worst case when it must go through the entire "height" of its subtree.
    - Huffman Tree
        - Determining the minimum element to extract from the heap is done in constant time because the minimum node is just the "root" of the min-heap. However, having to call heapify on the heap again to maintain its structure takes O(logn). This process is repeated for every node in the heap except the last one remaining which is guaranteed to be the minimum, so the time complexity is (n-1) * O(logn), or O(nlogn).
    - Overall time efficiency: O(nlogn)
- Space efficiency:
    - AVL - O(n)
        - The AVL has n nodes total in the worst case where each token is unique.
    - Heap - O(n)
        - The heap has n nodes total
    - Huffman Tree - O(n + (n-1)) = O(n)
        - The n-1 in this calculation is derived from there being an additional n-1 parents created in the huffman tree during the encoding process.
    - Overall space efficiency: O(n)

Compress:

- To compress a file this program first builds an AVL from the huffman codebook provided by the user. The nodes of the AVL contain a token and its associated encoded bytestring based on the codebook. The file is then read through with tokens being created from control characters and spaces acting as delimiters. The delimiters are also treated as individual tokens themselves. The AVL is searched through with these tokens acting as keys, and their associated encoding in the node the search finds is written to a new compressed file. This process is repeated for every token in the file until the original file is completely compressed.

- Time efficiency:
    - AVL-
        - Building an AVL from the huffman codebook is done in O(nlogn)
        - Searching for each token in the AVL is done in O(logn) time because in the worst case the token is found at the bottom of the AVL so the height of the tree must be traversed to find it. All n tokens are searched for in the AVL exactly once, resulting in a time complexity of O(nlogn) to search for every token in the AVL tree.
    - Writing to the compressed file:
        - Writing each token's encoding is done in constant time, and this is repeated for all n tokens resulting in a time complexity of O(n) for writing to the compressed file.
    - Overall time efficiency - O(nlogn)
- Space efficiency:
    - AVL - O(n)
        - The AVL has n nodes total in the worst case where each token is unique.
    - Overall space efficiency - O(n)

Decompress:

- The first step this program does when decompressing a file is it reads through the huffman codebook provided by the user and builds a huffman tree based on the codebook. The leaf nodes of the huffman tree contain a token and its associated huffman encoded bytestring. Then, the compressed file is read through one character at a time with each character indicating how to traverse the huffman tree. A '0" indicates moving left in the huffman tree and a "1" indicates moving right in the huffman tree. After a leaf node is reached in the huffman tree, its token is then written back to the original file. This entire process results in the compressed file being decompressed.
- Time efficiency:
    - Building a huffman tree from the codebook is done in O(nlogn)
    - The worst case for searching for a token in the huffman tree would be O(logn) if the token must be found by traversing the entire height of the tree. When repeated for all n tokens this results in a time complexity of O(nlogn) for searching for each token in the huffman tree.
    - Writing back to the original file is done in constant time, and this is done for all n tokens resulting in this process taking O(n) time complexity.
    - Overall time efficiency: O(nlogn)
- Space complexity
    - Huffman tree - O(n)
        - The huffman tree has n + (n-1) nodes because there are n tokens and (n-1) parents created for these leaf nodes.

- Overall space efficiency: O(n)

Other Notes:

- Recursive Functionality
  - In order to implement recursive flag capability, we first created our main data structure (AVL Tree or Huffman Tree). We then parsed through every eligible file and added it to this data structure. We would then pass this data structure into either our build, compress, or decompress function.
  - Build - since build outputs a single Huffman Codebook for the entire directory, all we needed to do was send our AVL Tree into our normal build method and construct the codebook file in that directory
  - Compress - this required a bit more work. After we initially created our AVL tree of nodes that contain the token and its encoding, we then had to recursively go through all directories/subdirectories and find any file that is eligible to be compressed (meaning any files that were not "hcz" files). We then constructed a compressed file, using our compress function, and emitted an hcz file in the same directory
  - Decompress - this was similar to compress. We first created the general Huffman Tree from the codebook. Then, we recursively went through our directory and any eligible files (files that ended with "hcz") were then applied to the decompress function.
  - Something important to note is that if we ever encountered an error with an individual file, the rest of the directory was still parsed and operations were still applied.
- Representation of Escape Characters
  - Something interesting that we ran into was figuring out a way to represent control characters. Because our delimiters for our tokens were control characters, we needed to represent these characters in the Huffman Codebook as well in order to have proper compression/decompression of a file. However, simply writing the control character to our codebook was not optimal as it caused our file to look very weird. For example, if we were to record a '\t' (the tab character), if printed would actually print a tab rather than the character '\t'. There are a few ways to go about this issue.
  - In order to uniquely represent control characters, we decided to convert these characters into their ASCII values and prepend a "/" to it. For example, a space bar would be converted to "/32" and '\n' would be converted to "/10".
  - Obviously, an issue arises when a token in a file is actually something like "/32" or "/10". If we simply stored these as they are, we may run into the problem of when parsing the codebook, we would not know if "/32" is a token or if it was a

space. In order to not run into this ambiguity, we decided that if we ever encountered a token that started with a "/", we prepended an additional "/" to that token and stored it in our data structure and codebook.
    - In order to actually convert our converted token into its actual token, we need to first know if what the token we are looking at needs to be converted. To do this, we can first look at the first character of every token. If this character is a "/", we know that it is a token that we modified before. This is because any token that either was a control character or actually started with a "/" had a "/" prepended.
    - When we find a potential token, we check the next character's value. If this character is also a "/", we know that it is not a control character, so to convert we just get the rest of the token.
    - If the character is not a "/", we know that it is a control character. So, we get the ASCII value after the "/" and convert it into a character.
- Errors/Warnings: An error or warning is printed for the following cases
    - Warning
        - Empty file
        - File already exists
        - Empty directory
    - Error
        - A certain file cannot be decompressed
        - No valid files in directory
        - Mallocing errors
    - Fatal Error
        - File does not exist
        - Flag errors

Final Thoughts/Reflection:

After rigorous testing, we believe that our program works very well and is good in terms of space and time efficiency. However, there are definitely areas where we could have improved. We made the tradeoff of time for space efficiency in our choice of data structure of AVL tree. If we wanted to be more time efficient, we could have used a hash table instead. Also, we believe we could have optimized our code a bit better because we did repeat some things quite a few times. Overall though, our program is well put together and is able to handle major test cases.