Asst2 file compression: ReadMe

Overview of **Asst2 File Compression**:

File Compression is designed to create a single executable named “fileCompressor” which takes in command-line arguments and is designed to build a codebook given a file/multiple files, compress files, decompress files, and perform all operations recursively within a directory.

The program contains three source files:

- fileCompressor.c
- fileHelperMethods.c
- structures.c.

structures.c contains all the data structures necessary to be used in this project. The main structures that are implemented in *structures.c* are an AVLNode, a TreeNode, a Heap, and a Queue.

- **structures.h** is the header file that contains all its structures as well as public methods signatures meant to be used in fileCompressor.c
- **structures_priv.h** is the header file that contains all of *structures.c*’s private helper method signatures used solely in *structures.c*

fileHelperMethods.c contains helper methods to do basic file operations and string manipulations. Since we’re not allowed to use f-methods, *fileHelperMethods.c* handles tasks such as reading and writing files.

- **fileHelperMethods.h** is the header file of *fileHelperMethods.c.* It also contains error-handling macros used throughout the entire program. It contains all of *fileHelperMethods.c*’s method signatures.
    - Note: the delimiter string to separate tokens is also defined as a macro here.

fileCompressor.c handles all the main functions for the project. *fileCompressor.c* handles buildcodebook, compress, decompress, and recurse.

(each method is explained in detail in the space and time efficiency analysis section of the ReadMe).

fileCompressor.c also handles input-argument checks from the terminal for the executable.

- **fileCompressor.h** is the header file of *fileCompressor.c*. It contains all *fileCompressor.c*’s method signatures.

Tokens

Each file is separated into token strings. How tokens are determined is how the file is separated by delimiters. The delimiters we chose for this project are

' ', ' \t ', ' \n ', ' \a ', ' \b ', ' \' ', ' \" ', ' \v ', ' \f ', ' \r ', ' \? ' , ' \0 ', ' \\' .

Every token in the file is separated by these delimiters.

ex] "**this is**\t**an**\n**example**" would be separated into four tokens: "this", "is", "an", and "example".

Note: each token is also represented in the codebook by an escape character.

ex] '\t' is represented by the string "\\t" in the codebook.

Conversions between the delimiter string representation and the actual delimiter itself are handled by *fileHelperMethods.c.*

How To Run **fileCompressor**

./fileCompressor <flag> <path or file> |codebook|

Note: ./fileCompressor is implemented so all flags can run in any order. However, flags must run before the files and the codebook must be the last argument if it is an argument.

.

- At most one flag of -b, -c, and -d can be passed in as an argument
  - -c is for running compress, a codebook must be passed in as an argument
  - -d is for running compress, a codebook must be passed in as an argument
  - -b is for running buildcodebook, only one file/path at most can be passed in as an argument
- a -R flag is optional if you want to run the program in recursive mode. Can only have at most 1 -R flag.
  - Note: in order to run in recurse mode, you ***should*** pass in a path as an argument. If a regular file is passed in, then the flag will still run, but will be treated as non-recursive (program will output a warning).
  - Note: if you don't run in recurse mode you **must** pass in a file, or the program will terminate.
- <path or file> is the path or file you want to operate the flag on. You can only have at most one path or file.
  - if you want to specify multiple files, pass in a folder with multiple files.
- |codebook| is the HuffmanCodebook generated to be used in compress or decompress. Must be last argument if passing in a codebook.
  - Note: if the codebook passed in **does not** match the project-specified format of a codebook, then the program will terminate.

The program will terminate if the arguments do not match the format specified above. The flags can be placed in any order (before the file arguments). The file arguments must follow as file/path, and then codebook (if passing in a codebook).

Structures Used And a Brief Space Analysis Per Structure:

- Token: a structure to associate a string “token” with either an int “frequency” to store the frequency of that token in each file, or a string “encoding” to store the huffman codebook encoding associated with that token.
    - The advantages of storing the token and frequency/encoding in a *Token struct* is that a token is automatically associated with information about that token and it allows for more compact and secure code.
    - All other structures have a *Token struct pointer* in its field because all structures need to store a token and some information associated with it.
    - An advantage of storing a *Token struct pointer* instead of the *Token struct* itself is to avoid time spent copying the *Token struct* while passing information from one data structure to the next. Each Token is going to be used throughout the program anyways and each token is only freed near the end of the program.
    - Note: the frequency and the encoding of the *Token struct* is stored in a nested **union** since each token only needs to have at most one piece of information associated with it.
        - So if the *Token struct* has a frequency, it doesn’t have an encoding, and vice versa. A boolean is used to keep track of whether or not the *Token struct* has a frequency or an encoding.
    - Therefore the size a *Token* would take is the size of the string and sizeof(int)/sizeof(encoding string) + 1 byte for the boolean

- TreeNode: Tree structure used to build a huffman tree (in huffman encoding) in *buildcodebook.*
    - Note: this tree is different from an AVL Tree because it does not need to be balanced. The TreeNode is designed to store a *Token Struct pointer* of a token string and the frequency associated with the token string.
    - TreeNode has a function *mergeTree()* which takes two *Trees* and combines them into one tree where the root is the combined frequency of the roots each individual *Tree*.
    - The size of the final Huffman Tree in *buildcodebook* is the number of distinct words in all files plus the number of times the Tree is merged in *buildcodebook.* If n is the number of distinct tokens, then a tree is merged $n - 1$ times. Then, the final Huffman Tree contains $2n - 1$ nodes.

- AVLNode: AVL Tree designed to keep track of tokens and the frequency of each token in a file in BALANCED tree which is used in *buildcodebook.*
    - *AVLNode* methods include *insertOrUpdateAVL()* which inserts a token if not found, or updates a token’s frequency if found. The method combined the two operations to avoid unnecessary time spent to search for a token initially since

both operations require a search through the tree anyways. Each insert/update in the tree is O(log(n)) where n is the number of nodes in the tree.
        - *insertOrUpdateAVL()* returns whether or not it inserted or updated so that the program can free duplicate strings if necessary. Changes to the AVL Tree are done by passing in a pointer to the address of the AVLNode.
    - Each balance is $O(1)$ since a single balance does at most two rotations (constant time)
    - The size of the AVL Tree of frequencies would be $O(n)$, where n is however many distinct words are found in all files by running *buildcodebook().*

- CodeNode: Also an AVL Tree. *CodeNode* is designed to associate tokens and the byte-encoding associated with each tokens. It is designed to create a "Codebook Tree" for relatively quick lookup of the elements in a Codebook which is used in *compress* and *decompress.*

  A distinct feature of the Codebook Tree is how it is ordered.
    - if running compress, the Codebook Tree is ordered based on each **token.**
    - if running decompress, the Codebook Tree is ordered based on each **encoding**

    - Note: Semantically a *CodeNode* is exactly the same as an *AVLNode* in code. However, a distinction was made for readability and also because an *AVLNode* and a *CodeNode* serve different purposes.
    - CodeNode methods include insert, and search which are both done in O(log(n)) time for a single run where n is the number of nodes in the tree. The Insert and Search functions take in a Comparison Mode which keeps track of how the tree is ordered (either by tokens or encodings).
    - There is a single method *buildCodebookTree()* which builds a Codebook Tree out of a Huffman Codebook.
    - CodeNode repurposes AVLNode's balance method since they are semantically the same.
    - The size of the Codebook Tree in *compress* or *decompress* is $O(n)$, where n is however many tokens there are in the given Huffman Codebook (so the number of distinct words).

- MinHeap: A Min Heap of *Token Structs* designed to be used in *buildcodebook* whilst running the *Huffman Coding* algorithm. It is designed to keep track of all tokens but have quick access to the token with the smallest frequency.
    - Note: creating the heap initially would take **linear** time. We implemented the heap by transferring all the Tokens in an AVL Tree into an array, then applying the **linear** heapify algorithm (sift down from the first non-leaf node). The proof for

this time efficiency can be found in Sesh's Data Structures book, but basically it is more efficient than adding a single Token to the heap array and then sifting it up each time.
    - Each sift down however takes $O(logn)$ time if n is the number of elements in the heap.
    - The heap has methods to remove the minimum Token and to peek at the minimum Token on the top of the heap.
    - The size of Min Heap in *buildcodebook* is $O(n)$, since it's the size of the AVL Tree which is the number of distinct tokens in each file passed (which is n).
        - Note: As soon as the Min Heap is created, the AVL Tree is freed because it is no longer needed.
        - Note: The Min Heap should be empty if *buildcodebook* was run successfully. The Min Heap array is automatically freed at the end of running the Huffman Coding algorithm.

- Queue: A Queue of Trees designed to be used in *buildcodebook* whilst running the *Huffman Coding* algorithm. A Queue keeps track of the front and end of a fancy doubly linked list (called a QueueItem) so that each enqueue and dequeue can be done in O(1) time.
    - A QueueItem is just a doubly linked list of trees to maintain the order of the Queue. Since we only want to give access to the front and end of a Queue, the QueueItem doubly linked list is wrapped in a Queue struct, so that programs outside of *structures.c* only have access to the Queue struct, and can't accidentally access the middle of a Queue for example.
    - The Queue has functions enqueue, dequeue, and peek.
        - Note enqueue enqueues a tree and dequeue returns a tree. peek returns the frequency of the root of the tree at the front of the Queue. There is no direct handling of a QueueItem.
        - Note: after each dequeue, the QueueItem node is freed, so if a Queue is entirely dequeued, all QueueItems are freed.
    - The size of a Queue depends on how many trees are created. Since trees are consistently being merged in *buildcodebook*. The max size a Queue could be at a given time is $\frac{n}{2}$ where n is the number of distinct tokens. (if each token is merged into a tree and then enqueued)
        - Note however, when a Queue is larger, the Min Heap is smaller since elements are being taken from the Min Heap to create a tree, then is added to the Queue in *buildcodebook*. In the end of running the Huffman Coding algorithm, the Queue should be empty if it was run successfully.

Space And Time Efficiency of **fileCompressor**

Variables:

- Let **n** be the number of distinct tokens in all files.
- Let **m** be the number of total tokens in all files.

  Note: $m \geq n$ in all cases
- Let **k** be the total number of bytes of all encodings for n-tokens
- Let **j** be the total number of bytes of all encodings for m-tokens

  Note: $j \geq k$ in all cases

Note: a distinction is to be made in that n and m are of different "units" than k and j. n and m represent tokens and k and j represent bytes. The number of bytes per each encoding can vary largely and has the potential to be much larger than the number of bytes of the token it represents. The reason for this distinction becomes evident as we analyze the main functions of *fileCompressor.c* .

---

buildcodebook:

buildcodebook takes in the -b flag. *buildcodebook* builds a single codebook of a file/directory of files. It goes through all the tokens in each file and gets the frequency of each token and stores it in an AVL Tree. Then *buildcodebook* calls upon *HuffmanCoding()* and performs the Huffman Coding algorithm by creating a Min Heap out of the frequency AVL Tree and a Queue of trees. Once the Huffman Coding is completed, it returns a Huffman Tree. Then *buildcodebook* recursively goes through the tree and finds the encoding for each token and then writes the token and encoding to a created codebook file.

➢ The codebook file is created in the directory of the executable of each token and its associated encoding.

Efficiency Analysis of *buildcodebook*:

1. buildFrequencyAVL()

   Run through files and retrieve all tokens. For every token encountered, add each token to the global AVL Tree. If the token was found, then update the frequency of the token, otherwise, add the token to the tree.

   Each insert/update would take $O(logx)$ time in the worst case where x is the number of nodes currently in the tree. Then the Worst Case for the all insert/updates is if we encounter every distinct word first, and then every duplicate word afterwards.

   i. Time for inserting n distinct words:

   $$log(1) + log(2) + log(3) + ... + log(n) = log(n!)$$

   Note that:

   $$log(\tfrac{n}{2}) + log(\tfrac{n}{2}) + ... + log(\tfrac{n}{2})$$
   $$\leq log(1) + log(2) + .. + log(n)$$
   $$\leq log(n) + log(n) + ... + log(n)$$

So:

$$(\frac{n}{2})log(\frac{n}{2}) \leq log(n!) \leq nlogn$$

Then we can bound $log(n)$ so that:

$$O(log(n!)) = O(nlogn)$$

ii. Time for updating $(m-n)$ duplicates:

- The tree already has n nodes in worst case. So for each update, the worst case would be $O(logn)$
- Since we are updating (m-n) times, the total worst case efficiency for updating is $O(\ (m-n)logn\ )$
-

Total worst case time efficiency to insert/update m words into a tree is:

$$O(nlogn) + O(\ (m-n)logn\ ) = O(\ mlogn\ )$$

Total space efficiency for the tree = $O(n)$ (only as many nodes as distinct words)

2. huffmancoding()

Huffman Coding Algorithm: Create a huffman tree from running the Huffman Coding Algorithm.

i. Build a Min Heap from AVL Tree :

- Traverses through AVL Tree and adds each Token from the tree to the heap array (not ordered) $O(n)$
- Perform linear heapify on the heap by sifting down non-leaf nodes (proof in Sesh's Data Structures Book) $O(n)$

  Note: we free the AVL Tree as soon as we create the heap

ii. Create a Queue data structures and remove the item with the minimum frequency from the heap and queue. Merge the two elements into a heap with the parent node being the combined frequencies. Enqueue the merged tree into the Queue. Rinse and Repeat until resulting with no items in the heap and a single tree in the Queue.

Note: we do $n-1$ merges regardless of the resulting tree, so there are $n-1$ extra nodes being created for the resulting huffman tree.

So the final huffman tree would take $2n-1$ space

iii. Worst case is we loop through the Huffman Coding Algorithm once per each item. So we would loop through n times.

Time Analysis per each iteration:

- Each peek from the heap and queue is $O(1)$
- Each remove min from the heap takes $O(logn)$ time to remove the heap, then sift down.

- Each enqueue/dequeue from the queue is $O(1)$
- Each merge tree is $O(1)$
  Time per all iterations: $O(1 + logn + 1 + 1) = O(logn)$

Total worst case time efficiency to run huffmancoding(): $O(n) + O(nlogn) = O(nlogn)$
Total space analysis= $O(2n - 1) = O(n)$

(note: the heap is freed after finishing huffman coding and each queue item is freed after every dequeue, so at this point in the program, only the huffman tree takes up space)

3. writeEncodings()
   Open a codebook file in the directory of the executable.
   Recursively traverse through a tree and create a string for each node that represents its encoding. The encoding is a parameter for *writeEncodings()*. Everytime the program traverses left, add a 0 to the encoding and recurse on the left node with the new encoding. Everytime the program traverses right, add a 1 to the encoding and recurse on the left node with the new encoding. Replace the root's frequency with the encoding then write that encoding to the codebook file if it contains a token. We are guaranteed that no encodings start with another encoding because of the Prefix property of a Huffman Tree.
   - traverse through all nodes in the huffman tree takes $O(n)$ time (since there is 2n-1 nodes in a Huffman Tree)
     i. for each iteration, a single write and create encoding is $O(1)$ time.
     ii. for each iteration, a new encoding is being created, so each encoding would take h number of bytes in memory where h is the height of that node. The height of the final tree and the size of each encoding highly varies depending on the frequencies of each token. So we assigned the variable k to represent the total number of bytes for all encodings. So overall space efficiency for storing tokens is $O(k)$

Therefore time efficiency of writing tokens and their encodings overall is $O(n)$
And space efficiency of storing encodings for every node is $O(k)$

So overall,
buildcodebook's worst case time efficiency is:

$$O(mlogn) + O(nlogn) + O(n) = O(mlogn)$$

buildcodebook's worst case space efficiency is:

$$O(n) + O(k) = O(n + k)$$

remarks:
- this analysis did not account for space used for reading the file as a string since it is inevitable and will be freed anyways after every run of *buildFrequencyAVL()*.

- $O(n+k)$ space is a little deceiving as n represents tokens of varying bytes and k is the number of total bytes all encodings take. So they are not using the same “unit” of space. However this distinction is important to make since k has the potential of being much larger than the number of bytes that n takes.

---

compress:

compress takes in a file/path and a huffman codebook and then builds an AVL Tree (Code Node) out of the codebook to quickly search up tokens and retrieve their encodings. It parses through every file for tokens and looks up each token in the AVL Tree and then writes the associated encoding into a new file.

- ➢ The compressed file is created in the same directory of the non-compressed file with the extension ".hcz".
- ➢ if a directory is passed, then *compress* compresses all regular files in that directory.
- ➢ if a token is not found in the codebook, the program issues a warning and deletes the compressed file created.

Efficiency Analysis of *compress*:

- creating: the AVL Tree (just once per run):
  - It would take $O(nlogn)$ time to create the AVL Tree out of the codebook since the codebook always has n tokens (number of distinct words in all files) and each insert in the worst case is $O(logn)$.
  - It would take $O(n+k)$ space to create the AVL Tree of n total tokens and k total encodings.

- We look up m tokens in every file passed in
  - for each token, it would take $O(logn)$ time in the worst case to search for a token in the AVL Tree. It takes $O(1)$ time to write the encoding to a file.

  Therefore it would take $O(mlogn)$ time to go through all files and write the encodings.

So overall,

compress's worst case time efficiency is:

$$O(mlogn) + O(nlogn) = O(mlogn)$$

compress's worst case space efficiency is:

$$O(n+k)$$

Note: the same notes in *buildcodebook* apply here as well.

- Remember that n represents tokens of varying bytes and k is the number of total bytes all encodings take.

decompress:

decompress takes in a compressed file (ending with .hcz) and a huffman codebook and then builds an AVL Tree (Code Node) out of the codebook to quickly search up encodings and retrieve their tokens.

It parses through the file by continuously creating substrings of 0's and 1's, and then looks for the associated token of that substring in the AVL Tree. If the token was not found, then it adds the next byte to the substring and continues searching until found. If found, then *decompress* writes the associated token into the decompressed file and starts again from the nest substring. *decompress* continuously does this until it reached the end of the file.

- ➢ The decompressed file is creates in the same directory of the compressed-file removing the extension ".hcz".
- ➢ if a directory is passed, then de*compress* decompresses all ".hcz" files in that directory.
- ➢ if there is still a substring at the end of the file and it is not found in the codebook, the program issues a warning and deletes the decompressed file created.

Efficiency Analysis:

- creating: the AVL Tree (just once per run):
  - It would take $O(nlogn)$ time to create the AVL Tree out of the codebook since the codebook always has n tokens (number of distinct words in all files) and each insert in the worst case is $O(logn)$.
  - It would take $O(n+k)$ space to create the AVL Tree of n total tokens and k total encodings.

- Since we're looking up each substring we create every time, we would look up roughly j times, which is the number of bytes of all compressed files.
  - for each substring, it would take $O(logn)$ time in the worst case to search for a token in the AVL Tree regardless if it was found or not.

  Therefore it would take $O(jlogn)$ time to go through all files and write the encodings.

So overall,

decompress's worst case time efficiency is:

$$O(nlogn) + O(jlogn) = O((n+j)logn)$$

decompress's worst case space efficiency is:

$$O(n+k)$$

Note: here, it is evident why we assigned j and k to variables. *decompress*'s time relies largely on the amount of bytes in all the compressed files, which is j.

recurse:
recurse takes in the -R flag along with any of the above flags and a directory. Then it applies each flag on every subdirectory of the directory passed in (along with the original directory of course).

- ➢ Note: if a file and not a path is passed into *recurse*, the program treats the operation as non-recursive.
- ➢ the analysis of *recurse* is the same because all that really changes is the number of tokens in total which we assigned to variable m.

Design Choices

The main design choice for this project is choosing which Data Structure to use for counting the frequencies of each token and also which Data Structure to use to quickly search through the Huffman Codebook. In the end, we chose to use an AVL Tree for both tasks.

Why AVL?:
The reason why we chose to use an AVL Tree is because an AVL Tree has relatively fast look-up time of $O(logn)$.

Design Dilemmas:
One design decision we were stuck on was whether or not to implement a Hash Table or an AVL Tree.
Even though a Hash Table has a faster lookup time if implemented correctly, we were also concerned about space efficiency. An AVL Tree will take up around $O(n)$ space if n is the number of distinct tokens. In order to implement a Hash Table with near constant lookup time, it would take around $O(n^2 + k)$ space, and even then, a Hash Table not guaranteed to have a constant lookup time. Furthermore, in the worst case of a Hash Table, all tokens will hash to the same index which would mean $O(n)$ look-up time, which is a lot worse than $O(logn)$ when n is large.

Therefore, we chose to implement an AVL Tree because of space efficiency and also because it is guaranteed to have an $O(logn)$ lookup time in the worst case no matter what.