fileCompression Implementation

cy287, zu9

Building Huffman Codebook

Token Processing

To record the frequency of the tokens of the files, we used a hashtable implementation, in the ideal case: it should be O(1) insert time where there exist no chaining and there are n tokens therefore best case is O(n). The worse case is if every token hashes to the same index then the insert time will be $O(n^2)$ where n is the number of tokens because it has to traverse through all the tokens to see if already exist and insert at the tail. However, the best case for memory will be O(n) where n is the number of tokens. The worse case for memory would be O(n) where n is the number of tokens. This is because our hashtable will double in size when the number of tokens equals the size of the hashtable and it will always require space for all the tokens or n.
runtime: O(n) (best case)
$O(n^2)$ (worse case)
memory: O(n) (best case)
O(n) (worse case)
where n is the number of tokens in the files.

Building the Huffman Codebook

To build our Huffman Codebook, we used a linked list implementation and a binary tree to construct the huffman trees. We used the linked list to sort the tokens in order of their frequencies so the lowest frequency would be the head and the highest frequency would be the tail.
To sort the frequencies, it will always take O(n) insertions because we need to get all the tokens and we will be taking it from the hashtable so we have to iterate through all of the hashtable, looking for all non-NULL entries. Finding the correct spot will take O(1) in the ideal case where it will insert in the head always. However the worse case would be O(n) where n is the number of tokens or inserting at the tail of the linked list. The memory for the linked list will always be O(n) where n is the number of tokens for both the best and worse case since it has to allocate space for all the tokens.
runtime: O(n) runtime (best case)
$O(n^2)$ runtime (worse case)
memory: O(n) (best case)

O(n) (worse case)
where n is the number of tokens in the files.

Generating Escape Sequence for Huffman Codebook

To generate the escape sequence, we have to check if the escape sequence generated is contained in the hashtable therefore it will take O(n) where n is the number of tokens since it has to iterate through the whole hashtable and see if any of the entries start with the escape sequence and if it does, it needs to recreate the escape sequence and repeat. The worse case would be $O(n^x)$ where n is the number of tokens and x is the length of the largest token since after the largest token, there will be a valid escape sequence since it will not exist in the hash table. The memory of the escape sequence will most likely be negligible meaning O(1) however in the worse case, it will be O(n) where n is the largest token.
runtime: O(n) runtime (best case)
$O(n^x)$ runtime (worse case)
memory: O(1) (best case)
O(n) (worse case)
Note in memory case, n is the largest token while n in runtime refers to number of tokens and x in runtime refers the length of the largest token.

Compression

Storing the Huffman Codebook

For compression, we used a hash table to store the Huffman Codebook. The hash table's keys was sorted by the token. It would take O(1) time to insert a token in the code book in the most ideal case where no tokens chained and there are n tokens therefore it would be O(n) time to insert all the tokens in the most ideal case. For the worse case, it would take O(n) time to insert a token if all the tokens are chained and there are n tokens therefore it would take $O(n^2)$ time to insert all the tokens in the worse case. The memory of the hashtable would be O(n) for both the worse and best case where n is the number of tokens.
runtime: O(n) runtime (best case)
$O(n^2)$ runtime (worse case)
memory: O(n) (best case)
O(n) (worse case)
where n is the number of tokens in the Huffman Codebook

Searching the Hashtable for Compression

For every token we read in, we would search for it in the hash table and return the bitstring therefore it would take O(1) for the best case where there exist no chaining and O(n) for the worse case, where all the tokens in the Huffman Codebook went to the same index and we have to do n time or the number of

tokens in the files.
runtime: O(n) runtime (best case)
O(n^2) runtime (worse case)
where n is the number of tokens in the files.

Decompression

Storing the Huffman Codebook

For decompression, we used a hash table to store the Huffman Codebook. The hash table's keys was sorted by the bitstring. It would take O(1) time to insert a token in the code book in the most ideal case where no tokens chained and there are n tokens therefore it would be O(n) time to insert all the tokens in the most ideal case. For the worse case, it would take O(n) time to insert a token if all the tokens are chained and there are n tokens therefore it would take O(n^2) time to insert all the tokens in the worse case. The memory of the hashtable would be O(n) for both the worse and best case where n is the number of tokens.
runtime: O(n) runtime (best case)
O(n^2) runtime (worse case)
memory: O(n) (best case)
O(n) (worse case)
where n is the number of tokens in the Huffman Codebook

Searching the Hashtable for Decompression

For every binary number (0 or 1) we read in, we construct it into a string and then search for it in the hashtable to get the word associated with the string. The best case for the search is O(1) if there exist no chaining and O(n) for the worse case, where all the tokens in the hash table went to the same index. We do this for every character in the files, represented by x. Therefore:
runtime: O(x) runtime (best case)
O(n^x) runtime (worse case)
where x is all the characters in the file and n is the all the tokens in the Huffman Codebook.