CI543 Report by Jake Ward

Section 1: Complexity & Huffman Coding

In this section, I show my workings behind and understanding of the complexity of my code. I will also Comment on what additions would need to be made to produce a fully functioning compression/decompression tool that stores compressed data in the form of binary codes. All versions of code are available at https://github.com/jake-ward2639/2nd-Year-Portfolio/tree/main/Huffman_Code

1.1 Encode

Working out complexity line-by-line, my encode method uses multiple other methods, the first of which being “freqTable” which has a time complexity of linear time/O(n), this is because there exists a for loop which will run for the length of the string input while all other lines have O(1) this is out weighted by the linear time as we are only accounting for the maximum runtime and this method is directly dependent on the length of the input, thus we drop the low order term.

The second method to be called is “treeFromFreqTable” which in turn calls my enqueue method. The enqueue/adding and dequeue/polling method hold O(log(n)) complexity as a consequence of being based on a heap. This is because, in the case of dequeue and enqueue, the heap is to possibly be restored shuffled and in computer science we assume log to be $\log_2$. Thus since the rest of treeFromFreqTable is O(1) the overall time complexity is dominated by O(log(n)) or logarithmic time.

The third method called is “buildCode” which almost directly calls the traverse method from either the leaf or branch class depending on the object calling it. This has a time complexity of simply O(n) seeing as how each branch or leaf is only created once but is dependent on the ArrayList input. Seeing as how the rest of buildCode is constant then O(n) dominates. Other than some other constant lines of code the final section to complicate the time complexity is a for loop based on the input, giving us a time complexity of O(n).

In summary with all these relevant line time complexities, we can calculate the overall time complexity of encoding. As (O(n) * O(log(n))) + O(n) + O(n) can be simplified to O(n(log(n))) + O(n) we need only drop the low order terms and thus the overall time complexity of Encode is O(n(log(n)))/O(nlogn). While not ideal, this was the optimal runtime for the algorithm while following the structure/guideline provided for us. This should also be the same for space complexity as encode is at a base level, based on a heap that is directly affected by the input and so has a space complexity of O(n). (O(n) * O(log(n))) + O(n) + O(n) = O(n(log(n)))

I acknowledge that Huffman coding should also theoretically have a complexity of O(n(log(n))) as it should take O(n) per each iteration in the input and O(log(n)) to determine the cheapest weight and insert.

1.2 Decode

All lines within the decode method besides the first are no methods to call that would exceed the O(n) time complexity put in place by the use of a for loop using the size of the input. The first line however calls the method “treeFromCode”.

treeFromCode calls no methods more complex than itself and contains several for loops based on the input, however since we drop constants the resulting complexity is O(n) as the complexity is proportional to the length of the character map input. As a result, the decode method has a complexity of O(n). Removing all printing used for additional testing improved the execution time of code by averaging a minute. I also acknowledge that time complexity isn't particularly important in reference to Huffman Coding since n is the number of symbols in the alphabet, which is a small number compared to the input length and complexity analysis only concerns when n grows to be exponentially large.

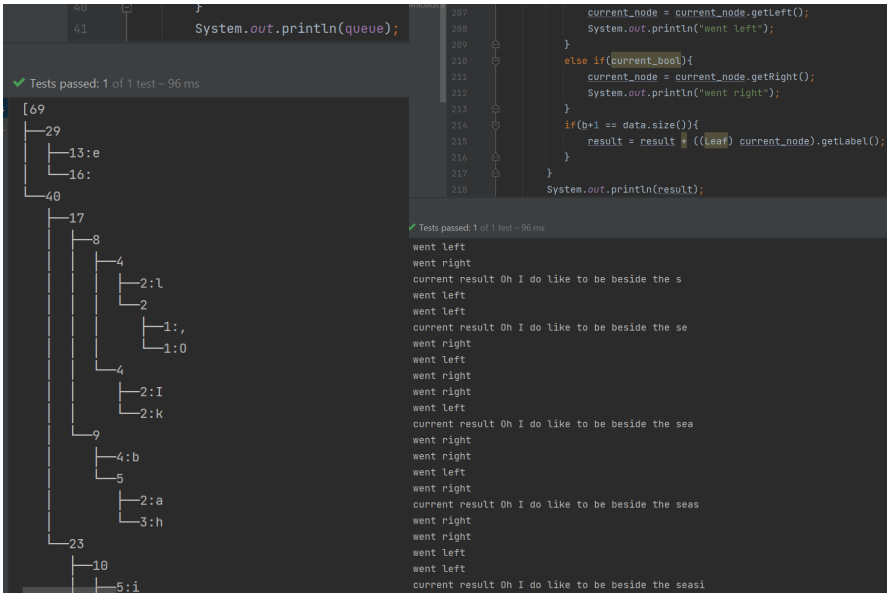
1.3 Applications of Huffman Coding

The use for Huffman coding is a conjunction of both lossless compression and cryptography, however in the field of computer science it is more likely to be used as a data compression tool. Real-World applications of Huffman coding are in algorithms such as DEFLATE, by proxy applying to image formats such as compressing as JPEGs and PNGs. As well as compression formats such as PKZIP, BZIP2, GZIP and audio formats such as MP3 files.

Huffman coding is often used in cryptography so that brute force attackers will not be able to find a meaningful result via natural language processing. As well as requiring two levels of decoding to be correctly implemented. Making brute force attacks on potentially sensitive information vastly less effective.

Like all compression methods, the reason for its application is in essence to protect from potential attackers and to compress to take up less space for reasons such as sending the information/data to other devices faster. This also means that it can only be decrypted using a generated key, in this case being the tree/map of characters and frequencies.

In order to convert the current code into a fully functioning compression method, I would need to implement methods to both output and save the key as well as the code or the decoded message/data depending on which method was called, as well as a method to detect whether decode or encode is the correct method to call based on the input type. I would need to possibly design a functioning UI for this or alternatively have to be used as a tool by the operating systems like other compression tools such as WinZip or 7zip.



Figures 1 - 2: Visual depiction of the functions of Huffman Coding encode and decode

Section 2: Application of Data Structures

In this section, I will cover the different relevant data structures we learned over the course of the module in the context of operating systems. I will not explain the basic data structures themselves so as to not prolong the report and allow for maximum discussion within the 4-page limit.

2.1 Sorting Methods

Radix, Stacks & Queues were the main sorting methods we explored with specifically a priority queue being integral to the functionality of Huffman coding. Stacks can be used by operating systems as “undo” functions in text editors and similarly as a form of backtracking to access the most recent element in a series such as recent paths in a file manager. They can be used to implement function calls and recursive functions and a compiler's check for accompanying brackets can be accomplished via stack. Stack frames are used in assembly code so that each subroutine can act independently of its location on the stack, and each subroutine can act as if it is the top of the stack. Stack frames are called upon when converting to and from assembly code as the local parameters require new storage. Local variables are stored on the stack but instead of utilizing pop or push, we use a stack pointer (a pointer in a special register pointing to the head of the stack). This is more efficient as instead of having to pop multiple elements we simply change the stack pointers value. While there are many ways of organising a stack frame the common method allows native interaction with C compiler-generated code. Upon entering a new activation record the stack pointer is increased in size by the size of the current frame. Meaning all local arguments can be accessed via their relative value to the stack pointer because the compiler has calculated storage required by each method invocation.

Queues are used by applications to store incoming data, they can be used to process synchronisation operating systems and as well as being generally used in programs queues used by the CPU for job and disk scheduling. Overall queues are used in operating system features like multiprogramming platform systems, scheduling algorithms and various built-in applications. The “round-robin” technique is implemented using queues, used for the time-sharing system and more, the circular queue is used to implement these algorithms.

2.2 Trees and Heaps

These were the basis for the functionality of the Huffman coding and so by proxy applies to all the previously mentioned compression methods made use of by operating systems. While a heap can refer to an area of memory used by the operating system, I refer to the tree-based data structure. The heap is the basis for the functionality of the priority queue and the binary heap/ binary tree.

The binary tree is the basis of the “Buddy memory allocation” where we manage memory in two increments. The method is fast with the maximal number of compactions required equal to $\log_2$. The binary tree represents memory blocks, each block “buddy”/corresponding block is located via an OR of the block's address and size, if the correct amount of space is available it is taken, however, if not then it is split into buddies and this process is repeated until everything is stored. However, this technique leads to internal fragmentation being high when space is requested is slightly larger than a small block but much smaller than the next size up. Meaning defragmentation will be necessary. When freeing space the

tree is followed up in reverse to the previous process, each buddy is reconnected via their OR until the largest block is formed and returned.

An operating system's disk file system is maintained as a tree structure with directories acting as tree nodes with files acting appropriately as leaves. The tree structure specifically is used because of its ease of addition and deletion. Searching binary trees is generally quite fast with a time complexity of O(log(n)) which outranks the heap in terms of searching speed, however, heaps can retrieve data in sorted order without the need for a priority queue. The tree data structure can be used by operating systems for syntax validation in many compilers as well as trees playing a major role in networking such as common internet protocols and storing router tables for high bandwidth routers, without these preinstalled with the operating system the device wouldn't be able to access networks and the internet as a whole.

2.3 Hash Tables

Hash functions are used in operating system features such as file systems, directory listings and page tables. Hashed page tables make use of hash tables to ensure every entry in the hash table has a linked list of elements hashed to the same location where the hashed value is the virtual number, this avoids any potential collisions as the same value of a hash function can be used for different page numbers. If the first element of the linked list matches the virtual number then the corresponding page frame is used to create the physical address. Clustered Page Tables function the same except that each entry in the hash table refers to various pages rather than a singular page.

In file systems, there should be a hash table per directory so that when following a pathname the root hash table is returned and query it for the first directory in the path. If it's a directory then the process is repeated with the next hashtable and part until the end of the path. Seeing as how hash tables are inherently unordered, you sort it within memory. In terms of directory listings/ implementation, the process is initially performed using a linked list where each file contains the pointers to the data blocks which are assigned to it and the next file in the directory. However, a hash table can be implemented alongside the linked lists to become more efficient (so the entire list isn’t searched only the hash table). This is done by assigning key-value pairs for each file in the directory and storing it with a hashtable, allowing for the key to be determined by applying the hash function on the filename while the key points to the corresponding file stored in the directory.

References

Knuth, D.E., 1985. Dynamic huffman coding. *Journal of algorithms*, *6*(2), pp.163-180.
Moffat, A., 2019. Huffman coding. *ACM Computing Surveys (CSUR)*, *52*(4), pp.1-35.
Nag, A., Biswas, S., Sarkar, D. and Sarkar, P.P., 2011. A novel technique for image steganography based on DWT and Huffman encoding. *International Journal of Computer Science and Security,(IJCSS)*, *4*(6), pp.497-610.
Nourani, M. and Tehranipour, M.H., 2005. RL-Huffman encoding for test compression and power reduction in scan applications. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *10*(1), pp.91-115.
Tyagi, N., 2021. *Real world applications of Huffman Coding*. [online] Medium. Available at: <https://nishikatyagi.medium.com/real-world-applications-of-huffman-coding-b6cf46fd0663> [Accessed 2 December 2021].
Stack Overflow. 2021. *Stack Overflow - Where Developers Learn, Share, & Build Careers*. [online] Available at: <https://stackoverflow.com/> [Accessed 16 November 2021].
Shobha Rani, N., 2021. *The Role of Data Structures in Multiple Disciplines of Computer Science- A Review*. International Journal of Scientific & Engineering Research, pp.3-4.