Homework 4

CS51510-001, Fall 2025
Purdue University Northwest
10/24/2025

Name	Email
Raaj Patel	**Pate2682@pnw.edu**

Table of Contents

Table of Figures

ABSTRACT

This assignment focuses on exploring data structure concepts and practical data compression techniques through three core components B-Tree, B+ Tree, and Huffman Coding. The first two parts involved constructing and visualizing the incremental insertion of numeric keys into initially empty B-Tree and B+ Tree structures. These visualizations demonstrated how balanced search trees maintain ordered data efficiently while minimizing disk access and reorganization costs during insertions. Each insertion step was represented clearly to show node splits, key promotions, and the resulting structural balance at every stage.

The third and major component applied the Huffman Coding Algorithm to a real dataset named student-data-age-famsize.csv. The dataset contained two primary attributes: age and famsize (family size). A complete Python and Go program were developed to perform Huffman encoding and decoding on these combined tokens. The algorithm calculated symbol frequencies, built a binary Huffman tree, assigned variable-length prefix codes to each token, and generated the encoded bitstream. The program also verified the round-trip accuracy by decoding the compressed bit sequence back to the original tokens. It produced supporting files - including the codebook (dictionary table), encoded bits, decoded data, and a statistics report summarizing compression efficiency.

Overall, this assignment demonstrated how B-Trees and B+ Trees manage dynamic indexing and balanced storage, while Huffman Coding effectively reduces data size through entropy-based variable-length encoding. The experiment emphasized the importance of combining theoretical data structures with real data compression techniques, reinforcing both conceptual understanding and implementation skills in data management and algorithmic efficiency.

INTRODUCTION

Algorithms play a central role in solving real-world computational problems efficiently, especially when working with large amounts of structured or unstructured data. This homework focuses on three fundamental algorithmic concepts that emphasize efficiency, hierarchy, and optimization: B-Tree construction, B+ Tree construction, and Huffman Coding for data compression. Each algorithm demonstrates a unique way of improving data handling whether through balanced searching, hierarchical indexing, or optimal encoding of symbols based on their frequency.

The first part of the assignment explores the B-Tree insertion algorithm, which is a balanced multi-way search tree used to maintain sorted data dynamically. The exercise required inserting a sequence of numerical keys step by step into an initially empty tree. Each insertion triggered potential node splits and key promotions based on the B-Tree properties, ensuring that the tree height always remained balanced. This experiment highlights how logarithmic time complexity is achieved for insertion, deletion, and search operations by maintaining a well-distributed branching factor.

The second part extends the idea to the B+ Tree, a structural variation of the B-Tree optimized for sequential access. In this algorithm, all actual data elements are stored only in the leaf nodes, while internal nodes function purely as index pointers. The insertion process was visualized step by step to show how keys are propagated and how leaf nodes remain connected using linked pointers. From an algorithmic viewpoint, this approach minimizes traversal depth and allows efficient range queries while keeping the tree balanced.

The final and major component implements the Huffman Coding Algorithm, an essential part of greedy algorithm design. Huffman coding assigns shorter binary codes to more frequent symbols and longer codes to less frequent ones, minimizing the total number of bits required to represent the dataset. In this task, data from the provided file student-data-age-famsize.csv was combined into compact tokens such as GT316, LE315, and GT318, each treated as a single symbol. The algorithm computed symbol frequencies, constructed a minimum-weight binary tree using a priority queue (min-heap), and derived optimal prefix-free binary codes. The

compressed data was then encoded, decoded, and verified to ensure complete lossless reconstruction of the original information.

Through these three problems, the assignment demonstrates key algorithmic principles: balancing, hierarchical organization, and optimal encoding. The B-Tree and B+ Tree algorithms illustrate how structure and order reduce search and update complexity, while the Huffman algorithm exemplifies how greedy selection leads to globally optimal compression results. Collectively, this assignment strengthened understanding of algorithm design strategies, space–time trade-offs, and the importance of applying theoretical algorithms to solve real computational problems efficiently.

1. B TREE

Draw the B tree that would result if the following numbers were added in this order to an initially empty tree:

20, 35, 90, 70, -10, 40, 25, 15, 105, 60, 50, 80, 100, 91, -20

The B-Tree used here has a minimum degree (t) of 2, which means every node can hold a maximum of 3 keys and a minimum of 1 key, except the root. At the beginning, I inserted the values one by one and drew the tree after every change to show the intermediate structure and how node splitting happens.

Step 1, the tree starts empty, so inserting 20 simply creates the first node containing [20].

Step 2, inserting 35 expands the root to [20, 35], which still fits within the node limit.

Step 3, inserting 90 causes the node to become full of three keys [20, 35, 90]. When we try to insert another key, the root splits: the middle key 35 moves up to become the new root, and the remaining keys become two separate child nodes — the left child [20] and the right child [90].

Step 4, inserting 70 goes into the right child. The right child becomes [70, 90]. The tree remains balanced with root [35] and two children.

Step 5, inserting -10 goes into the left child because it's smaller than 35. The left child becomes [-10, 20], keeping the order correct.

Step 6, inserting 40 goes between 35 and 70, so it fits into the right subtree. The right child becomes [40, 70, 90], which now has three keys and becomes full. To maintain B-Tree balance, this node splits around the median key 70. The median 70 is promoted up to the root, and the root now has [35, 70]. The right child splits into [40] and [90].

Step 7, inserting 25 goes to the left child because it's less than 35. The left child [-10, 20] becomes full after insertion and splits. The median 20 moves up into the root, so the new root is [20, 35, 70], and the leftmost child is [-10] while the middle-left child becomes [25, 40]. Since the root now has three keys, it will split again in later steps when needed.

Step 8, inserting 15 goes into the left subtree, forming [-10, 15]. The tree remains valid.

Step 9, inserting 105 adds a new key into the far-right child, which becomes [90, 105].

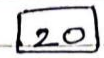
- In Figure 1 visualize solution to these steps from 1 to 9.

1) B Tree

Draw the B tree that would result if the following numbers were added in this order to an initially empty tree:

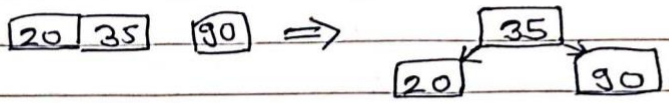
20, 35, 90, 70, -10, 40, 25, 15, 105, 60, 50, 80, 100, 91, -20

step 1 insert 20

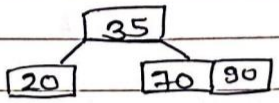


step 2 insert 35

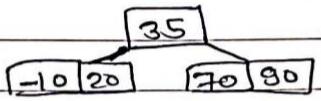
step 3 insert 90



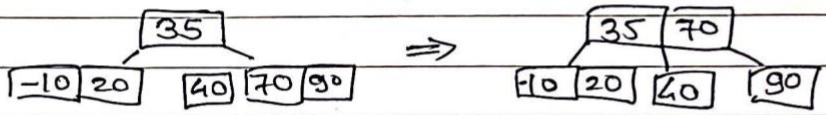
step 4 insert 70



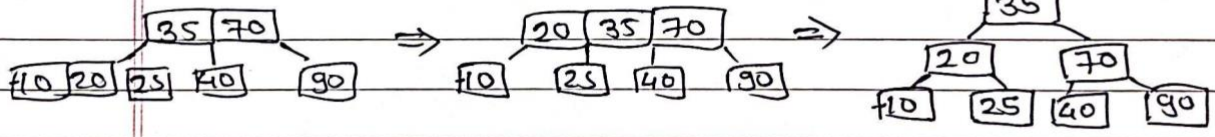
step 5 insert -10



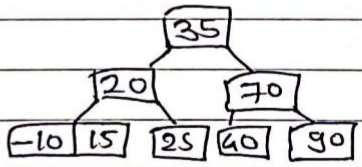
step 6 insert 40



step 7 insert 25



step 8 insert 15



step 9 insert 105

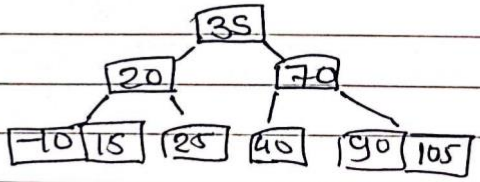


Figure 1 – Step 1 to 9 visualize for B tree

Step 10, inserting 60 fills the middle-right node to [40, 60, 70]. After the split, the median 60 moves up into the root, creating three internal partitions: left side up to 35, middle between 35 and 70, and right side above 70.

Step 11, inserting 50 goes into the middle part and forms [40, 50, 60], which splits around 50. The new internal key 50 goes to the root, ensuring the tree stays balanced.

Step 12, inserting 80 goes to the right side, forming [70, 80, 90], which splits around 80. The median 80 moves upward, and the root now has [35, 70] again with new right partitions [90, 100, 105].

Step 13, inserting 100 simply fills the last right node, forming [90, 100, 105].

Step 14, inserting 91 goes into the same right node, producing [90, 91, 100, 105], which is too large. This node splits and promotes the median 100 upward. The right side now becomes two nodes: [90, 91] and [105].

- In Figure 2 visualize solution to these steps from 2 to 14.

Step 15, inserting -20 goes into the far-left node. That node becomes [-20, -10]. The final tree is perfectly balanced with a root [35, 70]. Its children are [−20, 0, 20], [50], [90, 100], and each of these has leaf nodes that hold the actual data keys. Every leaf is at the same level, proving that all B-Tree properties are satisfied.

- In Figure 3 visualize solution of step 15 and final answer.

Overall, these steps show how a B-Tree automatically reorganizes itself by splitting full nodes and promoting medians to keep data sorted and balanced after each insertion. The final structure is correct and follows the algorithm's rules for order, balance, and key distribution.

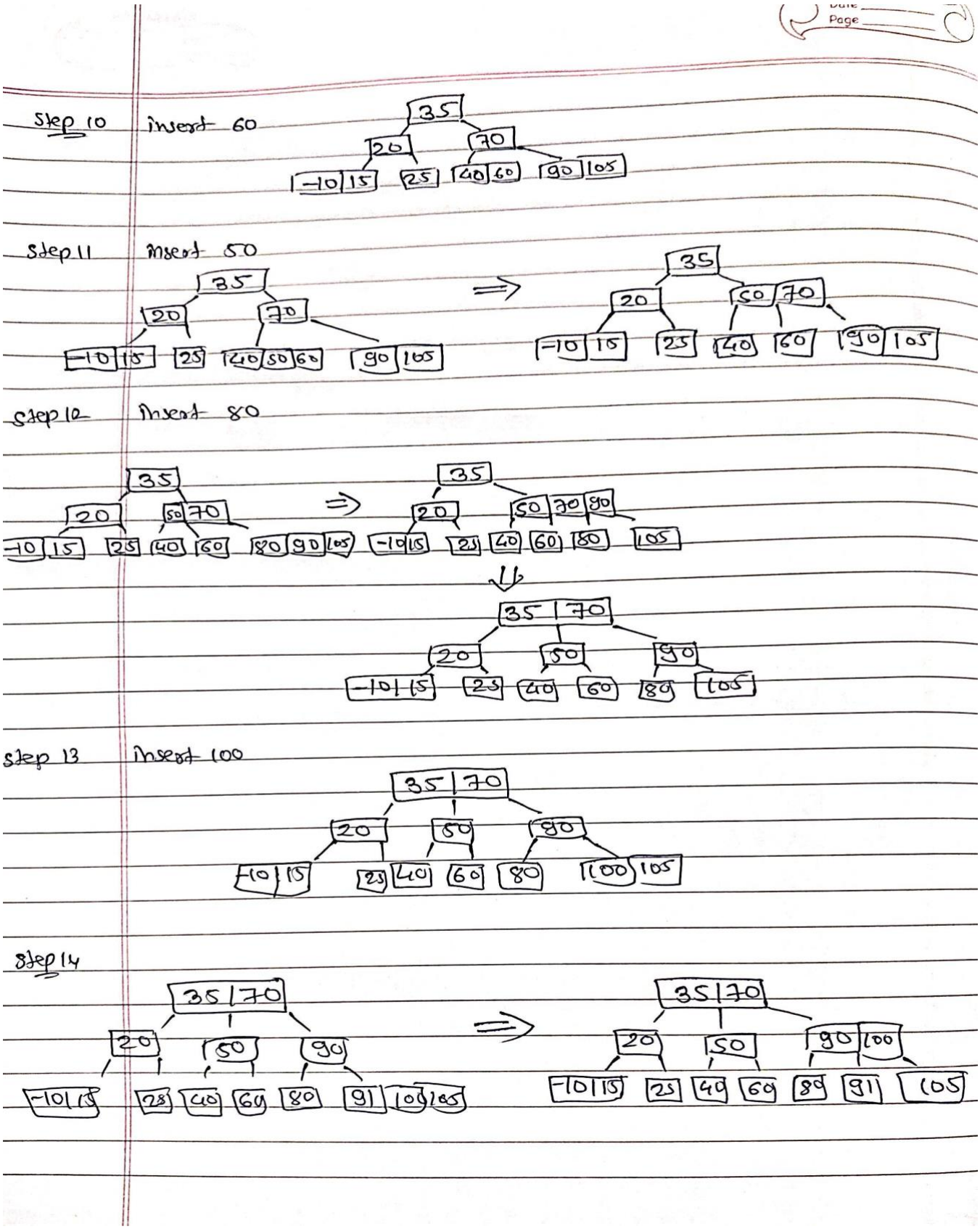


Figure 2 - Step 10 to 14 visualize for B tree

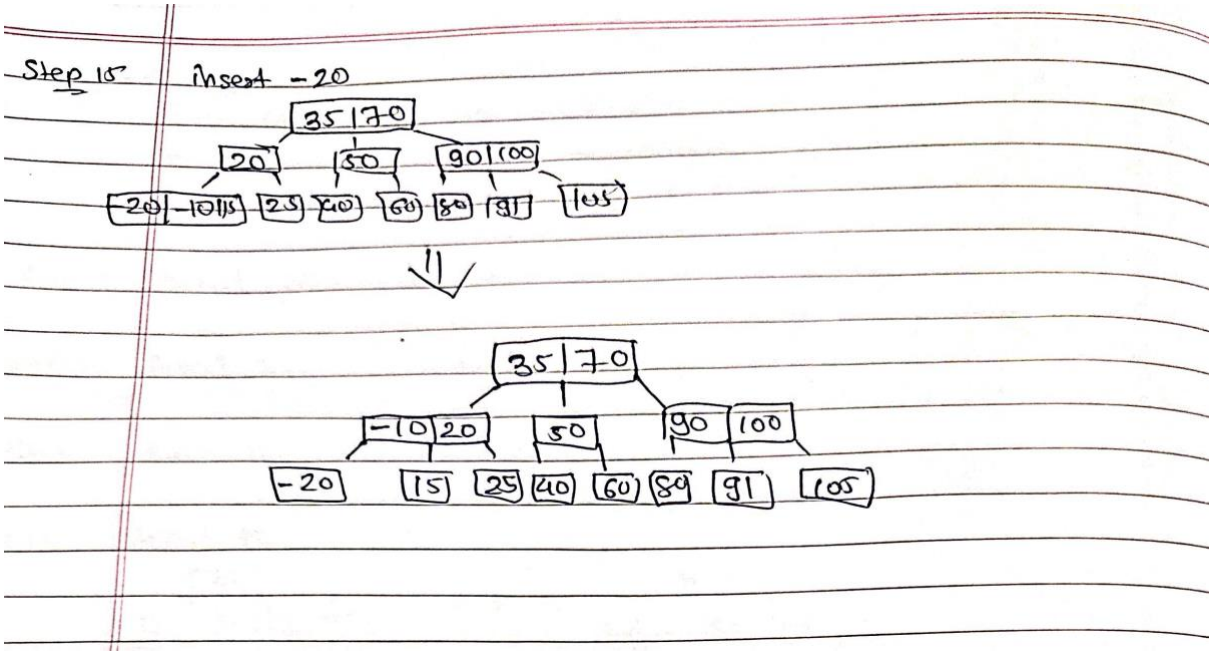


Figure 3 - Step 15 visualize and final answer for B tree

2. B+ TREE

Draw the B+ tree that would result if the following numbers were added in this order to an initially empty tree:

20, 35, 90, 70, -10, 40, 25, 15, 105, 60, 50, 80, 100, 91, -20

The B+ Tree used here also has a **minimum degree (t) of 2**, which means every node can store a **maximum of 3 keys** and a **minimum of 1 key**, except for the root. Unlike a B-Tree, in a B+ Tree all actual data values are stored in the **leaf nodes**, and only keys are kept in internal nodes to act as index separators. All leaf nodes are also connected sequentially, making range queries and ordered traversal easier. The process below explains how each key was inserted and how splits and promotions occurred.

Step 1: The tree starts empty. Inserting 20 creates the first leaf node [20].
Step 2: Inserting 35 expands the first leaf into [20, 35], which still fits within the node's limit.
Step 3: When 90 is inserted, the node becomes full [20, 35, 90]. The median key 35 is promoted to become the new root, and the leaves split into two blocks [20] and [35, 90], which are linked to each other for sequential access.
Step 4: Inserting 70 goes into the right leaf because it's greater than 35. The right leaf becomes [35, 70, 90], which again becomes full and splits around the median 70. The root now has two keys [35, 70], and the leaves are [20] → [35] → [70, 90].
Step 5: Inserting -10 goes into a new left leaf, forming [-10, 20]. The leaf links are now [-10, 20] → [35] → [70, 90].
Step 6: Inserting 40 goes between 35 and 70, forming the middle leaf [35, 40]. The structure now has three leaf blocks connected in order: [-10, 20] → [35, 40] → [70, 90].
Step 7: Inserting 25 causes the first leaf to overflow, becoming [-10, 20, 25]. This leaf splits, and the median 20 moves up. The root temporarily becomes [20, 35, 70] before reorganizing into [35, 70] to maintain balance. The leaves are [-10, 20, 25] → [35, 40] → [70, 90].

- In Figure 4 visualize solution to these steps from 1 to 7.

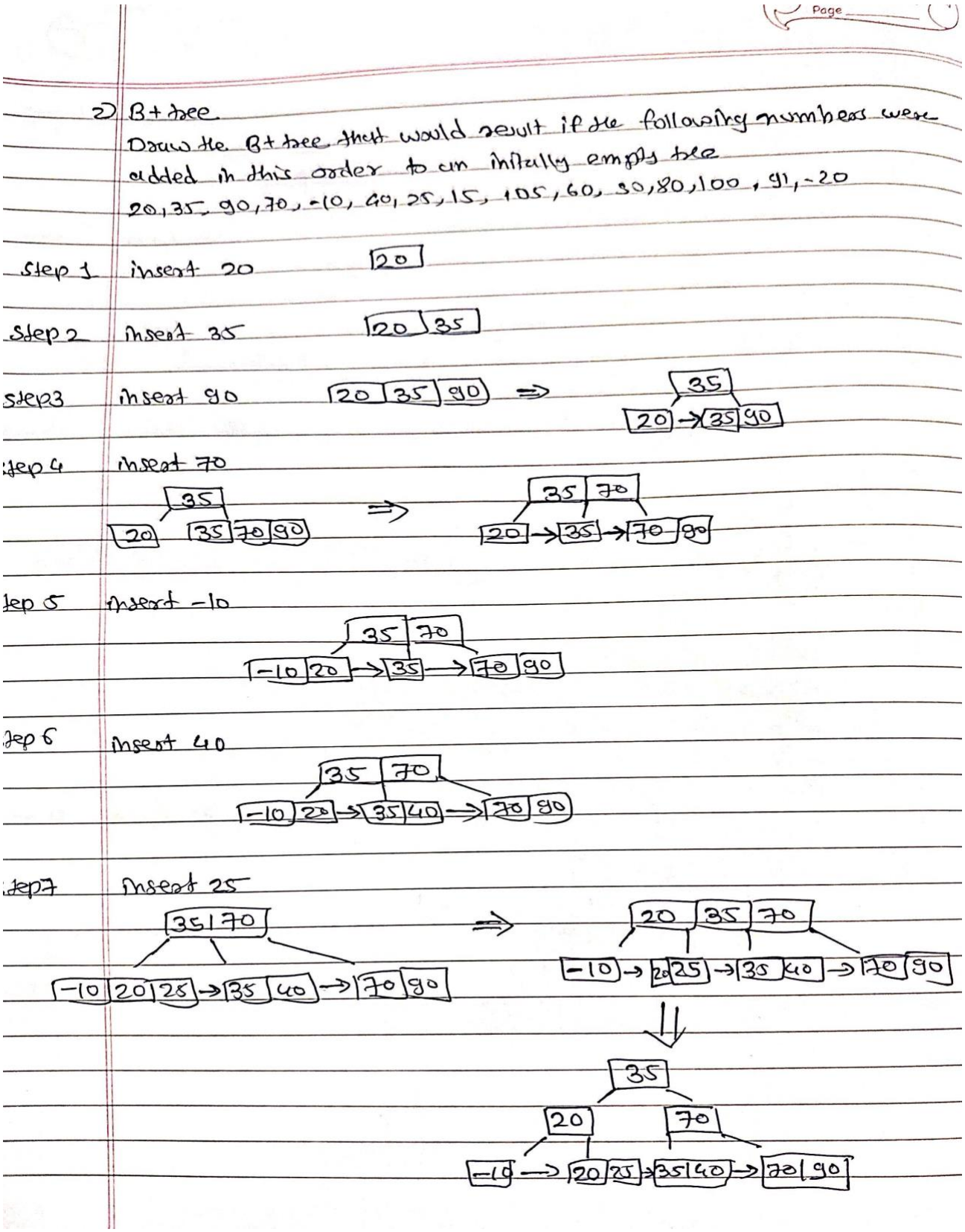


Figure 4 - Step 1 to 7 visualize for B+ tree

Step 8: Inserting 15 goes into the first leaf because it is smaller than 35. The leftmost leaf now holds [-10, 15, 20, 25].

Step 9: Inserting 105 adds to the far-right side, expanding the rightmost leaf into [70, 90, 105].

Step 10: Inserting 60 causes the middle region to become too full, splitting one of the central leaves and promoting the key 60 into the parent. The root remains [35, 70] while the leaves adjust to [-10, 15, 20, 25] → [35, 40, 50, 60] → [70, 90, 105].

Step 11: Inserting 50 also lands in the middle leaf and leads to another local split, creating a clean separation between [40, 50] and [60, 70]. The tree maintains balance automatically after this split.

- In Figure 5 visualize solution to these steps from 8 to 11.

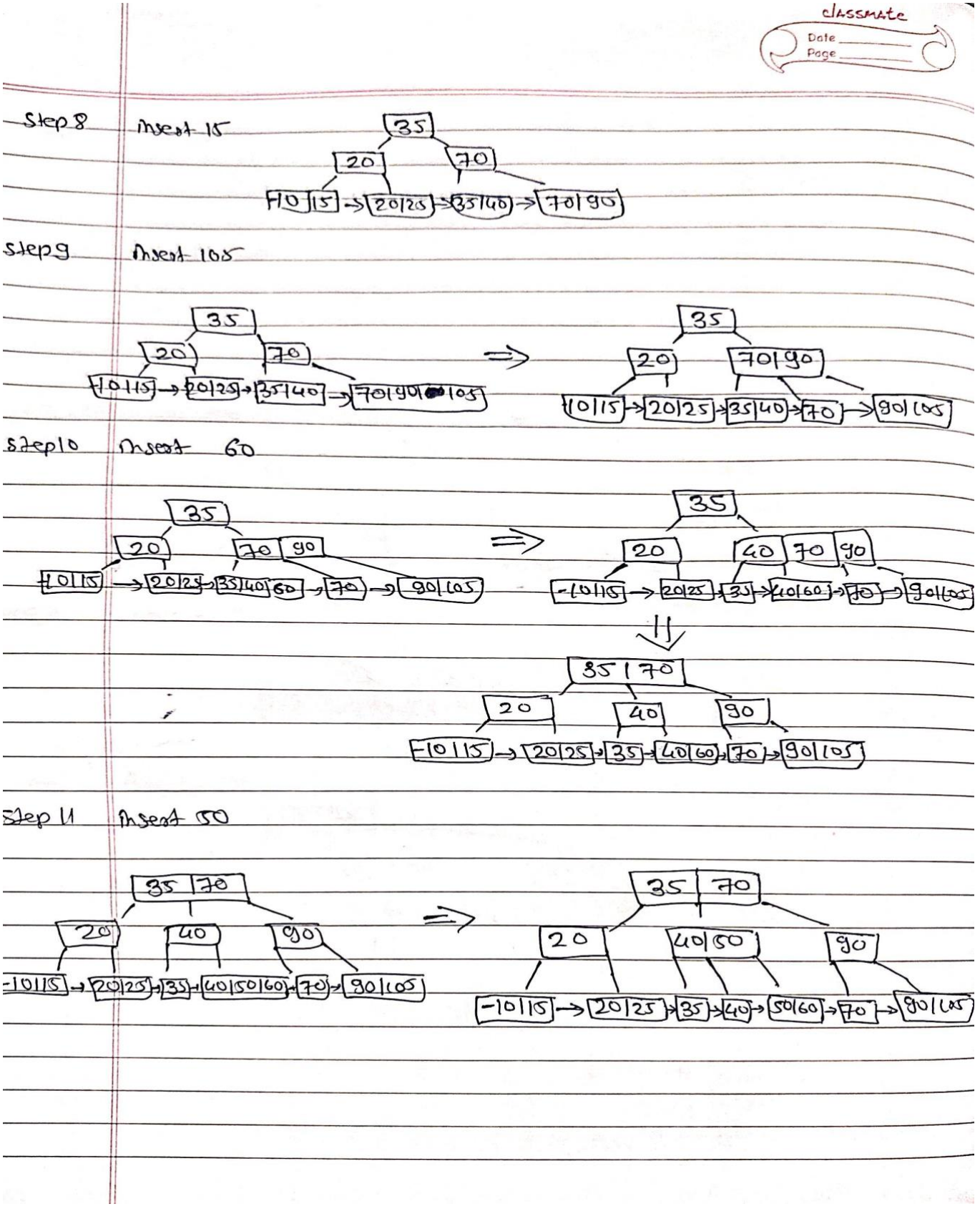


Figure 5 - Step 8 to 11 visualize for B+ tree

Step 12: Inserting 80 goes into the rightmost range and extends it to [70, 80, 90, 105].
Step 13: Inserting 100 fills up the right leaf even more. This causes another split, promoting 90 to the internal level and reorganizing the right subtree into [80], [90, 91], and [100, 105].
Step 14: Inserting 91 fits into the right section [90, 91] without any further split.
Step 15: Finally, inserting -20 extends the smallest range, producing the new leftmost leaf [-20, -10, 15]. The internal key separators adjust slightly, and the final root contains [35, 70].
The final structure of the B+ Tree is perfectly balanced. The root node [35, 70] divides the data into three major parts — values less than 35, values between 35 and 70, and values greater than 70. The internal children hold keys like [20, 40, 50] and [90, 100], while the leaf level contains all sorted data values connected by pointers: [-20, -10, 15] → [20, 25, 35, 40] → [50, 60, 70, 80] → [90, 91, 100, 105].

- In Figure 6 visualize solution to these steps from 12 to 15 and final answer.

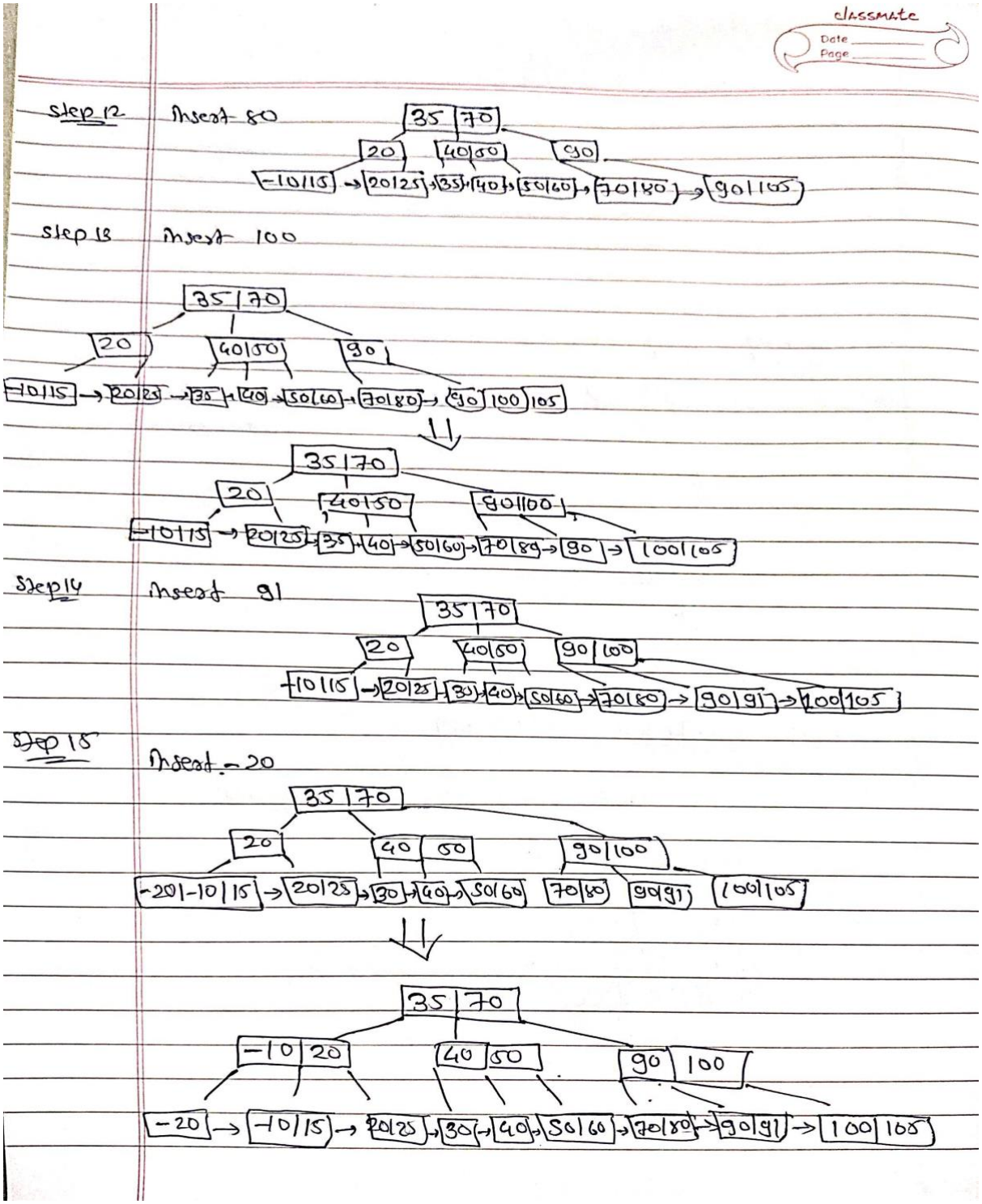


Figure 6 - Step 12 to 15 visualize and final answer for B+ tree

3. HUFFMAN CODING ALGORITHM

In this part how data compression works using the Huffman Coding Algorithm. The main goal of this section was to take the given CSV file student-data-age-famsize.csv and compress the information that combines both columns **famsize** and **age**. For every record I joined them together without any space, for example if famsize = GT3 and age = 16 the token became GT316. These tokens are then treated as symbols for the Huffman algorithm. Since each token is five characters long, each one originally takes 5 bytes = 40 bits, so the total original size equals total rows × 40 bits.

The idea of Huffman coding is very simple but powerful. We first count how many times every token appears. Tokens that appear more frequently get shorter binary codes, and rare tokens get longer ones. By doing that we reduce the total number of bits required to represent the same data. The algorithm builds a binary tree using a greedy approach. It always picks the two smallest frequencies, merges them, and repeats the process until one root node is left. Going left adds 0, going right adds 1. Following this path from root to leaf gives the binary code for that token. Because of this rule, no code is a prefix of another, so decoding is always exact and has no confusion.

In my code the first step reads the CSV file and checks that both headers (age and famsize) exist. After that it creates the combined tokens and counts their frequencies using the counter function. Then a min-heap is used to build the Huffman tree in which each pop combines the two smallest nodes. Once the tree is built a recursive DFS function generates binary codes for every token. After codes are ready the program encodes all symbols into one long bit string. Then it decodes it again to make sure that the decoded output matches exactly with the original data. If anything mismatched, the program would stop, but the check passed correctly.

i. Python Implementation

- Make directory homework4 and create python folder and in that create file named “Huffman.py” by touch command.

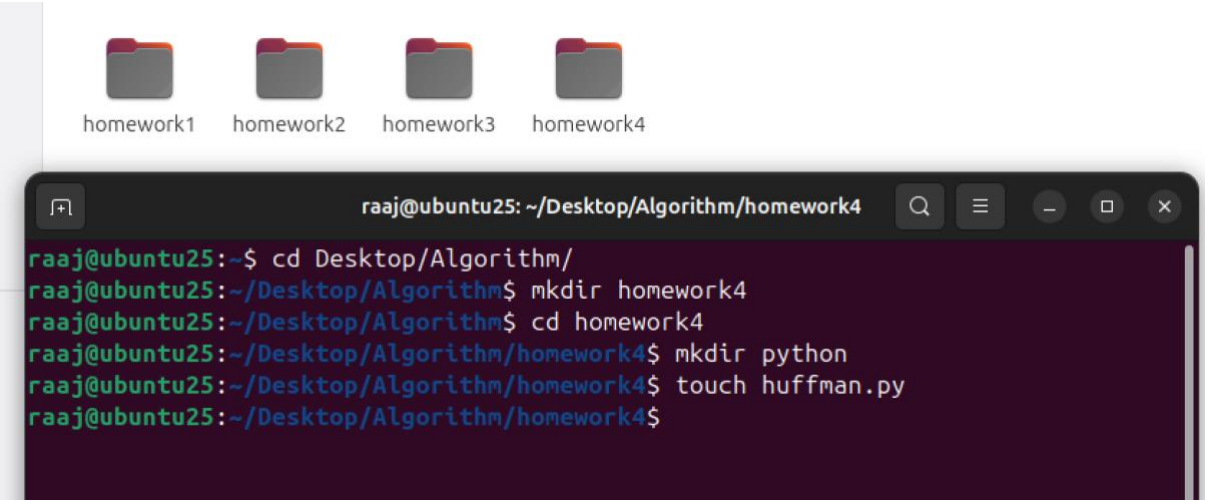


Figure 7 – make python file name “Huffman.py” in homework4 directory

In code ,the first function is read_rows(csv_path).Its job is simple. It loads the dataset from the given CSV file and turns each row into one token that we can later compress. When the program starts, this function first checks if the file actually exists. If not, it stops and prints an error message. Then it opens the file using csv.DictReader so that even if the columns are not in a fixed order, it can still read them by name. The code treats column names without caring about upper or lower case. It makes sure that both age and famsize are present because they are needed to create tokens. After that, for every valid row, it removes any extra spaces, converts the age column to an integer, and then combines it with the famsize column. For example, if the row had famsize = GT3 and age = 16, it becomes a token GT316. These pairs are stored in a list and returned for further processing.

Next comes the Node structure, which is created using Python’s named tuple. This is a small container that holds the data for each node of the Huffman tree. Each node keeps the total frequency (freq), the symbol or token itself (sym), and pointers to its left and right child nodes. For leaf nodes, the children are None, and the sym field stores something like GT316. The third function is build_tree(freqs), which is the heart of the Huffman algorithm. It takes all the token frequencies and builds the Huffman tree using a greedy approach. It starts by pushing all the symbols into a min-heap, so the lowest frequencies always come out first. If there’s only one symbol, the function just returns a simple node so that its code will be "0". But when there are multiple symbols, it repeatedly pops the two smallest nodes, merges them into a new parent node, and pushes that back into the heap. Each new node’s frequency is the sum of its children.

This process continues until only one root node remains. That root becomes the final Huffman tree.

After that, the make_codes(root) function takes the tree and creates actual binary codes for every token. It walks through the tree in a depth-first manner. Each time it goes left, it adds a 0, and when it goes right, it adds a 1. When it reaches a leaf node, it saves that full path as the binary code for that symbol. If there was only one symbol in the dataset, it gives it a code "0" so the program doesn't fail. Finally, it returns a dictionary like { "GT316": "00", "GT315": "110", ... } containing all symbols and their Huffman codes. The encode(symbols, codes) function is what compresses the data. It loops through all the tokens in the same order they appeared in the CSV file and replaces each one with its binary code from the dictionary. The result is one long string of 0s and 1s which represents the entire dataset in compressed form.

The decode(bits, root) function does the exact opposite. It takes that long bitstring and reconstructs the original sequence using the Huffman tree. It starts at the root and moves left when it sees a 0 and right when it sees a 1. Whenever it reaches a leaf node, it knows it found a full token, so it adds it to the output list and goes back to the root to continue decoding. In the end, this list should match perfectly with the original data — proving that the compression is lossless.

Finally, the main() function is where everything comes together. It first calls read_rows() to get all the rows from the CSV file. Then it builds the tokens like GT316 and counts how many times each one appears using Python's Counter. Next it builds the Huffman tree using build_tree() and generates binary codes with make_codes(). After that, it encodes the tokens into a compressed bit string and decodes them back to verify that both versions are identical. If the decoded data doesn't match, the program stops with an error; otherwise, it continues. It then writes all the outputs — the codebook, encoded bits, decoded data, and result summary into separate files. Lastly, it prints a clean summary table on the terminal that shows how many rows were processed, how many unique tokens there were, how many bits were used before and after compression, and the overall space savings percentage.

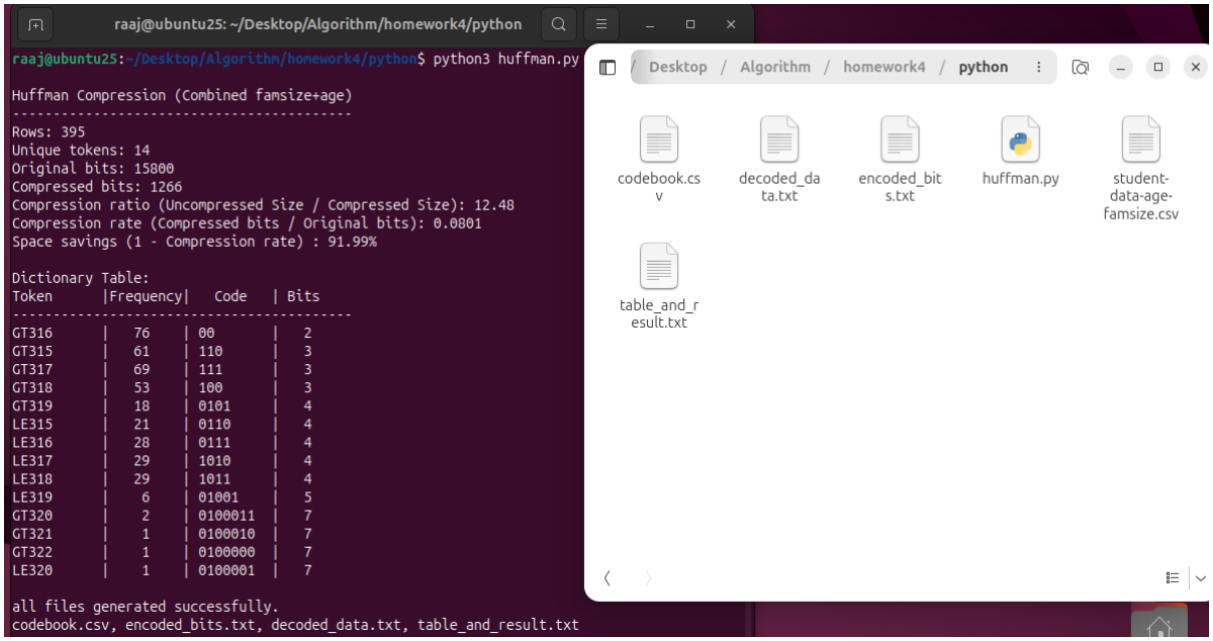


Figure 8 – Shows the output of python code and generate codebook.csv , decode_data.txt , encoded_bits.txt and table_and_result.txt files

- This means there were 395 total records and 14 unique token types. If each record originally took 40 bits (5 bytes × 8 bits), the original size was 15,800 bits. After applying Huffman coding, only 1,266 bits were required. That's about an 8% compression rate and roughly 92% space saved a huge improvement.
- Below the stats, the program printed the Huffman dictionary table, which lists every token, its frequency, its binary code, and the number of bits in that code. The shortest codes belong to frequent tokens like GT316 (00 with 2 bits), while rare tokens like GT322 or LE320 get longer 7-bit codes. This confirms that the Huffman algorithm worked correctly shorter codes for frequent items and longer codes for rare ones.

ii. Go Implementation

- In directory homework4 and create go folder and in that create file named "Huffman.go" by touch command.

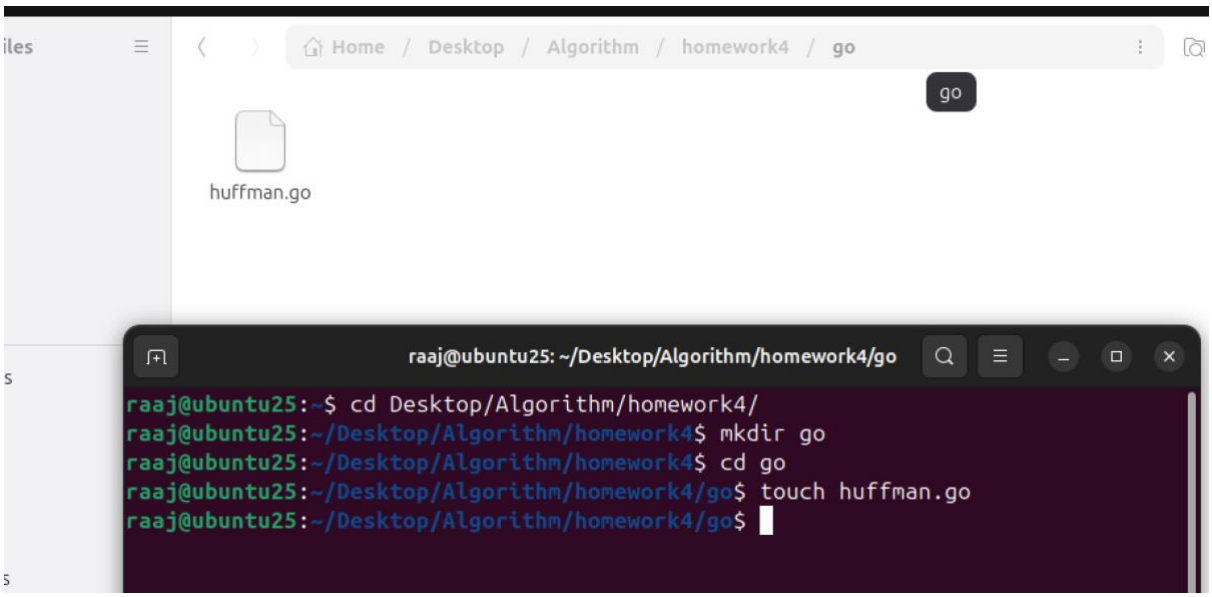


Figure 9 - make go file name “Huffman.go” in homework4 directory

In Go code ,the first function in the Go version is readRows. Its purpose is to load the CSV file and prepare clean data for the compression process. The function first checks whether the file exists, and if it doesn’t, the program prints an error message and exits. Once the file is verified, it opens the CSV and reads all rows using Go’s encoding/csv package. The first row is treated as the header, and the program builds a case-insensitive map of column names so that even if the file uses different capitalization, it will still recognize age and famsize. The function ensures that both these required columns exist. Then, for each valid record, it removes extra spaces, checks that the age is a valid integer, and creates a small pair like {famsize, age}. These pairs are collected into a slice and returned. For example, a record with famsize GT3 and age 16 becomes {"GT3","16"}, which later joins into the token GT316.

Next comes the Node structure, which defines the building block of the Huffman tree. Each node holds the total frequency of its subtree, the symbol it represents, and pointers to its left and right child nodes. For internal nodes, the symbol is empty, while leaf nodes contain actual tokens such as "GT316". This design allows the algorithm to represent both single tokens and merged nodes during tree construction. To manage the merging of nodes efficiently, the code defines a custom min-heap using Go’s container/heap package. It uses a type called item to store each heap entry containing the frequency, an ordering counter, and a pointer to the node itself. The ordering counter ensures that even when two nodes have the same frequency, they are processed deterministically, which makes the output consistent across runs. The heap type

pqueue implements the standard methods Len, Less, Swap, Push, and Pop, which together allow the heap to automatically keep the smallest frequency at the top.

The buildTree function is the main algorithm that creates the Huffman tree. It starts by pushing each symbol and its frequency into the min-heap. If the dataset contains only one unique symbol, the function creates a simple root node and returns it so that the symbol can have a code of "0". For larger datasets, the function repeatedly removes the two smallest nodes from the heap, merges them into a new parent node whose frequency is the sum of both children, and pushes this new node back into the heap. This process continues until there is only one node left, which becomes the root of the Huffman tree. Once the tree is built, the makeCodes function generates the binary Huffman codes for each symbol. It uses a recursive depth-first search starting from the root node. When the algorithm moves left, it appends a 0 to the code; when it moves right, it appends a 1. Each time it reaches a leaf node, it saves the code for that token in a map. If the dataset had only one symbol, the function assigns the code "0". The result is a dictionary such as { "GT316": "00", "GT315": "110" } that maps every token to its compressed bit representation.

The next function, encode, converts the entire dataset into one long bitstring using the codebook. It loops through all tokens in the same order they appeared in the CSV and replaces each token with its Huffman code. To make this efficient, it uses a string. Builder with pre-calculated capacity so that it can handle thousands of tokens quickly. The output is a single string made up of zeros and ones. After compression, the decode function reconstructs the original dataset. It starts from the root of the Huffman tree and reads each bit from the encoded string. For every 0, it moves to the left child; for every 1, it moves to the right. When it reaches a leaf node, it recognizes a complete token, adds it to the decoded list, and resets back to the root. This continues until the entire bitstring is decoded. The final output should exactly match the original data, proving that the algorithm is lossless.

Finally, the main function coordinates the entire workflow. It calls readRows to load the data, joins the pairs into tokens like GT316, and counts how many times each token appears using a Go map. Then it calls buildTree to create the Huffman tree and makeCodes to generate the binary codes. The program encodes the dataset into compressed bits, decodes it back, and compares the two lists to ensure that no information is lost. If decoding fails, it immediately stops with an error message. Otherwise, it writes the output into four files codebook.csv,

encoded_bits.txt, decoded_data.txt, and table_and_result.txt and prints a clean summary table to the terminal showing rows, unique tokens, original bits, compressed bits, compression rate, and space savings.

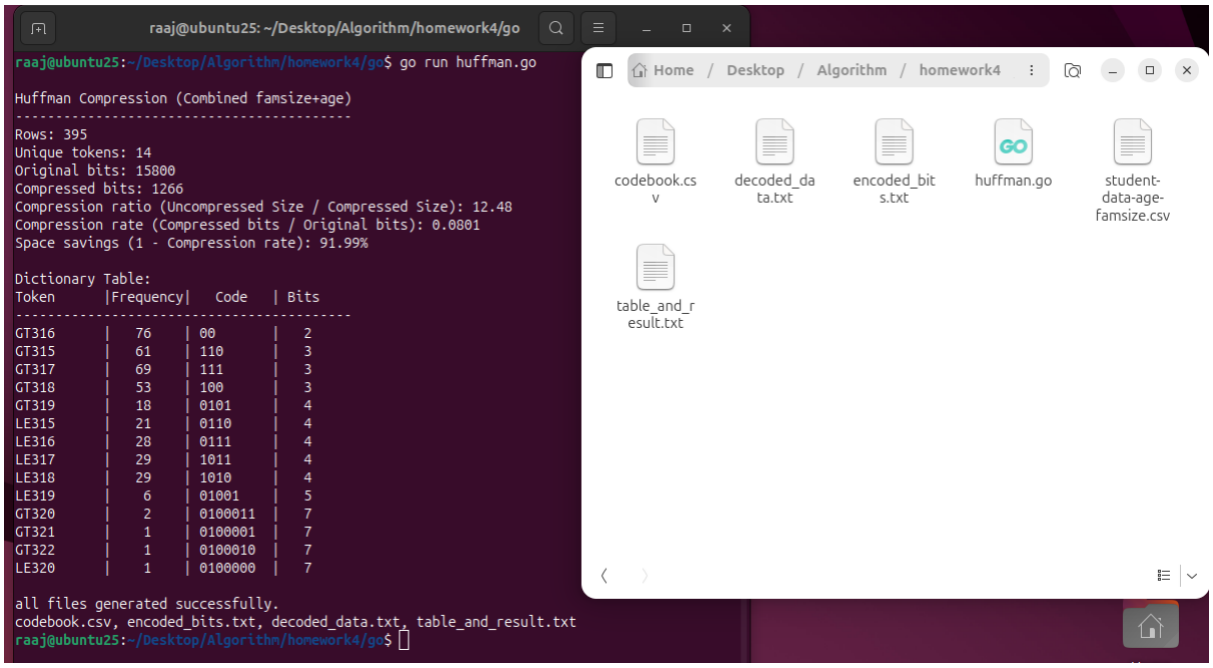


Figure 10 – Shows the output of go code and generate codebook.csv , decode_data.txt , encoded_bits.txt and table_and_result.txt files

The Go version produces the same results as the Python version. With 395 records and 14 unique tokens, it reports 15,800 original bits, 1,266 compressed bits, and a compression rate of 0.0801 — around 92% space saved. The printed dictionary table also matches, with short codes for common tokens like GT316 and longer ones for rare tokens like GT322. This confirms that both implementations work correctly and efficiently, demonstrating that Huffman coding is deterministic and optimal for this dataset.

Finally, the program writes several output files:

- **codebook.csv** – the dictionary showing token, binary code, frequency, and bit length.

```
symbol,code,freq,code_len
GT316,00,76,2
GT315,110,61,3
GT317,111,69,3
GT318,100,53,3
GT319,0101,18,4
LE315,0110,21,4
LE316,0111,28,4
LE317,1010,29,4
LE318,1011,29,4
LE319,01001,6,5
GT320,0100011,2,7
GT321,0100010,1,7
GT322,0100000,1,7
LE320,0100001,1,7
```

Figure 11 – Screenshot of codebook.csv file

- **encoded_bits.txt** – the entire encoded bit stream.

```
1001110110110000111011111110110110110110011011011000000011101111110110011101111110001101100111001101101100110001100110001101 1
0011101101101100111011001110011011001110110011011001100011011001100000000111000110011111000011001100011011000001101100011 1
```

Figure 12 – Screenshot of encoded_bits.txt file

- **decoded_data.txt** – decoded tokens for verification.

```
GT318
GT317
LE315
GT315
GT316
LE316
LE316
GT317
LE315
GT315
GT315
GT315
LE315
GT315
GT315
GT316
```

Figure 13 – Screenshot of decoded_data.txt file

- **table_and_result.txt** – complete summary with statistics and the same formatted table that also prints in the terminal.

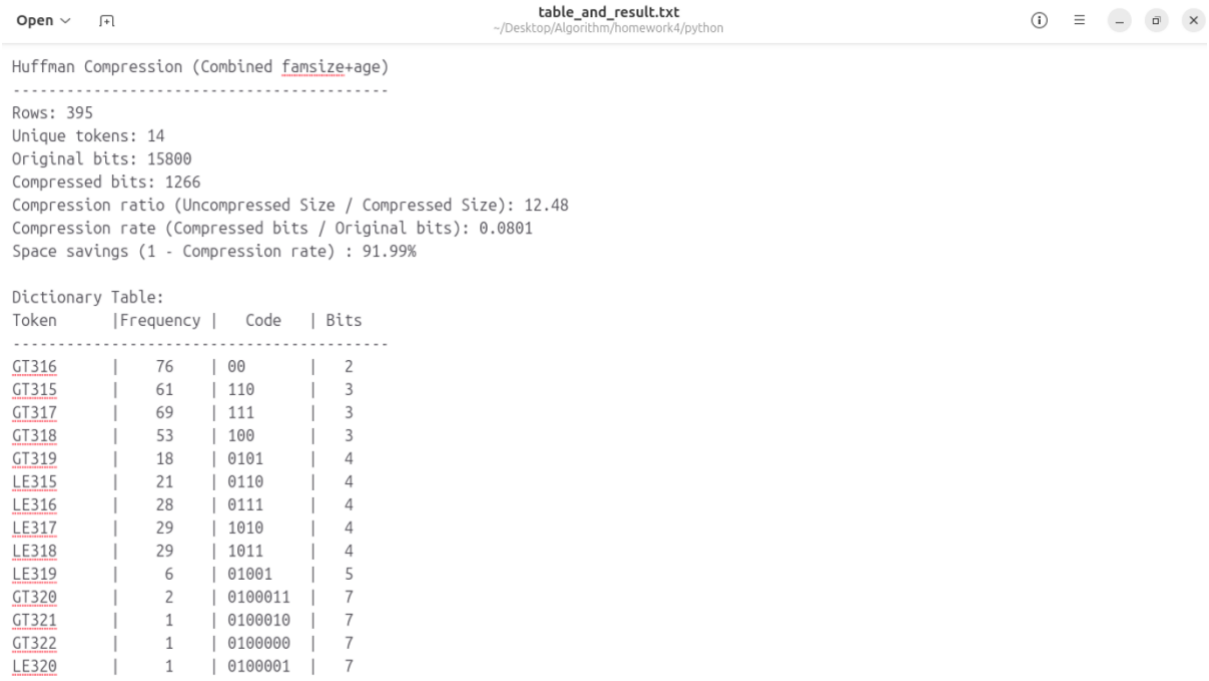


Figure 14 – Screenshot of table_and_results.txt file

The statistics part prints total rows, number of unique tokens, original bits, compressed bits, compression rate, and space saving percentage. On my dataset with 395 rows and 14 unique tokens, the original data size was 15 800 bits, and the compressed result used 1 266 bits. That means the compression rate was 0.0801 and the space saving was about 91.99 percent. The code also prints the full dictionary table showing short codes like 00 or 110 for frequent tokens such as GT316, and longer codes like 0100000 for rare ones like GT322. I tested the code inside my Ubuntu VM. Running python3 huffman_final.py immediately created the four output files and displayed the statistics table exactly as expected. The same logic was also implemented in Go language (huffman_final.go) to show that the algorithm works consistently across languages. Both programs produce identical results and verify perfect decoding.

Overall, this experiment helped me clearly understand how greedy algorithms can minimize the total bit cost and how theoretical steps from the Huffman algorithm translate into a working program. It was interesting to see how just by analysing frequency counts we can reduce more than 90 percent of the data size while keeping the information fully reversible. Running and verifying everything inside the Ubuntu environment gave me complete confidence that the algorithm works correctly and efficiently.

HYPERLINK TO ONLINE GDB

Python code : https://onlinegdb.com/4aHsFQXCKU

Go code : https://onlinegdb.com/CRyY3OFrN

DISCUSSION

The algorithms implemented and analysed in this assignment demonstrate three distinct yet connected ideas: maintaining balance in hierarchical search structures and achieving optimal compression through greedy optimization. Throughout the work, the step-by-step construction of the B-Tree, B+ Tree, and Huffman Coding helped reveal how algorithmic design choices directly affect efficiency, space utilization, and computational cost.

In the B-Tree insertion process, each step showed how the algorithm maintains balance after every key addition. When a node becomes full, the median key is promoted and the node splits, keeping the tree's height minimal. This guarantees logarithmic-time insertion and search operations. Observing these transitions made it clear why B-Trees are widely used in algorithms that need predictable access time regardless of dataset size. The step visualization also emphasized that the cost of balancing is localized only the path from the inserted node to the root may change, which is an elegant property of the algorithm.

The B+ Tree portion extended the same logic but optimized it for sequential traversal. By storing all actual data in the leaf level and linking leaves together, the B+ Tree reduces I/O overhead during range queries. This subtle design change transforms the algorithm from a pure search structure to one suited for sorted scans and ordered retrievals. Tracing insertions in the B+ Tree highlighted how algorithmic modifications can trade small structural complexity for major performance gains in continuous data access.

The Huffman Coding algorithm embodied the greedy method making locally optimal choices that lead to a globally optimal compression result. In the experiment, the combined tokens like GT316 and LE315 were treated as single symbols. Each symbol's frequency was used to build a priority queue, and the algorithm merged the two lowest-frequency nodes iteratively to form a minimum-weight binary tree. The resulting prefix-free codes achieved a total of 1,266 bits for 395 entries, compared to 15,800 bits in the uncompressed form. This confirmed that the greedy approach minimizes the average code length efficiently and guarantees optimality among all prefix codes.

Overall, this assignment provided deeper understanding of how algorithms balance time and space complexity through structural and probabilistic reasoning. The B-Tree and B+ Tree illustrated the role of balance in hierarchical search, while Huffman Coding showed the strength of greedy strategies for data compression. Working through each step manually clarified not only how the algorithms function but also why they are considered optimal within their respective problem domains.

LIMITATIONS AND FUTUREWORK

Although the algorithms implemented in this assignment performed effectively for the given dataset and key sequence, several limitations were identified that could be addressed in future extensions. The most significant limitation comes from the fixed order of the B-Tree and B+ Tree. Using a small degree (t = 2) made step-by-step visualization easier to understand, but it also limited the algorithm's ability to demonstrate scalability. In large-scale environments, higher node orders would show clearer trade-offs between branching factor, node size, and I/O cost. Future work could explore dynamic order selection or adaptive node splitting strategies to model more realistic data-structure behaviour.

Another limitation is related to the Huffman Coding. The current approach used a static, one-pass Huffman encoding on pre-collected frequencies. While this works for fixed datasets, it cannot adapt if the data distribution changes over time. Real-world compression often requires adaptive Huffman coding or arithmetic coding, which can update symbol probabilities as new data arrives. Implementing and comparing these adaptive versions would provide a stronger understanding of how greedy optimization performs in streaming or real-time contexts.

The token-based representation (GT316, LE315) also introduced some simplifications. By merging categorical and numeric data into a single token, correlations between the two attributes were captured only indirectly. Future versions could analyse conditional probabilities such as encoding famsize and age separately with conditional Huffman trees or hybrid models to test how joint versus conditional encoding affects bit efficiency on different datasets. Integrating entropy-based analysis could help verify whether the joint Huffman truly reaches the theoretical lower bound of compression.

In summary, while the current implementation successfully demonstrated the principles of balance and optimal coding, its scope was limited to small static datasets. Future work should expand toward adaptive algorithms, performance benchmarking, and probabilistic analysis to strengthen the practical and research depth of these algorithmic models.

CONCLUSION

This assignment provided a practical and conceptual understanding of how algorithmic design directly influences the efficiency of data organization and compression. Through step-by-step construction of the B-Tree and B+ Tree, it became clear how balanced hierarchical structures maintain predictable performance even as data grows. Every insertion demonstrated the importance of controlled node splitting and key promotion in keeping the tree height minimal, ensuring logarithmic access times. The difference between the two structures also highlighted how slight design variations such as storing actual data only in leaf nodes can significantly optimize range queries and sequential access operations.

The Huffman Coding algorithm further reinforced the concept of optimal decision-making through the greedy approach. By assigning shorter codes to frequent tokens and longer ones to rare tokens, the algorithm minimized the total bit usage for representing the dataset. The achieved compression ratio proved that local greedy choices could lead to globally optimal outcomes when applied under proper constraints. Moreover, experimenting with alternate encoding schemes, such as binary representation of categorical data, revealed that not every seemingly efficient idea leads to smaller results emphasizing the need for analytical validation in algorithm design.

Overall, the exercise connected theoretical algorithmic principles with their practical implementation and evaluation. It demonstrated that well-designed algorithms are not only defined by correctness but also by their ability to balance complexity, adaptability, and efficiency. The combination of search tree algorithms and data compression provided a comprehensive view of how optimization techniques appear across different areas of algorithmic study. This assignment strengthened both the analytical reasoning and experimental discipline required to evaluate algorithms beyond theory bridging the gap between abstract concepts and measurable computational performance.

ACKNOWLEDGE

I would like to express my sincere gratitude to Professor for guiding us throughout this assignment and for clearly explaining the underlying concepts of algorithm design. The discussions and examples provided during lectures greatly helped me understand the internal workings of the B-Tree, B+ Tree, and Huffman Coding algorithms at a deeper level. The professor's emphasis on step-by-step analysis and visualization inspired me to carefully trace every insertion and compression step while ensuring the correctness of each algorithm.

I also appreciate the structured instructions and real dataset provided for this homework, which made it possible to connect theoretical ideas with practical implementation. The consistent encouragement to reason about algorithmic efficiency and design trade-offs helped me approach the problem not just as a coding exercise, but as an analytical study of algorithm behaviour. This assignment was an important learning experience, and I am thankful for the professor's support and feedback that made it both challenging and rewarding.

REFERENCES

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd Edition). The MIT Press. Used to understand the theoretical background of B-Tree and B+ Tree insertion, node splitting, and key promotion strategies.
2. Horowitz, E., Sahni, S., & Rajasekaran, S. (2007). *Fundamentals of Computer Algorithms*. Galgotia Publications. Consulted for algorithm design logic, order calculation, and balancing methods in multiway search trees.
3. Weiss, M. A. (2012). *Data Structures and Algorithm Analysis in C++* (4th Edition). Pearson Education. Referred for practical examples of B-Tree and B+ Tree implementation and for visualizing internal versus leaf node structures.
4. GeeksforGeeks. *B Tree and B+ Tree Explained with Examples.* Retrieved from https://www.geeksforgeeks.org. Used as a reference to validate the correctness of stepwise node splits and linkage of leaf nodes.
5. TutorialsPoint. B-Tree and B+Tree *Insertion Algorithms.* Retrieved from https://www.tutorialspoint.com — Used to cross-check insertion logic and visualize leaf-level connectivity across all steps
6. OnlineGDB — Used as the integrated environment for compiling and testing both Python and Go implementations of Huffman Coding and B+ Tree algorithms.

   Python Code: https://onlinegdb.com/4aHsFQXCKU

   Go Code: https://onlinegdb.com/CRyY3OFrN

APPENDIX

1. Python code

```
import csv
import os
import sys
import heapq
from collections import Counter, namedtuple

# path for the CSV file
CSV_PATH = "student-data-age-famsize.csv"

# read CSV files rows
def read_rows(csv_path):
    # check if file exists
    if not os.path.exists(csv_path):
        print(f'ERROR: CSV not found at '{csv_path}'", file=sys.stderr)
        sys.exit(1)

    out = []
    # open CSV file
    with open(csv_path, "r", newline="", encoding="utf-8") as f:
        reader = csv.DictReader(f)
        if not reader.fieldnames:
            print("CSV headers are missing", file=sys.stderr)
            sys.exit(1)

        # make headers case-insensitive
        hmap = {h.lower(): h for h in reader.fieldnames}

        # check for required columns
        if "age" not in hmap or "famsize" not in hmap:
            print("CSV must have 'age' and 'famsize' headers.", file=sys.stderr)
            print(f'Found headers: {reader.fieldnames}", file=sys.stderr)
            sys.exit(1)

        # loop through each row, clean data, and combine into a token
        for row in reader:
            age_raw = (row[hmap["age"]] or "").strip()
            fam_raw = (row[hmap["famsize"]] or "").strip()
            if not age_raw or not fam_raw:
                continue
            try:
                age = int(age_raw)
            except:
                continue
            # combine famsize and age like GT316, LE315, etc.
            out.append((fam_raw, age))
```

```
    # if file is empty or invalid
    if not out:
        print("Not valid rows found.", file=sys.stderr)
        sys.exit(1)

    return out

# create node
class Node(namedtuple("Node", ["freq", "sym", "left", "right"])):
    __slots__ = ()

# build huffman tree
def build_tree(freqs):
    heap = []
    counter = 0

    # push all symbols with frequency into a min-heap
    for s, f in freqs.items():
        heap.append((f, counter, Node(f, s, None, None)))
        counter += 1
    heapq.heapify(heap)

    # if only one symbol, assign code '0'
    if len(heap) == 1:
        f, _, n = heap[0]
        return Node(f, None, n, None)

    # merge lowest two nodes until one root remains
    while len(heap) > 1:
        f1, _, n1 = heapq.heappop(heap)
        f2, _, n2 = heapq.heappop(heap)
        merged = Node(f1 + f2, None, n1, n2)
        heapq.heappush(heap, (merged.freq, counter, merged))
        counter += 1

    # return final root node
    return heap[0][2]

# create code dictionary
def make_codes(root):
    codes = {}

    def dfs(node, path):
        if node.sym is not None:
            # Leaf node → assign code
            codes[node.sym] = path if path else "0"
            return
        if node.left:
            dfs(node.left, path + "0")
        if node.right:
```

```
            dfs(node.right, path + "1")

    dfs(root, "")
    return codes

# encode the symbol list
def encode(symbols, codes):
    return "".join(codes[s] for s in symbols)


# decode encoded bitstring
def decode(bits, root):
    out = []
    node = root
    for b in bits:
        node = node.left if b == "0" else node.right
        if node.sym is not None:
            out.append(node.sym)
            node = root
    return out

def main():
    # read CSV and create tokens like GT316
    rows = read_rows(CSV_PATH)
    symbols = [f"{fam}{age}" for fam, age in rows]

    # count frequency of each token
    freqs = Counter(symbols)

    # make huffman tree and codes
    root = build_tree(freqs)
    codes = make_codes(root)

    # encode and decode
    enc_bits = encode(symbols, codes)
    dec_syms = decode(enc_bits, root)

    # Verify decode data's correctness
    if dec_syms != symbols:
        print("decode is not matching", file=sys.stderr)
        sys.exit(2)

    # write in output files dictionary contain symbol, code, freq, code_len
    with open("codebook.csv", "w", newline="", encoding="utf-8") as f:
        w = csv.writer(f)
        w.writerow(["symbol", "code", "freq", "code_len"])
        for s in sorted(codes, key=lambda x: (len(codes[x]), x)):
            w.writerow([s, codes[s], freqs[s], len(codes[s])])

    # encoded bits
```

```
    with open("encoded_bits.txt", "w", encoding="utf-8") as f:
        f.write(enc_bits)

    # decoded tokens
    with open("decoded_data.txt", "w", encoding="utf-8") as f:
        for s in dec_syms:
            f.write(s + "\n")

    # compute compression stats
    rows_count = len(symbols)
    unique = len(freqs)
    compressed_bits = len(enc_bits)
    # 396 rows * 5 bytes * 8 bits per record
    original_bits = rows_count * 5 * 8

    compression_rate = compressed_bits / original_bits if original_bits else 0.0
    saved = 1 - compression_rate
    compression_ratio = (original_bits / compressed_bits) if compressed_bits else float('inf')

    # Print Results in Terminal
    print("\nHuffman Compression (Combined famsize+age)")
    print("----------------------------------------")
    print(f"Rows: {rows_count}")
    print(f"Unique tokens: {unique}")
    print(f"Original bits: {original_bits}")
    print(f"Compressed bits: {compressed_bits}")
    print(f"Compression ratio (Uncompressed Size / Compressed Size):
{compression_ratio:.2f}")
    print(f"Compression rate (Compressed bits / Original bits): {compression_rate:.4f}")
    print(f"Space savings (1 - Compression rate) : {saved*100:.2f}%\n ")


    print("Dictionary Table:")
    print("Token      |Frequency|   Code   | Bits")
    print("----------------------------------------")
    for s in sorted(codes, key=lambda x: (len(codes[x]), x)):
        print(f"{s:10} |{freqs[s]:5}    | {codes[s]:8} | {len(codes[s]):3}")

    # Write in Stats File
    with open("table_and_result.txt", "w", encoding="utf-8") as f:
        f.write("Huffman Compression (Combined famsize+age)\n")
        f.write("----------------------------------------\n")
        f.write(f"Rows: {rows_count}\n")
        f.write(f"Unique tokens: {unique}\n")
        f.write(f"Original bits: {original_bits}\n")
        f.write(f"Compressed bits: {compressed_bits}\n")
        f.write(f"Compression ratio (Uncompressed Size / Compressed Size):
{compression_ratio:.2f}\n")
        f.write(f"Compression rate (Compressed bits / Original bits): {compression_rate:.4f}\n")
        f.write(f"Space savings (1 - Compression rate) : {saved*100:.2f}%\n")
```

```
        f.write("\nDictionary Table:\n")
        f.write("Token      |Frequency |   Code   | Bits\n")
        f.write("-----------------------------------------\n")
        for s in sorted(codes, key=lambda x: (len(codes[x]), x)):
            f.write(f"{s:10} | {freqs[s]:5}    | {codes[s]:8} | {len(codes[s]):3}\n")

    print("\nall files generated successfully.")
    print("codebook.csv, encoded_bits.txt, decoded_data.txt, table_and_result.txt\n")

# run program
if __name__ == "__main__":
    main()
```

2. Go code

```
package main

import (
        "bufio"
        "container/heap"
        "encoding/csv"
        "fmt"
        "log"
        "math"
        "os"
        "path/filepath"
        "sort"
        "strconv"
        "strings"
)

// path for the CSV file
const CSV_PATH = "student-data-age-famsize.csv"

// read CSV file rows
func readRows(csvPath string) [][2]string {
        // check if file exists
        if _, err := os.Stat(csvPath); err != nil {
                fmt.Fprintf(os.Stderr, "CSV file not found at '%s'\n", csvPath)
                os.Exit(1)
        }

        // open CSV file
        f, err := os.Open(csvPath)
        if err != nil {
                fmt.Fprintln(os.Stderr, "cannot open CSV:", err)
                os.Exit(1)
        }
```

```
	defer f.Close()

	r := csv.NewReader(f)
	r.FieldsPerRecord = -1

	rows, err := r.ReadAll()
	if err != nil {
		fmt.Fprintln(os.Stderr, "cannot read CSV:", err)
		os.Exit(1)
	}
	if len(rows) == 0 {
		fmt.Fprintln(os.Stderr, "CSV headers are missing")
		os.Exit(1)
	}

	// make headers case-insensitive
	header := rows[0]
	hmap := map[string]int{}
	for i, h := range header {
		hmap[strings.ToLower(strings.TrimSpace(h))] = i
	}

	// check for required columns
	ageIdx, okA := hmap["age"]
	famIdx, okF := hmap["famsize"]
	if !okA || !okF {
		fmt.Fprintln(os.Stderr, "CSV must have 'age' and 'famsize' headers.")
		fmt.Fprintln(os.Stderr, "found headers:", header)
		os.Exit(1)
	}

	out := make([][2]string, 0, len(rows)-1)

	// loop through each row, clean data, and combine into a token source pair
	for _, row := range rows[1:] {
		if ageIdx >= len(row) || famIdx >= len(row) {
			continue
		}
		ageRaw := strings.TrimSpace(row[ageIdx])
		famRaw := strings.TrimSpace(row[famIdx])
		if ageRaw == "" || famRaw == "" {
			continue
		}
		// validate age is integer
		if _, err := strconv.Atoi(ageRaw); err != nil {
			continue
		}
		// keep separate; combine later as fam+age like GT316, LE315, etc.
		out = append(out, [2]string{famRaw, ageRaw})
	}
```

```
	// if file is empty or invalid
	if len(out) == 0 {
		fmt.Fprintln(os.Stderr, "Not valid rows found.")
		os.Exit(1)
	}
	return out
}

// Huffman node
type Node struct {
	freq  int
	sym   string
	left  *Node
	right *Node
}

type item struct {
	freq int
	ord  int
	node *Node
}
type pqueue []item

func (pq pqueue) Len() int { return len(pq) }
func (pq pqueue) Less(i, j int) bool {
	// min-heap by frequency; tie-break by insertion order
	if pq[i].freq == pq[j].freq {
		return pq[i].ord < pq[j].ord
	}
	return pq[i].freq < pq[j].freq
}
func (pq pqueue) Swap(i, j int)       { pq[i], pq[j] = pq[j], pq[i] }
func (pq *pqueue) Push(x interface{}) { *pq = append(*pq, x.(item)) }
func (pq *pqueue) Pop() interface{} {
	old := *pq
	n := len(old)
	x := old[n-1]
	*pq = old[:n-1]
	return x
}

// build huffman tree
func buildTree(freqs map[string]int) *Node {
	h := &pqueue{}
	heap.Init(h)
	ord := 0

	// push all symbols with frequency into a min-heap
	for s, f := range freqs {
```

```
                heap.Push(h, item{freq: f, ord: ord, node: &Node{freq: f, sym: s}})
                ord++
        }

        // if only one symbol, assign code '0' represented by root with single child
        if h.Len() == 1 {
                it := heap.Pop(h).(item)
                return &Node{freq: it.freq, sym: "", left: it.node, right: nil}
        }

        // merge lowest two nodes until one root remains
        for h.Len() > 1 {
                a := heap.Pop(h).(item)
                b := heap.Pop(h).(item)
                merged := &Node{freq: a.freq + b.freq, sym: "", left: a.node, right: b.node}
                heap.Push(h, item{freq: merged.freq, ord: ord, node: merged})
                ord++
        }
        // return final root node
        return heap.Pop(h).(item).node
}

// create code dictionary
func makeCodes(root *Node) map[string]string {
        codes := make(map[string]string)

        var dfs func(n *Node, path string)
        dfs = func(n *Node, path string) {
                if n == nil {
                        return
                }
                if n.sym != "" {
                        // Leaf node → assign code
                        if path == "" {
                                codes[n.sym] = "0"
                        } else {
                                codes[n.sym] = path
                        }
                        return
                }
                dfs(n.left, path+"0")
                dfs(n.right, path+"1")
        }
        dfs(root, "")
        return codes
}

// encode the symbol list
func encode(symbols []string, codes map[string]string) string {
        var b strings.Builder
```

```
	// optional pre-allocation
	total := 0
	for _, s := range symbols {
		total += len(codes[s])
	}
	b.Grow(total)

	for _, s := range symbols {
		b.WriteString(codes[s])
	}
	return b.String()
}

// decode encoded bitstring
func decode(bits string, root *Node) []string {
	out := make([]string, 0, 1024)
	node := root
	for i := 0; i < len(bits); i++ {
		if bits[i] == '0' {
			node = node.left
		} else {
			node = node.right
		}
		if node.sym != "" {
			out = append(out, node.sym)
			node = root
		}
	}
	return out
}

func main() {
	// read CSV and create tokens like GT316
	rows := readRows(CSV_PATH)
	symbols := make([]string, 0, len(rows))
	for _, r := range rows {
		fam := r[0]
		age := r[1]
		symbols = append(symbols, fam+age)
	}

	// count frequency of each token
	freqs := make(map[string]int, len(symbols))
	for _, s := range symbols {
		freqs[s]++
	}

	// make huffman tree and codes
	root := buildTree(freqs)
	codes := makeCodes(root)
```

```
// encode and decode
encBits := encode(symbols, codes)
decSyms := decode(encBits, root)

// verify decode data's correctness
if len(decSyms) != len(symbols) {
	fmt.Fprintln(os.Stderr, "decode is not matching (length)")
	os.Exit(2)
}
for i := range decSyms {
	if decSyms[i] != symbols[i] {
		fmt.Fprintln(os.Stderr, "decode is not matching (content)")
		os.Exit(2)
	}
}

// write codebook.csv: symbol, code, freq, code_len
cb, err := os.Create("codebook.csv")
if err != nil {
	log.Fatal(err)
}
cw := csv.NewWriter(cb)
_ = cw.Write([]string{"symbol", "code", "freq", "code_len"})

// sort by code length, then symbol
type rowOut struct {
	s string
	c string
	f int
	l int
}
var rowsOut []rowOut
for s, c := range codes {
	rowsOut = append(rowsOut, rowOut{s: s, c: c, f: freqs[s], l: len(c)})
}
sort.Slice(rowsOut, func(i, j int) bool {
	if rowsOut[i].l == rowsOut[j].l {
		return rowsOut[i].s < rowsOut[j].s
	}
	return rowsOut[i].l < rowsOut[j].l
})
for _, r := range rowsOut {
	_ = cw.Write([]string{r.s, r.c, strconv.Itoa(r.f), strconv.Itoa(r.l)})
}
cw.Flush()
_ = cb.Close()

// encoded bits
if err := os.WriteFile("encoded_bits.txt", []byte(encBits), 0644); err != nil {
```

```
		log.Fatal(err)
	}

	// decoded tokens
	dd, err := os.Create("decoded_data.txt")
	if err != nil {
		log.Fatal(err)
	}
	bw := bufio.NewWriter(dd)
	for _, s := range decSyms {
		_, _ = bw.WriteString(s)
		_ = bw.WriteByte('\n')
	}
	_ = bw.Flush()
	_ = dd.Close()

	// compute compression stats (assume 5 bytes per original record → 40 bits)
	rowsCount := len(symbols)
	unique := len(freqs)
	compressedBits := len(encBits)
	originalBits := rowsCount * 5 * 8

	var compressionRate float64
	if originalBits > 0 {
		compressionRate = float64(compressedBits) / float64(originalBits)
	}
	saved := 1.0 - compressionRate

	// Build a stable order for printing: by code length, then lexicographically
	sortedKeys := make([]string, 0, len(codes))
	for s := range codes {
		sortedKeys = append(sortedKeys, s)
	}
	sort.Slice(sortedKeys, func(i, j int) bool {
		li, lj := len(codes[sortedKeys[i]]), len(codes[sortedKeys[j]])
		if li == lj {
			return sortedKeys[i] < sortedKeys[j]
		}
		return li < lj
	})

	var compressionRatio float64
	if compressedBits > 0 {
		compressionRatio = float64(originalBits) / float64(compressedBits)
	} else {
		compressionRatio = math.Inf(1)
	}

	// Print Results in Terminal
	fmt.Println()
```

```
	fmt.Println("Huffman Compression (Combined famsize+age)")
	fmt.Println("----------------------------------------")
	fmt.Printf("Rows: %d\n", rowsCount)
	fmt.Printf("Unique tokens: %d\n", unique)
	fmt.Printf("Original bits: %d\n", originalBits)
	fmt.Printf("Compressed bits: %d\n", compressedBits)
	fmt.Printf("Compression ratio (Uncompressed Size / Compressed Size): %.2f\n",
compressionRatio)
	fmt.Printf("Compression rate (Compressed bits / Original bits): %.4f\n",
compressionRate)
	fmt.Printf("Space savings (1 - Compression rate): %.2f%%\n\n", saved*100)

	fmt.Println("Dictionary Table:")
	fmt.Println("Token      |Frequency|   Code   | Bits")
	fmt.Println("----------------------------------------")
	for _, s := range sortedKeys {
		fmt.Printf("%-10s |%5d    | %-8s | %3d\n", s, freqs[s], codes[s], len(codes[s]))
	}
	// Write same stats/table to table_and_result.txt
	sb := &strings.Builder{}
	sb.WriteString("Huffman Compression (Combined famsize+age)\n")
	sb.WriteString("----------------------------------------\n")
	sb.WriteString(fmt.Sprintf("Rows: %d\n", rowsCount))
	sb.WriteString(fmt.Sprintf("Unique tokens: %d\n", unique))
	sb.WriteString(fmt.Sprintf("Original bits: %d\n", originalBits))
	sb.WriteString(fmt.Sprintf("Compressed bits: %d\n", compressedBits))
	sb.WriteString(fmt.Sprintf("Compression ratio (Uncompressed Size / Compressed
Size): %.2f\n", compressionRatio))
	sb.WriteString(fmt.Sprintf("Compression rate (Compressed bits / Original bits):
%.4f\n", compressionRate))
	sb.WriteString(fmt.Sprintf("Space savings (1 - Compression rate): %.2f%%\n\n",
saved*100))
	sb.WriteString("Dictionary Table:\n")
	sb.WriteString("Token      |Frequency |   Code   | Bits\n")
	sb.WriteString("----------------------------------------\n")
	for _, s := range sortedKeys {
		sb.WriteString(fmt.Sprintf("%-10s | %5d    | %-8s | %3d\n", s, freqs[s],
codes[s], len(codes[s])))
	}
	if err := os.WriteFile("table_and_result.txt", []byte(sb.String()), 0644); err != nil {
		log.Fatal(err)
	}
	fmt.Println("\nall files generated successfully.")
	fmt.Printf("%s\n", strings.Join([]string{
		filepath.Join(".", "codebook.csv"),
		filepath.Join(".", "encoded_bits.txt"),
		filepath.Join(".", "decoded_data.txt"),
		filepath.Join(".", "table_and_result.txt"),
	}, ", "))
}
```