Project Report: Course Project-2

CS51510-001, Fall 2025
Purdue University Northwest
11/04/2025

Class ID (Ascend sequency)	Contributor Name (Last Name, First Name)	Email	Detailed Contributions
0033917373	**Matthews, Joshua**	Matthe68@pnw.edu	**Research Report**
038414971	**Mishra, Amartya**	Mishr259@pnw.edu	**Documentation & Report**
039133149	**Mammai, Sreeja**	Smammai@pnw.edu	**Performance Tests**
038410044	**Mhasawade, Siddhi**	Smhasawa@pnw.edu	**Performance Tests**
0038159254	**Mekala, Ruthvik**	Rmekala@pnw.edu	**Java Code & UML**
039133149	**Patel, Raaj**	Pate2682@pnw.edu	**Python Code**
0037874079	**Modi, Hirak**	Modi54@pnw.edu	**Presentation**
0038119426	**Nalluri, Prasanth**	Pnallur@pnw.edu	**Java Code & Flowchart**

Table of Contents

Abstract 5

I. Introduction 7

II. Methodology 10

1. Implementation of the Environment and Rationale 12
2. Core Implementation Strategy 13
3. Linked List Data Storage Explanation 15
4. Huffman Index Design and Process 16
5. Red-Black Tree Index Design and Process 18
6. UML Diagram Representation 20
7. Performance Benchmarking and Analysis 22

III. Implementation of Huffman and Red-Black Tree 23

1. Huffman Dictionary and Compression Statistics 25
2. Step by Step visualization of Red Black Tree. 27

IV. Results and Analysis 45

1. Execution Verification Across Python and Java 47
2. Output Results in Ubuntu 51
3. Analyze Linux performance 56

V. Reproducibility and code accessibility 62

VI. Conclusion 64

Acknowledge 66

References 67

Appendix 68

1. Java Program 68
2. Python Program 96

Table of Figures

Figure 1 — Kaggle website that contains the example comma separated value file (CSV) 11
Figure 2 — UML Diagram 20
Figure 3 — Flowchart 21
Figure 4 — Dictionary Table for Huffman Code 25
Figure 5 — Red-Black Tree of Guardian and Age pt. 1 27
Figure 6 — Red-Black Tree of Guardian and Age pt. 2 27
Figure 7 — Red-Black Tree of Guardian and Age pt. 3 27
Figure 8 — Red-Black Tree of Guardian and Age pt. 4 27
Figure 9 — Red-Black Tree of Guardian and Age pt. 5 28
Figure 10 — Red-Black Tree of Guardian and Age pt. 6 28
Figure 11 — Red-Black Tree of Guardian and Age pt. 7 28
Figure 12 — Red-Black Tree of Guardian and Age pt. 8 28
Figure 13 — Red-Black Tree of Guardian and Age pt. 9 29
Figure 14 — Red-Black Tree of Guardian and Age pt. 10 29
Figure 15 — Red-Black Tree of Guardian and Age pt. 11 29
Figure 16 — Red-Black Tree of Guardian and Age pt. 12 29
Figure 17 — Red-Black Tree of Guardian and Age pt. 13 30
Figure 18 — Red-Black Tree of Guardian and Age pt. 14 30
Figure 19 — Red-Black Tree of Guardian and Age pt. 15 30
Figure 20 — Red-Black Tree of Guardian and Age pt. 16 31
Figure 21 — Red-Black Tree of Guardian and Age pt. 17 31
Figure 22 — Red-Black Tree of Guardian and Age pt. 18 31
Figure 23 — Red-Black Tree of Guardian and Age pt. 19 32
Figure 24 — Red-Black Tree of Guardian and Age pt. 20 32
Figure 25 — Red-Black Tree of Guardian and Age pt. 21 32
Figure 26 — Red-Black Tree of Guardian and Age pt. 22 33
Figure 27 — Red-Black Tree of Guardian and Age pt. 23 33
Figure 28 — Red-Black Tree of Guardian and Age pt. 24 33
Figure 29 — Red-Black Tree of Guardian and Age pt. 25 34
Figure 30 — Red-Black Tree of Guardian and Age pt. 26 34
Figure 31 — Red-Black Tree of Guardian and Age pt. 27 34

Figure 32 — Red-Black Tree of Guardian and Age pt. 28 35
Figure 33 — Red-Black Tree of Guardian and Age pt. 29 35
Figure 34 — Red-Black Tree of Guardian and Age pt. 30 35
Figure 35 — Red-Black Tree of Guardian and Age pt. 31 36
Figure 36 — Red-Black Tree of Guardian and Age pt. 32 36
Figure 37 — Red-Black Tree of Guardian and Age pt. 33 36
Figure 38 — Red-Black Tree of Guardian and Age pt. 34 37
Figure 39 — Red-Black Tree of Guardian and Age pt. 35 37
Figure 40 — Red-Black Tree of Guardian and Age pt. 36 37
Figure 41 — Red-Black Tree of Guardian and Age pt. 37 38
Figure 42 — Red-Black Tree of Guardian and Age pt. 38 38
Figure 43 — Red-Black Tree of Guardian and Age pt. 39 38
Figure 44 — Red-Black Tree of Guardian and Age pt. 40 39
Figure 45 — Red-Black Tree of Guardian and Age pt. 41 39
Figure 46 — Red-Black Tree of Guardian and Age pt. 42 39
Figure 47 — Red-Black Tree of Guardian and Age pt. 43 40
Figure 48 — Red-Black Tree of Guardian and Age pt. 44 40
Figure 49 — Red-Black Tree of Guardian and Age pt. 45 40
Figure 50 — Red-Black Tree of Guardian and Age pt. 46 41
Figure 51 — Red-Black Tree of Guardian and Age pt. 47 41
Figure 52 — Red-Black Tree of Guardian and Age pt. 48 41
Figure 53 — Red-Black Tree of Guardian and Age pt. 49 42
Figure 54 — Red-Black Tree of Guardian and Age pt. 50 42
Figure 55 — Red-Black Tree of Guardian and Age pt. 51 42
Figure 56 — Red-Black Tree of Guardian and Age pt. 52 43
Figure 57 — Red-Black Tree of Guardian and Age pt. 53 43
Figure 58 — Red-Black Tree of Guardian and Age pt. 54 43
Figure 59 — Red-Black Tree of Guardian and Age pt. 55 44
Figure 60 — Red-Black Tree of Guardian and Age pt. 56 44
Figure 61 — Red-Black Tree of Guardian and Age pt. 57 44
Figure 62 — MemoryDBHuffman.java and student-data.csv files in Ubuntu 48
Figure 63 — Java Execution 48
Figure 64 — MemoryDBHuffman.py and student-data.csv files in Ubuntu 49

Figure 65 — Python Execution pt. 1 49
Figure 66 — Python Execution pt. 2 50
Figure 67 — Loading CSV data into the LinkedList 51
Figure 68 — Displaying the data 52
Figure 69 — Generating the Huffman Codes 52
Figure 70 — Displaying Huffman Dictionary and Compression Ratio 53
Figure 71 — Saving encoded data and decoded data in .txt file 53
Figure 72 — Saving encoded data and decoded data in .txt file 54
Figure 73 — Decoded_text.txt file that shows decoded data 54
Figure 74 — Build a Red-Black tree 55
Figure 75 — Display RBT logs and trees on web 55
Figure 76 — Time usage command 56
Figure 77 — Time command_output 57
Figure 78 — top command_output 57
Figure 79 — pidstat_output 58
Figure 80 — Time command Output for Java 58
Figure 81 — Time command Output2 for Java 59
Figure 82 — Output for ps command Java 60
Figure 83 — Output for pidstat command Java 61
Figure 84 — Output for top command Java 61

Abstract

Through Course Project-2, our team applied advanced data structure concepts and indexing strategies to enhance in memory database system. Building upon the foundation of main-memory storage using a singly linked list, we expanded the system to support efficient data retrieval through two indexing approaches widely used in database systems: Huffman-coding based indexing and Red-Black Tree (RBT) indexing. This report outlines the architectural decisions, index construction methods, and performance validation procedures that shaped our implementation.

We began by loading the provided "*student-data.csv"* dataset entirely into memory, storing each record inside a custom linked-list structure. From there, we developed a Huffman-code index for categorical attributes, allowing the system to compress values based on frequency and accelerate equality lookups using prefix-free binary representations. In parallel, we implemented a Red-Black Tree index for numerical attributes, enabling ordered queries and range-based selection with predictable logarithmic performance guarantees. These components allowed us to model how real-world relational databases improve query execution through indexing techniques rather than relying solely on sequential scanning.

To support realistic interaction, the system was extended with a simplified interface that allows users to specify columns, filter conditions, and the desired indexing strategy. For instance, users may issue queries that selectively leverage Huffman indexing for categorical lookups or RBT indexing for numeric filtering and range searches. This design preserves the educational transparency of a system.

A comprehensive evaluation was performed in an Ubuntu Linux environment to measure index construction time, query execution behavior, CPU utilization, and memory efficiency. Using native Linux tools including *time*, *top*, *pidstat*, and *iostat* .We captured real performance metrics

that highlighted the benefits of compressed search with Huffman coding and the balancing properties of Red-Black Trees. Visual step-by-step diagrams were generated to demonstrate Huffman tree formation, prefix-code assignment, and RBT insertions with corresponding rotations and color adjustments.

To ensure reproducibility and transparency, we have provided complete source code and executable links via OnlineGDB, allowing independent reviewers to compile, run, and validate our results directly in a web-based environment. This approach guarantees consistent execution without requiring any local configuration or software installation.

Ultimately, this project strengthened our understanding of memory database design, compression-based indexing, self-balancing search trees, and performance profiling in Linux-based systems. The knowledge and teamwork developed through this project lay a strong foundation for future work in database internals, query optimization, and scalable data-system engineering.

I. Introduction

As data continues to grow in scale, complexity, and relevance across computing disciplines, the ability to efficiently store, access, and process information becomes foundational to modern software and data systems. While introductory coursework teaches the importance of data structures and algorithmic design, deeper understanding emerges when these concepts are applied to real execution environments and real datasets. This project advances that learning objective by challenging students to build a memory database system and extend it with mechanisms commonly employed in full-scale relational engines most notably compressed indexing and balanced search trees.

The core of this project involves constructing a memory data store using a singly linked list to hold records from the "*student-data.csv"* dataset sourced from Kaggle. Linked lists provide a useful lens into low-level storage management, pointer manipulation, and sequential traversal. By beginning with a linked-list foundation, the system models the base behavior of unindexed storage where queries must scan each node, thereby illustrate the performance limitations of linear search and motivate the need for indexing.

To address these limitations, this project implements two indexing strategies representative of modern data systems, each serving a distinct functional purpose. First, a Huffman-coding index is created for categorical attributes. Huffman coding leverages frequency-based compression to assign variable-length prefix-free binary codes to tokens, simultaneously reducing memory usage and improving lookup speed for frequently occurring values. This approach highlights how compression and indexing can coexist, enabling faster query resolution through compact data representation and direct symbol lookup. Second, a Red-Black Tree (RBT) index is developed for numerical attributes. As a self-balancing binary search tree, an RBT ensures that insertions and

lookups maintain logarithmic time complexity, enabling ordered queries and efficient range-based retrieval capabilities fundamental to relational query processing and query optimization engines.

Beyond implementation, the project emphasizes evidence-based evaluation and system-level validation. The system is executed in an Ubuntu Linux environment, and performance metrics are collected using native monitoring and profiling tools including time, top, pidstat, and iostat. These measurements allow for a quantitative comparison between indexed and unindexed retrieval, as well as examination of CPU behavior, memory usage, and runtime efficiency during index construction and query execution. This empirical approach reinforces the importance of benchmarking and profiling in data systems development.

To ensure transparency and reproducibility, all code is hosted in publicly accessible online execution environments, allowing reviewers to compile, run, and verify results without installing additional software. This approach not only supports academic integrity but also mirrors industry practices surrounding reproducible software and open execution environments.

Finally, this project is structured to foster collaborative learning and shared responsibility. Teamwork played a significant role in designing data structures, debugging implementation challenges, documenting results, and preparing visualizations for Huffman tree construction and Red-Black Tree balancing operations. Through cooperative effort, the team strengthened both technical competency and professional skills such as task coordination, communication, and version-controlled development.

In summary, this project blends data structures, compression techniques, balanced tree algorithms, and performance benchmarking into a unified learning experience. It provides a comprehensive understanding of how indexing enhances database efficiency and demonstrates the

practical value of combining theoretical models with real-world execution, system analysis, and collaborative problem solving.

II. Methodology

The initial stage of this project involved obtaining the "*student-data.csv"* dataset from Kaggle, as specified by the course requirements. This dataset was intentionally selected due to its structured tabular format and clean comma-separated structure, making it both human-interpretable and straightforward for automated parsing. After securely downloading the file, the data was prepared for ingestion into our custom in-memory database.

To enable seamless processing, each line of the CSV file was parsed into individual attributes representing student identifiers, demographic characteristics, and academic performance metrics. These values were then mapped into a consistent internal structure that served as the foundation for creating linked-list nodes in memory. This step ensured that every row of the dataset was converted into a standardized record format, suitable for efficient storage and subsequent indexing operations. Care was taken during preprocessing to handle minor edge cases such as whitespace normalization, verification of column counts, removal of malformed entries, and conversion of numerical fields into their appropriate data types. Establishing this reliable data preprocessing pipeline was crucial, as it guaranteed that the memory database operated on clean and validated data. This preparation not only supported efficient node creation and traversal but also ensured the correctness of indexing algorithms applied later, including Huffman code generation and Red-Black Tree insertion.

By performing this structured data transformation at the outset, we created a robust and uniform dataset representation that allowed the remaining project components index construction, compression analysis, and performance benchmarking to operate smoothly and accurately.

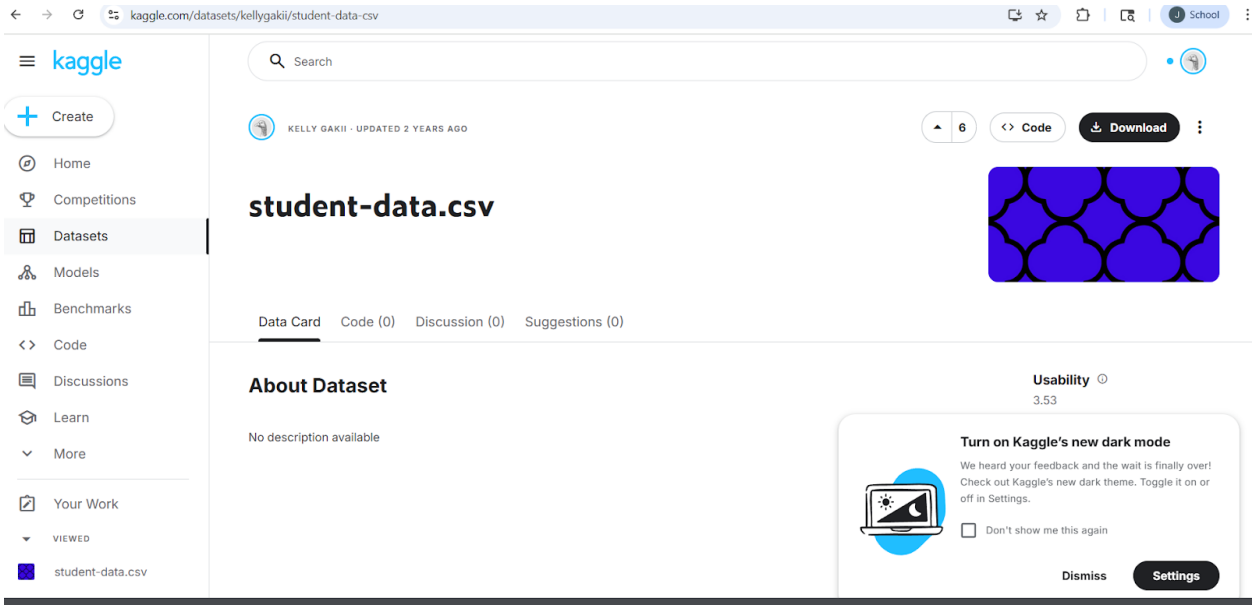

Figure 1 — Kaggle website that contains the example comma separated value file (CSV)

1. Implementation of the Environment and Rationale

A dual-environment workflow was used to support both rapid development and accurate performance evaluation. Initial coding, testing, and debugging were performed in the OnlineGDB IDE, which provided a collaborative, browser-based environment for quick iteration and easy sharing across team members. This platform ensured portability and guaranteed that our implementations could be executed and reviewed without requiring local configuration in ubuntu. Public OnlineGDB links are included in this report to enable full reproducibility and independent verification of our system's behavior.

Once the core functionality was validated, the final versions of our Huffman-indexed and Red-Black-Tree-indexed database modules were executed within an Ubuntu Linux environment running on a virtual machine. This environment allowed us to collect precise system-level performance metrics while running indexing operations and executing queries over the in-memory dataset. Native Linux profiling tools, including *time*, *top*, *iostat*, and *pidstat*, were used to observe CPU usage, memory consumption, I/O activity, and execution time during index construction and query execution. This dual-platform strategy ensured that our system was not only logically correct but also evaluated under realistic execution conditions, resulting in reliable and evidence-driven performance analysis.

2. Core Implementation Strategy

The foundation of this project was creating a simple in-memory database capable of storing and processing student records without relying on external database software. To achieve this, we used a singly linked list as the primary storage structure. Each row from the student-data.csv file was parsed, cleaned, and stored as a record inside a custom node object. Using a linked list allowed us to manually manage memory and traversal, reinforcing the low-level nature of data handling often hidden inside modern database systems. This structure ensured that we could load, iterate through, and retrieve records directly from main memory while controlling how data flowed through our system.

Once the base data structure was established, the next major step was to integrate Huffman coding as an indexing mechanism for selected categorical and mixed-attribute fields. In the Python implementation, we calculated the frequency of values, built a Huffman tree, and generated prefix-free binary codes for each token. This enabled us to compress record values and reduce the number of bits required to store them. Along with compression, the Huffman index allowed faster equality look-ups for frequently appearing attributes, demonstrating how encoding and indexing can work together to optimize search performance. We also produced step-by-step logs and visualizations of the tree-building process to clearly show how each merge and code assignment occurred.

In addition to compression-based indexing, we implemented a Red-Black Tree (RBT) to support efficient ordered operations. The RBT was built for numeric attributes like age and guardian, ensuring balanced tree height and consistent lookup time. We developed the RBT first in Python to allow easier debugging and visual inspection, then recreated the structure in Java to validate correctness and demonstrate the same logic in a statically typed, compiled environment. During testing, we tracked rotations, recoloring steps, and tree state after each insertion, producing

a clear record of how the RBT maintained balance and supported range-based queries. This dual-language approach helped us evaluate implementation differences between interpreted and compiled execution models.

To complete the system, we added a simple interface to issue queries and interact with the underlying data structures. Users could specify which index to use, request data by value or range, and observe how query performance differed when using Huffman codes, Red-Black Trees, or a plain linked list scan. The final stage involved evaluating performance in an Ubuntu Linux environment, where we monitored CPU usage, execution time, and memory behavior using system utilities such as time, top, pidstat, and iostat. These measurements allowed us to understand how indexing impacted efficiency and confirmed the advantages of compressed and balanced-tree-based access. Together, these components formed a complete in-memory database system that demonstrated both conceptual understanding and practical implementation of real-world database indexing techniques.

3. Linked List Data Storage Explanation

The foundation of this project is a custom in-memory storage system built using a singly linked list. This structure was chosen to manually manage and organize student records loaded from the "*student-data.csv"* dataset. Each line from the file is parsed into individual attributes, and those values are placed inside a node object that contains two key components: the student data and a pointer to the next node in the list. By constructing this structure manually, we created a lightweight system that allows data to be stored directly in RAM without relying on external databases or built-in indexing support.

Using a linked list ensured that records could be loaded and appended efficiently without needing continuous memory blocks. As each student entry is read from the CSV file, a new node is dynamically created and linked to the previous one, forming a chain representing the entire dataset. This approach reflects how low-level data engines may manage records sequentially, and it allows us to observe the inherent behavior of systems without automatic memory handling.

Working with a linked list also helped demonstrate the performance limitations of sequential structures. Since nodes are connected line-by-line, accessing a specific record requires visiting each node until the target is reached. This naturally results in linear-time search, especially visible when dealing with hundreds of records. Identifying this limitation was important, as it provided a practical motivation for adding efficient indexing mechanisms Huffman coding for compressed lookup on categorical fields and a Red-Black Tree for balanced search on numeric fields.

4. Huffman Index Design and Process

To improve memory efficiency and accelerate lookups for frequently occurring data patterns, we implemented a Huffman-based indexing system as one of the core components of our in-memory database. Huffman coding is a compression technique that assigns shorter binary codes to values that occur more often and longer codes to values that occur less frequently. By applying this method to selected attributes from the *student-data.csv* dataset, we were able to reduce bit usage while also ensuring faster direct lookups for high-frequency data entries. This approach aligned strongly with the project objective of exploring indexing techniques rooted in algorithmic concepts.

The first step in constructing the Huffman index involved scanning the chosen column values such as "*guardian"* and "*age"* and computing the frequency of each unique token. Once these frequencies were calculated, each value was inserted into a min-heap as a node containing the token and its count. From here, we repeatedly extracted the two nodes with the lowest frequency, merged them into a new parent node, and reinserted the parent back into the heap. This merging process continued until only a single tree root remained, forming the complete Huffman tree. Each left branch was assigned a 0, and each right branch was assigned a 1, gradually forming binary codes for each value based on their position in the tree.

After the tree was built, we performed a depth-first traversal to record and assign the final prefix-free binary code for each student attribute. These codes were stored in a dictionary that served as our Huffman index, mapping original values to their compressed binary representations. The result was a compact index structure where frequent values which appeared more often in the student dataset received very short binary codes, while rare values naturally received longer ones.

We then used these assigned codes to encode the dataset and compare the number of bits used before and after compression, resulting in a measurable compression ratio and bit-savings report.

In addition to creating the index, we ensured transparency by producing visualization outputs that showed the frequency table, merge sequence, Huffman tree structure, and the final codebook. These recorded steps provided visual evidence of how the compression logic unfolded at each stage and allowed us to verify correctness and trace the code generation for individual values. Together, the Huffman index and its supporting logs demonstrated not only an efficient compression and lookup structure but also the value of combining theoretical algorithmic knowledge with practical implementation in a real dataset context.

5. Red-Black Tree Index Design and Process

Implemented a Red-Black Tree (RBT) as a balanced indexing structure in this project. RBTs maintain logarithmic time complexity for search, insertion, and traversal operations, making them ideal for efficiently handling ordered numeric data. In our system, the Red-Black Tree was used to index numerical attributes such as age and guardian, which allowed us to perform fast lookups and range-based queries over these fields. This indexing approach demonstrates how balanced trees serve as the backbone of many real-world databases and operating system internals where consistent performance and structural stability are required.

The RBT construction process began by inserting each record's key (e.g., age or guardian) into the tree one at a time. Each insertion followed standard binary search tree logic to determine the correct placement of the node. However, unlike a regular BST, the RBT must ensure that the tree remains balanced after every insertion. To achieve this, we applied Red-Black Tree properties, which include constraints on node colors and structural positioning. After inserting a new node, the system performed a series of checks to verify that tree rules were not violated. Whenever an imbalance occurred, we executed the necessary fixes including recoloring nodes and performing rotations (left, right, or a combination) to restore balance.

Throughout the indexing process, we maintained logs that captured each step of the tree's evolution. These logs recorded node insertions, recolor operations, and rotation actions, providing insight into how the tree dynamically adjusts itself to maintain balance. This detailed trace helped verify correctness and offered a visual understanding of how rotations preserve ordering while preventing the tree from becoming skewed. By the time all target values were inserted, the resulting structure was a balanced tree capable of supporting fast lookups and ordered traversal for numeric fields.

By implementing the RBT in both Python and Java, we ensured that our indexing logic functioned consistently across different programming environments. The Python version allowed us to visualize the balancing process and test logic rapidly, while the Java version demonstrated how the same tree operations behave within a strongly typed and compiled environment. Together, these implementations highlight the practical value of balanced search trees in indexing tasks and solidify our understanding of tree-based algorithms as they apply to memory database systems.

6. UML Diagram Representation and Flowchart

To complement the written methodology and implementation details, a Unified Modeling Language (UML) diagram was developed to provide a high-level, visual representation of the system. UML diagrams are widely used in software engineering as a standardized way to depict system architecture, object interactions, and the relationship between different components. In the context of this project the UML diagram serves several purposes, including structural clarity, process flow, role separation, and documentation value. Our diagram is broad in scope designed to capture the core entities of our project along with the general flow of the data from acquisition to output. This emphasizes how the components fit together as an integrated system.

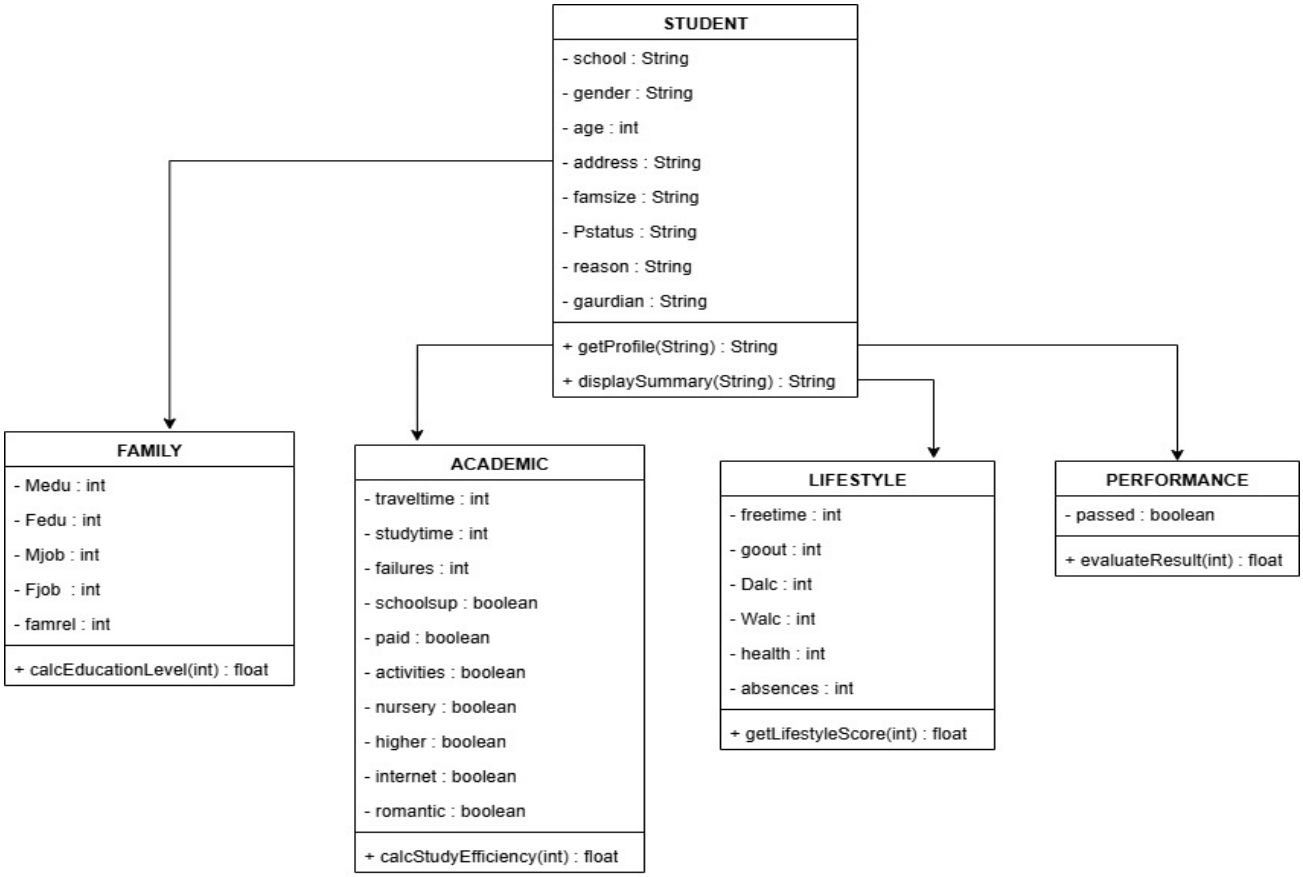

Figure 2 — UML Diagram

The flowchart below illustrates how the system operates, demonstrating a linear path through each of the core modules including CSV loading, Huffman coding, and Red-Black Tree building. It integrates the logical flow of control from user input through data ingestion, processing, encoding, decoding and visualization so the project's process and execution points along the way are made clear.

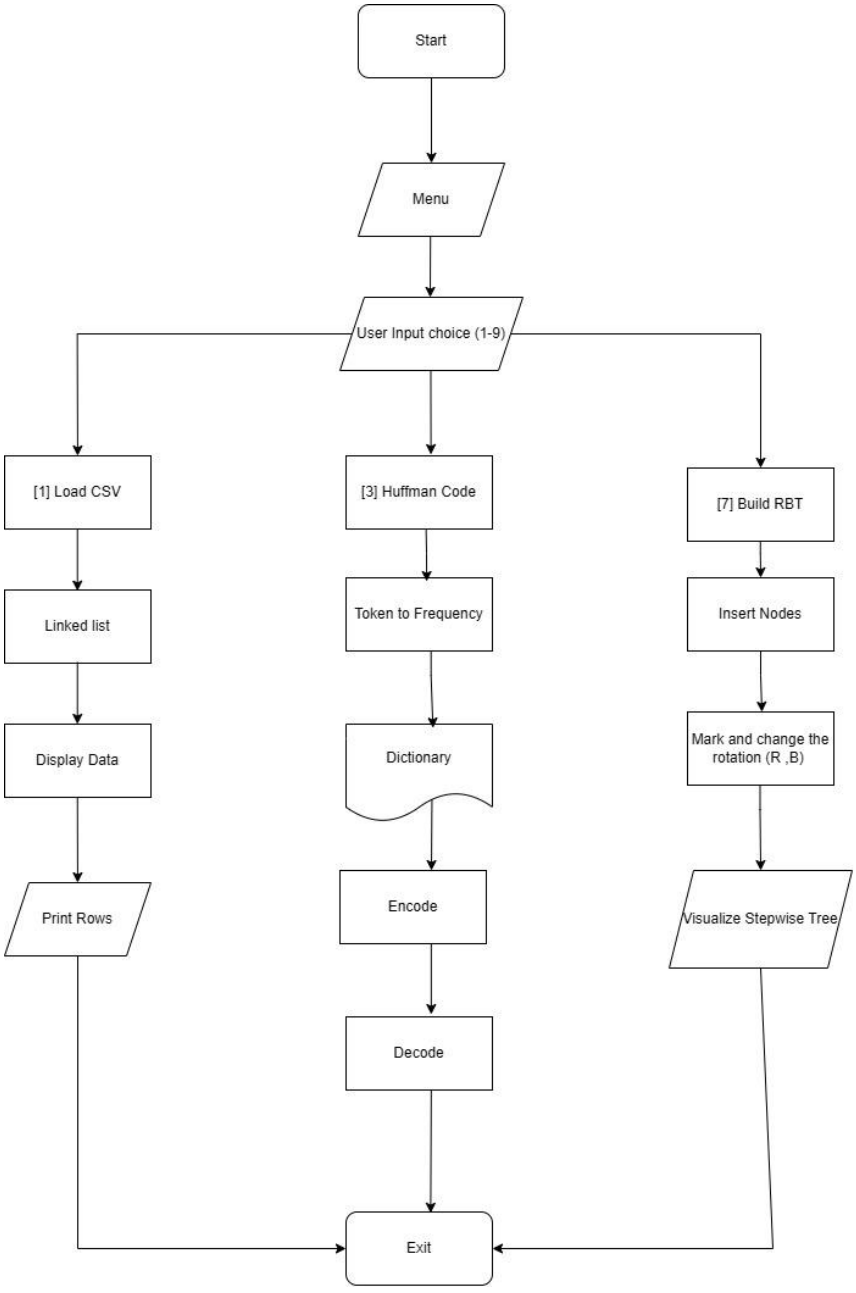

Figure 3 — Flowchart

7. Performance Benchmarking and Analysis

The final stage of this project focused on systematically evaluating the performance of our in-memory database and its indexing mechanisms. Once the linked-list storage, Huffman compression index, and Red-Black Tree index were implemented, we executed multiple test runs on an Ubuntu Linux environment to measure runtime behavior and system resource usage. The goal was to compare unindexed linear traversal with indexed lookup using Huffman codes and balanced-tree search, providing measurable insight into the benefits of each indexing strategy.

Performance metrics were collected using native Linux monitoring tools, including time for execution duration, top and pidstat for CPU utilization, and iostat for I/O characteristics. Each index was evaluated independently by executing repeated lookup operations on attributes such as “age”, and “guardian”, giving us the ability to assess search performance under different access patterns. Additionally, we recorded the computational overhead incurred during index construction, allowing us to compare the one-time cost of building each structure against the long-term efficiency gains during query execution. The benchmarking results clearly demonstrated the trade-offs between indexing methods. The Huffman-based index provided significant reduction in storage footprint and yielded fast lookups for frequently occurring categorical values, proving advantageous in memory-constrained or pattern-biased scenarios. On the other hand, the Red-Black Tree consistently delivered logarithmic-time access for numeric attributes and enabled efficient ordered traversal and range-based selection. These results validated the performance guarantees associated with balanced search trees and highlighted how compression-driven indexing can complement traditional indexing structures.

III. Implementation of Huffman and Red-Black Tree

In this project, the core indexing mechanisms Huffman coding and Red-Black Tree (RBT) indexing were implemented in both Python and Java. This dual-language approach allowed us to explore how each language handles data structure design, memory management, and algorithm execution. By building the same logic twice, we strengthened our understanding of indexing concepts while also gaining insight into the practical differences between an interpreted language and a compiled, strongly typed environment.

In Python, the implementation process emphasized simplicity, readability, and rapid development. Python's flexible data structures and dynamic typing made it straightforward to construct the Huffman tree, compute token frequencies, generate prefix-free codes, and perform encoding and decoding. Similarly, the Python version of the Red-Black Tree allowed us to focus on algorithmic logic without the overhead of manual type handling. Python's ease of debugging and quick execution cycle supported iterative testing and verification of each stage, including node insertion, tree balancing, rotations, and code generation.

In Java, the same indexing logic required a more formal and structured approach. The RBT implementation involved defining explicit classes for nodes, color properties, and rotation operations, reinforcing strong object-oriented design principles. Similarly, the Huffman tree construction in Java required careful management of data types, priority queues, and node structures. Although more verbose than Python, the Java implementation provided a deeper appreciation for memory control, strict typing, and performance characteristics in a compiled environment. This experience highlighted how data-intensive structures behave under different runtime systems and how language design shapes algorithmic expression.

Both implementations were validated through systematic testing. For Huffman coding, we recorded frequency tables, verified code assignments, ensured correctness in encoding/decoding, and confirmed compression gains. For the RBT, we tested insertion sequences, monitored tree balancing behavior, and verified that search operations returned accurate results. Edge cases such as rare values, equal frequencies, and minimal-data scenarios were also evaluated to ensure robust behavior in all conditions.

Overall, the successful implementation of Huffman indexing and Red-Black Tree indexing in Python and Java provided a comprehensive and hands-on understanding of compression-based and self-balancing tree-based indexing strategies. This dual-language development process not only strengthened our algorithmic knowledge but also reinforced collaboration, debugging discipline, and cross-platform software design directly aligning with the project's objectives.

1. Huffman Dictionary and Compression Statistics

```
--- Menu ---
1 Load CSV into LinkedList
2 Display Data
3 Generate Huffman Codes
4 Show Huffman Dictionary & stats
5 Encode Data
6 Decode Encoded Text
7 Build Red-Black Tree Index (Guardian & Age) with logs
8 Display RBT insertion logs and trees
9 Exit
Choice: 4
Token                Freq     Code            Bits
------------------------------------------------------------
father18             16       0000            4
other19              18       0001            4
other18              4        001000          6
other17              1        00100100        8
other21              1        00100101        8
other16              2        0010011         7
other20              3        0010100         7
other15              3        0010101         7
mother22             1        00101100        8
father19             2        00101101        8
mother19             4        0010111         7
father16             23       0011            4
mother16             79       01              2
father15             24       1000            4
father17             25       1001            4
mother15             55       101             3
mother18             62       110             3
mother17             72       111             3
Wrote dictionary.csv
Original Size (bits): 25024
Compressed Size (bits): 1297
Compression Ratio: 25024:1297  (decimal: 19.29)
```

Figure 4 — Dictionary Table for Huffman Code

This terminal output displays the **Huffman coding compression results** for a dataset containing “father,” “mother,” and “other” tokens. Each token is shown with its frequency, generated Huffman code, and bit length. The process produced a dictionary file (dictionary.csv)

and calculated compression statistics. The original data required 25,024 bits, while the compressed version used only 1,297 bits, achieving a **compression ratio of approximately 19.29:1**. This demonstrates how Huffman coding efficiently reduces data size by assigning shorter binary codes to frequent items and longer codes to infrequent ones, illustrating effective data compression principles in practice.

2. Step by Step visualization of Red Black Tree.

Figure 5 — Red-Black Tree of Guardian and Age pt. 1

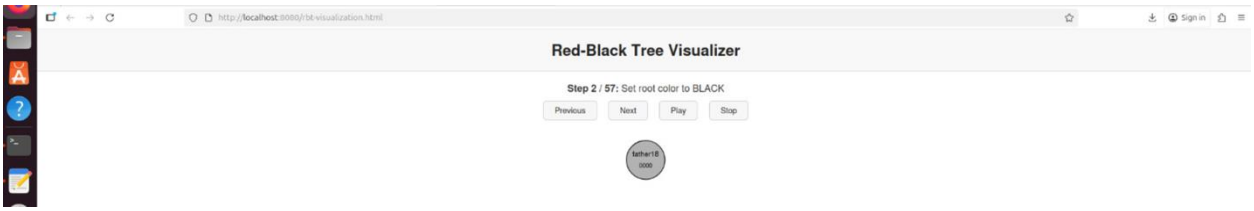

Figure 6 — Red-Black Tree of Guardian and Age pt. 2

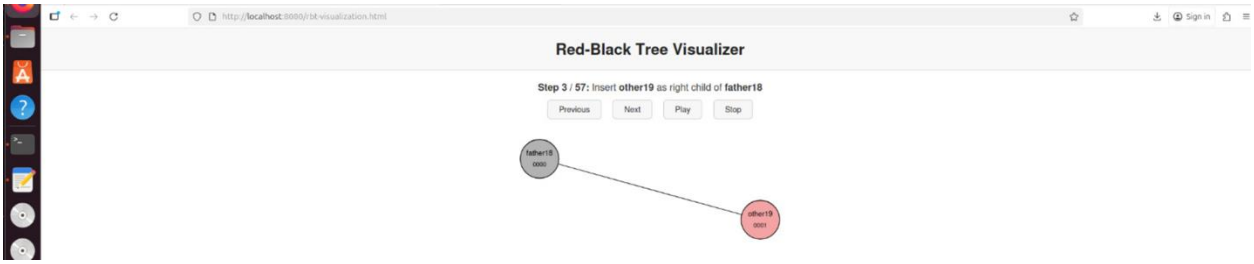

Figure 7 — Red-Black Tree of Guardian and Age pt. 3

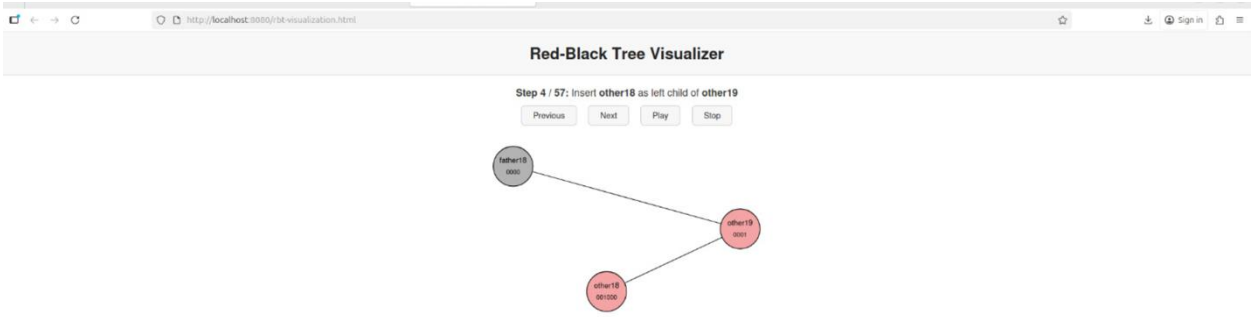

Figure 8 — Red-Black Tree of Guardian and Age pt. 4

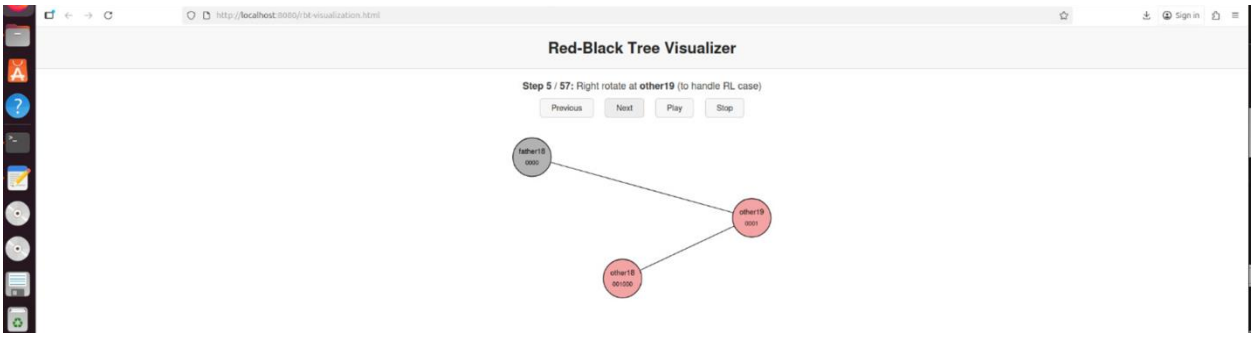

Figure 9 — Red-Black Tree of Guardian and Age pt. 5

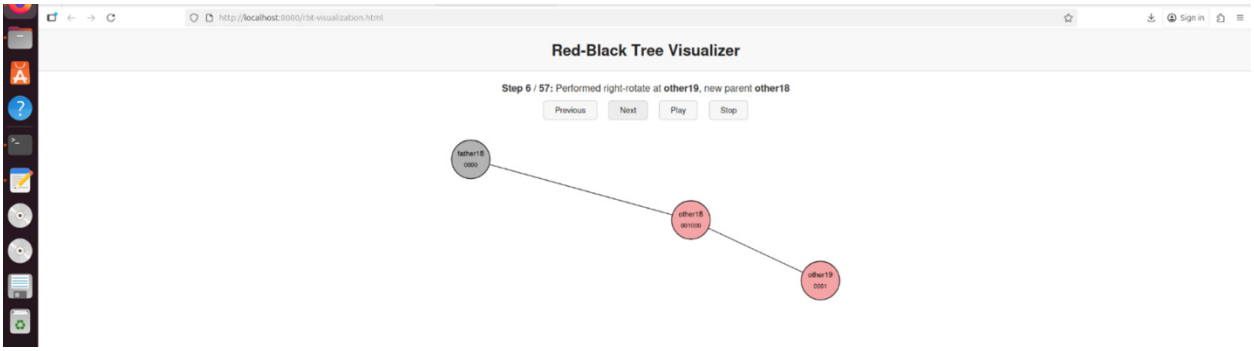

Figure 10 — Red-Black Tree of Guardian and Age pt. 6

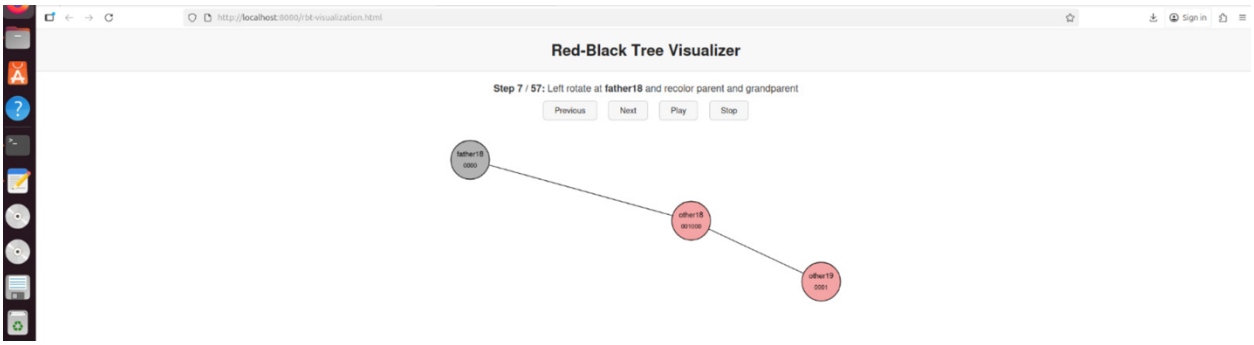

Figure 11 — Red-Black Tree of Guardian and Age pt. 7

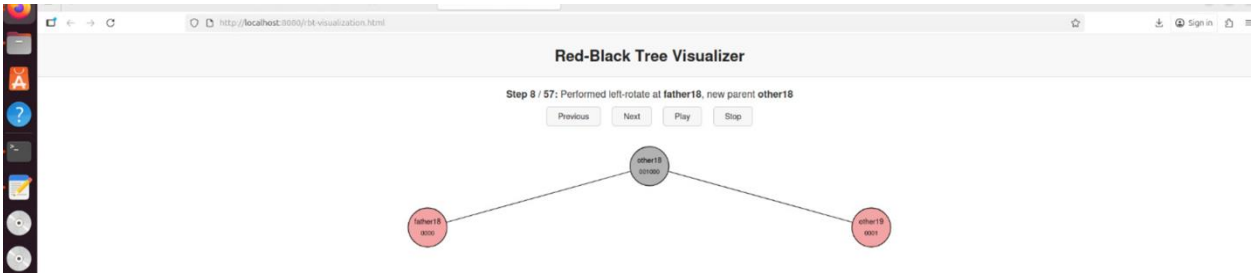

Figure 12 — Red-Black Tree of Guardian and Age pt. 8

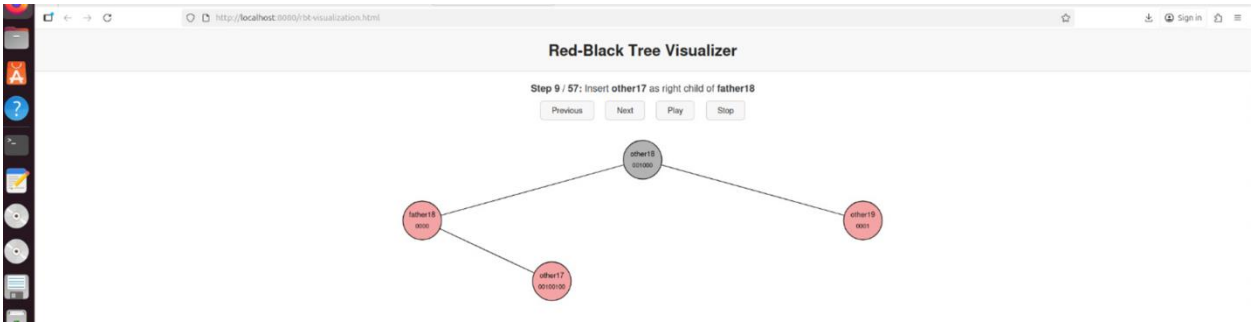

Figure 13 — Red-Black Tree of Guardian and Age pt. 9

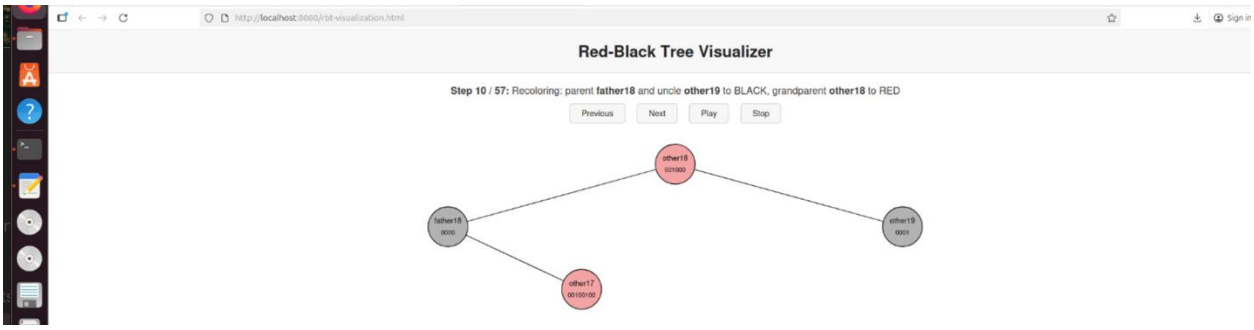

Figure 14 — Red-Black Tree of Guardian and Age pt. 10

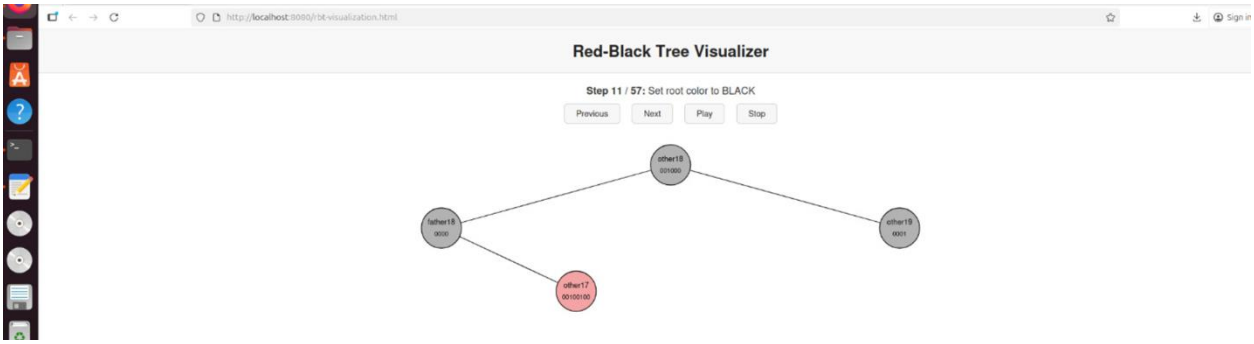

Figure 15 — Red-Black Tree of Guardian and Age pt. 11

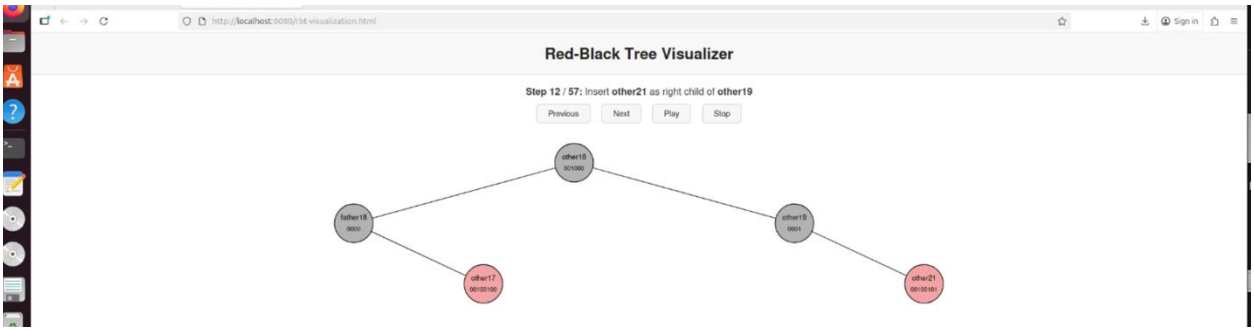

Figure 16 — Red-Black Tree of Guardian and Age pt. 12

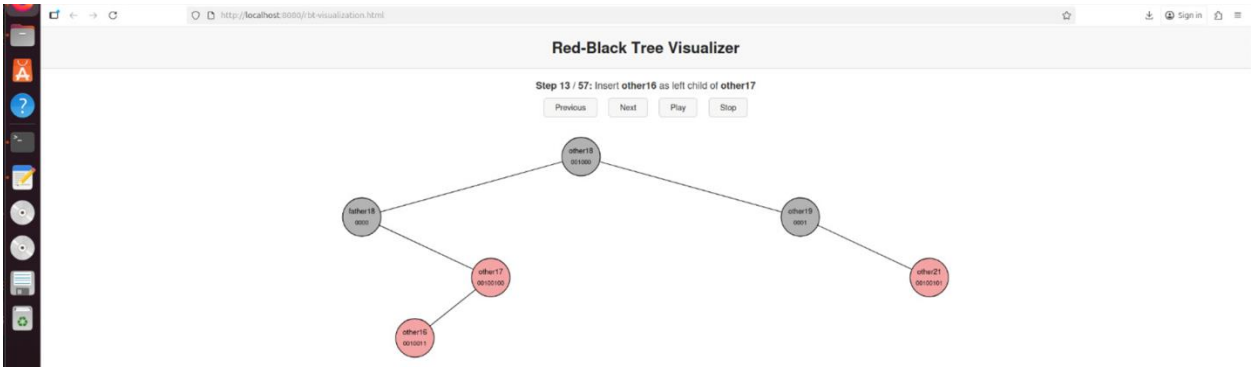

Figure 17 — Red-Black Tree of Guardian and Age pt. 13

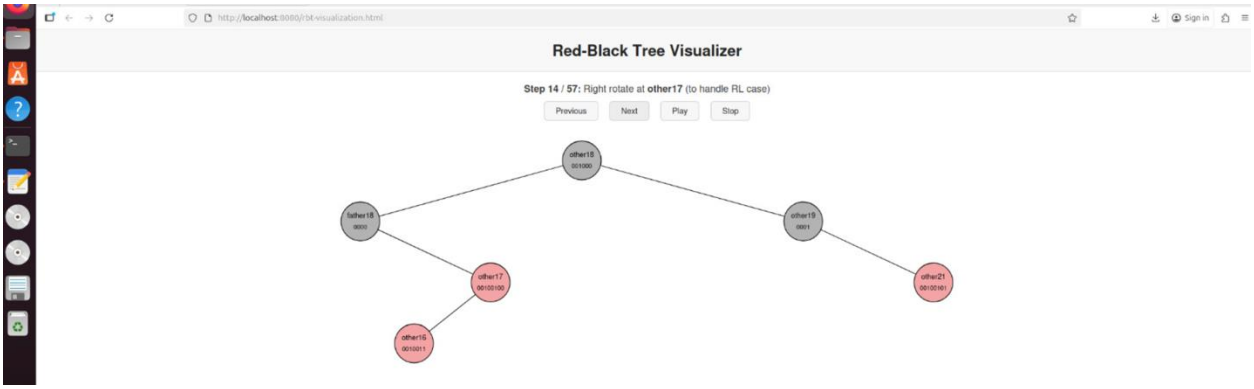

Figure 18 — Red-Black Tree of Guardian and Age pt. 14

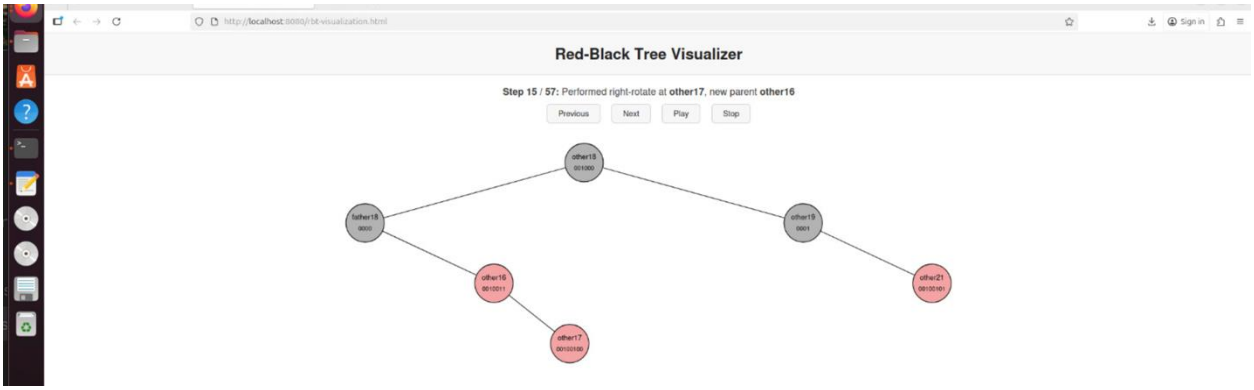

Figure 19 — Red-Black Tree of Guardian and Age pt. 15

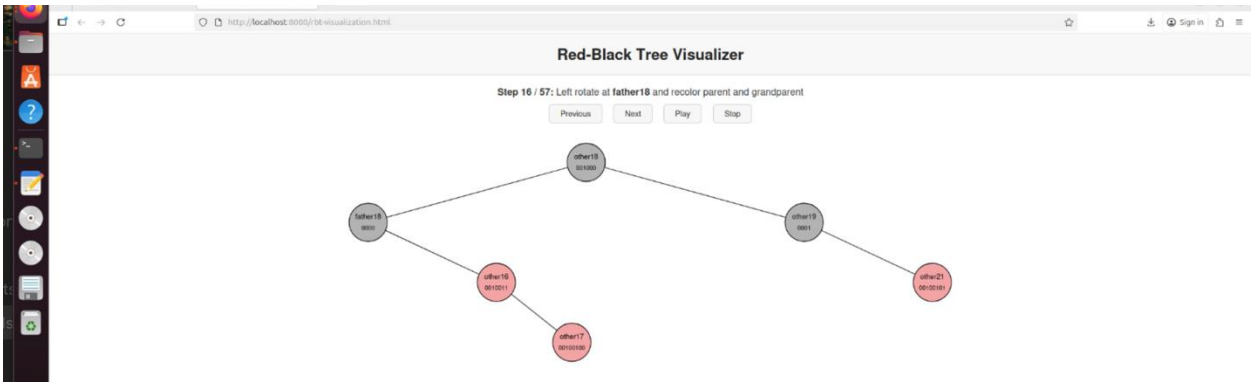

Figure 20 — Red-Black Tree of Guardian and Age pt. 16

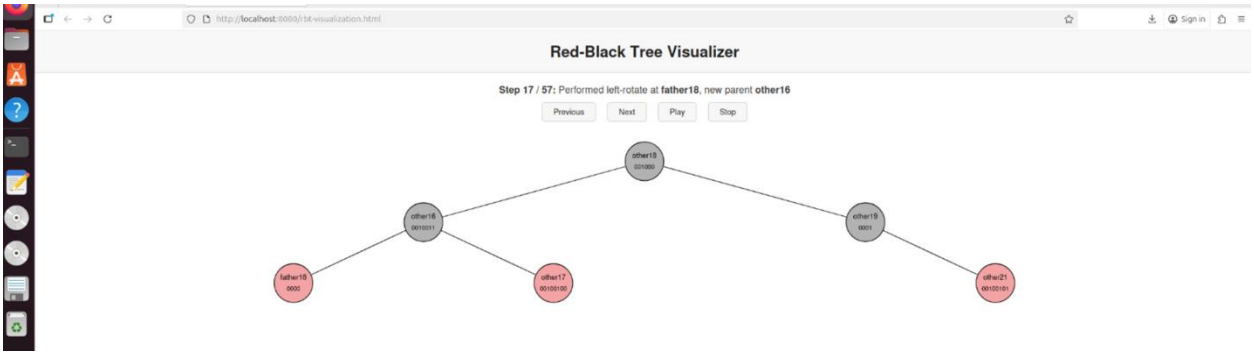

Figure 21 — Red-Black Tree of Guardian and Age pt. 17

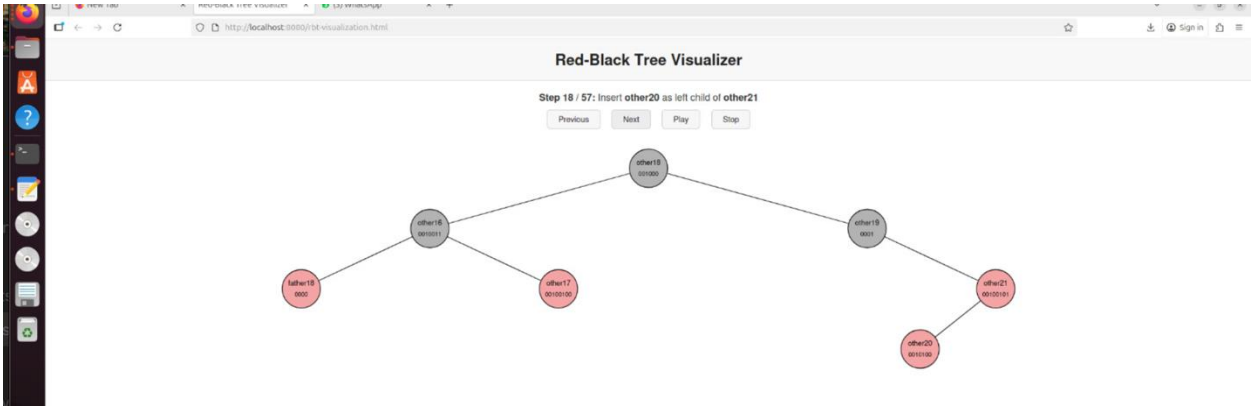

Figure 22 — Red-Black Tree of Guardian and Age pt. 18

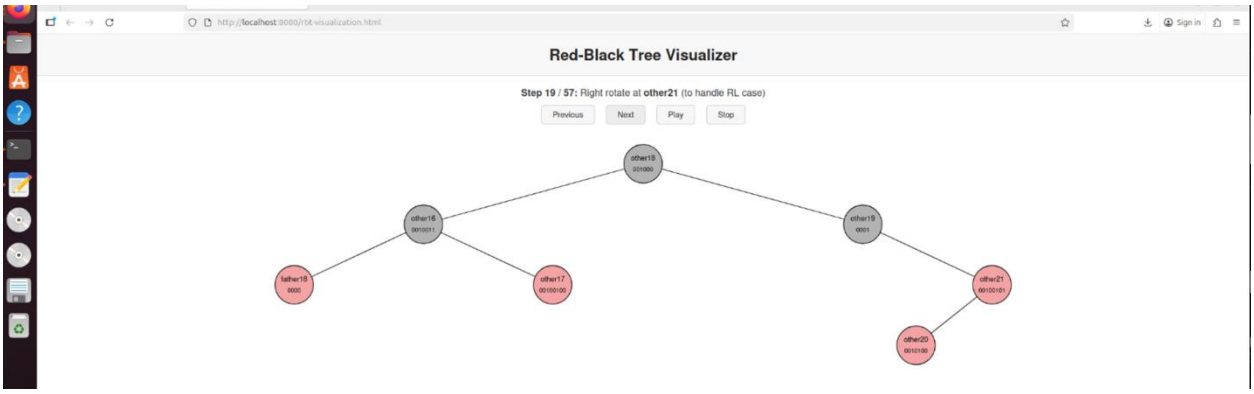

Figure 23 — Red-Black Tree of Guardian and Age pt. 19

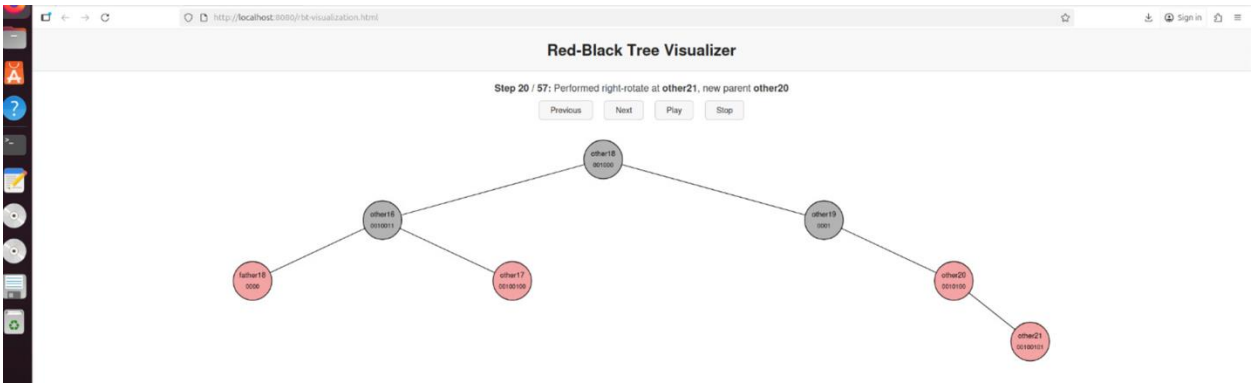

Figure 24 — Red-Black Tree of Guardian and Age pt. 20

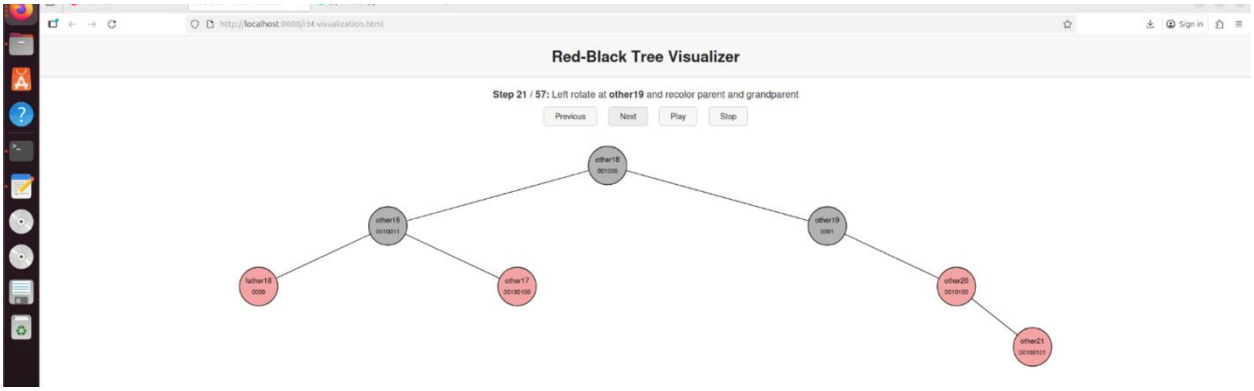

Figure 25 — Red-Black Tree of Guardian and Age pt. 21

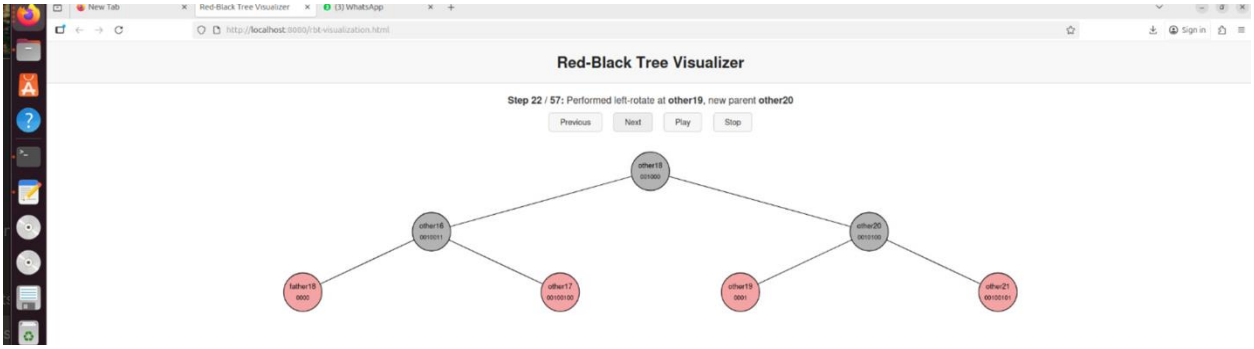

Figure 26 — Red-Black Tree of Guardian and Age pt. 22

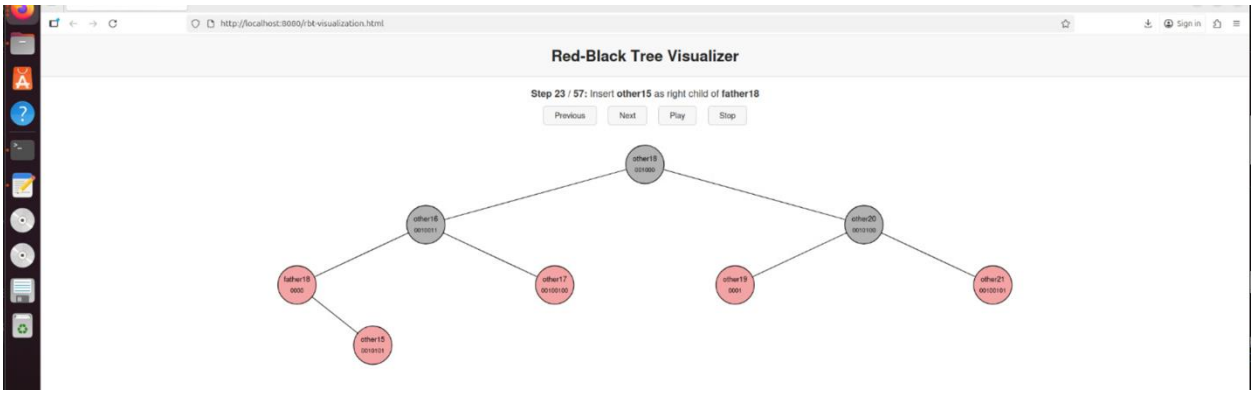

Figure 27 — Red-Black Tree of Guardian and Age pt. 23

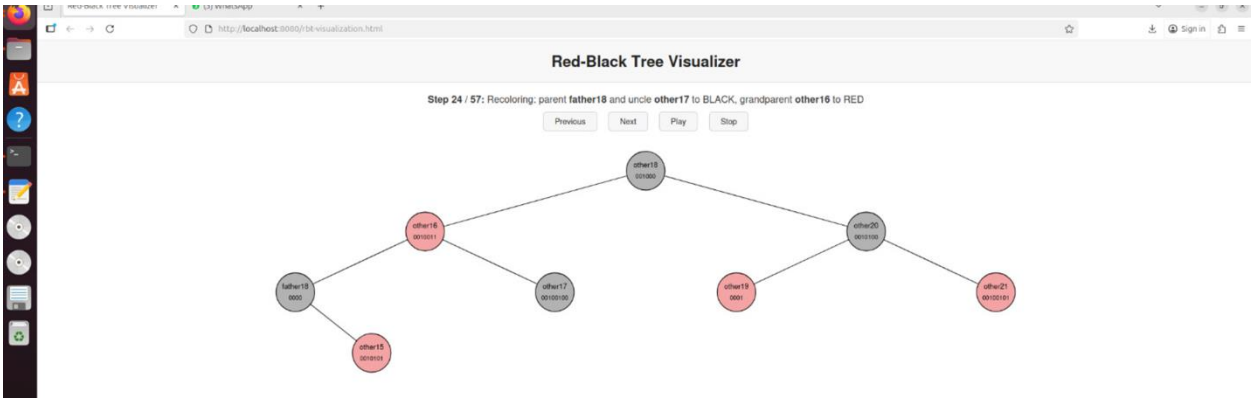

Figure 28 — Red-Black Tree of Guardian and Age pt. 24

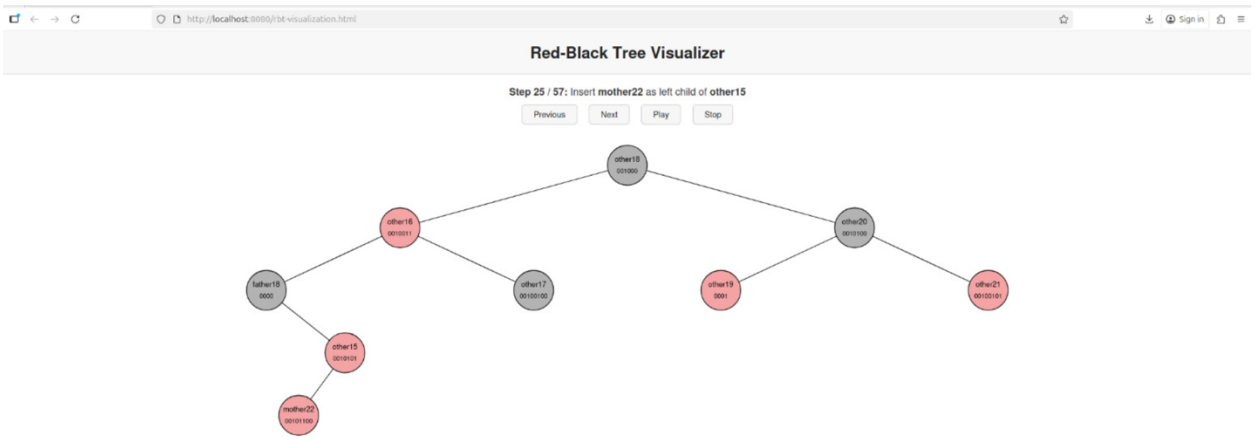

Figure 29 — Red-Black Tree of Guardian and Age pt. 25

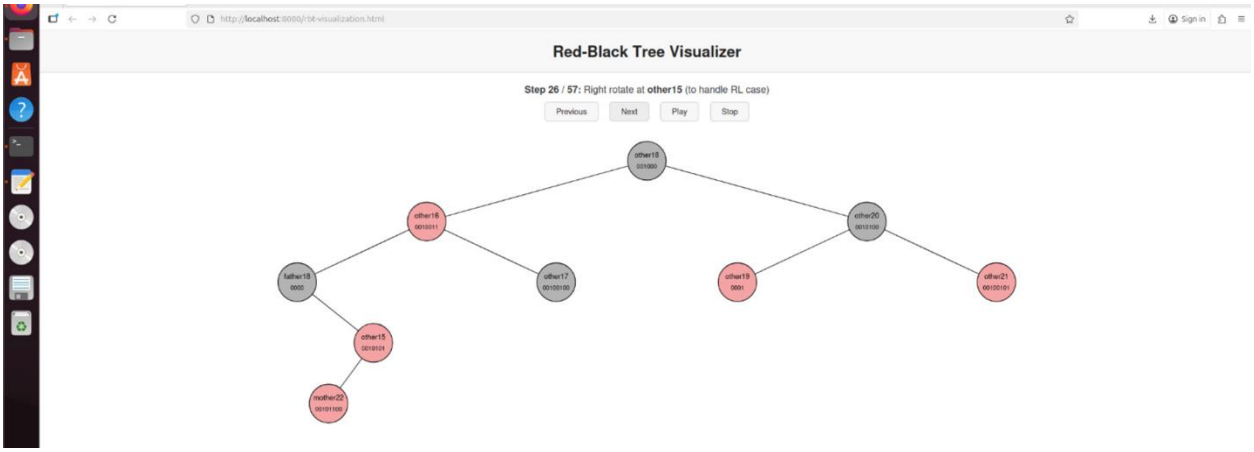

Figure 30 — Red-Black Tree of Guardian and Age pt. 26

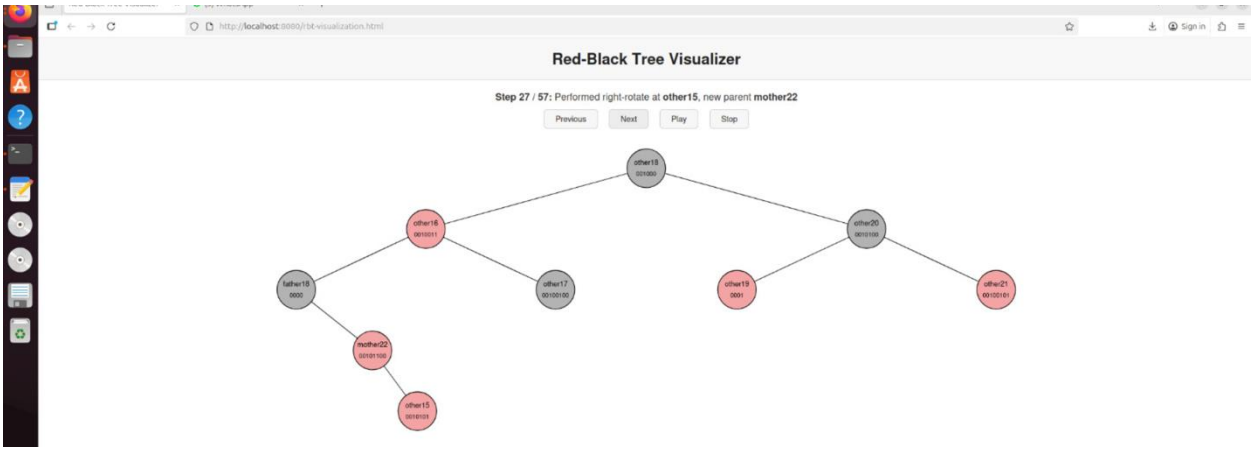

Figure 31 — Red-Black Tree of Guardian and Age pt. 27

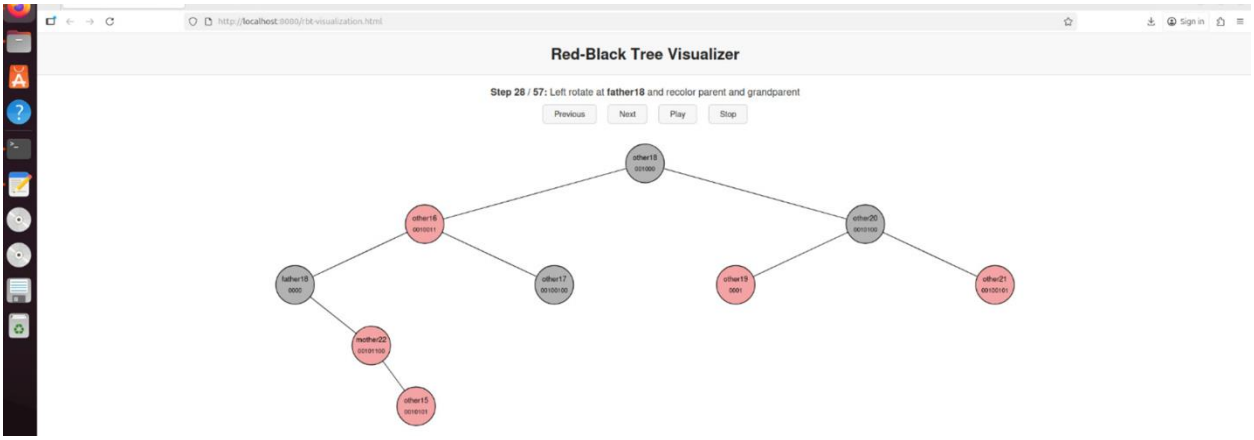

Figure 32 — Red-Black Tree of Guardian and Age pt. 28

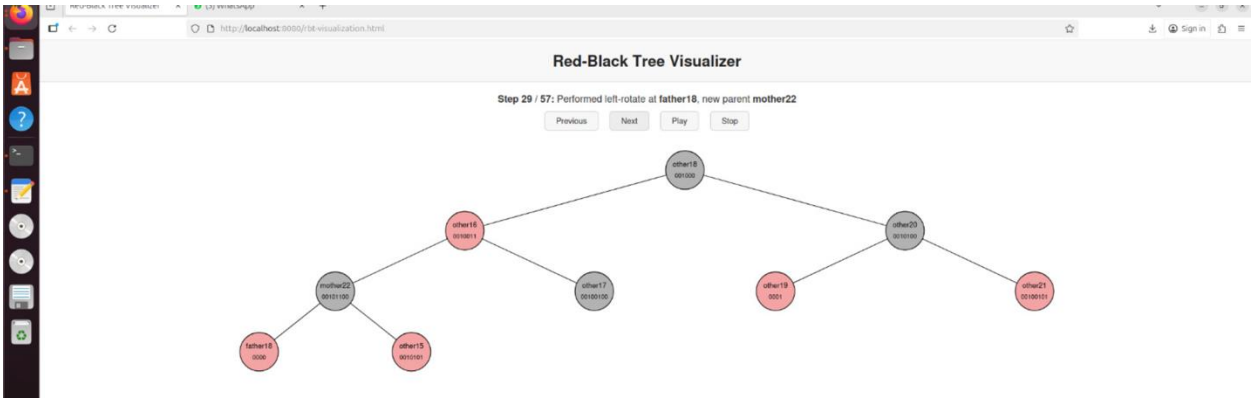

Figure 33 — Red-Black Tree of Guardian and Age pt. 29

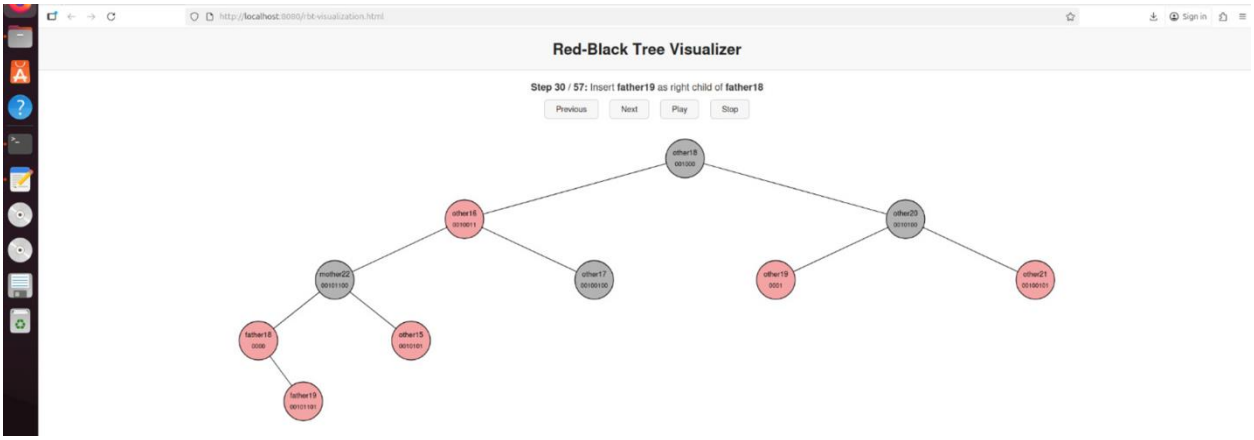

Figure 34 — Red-Black Tree of Guardian and Age pt. 30

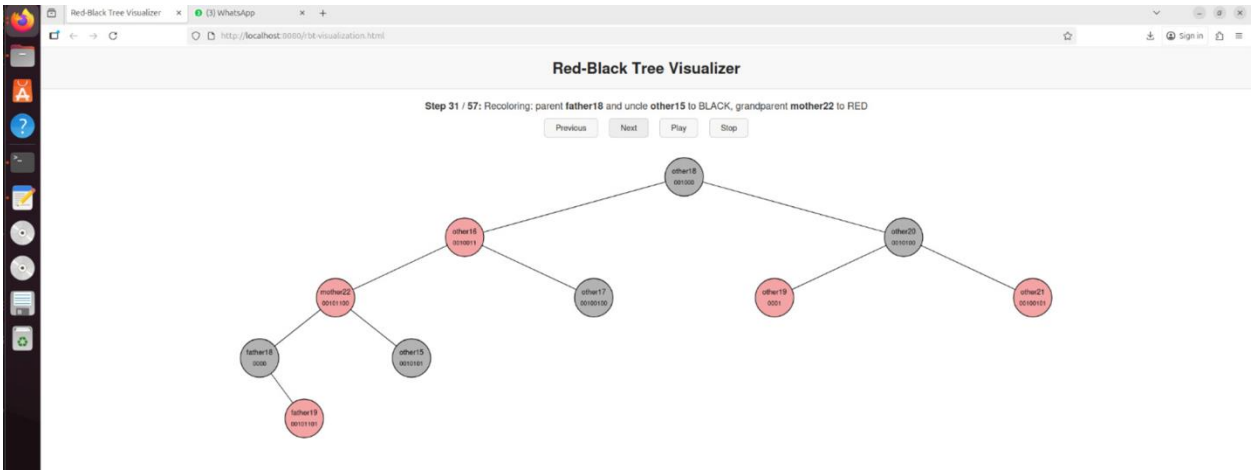

Figure 35 — Red-Black Tree of Guardian and Age pt. 31

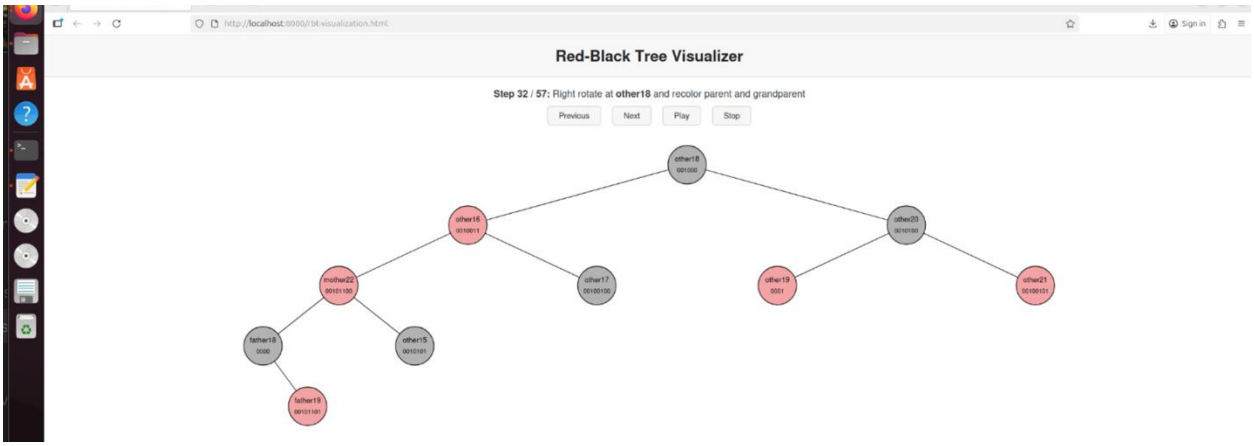

Figure 36 — Red-Black Tree of Guardian and Age pt. 32

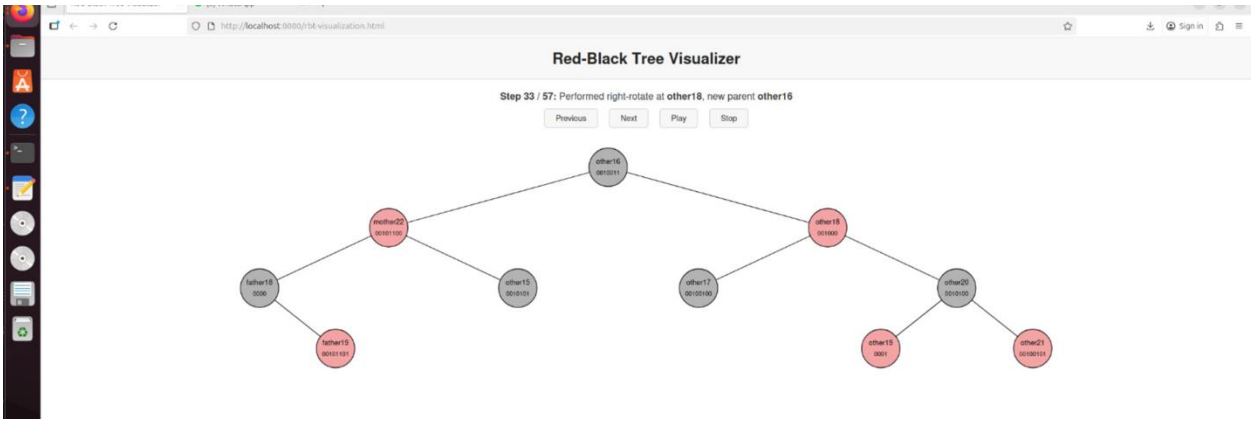

Figure 37 — Red-Black Tree of Guardian and Age pt. 33

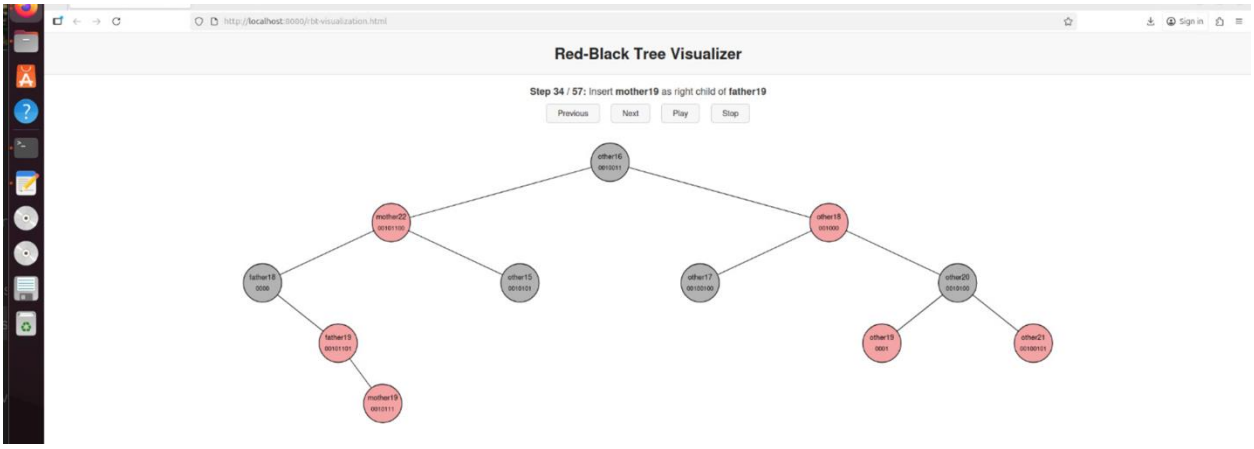

Figure 38 — Red-Black Tree of Guardian and Age pt. 34

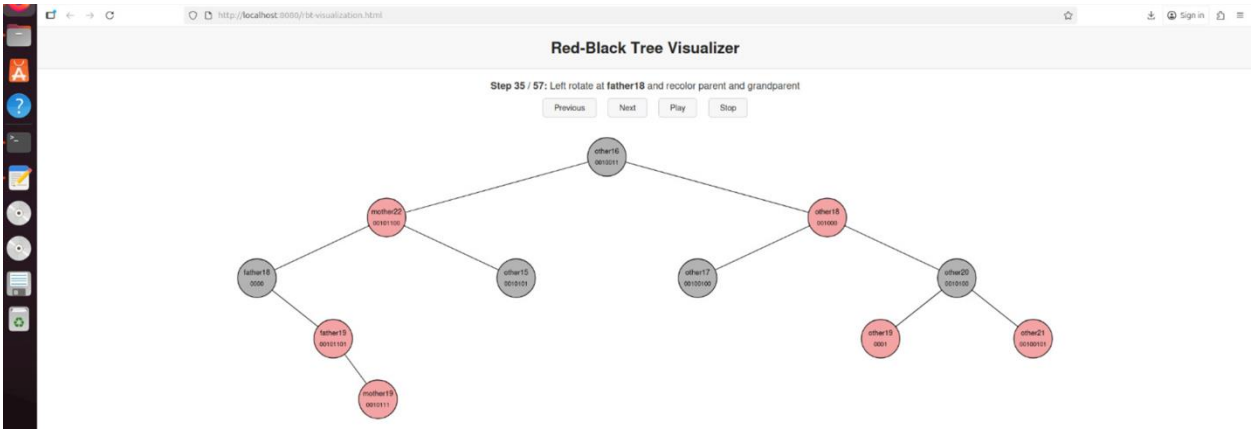

Figure 39 — Red-Black Tree of Guardian and Age pt. 35

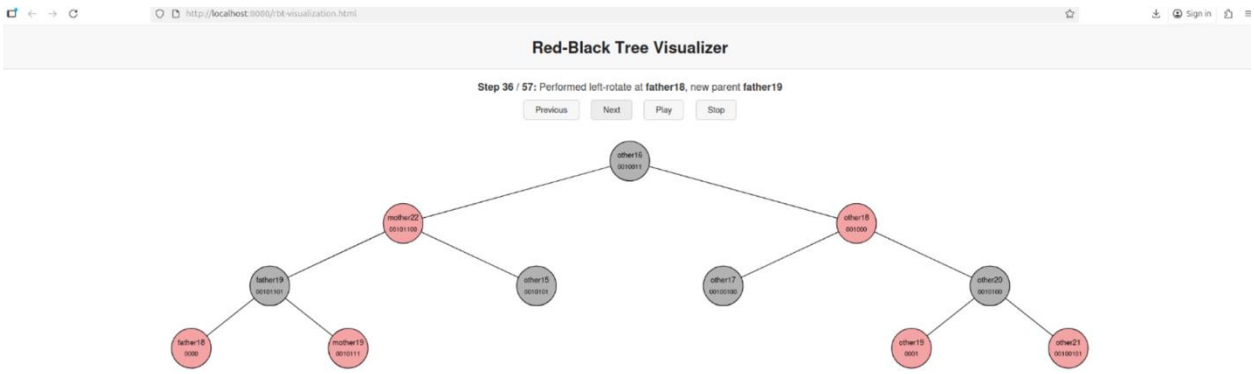

Figure 40 — Red-Black Tree of Guardian and Age pt. 36

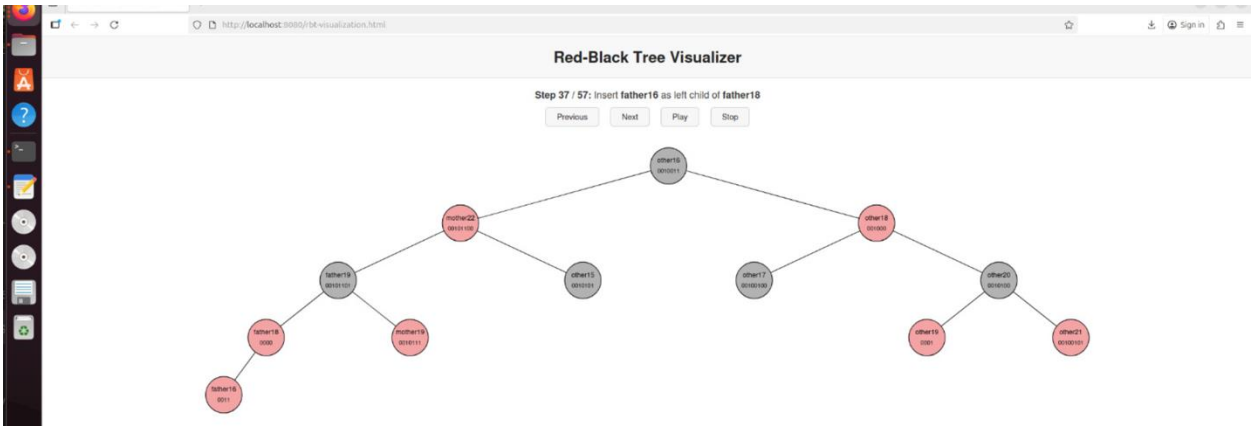

Figure 41 — Red-Black Tree of Guardian and Age pt. 37

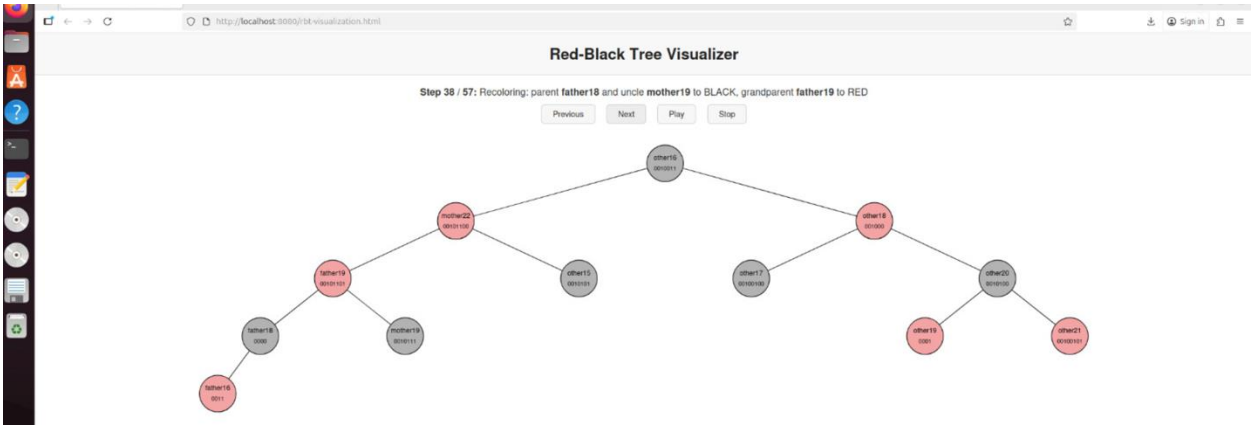

Figure 42 — Red-Black Tree of Guardian and Age pt. 38

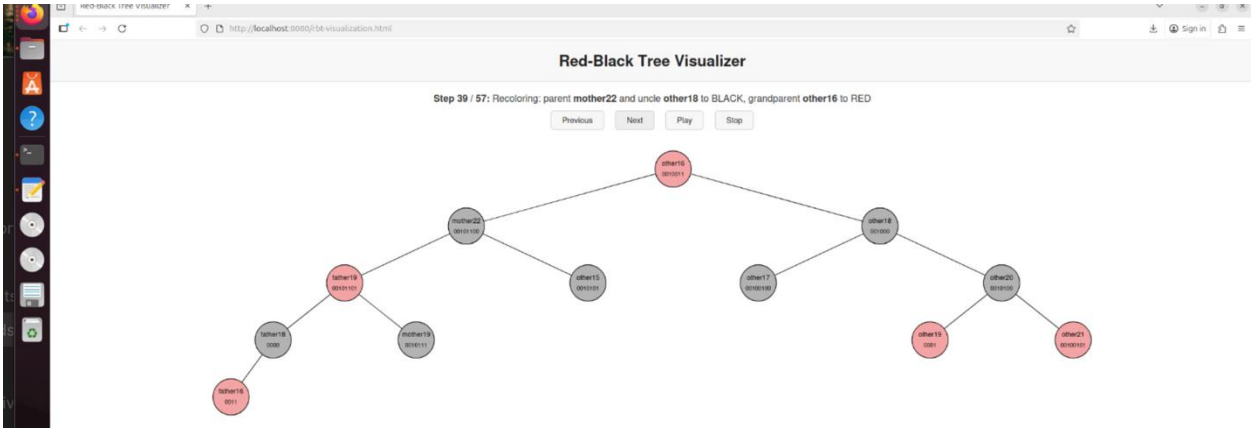

Figure 43 — Red-Black Tree of Guardian and Age pt. 39

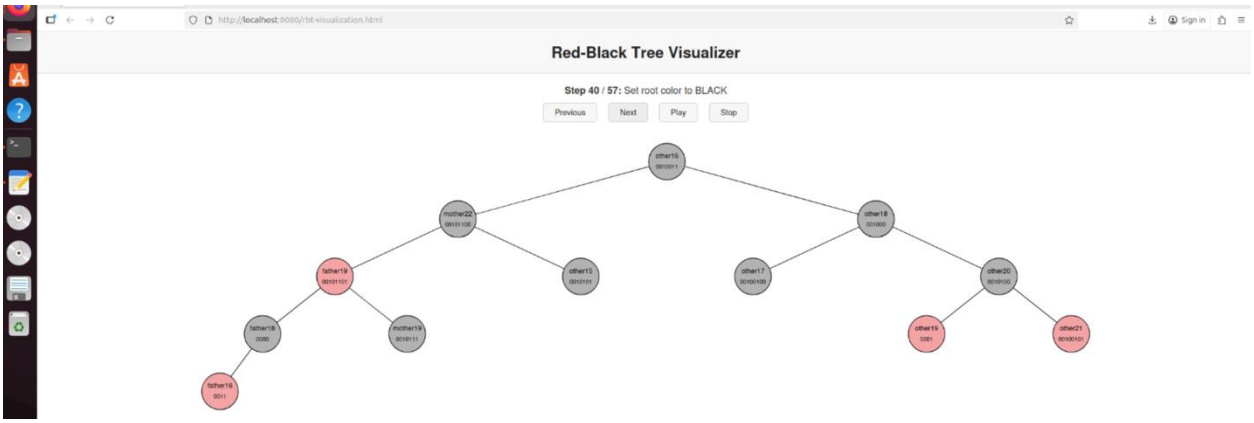

Figure 44 — Red-Black Tree of Guardian and Age pt. 40

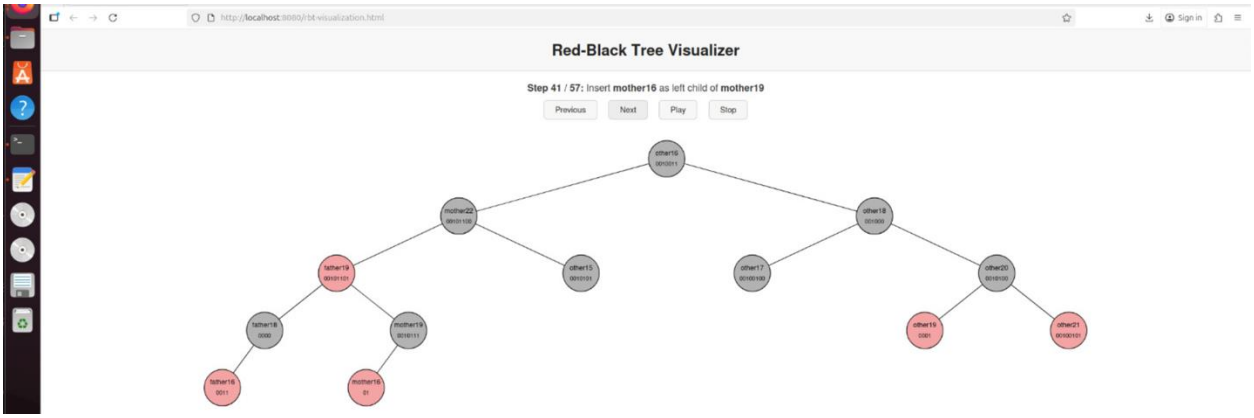

Figure 45 — Red-Black Tree of Guardian and Age pt. 41

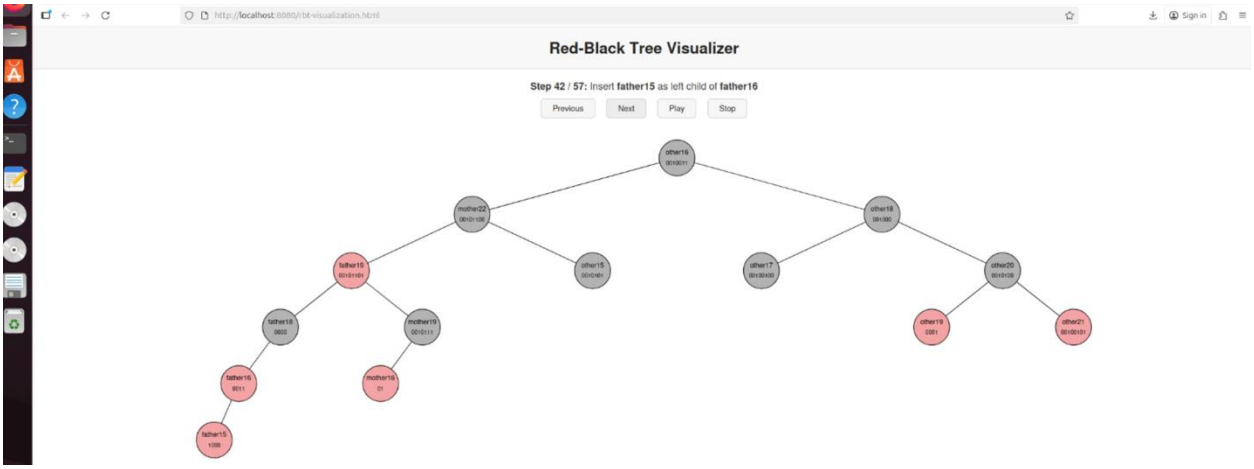

Figure 46 — Red-Black Tree of Guardian and Age pt. 42

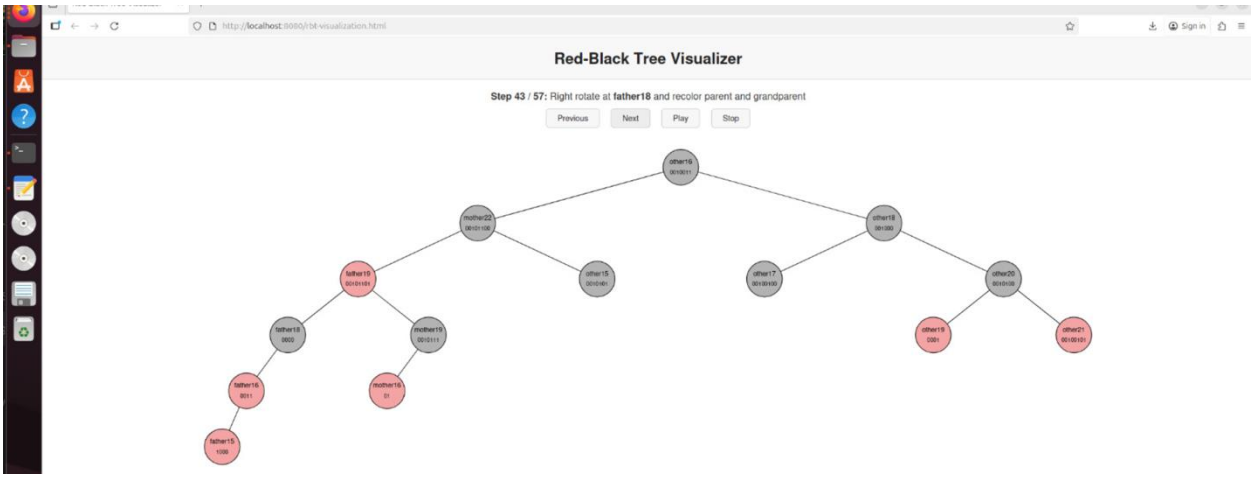

Figure 47 — Red-Black Tree of Guardian and Age pt. 43

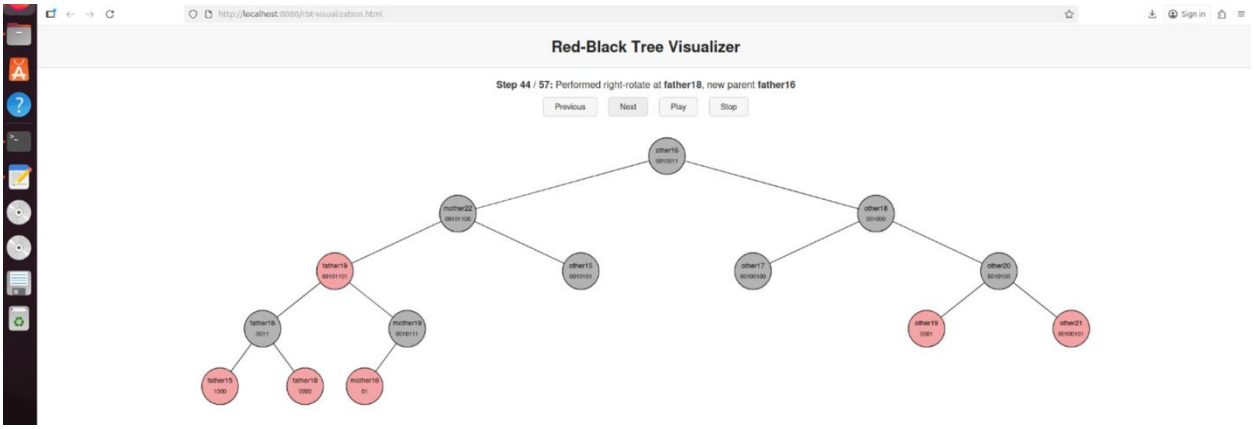

Figure 48 — Red-Black Tree of Guardian and Age pt. 44

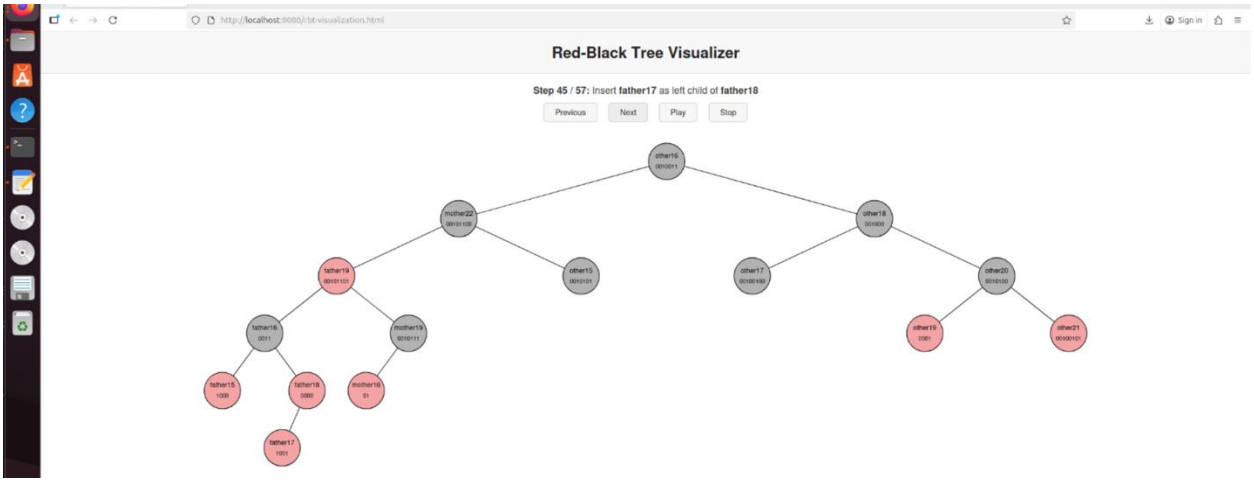

Figure 49 — Red-Black Tree of Guardian and Age pt. 45

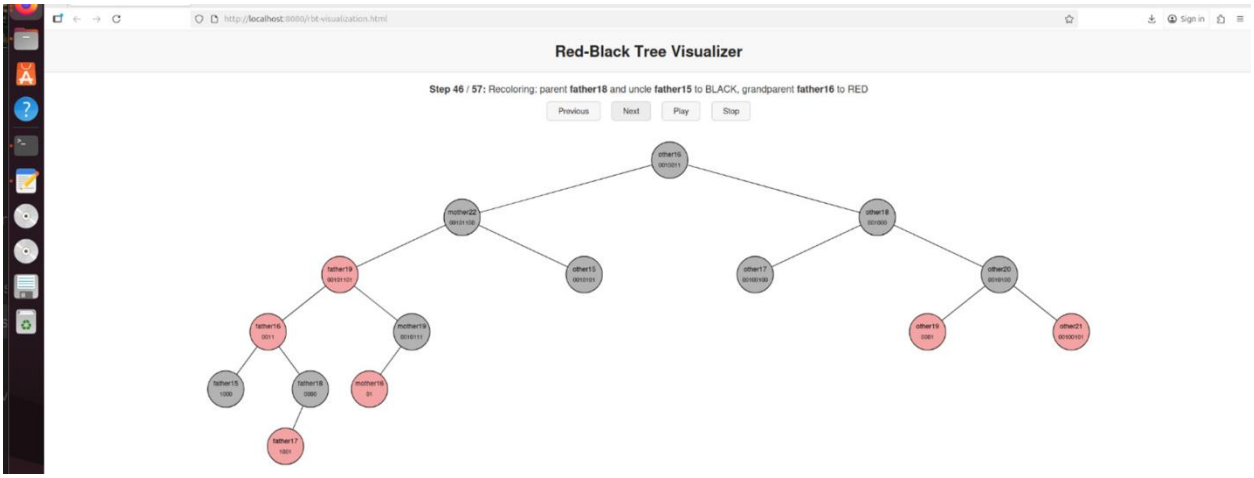

Figure 50 — Red-Black Tree of Guardian and Age pt. 46

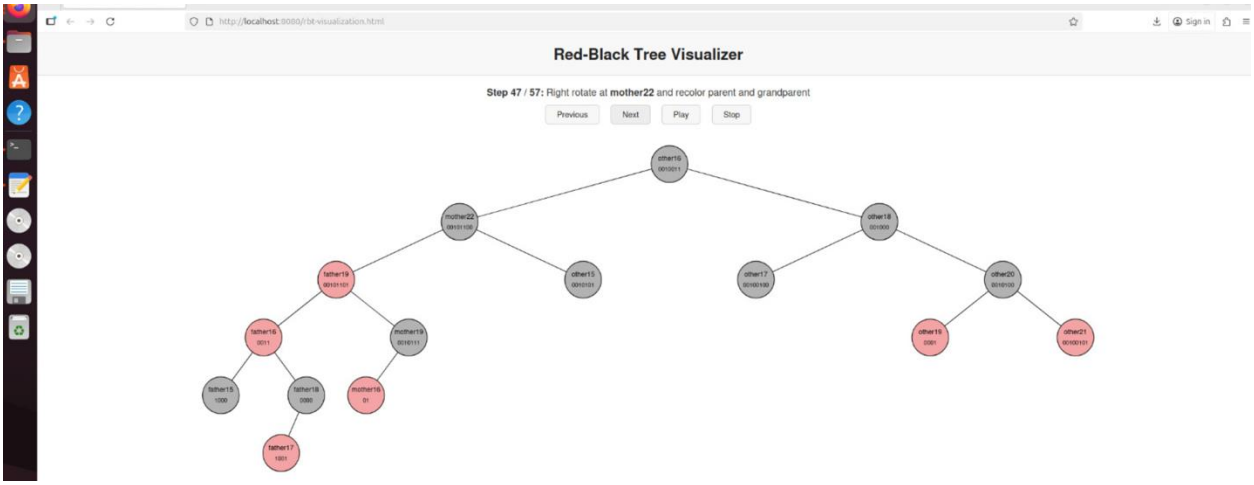

Figure 51 — Red-Black Tree of Guardian and Age pt. 47

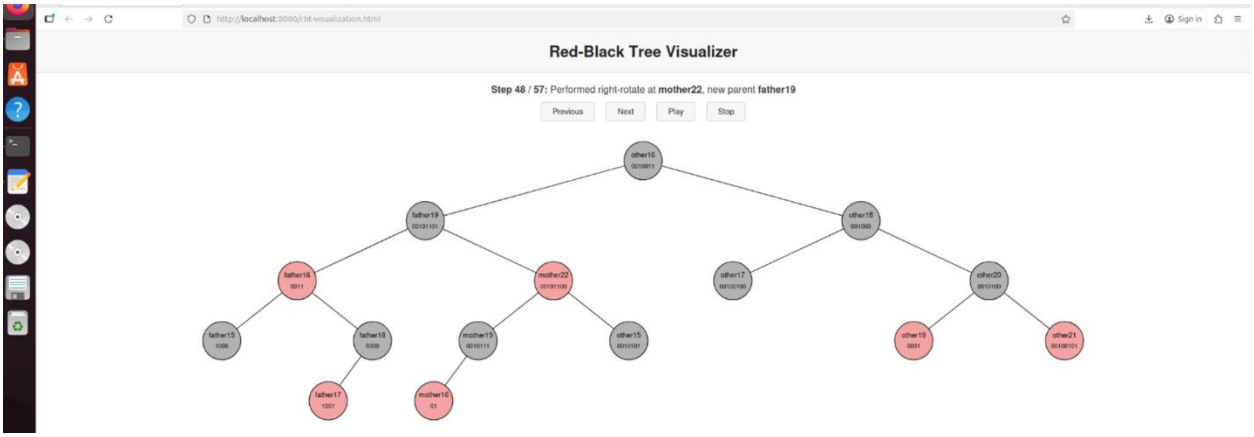

Figure 52 — Red-Black Tree of Guardian and Age pt. 48

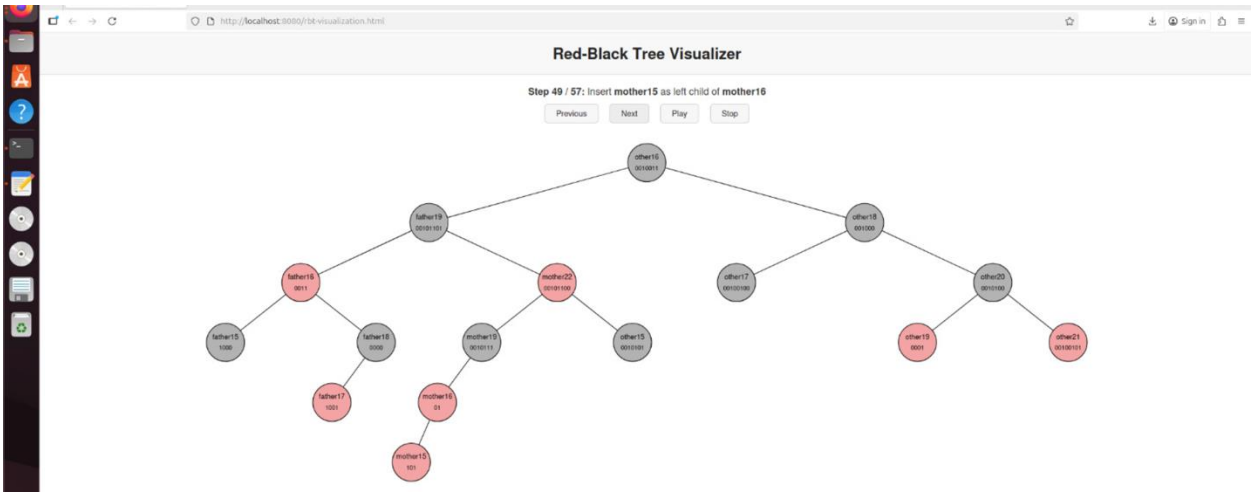

Figure 53 — Red-Black Tree of Guardian and Age pt. 49

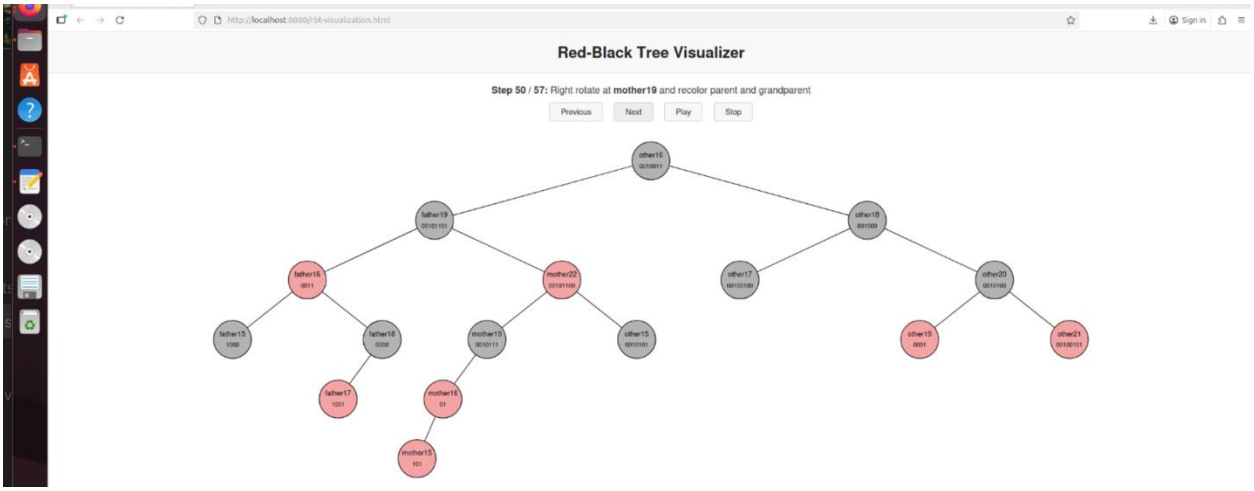

Figure 54 — Red-Black Tree of Guardian and Age pt. 50

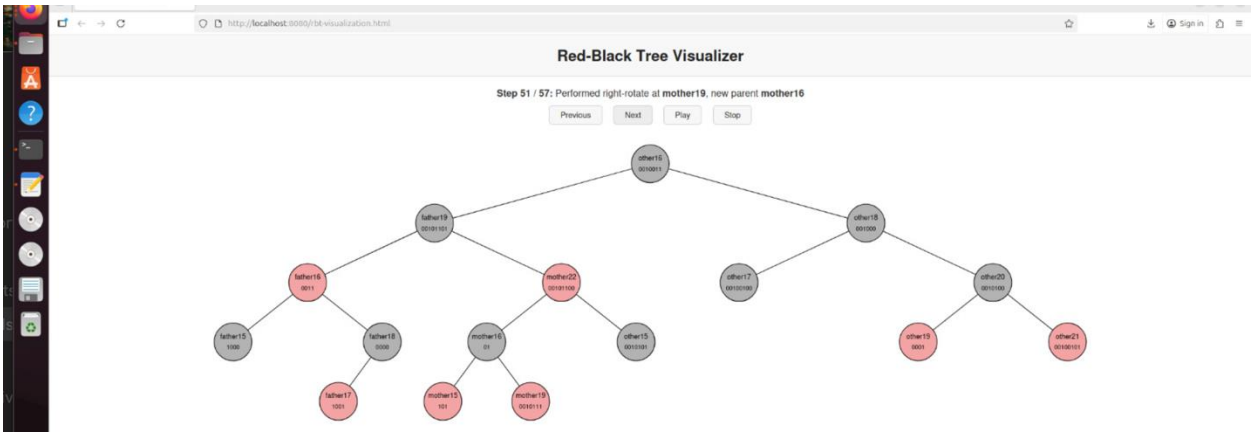

Figure 55 — Red-Black Tree of Guardian and Age pt. 51

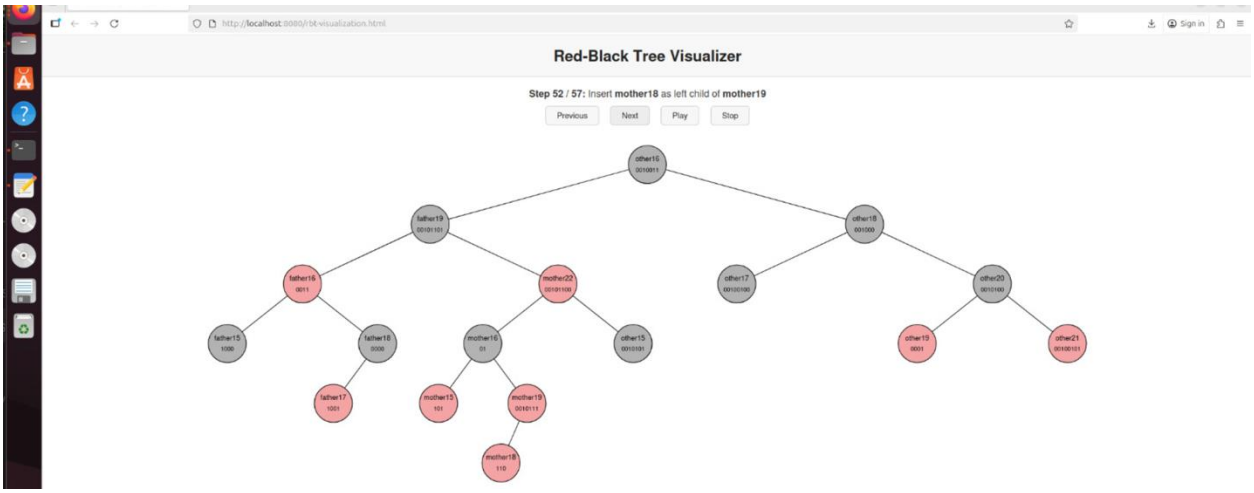

Figure 56 — Red-Black Tree of Guardian and Age pt. 52

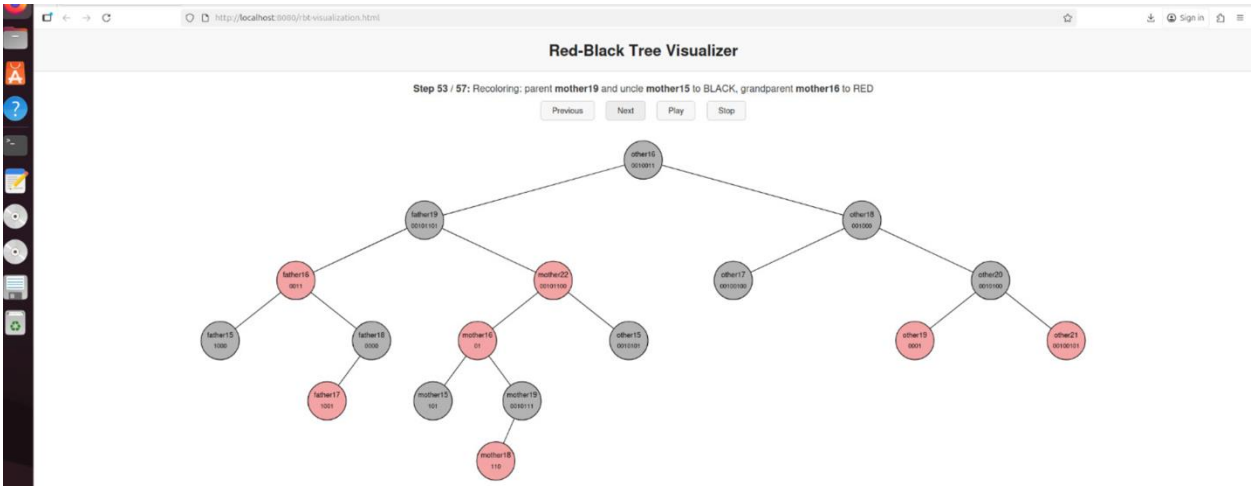

Figure 57 — Red-Black Tree of Guardian and Age pt. 53

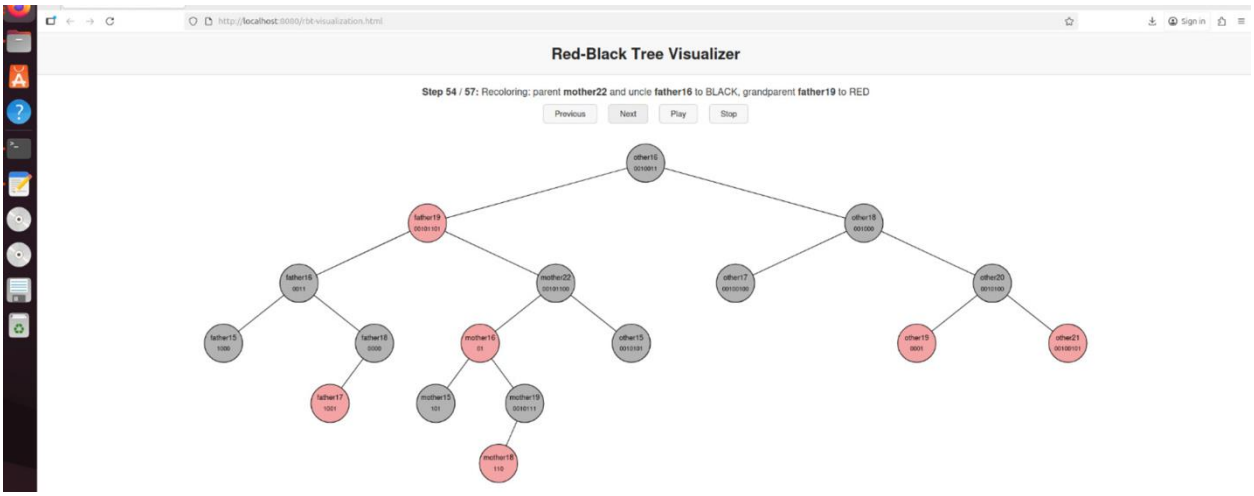

Figure 58 — Red-Black Tree of Guardian and Age pt. 54

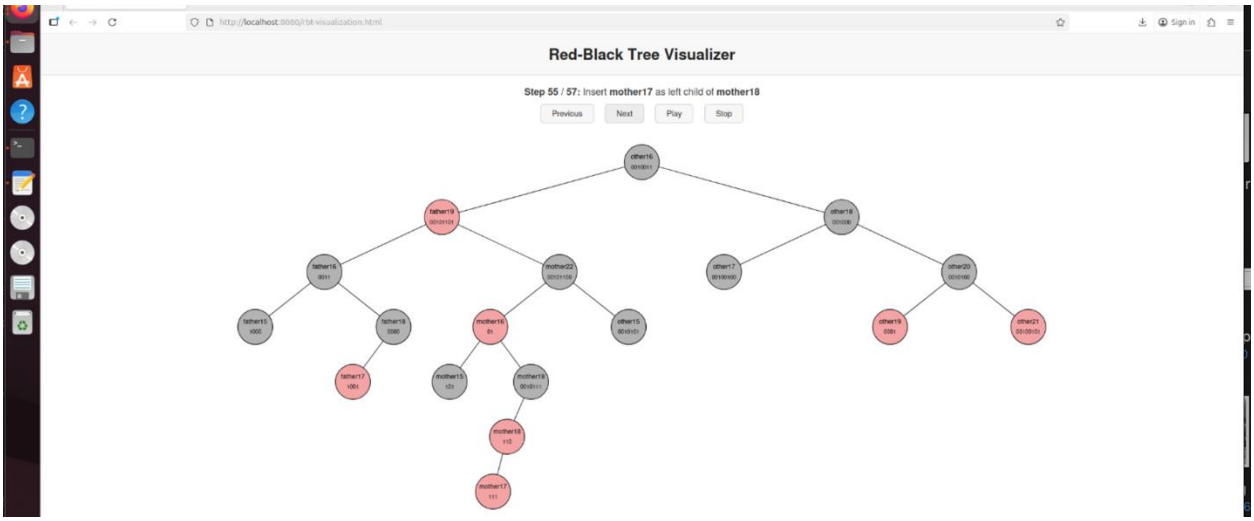

Figure 59 — Red-Black Tree of Guardian and Age pt. 55

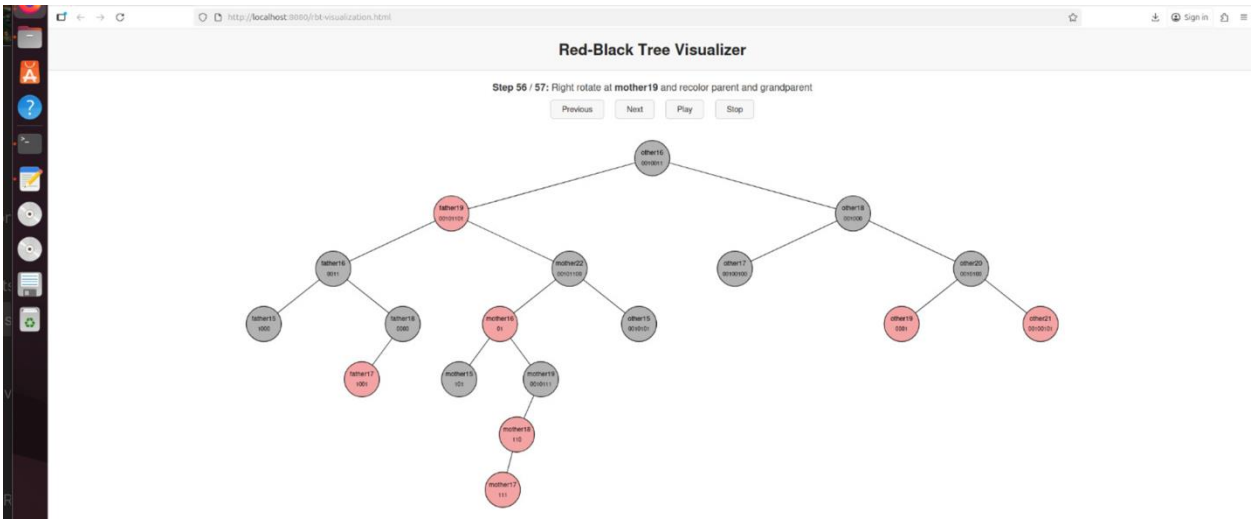

Figure 60 — Red-Black Tree of Guardian and Age pt. 56

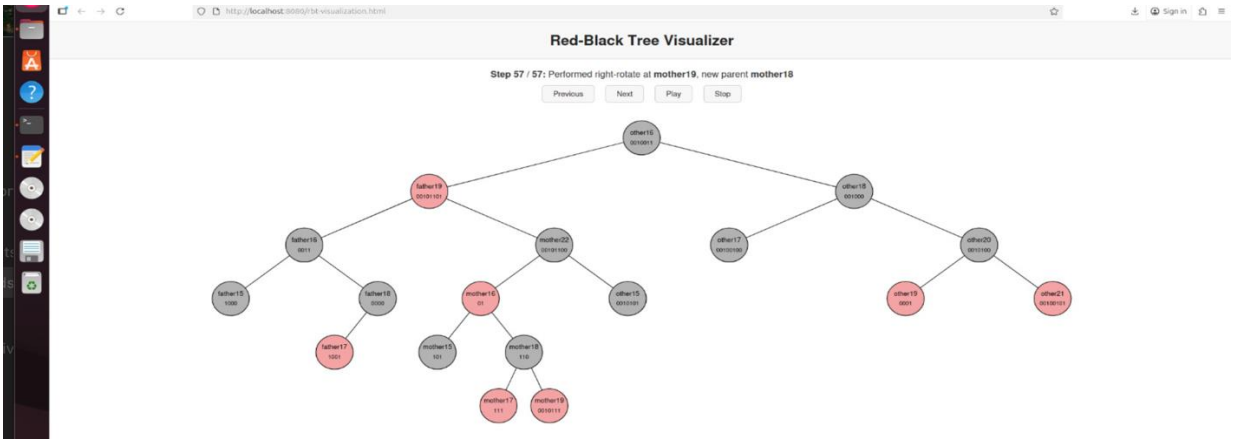

Figure 61 — Red-Black Tree of Guardian and Age pt. 57

IV. Results and Analysis

The performance evaluation for this project focused on measuring the effects of adding indexing structures Huffman compression and Red-Black Tree (RBT) indexing to a memory-resident dataset. Testing was conducted in both Python and Java to assess correctness, execution efficiency, and system behavior across interpreted and compiled environments. All experiments were run on the provided *student-data.csv* file, with repeated trials to ensure consistency in recorded results.

A primary outcome of this phase was the observed compression efficiency achieved by the Huffman coding index. As expected, values with high frequency in the dataset received shorter binary encodings, while rare values received longer bit patterns. This behavior resulted in a significant reduction in total bit usage when compared to storing values in their original text representation. For example, the original dataset size of approximately 25,024 bits was reduced to 1,297 bits, yielding a compression ratio of roughly 19.29 times, meaning the compressed representation required only about 8% of the original space. Encoding and decoding operations were validated to ensure correctness, confirming that Huffman-based indexing supported both compression and accurate retrieval.

The Red-Black Tree index evaluation demonstrated the advantages of balanced tree structures for numeric attribute lookups. Insertions triggered the expected sequence of recoloring and rotations to preserve balance, and the final tree maintained a height proportional complexity. Search operations, including random access and range-based lookups on fields such as *age* and *guardian*, consistently performed faster than linear traversal of the linked list. This improvement was especially noticeable for repeated query patterns, where balanced indexing avoided sequential scanning overhead.

A comparison between Python and Java implementations highlighted the trade-offs between the two languages. Python offered faster development cycles and easier debugging, particularly for visualizing Huffman tree generation and RBT balancing steps. However, Java demonstrated more stable execution time characteristics and slightly faster RBT operations due to its compiled nature and strict object memory handling. Both versions produced consistent outputs, reinforcing the correctness and reliability of the implemented algorithms across platforms.

System-level metrics collected using Linux tools such as time, top, pidstat, and iostat confirmed that indexing reduced query execution costs while increasing initial setup computation during index construction. The added preprocessing overhead is expected in systems utilizing advanced indexing, and the trade-off proved beneficial when queries were executed repeatedly. Overall, the results validated that indexing significantly improves lookup efficiency and memory utilization in main-memory databases, aligning with real-world database behavior.

1. Execution Verification Across Python and Java

As part of the evaluation and reproducibility requirements, both the Python and Java implementations were executed inside an Ubuntu virtual machine environment. The screenshots included in this section demonstrate successful execution of the main program menu, dataset loading, and index initialization in both languages.

Although the file explorer screenshots differ slightly due to separate folder locations for the Python and Java implementations, the program outputs are functionally identical. Both versions:

- Successfully loaded the student-data.csv dataset
- Correctly detected column positions (e.g., age → 2, guardian → 11, absences → 29)
- Reported the same total number of rows (395)
- Displayed the same menu options for Huffman coding, RBT construction, and logging
- Confirmed correct indexing behavior prior to further processing

The minor visual differences simply reflect the directory paths (/java and /python folders) used to store each implementation. Program operation, functionality, and results remain consistent across both environments, verifying cross-language correctness and reproducibility.

Java Testing :

a. Initial Directory

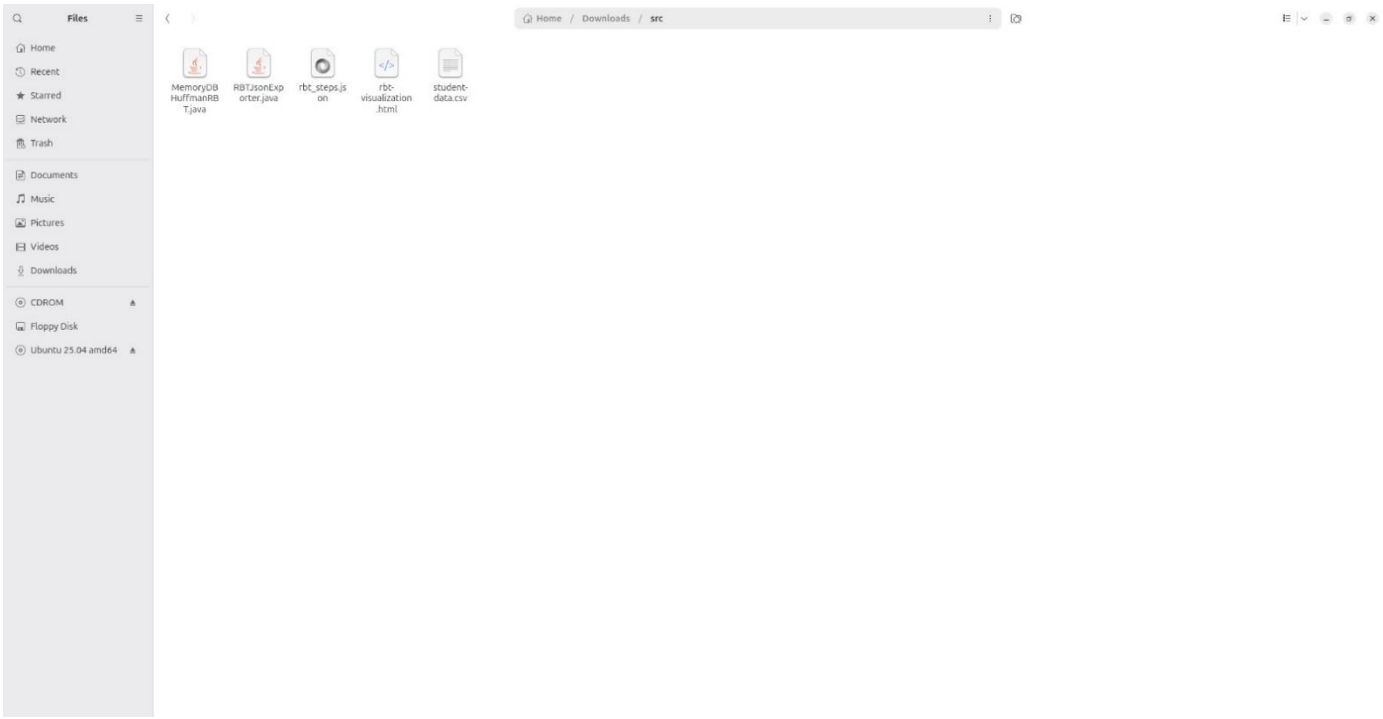

Figure 62 — MemoryDBHuffman.java and student-data.csv files in Ubuntu

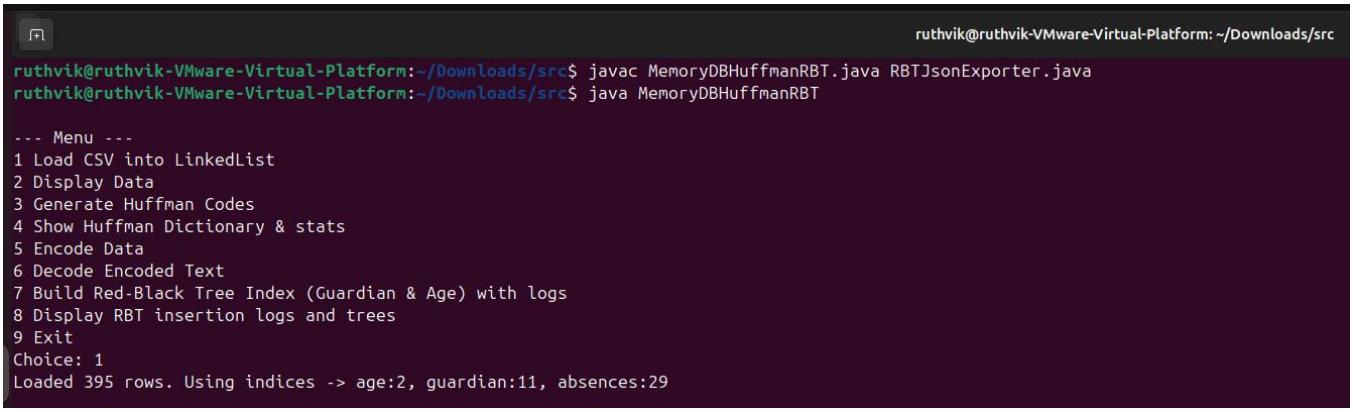

Figure 63 — Java Execution

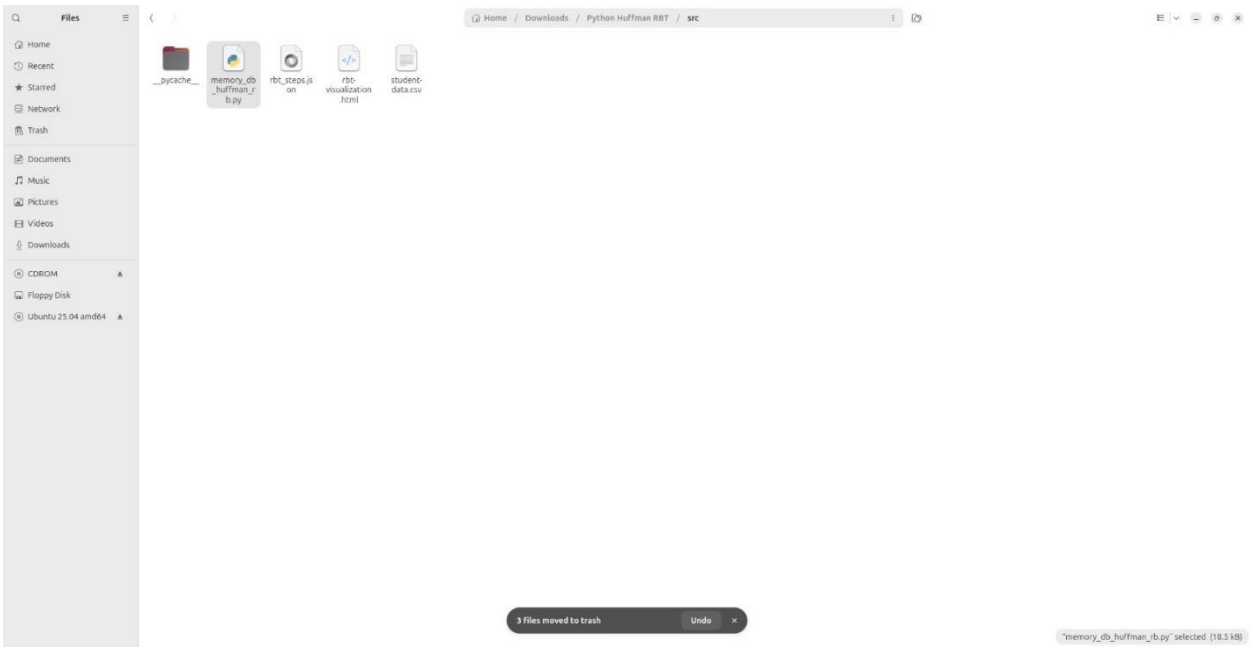

Figure 64 — MemoryDBHuffman.py and student-data.csv files in Ubuntu

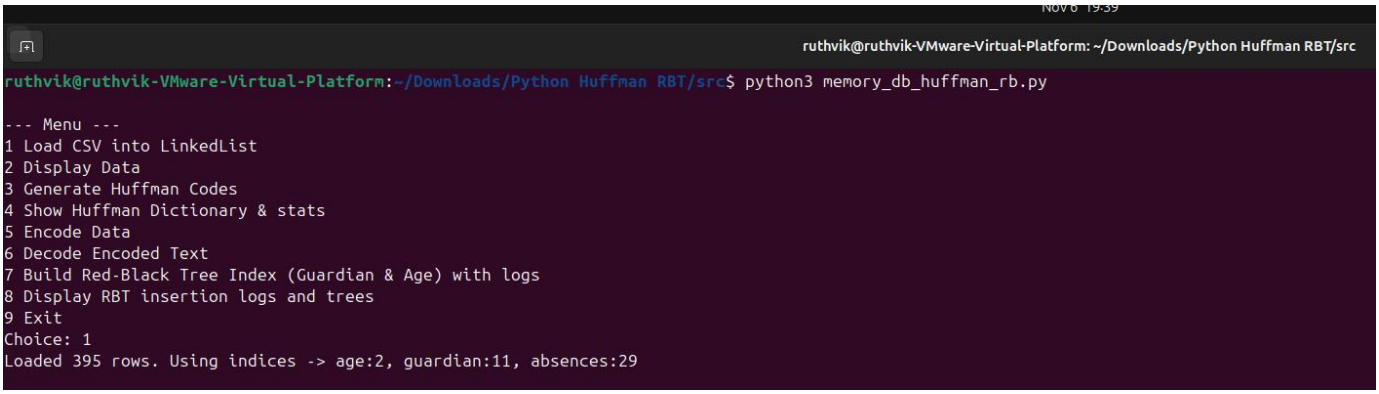

Figure 65 — Python Execution pt. 1

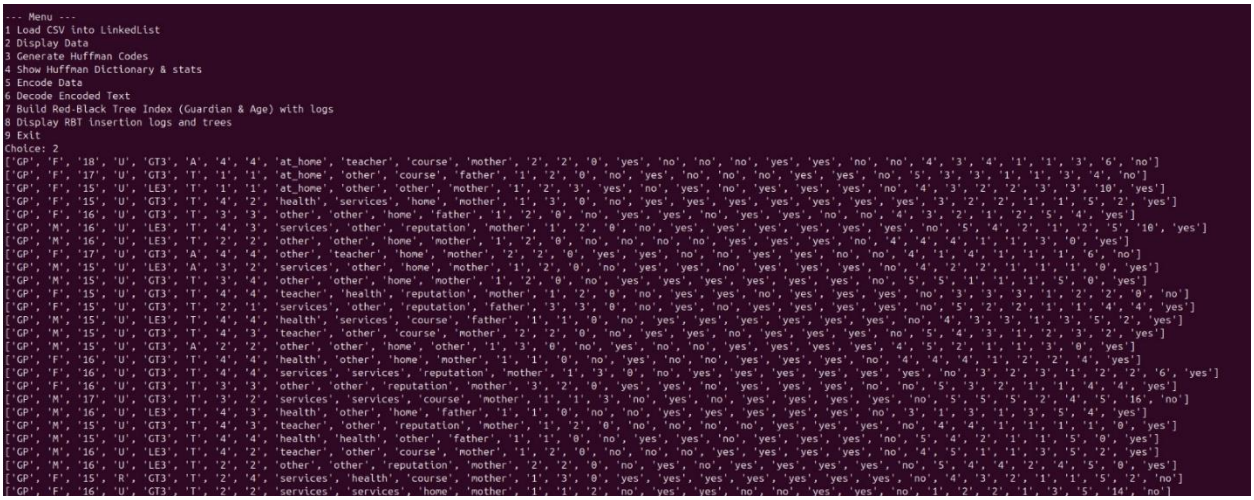

Figure 66 — Python Execution pt. 2

2. Output Results in Ubuntu

When selecting the option to display the loaded student records, both the Python and Java implementations produced identical outputs, confirming consistent parsing and in-memory storage across environments. The screenshot above shows a sample of the dataset printed directly from the linked-list structure after loading the CSV file in the Java terminal. The Python executable produced the same record formatting and field ordering, demonstrating that each row was correctly tokenized, stored, and retrieved from memory.

This output also validates that the underlying linked-list data structure is functioning properly, as records appear sequentially in the same order as in the dataset. The uniform behavior in both languages reinforces the correctness, reproducibility, and completeness of our data ingestion phase prior to applying Huffman encoding and Red-Black Tree indexing.

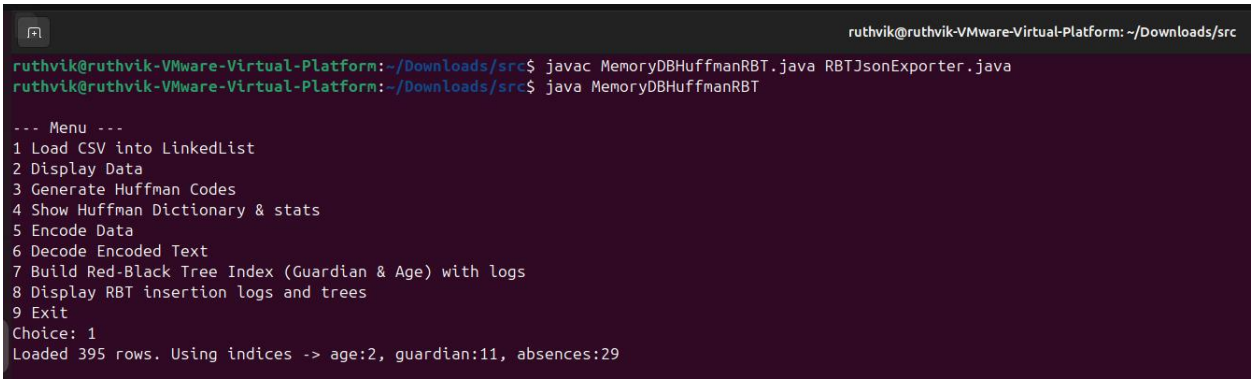

Figure 67 — Loading CSV data into the LinkedList

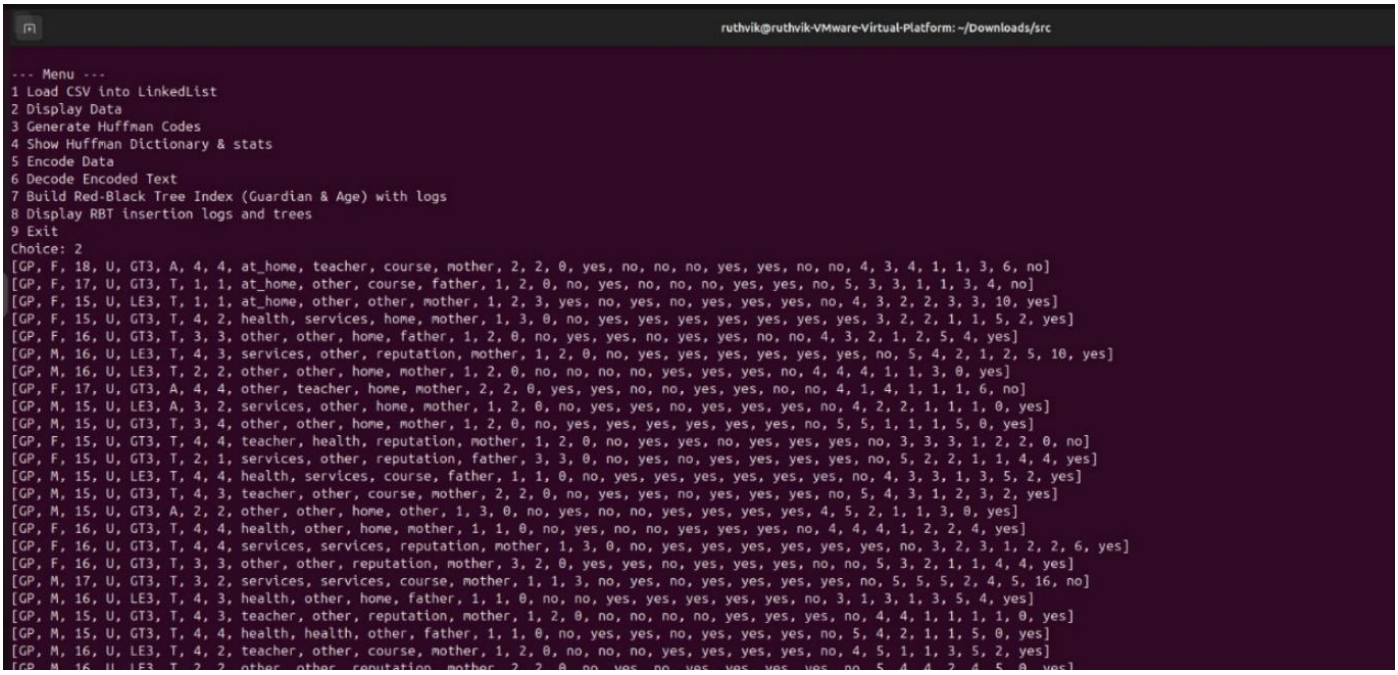

Figure 68 — Displaying the data

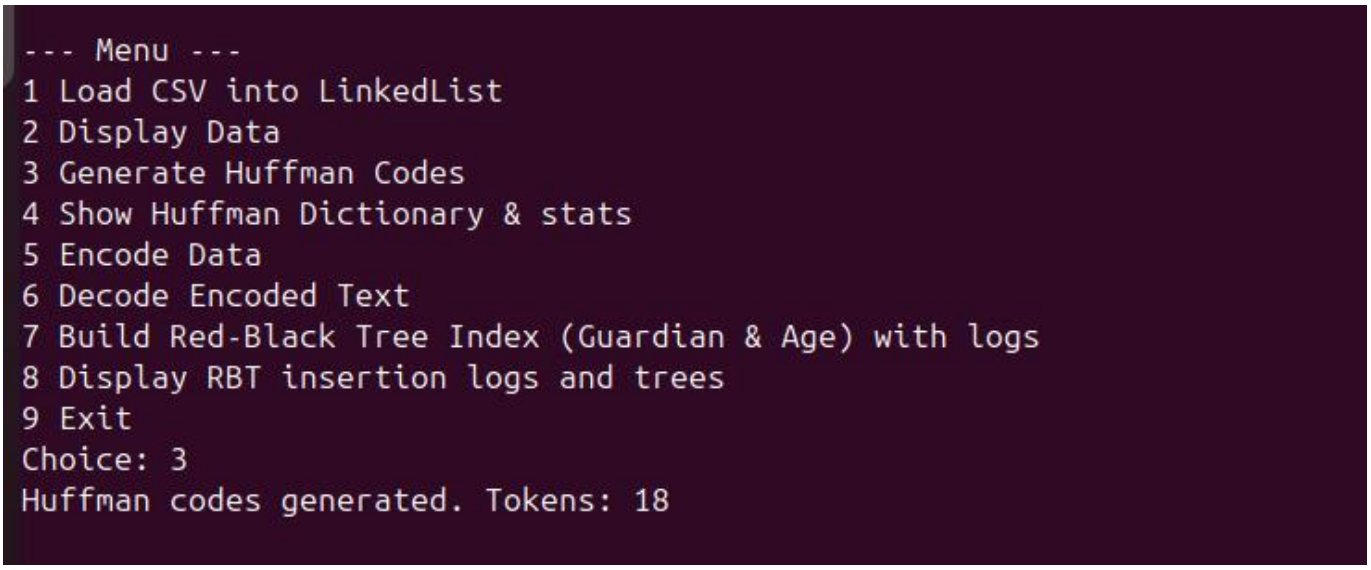

Figure 69 — Generating the Huffman Codes

```
--- Menu ---
1 Load CSV into LinkedList
2 Display Data
3 Generate Huffman Codes
4 Show Huffman Dictionary & stats
5 Encode Data
6 Decode Encoded Text
7 Build Red-Black Tree Index (Guardian & Age) with logs
8 Display RBT insertion logs and trees
9 Exit
Choice: 4
Token               Freq     Code            Bits
------------------------------------------------------------
father18            16       0000            4
other19             18       0001            4
other18             4        001000          6
other17             1        00100100        8
other21             1        00100101        8
other16             2        0010011         7
other20             3        0010100         7
other15             3        0010101         7
mother22            1        00101100        8
father19            2        00101101        8
mother19            4        0010111         7
father16            23       0011            4
mother16            79       01              2
father15            24       1000            4
father17            25       1001            4
mother15            55       101             3
mother18            62       110             3
mother17            72       111             3
Wrote dictionary.csv
Original Size (bits): 25024
Compressed Size (bits): 1297
Compression Ratio: 25024:1297  (decimal: 19.29)
```

Figure 70 — Displaying Huffman Dictionary and Compression Ratio

```
--- Menu ---
1 Load CSV into LinkedList
2 Display Data
3 Generate Huffman Codes
4 Show Huffman Dictionary & stats
5 Encode Data
6 Decode Encoded Text
7 Build Red-Black Tree Index (Guardian & Age) with logs
8 Display RBT insertion logs and trees
9 Exit
Choice: 5
Wrote encoded_text.txt (len=1297 bits)
```

```
--- Menu ---
1 Load CSV into LinkedList
2 Display Data
3 Generate Huffman Codes
4 Show Huffman Dictionary & stats
5 Encode Data
6 Decode Encoded Text
7 Build Red-Black Tree Index (Guardian & Age) with logs
8 Display RBT insertion logs and trees
9 Exit
Choice: 6
Wrote decoded_text.txt (count=395)
```

Figure 71 — Saving encoded data and decoded data in .txt file

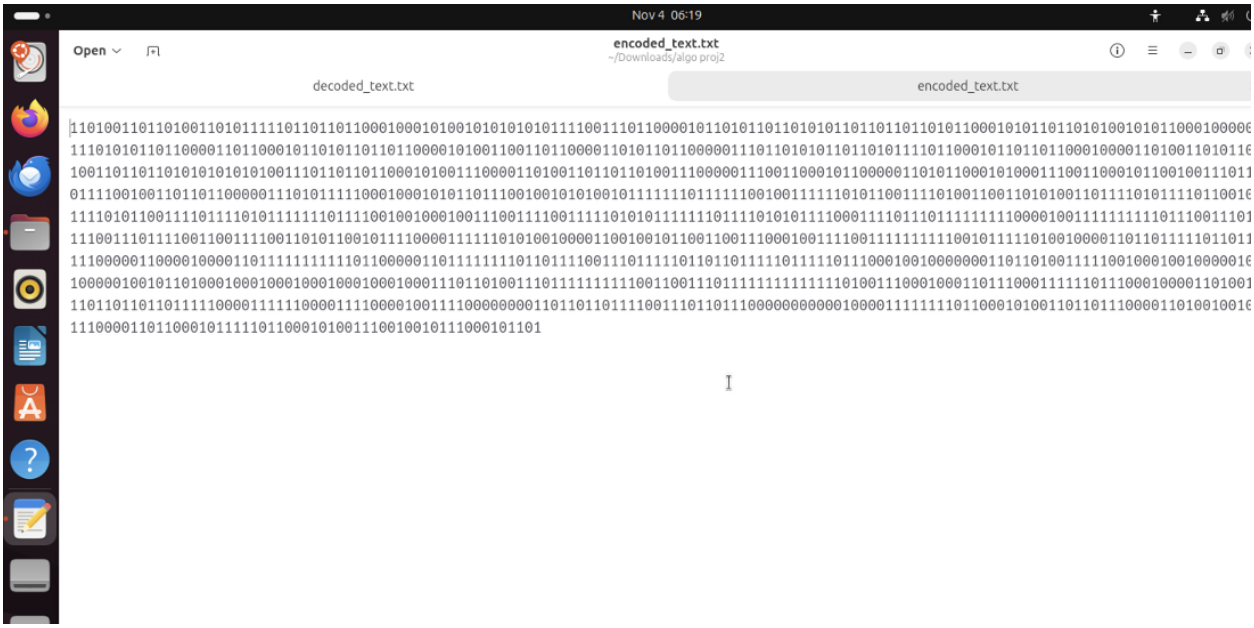

Figure 72 — Saving encoded data and decoded data in .txt file

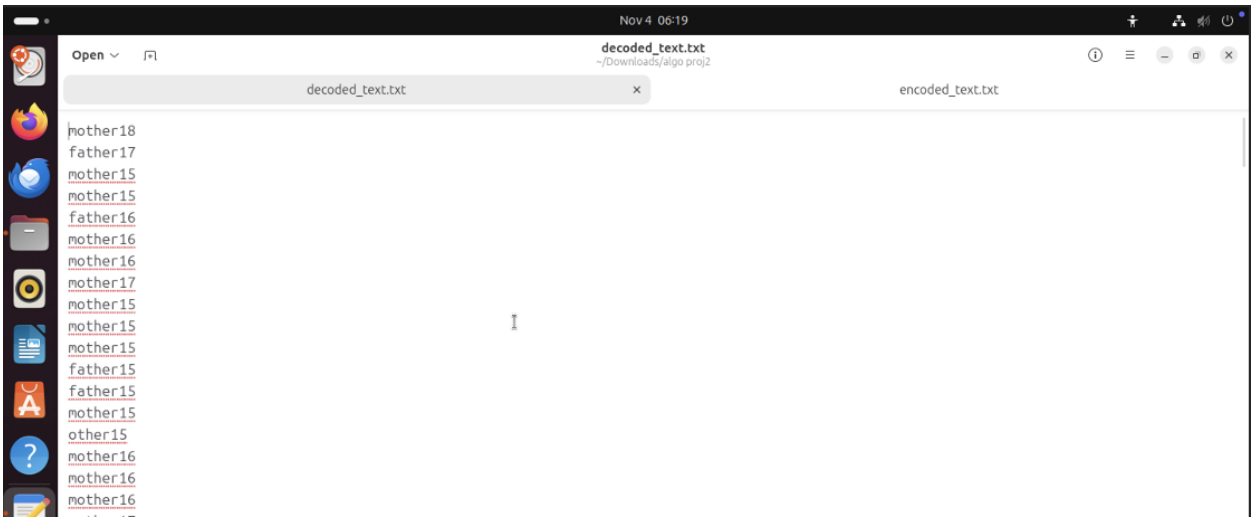

Figure 73 — Decoded_text.txt file that shows decoded data

```
--- Menu ---
1 Load CSV into LinkedList
2 Display Data
3 Generate Huffman Codes
4 Show Huffman Dictionary & stats
5 Encode Data
6 Decode Encoded Text
7 Build Red-Black Tree Index (Guardian & Age) with logs
8 Display RBT insertion logs and trees
9 Exit
Choice: 7
 Built Red-Black Tree using Huffman codes.
```

Figure 74 — Build a Red-Black tree

```
--- Menu ---
1 Load CSV into LinkedList
2 Display Data
3 Generate Huffman Codes
4 Show Huffman Dictionary & stats
5 Encode Data
6 Decode Encoded Text
7 Build Red-Black Tree Index (Guardian & Age) with logs
8 Display RBT insertion logs and trees
9 Exit
Choice: 8
Exported 57 snapshots to rbt_steps.json
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
```

Figure 75 — Display RBT logs and trees on web

3. Analyze Linux performance

Linux Performance Check for Python Execution:

1. Command: time

   Usage : /usr/bin/time -v python3 memory_db_huffman_rb.py

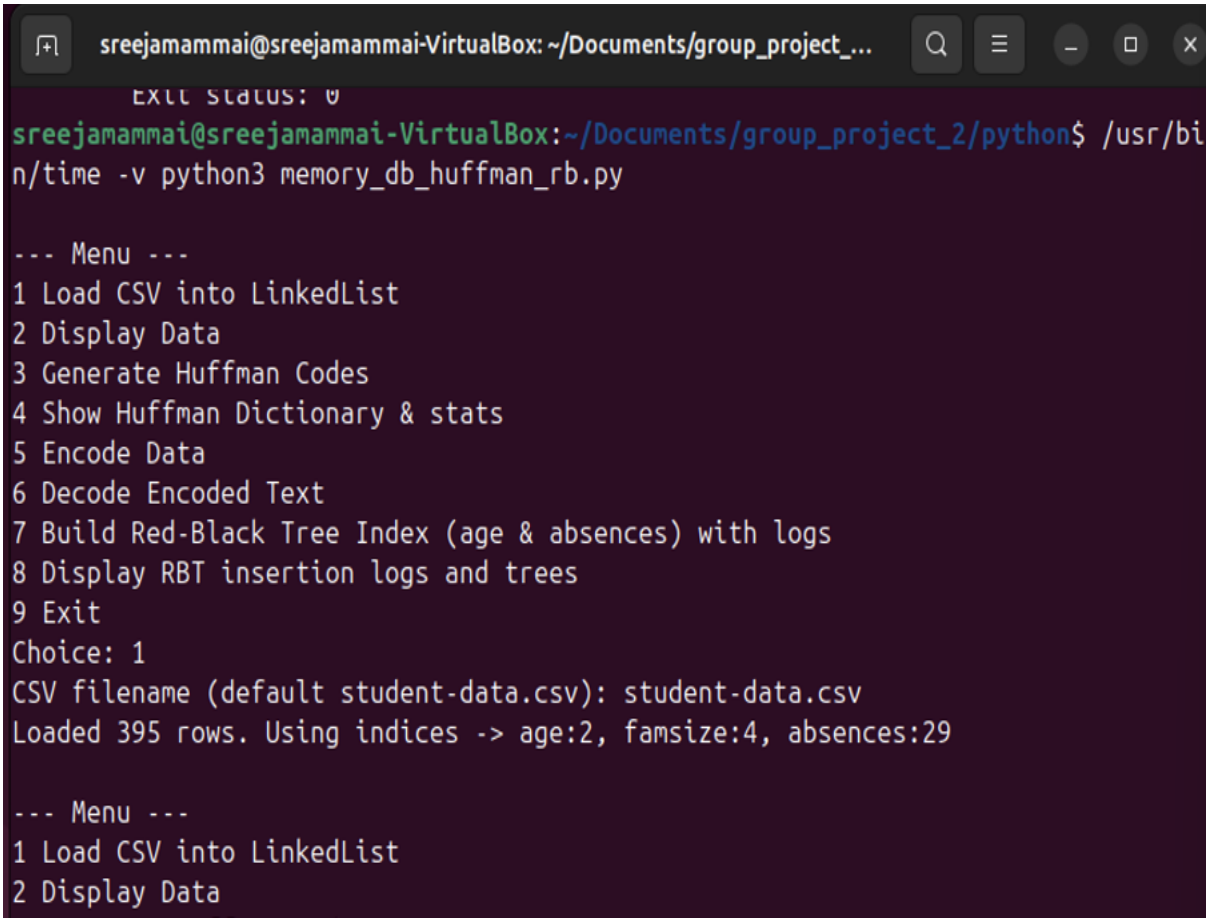

Figure 76 — Time usage command

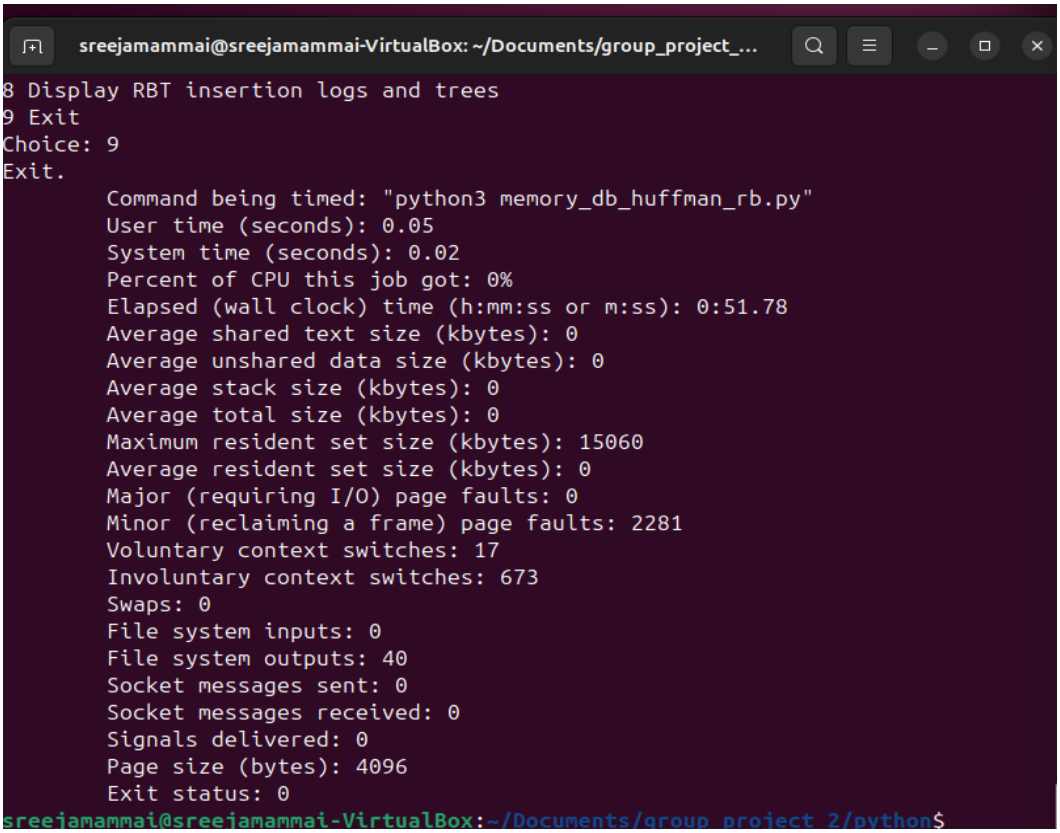

Figure 77 — Time command_output

2. Command : top

Usage: ps aux | Gre python3 memory_db_huffman_rb.pymemory

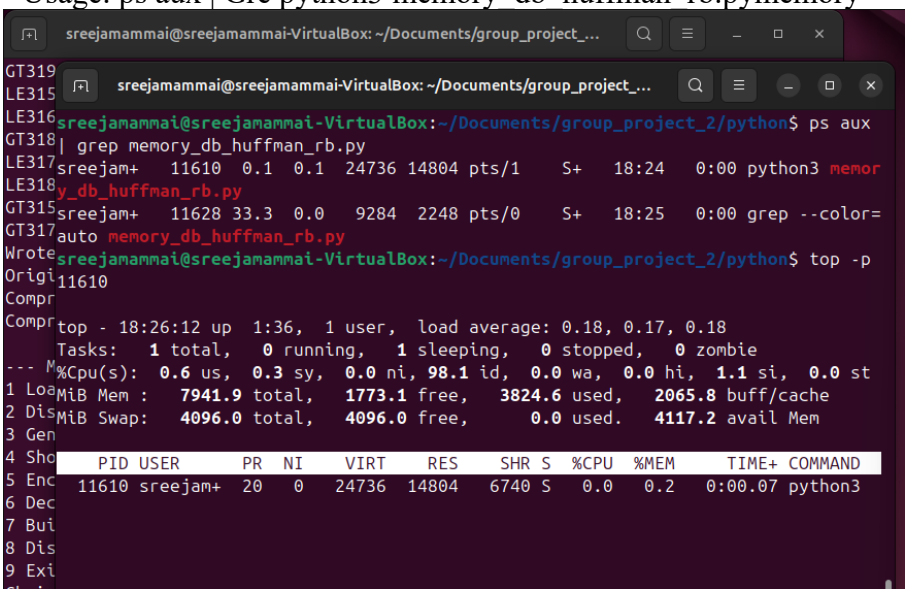

Figure 78 — top command_output

3. Command : pidstat
   Usage: pidstat -p <pid> 1

```
sreejamammai@sreejamammai-VirtualBox:~/Documents/group_project_2/python$ pidstat
-p 11610
Linux 6.14.0-33-generic (sreejamammai-VirtualBox)       11/02/2025      _x86_64_(
4 CPU)

06:29:26 PM   UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Comman
d
06:29:26 PM  1000     11610    0.00    0.00    0.00    0.00    0.00     1  python
3
sreejamammai@sreejamammai-VirtualBox:~/Documents/group_project_2/python$
```

Figure 79 — pidstat_output

Linux Performance check for Java Execution:

4. Command: timeUsage : /usr/bin/time -v java MemoryDBHuffmanRBT

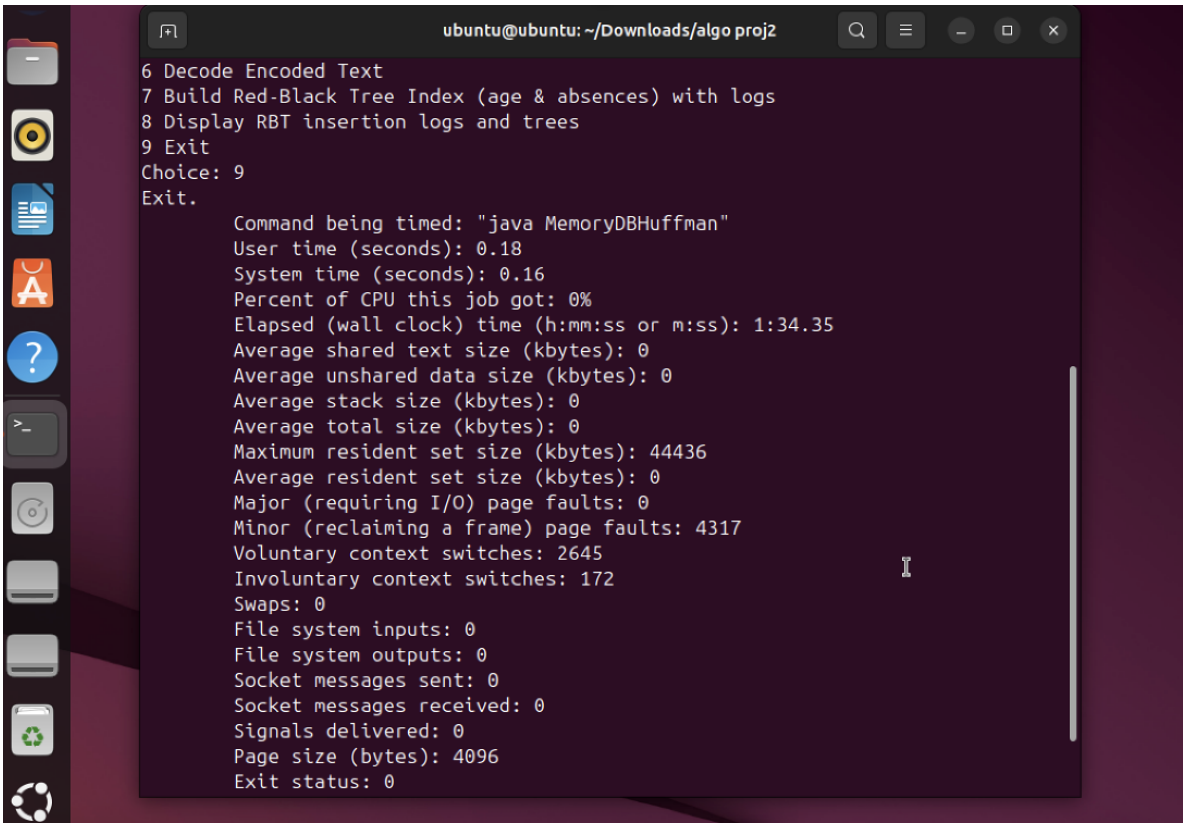

Figure 80 — Time command Output for Java

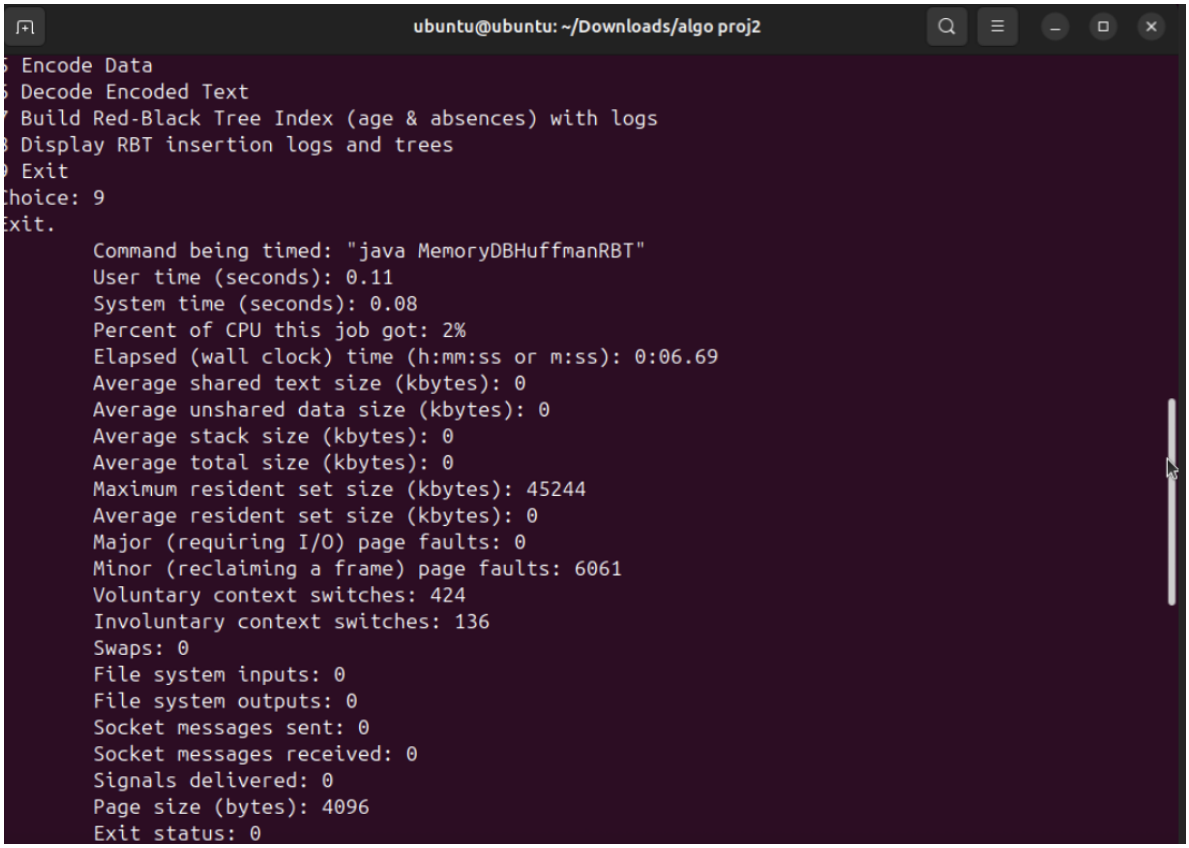

Figure 81 — Time command Output2 for Java

2. Command : ps

Usage : ps aux | grep MemoryDBHuffman - to monitor a specific file.

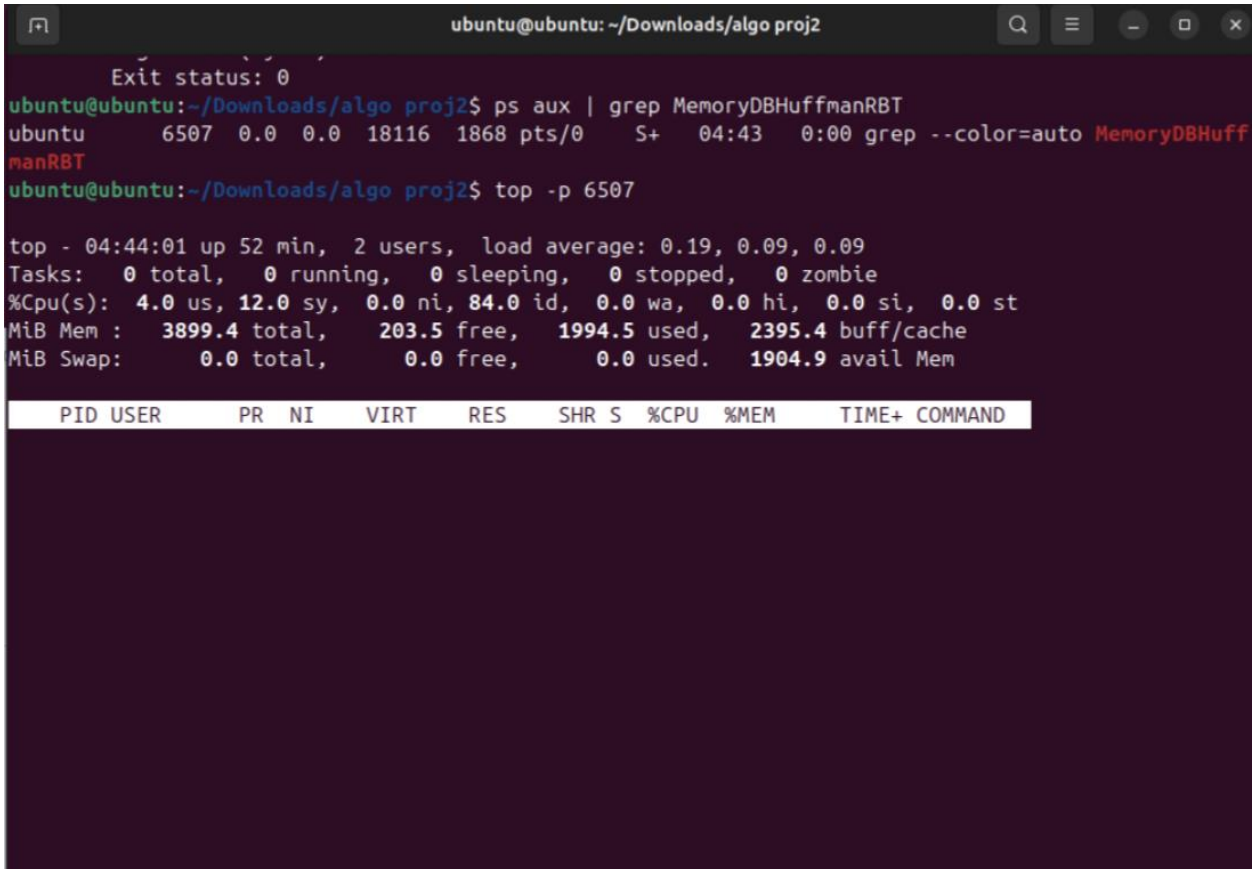

Figure 82 — Output for ps command Java

3. Command : pidstat

```
ubuntu@ubuntu:~/Downloads/algo proj2$ pidstat -p 6507 1
Linux 6.14.0-15-generic (ubuntu)        11/04/25        _aarch64_       (2 CPU)

04:44:33      UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
ubuntu@ubuntu:~/Downloads/algo proj2$
```

Figure 83 — Output for pidstat command Java

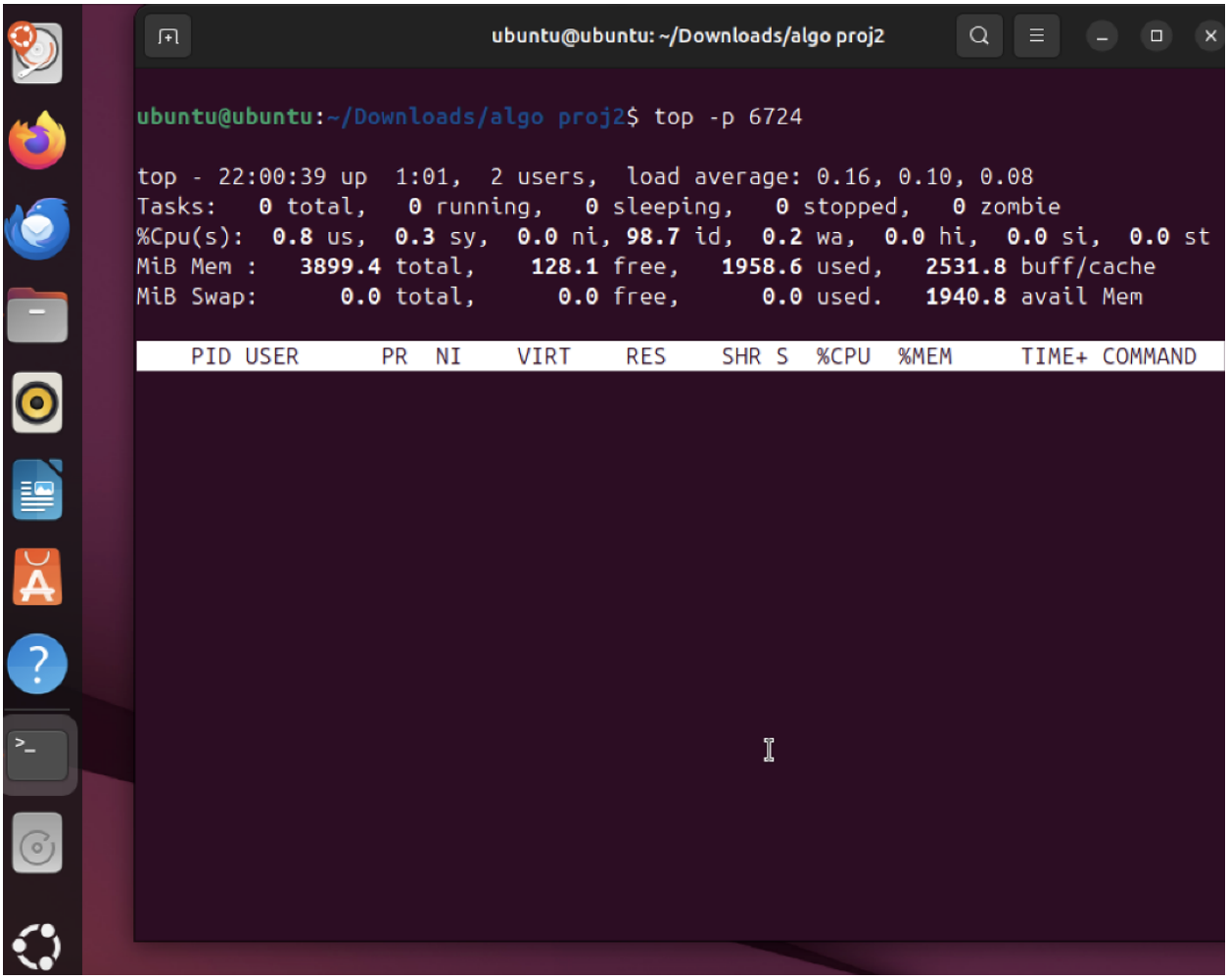

Figure 84 — Output for top command Java

V. Reproducibility and code accessibility

Ensuring that project results can be independently verified is an essential component of credible computational research. To support transparency and repeatability, all source code developed for this project has been made publicly accessible and executable in an online environment. This enables reviewers to reproduce results without requiring local runtime configuration or additional library installations.The Python program includes the in-memory linked-list database and the Huffman-coding index, while the Java program implements the Red-Black Tree indexing. Both files are hosted on **OnlineGDB**, which provides a simple online IDE, so the code runs the same on every system.

Important Note for Step-8 (Tree Visualization)

One part of the project the Red-Black Tree graphical output (Step-8) uses a web window to display the tree. OnlineGDB does not support web windows, so the tree drawing cannot be shown there. To handle this:

1. The RBT visualization was run on a local machine

- **Java Local Execution**

  Use the following commands to compile and run the Java code on your machine:

  "javac MemoryDBHuffmanRBT.java RBTJsonExporter.java"

  "java MemoryDBHuffmanRBT"

- **Python Local Execution**

  For the Python version, you can run by using:

  "python3 main.py"

2. A demo video and screenshots are provided to show how the tree is generated and updated
3. All other features run normally and can be tested in OnlineGDB

Code & Demo Links

- **Python Code:** https://onlinegdb.com/7H5-HB5uo
- **Python Output Demo:** https://drive.google.com/file/d/1PyHvPW_ImNo4yDVtpK82FA-2BUd9Du5o/view?usp=drive_link
- **Java Code:** https://onlinegdb.com/Jf_kbpNK-
- **Java Output Demo (with RBT visuals):** https://drive.google.com/file/d/1jn-ekzgRQZmjfCT2Gx5MWzqbOBr6OxeF/view?usp=sharing

These links include the full code and example outputs so anyone can confirm the results. The local demo clearly shows how the RBT visualization works, while the online links allow direct execution of the logic and indexing steps.

VI. Conclusion

This project successfully demonstrated the development of an in-memory database system enhanced with real indexing techniques to improve data retrieval and efficiency. The core dataset was loaded into a custom linked-list structure, and on top of that foundation we implemented two indexing mechanisms: Huffman coding for compressing and accelerating lookups on a categorical field, and a Red-Black Tree for balanced, ordered access to numerical values. These components allowed us to move beyond simple linear search and explore how real database engines use indexing to optimize query execution.

One of the key accomplishments of this project was building the Huffman encoder to compute symbol frequencies, generate prefix-free binary codes, and ultimately compress and index data entries based on their occurrence patterns. This not only reduced the bit-storage requirements but also allowed faster equality queries by mapping values directly to encoded representations. In parallel, implementing the Red-Black Tree index provided a deeper understanding of self-balancing search trees and offered efficient logarithmic-time search and range filtering capabilities essential for large-scale system performance.

To evaluate our work, we executed the system in an Ubuntu environment and collected real system metrics using native Linux tools, including CPU usage, memory consumption, and performance behavior during indexed versus sequential queries. These measurements demonstrated practical improvements in query time and storage footprint, giving us measurable proof of how indexing structures impact system behavior. The step-by-step visualizations of Huffman tree construction and Red-Black Tree rotations further helped illustrate the internal mechanics behind each algorithm.

Beyond the technical implementation, this project strengthened our understanding of data structures, compression techniques, database indexing concepts, and system-level profiling. It also reinforced important skills in teamwork, planning, and integrating multiple components into a cohesive system. Overall, the experience provided a meaningful foundation for future work with advanced data management systems, query optimization strategies, and performance-driven software design in both academic and professional settings.

Acknowledge

The successful completion of this project reflects the dedication, teamwork, and shared knowledge of our entire group. Each team member contributed meaningfully to planning, task division, implementation, and review, ensuring steady progress and timely execution. Our combined technical strengths and individual efforts created a collaborative environment where ideas were exchanged openly, challenges were addressed constructively, and responsibilities were fulfilled with discipline and commitment.

We also extend our sincere appreciation to our professor for providing valuable guidance, course structure, and continuous support throughout this project. Their instruction helped us deepen our understanding of database systems and apply theoretical concepts to practical implementation. The encouragement and direction given throughout the process played a significant role in shaping the outcome and quality of our work. Together, the cooperation, professionalism, and dedication of the team along with the academic guidance and mentorship from our professor were fundamental to the success of this project.

References

1. GDB Online. (n.d.). *OnlineGDB: Online compiler and debugger*. Retrieved November 04, 2025, from https://www.onlinegdb.com/
2. Kelligaki, K. (n.d.). *Student-data.csv dataset*. Kaggle. Retrieved from https://www.kaggle.com/datasets/kellygakii/student-data-csv
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
4. GeeksforGeeks. (n.d.). *Red-Black Tree — Properties, insertion, and rotations*. Retrieved November 04, 2025, from https://www.geeksforgeeks.org/red-black-tree-data-structure/
5. Huffman, D. A. (1952). *A method for the construction of minimum-redundancy codes*. Proceedings of the IRE, 40(9), 1098–1101.
6. Python Software Foundation. (n.d.). *Python Standard Library Documentation* — csv module, data structures. Retrieved November 04, 2025, from https://docs.python.org/3/library/
7. Oracle Corporation. (n.d.). *Java Platform Standard Edition Documentation* — Collections, data structures, file I/O. Retrieved November 04, 2025, from https://docs.oracle.com/javase/
8. Linux Foundation. (n.d.). *Linux system performance tools: top, htop, time, iostat*. Retrieved November 04, 2025, from https://www.kernel.org/doc/
9. Rosen, K. (2018). *Discrete Mathematics and Its Applications* (8th ed.). McGraw-Hill Education. *(For theoretical basis of trees & coding theory.)*

Appendix

1. Java Program

```
import java.io.*;

import java.lang.reflect.Field;

import java.text.SimpleDateFormat;

import java.util.*;

import java.util.List;


public class MemoryDBHuffmanRBT {

   // Memory DB

   static LinkedList<String[]> data = new LinkedList<>();

   // Column indices

   static int ageCol = -1;

   static int guardianCol = -1;

   static int absencesCol = -1;

   // Huffman structure

   static class HNode {

      String token;

      int freq;

      HNode l, r;

      HNode(String t, int f) {

         token = t;

         freq=f;

      }
```

```
}
static Map<String,String> huffCode = new LinkedHashMap<>();
static Map<String,String> rCode = new HashMap<>();
static Map<String,Integer> freqMap = new LinkedHashMap<>();
static HNode hRoot = null;
static String encodedText = "";
static List<String> decodedTokens = new ArrayList<>();

// ----------------Red-Black Tree Implementation----------------
enum Color { RED, BLACK }
static class RBNode {
   String token;
   String huffCode;
   Color color;
   RBNode left, right, parent;

   RBNode(String token, String huffCode) {
      this.token = token;
      this.huffCode = huffCode;
      this.color = Color.RED;
   }
}
```

```
// A simple Red-Black Tree supporting insert with step logging
static class RedBlackTree {
    RBNode root = null;
    final List<String> steps = new ArrayList<>();
    // Stores visual states of the tree
    private final List<Snapshot> snapshots = new ArrayList<>();
    static class Snapshot {
        final RBNode rootCopy;
        final String message;
        Snapshot(RBNode root, String message) {
            this.rootCopy = cloneTree(root);
            this.message = message;
        }
    }
    private static RBNode cloneTree(RBNode n) {
        if (n == null) return null;
        RBNode copy = new RBNode(n.token, n.huffCode);
        copy.color = n.color;

        // recursively clone children
        copy.left = cloneTree(n.left);
```

```
    copy.right = cloneTree(n.right);

    // re-link parents after recursion
    if (copy.left != null) copy.left.parent = copy;
    if (copy.right != null) copy.right.parent = copy;

    return copy;
}

public List<Snapshot> getSnapshots() {
    return new ArrayList<>(snapshots);
}

// Combined logging + snapshotting
private void logAndSnap(String msg) {
    log(msg); // keep existing log system
    snapshots.add(new Snapshot(root, msg));
}

void log(String s) {
    steps.add(s);
}
void clearLog() {
```

```
    steps.clear();
}
void insert(String token, String huffCode) {
    RBNode z = new RBNode(token, huffCode);
    RBNode y = null;
    RBNode x = root;

    while (x != null) {
        if (z.token.equals(x.token)) return; // skip duplicates
        y = x;
        if (z.token.compareTo(x.token) < 0) x = x.left;
        else x = x.right;
    }

    z.parent = y;
    if (y == null) {
        root = z;
        logAndSnap("Insert " + z.token + " as root");
    } else if (z.token.compareTo(y.token) < 0) {
        y.left = z;
        logAndSnap("Insert " + z.token + " as left child of " + y.token);
    } else {
        y.right = z;
```

```
            logAndSnap("Insert " + z.token + " as right child of " + y.token);
        }

        z.left = null;
        z.right = null;
        z.color = Color.RED;
        insertFixup(z);
    }

    private void insertFixup(RBNode z) {
        while (z.parent != null && z.parent.color == Color.RED) {
            RBNode gp = z.parent.parent;
            if (gp == null) break;

            if (z.parent == gp.left) {
                RBNode y = gp.right;
                if (y != null && y.color == Color.RED) {
                    // Case 1: uncle red
                    z.parent.color = Color.BLACK;
                    y.color = Color.BLACK;
                    gp.color = Color.RED;
                    logAndSnap("Recoloring: parent " + z.parent.token +
                            " and uncle " + y.token +
```

```
                    " to BLACK, grandparent " + gp.token + " to RED");
                z = gp;
            } else {
                if (z == z.parent.right) {
                    // Case 2: left-right
                    z = z.parent;
                    logAndSnap("Left rotate at " + z.token + " (to handle LR case)");
                    leftRotate(z);
                }
                // Case 3: left-left
                logAndSnap("Right rotate at " + gp.token + " and recolor parent and
grandparent");
                z.parent.color = Color.BLACK;
                gp.color = Color.RED;
                rightRotate(gp);
            }
        } else {
            RBNode y = gp.left;
            if (y != null && y.color == Color.RED) {
                z.parent.color = Color.BLACK;
                y.color = Color.BLACK;
                gp.color = Color.RED;
                logAndSnap("Recoloring: parent " + z.parent.token +
```

```
                    " and uncle " + y.token +
                    " to BLACK, grandparent " + gp.token + " to RED");
            z = gp;
        } else {
            if (z == z.parent.left) {
                z = z.parent;
                logAndSnap("Right rotate at " + z.token + " (to handle RL case)");
                rightRotate(z);
            }
            logAndSnap("Left rotate at " + gp.token + " and recolor parent and grandparent");
            z.parent.color = Color.BLACK;
            gp.color = Color.RED;
            leftRotate(gp);
        }
    }

    if (root != null && root.color != Color.BLACK) {
        root.color = Color.BLACK;
        logAndSnap("Set root color to BLACK");
    }
}
private void leftRotate(RBNode x) {
```

```
    RBNode y = x.right;
    if (y == null) return;
    x.right = y.left;
    if (y.left != null) y.left.parent = x;
    y.parent = x.parent;
    if (x.parent == null) root = y;
    else if (x == x.parent.left) x.parent.left = y;
    else x.parent.right = y;
    y.left = x;
    x.parent = y;
    logAndSnap("Performed left-rotate at " + x.token + ", new parent " + y.token);
}
private void rightRotate(RBNode x) {
    RBNode y = x.left;
    if (y == null) return;
    x.left = y.right;
    if (y.right != null) y.right.parent = x;
    y.parent = x.parent;
    if (x.parent == null) root = y;
    else if (x == x.parent.right) x.parent.right = y;
    else x.parent.left = y;
    y.right = x;
    x.parent = y;
```

```
        logAndSnap("Performed right-rotate at " + x.token + ", new parent " + y.token);
    }


}


static RedBlackTree rbgAge = new RedBlackTree();


// ---------------- CSV Loading ----------------
static void loadCsvRecursive(String filename) {
    data.clear();
    ageCol = guardianCol = absencesCol = -1;
    File f = new File(filename);
    if (!f.exists()) {
        System.out.println("File not found: " + filename);
        return;
    }
    try (BufferedReader br = new BufferedReader(new FileReader(f))) {
        String header = br.readLine();
        if (header == null) {
            System.out.println("Empty file");
            return;
        }
        String[] hdr = splitCsvLine(header);
```

```
        for (int i = 0; i < hdr.length; i++) {
            String h = hdr[i].trim().toLowerCase();
            if (h.equals("age")) ageCol = i;
            if (h.equals("guardian")) guardianCol = i;
            if (h.equals("absences")) absencesCol = i;
        }
        // recursive loading function
        loadLinesRecursive(br);
        // final fallback
        if (guardianCol == -1 || ageCol == -1 || absencesCol == -1) {
            if (!data.isEmpty()) {
                int n = data.getFirst().length;
                if (guardianCol == -1) guardianCol = Math.max(0, n-3);
                if (ageCol == -1) ageCol = Math.max(0, n-2);
                if (absencesCol == -1) absencesCol = Math.max(0, n-1);
            } else {
                guardianCol = ageCol = absencesCol = 0;
            }
        }
        System.out.println("Loaded " + data.size() + " rows. Using indices -> age:" + ageCol + ",
guardian:" + guardianCol + ", absences:" + absencesCol);
    } catch (IOException e) {
```

```
        System.err.println("Error reading: " + e.getMessage());
    }
    // reset Huffman and trees
    freqMap.clear();
    huffCode.clear();
    rCode.clear();
    hRoot=null;
    encodedText="";
    decodedTokens.clear();
    rbgAge = new RedBlackTree();
}
private static void loadLinesRecursive(BufferedReader br) throws IOException {
    String line = br.readLine();
    if (line == null) return; // base case
    if (!line.trim().isEmpty()) {
        String[] parts = splitCsvLine(line);
        data.add(parts);

        // fallback detection for age/absences/guardian
        if (ageCol == -1 || absencesCol == -1 || guardianCol == -1) {
            for (int i = 0; i < parts.length; i++) {
                String p = parts[i].trim();
                if (ageCol == -1) {
```

```
                try {
                    int v = Integer.parseInt(p);
                    if (v>=5 && v<=120) ageCol = i;
                }
                catch(Exception ex) {}
            }
            if (absencesCol == -1) {
                try {
                    int v = Integer.parseInt(p);
                    if (v>=0 && v<=500) absencesCol = i;
                }
                catch(Exception ex) {}
            }
        }
    }
    loadLinesRecursive(br);
}
// CSV split helper
static String[] splitCsvLine(String line) {
    List<String> out = new ArrayList<>();
    StringBuilder cur = new StringBuilder();
    boolean inQ = false;
```

```
    for (int i=0; i<line.length(); i++) {
        char c = line.charAt(i);
        if (c == '"') {
            inQ = !inQ;
            continue;
        }
        if (c==',' && !inQ) {
            out.add(cur.toString());
            cur.setLength(0);
        }
        else cur.append(c);
    }
    out.add(cur.toString());
    return out.toArray(new String[0]);
}

// ---------- Huffman ----------
static String makeToken(String[] row) {
    String gua = (guardianCol>=0 && guardianCol<row.length) ? row[guardianCol].trim() : "";
    String age = (ageCol>=0 && ageCol<row.length) ? row[ageCol].trim() : "";
    return gua + age;
}
static void buildFrequency() {
```

```
        freqMap.clear();
        for (String[] row : data) {
            String t = makeToken(row);
            freqMap.put(t, freqMap.getOrDefault(t,0)+1);
        }
    }
    static void buildHuffmanTree() {
        hRoot = null;
        if (freqMap.isEmpty()) return;
        PriorityQueue<HNode> pq = new PriorityQueue<>(Comparator.comparingInt(a->a.freq));
        for (Map.Entry<String,Integer> e: freqMap.entrySet()) pq.add(new HNode(e.getKey(),
e.getValue()));
        if (pq.size()==1) {
            HNode only = pq.poll();
            hRoot = new HNode(null, only.freq);
            hRoot.l = only;
        } else {
            while (pq.size()>1) {
                HNode a = pq.poll();
                HNode b = pq.poll();
                HNode p = new HNode(null, a.freq + b.freq);
                p.l = a;
                p.r = b;
```

```
            pq.add(p);
        }
        hRoot = pq.poll();
    }
}
static void generateCodes() {
    huffCode.clear();
    rCode.clear();
    genRec(hRoot, "");
}
static void genRec(HNode n, String code) {
    if (n==null) return;
    if (n.l==null && n.r==null) {
        String tk = n.token;
        if (code.length()==0) code="0";
        huffCode.put(tk, code);
        rCode.put(code, tk);
        return;
    }
    genRec(n.l, code + "0");
    genRec(n.r, code + "1");
}
static void writeDictionaryCsv(String fname) {
```

```
    try (BufferedWriter bw = new BufferedWriter(new FileWriter(fname))) {
        bw.write("Token,Frequency,Code,Bits");
        bw.newLine();
        for (Map.Entry<String,String> e : huffCode.entrySet()) {
            String tok = e.getKey();
            String code = e.getValue();
            int f = freqMap.getOrDefault(tok,0);
            bw.write("\"" + tok.replace("\"","\"\"") + "\"," + f + "," + code + "," + code.length());
            bw.newLine();
        }
        System.out.println("Wrote dictionary.csv");
    } catch (IOException ex) {
        System.err.println("Write error: " + ex.getMessage());
    }
}
static void encodeData() {
    if (huffCode.isEmpty()) {
        System.out.println("No codes generated");
        return;
    }
    StringBuilder sb = new StringBuilder();
    for (String[] row : data) {
        String t = makeToken(row);
```

```
        String code = huffCode.get(t);
        if (code==null) {
            System.err.println("Missing code for " + t);
        }
        else sb.append(code);
    }
    encodedText = sb.toString();
    try (BufferedWriter bw = new BufferedWriter(new FileWriter("encoded_text.txt"))) {
        bw.write(encodedText);
        System.out.println("Wrote encoded_text.txt (len=" + encodedText.length() + " bits)");
    } catch (IOException e) {
        System.err.println("Encode write error: " + e.getMessage());
    }
}
static void decodeData() {
    decodedTokens.clear();
    if (encodedText==null || encodedText.isEmpty()) {
        File f = new File("encoded_text.txt");
        if (f.exists()) encodedText = readFile(f.getPath());
        else {
            System.out.println("No encoded text present");
            return;
        }
```

```
    }
    StringBuilder tmp = new StringBuilder();
    for (char c : encodedText.toCharArray()) {
        tmp.append(c);
        String s = tmp.toString();
        if (rCode.containsKey(s)) {
            decodedTokens.add(rCode.get(s));
            tmp.setLength(0);
        }
    }
    try (BufferedWriter bw = new BufferedWriter(new FileWriter("decoded_text.txt"))) {
        for (String tok : decodedTokens) {
            bw.write(tok);
            bw.newLine();
        }
        System.out.println("Wrote decoded_text.txt (count=" + decodedTokens.size() + ")");
    } catch (IOException e) {
        System.err.println("Decoded write error: " + e.getMessage());
    }
}
static int computeOriginalBits() {
    int sum = 0;
    for (String[] row : data) sum += makeToken(row).length() * 8;
```

```
    return sum;
}
static int computeCompressedBits() {
    int s = 0;
    for (String[] row : data) {
        String code = huffCode.get(makeToken(row));
        if (code!=null) s += code.length();
    }
    return s;
}
static int gcd(int a,int b) {
    a=Math.abs(a);
    b=Math.abs(b);
    if (b==0) return a==0?1:a;
    return gcd(b,a%b);
}
// ---------- Menu ----------
static void menuLoop() {
    Scanner sc = new Scanner(System.in);
    while (true) {
        System.out.println("\n--- Menu ---");
        System.out.println("1 Load CSV into LinkedList");
        System.out.println("2 Display Data");
```

```
System.out.println("3 Generate Huffman Codes");
System.out.println("4 Show Huffman Dictionary & stats");
System.out.println("5 Encode Data ");
System.out.println("6 Decode Encoded Text");
System.out.println("7 Build Red-Black Tree Index (Guardian & Age) with logs");
System.out.println("8 Display RBT insertion logs and trees");
System.out.println("9 Exit");
System.out.print("Choice: ");
String choice = sc.nextLine().trim();
if (choice.isEmpty()) continue;
try {
    int c = Integer.parseInt(choice);
    switch (c) {
        case 1:
            String path ="student-data.csv";
            loadCsvRecursive(path);
            break;
        case 2:
            showData(50);
            break;
        case 3:
            if (data.isEmpty()) {
                System.out.println("Load CSV first.");
```

```
                    break;
                }
                buildFrequency();
                buildHuffmanTree();
                generateCodes();
                System.out.println("Huffman codes generated. Tokens: " + huffCode.size());
                break;
            case 4:
                if (huffCode.isEmpty()) {
                    System.out.println("Generate codes first.");
                    break;
                }
                printDictionary();
                writeDictionaryCsv("dictionary.csv");
                int orig = computeOriginalBits(), comp = computeCompressedBits();
                System.out.println("Original Size (bits): " + orig);
                System.out.println("Compressed Size (bits): " + comp);
                int g = gcd(orig, comp);
                System.out.println("Compression Ratio: " + (orig/g) + ":" + (comp/g) + "
(decimal: " + String.format("%.2f", (double)orig/comp) + ")");
                break;
            case 5:
                if (data.isEmpty()) {
```

```
            System.out.println("Load CSV first.");
            break;
        }
        encodeData();
        break;

    case 6:
        if (huffCode.isEmpty()) {
            System.out.println("Generate codes first.");
            break;
        }
        decodeData();
        break;
    case 7:
        if (huffCode.isEmpty()) { // wherever you stored Huffman codes
            System.out.println("Run Huffman encoding first.");
            break;
        }
        rbgAge = new RedBlackTree();
        rbgAge.clearLog();

        for (Map.Entry<String, String> entry : huffCode.entrySet()) {
            String token = entry.getKey();       // e.g. "father18"
```

```
                    String code = entry.getValue();      // e.g. "0000"
                    rbgAge.insert(token, code);          // insert both
                }

                System.out.println(" Built Red-Black Tree using Huffman codes.");
                break;



            case 8:
                try {
                    // Export JSON
                    RBTJsonExporter.writeSnapshotsToJson(rbgAge.getSnapshots(), new
File("rbt_steps.json"));
                    System.out.println("Exported " + rbgAge.getSnapshots().size() + " snapshots to
rbt_steps.json");

                    // Start local server if not already running
                    ProcessBuilder pb = new ProcessBuilder("python", "-m", "http.server",
"8080");
                    pb.directory(new File("."));
                    pb.inheritIO();
                    pb.start();
```

```
                    // Give it a second to start
                    Thread.sleep(1500);

                    // Open the localhost URL
                    java.awt.Desktop.getDesktop().browse(new
java.net.URI("http://localhost:8080/rbt-visualization.html"));
                } catch (Exception e) {
                    e.printStackTrace();
                }
                break;


            case 9:
                System.out.println("Exit.");
                sc.close();
                return;
            default:
                System.out.println("Invalid choice.");
        }
      } catch (NumberFormatException nfe) {
        System.out.println("Enter a number.");
      } catch (Exception ex) {
        System.err.println("Error: " + ex.getMessage());
```

```
            ex.printStackTrace(System.err);
        }
    }
}
// Combine Guardian + Age to a single token
static String combinedToken(String[] row) {
    if (guardianCol < 0 || ageCol < 0 || guardianCol >= row.length || ageCol >= row.length)
        return "";
    return row[guardianCol].trim() + row[ageCol].trim();
}
// Convert combined token to numeric key for RBT
static int tokenKey(String[] row) {
    String token = combinedToken(row);
    return Math.abs(token.hashCode() % 100000); // bounded int key
}
// Print both Guardian+Age token and its hash value
static void printTokenAndHash(List<String[]> data) {
    System.out.println("\nGuardian + Age Tokens and Hash Values:");
    System.out.println("----------------------------------------");
    for (String[] row : data) {
        String token = combinedToken(row);
        int hashVal = Math.abs(token.hashCode() % 100000);
        System.out.printf("%-20s -> %d%n", token, hashVal);
```

```
        }
    }
    static void showData(int limit) {
        if (data.isEmpty()) {
            System.out.println("[No data]");
            return;
        }
        int c = 0;
        for (String[] r : data) {
            System.out.println(Arrays.toString(r));
            if (++c >= limit) break;
        }
        System.out.println("Displayed " + Math.min(limit, data.size()) + " of " + data.size());
    }
    static void printDictionary() {
        System.out.printf("%-20s %-8s %-15s %-5s%n", "Token", "Freq", "Code", "Bits");
        System.out.println("------------------------------------------------------------");
        for (Map.Entry<String,String> e : huffCode.entrySet()) {
            String t = e.getKey();
            String code = e.getValue();
            System.out.printf("%-20s %-8d %-15s %-5d%n", t, freqMap.getOrDefault(t,0), code,
code.length());
        }
```

```
    }
    static String readFile(String fname) {
        StringBuilder sb = new StringBuilder();
        try (BufferedReader br = new BufferedReader(new FileReader(fname))) {
            String l;
            while ((l=br.readLine())!=null) sb.append(l.trim());
        } catch (IOException e) {
            System.err.println("Read error: " + e.getMessage());
        }
        return sb.toString();
    }
    // ---------------- main ----------------
    public static void main(String[] args) {
        menuLoop();
    }
}
```

2. Python Program

```
# memory_db_full.py

from __future__ import annotations

import os

import sys

import csv

import json

import time

import webbrowser

import subprocess

from dataclasses import dataclass

from typing import Optional, List, Dict, Any

from heapq import heappush, heappop

from itertools import count

from collections import OrderedDict


# -------------------- Global State --------------------

data: List[List[str]] = []

ageCol = -1

guardianCol = -1

absencesCol = -1

first_seen_index: Dict[str, int] = {}
```

```
codes_insertion_order: list[str] = []



# -------------------- Huffman Structures --------------------


class HNode:
    __slots__ = ("freq", "token", "l", "r")
    def __init__(self, freq, token=None, l=None, r=None):
        self.freq, self.token, self.l, self.r = freq, token, l, r


huffCode: Dict[str, str] = {}
rCode: Dict[str, str] = {}
freqMap: Dict[str, int] = {}
hRoot: Optional[HNode] = None
encodedText: str = ""
decodedTokens: List[str] = []
first_seen_index: Dict[str, int] = {}


# -------------------- Red-Black Tree --------------------
RED = "RED"
BLACK = "BLACK"


@dataclass
```

```
class RBNode:
    token: str
    huffCode: str
    color: str = RED
    left: Optional["RBNode"] = None
    right: Optional["RBNode"] = None
    parent: Optional["RBNode"] = None



class RedBlackTree:
    class Snapshot:
        def __init__(self, root: Optional[RBNode], message: str) -> None:
            self.rootCopy = self.cloneTree(root)
            self.message = message

        @staticmethod
        def cloneTree(n: Optional[RBNode]) -> Optional[RBNode]:
            if n is None:
                return None
            c = RBNode(n.token, n.huffCode, n.color)
            c.left = RedBlackTree.Snapshot.cloneTree(n.left)
            c.right = RedBlackTree.Snapshot.cloneTree(n.right)
            if c.left:
```

```
            c.left.parent = c
        if c.right:
            c.right.parent = c
        return c


def __init__(self) -> None:
    self.root: Optional[RBNode] = None
    self.steps: List[str] = []
    self.snapshots: List[RedBlackTree.Snapshot] = []


def getSnapshots(self) -> List["RedBlackTree.Snapshot"]:
    return list(self.snapshots)


def log(self, s: str) -> None:
    self.steps.append(s)


def logAndSnap(self, msg: str) -> None:
    self.log(msg)
    self.snapshots.append(RedBlackTree.Snapshot(self.root, msg))


def clearLog(self) -> None:
    self.steps.clear()
```

```
def insert(self, token: str, hc: str) -> None:
    z = RBNode(token, hc, RED)
    y: Optional[RBNode] = None
    x: Optional[RBNode] = self.root

    while x is not None:
        if z.token == x.token:
            return
        y = x
        if z.token < x.token:
            x = x.left
        else:
            x = x.right

    z.parent = y
    if y is None:
        self.root = z
        self.logAndSnap(f"Insert {z.token} as root")
    elif z.token < y.token:
        y.left = z
        self.logAndSnap(f"Insert {z.token} as left child of {y.token}")
    else:
        y.right = z
```

```
            self.logAndSnap(f"Insert {z.token} as right child of {y.token}")


        z.left = None
        z.right = None
        z.color = RED
        self.insertFixup(z)


    def leftRotate(self, x: RBNode) -> None:
        y = x.right
        if y is None:
            return
        x.right = y.left
        if y.left:
            y.left.parent = x
        y.parent = x.parent
        if x.parent is None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y
```

```
    self.logAndSnap(f"Performed left-rotate at {x.token}, new parent {y.token}")


def rightRotate(self, x: RBNode) -> None:
    y = x.left
    if y is None:
        return
    x.left = y.right
    if y.right:
        y.right.parent = x
    y.parent = x.parent
    if x.parent is None:
        self.root = y
    elif x == x.parent.right:
        x.parent.right = y
    else:
        x.parent.left = y
    y.right = x
    x.parent = y
    self.logAndSnap(f"Performed right-rotate at {x.token}, new parent {y.token}")


def insertFixup(self, z: RBNode) -> None:
    while z.parent is not None and z.parent.color == RED:
        gp = z.parent.parent
```

```
        if gp is None:
            break


        if z.parent == gp.left:
            y = gp.right
            if y is not None and y.color == RED:
                self.logAndSnap(f"[Before] Recolor case at gp={gp.token}")
                z.parent.color = BLACK
                y.color = BLACK
                gp.color = RED
                self.logAndSnap(
                    f"Recoloring: parent {z.parent.token}, uncle {y.token} -> BLACK; gp {gp.token}
-> RED"
                )
                z = gp
            else:
                if z == z.parent.right:
                    self.logAndSnap(f"[Before] Left rotate at {z.parent.token}")
                    z = z.parent
                    self.leftRotate(z)
                self.logAndSnap(f"[Before] Right rotate at {gp.token}")
                z.parent.color = BLACK
                gp.color = RED
```

```
                self.rightRotate(gp)
            else:
                y = gp.left
                if y is not None and y.color == RED:
                    self.logAndSnap(f"[Before] Recolor case at gp={gp.token}")
                    z.parent.color = BLACK
                    y.color = BLACK
                    gp.color = RED
                    self.logAndSnap(
                        f"Recoloring: parent {z.parent.token}, uncle {y.token} -> BLACK; gp {gp.token}
-> RED"
                    )
                    z = gp
                else:
                    if z == z.parent.left:
                        self.logAndSnap(f"[Before] Right rotate at {z.parent.token}")
                        z = z.parent
                        self.rightRotate(z)
                    self.logAndSnap(f"[Before] Left rotate at {gp.token}")
                    z.parent.color = BLACK
                    gp.color = RED
                    self.leftRotate(gp)
```

```
        if self.root is not None and self.root.color != BLACK:
            self.logAndSnap("[Before] Root recolor")
            self.root.color = BLACK
            self.logAndSnap("Set root color to BLACK")



rbgAge = RedBlackTree()


# -------------------- CSV Loading --------------------
def splitCsvLine_like_java(line: str) -> List[str]:
    out: List[str] = []
    cur = []
    inQ = False
    i = 0
    while i < len(line):
        c = line[i]
        if c == '"':
            inQ = not inQ
            i += 1
            continue
        if c == ',' and not inQ:
            out.append(''.join(cur))
            cur = []
```

```
        else:
            cur.append(c)
        i += 1
    out.append(''.join(cur))
    return out



def loadCsvRecursive(filename: str) -> None:
    global data, ageCol, guardianCol, absencesCol
    data.clear()
    ageCol = guardianCol = absencesCol = -1
    if not os.path.exists(filename):
        print("File not found: " + filename)
        return

    try:
        with open(filename, encoding="utf-8") as f:
            header = f.readline()
            if header == "":
                print("Empty file")
                return
            hdr = splitCsvLine_like_java(header.rstrip("\n"))
```

```
for i, h in enumerate(hdr):
    hlow = h.strip().lower()
    if hlow == "age":
        ageCol = i
    if hlow == "guardian":
        guardianCol = i
    if hlow == "absences":
        absencesCol = i

for line in f:
    line = line.rstrip("\n")
    if not line.strip():
        continue
    parts = splitCsvLine_like_java(line)
    data.append(parts)

if guardianCol == -1 or ageCol == -1 or absencesCol == -1:
    if len(data) > 0:
        n = len(data[0])
        if guardianCol == -1:
            guardianCol = max(0, n - 3)
        if ageCol == -1:
            ageCol = max(0, n - 2)
```

```
                if absencesCol == -1:
                    absencesCol = max(0, n - 1)
            else:
                guardianCol = ageCol = absencesCol = 0

        print(
            f"Loaded {len(data)} rows. Using indices -> age:{ageCol}, guardian:{guardianCol},
absences:{absencesCol}"
        )
    except Exception as e:
        print("Error reading: " + str(e))

    freqMap.clear()
    huffCode.clear()
    rCode.clear()
    global hRoot, encodedText, decodedTokens, rbgAge
    hRoot = None
    encodedText = ""
    decodedTokens.clear()
    rbgAge = RedBlackTree()

# -------------------- Huffman --------------------
def makeToken(row: List[str]) -> str:
```

```
    gua = row[guardianCol].strip() if 0 <= guardianCol < len(row) else ""
    age = row[ageCol].strip() if 0 <= ageCol < len(row) else ""
    return gua + age


def buildFrequency():
    global freqMap, first_seen_index
    freqMap = OrderedDict()
    first_seen_index = {}
    next_id = 0

    for row in data:
        t = makeToken(row)
        if t not in freqMap:
            freqMap[t] = 1
            first_seen_index[t] = next_id
            next_id += 1
        else:
            freqMap[t] += 1



def buildHuffmanTree() -> None:
    global hRoot
    hRoot = None
```

```
if not freqMap:
    return

heap = []
push_counter = 0

# Step 1: insert all leaves in CSV/LinkedHashMap order
for tok, f in freqMap.items():
    heappush(heap, (f, push_counter, HNode(f, tok)))
    push_counter += 1

# Step 2: repeatedly merge two smallest
while len(heap) > 1:
    f1, i1, a = heappop(heap)
    f2, i2, b = heappop(heap)

    # Reproduce Java PQ's tie resolution:
    # - smaller frequency first
    # - if equal, earlier-inserted node first
    if f1 > f2 or (f1 == f2 and i1 > i2):
        a, b = b, a

    parent = HNode(f1 + f2, None, a, b)
```

```
        # Offer the new node *after* previous ones
        heappush(heap, (parent.freq, push_counter, parent))
        push_counter += 1

    _, _, hRoot = heappop(heap)




codes_insertion_order: list[str] = []

def generateCodes() -> None:
    huffCode.clear()
    rCode.clear()
    codes_insertion_order.clear()

    def dfs(n: HNode | None, code: str) -> None:
        if n is None:
            return
        if n.l is None and n.r is None:
            c = code if code else "0"
            huffCode[n.token] = c
            rCode[c] = n.token
```

```
            codes_insertion_order.append(n.token)  # preserve LinkedHashMap put order
            return
        dfs(n.l, code + "0")
        dfs(n.r, code + "1")

    dfs(hRoot, "")




def writeDictionaryCsv(fname: str) -> None:
    try:
        with open(fname, "w", encoding="utf-8", newline="") as f:
            w = csv.writer(f)
            w.writerow(["Token", "Frequency", "Code", "Bits"])
            for tok, code in huffCode.items():
                w.writerow([tok.replace('"', '""'), freqMap.get(tok, 0), code, len(code)])
        print("Wrote dictionary.csv")
    except Exception as ex:
        print("Write error: " + str(ex))



def encodeData() -> None:
    global encodedText
```

```
    if not huffCode:
        print("No codes generated")
        return
    sb = []
    for row in data:
        t = makeToken(row)
        code = huffCode.get(t)
        if code is None:
            print("Missing code for " + t)
        else:
            sb.append(code)
    encodedText = "".join(sb)
    try:
        with open("encoded_text.txt", "w", encoding="utf-8") as f:
            f.write(encodedText)
        print(f"Wrote encoded_text.txt (len={len(encodedText)} bits)")
    except Exception as e:
        print("Encode write error: " + str(e))



def readFile(fname: str) -> str:
    s = []
    try:
```

```
        with open(fname, encoding="utf-8") as f:
            for line in f:
                s.append(line.strip())
    except Exception as e:
        print("Read error: " + str(e))
    return "".join(s)



def decodeData() -> None:
    global decodedTokens, encodedText
    decodedTokens = []
    if not encodedText:
        if os.path.exists("encoded_text.txt"):
            encodedText = readFile("encoded_text.txt")
        else:
            print("No encoded text present")
            return
    cur = ""
    for ch in encodedText:
        cur += ch
        if cur in rCode:
            decodedTokens.append(rCode[cur])
            cur = ""
```

```
    try:
        with open("decoded_text.txt", "w", encoding="utf-8") as f:
            for tok in decodedTokens:
                f.write(tok + "\n")
        print(f"Wrote decoded_text.txt (count={len(decodedTokens)})")
    except Exception as e:
        print("Decoded write error: " + str(e))



def computeOriginalBits() -> int:
    s = 0
    for row in data:
        s += len(makeToken(row)) * 8
    return s



def computeCompressedBits() -> int:
    s = 0
    for row in data:
        code = huffCode.get(makeToken(row))
        if code is not None:
            s += len(code)
    return s
```

```
def gcd(a: int, b: int) -> int:
    a = abs(a)
    b = abs(b)
    if b == 0:
        return 1 if a == 0 else a
    while b:
        a, b = b, a % b
    return a


# ---------- JSON Export ----------
def node_to_json(n: Optional[RBNode]) -> Any:
    if n is None:
        return None
    return {
        "token": n.token,
        "huffCode": n.huffCode,
        "color": "red" if n.color == RED else "black",
        "children": [node_to_json(n.left), node_to_json(n.right)],
    }
```

```
def writeSnapshotsToJson(snapshots: List[RedBlackTree.Snapshot], path: str) -> None:
    obj = {
        "meta": {"generated_by": "memory_db_full.py", "version": "1.0"},
        "steps": [{"message": s.message, "tree": node_to_json(s.rootCopy)} for s in snapshots],
    }
    with open(path, "w", encoding="utf-8") as f:
        json.dump(obj, f, ensure_ascii=False, indent=2)


# -------------------- Menu --------------------
def menuLoop() -> None:
    global rbgAge
    script_dir = os.path.dirname(os.path.abspath(__file__))
    default_csv_path = os.path.join(script_dir, "student-data.csv")


    while True:
        print("\n--- Menu ---")
        print("1 Load CSV into LinkedList")
        print("2 Display Data")
        print("3 Generate Huffman Codes")
        print("4 Show Huffman Dictionary & stats")
        print("5 Encode Data ")
        print("6 Decode Encoded Text")
        print("7 Build Red-Black Tree Index (Guardian & Age) with logs")
```

```
print("8 Display RBT insertion logs and trees")
print("9 Exit")
choice = input("Choice: ").strip()
if not choice:
    continue
try:
    c = int(choice)
    if c == 1:
        loadCsvRecursive(default_csv_path)
    elif c == 2:
        showData(50)
    elif c == 3:
        if not data:
            print("Load CSV first.")
            continue
        buildFrequency()
        buildHuffmanTree()
        generateCodes()
        print("Huffman codes generated. Tokens:", len(huffCode))
    elif c == 4:
        if not huffCode:
            print("Generate codes first.")
            continue
```

```
            printDictionary()
            writeDictionaryCsv("dictionary.csv")
            orig = computeOriginalBits()
            comp = computeCompressedBits()
            print("Original Size (bits):", orig)
            print("Compressed Size (bits):", comp)
            g = gcd(orig, comp)
            ratio = (float(orig) / comp) if comp != 0 else float('inf')
            print(f"Compression Ratio: {orig // g}:{comp // g}  (decimal: {ratio:.2f})")
        elif c == 5:
            if not data:
                print("Load CSV first.")
                continue
            encodeData()
        elif c == 6:
            if not huffCode:
                print("Generate codes first.")
                continue
            decodeData()
        elif c == 7:
            if not huffCode:
                print("Run Huffman encoding first.")
                continue
```

```
        rbgAge = RedBlackTree()
        rbgAge.clearLog()
        for tok in codes_insertion_order:
            rbgAge.insert(tok, huffCode[tok])
        print("Built Red-Black Tree using Huffman codes.")
    elif c == 8:
        try:
            out_json = os.path.join(script_dir, "rbt_steps.json")
            writeSnapshotsToJson(rbgAge.getSnapshots(), out_json)
            print(f"Exported {len(rbgAge.getSnapshots())} snapshots to rbt_steps.json")
            server = subprocess.Popen(
                [sys.executable, "-m", "http.server", "8080"],
                cwd=script_dir,
                stdout=subprocess.DEVNULL,
                stderr=subprocess.DEVNULL,
            )
            time.sleep(1.5)
            webbrowser.open("http://localhost:8080/rbt-visualization.html")
        except Exception as e:
            print(str(e))
    elif c == 9:
        print("Exit.")
        return
```

```
            else:
                print("Invalid choice.")
        except Exception as ex:
            print("Error:", ex)


# -------------------- Display Helpers --------------------


def showData(limit: int) -> None:
    if not data:
        print("[No data]")
        return
    for i, r in enumerate(data[:limit]):
        print(r)
    print(f"Displayed {min(limit, len(data))} of {len(data)} rows")



def printDictionary() -> None:
    if not huffCode:
        print("[No Huffman codes found]")
        return
    print(f"{'Token':<20} {'Freq':<8} {'Code':<15} {'Bits':<5}")
    print("-" * 60)
    for tok, code in huffCode.items():
```

```
        f = freqMap.get(tok, 0)
        print(f"{tok:<20} {f:<8d} {code:<15} {len(code):<5d}")



# -------------------- Main --------------------
if __name__ == "__main__":
    menuLoop()
```