Jaypee Institute of Information Technology, Sector - 62, Noida

B.Tech CSE III Semester

DS Lab PBL Report

Terminal Data Structures & Algorithms Visualizer

Submitted to

Dr. Dhanalekshmi G

Dr. Anupama Padha

Submitted by

Harsh Sharma	2401030232	B5
Karvy Singh	2401030234	B5
Rudra Kumar Singh	2401030237	B5

Letter of Transmittal

Dr. Dhanalekshmi G
Dr. Anupama Padha

Department of Computer Science & IT

Subject: Submission of Project "Terminal Data Structures & Algorithms Visualizer"

Respected Madam/Sir,

We are pleased to submit our project titled *"Terminal Data Structures & Algorithms Visualizer"* as part of the Data Structures Lab Project Based Learning (PBL) component of the B.Tech CSE III Semester curriculum. This report documents the design, implementation, and analysis of a terminal-based interactive visualizer for classical data structures and sorting algorithms implemented in C++.

We have endeavoured to cover the complete lifecycle of the project, including problem formulation, system design, algorithmic choices, implementation details, testing, and observations. Special emphasis has been laid on how different data structures (trees, heaps, B-trees, etc.) and algorithms (Quick Sort, Merge Sort) are modelled, visualized, and interacted with in a text-based terminal environment.

Thank you for your guidance and the opportunity to work on this project, which significantly strengthened our understanding of data structures, algorithms, and low-level terminal programming in Unix.

Sincerely,

Harsh Sharma (2401030232)
Karvy Singh (2401030234)
Rudra Kumar Singh (2401030237)

Date: November 25, 2025

Contents

1 Introduction

Data structures and algorithms form the foundation of computer science. While they are usually studied through diagrams and pseudocode, understanding their *dynamic behaviour*—how nodes move during rotations, how heaps restructure themselves, or how partitioning behaves in quick sort—is much easier with visual aids.

Most available visualizers are graphical and depend on GUI frameworks or web technologies. In this project we explore a different design space: a **pure terminal-based visualizer** running in a Unix environment, implemented entirely in C++ using raw terminal control and ASCII art. The goal is to provide:

- An interactive way to insert, delete, and inspect elements in various tree and heap structures.
- A step-by-step execution view for divide-and-conquer sorting algorithms.
- A modular architecture where new data structure "scenes" can be added easily.

The visualizer demonstrates the runtime behaviour of:

- Binary Search Tree (BST)
- AVL Tree
- Red–Black Tree
- Max Heap and Min Heap
- Binomial Heap and Fibonacci Heap
- B-Tree and B+ Tree
- Merge Sort and Quick Sort

All of these are integrated under a common scene-based framework controlled through a central event loop operating directly on the Unix terminal.

2 Problem Statement and Objectives

2.1 Problem Statement

Students often struggle to connect the abstract definitions of data structures and algorithms with their actual runtime behaviour. Existing tools are usually GUI-based, browser-based, or language-specific, which may not fit well in a Unix lab context focused on systems programming.

Hence, the problem we address is:

To design and implement a modular, extensible, and purely terminal-based data structures and algorithms visualizer in C++, suitable for Unix lab usage, that can demonstrate the real-time behaviour of classical tree structures, heaps, and sorting algorithms.

2.2 Objectives

- Implement a generic scene management framework to host multiple visualizations in a single executable.
- Provide interactive insertion and deletion for tree-based data structures and heaps.
- Provide step-by-step visualization for Quick Sort and Merge Sort with information about indices and partitions.
- Implement a reusable tree rendering engine based on an abstract `TreeScene` template.
- Use only terminal control (no external GUI frameworks) for portability and simplicity.
- Ensure the code is readable, modular, and easily extensible for additional data structures.

3 System Design

3.1 High-Level Architecture

The visualizer is designed around three main layers:

1. **Terminal Abstraction Layer**: Low-level interaction with the Unix terminal using raw mode, ANSI escape codes, and input polling (implemented in `tui.cpp` and `tui.h`).
2. **Scene Management Layer**: Abstract `Scene` interface representing different screens (menu, tree visualizers, sorting visualizers), and a global event loop in `main.cpp` that dispatches events to the current scene.
3. **Data Structure & Visualization Layer**: Concrete scenes such as `AVLImpl` embedded in a `TreeScene<Impl>`, or dedicated scenes like `MergeSortScene` and `QuickSortScene`, which internally maintain algorithm state and render it.

3.2 Class Diagram for Scene Architecture

Figure 1 shows a simplified class diagram of the scene and visualization framework. The base class `Scene` is extended by specific scenes such as `MenuScene`, `TreeScene<Impl>`, and algorithm scenes.

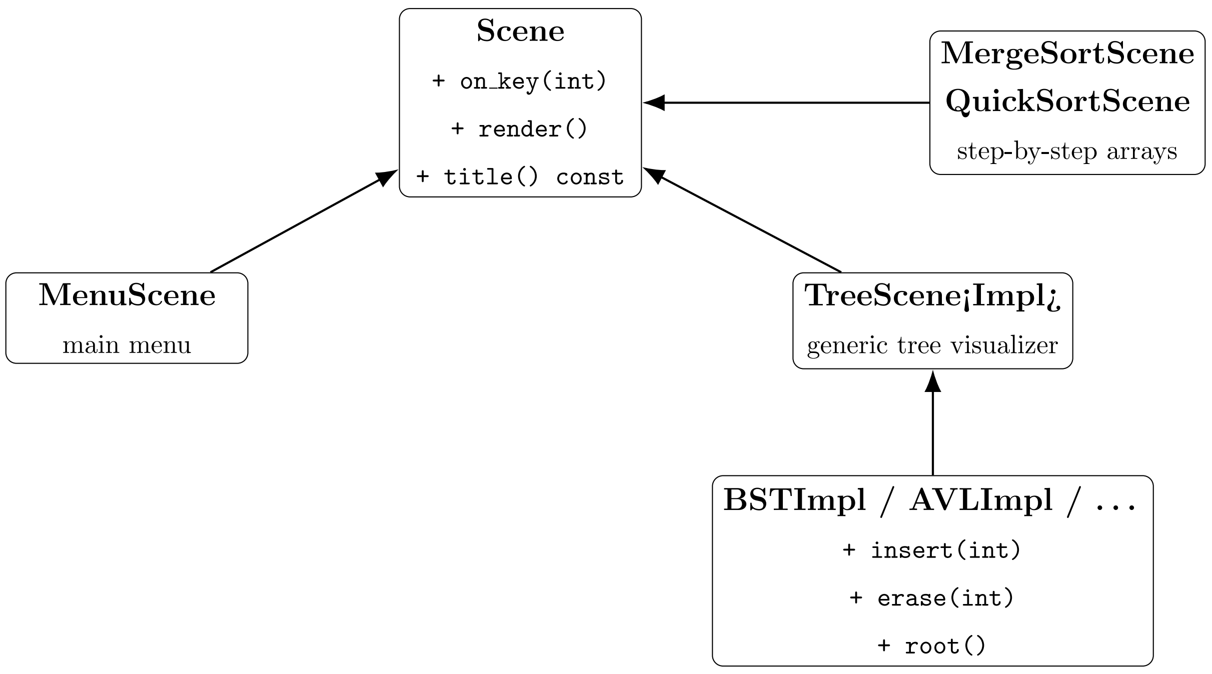


Figure 1: Simplified class diagram of the scene and visualization architecture.

3.3 Event Loop and Control Flow

The central event loop is implemented in `main.cpp`. It:

1. Initializes the terminal (`init()`).
2. Sets the current scene to the menu scene (`make_menu_scene()`).
3. Repeatedly:
   - Renders the current scene.
   - Polls for a key press using `poll_key()`.
   - Dispatches the key to `on_key()` of the current scene.
   - Checks the global `g_should_quit` flag.
4. Deinitializes the terminal (`deinit()`).

Figure 2 gives a flowchart of this control flow.

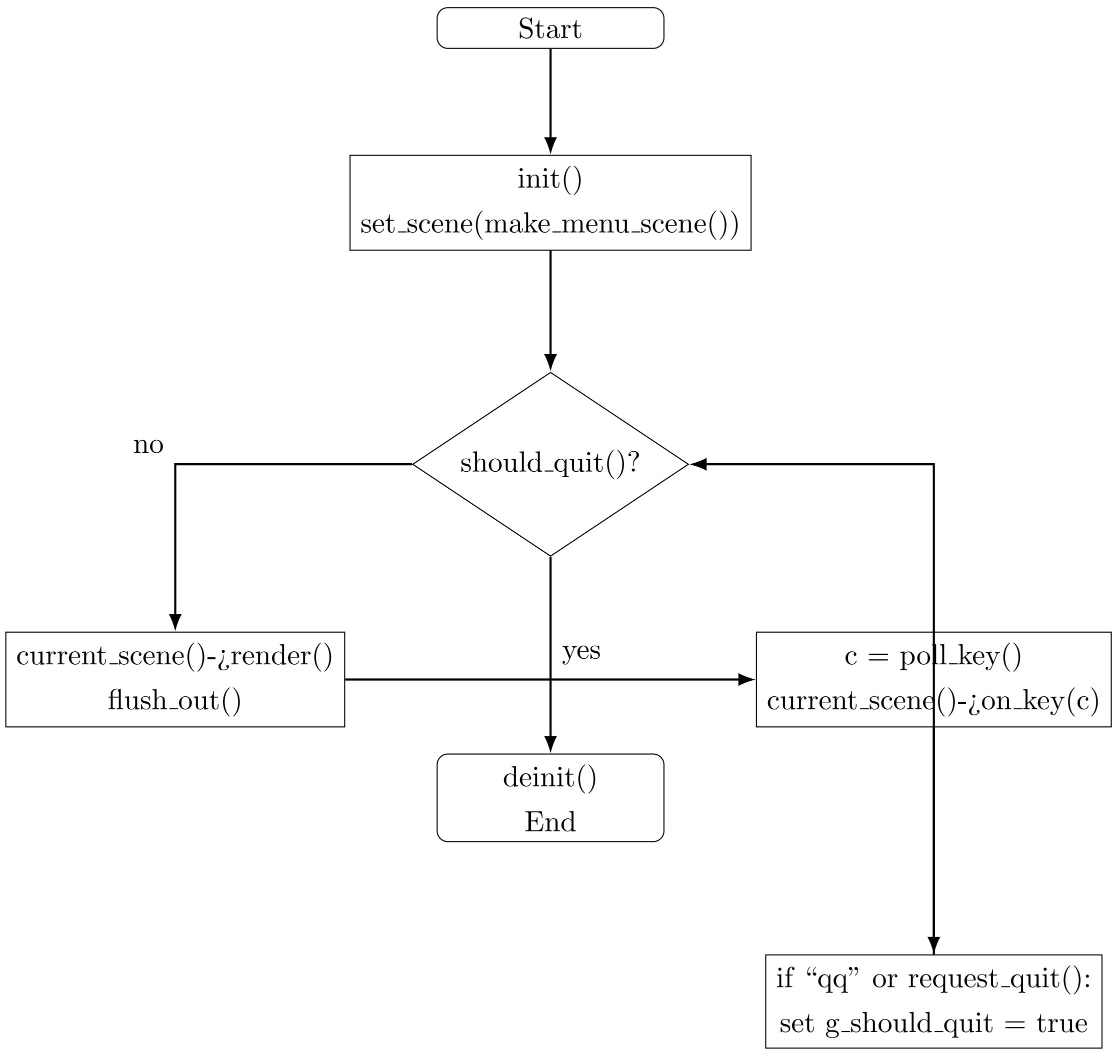

Start
init()
set_scene(make_menu_scene())
should_quit()?
no
yes
current_scene()-¿render()
flush_out()
c = poll_key()
current_scene()-¿on_key(c)
deinit()
End
if "qq" or request_quit():
set g_should_quit = true

Figure 2: Event loop and scene control flow in `main.cpp`.

3.4 Terminal Rendering Layer

The file `tui.cpp` abstracts low-level terminal operations:

- Switching the terminal into raw mode using `termios`.
- Clearing the screen and positioning the cursor with ANSI escape sequences.
- Reading keys (including special keys like arrows) via `select()` and non-canonical input.

On top of this, drawing helpers in `draw.cpp` and `tui.h` provide:

- `frame()`, `hline()`, `vline()` for ASCII box drawing.
- `draw_connector()` for connecting parent and child nodes in tree diagrams.

- `draw_node_label()` to render keys as `[value]`.

These are reused by all tree- and heap-based visualizations.

4 Data Structures and Their Visualization

4.1 Common Tree Visualization Framework (`TreeScene`)

The template `TreeScene<Impl>` in `render.h` is a generic scene for any binary tree-like structure. It expects the `Impl` type to define:

- `using Node = ...;` for the node type.
- `Node *root() const;` returning the root.
- `Node *left(Node*) const;` and `Node *right(Node*) const;` for child access.
- `void insert(int);` and `void erase(int);` to modify the structure.
- `void draw_label(int x, int y, Node *);` for per-node label rendering.
- `vector<int> sample() const;` for sample data.
- `void clear();` to deallocate.

`TreeScene` handles:

- Reading numeric input from the user.
- Converting `[Enter]` into an `insert(int)` call.
- Handling `d` for deletion and `r` for inserting a sample dataset.
- Computing node positions using a recursive layout algorithm so that the left and right subtrees are spaced horizontally.
- Drawing connectors and node labels for the entire tree.

This framework is reused by:

- `BSTImpl` (Binary Search Tree)
- `AVLImpl` (AVL Tree)
- `RBTImpl` (Red–Black Tree)
- `HeapImpl` (Max Heap)
- `MinHeapImpl` (Min Heap)

4.2 Binary Search Tree (BST)

The BST is implemented in `bst/bst.cpp` using a simple `NodeBST` structure with `data`, `left`, and `right` pointers. The operations are:

- **Insertion**: Recursive insertion (`insertAVL` reused as a standard BST insert), respecting the BST property:

$$\text{left subtree} < \text{node} < \text{right subtree}.$$

- **Deletion**: Handled in `deleteNodeBST` with three cases:
    1. Leaf node.
    2. Node with one child.
    3. Node with two children, where the in-order successor (`minValueNodeBST`) is used.

The `BSTImpl` struct plugs this node type into the `TreeScene` template by providing `insert`, `erase`, and `root`. As the user inserts or deletes values via the terminal, the tree is updated and re-rendered, allowing the user to observe structural changes.

4.3 AVL Tree

The AVL tree implementation in `avl/avl.cpp` extends the BST concept by maintaining height-balanced nodes:

- Each node `NodeAVL` stores its height.
- The `balance(NodeAVL*)` function computes the balance factor as the difference between left and right subtree heights.
- After each insertion or deletion, the node heights are updated and `balanceNodeAVL` applies:
    - Left rotation
    - Right rotation
    - Left-Right rotation
    - Right-Left rotation

The visualizer shows the effect of rotations in real-time. Inserting a skewed sequence such as `10, 20, 30` lets the user see the tree self-balancing into a more compact shape.

4.4 Red–Black Tree

The Red–Black Tree is implemented in `rb/rbt.cpp` using `NodeRBT` which stores:

- `data`
- Pointers to `parent`, `left`, `right`
- `color` field ('R' or 'B')

Key operations and invariants:

- New nodes are initially inserted as red.
- The `insertfix` function:
  - Recolors the parent and uncle when both are red.
  - Performs rotations (left or right) around the grandparent to fix double-red violations.
  - Ensures the root is always black.
- The `draw_label` method renders nodes as `[keyR]` or `[keyB]`, with 'R' shown in red using ANSI color codes.

This allows students to not only see the structure of the tree, but also how colors change to maintain the Red–Black properties.

4.5 Max Heap

The max heap implementation in `max_heaps/max_heaps.cpp` uses a `vector<int>` to store heap elements. The key functions are:

- `heapifyup(i)` to bubble an inserted element upwards while the parent is smaller.
- `heapifydown(i)` to restore the heap after extraction or key modification.
- `insert(int val)` and `extractmax()`.

For visualization, indices in the array are mapped to nodes via a secondary `nodes` vector of small `Node` structs that just store the index. The `root()`, `left()`, and `right()` methods adapt this to the tree visualization engine, displaying the heap as a binary tree.

4.6 Min Heap

The min heap (`min_heaps/min_heap.cpp`) mirrors the max heap but maintains the minimum element at the root:

- `getmin()` retrieves `heap[0]`.
- Insertion uses `heapifyup` with the comparison reversed.
- Extraction uses `heapifydown` to restore the min-heap property.

Like the max heap, it reuses the `TreeScene` rendering to show the implicit array as a binary tree.

4.7 Binomial Heap

The binomial heap is implemented in `bin_heaps/bin_heaps.cpp` with a `Node` structure supporting:

- `key`, `degree`
- Pointers to `parent`, `child`, and `sibling`

Key operations:

- `mergeRootLists` to merge two root lists sorted by degree.
- `unionHeaps` to link trees of equal degree and maintain the binomial heap invariants.
- `extractMin` which finds the minimum key in the root list, removes the corresponding tree, reverses its children, and unions them back.

The visualization uses a custom `BinomialHeapScene` which:

- Draws the roots side-by-side.
- Recursively draws children to show the structure of individual binomial trees.

4.8 Fibonacci Heap

The Fibonacci heap implementation in `fib_heaps/fib_heaps.cpp` uses circular doubly linked lists and lazy consolidation:

- Nodes (`FibNode`) maintain `key`, `degree`, `mark`, parent and child pointers, and circular left/right pointers.

- `insert` adds a node to the root list and updates the global minimum pointer.
- `extractMin`:
  - Moves the extracted node's children to the root list.
  - Calls `consolidate()` to merge roots of equal degree by linking trees.

The visualizer shows each root tree as a separate subtree and allows the user to observe how trees of equal degree coalesce after repeated extractions.

4.9 B-Tree

The B-Tree in `btree/btree.cpp` is parameterized by a minimum degree `t`. Each node stores multiple keys and child pointers:

- Insertion follows the standard B-Tree algorithm:
  - Split full children using `split_child`.
  - Insert into non-full nodes (`insert_non_full`).
- The `BTreeScene` stores a list of elements and reconstructs the tree after deletions.

Visualization is multi-key: each node is drawn as a box `[k_1|k_2|...]` and children are connected beneath. This helps understand how B-Trees maintain balance and high fan-out.

4.10 B+ Tree

The B+ Tree in `bptree/bptree.cpp` extends B-Trees by keeping all data in leaf nodes and linking leaves at the bottom level:

- Internal nodes guide the search and store separator keys.
- Leaf nodes store actual keys and maintain a linked list via the `next` pointer.
- `split_child` handles splitting both leaf and internal nodes differently.

The `BPlusTreeScene`:

- Draws internal nodes similarly to B-Trees.
- Records the positions of leaves and then draws horizontal arrows between them to illustrate the leaf-level linked list used for range queries.

5 Sorting Algorithms

5.1 Quick Sort Visualization

`quicksort/quicksort.cpp` implements `QuickSortScene`, which allows the user to:

- Enter integers interactively.
- Press `s` to compute a sequence of steps.
- Use `n` and `p` to move forward and backward through the recorded steps.

The key idea is an instrumented version of quick sort:

- Each time the algorithm scans or swaps elements in the partition procedure, it calls `record_step`.
- A `Step` struct stores:
  - The array snapshot.
  - `low`, `high`, and pivot index.
  - Scan indices `i` and `j`.
  - A textual description (`info`) such as “scan”, “swap”, or “pivot placed”.
- The `render()` method draws the current array and overlays index information.

This helps visualize the partitioning process and how sub-arrays are recursively sorted.

5.2 Merge Sort Visualization

Similarly, `mergesort/mergesort.cpp` defines `MergeSortScene`:

- Users type numbers and press `s` to start the sort.
- The instrumented merge function records the array after each write back to the main array.
- A `Step` struct stores `low`, `mid`, and `high` indices, along with the array snapshot and an `info` string.

The visualizer shows how two halves are merged repeatedly, reinforcing the concept of divide-and-conquer and stable merging.

6 User Interaction and Menu Navigation

6.1 Menu Scene

The `MenuScene` in `menu_scene.cpp` lists all available visualizations:

- BST
- AVL Tree
- Red–Black Tree
- Max Heap
- Min Heap
- Binomial Heap
- Fibonacci Heap
- B-Tree
- B+ Tree
- Merge Sort
- Quick Sort
- Quit

The user navigates using:

- Up/Down arrow keys to move the selection.
- `Enter` or Right arrow to activate the selected scene.
- `q q` sequence to quit from anywhere.

6.2 Key Bindings Inside Visualizers

Most scenes share common key bindings:

- Digit keys (`0--9`): type into the input buffer.
- `Enter`: convert buffer to number and insert into the current structure.
- `d`: delete the value currently in the buffer (where supported).
- `r`: insert a sample dataset.

- `c`: clear the current data structure or array.
- `b` or `Esc`: return to the main menu.
- `q q`: exit the program.

Sorting scenes additionally use:

- `s`: compute the full set of algorithm steps.
- `n/p`: navigate through recorded steps.

Heaps and priority structures may provide extra keys such as `x` for "extract minimum" in the Fibonacci heap scene.

7 Time Complexity Summary

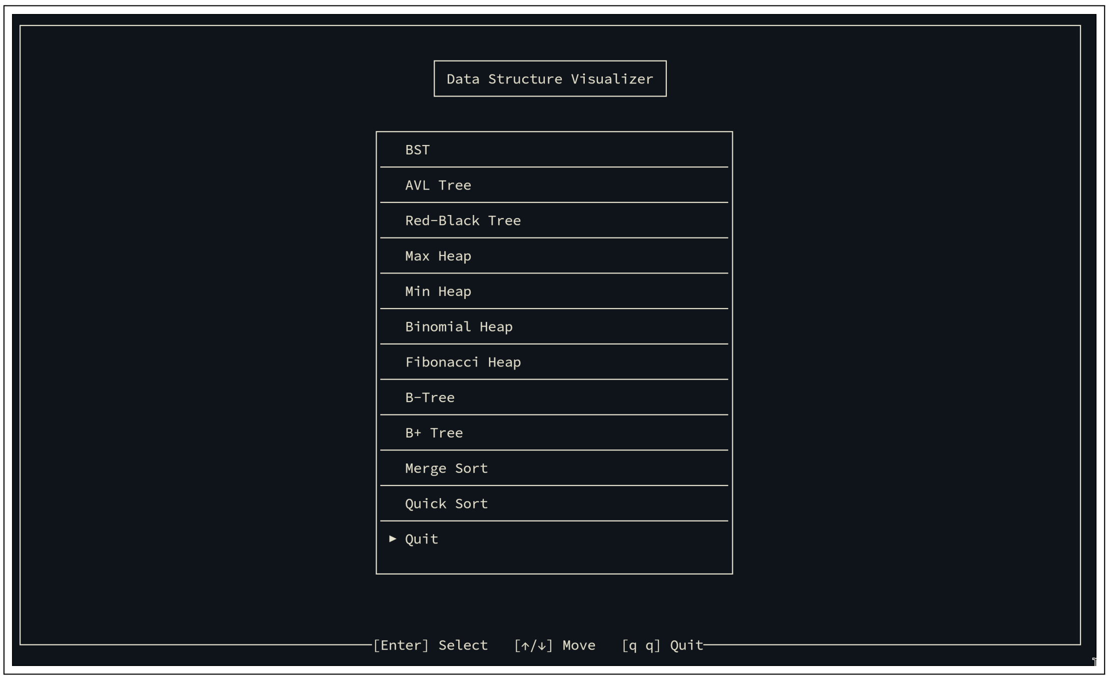
Table 1 summarizes the time complexity of the supported data structures and algorithms.

Structure / Algorithm	**Operation**	**Average**	**Worst Case**
BST	Search/Insert/Delete	$O(\log n)$	$O(n)$
AVL Tree	Search/Insert/Delete	$O(\log n)$	$O(\log n)$
Red–Black Tree	Search/Insert/Delete	$O(\log n)$	$O(\log n)$
Max/Min Heap	Insert / Extract top	$O(\log n)$	$O(\log n)$
Binomial Heap	Insert / Union / Extract min	$O(\log n)$	$O(\log n)$
Fibonacci Heap	Insert	$O(1)$	$O(1)$
	Extract min	$O(\log n)$	$O(\log n)$
B-Tree/B+ Tree	Search/Insert/Delete	$O(\log n)$	$O(\log n)$
Merge Sort	Sorting	$O(n \log n)$	$O(n \log n)$
Quick Sort	Sorting	$O(n \log n)$	$O(n^2)$

Table 1: Time complexity overview of implemented data structures and algorithms.

8 Screenshots

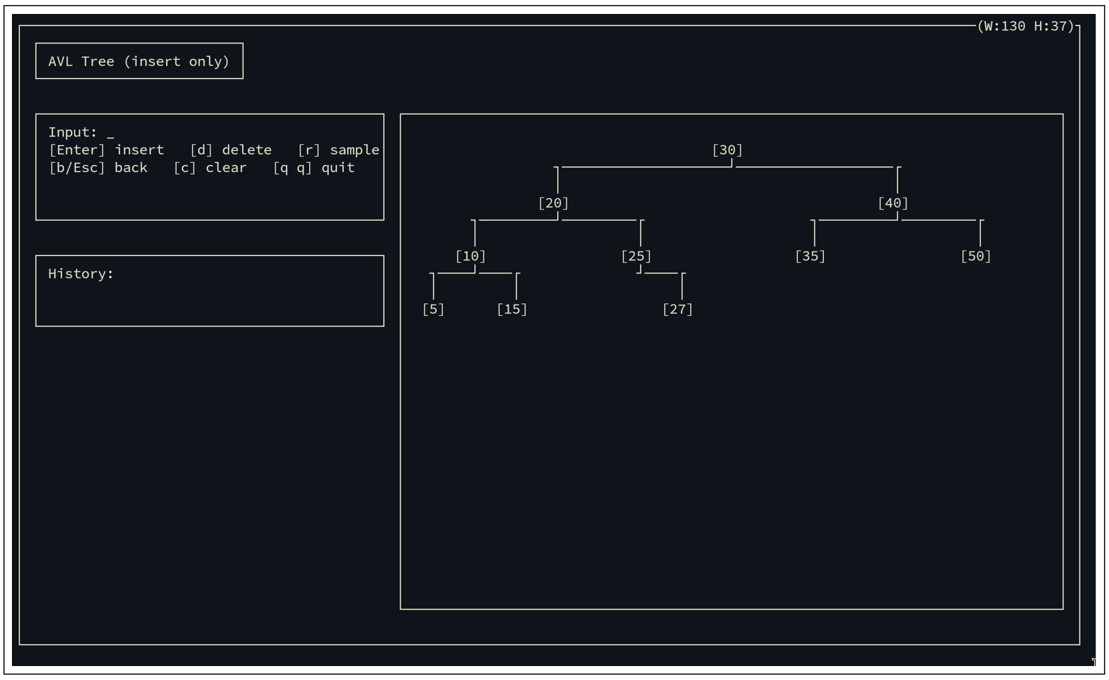
8.1 Main Menu



```
Data Structure Visualizer

  BST
  AVL Tree
  Red-Black Tree
  Max Heap
  Min Heap
  Binomial Heap
  Fibonacci Heap
  B-Tree
  B+ Tree
  Merge Sort
  Quick Sort
► Quit

[Enter] Select   [↑/↓] Move   [q q] Quit
```

Figure 3: Main menu of the Terminal Data Structures & Algorithms Visualizer (placeholder).

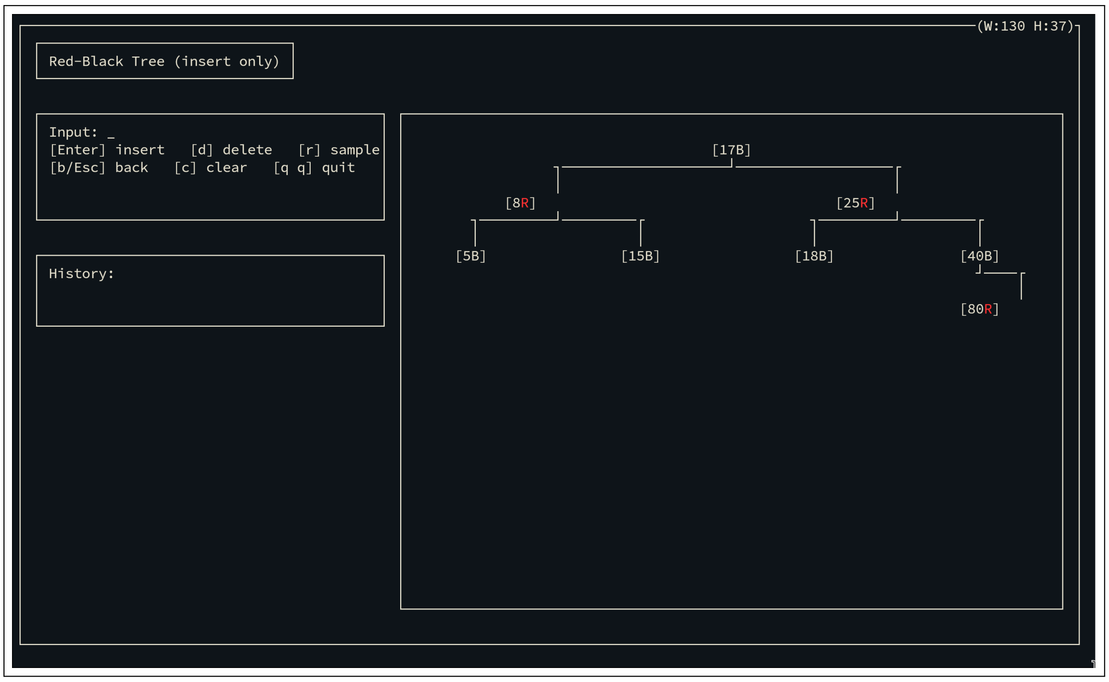
8.2 AVL Tree Visualization



```
(W:130 H:37)
AVL Tree (insert only)

Input: _
[Enter] insert   [d] delete   [r] sample
[b/Esc] back   [c] clear   [q q] quit

History:

                    [30]
         [20]                  [40]
    [10]      [25]        [35]       [50]
 [5]   [15]      [27]
```

Figure 4: AVL tree after multiple insertions demonstrating rotations (placeholder).

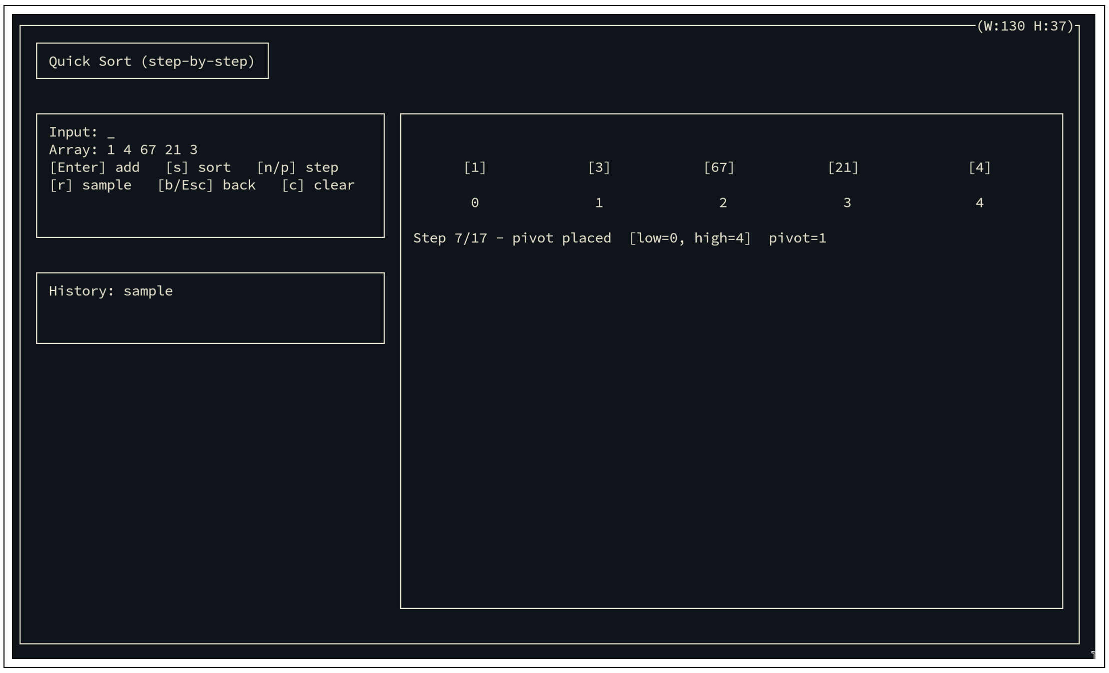
8.3 Red–Black Tree Visualization



```
(W:130 H:37)

Red-Black Tree (insert only)

Input: _
[Enter] insert   [d] delete   [r] sample
[b/Esc] back   [c] clear   [q q] quit

History:

                              [17B]
               ┌────────────────┴────────────────┐
             [8R]                              [25R]
      ┌────────┴────────┐               ┌────────┴────────┐
    [5B]              [15B]           [18B]             [40B]
                                                          └──┐
                                                           [80R]
```

Figure 5: Red–Black tree with node colors indicating balancing (placeholder).

8.4 Quick Sort Step-by-Step Visualization



```
(W:130 H:37)

Quick Sort (step-by-step)

Input: _
Array: 1 4 67 21 3
[Enter] add   [s] sort   [n/p] step
[r] sample   [b/Esc] back   [c] clear

History: sample

  [1]          [3]          [67]          [21]          [4]

   0            1             2             3            4

Step 7/17 - pivot placed  [low=0, high=4]  pivot=1
```

Figure 6: Quick sort scene showing pivot and partition indices (placeholder).

8.5 Future Enhancements

Possible extensions include:

- Adding more algorithms such as Dijkstra's shortest path, BFS/DFS visualizations, and hash tables.
- Supporting loading inputs from files and saving sessions.
- Adding simple profiling information (number of comparisons/rotations/swaps) alongside the visualization.
- Porting the same architecture to a GUI or web-based frontend while reusing the core data structure implementations.

References

- Course Material
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, MIT Press.
- Mark Allen Weiss, *Data Structures and Algorithm Analysis in C++*.