CS163 Project - Data Structure Visualization

Group 6: ARR-CHEESE
Trần Tôn Minh Kỳ, Phan Anh Khoa, Lê Lương Quốc Trung,
Phạm Võ Khang Duy

March 2025

Abstract

The "DSA Visualizer" is an interactive C++ application designed to visually demonstrate four fundamental data structures and algorithms: AVL Tree, Linked List, Hash Table, and Graph. Built using the raylib graphics library and TinyFileDialogs for file input, it aims to enhance understanding of DSA operations through animations and a user-friendly interface. Key features include insertion, deletion, search, undo/redo, random generation, file loading, specialized algorithms (e.g., Kruskal's for graphs), and a toggleable Instant/Step-by-Step mode for controlling animation speed. The project addresses the challenge of abstract DSA comprehension by providing a tangible, visual learning tool for students and enthusiasts.

Contents

1 Group Information

Phan Anh Khoa Developed the AVL tree module. Integrated every member's structures into a unified application. Developed the menu screen and overall project structure. Distinctly delineated the report's details and documentation.

Trần Tôn Minh Kỳ Developed the Linked List module. Assisted in Git setup and overall collaboration. Compiled the report in LaTeX.

Lê Lương Quốc Trung Developed the Hash Table module. Composed a section in Hash Table in the report, including design rationale and implementation overview.

Phạm Võ Khang Duy Developed the Graph module. Implemented Prim's and Kruskal's algorithm, as well as a comprehensive report on the Minimum Spanning Tree problem.

2 Introduction

Every computer science undergraduate always has a course on Data Structures & Algorithms, which is not without reason. They are the backbones of computer science. Fundamentally they are simply data management methods to ensure efficiency and cohesion. There is no one-size-fits-all data structure however; different structure must be used in a flexible manner for different circumstances, and thus the diversity of the data structures in a programmer's arsenal. This diversity is a double-edged sword, it may also pose a challenge to students trying to grasp the nuances behind each data structure.

Our DSA visualizer attempts to solve this problem by bridging the gap between the abstraction of the theories and the practicality of the codes. The DSA visualizer is thus an educational tool allowing students to see the gears behind every structure, the operations behind every algorithm. We aim to make our application as intuitive as possible, yet so simple and exciting that a layman will be eager to learn. Our goals are as follows:

- Develop an interactive visualization for Linked List, Hash Table, AVL Tree, and Graph.
- Implement core operations (insert, delete, search) with animations.
- Provide advance features (undo, redo, file input, line-by-line code demonstration).
- Ensure a user-friendly interface.
- Step-by-step or instant demonstration of the operations.

3 Data Storage

This project uses dynamic memory allocation to store nodes and edges. Typically, a linked list node will be implemented as follows:

```
struct Node {
  int val; // data stored in node
  Node* next; // pointer to the next node, nullptr if none
}
```

For hash tables, we will use linked list to store collisions. To enable the undo/redo features, we shall use `std::stack` to store deep copies of states. This is optimal in terms of efficiency and simplicity; the usage of stacks obeys the LIFO nature of the action of undoing/redoing.

We chose these methods of data storage with rationales that this should boost efficiency and simplicity by storing the data in memory as well as avoiding the complexity of a persistent file-based storage. This also allows real-time visualization and aligns with the current goals and operations.

4 Project Architecture

We outline the overall project's directory as follows:

```
DSA_Visualizer/
|-- main.cpp  <!-- Entry point and menu -->
|-- include/  <!-- Header files -->
|   |-- AVL.h           <!-- AVL Tree definitions -->
|   |-- HashTable.h     <!-- Hash Table definitions -->
|   |-- Graph.h         <!-- Graph definitions -->
|   |-- LinkedList.h    <!-- Linked List definitions -->
|   |-- Common.h        <!-- Shared utilities -->
|-- src/      <!-- Source files -->
|   |-- AVL.cpp         <!-- AVL Tree implementation -->
|   |-- HashTable.cpp   <!-- Hash Table implementation -->
|   |-- Graph.cpp       <!-- Graph implementation -->
|   |-- LinkedList.cpp  <!-- Linked List implementation -->
|   |-- Menu.cpp        <!-- Menu implementation -->
|   |-- Common.cpp      <!-- Shared utilities implementation -->
|-- external/ <!-- External libraries -->
    |-- raylib.h
    |-- tinyfiledialogs.h
```

In order to successfully compile and build these core files, auxiliary build files are also implemented on each member's end.

5 Implementation Details

5.1 Structs and Classes Implementations

5.1.1 Linked List

The involved files are `include/LinkedList.h` and `/src/LinkedList.cpp`. In order to describe a node, we propose the structure `NodeL` as follows:

```
struct NodeL {
  int value;
  NodeL* next;
  float x, y, targetX, targetY;
  NodeL(int val);
};
```

The above implementation includes the node's data and animation coordinates. Altogether, this builds a class `LinkedList` as follows:

```
class LinkedList {
public:
  bool instantMode;
  // Instant execution toggle. All animations are turned off and
  nodes are immediately present at their positions.

  NodeL* insert(NodeL* node, int value, std::vector<NodeL*>& path
  );
  // Recursively adds a node at the end, tracking the path for
  animation (O(n)).

  NodeL* deleteNode(NodeL* node, int value);
  // Traverses to find and remove the node, adjusting next
  pointers (O(n)).

  void search(int value, std::vector<NodeL*>& searchPath);
  // Traverses linearly, adding nodes to searchPath until value
  is found (O(n)).

  NodeL* undo(std::vector<NodeL*>& affectedPath);
  // Pops a state from history or redoStack, deep-copies the
  current state, and restores the previous/next state.

  void updateAnimation(float deltaTime);
  // Interpolates node positions horizontally (targetX).

  void draw(const std::vector<NodeL*>& highlightPath);
  // Draws nodes and links linearly, pulsing highlighted nodes.
}
```

5.1.2 Hash Table

The involved files are `include/HashTable.h` and `/src/HashTable.cpp`. In order to describe a node, we propose the structure `NodeH` as follows:

```
struct NodeH {
  int value;
  NodeH* next;
};
```

This implementation handles the data of the table itself. In order to efficiently handle the table's interface and data, we propose the implementation of two classes `HashTable` and `UI`. Their implementations are as follows:

```
class HashTable {
public:
  void insert(int value);
```

```
    // Hashes value (value % 19), adds to the chain if unique (O(1)
     on average, worst case O(n)).

    bool remove(int value);
    // Searches the chain at the hashed index and removes the node
    (O(n)).

    bool find(int value);
    // Searches the chain for value (O(n)).
  }

  class UI {
  public:
    bool instantMode;
    // Instant execution toggle. All animations are turned off and
    nodes are immediately present at their positions.

    void update();
    // Manages animation states (INDEX, EXISTING_NODES, NEW_NODE)
    with a timer, processes user input, and queues file-loaded
    values.

    void draw();
    // Renders table slots, nodes, buttons, and input box (linear
    rendering O(n)).
  }
```

5.1.3 Graph

The involved files are `include/Graph.h` and `/src/Graph.cpp`. In order to describe an edge and a vertex, we propose the structure `Vertex` and `Edge` as follows:

```
  struct Vertex {
    Vector2 position;
    int id;
    bool selected;
    float scale;
    float alpha;
    Vector2 targetPos;
  };

  struct Edge {
    int from;
    int to;
    int weight;
    bool highlighted;
    bool blurred;
  };
```

As such, the implementation of the class `GraphApp` is as follows:

```
  class GraphApp {
  public:
    bool instantMode;
    // Instant execution toggle. All animations are turned off and
    nodes are immediately present at their positions.

```

```
    void Update();
    // Applies a force-directed layout (repulsive forces between
    vertices, UI avoidance) to position nodes dynamically (O(n^2)).

    void Draw(bool& shouldReturn);
    // Renders vertices, edges, buttons, and input box with
    highlights (O(K + E)).

    void RunKruskalStepByStep();
    // Sorts edges by weight, applies Union-Find to build MST,
    highlights steps over time (O(ElogE)).

    void ProcessInsert();
    // Parses input text to add vertices and edges, ensuring
    uniqueness (O(K + E)).

    void ProcessDelete();
    // Parses input text to remove vertices and edges, ensuring
    uniqueness (O(K + E)).
}
```

5.1.4 AVL Tree

The involved files are `include/AVL.h` and `/src/AVL.cpp`. In order to describe a node, we implement the following struct:

```
struct Node {
    int key;
    int height;
    Node* left;
    Node* right;
    float x, y;
    float targetX, targetY;
    bool isDying;
    Node(int value);
};
```

As usual, the class implementation is as follows:

```
class AVLTree {
public:
    bool instantMode;
    // Instant execution toggle. All animations are turned off and
    nodes are immediately present at their positions.

    void insert(int key);
    // Recursively inserts a node with key, updating the path for
    animation. Balances the tree using AVL rotations if the balance
     factor (getHeight(left) - getHeight(right)) exceeds 1 or -1.

    void deleteNode(int key);
    // Recursively finds and removes the node, replacing it with
    the minimum of the right subtree if it has two children, then
    rebalances.

    void search(int key, std::vector<Node*>& searchPath);
```

```
    // Traverses the tree, adding nodes to searchPath until key is
    found or traversal ends.

    Node* undo(std::vector<Node*>& affectedPath);
    // Pops a state from history, deep-copies the current state,
    and restores the previous state.

    Node* redo(std::vector<Node*>& affectedPath);
    // Pops a state from redoStack, deep-copies the current state,
    and restores the next state.

    void updateAnimation(float deltaTime);
    // Uses linear interpolation (x += dx * deltaTime * 2.0f) to
    move nodes from current (x, y) to target positions (targetX,
    targetY).

    void draw(const std::vector<Node*>& highlightPath);
    // Recursively renders nodes (circles) and edges (lines) with
    raylib, highlighting nodes in highlightPath.
};
```

5.2 Usage

5.2.1 Linked List

The usage flow of the Linked List data structure is as follows:

1. User enters "15" and clicks "Search".
2. `runLinkedList()` calls `list.search(15, searchPath)`.
3. `LinkedList::search` builds `searchPath` (e.g. `[5,10,15]`).
4. `updateAnimation` adjusts position (horizontal at $y = 500$).
5. `draw` iterates `searchPath`, pulsing each node every 0.5 seconds (`searchTimer`).

This works in tandem with the background processes, which we called collaboration:

1. `NodeL` stores values and links.
2. `search` performs linear traversal.
3. raylib's timing (`GetFrameTime`) and drawing (`DrawCircle`) creates the animation effect.

5.2.2 Hash Table

The usage flow of the Hash Table data structure is as follows:

1. User clicks "Load", selects a file via `tinyfd_openFileDialog`.
2. `UI::update` reads integers into `insertQueue` (or inserts instantly with `instantMode`).

3. `update` processes `insertQueue`, calling `HashTable::insert` per value.
4. Animation states (`INDEX`, `EXISTING_NODES`, `NEW_NODE`) highlight slots and nodes.
5. `draw` renders the updated table.

This works in tandem with collaboration:

1. `NodeH` and `HashTable` manage data with chaining.
2. `UI` orchestrates animation and input handling.
3. `TinyFileDialogs` and raylib enable file input and visualization.

5.2.3 Graph

The usage flow of the Graph data structure is as follows:

1. User clicks "Kruskal's Algorithm".
2. `runGraph()` calls `app.Update()` and `app.Draw(shouldReturn)`.
3. `GraphApp::RunKruskalStepByStep` sorts edges by weights using `std::sort`, initializes parent for `Union-Find`, iterates edges every one second (`kruskalTimer`) while calling `Union-Find` to build MST and highlights accepted edges (`highlighted = true`).
4. `Draw` renders vertices, edges and highlights, showing MST progress.

This works in tandem with collaboration:

1. `Vertex` and `Edge` store graph data.
2. `RunKruskalStepByStep` implements Kruskal's with `Union-Find`.
3. raylib's `DrawLineV` and `DrawCircleV` visualize the process.

5.2.4 AVL Tree

The usage flow of the AVL Tree data structure is as follows:

1. User enters "42" in the input box and clicks "Insert"
2. `runAVL()` detects the click (`isButtonClicked(insertButton)`) and calls `tree.insert(42)`.
3. `AVLTree::insert` traverses the tree, inserts the node, and balances if needed (e.g. `rightRotate`).
4. `calculatePositions` assigns `targetX` and `targetY` (e.g. $x = 700 - \texttt{offset}, y = 50 + \texttt{depth} \times 100$).
5. `updateAnimation` moves the node to its target over frames.

6. `draw` renders the tree, showing the new node.

This works in tandem with collaboration:

1. `Node` provides the structure for tree storage.
2. `insert` uses recursive BST logic and AVL balancing.
3. raylib's `GetFrameTime` and drawing functions animate and display the result.

5.3 Enhancements

We shall now delineate the enhancements we have made on our visualizer. We made each function's purpose, logic and algorithm basis more explicit, such as AVL balancing and Kruskal's MST. There is a key flow for each DSA showing how user actions trigger method calls, data updates, animations and rendering. This clarifies collaborations of each DSA between their functionality and raylib's personality. This is also improved with respect to algorithms, e.g. linear interpolation and `Union-Find`

6 Technical Problems and Solutions

6.1 Merging individual DSA projects into a unified application

Initially, each DSA was handled rather disparately; that is, they were self-contained with each of their own `main()` function, window initialization and event loops. Creating a menu system for the whole project then revealed inconsistencies across window management and layouts, as well as coordinate mismatches. In the worst case scenario, this causes flickering and crashes. As such, we standardized the screensize to 1400×1000 across all DSA's, adjusted buttons' locations and remove independent window management functions in each DSA's `main.cpp`. This resulted in a seamless experience when switching across each DSA visualizer.

6.2 Multiple windows opening for visualizations

Post-merge, entering Linked List's, Hash Table's and Graph's visualizers opens a new window due to lingering `InitWindow()` calls, increasing resource usage and fragmented user experience. A fix on DSA merging corrected this issue.

6.3 Graph visualization shaking

An issue was found in the Graph visualizer, i.e. dual `BeginDrawing()`/`EndDrawing()` cycles and another in `runGraph()` for the "Return" button, making the visualizer shake excessively by disrupting the force-directed layout's synchronization. Moving the "Return" button rendering inside `BeginDrawing()`/`EndDrawing()` block

and aligning it with the force directed updates (`Update()` with $k = 15000.0f$) helped with visual coherence.

6.4 Animation lag in large datasets

Loading large files with 100 values or above into AVL Tree and Linked List DSA caused animation lag due to the recursive logic in `updateAnimation`. We precomputed `std::vector<Node*>` in `updateAnimation` to avoid recursive traversals per frame and reduced interpolation factor from `deltaTime`$\times 2.0f$ to `deltaTime`$\times 1.5$ for AVL Tree and Linked List respectively.

7 Feature Demonstration

8 Conclusion

Our DSA visualizer represents our achievements in creating an educational tool in computer science. We try our best to develop a robust application that highlights the utmost important aspects of each DSA.

Comprehensive visualization. Each DSA supports core operations, i.e. search, insert and delete. With real-time animation, line-by-line code demonstration and comprehensive structural changes, our visualizer successfully showcases advanced algorithmic behaviors.

Unified interface. The integration of standalone DSA into one unified menu provides a symmetrical menu and seamless user experience.

Interactive features. Aside from core operations, the visualizer also supports undo/redo, file input and random data generation, which is integral to an interactive DSA visualizer.

While we admire our current progress, much is left to be improved. We set our eyes on these following goals for a much more robust visualizer.

Additional DSA. The additions of Heaps, Tries or B-Trees could expand the visualizer's scopes and toolsets.

Persistent storage. As of now, data is stored in memory and will be reset on exit. A save/load feature using JSON or binary file would tackle this issue and allow the user to revisit DSA states.

Performance optimization. For extremely large datasets up to thousands of nodes, further optimizations are required to reduce lag and refine the program's control flow.

In recapitulation, this project not only demonstrates our technical skills but also lays a foundation for interactive visualizers in the field. We look forward to our next project in order to elevate the visualizer to make it more useful and comprehensive.

9 References

Raylib official website and documentation. The primary resource for understanding Raylib's graphics and input handling functions, such as `DrawText`, `DrawCircle`, `DrawLineV`, `GetMousePosition`, `IsMouseButtonPressed`, `GetFrameTime`, `BeginDrawing` and `EndDrawing`.

TinyFileDialogs project page. Official documentation for the TinyFileDialogs library, specifically the `tinyfd_openFileDialog` function, which enabled file selection for loading data into AVL Tree, Linked List, Hash Table, and Graph visualizations.

C++ Reference. A comprehensive reference for the C++ Standard Template Library (STL).

Stack Overflow. A community-driven Q&A platform providing solutions to common programming challenges, such as pointer management, animation techniques, and Raylib usage.

Introduction to Algorithms. A seminal textbook on algorithms, providing detailed explanations of AVL Tree balancing, Linked List operations, Hash Table chaining, and Graph algorithms like Kruskal's Minimum Spanning Tree.

GeeksforGeeks An online resource offering tutorials and code examples for data structures and algorithms, including AVL Trees, Linked Lists, Hash Tables, and Graphs.

GNU Compiler Collection (GCC) Documentation Documentation for the GCC compiler, used to compile the project with Raylib and TinyFileDialogs libraries.