Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report

on

"DSA Visualizer"

[Code No: COMP 202]

For partial fulfillment of Second Year/first Semester in Computer Engineering

Submitted By:

Ruyoj KC [23]

Submitted To:

Mr. Sagar Acharya

Department of Computer Science and Engineering

Submission Date: 2026-02-25

Bona fide Certificate

This is to certify that the project entitled

" DSA Visualizer "

is a bonafide work carried out by the student in partial fulfillment of the requirements for the Second Year /First Semester of the Bachelor of Engineering in Computer Engineering, under the guidance of Mr. Sagar Acharya, Department of Computer Science and Engineering.

Name of Student :

Ruyoj KC

The project work has been carried out with sincerity, dedication, and to the satisfaction of the department, adhering to the academic guidelines.

Project Supervisor

_____

Mr. Sagar Acharya

Department of Computer Science and Engineering

Date: 2026-02-25

Acknowledgement

I would like to express my sincere and heartfelt gratitude to my respected teacher, **Mr. Sagar Acharya**, for his continuous guidance, invaluable suggestions, and constant encouragement in the field of Data Structures and Algorithms. His excellent teaching, insightful explanations, and dedication to the subject inspired me to undertake and successfully complete this project. His support played a vital role in strengthening my understanding of the core concepts that formed the foundation of this work.

I would also like to extend my sincere appreciation to the Department of Computer Science and Engineering for providing me with the opportunity and platform to apply my theoretical knowledge in a practical and meaningful way. The academic environment, resources, and encouragement provided by the department were instrumental in helping me carry out and complete this project successfully.

Finally, I would like to express my gratitude to everyone who directly or indirectly supported and encouraged me during the course of this project. Their motivation and support contributed significantly to the successful completion of this work. I am truly thankful for their assistance and encouragement.

Thank You

Abstract

Data Structures and Algorithms form the core foundation of efficient, optimized, and logically structured software systems. This project introduces an interactive Data Structure Visualizer developed using **C++** with a graphical interface built using the raylib framework. The primary objective of this system is to demonstrate the practical application and behavior of fundamental data structures such as Stack, Queue, Linked List, and Binary Search Tree (BST) within an intuitive and user-friendly environment.

In this visualizer, Stack (std::vector<int>) follows the Last-In-First-Out (LIFO) principle and allows users to observe push and pop operations in real time. The Queue is implemented using std::queue<int>, strictly adhering to the First-In-First-Out (FIFO) mechanism as elements are enqueued and dequeued. The Linked List is represented using dynamically allocated nodes, showcasing sequential traversal and node removal behavior. The Binary Search Tree, built using dynamically linked TreeNode structures, includes animated positional transitions for each inserted node, enabling users to clearly understand tree balancing patterns and hierarchical relationships, even though linear structures do not include smooth animations. Additionally, the system incorporates automatic layouting for BST nodes, providing a clean structural visualization after every insertion.

The graphical interface is constructed using raylib's 2D rendering capabilities, offering clear, minimalistic, and responsive visuals. Buttons, highlighting states, real-time updates, and structured layouting collectively provide a seamless interactive learning experience. The project also includes sorting functionality for linear structures, random value generation, and a sidebar-based navigation system to allow users to explore each structure independently. Overall, this application demonstrates the integration of core data structures within a graphical environment, improving conceptual understanding through visual interaction and reinforcing essential principles of Data Structures and Algorithms.

Keywords: Data Structure Visualizer, C++, raylib, Stack, Queue, Linked List, Binary Search Tree, Visualization, Interactive Learning.

Table of Contents

List of Figures

Acronyms/Abbreviations

The list of all abbreviations used in the documentation are as follows :

DSA: Data Structures and Algorithms

BST: Binary Search Tree

GUI: Graphical User Interface

LIFO: Last In, First Out

FIFO: First In, First Out

OOP: Object-Oriented Programming

IDE: Integrated Development Environment

API: Application Programming Interface

FPS: Frames Per Second

UI: User Interface

UX: User Experience

CPP: C++ Programming Language

LL: Linked List

QDS: Queue Data Structure

SDS: Stack Data Structure

Chapter 1 Introduction

1.1 Background

Data Structures and Algorithms (DSA) play a vital role in computer science, providing the core principles required to design efficient, scalable, and optimized software solutions. Understanding how data structures operate internally—and how algorithms manipulate them—is fundamental for students and developers aiming to solve real-world computational problems. However, traditional methods of learning DSA often rely heavily on theoretical explanations or console-based implementations, which can limit conceptual clarity and hinder visualization of how these structures behave during runtime.

Many learners struggle to fully grasp dynamic behaviors such as how elements flow through a Queue, how nodes link together inside a Linked List, or how a Binary Search Tree rearranges itself during insertion. Without visual feedback, operations like traversal, insertion, and deletion become abstract concepts rather than intuitive processes. As a result, there is a notable gap between theoretical understanding and practical, real-time visualization of DSA operations.

To address this challenge, this project introduces an interactive DSA Visualizer built using C++ and the raylib graphics framework. The application provides a clean, modern interface where users can explore and observe the behavior of fundamental data structures including Stack, Queue, Linked List, and Binary Search Tree. Each structure is rendered visually, allowing learners to perform operations such as insert, remove, and search, while instantly seeing the outcome on screen. Unlike traditional console programs, this system emphasizes visual clarity, user interaction, and simplicity, making it easier for students to understand how data structures operate at a deeper level.

By presenting DSA concepts in an interactive graphical format, this project enhances engagement, supports intuitive learning, and demonstrates how theoretical data structures can be effectively implemented in a real-world application environment

1.2 Objectives

The primary objectives of the DSA Visualizer Project are as follows:

- To implement core data structures—Stack, Queue, Linked List, and Binary Search Tree—in C++, providing a practical demonstration of fundamental operations such as insertion, deletion, traversal, and ordering within an interactive graphical environment.

- To develop a real-time visualization system using the raylib graphics framework, enabling users to observe how each data structure behaves internally, thereby enhancing conceptual understanding through dynamic graphical representation.

- To design a clean, modern, and user-friendly graphical interface featuring selectable data structures, operation controls, and intuitive visual layouts that allow learners to explore DSA behavior with ease.

- To provide a Binary Search Tree with animated node placement, demonstrating hierarchical structuring, parent-child relationships, and ordered insertion as used in many algorithmic applications.

- To incorporate sorting functionality for each linear structure (Stack, Queue, Linked List) by extracting, ordering, and reconstructing their data, allowing users to visualize how arranged data differs from its unsorted state.

- To serve as an educational tool that bridges the gap between theoretical DSA concepts and practical visualization, making it easier for students to prepare for academic study, laboratory work, project presentations, and viva examinations.

1.3 Motivation and Significance

The motivation behind this project arises from the need to make Data Structures and Algorithms concepts more visual, interactive, and easier to understand. In traditional learning environments, these concepts are often explained using simple console-based examples that do not fully demonstrate their real-world applications. This project addresses that limitation by presenting data structures within a complete graphical application, allowing users to observe how each structure behaves and operates in real-time.

This project is significant as it naturally integrates multiple data structures in a meaningful and practical context rather than using them in isolation. Stacks are visualized with their LIFO (Last-In-First-Out) principle, Queues demonstrate FIFO (First-In-First-Out) behavior, Linked Lists show node connections, and Binary Trees illustrate hierarchical relationships through smooth animations. This integration helps demonstrate the relationship between theoretical concepts and their practical implementation in solving real-world problems.

Furthermore, the use of the raylib graphics library provides a clean and interactive visual interface with smooth animations, which improves usability and presentation quality. The project not only strengthens programming and problem-solving skills but also serves as an effective educational tool for learning and demonstrating DSA concepts. Its practical implementation, visual clarity, and interactive design make it highly suitable for academic evaluation, project demonstration, and viva examinations.

Chapter 2 Related Works

Several research studies and educational tools have explored the visualization of data structures and algorithms. Online platforms such as VisuAlgo and Algorithm Visualizer provide interactive, web-based demonstrations of various data structures, including Stack and Queue operations. VisuAlgo, developed by Dr. Steven Halim from the National University of Singapore, animates figures from prominent textbooks like Introduction to Algorithms (CLRS), making complex concepts more accessible to learners worldwide.

Traditional implementations of data structures are commonly taught using console-based C++ programs, focusing on theoretical concepts without graphical visualization. Standard textbooks such as Introduction to Algorithms by Cormen et al. and Data Structures, Algorithms, and Applications in C++ by Sahni provide comprehensive theoretical explanations but lack practical GUI-based implementations for interactive learning.

Several educational applications have been developed using various frameworks to visualize algorithms. The manim-dsa library enables educators to create animated data structure visualizations for educational videos. A Qt C++ project called "Algorithm Visualizer" offers step-by-step animations for sorting algorithms, backtracking problems, and tree structures. Recent research by Keçeci (2025) introduced the "Keçeci Layout" for visualizing structured systems, demonstrating ongoing academic interest in visualization methodologies.

While these platforms demonstrate individual algorithm behavior effectively, implementations that explicitly connect Stack, Queue, Linked List, and Binary Tree operations within a unified graphical interface, along with smooth animation mechanisms, are not widely documented. This project addresses that gap by combining interactive visualization, real-time operations, and direct implementation of fundamental data structures in a single desktop application using the raylib graphics framework.

Chapter 3 Design and Implementation

3.1 Implementation

The system was implemented using an object-oriented programming approach in C++, with the graphical user interface developed using the raylib graphics library. The project is organized into several classes including TreeNode, BinaryTree, Stack, Queue, LinkedList, and Visualizer, each designed with clearly defined roles to ensure modularity, maintainability, and clarity. This structured approach allows the separation of data structure logic, visualization components, and user interface. Special emphasis was placed on ensuring that each data structure—Stack, Queue, Linked List, and Binary Tree—was not only implemented internally but also visibly demonstrated through graphical representation and user interaction.

3.1.1 Implementation Planning

The implementation was carefully planned to demonstrate four fundamental DSA concepts: Stack, Queue, Linked List, and Binary Tree. The Stack data structure was implemented to visualize LIFO (Last-In-First-Out) behavior through vertical stacking of elements. The Queue was implemented to demonstrate FIFO (First-In-First-Out) behavior with elements arranged horizontally. The Linked List was designed to show node connections with arrows between elements. The Binary Tree was implemented with smooth animations to illustrate hierarchical relationships and BST insertion rules.

During the planning phase, raylib was selected as the primary graphics library due to its simplicity, efficiency, and cross-platform compatibility. A custom Animation system was developed to handle smooth transitions for tree nodes, ensuring visual continuity when elements are added or removed. The development process was divided into several key tasks, including implementation of core data structures, graphical visualization logic, user interface design with interactive buttons, animation handling, and the integration of sorting functionality to enhance the educational value of the application.

3.1.2 Requirement Analysis

Functional Requirements

The system must provide interactive visualization of four data structures: Stack, Queue, Linked List, and Binary Tree. Users must be able to insert random values (1-100) into each structure, remove elements according to structure-specific rules (pop from stack, dequeue from queue, remove from linked list), and sort elements in ascending order. The Binary Tree must maintain BST properties during insertion and display smooth animations as nodes reposition. The interface must include a sidebar with mode selection buttons that highlight the active structure, and all visualizations must update in real-time with clear representation of elements and their relationships.

Non-Functional Requirements

The application must run reliably without crashes, maintain a consistent frame rate of 60 FPS for smooth animations, and provide responsive user input handling. The interface should be clean and intuitive with proper visual feedback for button hover states and active modes. The system must correctly implement all data structure operations and accurately reflect their behavior through the visual representation.

3.1.3 System Design

The system follows a modular architecture separating data structures, visualization logic, and user interface. The main.cpp file handles the application loop, window management, and user input processing. Data structure classes (Stack, Queue, LinkedList, BinaryTree) manage the core logic and data storage. The Visualizer class contains all drawing functions for each structure. The UI module provides reusable button components and interface elements.

Data Structure Implementations

Stack Implementation:

The Stack class contains a vector to store elements, with push() and pop() operations. Visualization shows elements stacked vertically from bottom to top, with each element displayed as a colored rectangle containing its value. The LIFO principle is visually demonstrated as new elements appear above existing ones and removal always takes from the top.

Queue Implementation:

The Queue class uses a standard queue container with enqueue() and dequeue() operations. Visualization displays elements horizontally from left to right, with each element shown as a square. New elements appear at the back (right side) and removal occurs from the front (left side), clearly demonstrating FIFO behavior.

Linked List Implementation:

The LinkedList class manages nodes with value and next pointer, supporting insert() and remove() operations. Visualization shows nodes as purple squares connected by black arrows, with each node displaying its value. The sequential nature of linked lists is visually represented through the arrow connections between nodes.

Binary Tree Implementation:

The BinaryTree class implements BST insertion rules with AnimatedPos for smooth node movement. The tree uses recursive insertion where smaller values go left and larger values go right. Visualization draws nodes as blue circles with connecting lines, and the UpdatePositions function recalculates layout for optimal spacing. Smooth animation is achieved through the AnimatedPos::Update() method called each frame.

Visualizer Class

The Visualizer class contains static methods for drawing each data structure. DrawStack() renders stack elements vertically with consistent spacing. DrawQueue() renders queue elements horizontally with proper FIFO ordering. DrawLinkedList() traverses the list and draws nodes with connecting arrows. DrawTree() performs a level-order traversal using a queue, updating node positions and drawing connections before rendering nodes as circles.

UI Components

The UI module provides Button structures with rect, text, and active state. DrawButton() handles hover detection, color transitions, and click detection with visual feedback for different states. DrawHeader() renders a title bar at the top of the window, and DrawSidebar() creates a light blue panel for button placement.

Mode Management

The application uses an enum (STACK, QUEUE, LINKED_LIST, TREE) to track the currently selected structure. Button states reflect the current mode with active highlighting, and all insert/remove/sort operations are routed to the appropriate structure based on the selected mode.

Data Structure Integration

Stack: LIFO behavior with vertical visualization

Queue: FIFO behavior with horizontal visualization

Linked List: Node connections with arrow visualization

Binary Tree: Hierarchical structure with smooth animations

This structured implementation ensures that each data structure is not only functionally correct but also educationally visible, providing an interactive and comprehensive demonstration of fundamental DSA concepts in a real-world application.

3.1.4 Development

The development of the project was carried out in phased stages to ensure modularity, correctness, and stability. Initially, the core data structure classes were implemented and tested with basic operations in isolation. This phase validated the correct functioning of Stack push/pop, Queue enqueue/dequeue, Linked List insertion/removal, and Binary Tree BST insertion before integrating the graphical interface.

Subsequently, the raylib graphical interface was developed around the validated core logic. The visualization functions were implemented incrementally, starting with simple rectangle drawing and progressing to full structure representations with proper spacing and alignment. The Animation system for tree nodes was developed to provide smooth transitions when nodes are repositioned after insertion.

The major development phases of the project can be summarized as follows:

1. **Core class design:** Stack, Queue, LinkedList, BinaryTree, TreeNode
2. **Console validation:** Testing all operations for correctness
3. **Basic raylib setup:** Window initialization and drawing loop
4. **UI development:** Button system with hover states and click detection
5. **Visualization implementation:** Drawing functions for each structure
6. **Animation system:** AnimatedPos for smooth tree node movement
7. **Mode management:** Enum-based structure selection
8. **Integration:** Connecting UI buttons to structure operations
9. **Testing and refinement:** Ensuring stable execution and visual clarity

This structured development approach ensured that the project was robust, interactive, and educational, successfully integrating core DSA concepts with a clean graphical interface.

3.1.5 Testing and Debugging

The project underwent testing to ensure stability, correctness, and smooth user interaction. Initial testing focused on data structure operations in isolation, verifying that Stack push/pop maintained LIFO order, Queue operations preserved FIFO behavior, Linked List operations correctly managed node connections, and Binary Tree insertions followed BST rules.

Integration testing verified that UI buttons correctly triggered the appropriate structure operations and that visualizations updated accurately after each action. The sort functionality was tested across all structures, ensuring that elements were properly sorted while maintaining structure integrity.

Memory management was carefully handled, particularly for the Binary Tree where recursive insertion could potentially lead to stack overflow with very deep trees. The updatePositions function was tested to ensure proper layout recalculation without causing visual overlap or excessive spacing.

Through iterative testing and refinement, the system now performs reliably, providing a crash-free, interactive, and visually accurate experience that demonstrates the intended Stack, Queue, Linked List, and Binary Tree concepts.

3.2 System Requirement Specifications

The system was developed and tested with specific software and hardware requirements to ensure proper functionality, performance, and compatibility. The specifications are detailed below.

3.2.1 Software Specifications

The following software components are required for the development and execution of the system:

- **Programming Language:** C++11 or later
- **Graphics Library:** raylib (version 4.0 or later)
- **Compiler:** MinGW (Windows), GCC (Linux/Mac), or MSVC
- **Build Tools:** Command-line compiler with necessary linker flags
- **Operating System:** Windows 7+, Linux, or macOS

These components provide a stable foundation for implementing both the backend logic and the graphical frontend, enabling efficient development, debugging, and testing of the DSA Visualizer application.

3.2.2 Hardware Specifications

The system requires the following hardware to operate efficiently:

- **Processor:** Intel Core i3 or equivalent (minimum); Intel Core i5 recommended
- **Memory (RAM):** 2 GB minimum; 4 GB recommended
- **Storage:** 100 MB minimum for executable and project files
- **Display:** 1024×768 resolution minimum; 1200×700 recommended for optimal viewing
- **Input Devices:** Standard keyboard and mouse

These specifications ensure smooth execution of the system, support real-time animations at 60 FPS, and provide a responsive and user-friendly experience while interacting with the visualizations and exploring data structure operations.

Chapter 4 Discussion on the achievements

4.1 Project Overview

The DSA Visualizer project is a C++ application developed using the raylib graphics library, presenting four fundamental data structures as interactive visualizations. The application features a clean, professional interface with a sidebar for structure selection, a header bar, and a main visualization area. Users can interact with Stack, Queue, Linked List, and Binary Tree structures through intuitive buttons for inserting random values, removing elements, and sorting.

The left-hand sidebar displays mode selection buttons (Stack, Queue, Linked List, Binary Tree) with active state highlighting. The main visualization area dynamically updates based on the selected structure, showing elements with appropriate visual representations—vertical stacks for Stack, horizontal queues for Queue, connected nodes for Linked List, and animated circles with connecting lines for Binary Tree. Each structure provides real-time feedback as elements are added, removed, or sorted..

4.2 Implemented Features

4.2.1 DSA Implementations

Stack Implementation

The Stack is implemented using a vector container with standard push() and pop() operations. Visualization shows elements stacked vertically from bottom to top, with each element displayed as a green rectangle containing its value. The LIFO (Last-In-First-Out) property is visually demonstrated as new elements appear above existing ones, and removal always takes from the top. The stack visualization includes proper spacing between elements and clear value labeling.

Queue Implementation

The Queue uses a standard queue container with enqueue() and dequeue() operations. Visualization displays elements horizontally from left to right, with each element shown as an orange square. New elements are added to the back (right side) and removal occurs from the front (left side), clearly demonstrating FIFO (First-In-First-Out) behavior. The queue maintains consistent spacing between elements and shows values centered within each square.

Linked List Implementation

The Linked List is implemented as a custom class with nodes containing a value and pointer to the next node. Visualization shows nodes as purple squares connected by black arrows, with each node displaying its value. The insert() operation adds new nodes to the list, while remove() deletes nodes appropriately. The arrow connections visually represent the linked nature of the structure, helping users understand how nodes reference each other in memory.

Binary Tree Implementation

The Binary Tree implements Binary Search Tree (BST) insertion rules where smaller values go to the left and larger values go to the right. Each TreeNode contains a value, left and right child pointers, and an AnimatedPos structure for smooth movement. The updatePositions() function recalculates node positions after insertion to maintain optimal visual layout. The animation system calls pos.Update() each frame, creating smooth transitions as nodes move to their target positions. Visualization draws nodes as blue circles with connecting lines and displays values centered within each circle.

Sort Functionality

Each data structure includes a sort feature that arranges elements in ascending order. For Stack and Queue, elements are temporarily transferred to vectors, sorted using std::sort(), and then rebuilt. For Linked List, values are extracted, sorted, and used to reconstruct the list. The Binary Tree maintains its BST property inherently, so sorting is not applicable. This feature demonstrates how sorting algorithms can be applied to different data structures while preserving their fundamental characteristics.

4.2.2 GUI and Interaction

Sidebar Navigation

The left sidebar contains buttons for each data structure (Stack, Queue, Linked List, Binary Tree). Buttons change appearance on hover and highlight in blue when their associated structure is active. This provides clear visual feedback about the current mode and available options. The sidebar maintains consistent positioning throughout the application lifecycle.

Control Buttons

Three primary control buttons are positioned below the structure selection:

Insert Random: Adds a random value (1-100) to the currently selected structure

Remove: Removes an element according to structure-specific rules

Sort: Sorts elements in ascending order

Each button provides visual feedback on hover and click, ensuring intuitive interaction.

Visualization Area

The main area (positioned to the right of the sidebar) dynamically renders the selected data structure. Each structure has its own dedicated drawing function in the Visualizer class:

DrawStack(): Renders vertical stack elements with proper spacing

DrawQueue(): Renders horizontal queue elements with FIFO ordering

DrawLinkedList(): Renders connected nodes with arrow indicators

DrawTree(): Performs level-order traversal to render nodes and connections

Animation System

The Binary Tree features smooth animations through the AnimatedPos structure. Each node maintains current and target positions, with an Update() function that moves the node incrementally toward its target. This creates fluid motion when nodes are repositioned after insertion, making the tree reorganization visually clear and engaging.

Header and Layout

A consistent header bar displays the application title "DSA Visualizer" at the top of the window. The sidebar begins below the header, and the visualization area occupies the remaining space. This clean layout ensures all elements are easily accessible and visually organized.

4.3 Achievements

During the development of the DSA Visualizer project, several key achievements were realized:

Comprehensive Data Structure Implementation

A complete set of fundamental data structures was implemented and integrated into a single application. The Stack, Queue, Linked List, and Binary Tree classes each correctly implement their respective operations while maintaining data integrity. The Binary Tree includes a sophisticated animation system for smooth node transitions, demonstrating advanced C++ programming concepts.

Clean GUI Design

The raylib-based interface provides a professional, responsive user experience. The sidebar navigation with active state highlighting, hover effects, and consistent layout creates an intuitive interaction model. The visualization area dynamically updates to reflect the current structure, ensuring users always have clear visual feedback.

Cross-Platform Compatibility

By using raylib, the application maintains compatibility across Windows, Linux, and macOS platforms with minimal modifications. The build process is straightforward, requiring only standard compiler tools and the raylib library.

Educational Impact

The project successfully transforms abstract data structure concepts into tangible visual experiences. Users can see exactly how elements move in a stack, how queues process elements in order, how linked list nodes connect, and how binary trees maintain hierarchical relationships. This visual approach accelerates understanding and retention of these fundamental concepts.

Technical Skill Development

Development of this project strengthened proficiency in:

- C++ object-oriented programming and memory management
- Data structure implementation and algorithm design
- Graphics programming with raylib
- Event handling and user input processing
- Animation systems and smooth transitions
- Debugging and testing methodologies

Modular Architecture

The clean separation between data structures (structures folder), visualization logic (visuals folder), and user interface (ui folder) demonstrates professional software design principles. This modularity makes the codebase maintainable, extensible, and easy to understand..

Chapter 5 Conclusion and Recommendation

The DSA Visualizer project successfully accomplished its primary objective of creating an interactive desktop application that explicitly demonstrates four fundamental data structures—Stack, Queue, Linked List, and Binary Tree—within a unified graphical environment. The Stack is implemented using a vector container with push/pop operations, clearly visualizing LIFO behavior through vertical element stacking. The Queue utilizes a standard queue container with enqueue/dequeue operations, demonstrating FIFO behavior through horizontal element arrangement. The Linked List implements custom node structures with pointer connections, visually represented by purple squares connected by arrows. The Binary Tree implements BST insertion rules with smooth animations through the AnimatedPos system, allowing users to observe hierarchical relationships and node repositioning in real-time.

The graphical user interface provides an engaging and intuitive experience, incorporating sidebar navigation with active button highlighting, hover effects, and dedicated control buttons for insert, remove, and sort operations. The visualization area dynamically updates based on the selected structure, ensuring clear visual feedback for all user actions. All data structure operations function correctly and reliably, resulting in a stable application that runs smoothly at 60 FPS.

5.1 Limitations

Despite its successful implementation, the project has certain limitations:

1. **Fixed Element Positioning**: The spacing between elements in Stack, Queue, and Linked List visualizations is fixed and cannot be adjusted by users. This may cause overlapping when many elements are added.
2. **Binary Tree Layout Optimization:** While the updatePositions() function attempts to create balanced layouts, extremely deep or unbalanced trees may still experience visual overlap or suboptimal spacing.
3. **Limited Sorting Visualization:** The sort operation occurs instantly rather

than step-by-step, so users cannot observe the sorting process itself—only the before and after states.

4. **Static Animation Speed:** The animation speed for tree nodes is fixed at 8 pixels per frame and cannot be adjusted by users.
5. **No Persistent Storage:** The application does not save user data or structure states between sessions; all data is lost when the program closes.
6. **Desktop Platform Only:** The application runs only on desktop platforms (Windows, Linux, Mac) and does not support mobile or web deployment.

5.2 Future Enhancement

The current implementation of the DSA Visualizer provides a stable and educational platform, but several enhancements can further improve functionality, interactivity, and learning value:

1. **Adjustable Animation Speed:** Incorporate controls to allow users to slow down or speed up tree node animations, providing a more customizable learning experience.

2. **Step-by-Step Sorting Visualization:** Implement visual sorting algorithms (bubble sort, insertion sort, etc.) that show each comparison and swap in real-time, helping users understand how different sorting algorithms work.

3. **Dynamic Tree Balancing:** Add AVL or Red-Black tree implementations to demonstrate self-balancing tree concepts alongside the basic BST.

4. **Search Operations:** Implement and visualize search operations for each structure, highlighting elements as they are found or showing traversal paths in trees.

5. **Additional Data Structures:** Expand the application to include graphs, hash tables, and other advanced structures with appropriate visualizations.

References

1 Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

2 Sahni, S. (2005). Data Structures, Algorithms, and Applications in C++ (2nd ed.). Universities Press.

3 raysan5. (2025). raylib Documentation: Graphics Programming Library. Retrieved from https://www.raylib.com/

4 VisuAlgo. (n.d.). Data Structure and Algorithm Visualizations. Retrieved February 2026, from https://visualgo.net/

5 Algorithm Visualizer. (n.d.). Interactive Algorithm Animations. Retrieved February 2026, from https://algorithm-visualizer.org/

6 Halim, S., & Halim, F. (2011). VisuAlgo: Visualising Data Structures and Algorithms through Animation. Presented at the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education.

7 Stroustrup, B. (2013). The C++ Programming Language (4th ed.). Addison-Wesley.

8 GeeksforGeeks. (2025). Data Structures Tutorial with Visualizations. Retrieved from https://www.geeksforgeeks.org/data-structures/

APPENDIX

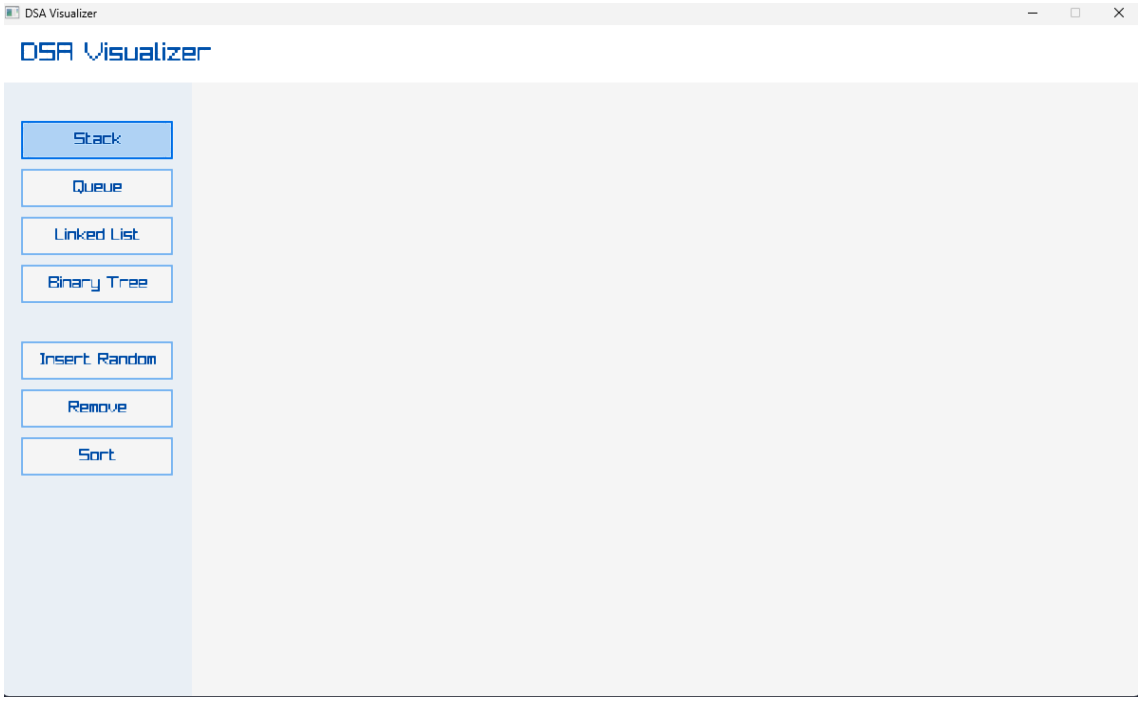

Fig A.1 Application

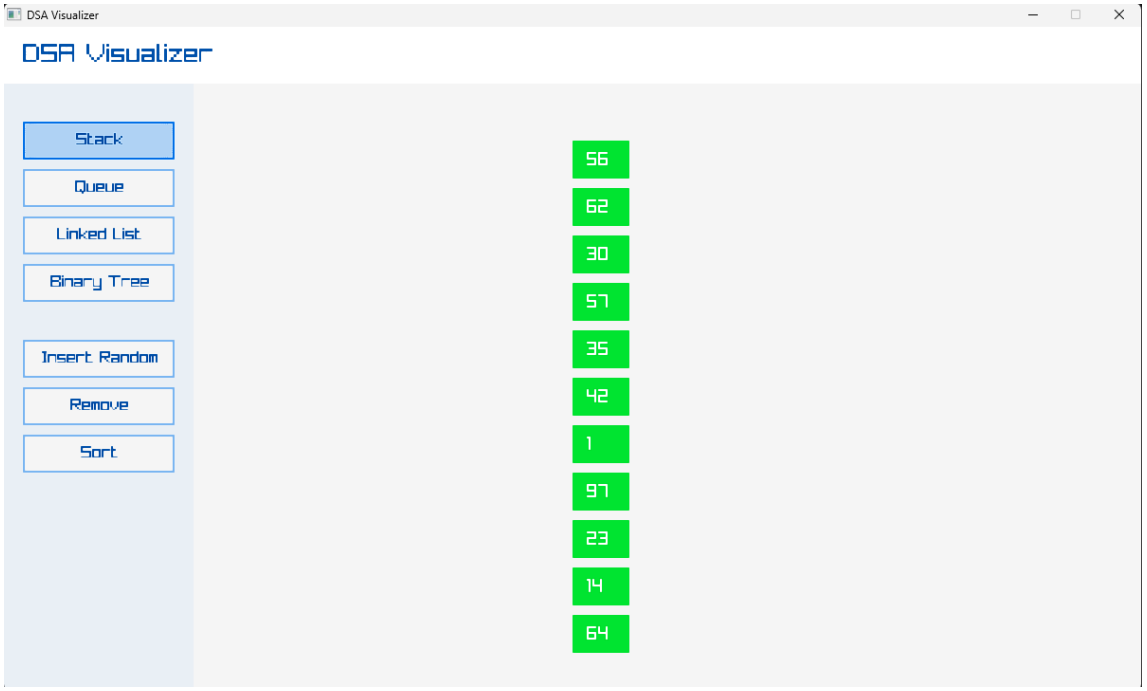

Fig A.2 Inserting random numbers in the stack

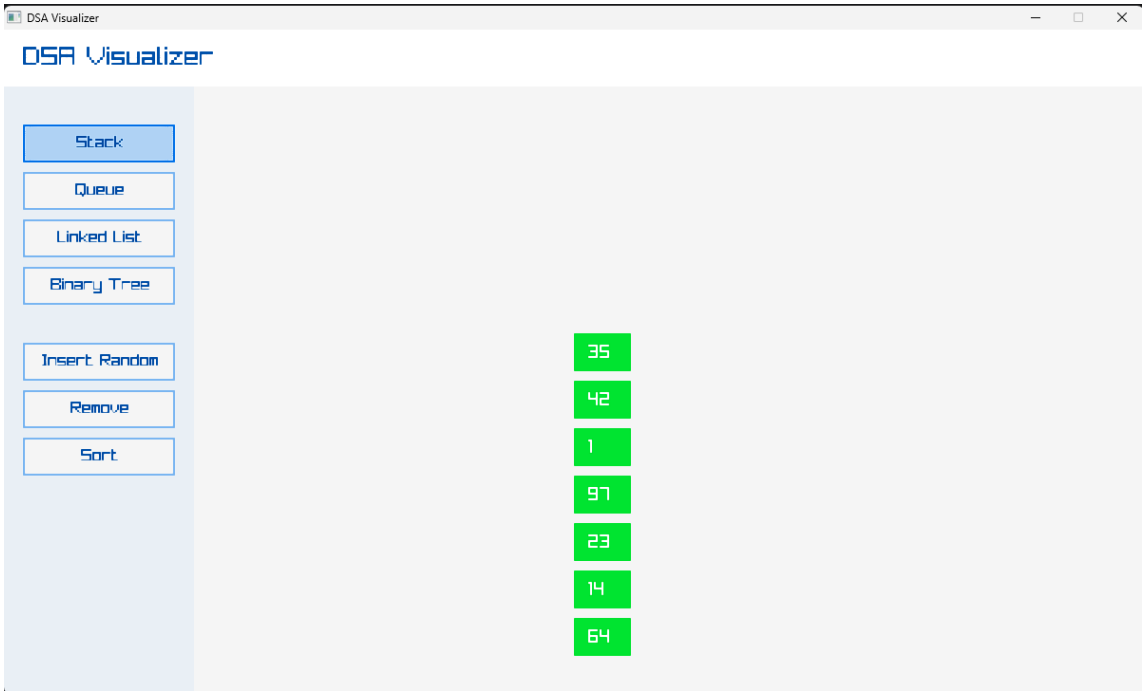

Fig A.3 Removing numbers from the stack

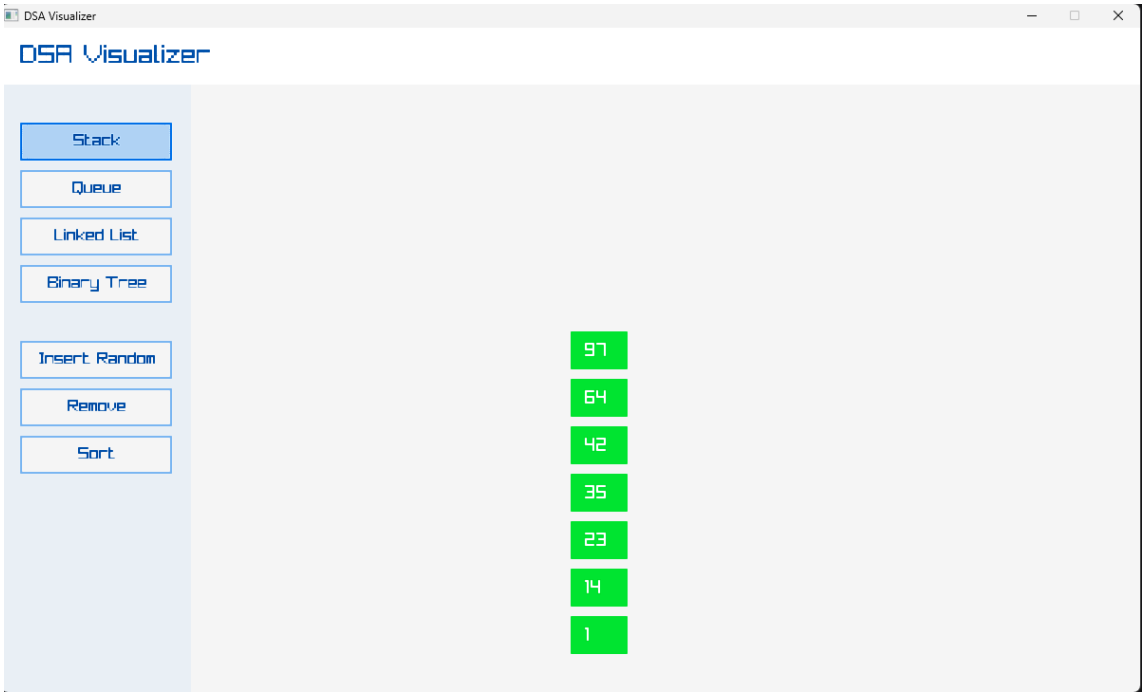

Fig A.4 Sorting the numbers in the stack

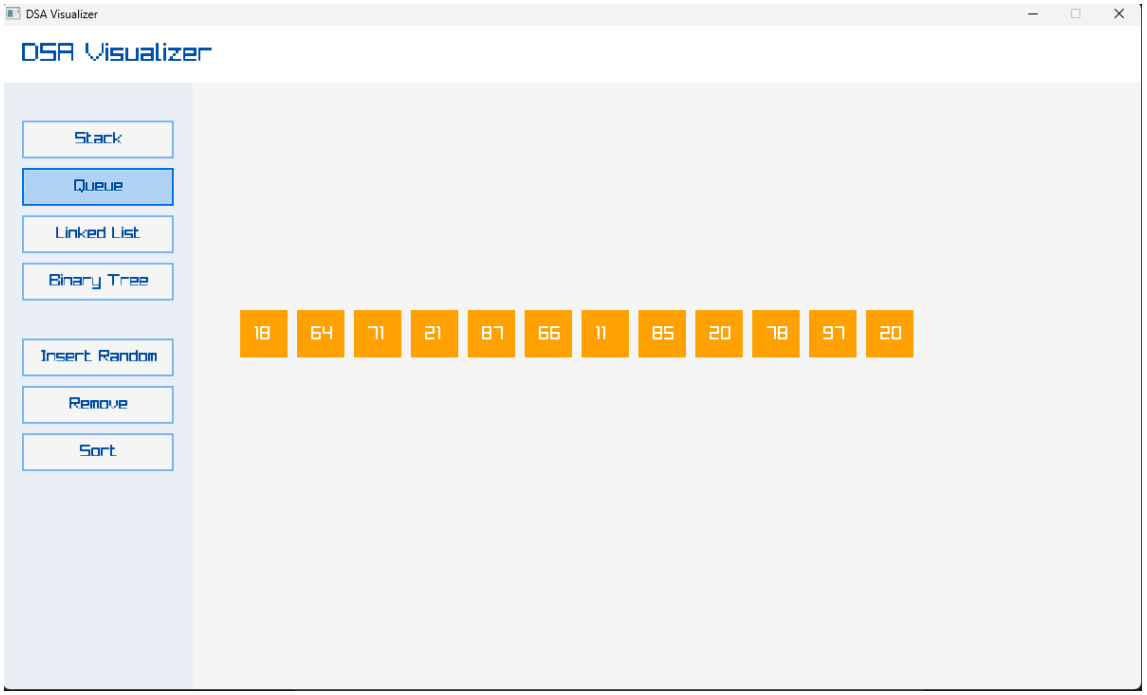

Fig A.5 Inserting random numbers in the queue

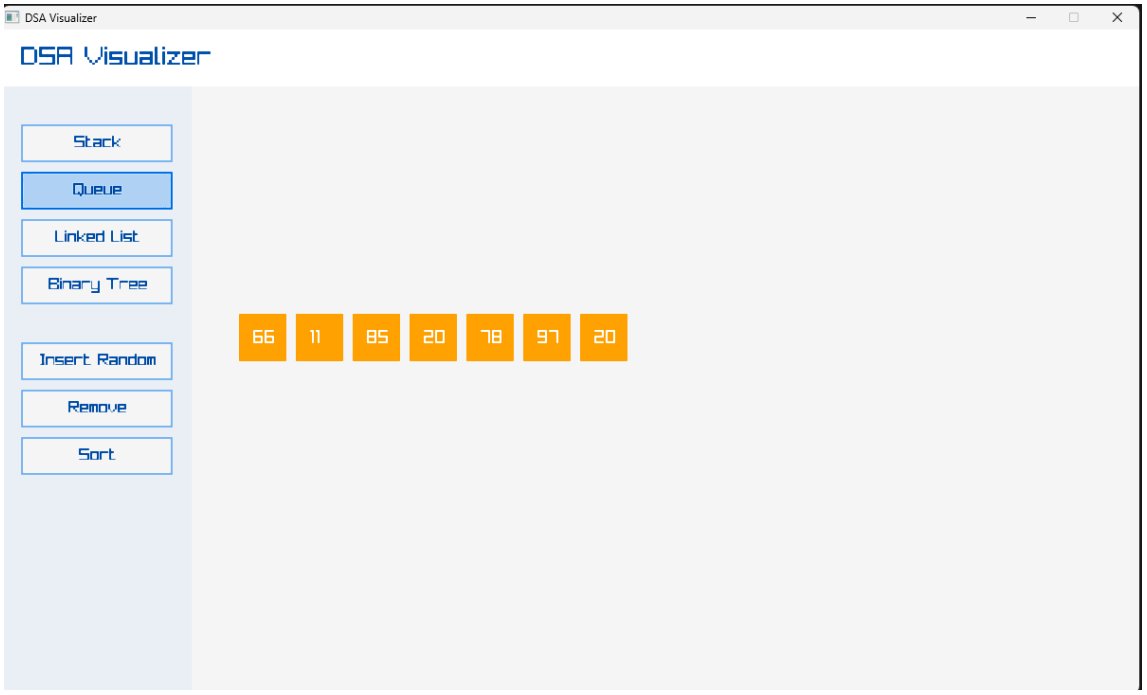

Fig A.6 Removing numbers from the queue

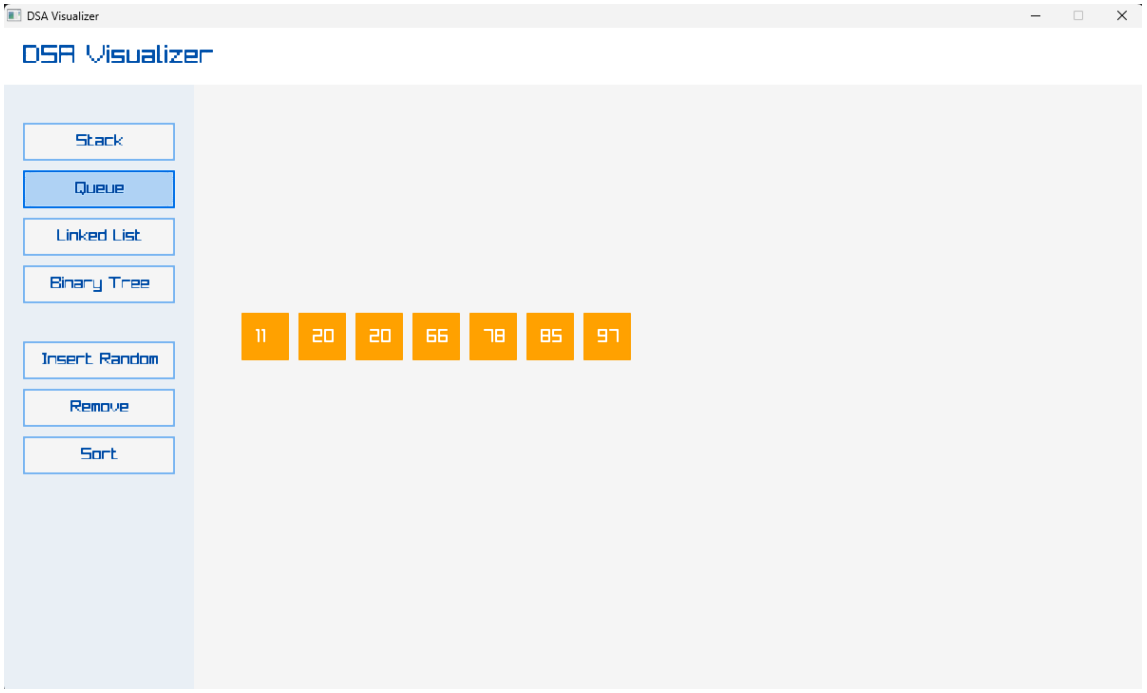

Fig A.7 Sorting elements in the queue

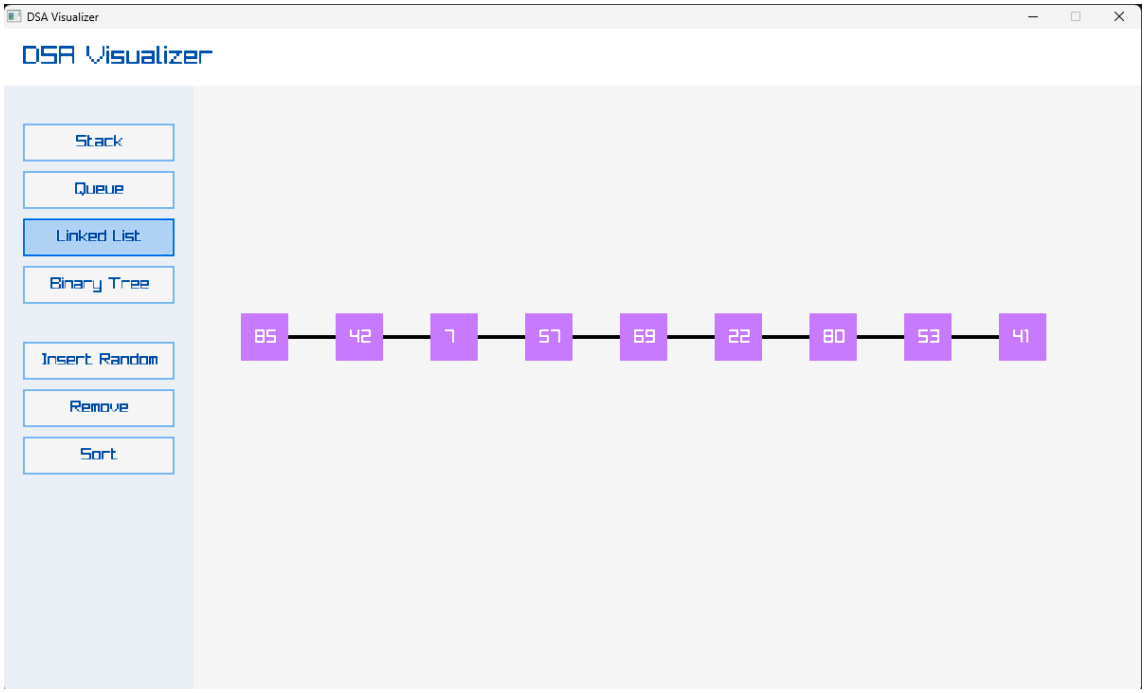

Fig A.8 Inserting random numbers in the linked list

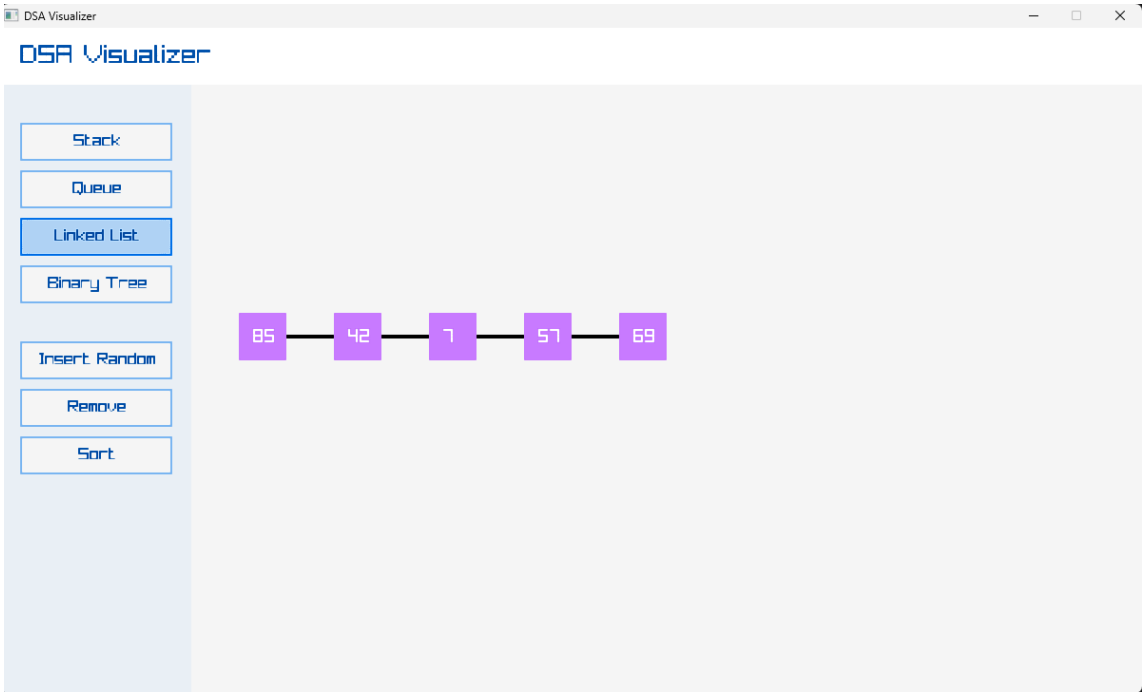

Fig A.9 Removing numbers from the linked list

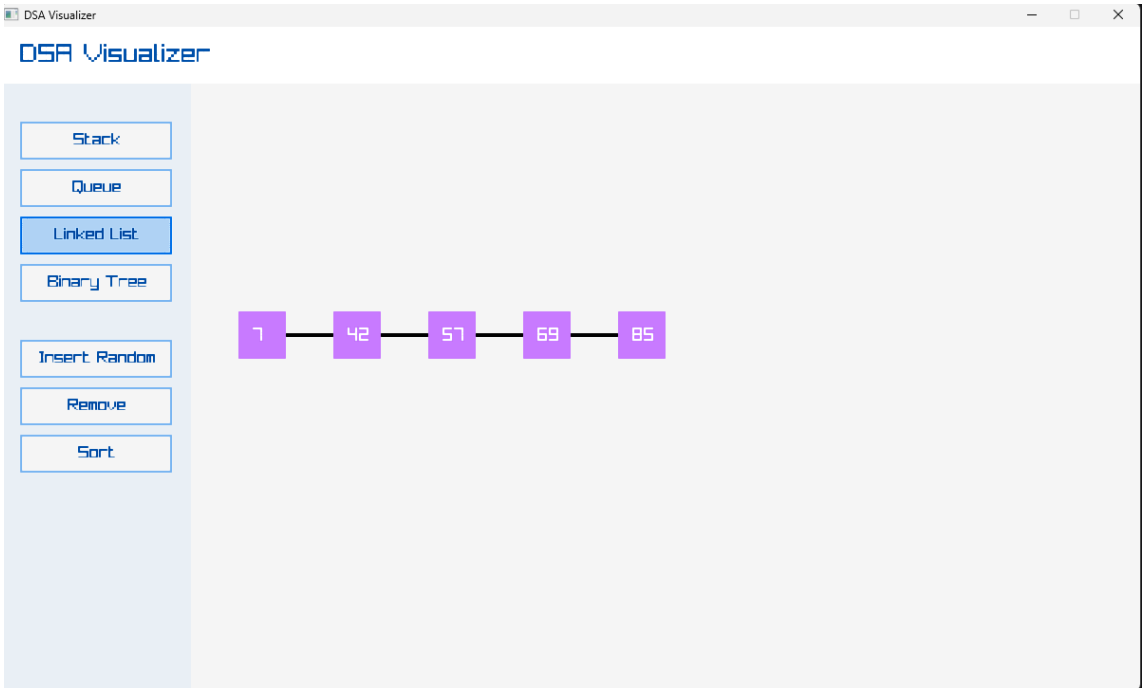

Fig A.10 Sorting elements in the linked list

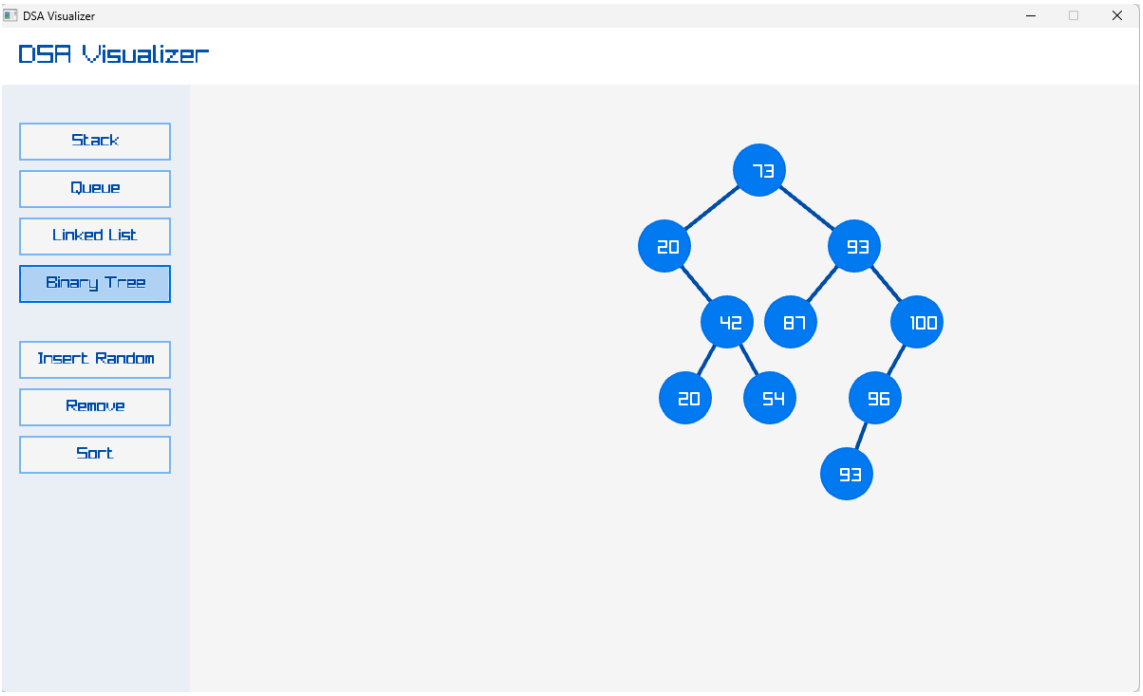

Fig A.11 Creating a binary tree