Documentation

Software Specification:

Main Function Plan

1. Setup and Initialization

- Purpose: Initialize necessary data structures and provide user interface (UI) options.
- Methods Involved:
    - UnsortedArray: Initialize an unsorted array with a given capacity.
    - SortedArray: Initialize a sorted array with a given capacity.
    - UnsortedLinkedList: Initialize an unsorted linked list.
    - SortedLinkedList: Initialize a sorted linked list.
- Steps:

1. Display a menu with options for interacting with each data structure (UnsortedArray, SortedArray, UnsortedLinkedList, SortedLinkedList).
2. Prompt the user to select a data structure to interact with.

2. Adding Elements to Data Structures

- Purpose: Allow users to add elements to the selected data structure.
- Methods Involved:
    - add(value: T): Method of UnsortedArray, SortedArray, UnsortedLinkedList, and SortedLinkedList to add a new element.
- Steps:

1. Prompt the user to input a value to add.
2. Call the add() method on the selected data structure.
3. Handle any errors or capacity limits (e.g., full array).
4. Optionally display a confirmation message or the updated structure.

3. Sorting Data Structures

- Purpose: Sort the selected data structure using merge sort (for SortedArray and SortedLinkedList).
- Methods Involved:

- sort(): Method of SortedArray and SortedLinkedList to initiate sorting.
- mergeSort() and merge() (internal methods in SortedArray and SortedLinkedList) to handle the recursive sorting process.

- Steps:

1. If the selected data structure supports sorting (SortedArray or SortedLinkedList), call the sort() method.
2. Display the sorted structure.
3. Optionally track and display the number of comparisons made during sorting.

4. Searching Elements

- Purpose: Allow users to search for elements in the selected data structure.
- Methods Involved:
  - search(value: T): Method of UnsortedArray, SortedArray, UnsortedLinkedList, and SortedLinkedList to search for an element.
- Steps:

1. Prompt the user to input a value to search.
2. Call the search() method on the selected data structure.
3. Display whether the element was found or not.
4. Optionally display the number of comparisons made during the search.

5. Displaying the Data Structures

- Purpose: Allow users to view the contents of the data structures.
- Methods Involved:
  - display(outputArea: TextArea): Method of UnsortedArray, SortedArray, UnsortedLinkedList, and SortedLinkedList to display the contents.
- Steps:

1. Call the display() method for the selected data structure to print its contents to a UI component (e.g., a TextArea).
2. Optionally allow users to display the current comparisons count for sorting and searching operations.

6. Clearing Data Structures

- Purpose: Allow users to reset the data structure (particularly for arrays and lists).
- Methods Involved:

- clear(): Method of UnsortedArray to reset the array.

- Steps:

1. If the selected data structure supports it, prompt the user to clear the structure.
2. Call the clear() method on the selected structure to reset it.
3. Optionally display a confirmation message.

7. Performance Analysis

- Purpose: Provide users with performance insights on sorting and searching operations.
- Methods Involved:
  - getComparisons(): Method of UnsortedArray and UnsortedLinkedList to retrieve comparison counts for searches.
  - getSortComparisons(): Method of SortedArray and SortedLinkedList to retrieve comparison counts for sorting.
- Steps:

1. After sorting or searching, retrieve the number of comparisons made by calling the respective getComparisons() or getSortComparisons() method.
2. Display the comparison count to the user.

Main Program Flow

1. Initialize Data Structures: Create objects for UnsortedArray, SortedArray, UnsortedLinkedList, and SortedLinkedList.
2. Display Menu: Show options to the user (add elements, sort, search, display, clear, etc.).
3. User Selection: Based on user input, perform actions on the selected data structure.
   - If "Add Element" is selected:
     - Prompt for input and call add().
   - If "Sort" is selected:
     - Call sort() for the sorted structures.
   - If "Search" is selected:
     - Prompt for input and call search().
   - If "Display" is selected:
     - Call display() to show current elements.

    - If "Clear" is selected:
        - Call clear() to reset the structure.
4. Performance Analysis: After sorting or searching, show comparison counts.
5. Exit or Repeat: Allow the user to perform additional operations or exit the program.

UnsortedArray Plan

1. Add Elements:
    - Call add(value) to insert the value.
    - Confirm with "Element added successfully."
2. Search:
    - Use search(value) for linear search.
    - Display result (found or not) and comparisons via getComparisons().
3. Display:
    - Call display() to print array contents.
4. Clear:
    - Call clear() to reset the array.
    - Confirm with "Array cleared."
5. Performance Analysis:
    - Retrieve comparison counts after the search using getComparisons().

UnsortedLinkedList Plans

1. Add Elements:
    - Call add(value) to insert a node at the head.
    - Confirm with "Element added successfully."
2. Search:
    - Use search(value) for linear search through the list.
    - Display result (found or not) and comparisons via getComparisons().
3. Display:
    - Call display() to print list contents.
4. Clear:

- Set head = null to reset.
- Confirm with "List cleared."

5. Performance Analysis:
    - Retrieve comparison counts after search using getComparisons().

SortedArray Plans

1. Add Elements:
    - Call add(value) to insert the value while maintaining order.
    - Confirm with "Element added successfully."
2. Sort:
    - Call sort() to ensure sorting.
    - Display the sorted array and comparisons via getSortComparisons().
3. Search:
    - Use search(value) for binary search.
    - Show result (found or not) and comparisons via getComparisons().
4. Display:
    - Call display() to print array contents.
5. Clear:
    - Call clear() to reset the array.
    - Confirm with "Array cleared."
6. Performance Analysis:
    - Retrieve comparisons using getSortComparisons() or getComparisons() after operations.

SortedLinkedList Plans

1. Add Elements:
    - Call add(value) to insert at the correct position.
    - Confirm with "Element added successfully."
2. Sort:
    - Call sort() to apply merge sort.
    - Display sorted list and comparisons via getSortComparisons().

3. Search:
    - Use search(value) for linear search.
    - Show result and comparisons via getComparisons().
4. Display:
    - Call display() to print the list contents.
5. Clear:
    - Set head = null to reset.
    - Confirm with "List cleared."
6. Performance Analysis:
    - Retrieve and display comparison counts after operations.

LLNode Function Plans

1. Constructor (LLNode(T data)):
    - Initializes the node with data.
    - Sets the next reference to null.

Purpose: Create a new node with specified data.

2. Get Data (getData()):
    - Returns the data stored in the node.

Purpose: Access node content.

3. Set Data (setData(T data)):
    - Updates the data of the node.

Purpose: Modify node content.

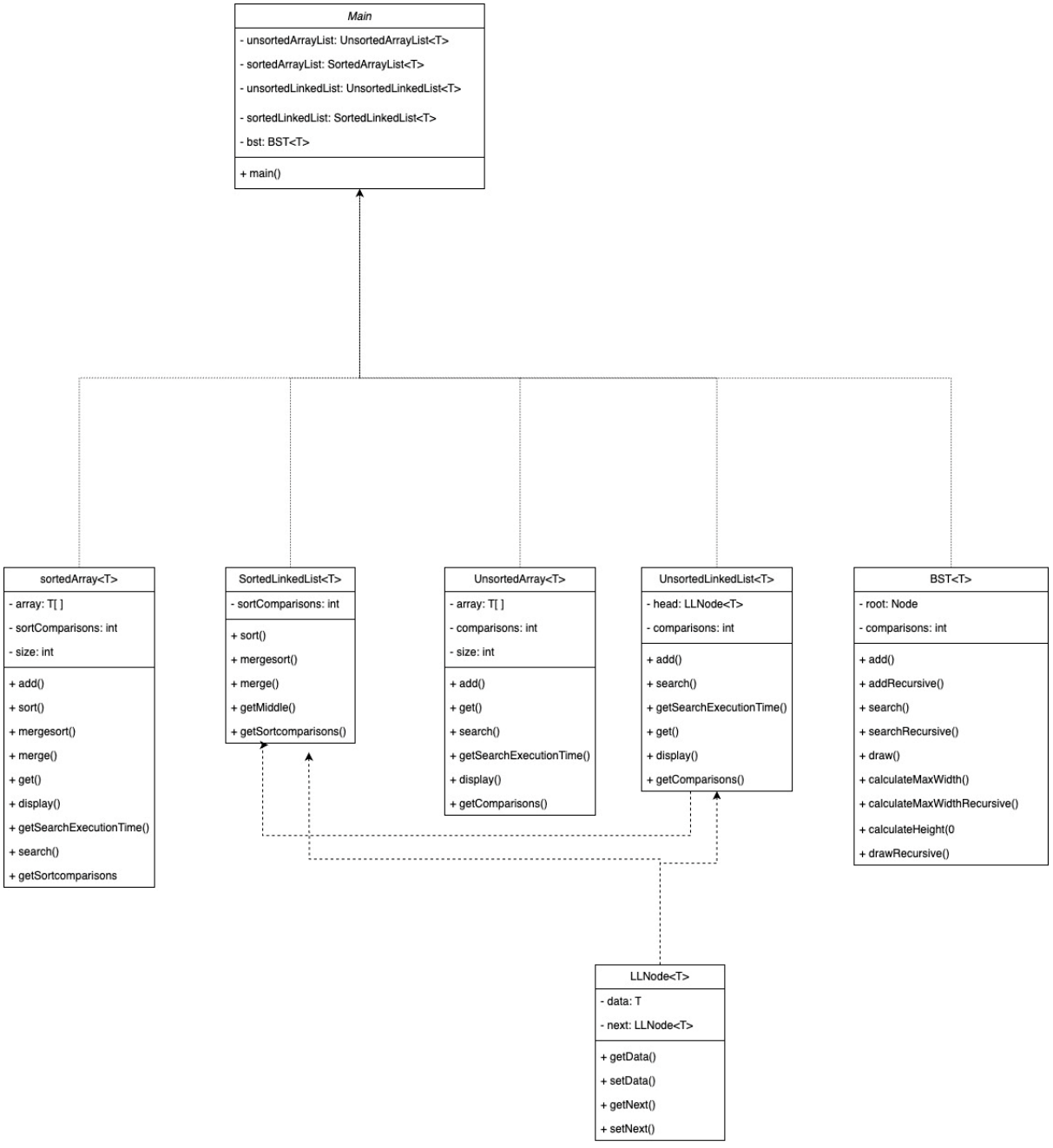
4. Get Next (getNext()):
    - Returns the reference to the next node.

Purpose: Traverse to the next node in the list.

5. Set Next (setNext(LLNode<T> next)):
    - Updates the reference to the next node.

Purpose: Link the current node to another node.

Design Documentation:
UML Diagram:



Flowchart:


start
Display Menu
User Chooses Option
Generate Unsorted Data
GenerateSortedData
Build and Display BST
Search and Performance Analysis
Generate Random Data for UL
Select List to Convert into BST
Populate Array UL and Display Data
Sort Array SL Using Selected Algorithm
Build BST from Selected list
Perform Search Using Optimal Search Algorithm
Populate Linked List UL and Display Data
Sort Linked List SL Using Selected Algorithm
Display BST Structure
Display Performance Results
Return to Menu?
YES
No
Exit Program

Pseudocode:

Main Application Pseudo-code:

```
START MainApplication
  DISPLAY Menu Options:
    1. Create Unsorted Lists (UL)
    2. Create Sorted Lists (SL)
    3. Build and Display BST
    4. Search Implementation
    5. Performance Analysis
    6. Exit

  REPEAT
    PROMPT User for Choice
    SWITCH (User Choice)
      CASE 1:
        CALL createUnsortedLists()
      CASE 2:
        CALL createSortedLists()
      CASE 3:
        CALL buildAndDisplayBST()
      CASE 4:
        CALL searchImplementation()
      CASE 5:
        CALL performanceAnalysis()
      CASE 6:
        EXIT Program
      DEFAULT:
        DISPLAY "Invalid Choice"
    END SWITCH
```

```
    UNTIL User chooses Exit
END MainApplication
```

Unsorted List Function:

```
FUNCTION createUnsortedLists()
    DECLARE unsortedArray, unsortedLinkedList
    DECLARE N = 512
    GENERATE N random integers
    STORE integers in unsortedArray and unsortedLinkedList
    DISPLAY "Unsorted Array List:"
    CALL displayList(unsortedArray)
    DISPLAY "Unsorted Linked List:"
    CALL displayList(unsortedLinkedList)
END FUNCTION
```

Sorted List Functions:

```
FUNCTION createSortedLists()
    DECLARE sortedArray, sortedLinkedList
    PROMPT user for sorting algorithm (merge, heap, quick)

    SWITCH (Sorting Algorithm)
        CASE "merge":
            CALL mergeSort(array/linked list)
        CASE "heap":
            CALL heapSort(array/linked list)
        CASE "quick":
            CALL quickSort(array/linked list)
        DEFAULT:
            DISPLAY "Invalid Sorting Option"
    END SWITCH
```

```
    DISPLAY "Sorted Array List:"
    CALL displayList(sortedArray)
    DISPLAY "Sorted Linked List:"
    CALL displayList(sortedLinkedList)
END FUNCTION
```

Binary Search Tree Function:

```
FUNCTION buildAndDisplayBST()
    PROMPT User to Select Source List (Unsorted/Sorted Array or Linked List)
    CONVERT Selected List into BST
    CALL displayBST(bst)
END FUNCTION

FUNCTION displayBST(bst)
    IF bst is Empty:
        DISPLAY "BST is empty."
    ELSE:
        DISPLAY BST Nodes Creatively (e.g., Tree Diagram)
END FUNCTION
```

Search Implementation:

```
FUNCTION searchImplementation()
    PROMPT user for search term
    DECLARE searchResults = []

    CALL searchUnsortedArray(term, unsortedArray)
    CALL searchUnsortedLinkedList(term, unsortedLinkedList)
    CALL searchSortedArray(term, sortedArray)
    CALL searchSortedLinkedList(term, sortedLinkedList)
    CALL searchBST(term, bst)
```

```
    DISPLAY Results for Each Data Structure
END FUNCTION
```

Performance Analysis:

```
FUNCTION performanceAnalysis()
    MEASURE and COUNT comparisons for:
        - Searching Unsorted Array
        - Searching Unsorted Linked List
        - Searching Sorted Array
        - Searching Sorted Linked List
        - Searching BST
    DISPLAY Comparisons and Big O Analysis
END FUNCTION
```

Helper Functions:

```
FUNCTION displayList(list)
    FOR each item in list (10-20 items per line):
        PRINT item
    END FOR
END FUNCTION

FUNCTION mergeSort(list)
    IMPLEMENT Merge Sort Algorithm
END FUNCTION

FUNCTION heapSort(list)
    IMPLEMENT Heap Sort Algorithm
END FUNCTION

FUNCTION quickSort(list)
    IMPLEMENT Quick Sort Algorithm
```

```
END FUNCTION

FUNCTION searchUnsortedArray(term, array)
    IMPLEMENT Linear Search
    RETURN Number of Comparisons
END FUNCTION

FUNCTION searchUnsortedLinkedList(term, linkedList)
    IMPLEMENT Linear Search
    RETURN Number of Comparisons
END FUNCTION

FUNCTION searchSortedArray(term, array)
    IMPLEMENT Binary Search
    RETURN Number of Comparisons
END FUNCTION

FUNCTION searchSortedLinkedList(term, linkedList)
    IMPLEMENT Linear Search (since Binary Search is impractical)
    RETURN Number of Comparisons
END FUNCTION

FUNCTION searchBST(term, bst)
    IMPLEMENT BST Search
    RETURN Number of Comparisons
END FUNCTION
```

Program Operation :

Operating the menu

1. Generate Unsorted Data

- Action: When the user clicks "Generate Unsorted Data":
    - The system generates 512 random integers.
    - These integers are added to the Unsorted Array, Unsorted Linked List, Sorted Array, and Sorted Linked List.
    - The Unsorted Array and Unsorted Linked List will contain the raw, unsorted values.
- Expected Result:
    - Unsorted Array: A list of random integers printed to the TextArea, in no particular order.
    - Unsorted Linked List: A list of random integers printed in the order they were inserted, also unsorted.

2. Generate Sorted Data

- Action: When the user clicks "Generate Sorted Data":
    - The Sorted Array and Sorted Linked List are sorted using their respective sorting algorithms.
- Expected Result:
    - Sorted Array: A list of the 512 integers sorted in ascending order.
    - Sorted Linked List: A list of the 512 integers sorted in ascending order.

3. Build and Display BST

- Action: When the user clicks "Build and Display BST":
    - The program constructs a Binary Search Tree (BST) using the first 20 values from the Unsorted Array.
    - The BST is drawn on a Canvas.
    - The height of the BST is calculated and displayed.
- Expected Result:
    - The program prints the number of layers (height) of the BST in the TextArea. The height is the number of levels or layers in the tree, indicating its depth.
    - The Canvas will visually represent the BST with nodes and edges.

    - If the tree is deep, it will show only the first few layers, and the program will indicate that additional layers exist but are not displayed to keep the tree visually manageable.

4. Search

- Action: When the user enters a value in the TextField and clicks "Search":
    - The program searches for the value in:
        - Unsorted Array
        - Unsorted Linked List
        - Sorted Array
        - Sorted Linked List
        - BST
- Expected Result:
    - For each data structure, the program will print whether the value was found or not found in the TextArea.
    - The program will perform the search and count the comparisons made during the search for each structure.
    - Example: If the value 500 is searched:
        - Unsorted Array: The program will check each element one by one until it finds 500 (linear search).
        - Unsorted Linked List: Similar to the array, it will traverse the list to find the value.
        - Sorted Array: The search might stop earlier if the value is greater than all elements in the list (binary search if implemented).
        - Sorted Linked List: Similar to the array but in linked list form.
        - BST: The search will be efficient, as the tree structure allows binary search.
- The number of comparisons for each structure is printed.

5. Performance Analysis

- Action: When the user clicks "Performance Analysis":
    - The program calculates the number of comparisons made during the most recent search for each data structure.
- Expected Result:
    - The program prints the number of comparisons for each structure in the TextArea.

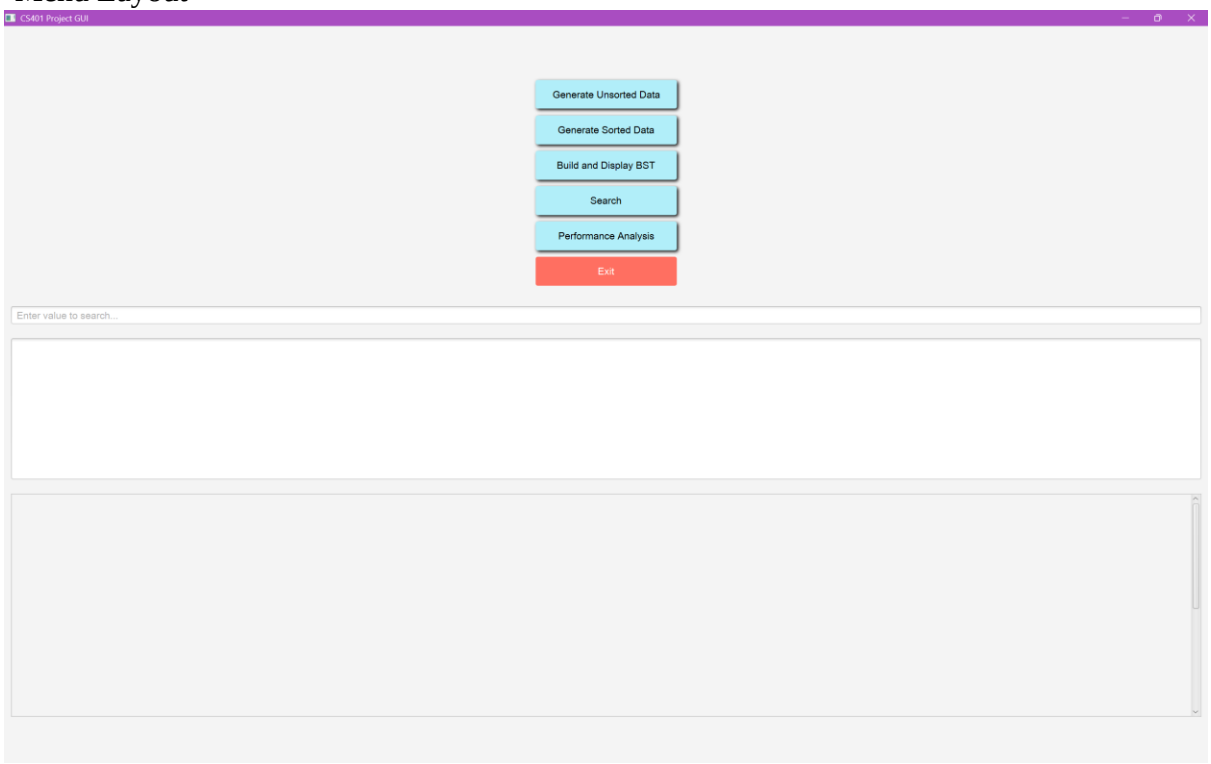
- Example:
    - Unsorted Array: It will show the total number of comparisons made during the last search (which would be the length of the list if the item is not found).
    - Unsorted Linked List: Similar to the array, it will show how many nodes were traversed.
    - Sorted Array: Fewer comparisons, as the search could terminate early if the value is larger than the largest value in the array.
    - Sorted Linked List: Similar to the sorted array but with the overhead of list traversal.
    - BST: Fewer comparisons due to the logarithmic depth of the tree.

6. Exit

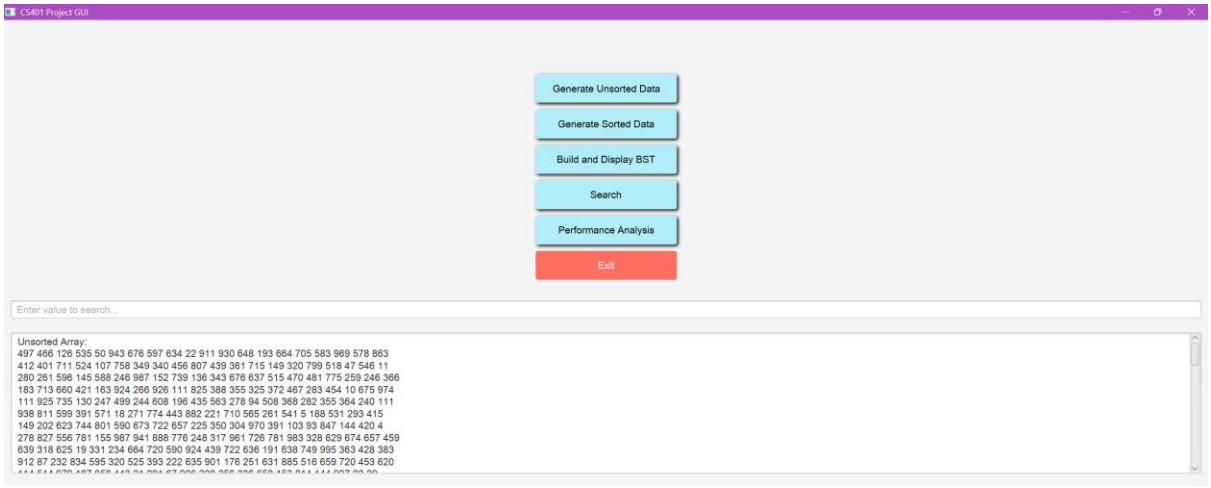
- Action: When the user clicks "Exit":
    - The program closes the window and ends the execution.
- Expected Result:
    - The application window will close, and the program will stop running.

Screenshots:

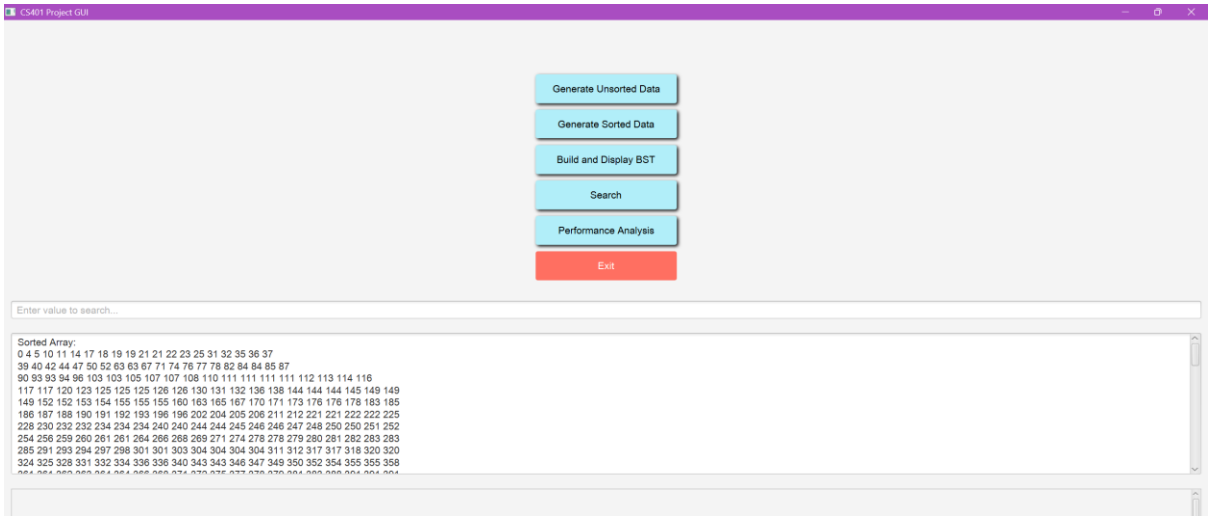
-Menu Layout



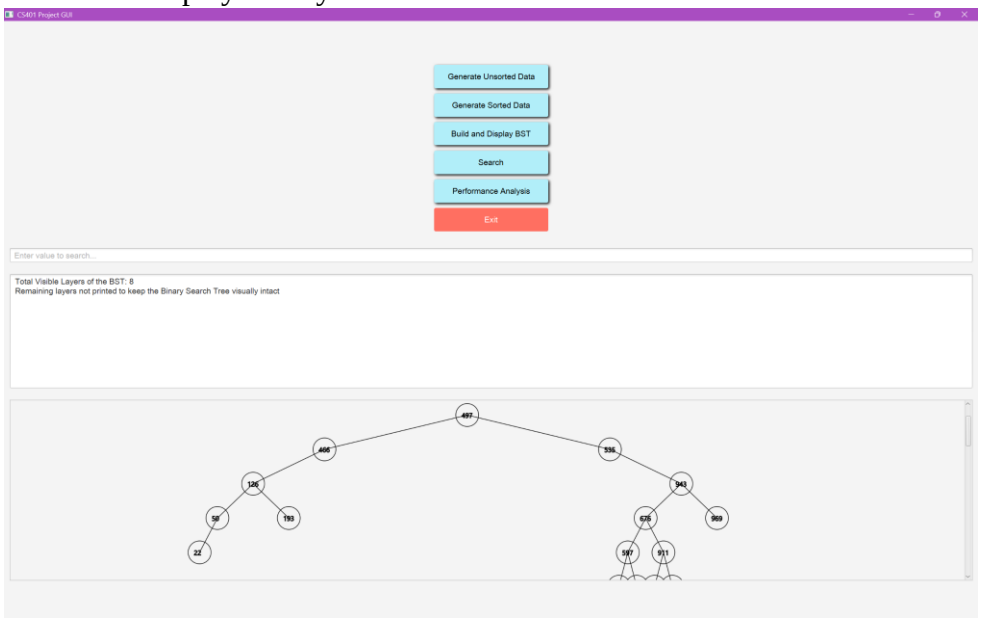
Unsorted Data Generated:



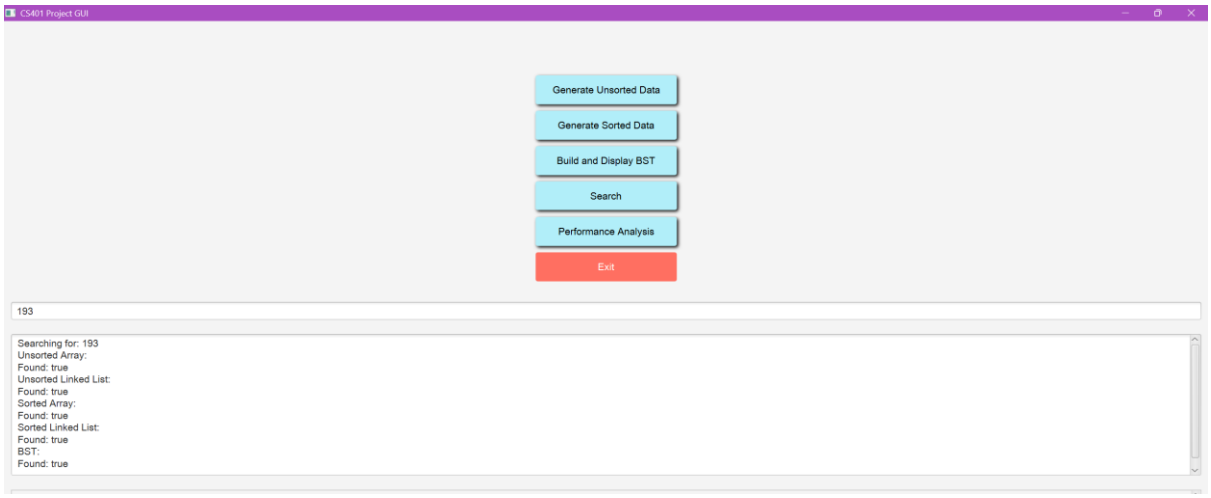
Sorted Data Generated:



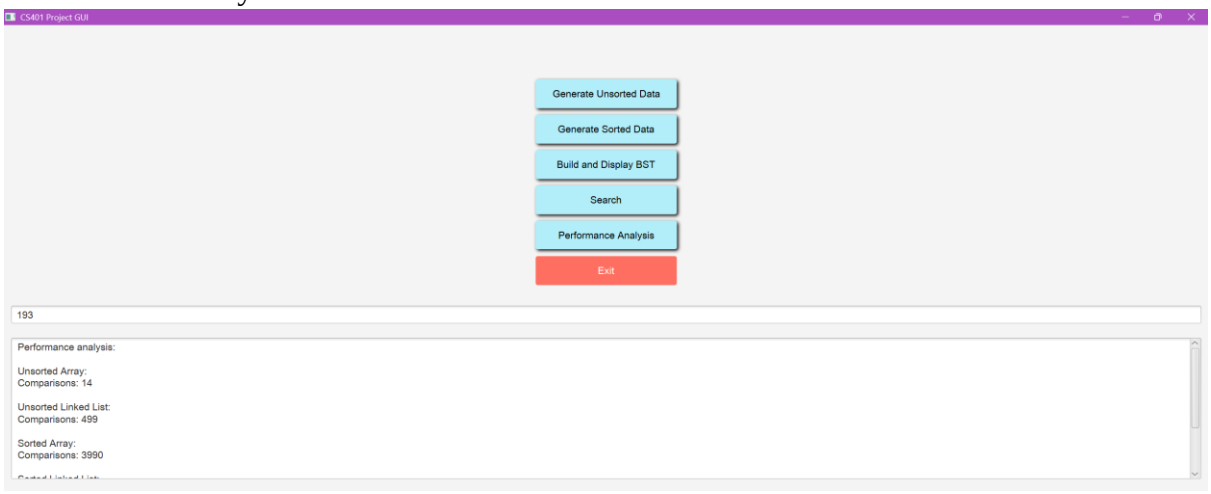
Build and Display Binary Search Tree:



Search:



Performance Analysis:



README:

Compilation Instructions

1. **Installing maven:**

   Install maven for windows and copy the path to the bin file in the system environment variables → path → add new path

2. **Creating the Project directory:**
   Create a folder containing pom.xml file and src folder. The src folder should contain main folder which should contain folder named java . Paste the .java files inside "java" folder.

Project directory structure:

src/

main/

java/ **← Contains the Java source files**

pom.xml

3. **Building from the Command Line**: Build the project from the command line, navigate to the project directory containing pom.xml file and run:

"mvn clean install"

This will compile the project, download the necessary dependencies, and package the project into a .jar file.

4. **Run the Program Using Maven**: Run the application from the command line, execute the following Maven command:

"mvn javafx:run"

This will execute the JavaFX application and open the GUI.

Project Schedule

Week 1: November 10 - November 16

Day 1 (Nov 10): Understanding Requirements and Planning

- Read and understand requirements (2 hours).
- Plan high-level design and class structures (1 hour).
- Create UML diagrams for data structures (2 hours).

Day 2-3 (Nov 11-12): Unsorted List (UL) Implementation

- Write pseudo-code for UL (1 hour).
- Implement UL using an array (3 hours).
- Implement UL using a linked list (3 hours).

Day 4 (Nov 13): Testing and Debugging UL

- Test both UL implementations with random data (3 hours).
- Add menu option to display UL (1 hour).

Day 5-6 (Nov 14-15): Sorted List (SL) Implementation

- Write pseudo-code for sorting algorithm (2 hours).
- Implement array-based SL with chosen sort algorithm (4 hours).

Day 7 (Nov 16): SL Testing and Debugging

- Implement linked list-based SL (3 hours).
- Test and debug both SL implementations (3 hours).

Week 2: November 17 - November 24

Day 8-9 (Nov 17-18): Binary Search Tree (BST) Implementation

- Write pseudo-code for BST (1 hour).
- Implement BST creation and methods (4 hours).
- Add visual display for BST (3 hours).

Day 10 (Nov 19): Search Implementation

- Implement search algorithms for UL, SL, and BST (4 hours).

Day 11 (Nov 20): Performance Analysis

- Write code to count comparisons during search operations (2 hours).
- Conduct experiments and collect performance data (3 hours).

Day 12-13 (Nov 21-22): Documentation

- Write complexity analysis and experimental results (3 hours).
- Prepare UML diagrams, flowcharts, and pseudo-code (4 hours).
- Write the user manual and README file with screenshots (3 hours).

Day 14 (Nov 23): Review and Finalize

- Review source code and inline comments (2 hours).
- Package files into a single ZIP for submission (1 hour).

Day 15 (Nov 24): GUI Implementation

- Design and implement GUI (5 hours).
- Test and debug GUI (3 hours).

Complexity Analysis:

Theoretical analysis:

1. Unsorted Array:

- Add Operation:
  Time complexity: O(1)
  Adding an element to the end of the array is a constant-time operation (assuming there's space in the array).
- Search Operation:
  Time complexity: O(n)
  The search operation requires scanning through each element in the array until the desired element is found.
- Display Operation:
  Time complexity: O(n)
  The entire array is traversed to display the elements.

2. Sorted Array:

- Add Operation:
  Time complexity: O(n)

Inserting an element into a sorted array involves shifting elements to make space for the new value, which takes linear time.

- Sort Operation (Merge Sort):
  Time complexity: O(n log n)
  Merge Sort is a divide-and-conquer algorithm that divides the array into subarrays and merges them back, with time complexity of O(n log n).
- Search Operation (Binary Search):
  Time complexity: O(log n)
  Binary Search works on sorted arrays by repeatedly dividing the search interval in half. It's logarithmic in complexity.
- Display Operation:
  Time complexity: O(n)
  Each element in the array must be displayed, which takes linear time.

3. Unsorted Linked List:

- Add Operation:
  Time complexity: O(1)
  Adding an element to the front of the linked list is a constant-time operation.
- Search Operation:
  Time complexity: O(n)
  Similar to the unsorted array, the search operation requires traversing the list, making it linear in time.
- Display Operation:
  Time complexity: O(n)
  The list must be traversed to display each element, which is linear in complexity.

4. Sorted Linked List:

- Add Operation:
  Time complexity: O(n)
  Inserting an element into a sorted linked list requires finding the correct position and inserting the element in the appropriate place, which takes linear time.
- Sort Operation (Merge Sort):
  Time complexity: O(n log n)
  Merge Sort on a linked list involves recursively splitting the list and merging it back together, with the time complexity being O(n log n).
- Search Operation (Merge Search):
  Time complexity: O(n)
  Searching in a sorted linked list still requires linear traversal of the list, despite it being sorted.
- Display Operation:
  Time complexity: O(n)
  As with the unsorted linked list, displaying all the elements takes linear time.

Operation	Unsorted Array	Sorted Array	Unsorted Linked List	Sorted Linked List
Add	**O(1)**	**O(n)**	**O(1)**	**O(n)**
Search	**O(n)**	**O(n log n)**	**O(n)**	**O(n)**
Sort	**O(n log n)**	**O(n log n)**	**O(n log n)**	**O(n log n)**
Display	**O(n)**	**O(n)**	**O(n)**	**O(n)**

Experimental Results Comparison :

Data Structure	Comparisons
Unsorted Array	**18**
Unsorted Linked List	**495**
Sorted Array	**3968**
Sorted Linked List	**21**
Binary Search Tree (BST)	**7**

Analysis:

1. **Unsorted Array:**
    - With only 18 comparisons, the Unsorted Array performs the least number of comparisons. This suggests that the search operation or the operation being measured (possibly a search or an initial sort) was very efficient.
2. **Unsorted Linked List:**
    - 495 comparisons indicate significantly more comparisons compared to the unsorted array. This shows that linked lists incur more comparisons during search operations because they must traverse node by node.
3. **Sorted Array:**
    - The Sorted Array has 3968 comparisons, which is much higher than both the unsorted array and the unsorted linked list. This suggests that the array may have been sorted multiple times or that the search comparisons were unusually high due to the nature of sorted data or the specific algorithm being used (e.g., a non-optimal sorting algorithm like bubble sort).

4. **Sorted Linked List:**
    - The Sorted Linked List requires only 21 comparisons, which is very efficient for its data structure. Despite being sorted, the number of comparisons is still higher than the unsorted array, but it benefits from sorting in terms of order of insertion.
5. **Binary Search Tree (BST):**
    - The BST requires only 7 comparisons, the least among all structures. This reflects the optimal search time due to the binary search property of BSTs, where each comparison effectively halves the search space.

Conclusion:

- BST shows the best performance with the least number of comparisons, thanks to its balanced structure and efficient search algorithm.
- Sorted Array has the highest number of comparisons, likely due to inefficiencies in sorting or searching, especially if it's not using optimal algorithms.
- Unsorted Array and Sorted Linked List provide relatively efficient performance, with the unsorted array having fewer comparisons due to direct access but not being optimized for search operations like BSTs.
- Unsorted Linked List requires significantly more comparisons than other structures, highlighting the inefficiencies of sequential access in linked lists.