CS 401 Final Project

A20586593 Seonghwan Lim

1. Software Specification

1.1. Common Methods (Implementation of CollectionInterface)

1) add(T element): Inserts data and returns success status.

2) remove(T target): Removes specific data and returns success status.

3) contains(T target): Checks whether the data is present in the collection.

4) get(T target): Returns specific data (returns null if not found).

5) size(): Returns the number of elements in the collection.

1.2. Features by Data Structure

1) UnsortedArray

- quickSort(): Sorts the array using quick sort.

- find(): Searches for data using linear search.

2) SortedArray

- add(): Inserts data while maintaining the sorted order.

- find(): Searches for data using binary search for faster results.

3) UnsortedLinkedList

- add(): Adds data to the end of the linked list.

4) SortedLinkedList

- add(): Maintains the sorted order during data insertion.

5) BinarySearchTree

- add(): Inserts data into the tree structure.

- remove(): Removes specific data from the tree.

- min() / max(): Returns the minimum or maximum value.

- buildBalancedTree(): Constructs a balanced binary search tree.

1.3. Performance Analysis

1) Allows users to compare contains operation performance across all data structures.

2) Reports the number of comparisons required for each data structure.

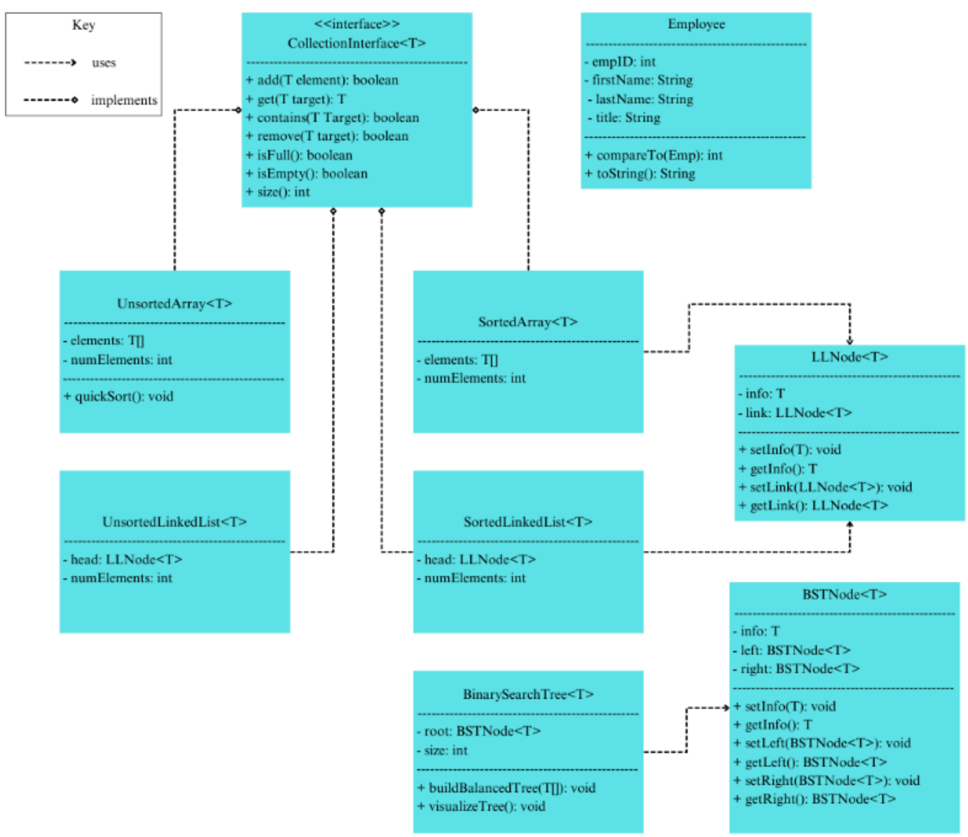
1.4. GUI Integration (Main.java)

1) loadCsvFile(): Reads data from a CSV file to create Employee objects.

2) createBinarySearchTree(): Constructs and visualizes a binary search tree using sorted data.

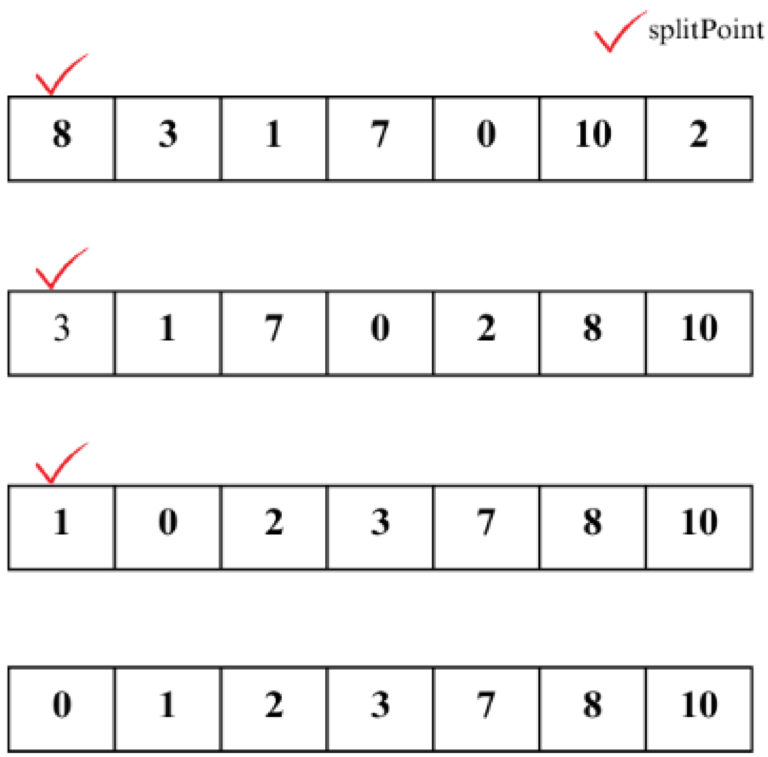
2. Design Documentation

2.1 UML diagrams



2.2. Flow charts

1) Quick sort



- Initial Array: [8, 3, 1, 7, 0, 10, 2] is divided using the split point 8.

    - Left: [3, 1, 7, 0, 2] (values smaller than the split point)

    - Right: [10] (values larger than the split point)

- First Split: The split point 8 is placed in its correct position.

- Sorting the Left Subarray:

    - Pivot 3 is selected, dividing the array into [1, 0, 2] and [7].

- Recursive Sorting:

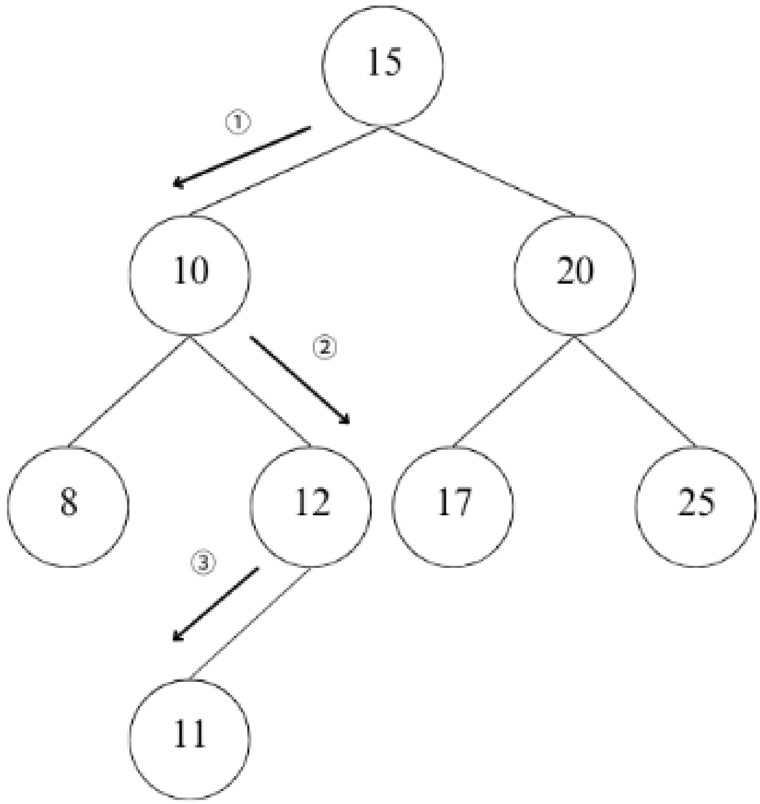
    - [1, 0, 2] is sorted using split point 1, resulting in [0, 1, 2].

- Result: All parts are sorted, producing [0, 1, 2, 3, 7, 8, 10].

- Quick Sort works through pivot selection → partitioning → recursive sorting, with an average time complexity of $\mathbf{O}(\boldsymbol{Nlog_2\,N})$.

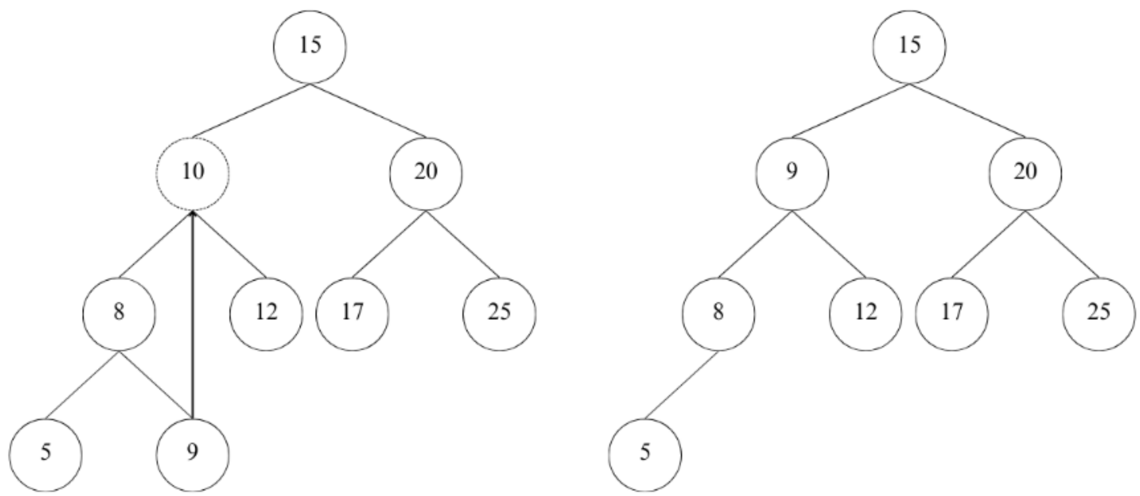
2) Binary Search Tree

① Situation 1: Add 11



- Start at Root: Compare 11 with 15. Since 11 <= 15, move to the left child (10).
- Compare with 10: Since 11 > 10, move to the right child (12).
- Compare with 12: Since 11 <= 12, insert 11 as the left child of 12.
- Resulting Tree:
    - Root: 15
    - Left Subtree: 10 → 8, 12 → 11
    - Right Subtree: 20 → 17, 25
- This maintains the Binary Search Tree's order: left <= parent < right.

② Situation 2: Remove 9



- Find 9: Start at 15, go left to 10, then to 8, and finally to 9.

- Remove 9: Since 9 is a leaf node (no children), simply remove it by setting the right child of 8 to null.

- Resulting Tree:

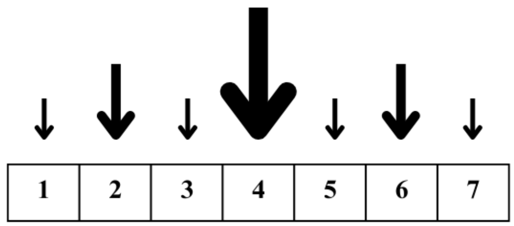
    - Root: 15

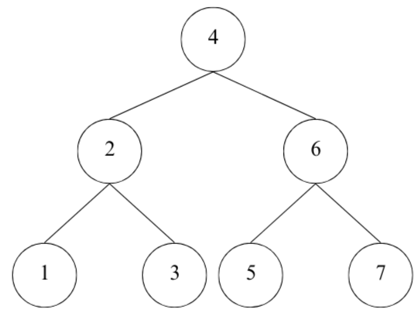
    - Left Subtree: 10 → 8 → 5, 12

    - Right Subtree: 20 → 17, 25

- This maintains the Binary Search Tree's order: left <= parent < right.

3) Balanced Tree





- Input: A sorted array [1, 2, 3, 4, 5, 6, 7].

- Choose the middle element of the array as the root:

    - Middle element: 4.

- Divide the array into two halves:

    - Left subarray: [1, 2, 3] → used for the left subtree.

    - Right subarray: [5, 6, 7] → used for the right subtree.

- Recursively repeat the process for each subarray:

    - Find the middle of each subarray to create child nodes.

    - Stop when the subarray has only one element (it becomes a leaf node).

3.3. Pseudo Code

1) Unsorted Array

Linear Search

```
FUNCTION linearSearch(array, target):
    FOR i FROM 0 TO size(array) - 1:
        IF array[i] == target:
            RETURN TRUE
    RETURN FALSE
```

Remove Element

```
FUNCTION removeUnsortedArray(array, target):
    FOR i FROM 0 TO size(array) - 1:
        IF array[i] == target:
            array[i] = array[size(array) - 1]
            size(array) -= 1
            RETURN TRUE
    RETURN FALSE
```

2) Sorted Array

Add Element

```
FUNCTION addSortedArray(array, element):
    index = 0
    WHILE index < size(array) AND array[index] < element:
        index++
    SHIFT elements from index to end one step right
    array[index] = element
```

Binary Search

```
FUNCTION binarySearch(array, target):
    low = 0, high = size(array) - 1
    WHILE low <= high:
        mid = (low + high) / 2
        IF array[mid] == target:
            RETURN TRUE
        ELSE IF array[mid] < target:
            low = mid + 1
        ELSE:
            high = mid - 1
    RETURN FALSE
```

3) Unsorted Linked List

Search Element

```
FUNCTION searchUnsortedList(head, target):
    current = head
    WHILE current IS NOT NULL:
        IF current.value == target:
            RETURN TRUE
        current = current.next
    RETURN FALSE
```

Remove Element

```
FUNCTION removeUnsortedList(head, target):
    current = head, previous = NULL
    WHILE current IS NOT NULL:
        IF current.value == target:
            IF previous IS NULL:
                head = current.next
            ELSE:
                previous.next = current.next
            RETURN TRUE
        previous = current
        current = current.next
    RETURN FALSE
```

4) Sorted Linked List

Add Element

```
FUNCTION addSortedList(head, element):
    newNode = Node(element)
    current = head, previous = NULL
    WHILE current IS NOT NULL AND current.value < element:
```

Remove Element

```
FUNCTION removeSortedList(head, target):
    current = head, previous = NULL
    WHILE current IS NOT NULL:
        IF current.value == target:
            IF previous IS NULL:
```

```
        previous = current
        current = current.next
    IF previous IS NULL:
        newNode.next = head
        head = newNode
    ELSE:
        previous.next = newNode
        newNode.next = current
```

```
                head = current.next
            ELSE:
                previous.next = current.next
            RETURN TRUE
        previous = current
        current = current.next
    RETURN FALSE
```

5) Binary Search Tree (BST)

Add Node

```
FUNCTION addNode(node, value):
    IF node IS NULL:
        RETURN new Node(value)
    IF value <= node.value:
        node.left = addNode(node.left, value)
    ELSE:
        node.right = addNode(node.right, value)
    RETURN node
```

Remove Node

```
FUNCTION removeNode(node, value):
    IF node IS NULL:
        RETURN NULL
    IF value < node.value:
        node.left = removeNode(node.left, value)
    ELSE IF value > node.value:
        node.right = removeNode(node.right, value)
    ELSE:
        IF node.left IS NULL:
            RETURN node.right
        ELSE IF node.right IS NULL:
            RETURN node.left
        predecessor = findMax(node.left)
        node.value = predecessor.value
        node.left = removeNode(node.left,
predecessor.value)
    RETURN node
```

3. User Manual

3.1. Program Operation Instructions

1) Run the Main.jar file from the command prompt.

2) The program, Data Structure GUI, will launch.

3) Use Load CSV to input the provided CSV file.

4) You can create data structures using the following options: Create Unsorted Array, Sorted Array, Unsorted Linked List, and Sorted Linked List.

5) In Unsorted Array, quick sort is available.

6) In Sorted Array, you can generate a Binary Search Tree.

7) Each data structure supports the following operations: add, remove, and contains.

8) Click Performance Analysis to perform a contains comparison.

3.2. Expected Results

1) Program Launch

- Running the Main.jar file launches the Data Structure GUI.

- The main menu displays with buttons like Load CSV, Create Data Structures, and Performance Analysis.

2) Load CSV

- Clicking Load CSV opens a file chooser dialog.

- Upon selecting a valid CSV file, an alert message displays: "CSV loaded successfully."

- The loaded employee data is now available for use in creating data structures.

3) Create Data Structures

- Clicking on any of the following buttons creates the respective data structure:

    - Unsorted Array: A tab displays the employee data in its original order.

    - Sorted Array: A tab displays the employee data in sorted order (by Employee ID).

    - Unsorted Linked List: Employee data is shown in the order it was loaded.

    - Sorted Linked List: Employee data is shown in sorted order.

4) Quick Sort (Unsorted Array)

- Clicking Quick Sort on the Unsorted Array tab creates a new tab with the sorted array.

- Expected message: "Quick Sort completed. Check the new tab for results."

5) Binary Search Tree (Sorted Array)

- Clicking Create Binary Search Tree on the Sorted Array tab generates a Balanced Binary Search Tree.

- A new tab displays the tree structure in table format, with parent-child relationships labeled (e.g., Root → Left).

6) Add, Remove, Contains

- Add: Adds a new employee to the selected data structure, and the table updates with the new record.

- Remove: Deletes a specific employee by ID, and the table removes the corresponding row.

- Contains: Displays a dialog box indicating whether the employee exists in the data structure: "Employee found." or "Employee not found."

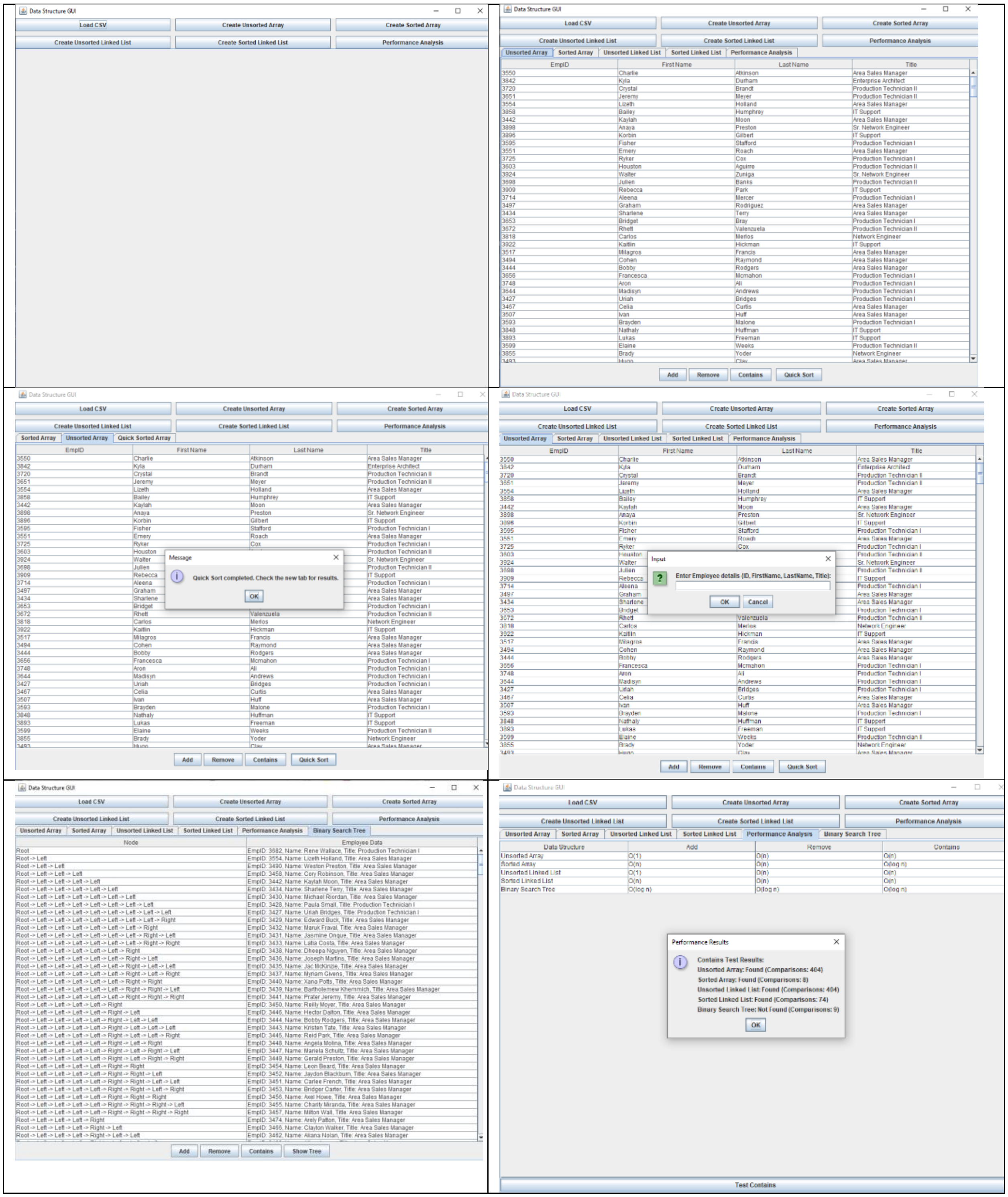
7) Performance Analysis

- Clicking Performance Analysis opens a tab with:

- A table showing the Big-O complexities of each data structure.

- The Test Contains button allows users to compare contains operations.

3.3. Screen Shots



4. Readme

4.1. Execution Instructions

Run the program with the following command:

"java -jar Main.jar"

4.2. Dependencies

1) Java Version: Requires JDK 8 or later.

2) GUI Environment: The program uses javax.swing for its graphical user interface (GUI). Ensure your runtime environment supports GUI applications.

3) Data Structure Size Limits: Array-based data structures (e.g., UnsortedArray, SortedArray) require a predefined size during initialization. The default size is 1024, but you can adjust it if needed.

4.3. Notes

1) CSV File Format: When loading a CSV file, ensure it follows this format:
EmpID,FirstName,LastName,Title

Each row should contain a unique ID, first name, last name, and job title for an employee.

2) Data Structure Creation: Use the buttons in the GUI to create data structures (e.g., arrays, linked lists, or binary search tree). Each structure is displayed in a separate tab.

3) Tab Operations: Within each tab, you can:

- Add elements

- Remove elements

- Check if an element exists (contains)

- Access additional features (e.g., Quick Sort, visualization) available for certain data structures.

4) Performance Analysis:

- Ensure all data structures are created before proceeding with performance testing.

- Use the Test Contains feature in the Performance Analysis tab to evaluate and compare data structures effectively.

5. Data Files

Extracted 512 records of empID, firstName, lastName, and title from the Employee/HR Dataset (All in One) and shuffled them randomly.

Employee/HR Dataset (All in One)

6. Project Schedule

6.1. Setup (Week 1)

1) Initial Environment Setup: Define project requirements, design the structure of data management system, and start coding the core classes (e.g., Employee, UnsortedArray, LinkedList).

6.2. Core Development (Week 2, 3)

1) Implement Core Data Structures: Develop unsorted/sorted arrays, linked lists, and binary search tree classes.

2) Create GUI: Develop the graphical user interface for data structure operations.

6.3. Testing and Debugging (Week 3)

1) Functional Testing: Test all operations (add, remove, contains, sort) for accuracy and performance.

2) Documentation and Handover: Finalize README, deployment scripts, and project handover.

7. Complexity Analysis

7.1. Theoretical analysis

Data Structure	Add	Remove	Contains
Unsorted Array	O(1)	O(N)	O(N)
Sorted Array	O(N)	O(N)	O($log_2 N$) (Binary Search)
Unsorted Linked List	O(1)	O(N)	O(N)
Sorted Linked List	O(N)	O(N)	O(N)
Binary Search Tree	O($log_2 N$) (Balanced)	O($log_2 N$) (Balanced)	O($log_2 N$) (Balanced)

7.2. Experimental results comparison (Contains)

1) Data ID Description

- ID 3550: The first ID of the unsorted data. It is the first element in an array or linked list, with a fixed position.

- ID 3800: The last ID of the unsorted data. It is the value located at the end of an array or linked list.

- ID 3427: The first ID of the sorted data. As the first element in a sorted data structure, it has the smallest value.

- ID: 3938: The last ID of the sorted data. In a sorted data structure, it has the largest value.

- ID: 3682: The median value of the sorted data. It is a value close to the middle position in the sorted data.

2) Experimental Results Data Comparison

Below is a summary table of the number of comparisons required to find the data for each ID.

Data Structure	ID 3550	ID 3800	ID 3427	ID 3938	ID 3682
Unsorted Array	1	512	30	480	107
Sorted Array	7	8	9	10	1
Unsorted Linked List	1	512	30	480	107
Sorted Linked List	124	374	1	512	256
Binary Search Tree	7	8	9	10	1

3) Experimental Results Analysis

- Unsorted Array and Linked List

- ID 3550: Has a fixed position, allowing direct access without any comparisons (1 comparison).

- ID 3800: Requires searching through to the end of the array/list, necessitating a number of comparisons equal to the total data size of 512.

- ID 3682: On average, approximately half of the data size needs to be compared, resulting in about 107 comparisons.

- Sorted Array

- Binary Search: Utilizes Binary Search for high search efficiency in sorted data structures.

- The number of comparisons is similar across all IDs, with the median value (ID 3682) requiring the fewest comparisons (1 comparison).

- Sorted Linked List

- Sequential Search: Searches by sequentially comparing nodes, leading to performance differences based on the position of the ID.

- ID 3427: As the first ID, it incurs a low search cost (1 comparison).

- ID 3938: Located at the end of the list, it incurs the highest search cost (512 comparisons).

- Binary Search Tree (BST):

- BSTs are sorted structures that typically exhibit $\mathbf{O}(\boldsymbol{log_2 N})$ performance on average.

- The number of comparisons is similar across all IDs, with the median value (ID 3682) requiring the fewest comparisons (1 comparison).

7.3. Results

* N = 512

Data Structure	Theoretical Complexity	Experimental Complexity	Comment
Unsorted Array	O(N) = 512	Best: 1 Worst: 512	Uses linear search; the number of operations varies significantly based on data position.
Sorted Array	O($log_2 N$) = 9	Best: 1 Worst: 10	Demonstrates the efficiency of binary search. Differences become more pronounced with larger data sizes.
Unsorted Linked List	O(N) = 512	Best: 1 Worst: 512	Uses linear search and shows the same results as an unsorted array.
Sorted Linked List	O(N) = 512	Best: 1 Worst: 512	Although sorted, it highlights the limitations of linear search in linked lists.
Binary Search Tree	O($log_2 N$) = 9	Best: 1 Worst: 10	Confirming the strengths of a balanced BST.

1) Unsorted Data: Arrays and linked lists are quick at searching for the first ID, but they incur high costs when searching for the last ID or a middle value due to linear search.

2) Sorted Data: Arrays are efficient with binary search, and Binary Search Trees (BSTs) exhibit similar performance. In contrast, sorted linked lists have low performance because they still rely on linear search.

3) Efficiency: The most efficient data structures are Binary Search Trees and Sorted Arrays. Linked lists show poor performance regardless of whether they are sorted.

4) Differences Between Theory and Practice in Binary Search:

- Effect of Integer Division: When calculating the mid index, integer division can truncate decimal points, potentially affecting the number of iterations required.

- Loop Condition: The condition low <= high can cause additional comparisons to occur even when the search is concluding.