Regent College | University Of Bolton

APR25 - SWE5202_0425 - Data Structures and Algorithms

Student Number: Jose Garrido HE23621

Module Name: Data Structures and Algorithms

Year/Semester: 2025 / Semester April 2025

Task 1 :

a. Explain the concepts of Time and Space Complexity

Time complexity is all about figuring out how an algorithm's runtime changes as the amount of data it has to work with gets bigger. Think of it less like timing a race with a stopwatch and more like counting the number of steps a runner takes. The actual time in seconds can change depending on the computer you're using, the programming language, or even what other programs are running in the background. Instead, we count the basic operations like comparisons, additions, or assignments that the algorithm has to do. This gives us a consistent way to judge and compare how efficient different algorithms are, no matter where or how they're run (Kleinberg & Tardos, 2005).

This kind of analysis is important because it helps us predict how an algorithm will behave as the problem gets larger. An algorithm that's lightning-fast with a small list of items might slow to a crawl when that list grows to a million items. This concept of how well an algorithm handles growing inputs is called scalability.

To talk about this growth rate in a standardized way, we use Asymptotic Notation. The most famous of these is Big O notation, written as O(·). Big O describes the worst-case scenario, giving us an upper limit on how much the runtime will grow. For example, an algorithm with a time complexity of O(n) is said to have linear growth. This means if you double the input size, the runtime will roughly double. On the other hand, an algorithm with O(n2) complexity has quadratic growth. If you double the input size for this one, the runtime will roughly quadruple , which can become very slow, very quickly (Sedgewick & Wayne, 2011). Understanding Big O helps us choose the most suitable algorithm for their needs, ensuring their applications remain efficient even under heavy loads.

Space Complexity is the measure of how much memory, or RAM, an algorithm needs to do its job. Just like with time complexity, we analyze it based on the size of the input, which we call n. This analysis is vital for understanding an algorithm's memory footprint, which is especially important for software running on devices with limited memory like smartphones or for systems that need to process enormous datasets.

When we talk about space, we break it down into two kinds. First is the Input Space, which is simply the memory needed to store the input data itself. For example, if you have an array of a million numbers, that's your input space. Second, and more importantly for analysis, is the Auxiliary Space. This is all the extra or temporary memory an algorithm uses while it's running (Kleinberg & Tardos, 2005). This could be new variables, data structures, or copies of the input it creates to work with.

Typically, when we analyze an algorithm's space complexity, they are most interested in the auxiliary space. The reason is that the input space is a fixed requirement; you can't get around needing space for the data you're given. The real difference in efficiency comes from how much additional memory an algorithm requires. An algorithm that can solve a problem in-place without much extra memory (using constant, or O(1), auxiliary space) is often preferred over one that needs to create a full copy of the input (using linear, or O(n), auxiliary space). Just like time complexity, we use Big O notation to describe this growth, providing a clear way to compare the memory efficiency of different approaches (Sedgewick & Wayne, 2011).

Time and space complexity often interact when evaluating algorithm efficiency. An algorithm may be fast but use more memory, or use less memory but take longer. For example, using extra space for caching can reduce computation time, trading space for speed. Conversely, in-place algorithms reduce memory usage but may require more steps to complete tasks. Understanding this balance is crucial when designing systems for real-world constraints, ensuring algorithms are not only fast but also fit within memory limits, which is essential for scalable and practical software solutions.

b.

I.

The time complexity of the java code is O(log n)

II.

The time complexity of the given Java code is O(log n) because the for loop in the code does not increment the variable x by a constant amount in each iteration.

Instead, it multiplies x by 2 in each step (x *= 2). This means that the value of x grows exponentially with each iteration. Initially, x is set to 1, and in each subsequent iteration, x becomes 2, 4, 8, and so on, until x becomes greater than n. The loop will therefore run as many times as it takes for x to reach or exceed n by repeatedly doubling, which is approximately the number of times you can double 1 before reaching n. This number is equivalent to the base-2 logarithm of n.

The loop's body contains only a single print statement, which takes constant time, the overall time complexity of the code is dominated by the number of iterations, which is logarithmic in n. Hence, the code has a time complexity of O(log n)

c.

I.

The time complexity of the java code is O(n)

II.

First, the function checks whether n is less than or equal to 1. If this condition is true, the function returns immediately, and no further processing occurs. This check takes constant time O(1). Next, the function declares two integer variables, i and j. This declaration also takes constant time and does not impact the overall growth of the runtime as n increases. The code consists of a nested loop, the outer loop runs from i = 1 to i = n, incrementing by 1 in each iteration. This means that regardless of the operations inside the loop, it will execute exactly n times. Inside this outer loop, there is another for loop with the variable j, intended to run from j = 1 to j = n. However, within the body of this inner loop, there is a break statement placed immediately after the print statement. This break statement causes the inner loop to terminate after only one iteration, regardless of the value of n. The inner loop runs only once each time because of the break, and since the outer loop runs n times, the print statement runs n times in total, so the time complexity is O(n).

Why: The outer loop runs from 1 to n (n iterations). The inner loop executes exactly **once per outer iteration** because it prints once and immediately hits `break`. That makes the inner work **O(1)** for each outer iteration, so total work is **n × O(1) = O(n)**.

Task II :

a. Stack Data Structure Java Implementation

I. MinStack.java Implementation

```
package Task2;
import java.util.*;

public class MinStack {
    // Main stack to store all elements
    private Stack<Integer> stack;
    // Auxiliary stack to keep track of minimum elements
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<>();    // Initialize main stack
        minStack = new Stack<>(); // Initialize min stack
    }

    public void push(int item) {
        stack.push(item); // Push item to main stack
        // If minStack is empty or new item is <= current min, push to minStack
        if (minStack.isEmpty() || item <= minStack.peek()) {
            minStack.push(item);
        }
    }

    public int pop() {
        int removed = stack.pop(); // Pop from main stack
        // If popped item is the current min, pop from minStack as well
        if (removed == minStack.peek()) {
            minStack.pop();
        }
        return removed; // Return popped item
    }

    public int peek() {
        return stack.peek(); // Return top element of main stack
    }

    public boolean is_empty() {
        return stack.isEmpty(); // Check if main stack is empty
    }

    public int get_min() {
        return minStack.peek(); // Return current minimum element
    }
}
```

II. MinStackTest.java Implementation

```
// MinStackTest.java
public class MinStackTest {
    public static void main(String[] args) {
        MinStack minStack = new MinStack();

        // Demonstrating push operations
        minStack.push(5);
        minStack.push(3);
        minStack.push(7);
        minStack.push(2);

        // Retrieving current minimum
        System.out.println("Current minimum: " + minStack.get_min()); //
Expected: 2

        // Peek top element
        System.out.println("Top element: " + minStack.peek()); // Expected:
2

        // Pop top element
        System.out.println("Popped element: " + minStack.pop()); //
Expected: 2
```

```
        // Check current minimum after pop
        System.out.println("Current minimum after pop: " +
minStack.get_min()); // Expected: 3

        // Check if stack is empty
        System.out.println("Is stack empty? " + minStack.is_empty()); //
Expected: false

        // Pop remaining elements to test is_empty
        minStack.pop();
        minStack.pop();
        minStack.pop();

        System.out.println("Is stack empty after removing all elements? " +
minStack.is_empty()); // Expected: true
    }
}
```

III. MinStackTest.java Running Output

```
Current minimum: 2
Top element: 2
Popped element: 2
Current minimum after pop: 3
Is stack empty? false
Is stack empty after removing all elements? true
```

The output demonstrates the correct functionality of the implemented stack and its get_min operation at each stage.

Initially, after pushing the values 5, 3, 7, and 2 onto the stack, the get_min method correctly identifies 2 as the current minimum value. The peek method then confirms that 2 is the top element of the stack, reflecting the last value pushed. When the pop method is called, it removes and returns 2, which was at the top of the stack.

Following this removal, get_min accurately updates the current minimum to 3, reflecting the next smallest value remaining in the stack. The is_empty method confirms that the stack still contains elements at this stage.

Finally, after removing all remaining elements, is_empty returns true, indicating that the stack is now empty. These outputs collectively confirm that the stack operations and the get_min feature function correctly and efficiently throughout different stages of use.

Shallow vs Deep Copy; Comparable vs Comparator

Shallow vs Deep Copy. The copying process creates duplicate references that lead to identical base objects, so modifications made through one reference become accessible through the other reference. The deep copy operation creates new objects (such as new nodes) which duplicate the content so that the copied objects exist independently from the originals. Linked structures require deep copy operations because they involve making new nodes while restructuring links instead of just duplicating head references.

Comparable vs Comparator. The `Comparable<T>` interface should be implemented when your class has one intrinsic ordering, such as student grades. A class should use `Comparator<T>` when developers need different sorting options which do not modify the original class structure. The Comparable interface exists inside the class while Comparators operate as external strategies which you can change depending on your use case requirements.

Task III :

a.

The SinglyLinkedList class manages a list of integers using nodes, where each node holds a data value and a reference to the next node. It allows inserting elements at the front using insert_front, which creates a new node and updates the head, or at the end using insert_end, which traverses to the last node and attaches the new node. The delete method removes the first node containing a specified value by adjusting the next references to bypass the target node. The display method traverses the list from head to end, printing each node’s data, showing the list’s current contents.

```
package Task3;

public class SinglyLinkedList {
    // Inner class representing a node in the linked list
    class Node {
        int data;      // Data stored in the node
        Node next;     // Reference to the next node

        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    private Node head; // Reference to the head (first node) of the list

    // Insert a new node at the front of the list
    public void insert_front(int value) {
        Node newNode = new Node(value); // Create new node
        newNode.next = head;            // Point new node to current head
        head = newNode;                 // Update head to new node
    }

    // Insert a new node at the end of the list
    public void insert_end(int value) {
        Node newNode = new Node(value); // Create new node
        if (head == null) {             // If list is empty
            head = newNode;             // Set head to new node
            return;
        }
        Node temp = head;
        while (temp.next != null) {     // Traverse to the last node
            temp = temp.next;
        }
        temp.next = newNode;            // Link last node to new node
    }

    // Delete the first node with the specified value
    public void delete(int value) {
        if (head == null) return;       // If list is empty, do nothing
        if (head.data == value) {       // If head needs to be deleted
            head = head.next;           // Move head to next node
            return;
        }
        Node temp = head;
        // Traverse to the node before the one to delete
        while (temp.next != null && temp.next.data != value) {
            temp = temp.next;
        }
        if (temp.next != null) {        // If node to delete is found
            temp.next = temp.next.next; // Bypass the node to delete
        }
    }

    // Display all elements in the list
    public void display() {
        Node temp = head;
        while (temp != null) {          // Traverse the list
            System.out.print(temp.data + " "); // Print node data
            temp = temp.next;           // Move to next node
        }
        System.out.println();           // Newline after printing list
    }
}
```

b.

The DoublyLinkedList class manages integers using nodes that store a value along with references to both the previous and next nodes, allowing bidirectional traversal. The insert_front method creates a new node at the front, updating the head and previous links. The insert_end method appends a new node at the end by traversing to the last node and updating the next and prev references accordingly. The delete method removes the first node containing a specified value, correctly adjusting neighboring next and prev pointers to maintain list integrity. The display method traverses from head to end, printing all node values sequentially

```
public class DoublyLinkedList {
    // Inner class representing a node in the doubly linked list
    class Node {
        int data;      // Value stored in the node
        Node prev, next; // References to previous and next nodes

        Node(int data) {
            this.data = data;
        }
    }

    private Node head; // Reference to the head (first node) of the list

    // Insert a new node at the front of the list
    public void insert_front(int value) {
        Node newNode = new Node(value); // Create new node
        newNode.next = head;            // New node points to current head
        if (head != null)
            head.prev = newNode;        // Update previous head's prev to new node
        head = newNode;                 // Update head to new node
    }

    // Insert a new node at the end of the list
    public void insert_end(int value) {
        Node newNode = new Node(value); // Create new node
        if (head == null) {             // If list is empty
            head = newNode;             // New node becomes head
            return;
        }
        Node temp = head;
        while (temp.next != null) {     // Traverse to the last node
            temp = temp.next;
        }
        temp.next = newNode;            // Last node's next points to new
node
        newNode.prev = temp;            // New node's prev points to last
node
    }

    // Delete the first node with the given value
    public void delete(int value) {
        Node temp = head;
        // Find the node with the given value
        while (temp != null && temp.data != value) {
            temp = temp.next;
        }
        if (temp == null) return;       // Value not found, do nothing
        if (temp.prev != null)
            temp.prev.next = temp.next; // Bypass temp in the list
        else
            head = temp.next;           // If deleting head, update head
        if (temp.next != null)
            temp.next.prev = temp.prev; // Update next node's prev
    }

    // Display the list from head to end
    public void display() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " "); // Print current node's data
            temp = temp.next;                  // Move to next node
        }
        System.out.println();
    }
}
```

c.

The BinarySearchTree class organizes integers using a tree structure where each node contains a value and references to left and right children, maintaining the property that left children hold smaller values and right children hold larger values. The insert method adds a value recursively at its correct position. The search method checks if a value exists by traversing left or right based on comparisons. The delete method removes a specified value, handling cases with zero, one, or two children while maintaining structure by replacing it with the in-order successor. The in_order_traversal method prints all values in sorted ascending order.

```java
public class BinarySearchTree {
    // Inner class representing a node in the BST
    class TreeNode {
        int data;
        TreeNode left, right;
        TreeNode(int data) { this.data = data; }
    }

    private TreeNode root; // Root node of the BST

    // Public method to insert a value into the BST
    public void insert(int value) {
        root = insertRec(root, value);
    }

    // Helper method for recursive insertion
    private TreeNode insertRec(TreeNode node, int value) {
        if (node == null) return new TreeNode(value); // Insert new node if current is null
        if (value < node.data) node.left = insertRec(node.left, value); // Go left
        else if (value > node.data) node.right = insertRec(node.right, value); // Go right
        return node; // Return the (possibly updated) node
    }

    // Public method to search for a value in the BST
    public boolean search(int value) {
        return searchRec(root, value);
    }

    // Helper method for recursive search
    private boolean searchRec(TreeNode node, int value) {
        if (node == null) return false; // Value not found
        if (node.data == value) return true; // Value found
        // Search left or right subtree based on comparison
        return value < node.data ? searchRec(node.left, value) : searchRec(node.right, value);
    }
    // Public method to delete a value from the BST
    public void delete(int value) {
        root = deleteRec(root, value);
    }

    // Helper method for recursive deletion
    private TreeNode deleteRec(TreeNode node, int value) {
        if (node == null) return null; // Value not found
        if (value < node.data) node.left = deleteRec(node.left, value); //
Go left
        else if (value > node.data) node.right = deleteRec(node.right,
value); // Go right
        else {
            // Node with only one child or no child
            if (node.left == null) return node.right;
            else if (node.right == null) return node.left;
            // Node with two children: get the inorder successor (smallest
in the right subtree)
            node.data = minValue(node.right);
            // Delete the inorder successor
            node.right = deleteRec(node.right, node.data);
        }
        return node; // Return the (possibly updated) node
    }

    // Helper method to find the minimum value in a subtree
    private int minValue(TreeNode node) {
        int minVal = node.data;
        while (node.left != null) {
            node = node.left;
            minVal = node.data;
        }
        return minVal;
    }

    // Public method to perform in-order traversal of the BST
    public void in_order_traversal() {
        inOrderRec(root);
        System.out.println();
    }

    // Helper method for recursive in-order traversal
    private void inOrderRec(TreeNode node) {
        if (node != null) {
            inOrderRec(node.left); // Visit left subtree
            System.out.print(node.data + " "); // Print node data
            inOrderRec(node.right); // Visit right subtree
        }
    }
}
```

```
public class AdjacencyMatrixGraph {
    private int[][] adjMatrix;
    private int numVertices;

    public AdjacencyMatrixGraph(int numVertices) {
        this.numVertices = numVertices;
        adjMatrix = new int[numVertices][numVertices];
    }

    public void addEdge(int src, int dest) {
        adjMatrix[src][dest] = 1;
        adjMatrix[dest][src] = 1; // since undirected
    }

    public void printMatrix() {
        System.out.println("Adjacency Matrix:");
        for (int i = 0; i < numVertices; i++) {
            for (int j = 0; j < numVertices; j++) {
                System.out.print(adjMatrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

```
Printing all keys and their corresponding values:
Ines : 60
Paulo : 70
Jose : 50
Parita : 80
```

The matrix is symmetric because the graph is undirected.

Test Classes :

The Task3Test class thoroughly tests SinglyLinkedList, DoublyLinkedList, and BinarySearchTree implementations to verify correctness. It performs various insertions at the front and end, deletions of specific values, and displays the lists to confirm updates. For the BinarySearchTree, it inserts multiple nodes, performs in-order traversal to display sorted order, searches for existing and non-existing values, and deletes a node before displaying the updated structure. Finally, it tests an AdjacencyMatrixGraph by adding edges and printing its matrix to visualize connections. This structured testing ensures that insertions, deletions, and searches across all structures function correctly.

```java
public class Task3Test {
    public static void main(String[] args) {
        // SinglyLinkedList test
        SinglyLinkedList sll = new SinglyLinkedList(); // Create a new singly linked list
        sll.insert_front(10); // Insert 10 at the front
        sll.insert_end(20);   // Insert 20 at the end
        sll.insert_front(5);  // Insert 5 at the front
        sll.display();        // Display the list
        sll.delete(10);       // Delete node with value 10
        sll.display();        // Display the list again

        // DoublyLinkedList test
        DoublyLinkedList dll = new DoublyLinkedList(); // Create a new doubly linked list
        dll.insert_front(30); // Insert 30 at the front
        dll.insert_end(40);   // Insert 40 at the end
        dll.insert_front(25); // Insert 25 at the front
        dll.display();        // Display the list
        dll.delete(40);       // Delete node with value 40
        dll.display();        // Display the list again

        // BinarySearchTree test
        BinarySearchTree bst = new BinarySearchTree(); // Create a new binary search tree
        bst.insert(50); // Insert 50 as root
        bst.insert(30); // Insert 30
        bst.insert(70); // Insert 70
        bst.insert(20); // Insert 20
        bst.insert(40); // Insert 40
        bst.insert(60); // Insert 60
        bst.insert(80); // Insert 80
        System.out.print("In-order traversal: ");
        bst.in_order_traversal(); // Print in-order traversal
        System.out.println("Search 40: " + bst.search(40));   // Search for 40
        System.out.println("Search 100: " + bst.search(100)); // Search for 100
        bst.delete(30); // Delete node with value 30
        System.out.print("In-order traversal after deleting 30: ");
        bst.in_order_traversal(); // Print in-order traversal again

        // AdjacencyMatrixGraph test for provided 0-1-2-3 graph
        AdjacencyMatrixGraph graph = new AdjacencyMatrixGraph(4); // Create
a graph with 4 vertices
        graph.addEdge(0, 1); // Add edge between 0 and 1
        graph.addEdge(1, 2); // Add edge between 1 and 2
        graph.addEdge(0, 3); // Add edge between 0 and 3
        graph.printMatrix(); // Print adjacency matrix
    }
}
```

Output:

```
5 10 20
5 20
25 30 40
25 30
In-order traversal: 20 30 40 50 60 70 80
Search 40: true
Search 100: false
In-order traversal after deleting 30: 20 40 50 60 70 80
Adjacency Matrix:
0 1 0 1
1 0 1 0
0 1 0 0
1 0 0 0
```

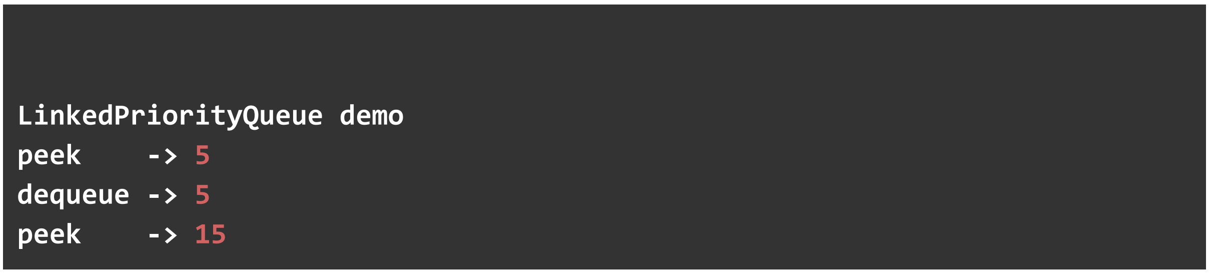
LO3 Addendum: Queues & Priority Queues (Linked)

Linked Queue (FIFO). The implementation uses a singly linked list structure with head (dequeue) and tail (enqueue). The enqueue operation adds elements at the tail (O(1)) and dequeue removes elements from the head (O(1)) and peek reads the head (O(1)). The implementation includes checks for empty conditions and resets the tail pointer when the last node is dequeued.

Linked Priority Queue. The data structure uses a sorted singly linked list with ascending order. The enqueue operation determines the insertion position (O(n)) but peek and dequeue access the head in O(1) time to maintain the minimum value at the front and support duplicate values.
The console displays the Queue and Priority Queue outputs as evidence.

Output:

```
LinkedQueue demo
dequeue -> 1
peek    -> 2
```



```
LinkedPriorityQueue demo
peek    -> 5
dequeue -> 5
peek    -> 15
```

Task IV :

I.

The MapExample class in Java is a straightforward demonstration of how to use a HashMap to store, retrieve, and manage data in the form of key-value pairs. In this specific example, the program uses the names of students as the keys and their respective marks as the values, allowing us to efficiently track and manipulate student marks without requiring complex data structures. Inside the main method, it begins by creating a HashMap instance named students. Using the put() method, it inserts entries like "ram" with 50, and other student names with their corresponding marks, storing them internally in a hashed structure that allows for quick access later. The System.out.println(students) prints out the entire map, giving a snapshot of the stored data. It then demonstrates removal with remove("raju"), showing how to delete a specific student from the dataset. The method containsKey() checks for the existence of a key, ensuring we can confirm if a student's record is present before we try to retrieve it. Lastly, using keySet() and a for-each loop, it prints out each student and their mark, illustrating clear usage of HashMap methods for practical applications in Java programming and for learning efficient data management.

II.

Inserting a key-value pair into a HashMap in Java is both straightforward and highly efficient, making it a practical tool for handling data that requires fast retrieval and insertion. This process uses the put(key, value) method, where the key is what you will later use to access the data, and the value is the data you wish to store. Internally, the HashMap uses the hashCode() of the key to determine the appropriate bucket to place the entry, ensuring quick access while minimizing the likelihood of collisions. If a collision does occur, meaning another key has the same hash, Java handles it internally through chaining or open addressing. If you use put() again with the same key but a different value, the HashMap will replace the old value with the new one, updating the entry efficiently. This feature is particularly useful when updating data such as changing a student’s mark. Using HashMaps in this manner reduces the complexity of searching and storing data while making your code cleaner, faster, and easier to manage.

III.

To retrieve the value associated with a specific key in a HashMap, you use the get(key) method provided by the Java HashMap class. This method takes the key as an argument and returns the corresponding value if it exists within the map. If the key does not exist in the map, the method will return null, which you can handle with conditional checks to prevent errors in your program flow. In the code, if you want to retrieve the marks for the student "kumar", you can use: System.out.println(students.get("kumar")); This line will print the value 80 to the console since "kumar" has been associated with that value in the map. Internally, the HashMap calculates the hash code of the key "kumar" to quickly locate and return the associated value without scanning every element in the map, which makes retrieval operations extremely efficient. Using get() is particularly useful when you need to access and utilize specific information stored in your data structure, such as fetching marks for processing results, generating reports, or displaying student information dynamically

IV.

The remove(key) method in a HashMap serves the important role of allowing you to delete a specific key-value pair from the map based on the key you provide. When you call this method, it removes the mapping for the specified key if it exists and returns the value that was previously associated with the key. If the key does not exist, it will return null, which can help you determine whether the removal was successful or if the key was not present in the first place. In the code : System.out.println(students.remove("raju")); This line removes the student "raju" and prints out 70, which was the mark associated with "raju" before removal. The remove() method is particularly useful when managing dynamic data structures where you need to delete entries as data changes, such as when a student withdraws from a course or when cleaning up outdated entries to maintain accuracy and reduce memory usage. Using remove() helps maintain the integrity and relevance of your data, ensures your dataset only contains active, meaningful entries, and provides a clear, readable way to manage data removal operations, especially during data cleanup tasks or dynamic updates.

V.

The containsKey(key) method in Java's HashMap is used to check if a particular key is present in the map. It returns true if the key exists and false if it does not. This method is essential when you need to confirm the existence of a key before performing operations such as retrieving its value with get() to prevent null values or before updating or removing entries, ensuring you only interact with valid, present data within your map. In the code: boolean checkkey = students.containsKey("ram"); System.out.println(checkkey); This checks if "ram" is one of the keys in the students map and prints true since "ram" was added earlier using put(). The role of containsKey here is to verify the existence of "ram" before potentially performing further operations, such as retrieving the marks or updating the record, ensuring that your program logic handles cases gracefully when the key may not exist. Using containsKey improves program reliability, prevents unnecessary errors, and makes the code more readable and logical by explicitly handling the presence check before using sensitive operations like get() or remove() on your data within Java applications.

B. Java Program to implement ArrayList with Iterator.

```
import java.util.ArrayList;
```

```java
import java.util.Iterator;

import java.util.List;
```

```java
/**
 * Prints only the integers < 50 from an ArrayList using an Iterator.
 *
 * Time complexity:
 *  - Building the list below is O(n)
 *  - Single pass with Iterator is O(n)
 * Space complexity: O(n) for the list.
 */
public class ArrayListIteratorExample {
    public static void main(String[] args) {
        // Example data set. Replace with your own values if desired.
        List<Integer> nums = new ArrayList<>();
        for (int i = 0; i < 100; i += 7) {  // 0, 7, 14, ..., 98
            nums.add(i);
        }

        // REQUIRED: Use an Iterator to traverse and print only values < 50
        Iterator<Integer> it = nums.iterator();
        while (it.hasNext()) {                  // O(n) traversal
            int v = it.next();                  // O(1)
            if (v < 50) {                       // filter condition
                System.out.println(v);          // prints: 0 7 14 21 28 35 42
49
            }
        }
    }
}
```

Output:

```
0
7
14
21
28
35
42
49
```

C. Time complexity of list-based Map

In a List-based Map implementation, such as one that uses an ArrayList or LinkedList to store key-value pairs without hashing, the operations put(item), get(item), and remove(item) exhibit a time complexity of O(n) in the worst case. This is because each of these operations requires searching through the list sequentially to locate the key associated with the desired operation. When using put(item), the implementation typically checks whether the key already exists in the list before deciding whether to update the existing value or insert a new key-value pair. This check requires iterating through the list to find a matching key, leading to a linear scan if the key is located at the end or not present at all. Similarly, the get(item) operation requires traversing the list to locate the key in order to retrieve the corresponding value. If the key is at the end of the list or absent, the search must check every entry, resulting in O(n) time. For the remove(item) operation, the implementation also needs to search for the key before removing the key-value pair from the list. This necessitates a linear traversal to find the target key, leading to a worst-case time complexity of O(n). In summary, List-based Map implementations are less efficient for large datasets because they require a linear search for key lookups, insertions, and deletions. This contrasts with HashMap, which typically achieves O(1) average time complexity for these operations due to its underlying hash-based indexing, allowing for faster data management and retrieval in practice. (Cormen et al., 2009; Oracle Java Documentation).

```java
import java.util.HashMap;
import java.util.Map;

public class PrintHashMapExample {
    public static void main(String[] args) {
        // Create a HashMap to store student names and their scores
        Map<String, Integer> students = new HashMap<>();
        // Add student names and scores to the map
        students.put("Jose", 50);
        students.put("Ines", 60);
        students.put("Paulo", 70);
        students.put("Parita", 80);

        // Print a header message
        System.out.println("Printing all keys and their corresponding 
values:");
        // Iterate over the keys of the map and print each key-value pair
        for (String key : students.keySet()) {
            System.out.println(key + " : " + students.get(key));
        }
    }
}
```

Enums with Maps. An `enum` gives a fixed, type-safe set of values (e.g., `Grade {A,B,C,D,F}`). Using enums as Map keys or values improves readability and prevents invalid states, while keeping lookups and updates identical to standard object keys.

How a Hash Table operates.

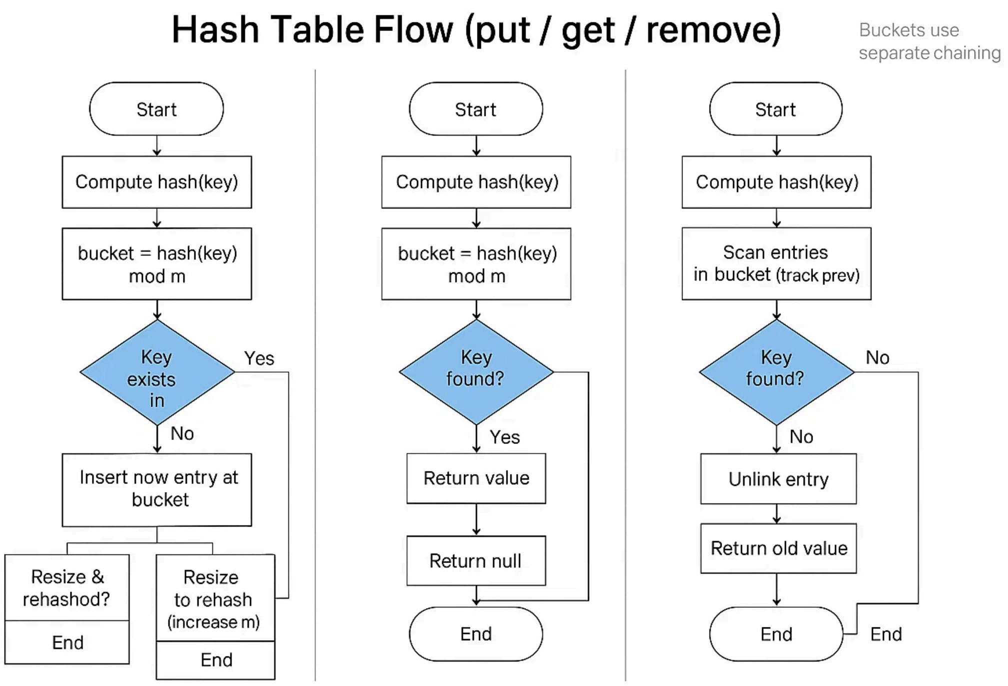
A hash table maps a key to a bucket index using a hash function; all operations begin by computing the hash and locating the bucket. Within the chosen bucket, the table either **updates** an existing entry (same key) or **traverses** entries to find the target key. With a suitable hash function and a controlled load factor, the **expected time** for `put`, `get`, and `remove` is **O(1)**; under heavy collisions, the **worst case** can degrade toward **O(n)** when many keys land in the same bucket. (Task 4 requires an **exclusive flow chart** for the hash table.)

Figure X — Hash Table operations (put / get / remove). Each flow starts by computing the key's hash and selecting `bucket = hash(key) mod m`.

- **put(key, value):** go to bucket → if key exists **update value** → else **insert new entry** (and resize when load factor exceeds a threshold).

- **get(key):** go to bucket → scan entries for key → **return value** if found, else **null**.

- **remove(key):** go to bucket → scan entries for key → **unlink entry** and **return old value** if found, else **null**.
  Expected time is **O(1)** per operation; worst-case is **O(n)** if many keys collide in one bucket.



Expected time *O(!)*with good hashing and load factor; worst case *O(h)* under heavy collisions.

Skip Lists (overview).

A skip list augments a sorted linked list with extra “levels” that skip over multiple nodes. Search starts at the highest level, moves forward while the next key ≤ target, and drops down a level when it would overshoot. With a geometric distribution of levels, the **expected height is O(log n)** and the **expected** time for search/insert/delete is **O(log n)**. Skip lists offer balanced-tree-like performance using only pointer operations (no rotations), are relatively simple to implement, and often work well in concurrent settings. The trade-off is a modest space overhead for the extra forward pointers.

Declaration of Authorship and Tools

I confirm that the report and all code presented were authored by me. I did not use generative AI to produce content or code for this submission. Tools used were limited to the Java JDK, an IDE (e.g., IntelliJ/VS Code), and basic spelling/grammar checking. All sources are cited in the References.

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press
2. Kleinberg, J., & Tardos, É. (2005). Algorithm Design. Pearson.
3. Sedgwick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
5. Pugh, W. (1990) ‘Skip lists: A probabilistic alternative to balanced trees’, *Communications of the ACM*, 33(6), pp. 668–676.
6. Pagh, R. and Rodler, F.F. (2004) ‘Cuckoo hashing’, *Journal of Algorithms*, 51(2), pp. 122–144.
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2022) *Introduction to Algorithms*. 4th edn. Cambridge, MA: MIT Press.
8. Goodrich, M.T., Tamassia, R. and Goldwasser, M.H. (2014) *Data Structures and Algorithms in Java*. 6th edn. Hoboken, NJ: Wiley.
9. Weiss, M.A. (2012) *Data Structures and Algorithm Analysis in Java*. 3rd edn. Boston, MA: Pearson.
10. Skiena, S.S. (2020) *The Algorithm Design Manual*. 3rd edn. Cham: Springer.
11. Oracle (2025) *Java SE 17 API Specification* Available at: docs.oracle.com (Accessed: 10 August 2025).
12. Oracle (2025) *Java SE 17 API Specification.* Available at: docs.oracle.com (Accessed: 10 August 2025).
13. Oracle (2025) *Java SE 17 API Specification* Available at: docs.oracle.com (Accessed: 10 August 2025).
14. Oracle (2025) *The Collections Framework — Overview*. Available at: docs.oracle.com (Accessed: 10 August 2025).