FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY

Department of Computer Science

Lab Manual

DATA STRUCTURE AND ALGORITHMS

Instructor's Name: _____

Student's Name: _____

Roll No.: _____ Batch: _____

Semester: _____ Year: _____

Department: _____

FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY

Department of Computer Science

Lab Manual

DATA STRUCTURE AND ALGORITHMS

Prepared By:

Reviewed / Approved By

_____

Table of Contents

#	Date	List of Experiment	Marks Obtained	Signature
1.	__ / __ / __	**Arrays**		
2.	__ / __ / __	**Sorting Algorithms**		

3.	__/__/__	**Linked Lists**		
4.	__/__/__	**Doubly Linked Lists**		
5.	__/__/__	**Stacks and Recursion**		
6.	__/__/__	**Queues**		
7.	__/__/__	**Priority Queues and Heaps**		
8.	__/__/__	**Open Ended Lab (OEL)**		
9.	__/__/__	**Hashing**		
10.	__/__/__	**Binary Search Trees (BST)**		
11.	__/__/__	**AVL Trees**		
12.	__/__/__	**B+ Trees**		
13.	__/__/__	**Graphs and Graph Traversal**		
14.	__/__/__	**Assessment of (OEL) and (CCA)**		

Psychomotor Rubrics Assessment Hardware Lab

Course Name (Course Code): _____

Semester: _____

Lab #	Score Allocation

	Experimental Setup Marks (3)	**Procedure Marks (2)**	**Experimental Results Marks (3)**	**Laboratory Manual Marks (2)**	**Total Marks (10)**
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
Total Mark		140		Total Obtained Marks	

Overall Score:_____out of 14 ***Examined by:_____***

(Obtained Score / 140) x 14 ***(Name and Signature of lab instructor)***

Psychomotor Rubrics for Software based Lab

Course Name (Course Code): _____

Semester: _____

Criteria	Exceeds Expectations (>=90%)	Meets Expectations (70%-89%)	Developing (50%-69%)	Unsatisfactory (<50%)
Software Skills	Ability to use software with its standard and advanced features without assistance	Ability to use software with its standard and advanced features with minimal assistance	Ability to use software with its standard features with assistance	Unable to use the software
Programming/ Simulation	Ability to program/ simulate the lab tasks with simplification	Ability to program/ simulate the lab tasks without errors	Ability to program/ simulate lab tasks with errors	Unable to program/simulate
Results	Ability to achieve all the desired results with alternate ways	Ability to achieve all the desired results	Ability to achieve most of the desired results with errors	Unable to achieve the desired results
Laboratory Manual	All sections of the report are very well written and technically accurate.	All sections of the report are technically accurate.	Few sections of the report contain technical errors.	All sections of the report contain multiple technical errors.

FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY

Psychomotor Rubrics Assessment Software based Lab

Course Name (Course Code): _____
Semester: _____

Lab #	Score Allocation				
	Software Skills Marks (3)	**Programming/ Simulation Marks (2)**	**Experimental Results Marks (3)**	**Laboratory Manual Marks (2)**	**Total Marks (3)**
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
Total Marks		140		Total Obtained marks	

Overall Score:_____out of 14 ***Examined by:_____***
(Obtained Score / 140) x 14 ***(Name and Signature of lab instructor)***

Affective Domain Rubrics Assessment

Course Name (Course Code): _____

Semester: _____

CATEGORY	Excellent (100% - 85%)	Good (84% - 75%)	Fair (74% - 60%)	Poor (Less than 60%)
Speaks Clearly	Speaks clearly and distinctly all the time, and confidently.	Speaks clearly and distinctly most of the time, but is confused for a brief period of time, however, recovers.	Speaks clearly and distinctly most of the time, but seems not confident about what has been delivered. Shows lack of confidence.	Often mumbles or cannot be understood and clearly lacks confidence in delivering the content
Points:				
Preparedness	Student is completely prepared and has obviously rehearsed.	Student seems pretty prepared but might have needed a couple more rehearsals.	The student is somewhat prepared, but it is clear that rehearsal was lacking.	Student does not seem at all prepared to present.
Points				
Answer back	Student calmly listens to the questions and responds to the question confidently and correctly	Student calmly listens to the questions, responds confidently but some of the responses are incorrect.	Student shows anxiety while listening to the questions, and gives some correct responses, but some of the responses are incorrect.	Student shows anxiety while listening to the questions, and most of the responses are incorrect.
Points:				
Posture, Eye Contact & Speaking Volume	Stands up straight, looks relaxed and confident. Establishes eye contact with everyone in the room during the presentation. Volume is loud enough to be heard by all members in the audience throughout the presentation.	Stands up straight and establishes eye contact with everyone in the room during the presentation. Volume is loud enough to be heard by the audience, but is sometimes not audible.	Sometimes stands up straight and establishes eye contact. Volume is loud enough to be heard by the audience, but many sentences spoken are not clear.	Lazy and informal posture. Does not look at people during the presentation. Volume is also too soft to be heard by the audience.
Points:				

Overall Score:_____out of 14 ***Examined by:_____***

(Name and Signature of lab instructor)

Open Ended Lab Assessment Rubrics

Course Name (Course Code): _____

Semester: _____

Criteria and Scales			
Excellent (10-8)	**Good (7-5)**	**Average (4-2)**	**Poor (1-0)**
Criterion 1: Problem Definition and Solution Design: How well the problem statement is understood by the student			
Understands the problem clearly and clearly identifies the underlying issues.	Adequately understands the problem and identifies the underlying issues.	Inadequately defines the problem and identifies the underlying issues.	Fails to define the problem adequately and does not identify the underlying issues.
Criterion 2: Implementation of Data Structures: Correct implementation of required data structures.			
Contains all the information needed for solving the problem	Good implementation, leading to a successful solution	Mediocre implementation may or may not lead to an adequate solution	No apparent implementation
Criterion 3: Algorithm Efficiency: Effective and efficient implementation of required algorithms.			
Effective and efficient implementation with complete notations	Effective and efficient implementation with incomplete notations	Effective and efficient implementation with improper naming convention and notations	Not Effective and efficient implementation
Criterion 4: Code Quality: How complete and accurate the code is along with the assumptions			
Complete Code according to the class diagram of the given case with clear assumptions	Incomplete Code according to the class diagram of the given case with clear assumptions	Incomplete Code according to the class diagram of the given case with unclear assumptions	Wrong code and naming conventions
Criterion 5: Report: How thorough and well organized is the solution			
All the necessary information clearly organized for easy use in solving the problem	Good information organized well that could lead to a good solution	Mediocre information which may or may not lead to a solution	No report provided

Open Ended Lab Assessment Rubrics

Course Name (Course Code): _____
Semester: _____

Criteria and Scales				
Excellent (10-8)	**Good (7-5)**	**Average (4-2)**	**Poor (1-0)**	**Total Marks 10**
Criterion 1: Understanding the Problem: How well the problem statement is understood by the student				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
Criterion 2: Implementation of Data Structures: Correct implementation of required data structures.				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
Criterion 3: Algorithm Efficiency: Effective and efficient implementation of required algorithms.				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
Criterion 4: Code Quality: How complete and accurate the code is along with the assumptions				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
Criterion 5: Report: How thorough and well organized is the solution				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
Total				**(____/5)**

Total marks obtained: _____

Name and Signature of lab instructor: _____

FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY

Rubrics for Lab Project / CCA

Course Name (Course Code): _____

Semester: _____

Criteria	Exceeds Expectations (>=90%)	Meets Expectations (70%-89%)	Developing (50%-69%)	Unsatisfactory (<50%)
Project Presentation + Project Demonstration	Ability to demonstrate the project with achievement of required objectives having clear understanding of project limitations and future enhancements. Hardware and/or Software modules are fully functional, if applicable.	Ability to demonstrate the project with achievement of required objectives but understanding of project limitations and future enhancements is insufficient. Hardware and/or Software modules are functional, if applicable.	Ability to demonstrate the project with achievement of a*t least 50% required objectives and insufficient understanding of project limitations and future enhancements. Hardware and/or Software modules are partially functional, if applicable.	Ability to demonstrate the project with achievement of less than 50% required objectives and lacks in understanding of project limitations and future enhancements. Hardware and/or Software modules are not functional, if applicable.
Project Report	All sections of the Project report are very well- written and technically accurate.	All sections of the Project report are technically accurate.	Few sections of the Project report contain technical errors.	Project report has several grammatical/ spelling errors and sentence construction is poor.
Viva	Able to answer the questions easily and correctly across the project.	Able to answer the questions related to the project	Able to answer the questions but with mistakes	Unable to answer the questions

Total marks: _____

Name and Signature of lab instructor: _____

FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY

Project / CCA Rubric based Assessment

Course Name (Course Code): _____

Semester: _____

Project #	Score Allocation			
	Project Presentation + Project Demonstration Marks (5)	**Project Report Marks (3)**	**Viva Marks (3)**	**Total Marks (10)**
1				
2				
Total Obtained Score				

Total marks obtained: _____

Name and Signature of lab instructor: _____

Mid Term Rubrics

Course Name (Course Code): _____

Semester: _____

Criteria	Exceeds Expectations (>=90%)	Meets Expectations (70%-89%)	Developing (50%-69%)	Unsatisfactory (<50%)
Performance	Able to present full knowledge of both problem and solution.	Able to present adequate knowledge of both problem and solution	Able to present sufficient knowledge of both problem and solution	No or very less knowledge of both problems and solution
Viva	Able to answer the questions easily and correctly	Able to answer the questions	Able to answer the questions but with mistakes	Unable to answer the questions

Mid Term Rubrics based Assessment

Course Name (Course Code): **_____**

Semester, Batch: **_____**

Score Allocation	
Performance	_____ /20
Viva	_____ /5
Total Obtained Score	**_____/ 25**

Examined by: **_____**

(Name and Signature of concerned lab instructor

Final Term Rubrics

Course Name (Course Code): _____
Semester: _____

Criteria	Exceeds Expectations (>=90%)	Meets Expectations (70%-89%)	Developing (50%-69%)	Unsatisfactory (<50%)
Performance	Able to present full knowledge of both problem and solution.	Able to present adequate knowledge of both problem and solution	Able to present sufficient knowledge of both problem and solution	No or very less knowledge of both problems and solution
Viva	Able to answer the questions easily and correctly	Able to answer the questions	Able to answer the questions but with mistakes	Unable to answer the questions

Final Term Rubrics based Assessment

Course Name (Course Code): _____
Semester, Batch: _____

Score Allocation	
Performance	_____ /45
Viva	_____ /5
Total Obtained Score	**_____/ 50**

Examined by: _____
(Name and Signature of concerned lab instructor)

Final Lab Assessment

Assessment Tool	CLO-1 (20)	CLO-2 (20)	CLO-3 (10)
Lab Manual			
Subject Project / Viva			
Lab Exam / Viva			
Score Obtained			
Total Score: _____ out of 50			

Examined by: _____

(Name and Signature of concerned lab instructor)

Lab Session 1: Exploring Arrays in Java

Objective:
This lab helps students understand arrays, a fundamental data structure in Java. Students will learn to create, manipulate, and perform basic operations on arrays.

Part 1: Creating and Initializing an Array

1. **Explanation:**
   - An array is a fixed-size data structure that holds elements of the same data type. It is indexed starting from 0.
2. **Steps:**
   - Define an integer array of size 5 and initialize it with values [5, 2, 9, 1, 6].
   - Print the elements of the array using a loop to demonstrate indexing.

```
1. int[] numbers = {5, 2, 9, 1, 6};
2. System.out.println("Array elements:");
3. for (int i = 0; i < numbers.length; i++) {
4.     System.out.println("Element at index " + i + ": " + numbers[i]);
5. }
6.
```

3. **Task:**
   - Explain the concept of array indexing and how each element in an array can be accessed using its index.

Part 2: Basic Array Operations

1. **Explanation:**
   - Arrays support several operations such as inserting, deleting, updating, and finding elements. Since arrays have a fixed size, we may need to create a new array to add or remove elements.
2. **Steps:**
   - **Update an Element:** Modify the third element (index 2) to 7 and print the updated array.

```
1. numbers[2] = 7;  // Update element at index 2
2. System.out.println("Updated array: " + Arrays.toString(numbers));
3.
```

   - **Search for an Element:** Write a method to search for a specific element in the array and return its index. If the element is not found, return -1.

```
1. public static int searchElement(int[] arr, int key) {
2.     for (int i = 0; i < arr.length; i++) {
3.         if (arr[i] == key) {
4.             return i;
5.         }
6.     }
7.     return -1;
8. }
9.
```

3. **Task:**
    - Search for an element (e.g., 7) and print its index or a "not found" message.

Part 3: Sorting the Array

1. **Explanation:**
    - Sorting is a common operation performed on arrays. It involves arranging elements in a specific order, such as ascending or descending.
2. **Steps:**
    - Use Arrays.sort to sort the array in ascending order and print the sorted array.

```
1. Arrays.sort(numbers);
2. System.out.println("Sorted array: " + Arrays.toString(numbers));
3.
```

3. **Task:**
    - Discuss the efficiency of the built-in sort function and its use in real-world applications.

Part 4: Practical Application - Finding the Maximum and Minimum Elements

1. **Explanation:**
    - Arrays are frequently used in real-life applications to store and analyze data. Here, students will find the maximum and minimum elements in the array.
2. **Steps:**
    - Write a method to find the maximum and minimum values in the array.

```
1. public static int findMax(int[] arr) {
2.     int max = arr[0];
3.     for (int i = 1; i < arr.length; i++) {
4.         if (arr[i] > max) {
5.             max = arr[i];
6.         }
7.     }
8.     return max;
9. }
10.
```

```
11. public static int findMin(int[] arr) {
12.     int min = arr[0];
13.     for (int i = 1; i < arr.length; i++) {
14.         if (arr[i] < min) {
15.             min = arr[i];
16.         }
17.     }
18.     return min;
19. }
20.
```

3. **Task:**
    - Call findMax and findMin methods on the array and print the results.

Part 5: Applying Array Operations in a Small Project

1. **Explanation:**
    - Small projects or challenges consolidate understanding of array manipulation. In this task, students will calculate the average grade after removing the lowest grade from a list of grades.
2. **Steps:**
    - Create an integer array of student grades: [78, 92, 88, 65, 70].
    - Sort the array, remove the lowest grade, calculate the sum of the remaining grades, and then calculate and print the average.

```
1. int[] grades = {78, 92, 88, 65, 70};
2. Arrays.sort(grades);
3. int sum = 0;
4. for (int i = 1; i < grades.length; i++) {  // Start from 1 to skip the lowest grade
5.     sum += grades[i];
6. }
7. double average = (double) sum / (grades.length - 1);
8. System.out.println("Average grade after removing the lowest: " + average);
9.
```

3. **Task:**
    - Explain how removing the lowest grade affects the average calculation.

Home Tasks: Further Exploration of Arrays

Home Task 1: Array Rotation

1. **Objective:** Practice manipulating elements in an array by rotating it.
2. **Task:** Write a method to rotate the elements of an array by k positions to the right. For example, rotating [1, 2, 3, 4, 5] by 2 positions should yield [4, 5, 1, 2, 3].
    - Use the modulo operator to handle rotations greater than the array size.
3. **Example Output:**
    - **Input:** [1, 2, 3, 4, 5], k = 2

- **Output:** [4, 5, 1, 2, 3]

Home Task 2: Removing Duplicates from a Sorted Array

1. **Objective:** Practice managing unique elements within an array.
2. **Task:** Write a method that takes a sorted array as input and removes duplicate values. Return a new array that contains only unique elements.
   - For example, the input [1, 2, 2, 3, 4, 4, 5] should result in [1, 2, 3, 4, 5].
3. **Example Output:**
   - **Input:** [1, 2, 2, 3, 4, 4, 5]
   - **Output:** [1, 2, 3, 4, 5]

Home Task 3: Finding Pairs with a Given Sum

1. **Objective:** Practice searching and indexing within an array.
2. **Task:** Write a method that finds all pairs of numbers in an array that add up to a specified target sum. Return a list of pairs that meet the condition.
   - For example, in the array [1, 2, 3, 4, 5] with a target sum of 6, the pairs would be (1, 5) and (2, 4).
3. **Example Output:**
   - **Input:** Array [1, 2, 3, 4, 5], Target 6
   - **Output:** (1, 5), (2, 4)

Submission Guidelines:

- Each task should be submitted as a separate Java file with comments explaining each part of the code.
- Attach screenshots of the output for each task.

Grading Rubric:

- **Lab Task:** Completion and understanding of basic array operations (50 points).
- **Home Task 1 (Array Rotation):** Correctly rotating the array by a specified number of positions (20 points).
- **Home Task 2 (Removing Duplicates):** Accurate removal of duplicates and returning a unique array (15 points).
- **Home Task 3 (Finding Pairs with Sum):** Correct identification of pairs that add up to the target sum (15 points).

This lab and these home tasks provide a comprehensive introduction to arrays, allowing students to develop a foundational understanding of array manipulation and its practical applications.

Lab Session 2: Understanding Sorting Algorithms

Objective:

This lab aims to help students understand and implement the Bubble Sort, Selection Sort, and Insertion Sort algorithms in Java. By the end of the lab, students will understand how these sorting algorithms work and when each is useful.

Part 1: Bubble Sort

1. **Explanation:**
   - Bubble Sort is a simple comparison-based algorithm where each pair of adjacent elements is compared, and they are swapped if they are in the wrong order. This process repeats until the array is sorted.
2. **Steps:**
   - Initialize an array of integers [64, 25, 12, 22, 11].
   - Implement the Bubble Sort algorithm and explain each part of the code.

```
1. public static void bubbleSort(int[] arr) {
2.     int n = arr.length;
3.     for (int i = 0; i < n - 1; i++) {
4.         for (int j = 0; j < n - i - 1; j++) {
5.             if (arr[j] > arr[j + 1]) {
6.                 // Swap arr[j] and arr[j+1]
7.                 int temp = arr[j];
8.                 arr[j] = arr[j + 1];
9.                 arr[j + 1] = temp;
10.            }
11.        }
12.    }
13. }
14.
```

3. **Output:**
   - Print the array after each pass of the outer loop to show the sorting process.
4. **Task:**
   - Explain why the algorithm has a time complexity of $O(n^2)$ and when it might be useful (e.g., for small or nearly sorted arrays).

Part 2: Selection Sort

1. **Explanation:**
   - Selection Sort repeatedly finds the minimum element (for ascending order) from the unsorted portion and places it at the beginning.
2. **Steps:**

- Use the same array [64, 25, 12, 22, 11].
- Implement the Selection Sort algorithm, explaining each part of the code.

```
1. public static void selectionSort(int[] arr) {
2.     int n = arr.length;
3.     for (int i = 0; i < n - 1; i++) {
4.         int minIndex = i;
5.         for (int j = i + 1; j < n; j++) {
6.             if (arr[j] < arr[minIndex]) {
7.                 minIndex = j;
8.             }
9.         }
10.         // Swap the found minimum element with the first element
11.         int temp = arr[minIndex];
12.         arr[minIndex] = arr[i];
13.         arr[i] = temp;
14.     }
15. }
16.
```

3. **Output:**
   - Print the array after each pass of the outer loop to show the sorting process.
4. **Task:**
   - Explain why Selection Sort is also $O(n^2)$ and how it differs from Bubble Sort (fewer swaps but still slow for large datasets).

Part 3: Insertion Sort

1. **Explanation:**
   - Insertion Sort builds the final sorted array one item at a time by repeatedly picking the next item and inserting it into its correct position within the already sorted portion of the array.
2. **Steps:**
   - Use the array [64, 25, 12, 22, 11].
   - Implement the Insertion Sort algorithm, explaining each part of the code.

```
1. public static void insertionSort(int[] arr) {
2.     int n = arr.length;
3.     for (int i = 1; i < n; i++) {
4.         int key = arr[i];
5.         int j = i - 1;
6.         while (j >= 0 && arr[j] > key) {
7.             arr[j + 1] = arr[j];
8.             j = j - 1;
9.         }
10.         arr[j + 1] = key;
11.     }
12. }
13.
```

3. **Output:**
   - Print the array after each insertion step.

4. **Task:**
    - Explain why Insertion Sort has a time complexity of $O(n^2)$ but can be efficient for small or nearly sorted arrays.

Final Task: Comparing Sorting Algorithms

Objective:

- Compare the efficiency of Bubble Sort, Selection Sort, and Insertion Sort.

Steps:

- Create a method to generate an array of random integers.
- Use arrays of different sizes (e.g., 10, 100, 1000 elements) and measure the time taken by each sorting algorithm.
- Print out the time taken by each algorithm for each array size.

Home Tasks: Implementing and Exploring Other Sorting Algorithms

Home Task 1: Implement Merge Sort

1. **Objective:** Practice implementing a more efficient sorting algorithm.
2. **Task:** Write a Java program to implement Merge Sort on an integer array.
    - Explain how Merge Sort divides the array into halves, recursively sorts each half, and then merges the sorted halves.
    - Analyze the time complexity ($O(n \log n)$) and explain why it performs better on larger arrays compared to $O(n^2)$ algorithms.

Home Task 2: Implement Quick Sort

1. **Objective:** Implement a popular divide-and-conquer sorting algorithm.
2. **Task:** Write a Java program to implement Quick Sort on an integer array.
    - Use the last element as the pivot and partition the array into elements less than the pivot and elements greater than the pivot.
    - Recursively apply the same process to each partition.
    - Analyze the average and worst-case time complexity ($O(n \log n)$ average, $O(n^2)$ worst) and discuss cases where Quick Sort performs well.

Home Task 3: Implement Heap Sort

1. **Objective:** Explore a sorting algorithm based on the binary heap data structure.
2. **Task:** Write a Java program to implement Heap Sort on an integer array.
    - Explain how Heap Sort builds a max-heap and repeatedly extracts the maximum element, placing it at the end of the array.
    - Discuss the time complexity (O(n log n)) and memory efficiency of Heap Sort (in-place sorting).

Submission Guidelines:

- Each sorting algorithm should be implemented in a separate Java file.
- Include comments explaining each part of the code.
- Attach screenshots of the output for sample arrays.

Grading Rubric:

- **Lab Tasks:** Completion and understanding of Bubble, Selection, and Insertion Sort (50 points).
- **Home Task 1 (Merge Sort):** Correctness and explanation of Merge Sort (20 points).
- **Home Task 2 (Quick Sort):** Correctness and explanation of Quick Sort (15 points).
- **Home Task 3 (Heap Sort):** Correctness and explanation of Heap Sort (15 points).

These tasks provide a comprehensive overview of sorting algorithms, starting with basic sorts and progressing to more efficient techniques.

Lab Session 3: Understanding Singly Linked Lists

Objective:
This lab task introduces students to the concept of singly linked lists, a fundamental data structure. Students will learn to create a linked list, add elements, delete elements, and traverse the list. By the end of the lab, students should understand how linked lists work and how they differ from arrays.

Part 1: Creating a Node

1. **Explanation:**
   - A singly linked list consists of nodes where each node contains data and a reference (or pointer) to the next node in the sequence.
2. **Steps:**
   - Start by creating a Node class that will represent each node in the linked list.

```
1. class Node {
2.     int data;
3.     Node next;
4.
5.     // Constructor
6.     public Node(int data) {
7.         this.data = data;
8.         this.next = null;
9.     }
10. }
11.
```

3. **Task:**
   - Explain the structure of the Node class, focusing on the data field and the next pointer.

Part 2: Creating a Linked List Class

1. **Explanation:**
   - Create a LinkedList class to represent the linked list, which will manage the nodes. This class will include methods to add, remove, and display nodes in the list.
2. **Steps:**
   - Define a LinkedList class with a head pointer, which points to the first node in the list.

```
1. class LinkedList {
2.     Node head;  // Head of the list
3.
4.     // Constructor
5.     public LinkedList() {
6.         head = null;
7.     }
8. }
9.
```

3. **Task:**
   - Explain the purpose of the head pointer, which keeps track of the first node in the linked list.

Part 3: Inserting Nodes

1. **Explanation:**
   - Write a method to add nodes to the linked list. We'll start with adding nodes to the end of the list.
2. **Steps:**
   - Implement the add method in the LinkedList class.

```
1. public void add(int data) {
2.     Node newNode = new Node(data);
3.     if (head == null) {
4.         head = newNode;
5.     } else {
6.         Node current = head;
7.         while (current.next != null) {
8.             current = current.next;
9.         }
10.        current.next = newNode;
11.    }
12. }
13.
```

3. **Task:**
   - Add elements [10, 20, 30, 40] to the linked list and print the list after each insertion to observe the process.

Part 4: Displaying the List

1. **Explanation:**
   - Implement a method to display all elements in the list by traversing from the head to the end.
2. **Steps:**
   - Add a display method to print all the elements in the linked list.

```
1. public void display() {
2.     Node current = head;
3.     while (current != null) {
4.         System.out.print(current.data + " -> ");
5.         current = current.next;
6.     }
7.     System.out.println("null");
8. }
9.
```

3. **Task:**
   - After adding elements [10, 20, 30, 40], use the display method to show the structure of the list.

Part 5: Deleting a Node

1. **Explanation:**
   - Write a method to delete a node with a specified value. This involves locating the node and adjusting the pointers to exclude it from the list.
2. **Steps:**
   - Implement the delete method in the LinkedList class.

```
1. public void delete(int key) {
2.     Node current = head, prev = null;
3.     if (current != null && current.data == key) {
4.         head = current.next;  // Remove the head node
5.         return;
6.     }
7.     while (current != null && current.data != key) {
8.         prev = current;
9.         current = current.next;
10.    }
11.    if (current == null) return;  // Key not found
12.    prev.next = current.next;     // Remove node
13. }
14.
```

3. **Task:**
   - Delete a specific element (e.g., 20) from the list and display the updated list.

Part 6: Searching for an Element

1. **Explanation:**
   - Write a method to search for an element by value in the linked list.
2. **Steps:**
   - Implement a search method that returns whether a specified element exists in the list.

```
1. public boolean search(int key) {
2.     Node current = head;
3.     while (current != null) {
4.         if (current.data == key) return true;
5.         current = current.next;
6.     }
7.     return false;
8. }
9.
```

3. **Task:**
   - Search for an element (e.g., 30) in the list and display whether it was found.

Home Tasks: Expanding Linked List Operations

Home Task 1: Reverse the Linked List

1. **Objective:** Practice manipulating node pointers by reversing the list.
2. **Task:** Write a method in the LinkedList class that reverses the linked list.
    - Traverse the list, reverse the direction of each pointer, and update the head to the last node.
3. **Example Output:**
    - **Input List:** 10 -> 20 -> 30 -> 40 -> null
    - **Reversed List:** 40 -> 30 -> 20 -> 10 -> null

Home Task 2: Find the Middle Element of the Linked List

1. **Objective:** Practice list traversal and indexing to find the middle element.
2. **Task:** Write a method in the LinkedList class to find the middle element of the linked list.
    - Use a two-pointer technique: one pointer moves one step at a time, while the other moves two steps.
    - When the faster pointer reaches the end, the slower pointer will be at the middle.
3. **Example Output:**
    - **Input List:** 10 -> 20 -> 30 -> 40 -> 50 -> null
    - **Middle Element:** 30

Home Task 3: Detect and Remove Cycles in the Linked List

1. **Objective:** Understand how to detect cycles in a linked list, an important problem in linked list manipulation.
2. **Task:** Write a method in the LinkedList class to detect a cycle in the list using Floyd's Cycle-Finding Algorithm (fast and slow pointer method). If a cycle is detected, remove it.
    - **Cycle Detection:** If the fast and slow pointers meet, there is a cycle.
    - **Cycle Removal:** Find the node where the cycle begins and set its next pointer to null.
3. **Example Output:**
    - **Input List with Cycle:** 10 -> 20 -> 30 -> 40 -> 20 (cycle)
    - **Output after Cycle Removal:** 10 -> 20 -> 30 -> 40 -> null

---

Submission Guidelines:

- Submit each task as a separate Java file with comments explaining each part.
- Include screenshots of the output for each task.

Grading Rubric:

- **Lab Task:** Proper implementation of add, delete, display, and search methods (50 points).
- **Home Task 1 (Reverse the Linked List):** Correct implementation and understanding of pointer manipulation (20 points).
- **Home Task 2 (Find the Middle Element):** Correctness and efficiency in finding the middle element (15 points).
- **Home Task 3 (Detect and Remove Cycle):** Successful cycle detection and removal using pointers (15 points).

This lab and set of home tasks will provide a solid understanding of singly linked lists and prepare students for more advanced linked list operations.

Lab Session 4: Understanding Doubly Linked Lists

Objective:
This lab introduces students to doubly linked lists, a data structure in which each node contains pointers to both the next and previous nodes. Students will learn to create a doubly linked list, add nodes at different positions, delete nodes, and traverse the list in both directions.

Part 1: Creating a Node for the Doubly Linked List

1. **Explanation:**
   - In a doubly linked list, each node has three components: data, a pointer to the next node, and a pointer to the previous node.
2. **Steps:**
   - Start by creating a Node class that represents each node in the doubly linked list.

```
1. class Node {
2.     int data;
3.     Node next;
4.     Node prev;
5.
6.     // Constructor
7.     public Node(int data) {
8.         this.data = data;
9.         this.next = null;
10.         this.prev = null;
11.     }
12. }
13.
```

3. **Task:**
   - Explain the Node class structure, highlighting the purpose of the next and prev pointers.

Part 2: Creating a Doubly Linked List Class

1. **Explanation:**
   - Create a DoublyLinkedList class to manage the nodes. This class will contain methods to add, delete, and display nodes.
2. **Steps:**
   - Define the DoublyLinkedList class with a head pointer to keep track of the first node.

```
1. class DoublyLinkedList {
2.     Node head;
3.
4.     // Constructor
5.     public DoublyLinkedList() {
6.         head = null;
7.     }
8. }
9.
```

3. **Task:**

- Explain the purpose of the head pointer, which points to the start of the doubly linked list.

Part 3: Inserting Nodes

1. **Explanation:**
   - Write a method to add nodes at the beginning and at the end of the doubly linked list.
2. **Steps:**
   - Implement the addAtBeginning and addAtEnd methods in the DoublyLinkedList class.

```
1. // Add a node at the beginning
2. public void addAtBeginning(int data) {
3.     Node newNode = new Node(data);
4.     if (head != null) {
5.         newNode.next = head;
6.         head.prev = newNode;
7.     }
8.     head = newNode;
9. }
10.
11. // Add a node at the end
12. public void addAtEnd(int data) {
13.     Node newNode = new Node(data);
14.     if (head == null) {
15.         head = newNode;
16.         return;
17.     }
18.     Node current = head;
19.     while (current.next != null) {
20.         current = current.next;
21.     }
22.     current.next = newNode;
23.     newNode.prev = current;
24. }
25.
```

3. **Task:**
   - Add elements [10, 20, 30, 40] to the beginning and [50, 60] to the end of the list, then display the list after each addition.

Part 4: Displaying the List in Both Directions

1. **Explanation:**
   - Implement a method to display the elements of the doubly linked list from head to tail (forward traversal) and from tail to head (backward traversal).
2. **Steps:**
   - Add displayForward and displayBackward methods in the DoublyLinkedList class.

```
1. // Display list forward
2. public void displayForward() {
```

```
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " -> ");
        current = current.next;
    }
    System.out.println("null");
}

// Display list backward
public void displayBackward() {
    Node current = head;
    if (current == null) return;  // Empty list
    while (current.next != null) {  // Move to end of list
        current = current.next;
    }
    while (current != null) {  // Display in reverse
        System.out.print(current.data + " -> ");
        current = current.prev;
    }
    System.out.println("null");
}

```

3. **Task:**
    - After adding the nodes as above, display the list both forward and backward.

Part 5: Deleting a Node

1. **Explanation:**
    - Write a method to delete a node by value. This requires updating pointers for both the next and previous nodes.
2. **Steps:**
    - Implement a deleteNode method that finds and deletes a node with the specified value.

```
public void deleteNode(int key) {
    Node current = head;

    // Find the node to delete
    while (current != null && current.data != key) {
        current = current.next;
    }

    if (current == null) return;  // Key not found

    if (current.prev != null) {
        current.prev.next = current.next;
    } else {
        head = current.next;  // Deleting the head node
    }

    if (current.next != null) {
        current.next.prev = current.prev;
    }
}

```

3. **Task:**

- Delete a specific node (e.g., 30) and display the list forward and backward to show the changes.

Home Tasks: Expanding on Doubly Linked List Operations

Home Task 1: Inserting at a Specific Position

1. **Objective:** Practice inserting nodes at a specific position in the list.
2. **Task:** Write a method in the DoublyLinkedList class to insert a node at a given position (0-based index).
   - If the position is 0, insert at the beginning. Otherwise, traverse the list to insert the node at the specified index.
3. **Example Output:**
   - **Input List:** 10 -> 20 -> 40 -> null
   - **Insert 30 at position 2:** 10 -> 20 -> 30 -> 40 -> null

---

Home Task 2: Finding and Counting Elements

1. **Objective:** Develop skills to find nodes and count occurrences.
2. **Task:** Write two methods:
   - A find method that searches for a value and returns its position or -1 if not found.
   - A countOccurrences method that counts how many times a specified value appears in the list.
3. **Example Output:**
   - **Input List:** 10 -> 20 -> 30 -> 20 -> 40 -> null
   - **Find Position of 30:** Position 2
   - **Count Occurrences of 20:** 2

Home Task 3: Sorting the Doubly Linked List

1. **Objective:** Practice sorting algorithms on a doubly linked list.
2. **Task:** Write a method to sort the doubly linked list in ascending order.
   - Implement a simple sorting algorithm like bubble sort or selection sort that can handle node pointers.
   - Sort the list by rearranging the next and prev pointers instead of swapping data.
3. **Example Output:**
   - **Input List:** 40 -> 20 -> 30 -> 10 -> null
   - **Sorted List:** 10 -> 20 -> 30 -> 40 -> null

Submission Guidelines:

- Submit each task as a separate Java file with comments explaining each part of the code.
- Include screenshots of the output for each task to demonstrate functionality.

Grading Rubric:

- **Lab Task:** Successful implementation of adding, deleting, and displaying nodes (50 points).
- **Home Task 1 (Insert at Specific Position):** Proper insertion at the correct position (20 points).
- **Home Task 2 (Find and Count):** Accurate implementation of find and count functionalities (15 points).
- **Home Task 3 (Sort Doubly Linked List):** Correct sorting of nodes and pointer adjustment (15 points).

This lab and these home tasks will help students understand doubly linked list structures, pointer manipulation, and data organization within linked lists.

Lab Session 5: Understanding Stacks and Recursion

Objective:
This lab introduces students to the stack data structure and recursion, two fundamental concepts in data structures and algorithms. Students will learn to implement a stack using arrays, perform basic stack operations, and apply recursion to solve common problems.

Part 1: Implementing a Stack using Arrays

1. **Explanation:**
   - A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, where elements are added and removed from the top.
   - Basic stack operations include push, pop, peek, and isEmpty.
2. **Steps:**
   - Create a Stack class with an array and necessary variables to keep track of the stack's top and maximum capacity.

```
1. class Stack {
2.     private int maxSize;
3.     private int[] stackArray;
4.     private int top;
5.
6.     // Constructor to initialize stack
7.     public Stack(int size) {
8.         maxSize = size;
9.         stackArray = new int[maxSize];
10.        top = -1;  // Stack is initially empty
11.    }
12. }
13.
```

3. **Task:**
   - Explain the basic structure of the Stack class and the importance of the top variable in tracking the last inserted element.

Part 2: Implementing Stack Operations

1. **Explanation:**
   - Implement essential operations for the stack: push (to add an element), pop (to remove the top element), peek (to view the top element), and isEmpty (to check if the stack is empty).
2. **Steps:**
   - Implement each method in the Stack class.

```
1. // Push operation
2. public void push(int value) {
3.     if (top == maxSize - 1) {
4.         System.out.println("Stack overflow. Cannot push " + value);
```

```
        return;
    }
    stackArray[++top] = value;
}

// Pop operation
public int pop() {
    if (top == -1) {
        System.out.println("Stack underflow. Stack is empty.");
        return -1;
    }
    return stackArray[top--];
}

// Peek operation
public int peek() {
    if (top == -1) {
        System.out.println("Stack is empty.");
        return -1;
    }
    return stackArray[top];
}

// isEmpty operation
public boolean isEmpty() {
    return top == -1;
}

```

3. **Task:**
    - Test each operation by creating a Stack object, pushing and popping elements, and using peek to check the top element.

Part 3: Application of Stack - Reversing a String

1. **Explanation:**
    - One common application of a stack is reversing a string. By pushing each character onto the stack and then popping them, we reverse the order of the characters.
2. **Steps:**
    - Write a method that takes a string, pushes each character onto a stack, and then pops each character to form the reversed string.

```
public static String reverseString(String str) {
    Stack stack = new Stack(str.length());
    for (int i = 0; i < str.length(); i++) {
        stack.push(str.charAt(i));
    }
    StringBuilder reversed = new StringBuilder();
    while (!stack.isEmpty()) {
        reversed.append((char) stack.pop());
    }
    return reversed.toString();
}

```

3. **Task:**
    - Call reverseString("Hello") and verify that the output is "olleH".

Part 4: Introduction to Recursion

1. **Explanation:**
   - Recursion is a technique where a method calls itself to solve a smaller instance of the same problem. It often involves a base case to prevent infinite recursion.
2. **Steps:**
   - Explain the structure of a recursive method and the concept of base cases.
3. **Task:**
   - Write a simple recursive method to calculate the factorial of a number n, with the base case as n == 0 (since 0! = 1).

```
1. public static int factorial(int n) {
2.     if (n == 0) {
3.         return 1;
4.     }
5.     return n * factorial(n - 1);
6. }
7.
```

4. **Example Output:**
   - **Input:** factorial(5)
   - **Output:** 120

Part 5: Application of Recursion - Fibonacci Sequence

1. **Explanation:**
   - The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, starting from 0 and 1. Recursive calls can be used to generate Fibonacci numbers.
2. **Steps:**
   - Implement a recursive method to calculate the nth Fibonacci number. The base cases are fibonacci(0) = 0 and fibonacci(1) = 1.

```
1. public static int fibonacci(int n) {
2.     if (n == 0) return 0;
3.     if (n == 1) return 1;
4.     return fibonacci(n - 1) + fibonacci(n - 2);
5. }
6.
```

3. **Example Output:**
   - **Input:** fibonacci(5)
   - **Output:** 5

Home Tasks: Further Applications of Stacks and Recursion

Home Task 1: Balancing Parentheses Using Stack

1. **Objective:** Use a stack to verify if parentheses in an expression are balanced.
2. **Task:** Write a Java program that checks if a string containing parentheses, braces, and brackets is balanced. For example, "{[()]}" is balanced, while "{[(])}" is not.
    - Use a stack to push opening brackets and pop when encountering closing brackets.
3. **Example Output:**
    - **Input:** "{[()]}" → Balanced
    - **Input:** "{[(])}" → Not Balanced

Home Task 2: Recursive Solution for Power Calculation

1. **Objective:** Practice recursion by implementing a function to calculate powers.
2. **Task:** Write a recursive method power(base, exponent) that calculates base^exponent.
    - The base case is when exponent == 0 (since any number to the power of 0 is 1).
    - If exponent is negative, calculate the reciprocal (e.g., power(2, -3) = 1 / power(2, 3)).
3. **Example Output:**
    - **Input:** power(2, 3) → 8
    - **Input:** power(5, -2) → 0.04

Home Task 3: Implementing Tower of Hanoi with Recursion

1. **Objective:** Practice recursion by solving the Tower of Hanoi problem.
2. **Task:** Write a recursive solution for the Tower of Hanoi, where the objective is to move n disks from one peg to another with the help of an auxiliary peg.
    - Print each move in the format: "Move disk X from A to B".
3. **Example Output:**
    - **Input:** towerOfHanoi(3, 'A', 'C', 'B')
    - **Output:**

Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

Submission Guidelines:

- Each task should be submitted as a separate Java file.
- Include comments explaining each part of the code.
- Attach screenshots of the output for each task.

Grading Rubric:

- **Lab Task:** Correct implementation of stack operations and recursion methods (50 points).
- **Home Task 1 (Balancing Parentheses):** Accurate use of stack for validation (20 points).
- **Home Task 2 (Power Calculation):** Correct recursive implementation for positive and negative exponents (15 points).
- **Home Task 3 (Tower of Hanoi):** Correct solution with printed moves (15 points).

This lab and set of home tasks provide a comprehensive introduction to stacks and recursion, covering basic concepts and practical applications.

Lab Session 6: Understanding Queues

Objective:
This lab introduces students to the queue data structure, a fundamental concept in data structures and algorithms. Students will learn to implement a queue using arrays, perform basic queue operations, and explore different types of queues.

Part 1: Implementing a Simple Queue Using Arrays

1. **Explanation:**
   - A queue is a linear data structure that follows the First In, First Out (FIFO) principle, where elements are added at the back (enqueue) and removed from the front (dequeue).
   - Common queue operations include enqueue, dequeue, peek, and isEmpty.
2. **Steps:**
   - Create a Queue class using an array, with variables to keep track of the front, rear, and the maximum size of the queue.

```
1. class Queue {
2.     private int maxSize;
3.     private int[] queueArray;
4.     private int front;
5.     private int rear;
6.     private int currentSize;
7.
8.     // Constructor to initialize queue
9.     public Queue(int size) {
10.        maxSize = size;
11.        queueArray = new int[maxSize];
12.        front = 0;
13.        rear = -1;
14.        currentSize = 0;
15.    }
16. }
17.
```

3. **Task:**
   - Explain the Queue class structure, highlighting the purpose of front, rear, and currentSize.

Part 2: Implementing Queue Operations

1. **Explanation:**
   - Implement essential operations for the queue: enqueue (to add an element at the rear), dequeue (to remove the front element), peek (to view the front element), and isEmpty (to check if the queue is empty).
2. **Steps:**
   - Implement each method in the Queue class.

```
1. // Enqueue operation
2. public void enqueue(int value) {
3.     if (currentSize == maxSize) {
4.         System.out.println("Queue is full. Cannot enqueue " + value);
5.         return;
6.     }
7.     rear = (rear + 1) % maxSize;  // Circular increment
8.     queueArray[rear] = value;
9.     currentSize++;
10. }
11.
12. // Dequeue operation
13. public int dequeue() {
14.     if (currentSize == 0) {
15.         System.out.println("Queue is empty. Cannot dequeue.");
16.         return -1;
17.     }
18.     int temp = queueArray[front];
19.     front = (front + 1) % maxSize;  // Circular increment
20.     currentSize--;
21.     return temp;
22. }
23.
24. // Peek operation
25. public int peek() {
26.     if (currentSize == 0) {
27.         System.out.println("Queue is empty.");
28.         return -1;
29.     }
30.     return queueArray[front];
31. }
32.
33. // isEmpty operation
34. public boolean isEmpty() {
35.     return currentSize == 0;
36. }
37.
```

3. **Task:**
    - Test each operation by creating a Queue object, enqueuing and dequeuing elements, and using peek to check the front element.

Part 3: Implementing a Circular Queue

1. **Explanation:**
    - Circular Queues allow the rear pointer to wrap around to the beginning of the array when it reaches the end, making efficient use of space.
    - Explain how the use of modulo operator (index + 1) % maxSize enables this circular behavior.
2. **Steps:**
    - Modify the Queue class created above so that the enqueue and dequeue operations use circular logic for front and rear pointers.

3. **Task:**
    - Test the circular queue by enqueuing elements beyond the array's length and observing how the circular behavior maintains queue functionality.

Part 4: Application of Queue - Simulating a Printer Queue

1. **Explanation:**
    - Queues are often used in scheduling tasks, such as managing a printer queue where print jobs are processed in the order they are received.
2. **Steps:**
    - Write a method to simulate a printer queue. Enqueue print jobs (represented by job IDs), and dequeue them as they are "processed."

```
1. public void simulatePrinterQueue(Queue printerQueue) {
2.     // Enqueue jobs
3.     printerQueue.enqueue(101);
4.     printerQueue.enqueue(102);
5.     printerQueue.enqueue(103);
6.
7.     // Process jobs
8.     while (!printerQueue.isEmpty()) {
9.         System.out.println("Processing job ID: " + printerQueue.dequeue());
10.    }
11. }
12.
```

3. **Task:**
    - Call the simulatePrinterQueue method and verify that print jobs are processed in the order they were added to the queue.

Home Tasks: Advanced Queue Implementations

Home Task 1: Implementing a Queue Using Linked List

1. **Objective:** Implement a queue using a linked list instead of an array.
2. **Task:** Create a LinkedQueue class that uses a linked list to manage queue elements. Implement enqueue, dequeue, peek, and isEmpty methods.
    - Use a Node class to represent each element, with pointers for front and rear.
3. **Example Output:**
    - **Input Operations:** enqueue(10), enqueue(20), dequeue(), peek()
    - **Output:** Front element after operations: 20

Home Task 2: Implement a Priority Queue

1. **Objective:** Implement a basic priority queue using an array.

2. **Task:** Write a PriorityQueue class where each element has a priority, and elements are dequeued in order of highest priority.
    - Elements with higher priority values are dequeued before those with lower values.
3. **Example Output:**
    - **Input Operations:** enqueue(30, priority 2), enqueue(40, priority 1), enqueue(50, priority 3)
    - **Output on Dequeue:** 50, 30, 40 (based on priority order)

Home Task 3: Implement Recursive Queue Reversal

1. **Objective:** Practice recursion with a queue by reversing its order.
2. **Task:** Write a method to reverse a queue using recursion. Use only the enqueue, dequeue, and recursive function calls to reverse the queue's order.
    - The base case is when the queue is empty.
3. **Example Output:**
    - **Input Queue:** 10 -> 20 -> 30 -> null
    - **Reversed Queue:** 30 -> 20 -> 10 -> null

Submission Guidelines:

- Each task should be submitted as a separate Java file with comments explaining each part.
- Include screenshots of the output for each task to demonstrate functionality.

Grading Rubric:

- **Lab Task:** Correct implementation of queue operations and understanding of circular queue (50 points).
- **Home Task 1 (Linked List Queue):** Proper use of linked list structure in queue (20 points).
- **Home Task 2 (Priority Queue):** Correct handling of priorities in queue operations (15 points).
- **Home Task 3 (Recursive Queue Reversal):** Correct reversal of queue using recursion (15 points).

This lab and set of home tasks will provide students with a thorough understanding of queues, both conceptually and practically, while exploring variations like circular, linked, and priority queues.

Lab Session 7: Understanding P - Queues, Heaps

Objective:
This lab introduces students to priority queues and heaps, two essential data structures used to manage elements based on priority. Students will learn how to implement a priority queue using a binary heap and perform fundamental heap operations.

Part 1: Introduction to Heaps and Priority Queues

1. **Explanation:**
   - A **heap** is a binary tree-based data structure with a specific order property: in a **max heap**, each parent node is greater than or equal to its children, while in a **min heap**, each parent is smaller than or equal to its children.
   - A **priority queue** is a queue where each element has a priority, and elements are dequeued in priority order (typically highest to lowest or vice versa).
   - Heaps are commonly used to implement priority queues as they allow efficient insertion and deletion.

Part 2: Implementing a Max Heap

1. **Explanation:**
   - A binary heap can be represented using an array, where the parent of an element at index i is at index (i-1)/2, the left child is at 2*i + 1, and the right child is at 2*i + 2.
2. **Steps:**
   - Create a MaxHeap class with an array to store elements, and methods for basic heap operations: insert, remove, and heapify.

```
1. class MaxHeap {
2.     private int[] heapArray;
3.     private int maxSize;
4.     private int size;
5.
6.     // Constructor to initialize the max heap
7.     public MaxHeap(int maxSize) {
8.         this.maxSize = maxSize;
9.         this.size = 0;
10.         heapArray = new int[maxSize];
11.     }
12.
13.     // Get the parent index
14.     private int parent(int i) { return (i - 1) / 2; }
15.     private int leftChild(int i) { return 2 * i + 1; }
16.     private int rightChild(int i) { return 2 * i + 2; }
17. }
18.
```

3. **Task:**

- Explain the structure of the MaxHeap class, especially the use of array indices for managing parent-child relationships.

Part 3: Inserting into the Max Heap

1. **Explanation:**
   - To maintain the heap property after inserting an element, we need to **heapify up**: place the new element at the end of the heap and repeatedly swap it with its parent if it's greater, until the max-heap property is restored.
2. **Steps:**
   - Implement the insert method with heapify-up logic.

```
1. public void insert(int value) {
2.     if (size == maxSize) {
3.         System.out.println("Heap is full. Cannot insert " + value);
4.         return;
5.     }
6.     heapArray[size] = value;
7.     int current = size;
8.     size++;
9.
10.     // Heapify up
11.     while (current > 0 && heapArray[current] > heapArray[parent(current)]) {
12.         swap(current, parent(current));
13.         current = parent(current);
14.     }
15. }
16.
17. private void swap(int i, int j) {
18.     int temp = heapArray[i];
19.     heapArray[i] = heapArray[j];
20.     heapArray[j] = temp;
21. }
22.
```

3. **Task:**
   - Insert values [30, 20, 15, 5, 10, 25, 40] and print the heap array after each insertion to observe the heap property.

Part 4: Removing the Maximum Element

1. **Explanation:**
   - In a max heap, the maximum element is at the root. Removing it involves replacing the root with the last element, reducing the heap size, and **heapifying down** to maintain the max-heap property.
2. **Steps:**
   - Implement the removeMax method with heapify-down logic.

```
1. public int removeMax() {
2.     if (size == 0) {
```

```
3.        System.out.println("Heap is empty. Cannot remove.");
4.        return -1;
5.    }
6.    int root = heapArray[0];
7.    heapArray[0] = heapArray[size - 1];
8.    size--;
9.
10.   // Heapify down
11.   heapifyDown(0);
12.
13.   return root;
14. }
15.
16. private void heapifyDown(int i) {
17.    int largest = i;
18.    int left = leftChild(i);
19.    int right = rightChild(i);
20.
21.    if (left < size && heapArray[left] > heapArray[largest]) {
22.        largest = left;
23.    }
24.    if (right < size && heapArray[right] > heapArray[largest]) {
25.        largest = right;
26.    }
27.    if (largest != i) {
28.        swap(i, largest);
29.        heapifyDown(largest);
30.    }
31. }
32.
```

3. **Task:**
    - Call removeMax and print the heap array after each removal to observe the changes and verify that the max-heap property is preserved.

Part 5: Implementing a Priority Queue with Max Heap

1. **Explanation:**
    - Now that we have a max heap, we can use it to implement a priority queue where elements with the highest priority (or value) are dequeued first.
2. **Steps:**
    - Implement enqueue and dequeue in a PriorityQueue class using the MaxHeap methods.

```
1. class PriorityQueue {
2.    private MaxHeap maxHeap;
3.
4.    public PriorityQueue(int maxSize) {
5.        maxHeap = new MaxHeap(maxSize);
6.    }
7.
8.    public void enqueue(int value) {
9.        maxHeap.insert(value);
10.   }
11.
12.   public int dequeue() {
13.       return maxHeap.removeMax();
14.   }
15. }
```

16.

3. **Task:**
    - Test the priority queue by enqueuing elements [40, 20, 50, 10, 30] and dequeuing them, observing that elements are dequeued in descending order of priority.

Home Tasks: Further Applications of Priority Queues and Heaps

Home Task 1: Implement a Min Heap

1. **Objective:** Understand and implement a min heap, where the smallest element is always at the root.
2. **Task:** Write a MinHeap class with insert, removeMin, and heapifyUp/heapifyDown methods to maintain the min-heap property.
    - Implement insert and removeMin operations similar to the max heap but adjusted for the min-heap property.
3. **Example Output:**
    - **Input Operations:** insert(30), insert(10), insert(20), removeMin()
    - **Output after each operation:** [10, 20, 30] after insertions, [20, 30] after removal of minimum.

Home Task 2: Implement a Heap Sort

1. **Objective:** Use a max heap to implement heap sort, sorting an array in ascending order.
2. **Task:** Write a heapSort method that first builds a max heap from the array, then repeatedly removes the maximum element and places it at the end of the array, shrinking the heap each time.
3. **Example Output:**
    - **Input Array:** [12, 11, 13, 5, 6, 7]
    - **Sorted Array:** [5, 6, 7, 11, 12, 13]

Home Task 3: Implement a Median Finder Using Heaps

1. **Objective:** Use two heaps (max heap and min heap) to find the median of a dynamic set of integers.
2. **Task:** Write a MedianFinder class with methods to:
    - Insert elements into two heaps: a max heap for the lower half of numbers and a min heap for the upper half.
    - Balance the heaps to ensure the median can be found efficiently.
    - If the total number of elements is odd, return the root of the max heap; if even, return the average of the roots of both heaps.
3. **Example Output:**

- **Input Elements:** 5, 10, 15
- **Median:** 10
- **Input Elements:** 5, 10, 15, 20
- **Median:** (10 + 15) / 2 = 12.5

Submission Guidelines:

- Each task should be submitted as a separate Java file with comments explaining each part.
- Include screenshots of the output for each task.

Grading Rubric:

- **Lab Task:** Correct implementation of max heap operations and priority queue (50 points).
- **Home Task 1 (Min Heap):** Proper min-heap structure and operations (20 points).
- **Home Task 2 (Heap Sort):** Accurate sorting of array using max heap (15 points).
- **Home Task 3 (Median Finder):** Efficient use of two heaps for median calculation (15 points).

This lab and the home tasks provide students with a solid understanding of heaps and priority queues, including practical implementations and applications.

Lab Session 8: Open-Ended Lab

Build a Real-World Application with Custom Data Structures

Objective

To design and implement a real-world application by using custom data structures and algorithms. Students will learn how to choose the best data structures for their application needs, optimize algorithms for efficiency, and tackle real-world computational challenges.

Lab Structure

1. Proposal Submission (Problem Definition)

- **Task**: Each student or team will submit a brief proposal describing the problem they aim to solve, the proposed application, and the primary data structures and algorithms they plan to use.
- **Details to Include**:
  - Description of the problem and application.
  - Justification of chosen data structures (e.g., graphs, heaps, trees, hash tables).
  - Description of any algorithmic challenges they foresee and their plan to address them.
- **Deliverable**: A one-page proposal document that outlines the problem and initial design choices.

2. Data Structure Design

- **Task**: Based on their proposal, students will create custom data structures tailored to their application.
  - Implement key data structures from scratch if necessary, or customize existing ones (e.g., AVL trees, adjacency lists, priority queues).
  - Design data structures to handle data storage, retrieval, and manipulation efficiently.
- **Deliverable**: Code for the data structures, including comments on how they support the application's needs.

3. Algorithm Development

- **Task**: Implement algorithms to handle specific operations within the application.
  - Ensure that each algorithm is efficient and optimized for the data structure in use.
  - Focus on both standard algorithms (e.g., sorting, searching, shortest path) and any unique algorithms needed for the application.

- **Deliverable**: Code for each algorithm with documentation and an explanation of its time and space complexity.

4. Integration and Optimization

- **Task**: Integrate data structures and algorithms into a cohesive application.
  - Address and optimize any performance issues.
  - Run test cases to validate the application's functionality and efficiency.
- **Deliverable**: A working application with well-structured code and optimizations for both data structures and algorithms.

Suggested Application Ideas

Here are some application ideas to inspire students, though they are encouraged to come up with unique concepts:

Social Network Simulation

- Create a network of users, with connections represented by a graph.
- Implement algorithms for friend recommendations, shortest path to connect two users, and community detection.

Navigation System for a Map

- Design a city map using a graph, where intersections are nodes and roads are edges.
- Implement algorithms for shortest path (Dijkstra's or A*), fastest route based on real-time traffic, and alternative route suggestions.

Inventory Management System

- Create a system to manage inventory for a warehouse, with support for searching, sorting, and optimizing space.
- Use data structures like hash tables for quick item lookup and trees or heaps for sorting by priority.

Library Catalog System

- Implement a system to catalog books by various attributes (title, author, genre).
- Design search and sorting algorithms and provide recommendations based on a user's search history.

Dynamic Event Scheduler

- Create an event scheduling system that manages overlapping events and optimizes space/time usage.
- Use data structures like interval trees for conflict detection and heaps for prioritizing events.

File System Simulation

- Design a file system with directories and files, supporting operations like create, delete, search, and list directory contents.
- Use trees to model directory structures and hash tables for fast file searches.

Additional Features (Advanced)

Encourage students to implement additional features to increase complexity and depth:

1. **Real-Time Data Handling**: Implement dynamic data updates, such as real-time user connections in a social network or live traffic updates in a navigation system.
2. **Complex Search and Filter**: Add complex filtering, ranking, or multi-criteria search functions to allow users to find specific records based on custom parameters.
3. **Concurrency and Parallelism**: If the application requires high performance, implement concurrent data access or parallel processing to improve efficiency.
4. **Data Persistence**: Allow the application to save and retrieve data from a file or database, providing persistence across sessions.
5. **Data Visualization**: Visualize data structures (e.g., show graphs or trees) or outcomes to give a clear view of how data is being managed.

Assessment Criteria

1. **Problem Definition and Solution Design**
   - The clarity and relevance of the problem chosen.
   - Appropriateness of data structures and algorithms to solve the problem effectively.
2. **Implementation of Data Structures**
   - Correct implementation of required data structures.
   - Optimization for efficiency, readability, and maintainability.
3. **Algorithm Efficiency**
   - Effective and efficient implementation of required algorithms.
   - Appropriate choice and optimization of algorithms for specific operations.
4. **Code Quality and Documentation**
   - Clean and modular code, with clear comments explaining the purpose and function of each part.
   - Proper error handling and input validation.
5. **Optional Features and Creativity**
   - Successful implementation of additional features and enhancements.
   - Overall creativity and complexity of the solution.

Reflection Report

After completing the lab, students will write a brief report covering:

- Challenges faced and how they overcame them.
- An analysis of their data structure and algorithm choices.
- Reflections on what they learned about data structure and algorithm design.

Learning Outcomes

By completing this open-ended lab, students will:

- Develop critical skills in choosing and implementing efficient data structures.
- Gain experience in solving real-world problems using algorithms.
- Understand the importance of time and space complexity in algorithm design.
- Enhance problem-solving, debugging, and optimization skills.

This lab encourages students to engage deeply with data structures and algorithms, apply them in practical ways, and exercise creativity in building solutions, all of which are essential skills for advanced programming and software development.

Lab Session 9: Understanding Hashing

Objective:
This lab introduces students to the concept of hashing, a fundamental technique in data structures and algorithms used to store and retrieve data efficiently. Students will learn to implement a simple hash table, handle collisions, and understand hash functions.

Part 1: Introduction to Hashing and Hash Tables

1. **Explanation:**
   - **Hashing** is a technique that maps data to a fixed-size array (hash table) using a hash function.
   - A **hash function** computes an index for each key, mapping it to a position in the hash table.
   - **Collisions** occur when multiple keys hash to the same index. They are resolved using techniques like chaining or open addressing.
2. **Task:**
   - Discuss the importance of hashing in applications like databases and dictionaries for quick data access.

Part 2: Implementing a Simple Hash Function

1. **Explanation:**
   - A hash function takes an input (key) and produces an index. In this task, we'll create a simple hash function that returns key % tableSize.
2. **Steps:**
   - Implement a hashFunction method in a HashTable class.

```
class HashTable {
    private int tableSize;
    private Integer[] table;

    public HashTable(int size) {
        tableSize = size;
        table = new Integer[tableSize];
    }

    // Simple hash function
    private int hashFunction(int key) {
        return key % tableSize;
    }
}

```

3. **Task:**
   - Explain the hash function and demonstrate its output for different keys, such as 10, 15, and 21.

Part 3: Inserting Elements into the Hash Table

1. **Explanation:**
   - Inserting an element requires computing the hash for a key and placing it at the corresponding index. If a collision occurs, we can handle it using chaining (linked lists) or open addressing.
2. **Steps:**
   - Implement an insert method using open addressing (linear probing) to handle collisions.

```
public void insert(int key) {
    int index = hashFunction(key);
    int originalIndex = index;
    int i = 1;

    // Linear probing to handle collisions
    while (table[index] != null) {
        index = (originalIndex + i) % tableSize;
        i++;
        if (index == originalIndex) {
            System.out.println("Hash table is full, cannot insert " + key);
            return;
        }
    }
    table[index] = key;
    System.out.println("Inserted " + key + " at index " + index);
}

```

3. **Task:**
   - Insert elements [10, 15, 20, 25, 30] and observe how linear probing handles collisions.

Part 4: Searching for an Element in the Hash Table

1. **Explanation:**
   - Searching for an element involves computing the hash for the key and probing to find the element, if necessary.
2. **Steps:**
   - Implement a search method to find an element in the hash table.

```
public boolean search(int key) {
    int index = hashFunction(key);
    int originalIndex = index;
    int i = 1;

    while (table[index] != null) {
        if (table[index] == key) {
            System.out.println("Found " + key + " at index " + index);
            return true;
        }
        index = (originalIndex + i) % tableSize;
        i++;
        if (index == originalIndex) {
```

```
            break;
        }
    }
    System.out.println("Key " + key + " not found.");
    return false;
}
```

3. **Task:**
    - Test the search function for keys that exist (e.g., 15) and don't exist (e.g., 50) in the table.

Part 5: Removing an Element from the Hash Table

1. **Explanation:**
    - Removing an element involves locating it and setting its position in the array to null. To avoid breaking the hash table structure, we use a placeholder (e.g., -1) instead of setting it directly to null.
2. **Steps:**
    - Implement a delete method to remove an element.

```
public void delete(int key) {
    int index = hashFunction(key);
    int originalIndex = index;
    int i = 1;

    while (table[index] != null) {
        if (table[index] == key) {
            table[index] = -1;  // Mark as deleted
            System.out.println("Deleted " + key + " from index " + index);
            return;
        }
        index = (originalIndex + i) % tableSize;
        i++;
        if (index == originalIndex) {
            break;
        }
    }
    System.out.println("Key " + key + " not found.");
}
```

3. **Task:**
    - Delete an element (e.g., 15) and verify that it no longer appears in the hash table.

Home Tasks: Advanced Hashing Challenges

Home Task 1: Implementing Hash Table with Chaining

1. **Objective:** Use chaining to handle collisions instead of linear probing.
2. **Task:** Modify the HashTable class so that each array index contains a linked list. When a collision occurs, insert the key into the linked list at that index.

- Implement insert, search, and delete methods using chaining.

3. **Example Output:**
   - **Input Elements:** [10, 20, 15, 25]
   - **Output after Insertion:** Linked lists at collided indices showing the elements as chains.

Home Task 2: Counting Word Frequencies Using a Hash Table

1. **Objective:** Practice using a hash table to solve a real-world problem.
2. **Task:** Write a Java program that takes a string of words as input, splits it into individual words, and uses a hash table to count the frequency of each word.
   - Implement a hash table to store each word and its count.
3. **Example Output:**
   - **Input:** "apple banana apple orange banana apple"
   - **Output:** {"apple": 3, "banana": 2, "orange": 1}

Home Task 3: Implementing a Hash Table for Storing Student Records

1. **Objective:** Practice using hash tables to store objects with key-value pairs.
2. **Task:** Create a Student class with fields for ID, name, and grade. Use a hash table to store student records, using the student ID as the key.
   - Implement addStudent, removeStudent, and getStudent methods for managing student records.
3. **Example Output:**
   - **Add Student:** ID: 101, Name: John, Grade: A
   - **Retrieve Student:** For ID 101, output Name: John, Grade: A

Submission Guidelines:

- Each task should be submitted as a separate Java file with comments explaining each part of the code.
- Attach screenshots of the output for each task.

Grading Rubric:

- **Lab Task:** Completion and understanding of hash function, insert, search, and delete operations (50 points).
- **Home Task 1 (Chaining):** Proper handling of collisions using linked lists (20 points).
- **Home Task 2 (Word Frequency Counter):** Correct counting of word frequencies (15 points).
- **Home Task 3 (Student Records):** Accurate insertion, retrieval, and deletion of student records using hash table (15 points).

This lab and these home tasks will help students gain a practical understanding of hashing and hash tables, their real-world applications, and collision handling techniques.

Lab Session 10: Understanding Binary Search Trees

Objective:

This lab introduces students to Binary Search Trees (BSTs), a fundamental data structure that allows for efficient searching, insertion, and deletion. Students will learn to implement a BST in Java and perform basic operations.

Part 1: Introduction to Binary Search Trees

1. **Explanation:**
   - A Binary Search Tree (BST) is a binary tree where each node has at most two children, and all nodes follow these properties:
     - The left subtree contains only nodes with values less than the parent node's value.
     - The right subtree contains only nodes with values greater than the parent node's value.
2. **Task:**
   - Discuss the advantages of BSTs for data organization, efficient searching, and quick insertion and deletion operations.

Part 2: Creating a Node Class for the BST

1. **Explanation:**
   - Each node in a BST stores a data value and references to its left and right children.
2. **Steps:**
   - Create a Node class to represent each node in the BST.

```
1. class Node {
2.     int data;
3.     Node left, right;
4.
5.     public Node(int data) {
6.         this.data = data;
7.         left = right = null;
8.     }
9. }
10.
```

3. **Task:**
   - Explain the purpose of the left and right pointers, which point to the left and right children of the node, respectively.

Part 3: Creating the Binary Search Tree Class

1. **Explanation:**
   - The BST class will manage the root of the tree and implement operations like insertion, deletion, and search.
2. **Steps:**
   - Create a BST class with a root node and methods to insert, delete, and search nodes.

```
1. class BST {
2.     Node root;
3.
4.     // Constructor
5.     public BST() {
6.         root = null;
7.     }
8. }
9.
```

3. **Task:**
   - Describe how the root node acts as the starting point for all operations in the tree.

Part 4: Inserting Nodes into the BST

1. **Explanation:**
   - Inserting a node involves comparing the new value with existing nodes, moving left if the new value is smaller and right if it's larger, until an appropriate position is found.
2. **Steps:**
   - Implement the insert method to add nodes to the BST.

```
 1. public void insert(int data) {
 2.     root = insertRec(root, data);
 3. }
 4.
 5. // Recursive method to insert a new node
 6. private Node insertRec(Node root, int data) {
 7.     if (root == null) {
 8.         root = new Node(data);
 9.         return root;
10.     }
11.     if (data < root.data) {
12.         root.left = insertRec(root.left, data);
13.     } else if (data > root.data) {
14.         root.right = insertRec(root.right, data);
15.     }
16.     return root;
17. }
18.
```

3. **Task:**
   - Insert values [50, 30, 70, 20, 40, 60, 80] and visualize the tree structure. Explain how the tree's structure changes with each insertion.

Part 5: Searching for a Node in the BST

1. **Explanation:**
   - Searching in a BST is efficient because we can discard half of the remaining nodes at each step. Start at the root and go left if the value is smaller or right if it's larger.
2. **Steps:**
   - Implement the search method to find a value in the BST.

```
public boolean search(int key) {
    return searchRec(root, key);
}

private boolean searchRec(Node root, int key) {
    if (root == null) {
        return false;
    }
    if (root.data == key) {
        return true;
    }
    return key < root.data ? searchRec(root.left, key) : searchRec(root.right, key);
}

```

3. **Task:**
   - Search for values like 40 (exists) and 90 (does not exist) and observe the traversal path.

Part 6: Traversing the BST

1. **Explanation:**
   - Traversing a BST involves visiting each node in a specific order. Common traversal orders include:
     - **In-order (Left, Root, Right):** Yields nodes in ascending order.
     - **Pre-order (Root, Left, Right):** Useful for creating a copy of the tree.
     - **Post-order (Left, Right, Root):** Used to delete the tree.
2. **Steps:**
   - Implement the inOrder traversal method.

```
public void inOrder() {
    inOrderRec(root);
    System.out.println();
}

private void inOrderRec(Node root) {
    if (root != null) {
        inOrderRec(root.left);
        System.out.print(root.data + " ");
        inOrderRec(root.right);
    }
}

```

3. **Task:**

- Perform an in-order traversal on the BST created in Part 4 and verify that the output is sorted.

Part 7: Deleting a Node from the BST

1. **Explanation:**
   - Deleting a node requires handling three cases:
     - The node has no children (simply remove it).
     - The node has one child (replace the node with its child).
     - The node has two children (replace it with its in-order successor or predecessor).
2. **Steps:**
   - Implement the delete method to remove a node from the BST.

```
public void delete(int key) {
    root = deleteRec(root, key);
}

private Node deleteRec(Node root, int key) {
    if (root == null) return root;

    if (key < root.data) {
        root.left = deleteRec(root.left, key);
    } else if (key > root.data) {
        root.right = deleteRec(root.right, key);
    } else {
        if (root.left == null) return root.right;
        else if (root.right == null) return root.left;

        root.data = minValue(root.right);
        root.right = deleteRec(root.right, root.data);
    }
    return root;
}

private int minValue(Node root) {
    int min = root.data;
    while (root.left != null) {
        min = root.left.data;
        root = root.left;
    }
    return min;
}

```

3. **Task:**
   - Delete a node with two children (e.g., 50) and observe how the in-order successor replaces the deleted node.

Home Tasks: Advanced BST Challenges

Home Task 1: Checking if a Tree is a BST

1. **Objective:** Verify if a given binary tree satisfies the properties of a BST.
2. **Task:** Write a method isBST that checks whether a binary tree is a BST. Use recursion to verify that all nodes in the left subtree are less than the root and all nodes in the right subtree are greater.
3. **Example Output:**
    - **Input Tree:** [50, 30, 70, 20, 40, 60, 80]
    - **Output:** true (is a BST)

Home Task 2: Finding the Lowest Common Ancestor

1. **Objective:** Practice tree traversal by finding the Lowest Common Ancestor (LCA).
2. **Task:** Write a method findLCA that finds the LCA of two nodes in a BST. The LCA of two nodes is the lowest node that has both nodes as descendants.
    - For example, in a BST with nodes [50, 30, 70, 20, 40, 60, 80], the LCA of 20 and 40 is 30.
3. **Example Output:**
    - **Input Tree:** [50, 30, 70, 20, 40, 60, 80], Nodes: 20 and 40
    - **Output:** 30

Home Task 3: Converting BST to Sorted Doubly Linked List

1. **Objective:** Practice recursion and tree traversal by converting a BST to a sorted doubly linked list.
2. **Task:** Write a method bstToDoublyLinkedList that converts a BST to a sorted doubly linked list, where the left pointer of each node points to the previous node, and the right pointer points to the next node.
3. **Example Output:**
    - **Input Tree:** [10, 5, 15, 2, 8, 12, 20]
    - **Output List:** 2 <-> 5 <-> 8 <-> 10 <-> 12 <-> 15 <-> 20

Submission Guidelines:

- Each task should be submitted as a separate Java file with comments explaining each part.
- Attach screenshots of the output for each task.

Grading Rubric:

- **Lab Task:** Completion and understanding of BST operations (50 points).
- **Home Task 1 (Checking if Tree is BST):** Accurate verification of BST properties (20 points).
- **Home Task 2 (Finding LCA):** Correct identification of lowest common ancestor (15 points).

- **Home Task 3 (BST to Doubly Linked List):** Correct transformation of BST to a sorted doubly linked list (15 points).

This lab and these home tasks provide a comprehensive introduction to Binary Search Trees, allowing students to gain proficiency in tree structures, traversal techniques, and practical applications.

Lab Session 11: Understanding AVL Trees

Objective:

This lab introduces students to AVL Trees, a type of self-balancing Binary Search Tree (BST). Students will learn how AVL Trees maintain balance through rotations, ensuring efficient insertion, deletion, and searching operations.

Part 1: Introduction to AVL Trees

1. **Explanation:**
   - An AVL Tree is a self-balancing binary search tree where the difference between the heights of left and right subtrees (balance factor) is at most 1 for every node.
   - To maintain this balance, AVL Trees use rotations (single and double rotations) after insertions or deletions.
2. **Task:**
   - Discuss the importance of balancing in trees and how AVL Trees provide better performance than unbalanced BSTs by maintaining O(log n) height.

Part 2: Creating a Node Class for AVL Tree

1. **Explanation:**
   - Each node in an AVL Tree stores a data value, pointers to its left and right children, and a height attribute to help maintain balance.
2. **Steps:**
   - Create a Node class to represent each node in the AVL Tree.

```
class Node {
    int data, height;
    Node left, right;

    public Node(int data) {
        this.data = data;
        this.height = 1;  // Height is initially set to 1 for new nodes
    }
}

```

3. **Task:**
   - Explain the purpose of the height attribute and how it helps in calculating the balance factor for rotations.

Part 3: Creating the AVL Tree Class with Insert and Balance Operations

1. **Explanation:**
   - The AVLTree class will manage the tree's root and implement the insert, balance, and rotation methods. Each insertion may require rebalancing by applying rotations to keep the tree balanced.
2. **Steps:**
   - Create an AVLTree class with a root node and methods for insertion, rotations, and height calculations.

```
class AVLTree {
    Node root;

    // Utility method to get the height of the tree
    private int height(Node node) {
        return node == null ? 0 : node.height;
    }

    // Calculate the balance factor of the node
    private int getBalance(Node node) {
        return node == null ? 0 : height(node.left) - height(node.right);
    }
}

```

3. **Task:**
   - Explain the role of the height and getBalance methods in maintaining the AVL Tree properties.

Part 4: Implementing Rotations for Balancing

1. **Explanation:**
   - AVL Trees use rotations to maintain balance. There are four types of rotations:
     - **Right Rotation (RR):** Used when a node is left-heavy.
     - **Left Rotation (LL):** Used when a node is right-heavy.
     - **Left-Right Rotation (LR):** A left rotation followed by a right rotation.
     - **Right-Left Rotation (RL):** A right rotation followed by a left rotation.
2. **Steps:**
   - Implement methods for the rotations in the AVLTree class.

```
// Right Rotation
private Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;

    // Perform rotation
```

```
    x.right = y;
    y.left = T2;

    // Update heights
    y.height = Math.max(height(y.left), height(y.right)) + 1;
    x.height = Math.max(height(x.left), height(x.right)) + 1;

    // Return new root
    return x;
}

// Left Rotation
private Node leftRotate(Node x) {
    Node y = x.right;
    Node T2 = y.left;

    // Perform rotation
    y.left = x;
    x.right = T2;

    // Update heights
    x.height = Math.max(height(x.left), height(x.right)) + 1;
    y.height = Math.max(height(y.left), height(y.right)) + 1;

    // Return new root
    return y;
}

```

3. **Task:**
    - Explain each rotation and under what conditions each one is applied to maintain the AVL balance.

Part 5: Inserting a Node and Balancing the AVL Tree

1. **Explanation:**
    - Inserting a node requires balancing the tree if the balance factor exceeds ±1 after insertion. Based on the balance factor, an appropriate rotation is applied.
2. **Steps:**
    - Implement the insert method, which uses the balance factor to determine and perform the necessary rotation.

```
public Node insert(Node node, int data) {
    // 1. Perform the normal BST insertion
    if (node == null) {
        return new Node(data);
    }
    if (data < node.data) {
        node.left = insert(node.left, data);
    } else if (data > node.data) {
        node.right = insert(node.right, data);
    } else {
        return node;  // Duplicate data is not allowed in BST
    }

```

```
    // 2. Update height of this ancestor node
    node.height = 1 + Math.max(height(node.left), height(node.right));

    // 3. Get the balance factor
    int balance = getBalance(node);

    // 4. If the node is unbalanced, perform rotations
    if (balance > 1 && data < node.left.data) {
        return rightRotate(node);
    }
    if (balance < -1 && data > node.right.data) {
        return leftRotate(node);
    }
    if (balance > 1 && data > node.left.data) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }
    if (balance < -1 && data < node.right.data) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    return node;
}

```

3. **Task:**
    - Insert values [30, 20, 40, 10, 25, 50, 5] and observe how rotations keep the tree balanced.

Part 6: Traversing the AVL Tree

1. **Explanation:**
    - Traversing the AVL Tree is similar to traversing a BST. An in-order traversal will display the values in ascending order, verifying the AVL Tree's correctness as a BST.
2. **Steps:**
    - Implement an in-order traversal to print the elements of the tree.

```
public void inOrder(Node node) {
    if (node != null) {
        inOrder(node.left);
        System.out.print(node.data + " ");
        inOrder(node.right);
    }
}

```

3. **Task:**
    - Perform an in-order traversal of the AVL Tree created in Part 5 to ensure the output is sorted.

Home Tasks: Further Exploration of AVL Trees

Home Task 1: Deleting a Node from the AVL Tree

1. **Objective:** Practice removing nodes while maintaining the AVL property.
2. **Task:** Write a delete method for the AVL Tree that removes a specified node, rebalances the tree, and applies rotations as needed.
3. **Example Output:**
    - **Input Tree:** [30, 20, 40, 10, 25, 50, 5]
    - **Delete Node:** 20
    - **Output (In-order Traversal):** [5, 10, 25, 30, 40, 50]

Home Task 2: Finding the Height of the AVL Tree

1. **Objective:** Practice recursive tree traversal to determine the height of the AVL Tree.
2. **Task:** Write a method findHeight that returns the height of the AVL Tree from the root node. Test this with different AVL Trees.
3. **Example Output:**
    - **Input Tree:** [10, 20, 30, 40, 50, 25]
    - **Output Height:** 3

Home Task 3: Checking if a Binary Tree is an AVL Tree

1. **Objective:** Verify whether a given binary tree satisfies the properties of an AVL Tree.
2. **Task:** Write a method isAVLTree that checks if a binary tree is balanced according to AVL Tree rules (i.e., the balance factor of each node is between -1 and 1).
3. **Example Output:**
    - **Input Tree:** [30, 20, 40, 10, 25, 50, 5]
    - **Output:** true (the tree is an AVL Tree)

Submission Guidelines:

- Each task should be submitted as a separate Java file with comments explaining each part of the code.
- Attach screenshots of the output for each task.

Grading Rubric:

- **Lab Task:** Completion and understanding of AVL Tree operations, including insertion and balancing (50 points).
- **Home Task 1 (Delete from AVL Tree):** Accurate deletion and rebalancing of nodes (20 points).
- **Home Task 2 (Find Height):** Correct calculation of tree height (15 points).
- **Home Task 3 (Check if AVL Tree):** Correct verification of AVL Tree properties (15 points).

Lab Session 12: Understanding B+ Trees

Objective:

This lab introduces students to B+ Trees, an advanced tree structure commonly used in database indexing for efficient data storage and retrieval. Students will learn the basics of B+ Tree structure, insertion, and node splitting.

Part 1: Introduction to B+ Trees

1. **Explanation:**
   - A B+ Tree is a self-balancing tree data structure that maintains sorted data and allows efficient insertion, deletion, and search operations.
   - Unlike a Binary Search Tree or AVL Tree, each node in a B+ Tree can have more than two children, and all values are stored at the leaf level. Internal nodes store only keys for navigation.
2. **Task:**
   - Discuss the structure of B+ Trees and how they differ from BSTs and AVL Trees. Explain their use in databases due to their high fan-out and ability to handle large volumes of data.

Part 2: Creating the Node Class for B+ Tree

1. **Explanation:**
   - In a B+ Tree, nodes can be either **internal nodes** (which contain only keys for navigation) or **leaf nodes** (which contain actual data values).
   - Nodes have a maximum number of keys they can hold, typically defined as order - 1, where order is the maximum number of children each node can have.
2. **Steps:**
   - Define a Node class to represent nodes in the B+ Tree. Each node will contain a list of keys, a list of children for internal nodes, and a pointer to the next node for leaf nodes.

```
1. import java.util.ArrayList;
2. import java.util.List;
3.
4. class Node {
5.     List<Integer> keys;  // Holds the keys in the node
6.     List<Node> children; // Holds references to child nodes (only for internal nodes)
7.     Node next;           // Pointer to the next node (used in leaf nodes)
8.
9.     boolean isLeaf;
```

```
10.
11.     public Node(boolean isLeaf) {
12.         this.isLeaf = isLeaf;
13.         keys = new ArrayList<>();
14.         children = isLeaf ? null : new ArrayList<>();
15.         next = null;
16.     }
17. }
18.
```

3. **Task:**
    - Explain the isLeaf attribute, which helps differentiate between leaf and internal nodes. Leaf nodes contain data and a pointer to the next node, while internal nodes contain keys and child pointers.

Part 3: Creating the B+ Tree Class with Insertion and Splitting

1. **Explanation:**
    - A B+ Tree splits nodes when they reach their maximum capacity, creating a new node and redistributing keys. Internal nodes maintain pointers to child nodes to guide search operations.
2. **Steps:**
    - Create a BPlusTree class with an order attribute to define the tree's branching factor. Implement an insert method with the logic to handle insertion and node splitting.

```
 1. class BPlusTree {
 2.     private int order;
 3.     private Node root;
 4.
 5.     public BPlusTree(int order) {
 6.         this.order = order;
 7.         root = new Node(true);  // Start with an empty leaf node as root
 8.     }
 9.
10.     // Insert a key into the B+ Tree
11.     public void insert(int key) {
12.         Node newRoot = insertRecursive(root, key);
13.         if (newRoot != null) {
14.             root = newRoot;  // Update root if split occurs
15.         }
16.     }
17.
18.     private Node insertRecursive(Node node, int key) {
19.         if (node.isLeaf) {
20.             insertInLeaf(node, key);
21.             if (node.keys.size() >= order) {
22.                 return splitLeaf(node);
23.             }
24.         } else {
25.             int index = 0;
26.             while (index < node.keys.size() && key > node.keys.get(index)) {
```

```
                index++;
            }
            Node child = insertRecursive(node.children.get(index), key);
            if (child != null) {
                insertInInternal(node, child.keys.get(0), child);
                if (node.keys.size() >= order) {
                    return splitInternal(node);
                }
            }
        }
        return null;
    }
}

```

3. **Task:**
    - Explain the recursive insertRecursive method, which inserts keys in the appropriate leaf node and handles splitting if the maximum capacity is reached.

Part 4: Implementing Leaf and Internal Node Splitting

1. **Explanation:**
    - When a node reaches maximum capacity, it must be split. For leaf nodes, a new leaf node is created, and keys are redistributed. For internal nodes, a new internal node is created, and a middle key is pushed up to the parent.
2. **Steps:**
    - Implement splitLeaf and splitInternal methods in the BPlusTree class.

```
// Split a leaf node
private Node splitLeaf(Node leaf) {
    int midIndex = (order + 1) / 2;
    Node newLeaf = new Node(true);

    // Move half of the keys to the new leaf
    newLeaf.keys.addAll(leaf.keys.subList(midIndex, leaf.keys.size()));
    leaf.keys.subList(midIndex, leaf.keys.size()).clear();

    // Update pointers
    newLeaf.next = leaf.next;
    leaf.next = newLeaf;

    return newLeaf;
}

// Split an internal node
private Node splitInternal(Node node) {
    int midIndex = order / 2;
    Node newInternal = new Node(false);

    // Move half of the keys and children to the new internal node
    newInternal.keys.addAll(node.keys.subList(midIndex + 1, node.keys.size()));
    newInternal.children.addAll(node.children.subList(midIndex + 1, node.children.size()));

    // Clear old references
```

```
27.     node.keys.subList(midIndex, node.keys.size()).clear();
28.     node.children.subList(midIndex + 1, node.children.size()).clear();
29.
30.     return newInternal;
31. }
32.
```

3. **Task:**
    - Explain how keys and pointers are transferred to the new node during a split and how the split helps maintain the tree's balance.

Part 5: In-Order Traversal of the B+ Tree

1. **Explanation:**
    - Traversing a B+ Tree can be done from left to right at the leaf level, following the linked leaf nodes. This produces a sorted order of all keys in the tree.
2. **Steps:**
    - Implement an inOrderTraversal method to print all keys in ascending order.

```
 1. public void inOrderTraversal() {
 2.     Node current = root;
 3.     while (!current.isLeaf) {
 4.         current = current.children.get(0);  // Move to the leftmost leaf node
 5.     }
 6.     while (current != null) {
 7.         for (int key : current.keys) {
 8.             System.out.print(key + " ");
 9.         }
10.         current = current.next;
11.     }
12.     System.out.println();
13. }
14.
```

3. **Task:**
    - Insert a series of values [10, 20, 5, 15, 25, 30, 35] and perform an in-order traversal to ensure that the output is sorted.

Home Tasks: Further Exploration of B+ Trees

Home Task 1: Implement a Search Method in B+ Tree

1. **Objective:** Practice retrieving data efficiently by searching in the B+ Tree.
2. **Task:** Write a search method that searches for a specific key in the B+ Tree and returns true if the key is found or false otherwise.
    - Test the search method with keys that are present and absent in the tree.
3. **Example Output:**
    - **Input Tree:** [10, 20, 5, 15, 25, 30, 35]

- **Search Key:** 20 → Output: true
- **Search Key:** 40 → Output: false

Home Task 2: Implement Deletion in the B+ Tree

1. **Objective:** Learn how to delete keys while maintaining the B+ Tree properties.
2. **Task:** Write a delete method to remove a key from the B+ Tree, redistributing keys and merging nodes as necessary to maintain the B+ Tree properties.
   - Implement deletion logic for cases where nodes need to be merged if they go below the minimum number of keys.
3. **Example Output:**
   - **Input Tree:** [10, 20, 5, 15, 25, 30, 35]
   - **Delete Key:** 15
   - **Output (In-order Traversal):** [5, 10, 20, 25, 30, 35]

Home Task 3: Implement Range Search in the B+ Tree

1. **Objective:** Use B+ Tree structure to perform an efficient range search.
2. **Task:** Write a rangeSearch method that takes a range of values [start, end] and returns all keys within that range.
   - Start the search at the first key in the range and use the linked leaf nodes to gather all keys until the end of the range.
3. **Example Output:**
   - **Input Tree:** [10, 20, 5, 15, 25, 30, 35]
   - **Range Search (15 to 30):** Output: [15, 20, 25, 30]

Submission Guidelines:

- Each task should be submitted as a separate Java file with comments explaining each part of the code.
- Attach screenshots of the output for each task.

Grading Rubric:

- **Lab Task:** Completion and understanding of B+ Tree insertion and traversal (50 points).
- **Home Task 1 (Search Method):** Correctly finds keys in the tree (20 points).
- **Home Task 2 (Deletion):** Accurate deletion and rebalancing of nodes (15 points).
- **Home Task 3 (Range Search):** Efficiently retrieves all keys within a specified range (15 points).

This lab and these home tasks provide a comprehensive introduction to B+ Trees, covering key insertion, search, and deletion processes, along with practical applications in range queries and data retrieval.

Lab Session 13: Understanding Graph Data Structures

Objective:

This lab introduces students to the graph data structure, an essential concept in data structures and algorithms used to represent relationships between entities. Students will learn to represent a graph using adjacency lists and matrices, perform simple traversals, and understand fundamental graph operations.

Part 1: Introduction to Graphs

1. **Explanation:**
   - A **graph** is a collection of nodes (or vertices) connected by edges. Graphs can be **directed** (edges have a direction) or **undirected** (edges do not have a direction).
   - Graphs can be represented using **adjacency lists** or **adjacency matrices**.
2. **Task:**
   - Discuss the types of graphs (directed, undirected, weighted, unweighted) and their applications in real-world problems, such as social networks, road maps, and computer networks.

Part 2: Representing a Graph Using an Adjacency List

1. **Explanation:**
   - An adjacency list representation uses an array of lists, where each index represents a node, and the list at each index contains the nodes it's connected to.
2. **Steps:**
   - Create a Graph class with an adjacency list representation.

```
 1. import java.util.ArrayList;
 2. import java.util.LinkedList;
 3. import java.util.List;
 4.
 5. class Graph {
 6.     private int vertices;
 7.     private List<List<Integer>> adjacencyList;
 8.
 9.     public Graph(int vertices) {
10.         this.vertices = vertices;
11.         adjacencyList = new ArrayList<>(vertices);
12.         for (int i = 0; i < vertices; i++) {
13.             adjacencyList.add(new LinkedList<>());
```

```
        }
    }

    // Add an edge to the graph
    public void addEdge(int src, int dest) {
        adjacencyList.get(src).add(dest);
        adjacencyList.get(dest).add(src); // For undirected graph
    }
}

```

3. **Task:**
    - Explain the adjacency list structure and its advantages, especially in terms of memory efficiency for sparse graphs.

Part 3: Representing a Graph Using an Adjacency Matrix

1. **Explanation:**
    - An adjacency matrix is a 2D array where matrix[i][j] is 1 if there's an edge from node i to node j, and 0 otherwise. This representation is useful for dense graphs.
2. **Steps:**
    - Extend the Graph class with an adjacency matrix representation.

```
class GraphMatrix {
    private int vertices;
    private int[][] adjacencyMatrix;

    public GraphMatrix(int vertices) {
        this.vertices = vertices;
        adjacencyMatrix = new int[vertices][vertices];
    }

    // Add an edge to the graph
    public void addEdge(int src, int dest) {
        adjacencyMatrix[src][dest] = 1;
        adjacencyMatrix[dest][src] = 1; // For undirected graph
    }

    // Print adjacency matrix
    public void printMatrix() {
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                System.out.print(adjacencyMatrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}

```

3. **Task:**
    - Add a few edges and display the adjacency matrix for a sample graph. Discuss the time complexity for adding and checking edges with an adjacency matrix.

Part 4: Depth-First Search (DFS) Traversal

1. **Explanation:**
   - Depth-First Search (DFS) is a traversal technique that explores as far as possible along each branch before backtracking. DFS can be implemented using recursion (implicitly using a stack) or an explicit stack.
2. **Steps:**
   - Implement a DFS method using recursion in the Graph class with an adjacency list.

```
1. public void DFS(int startVertex) {
2.     boolean[] visited = new boolean[vertices];
3.     DFSRecursive(startVertex, visited);
4.     System.out.println();
5. }
6.
7. private void DFSRecursive(int vertex, boolean[] visited) {
8.     visited[vertex] = true;
9.     System.out.print(vertex + " ");
10.    for (int adjVertex : adjacencyList.get(vertex)) {
11.        if (!visited[adjVertex]) {
12.            DFSRecursive(adjVertex, visited);
13.        }
14.    }
15. }
16.
```

3. **Task:**
   - Perform a DFS traversal starting from a specific vertex, such as 0, and observe the order in which nodes are visited.

Part 5: Breadth-First Search (BFS) Traversal

1. **Explanation:**
   - Breadth-First Search (BFS) is a traversal technique that explores nodes layer by layer. It uses a queue to keep track of nodes at the current level.
2. **Steps:**
   - Implement a BFS method in the Graph class.

```
1. import java.util.LinkedList;
2. import java.util.Queue;
3.
4. public void BFS(int startVertex) {
5.     boolean[] visited = new boolean[vertices];
6.     Queue<Integer> queue = new LinkedList<>();
7.
8.     visited[startVertex] = true;
9.     queue.add(startVertex);
10.
11.    while (!queue.isEmpty()) {
12.        int vertex = queue.poll();
```

```
13.         System.out.print(vertex + " ");
14.         for (int adjVertex : adjacencyList.get(vertex)) {
15.             if (!visited[adjVertex]) {
16.                 visited[adjVertex] = true;
17.                 queue.add(adjVertex);
18.             }
19.         }
20.     }
21.     System.out.println();
22. }
23.
```

3. **Task:**
    - Perform a BFS traversal starting from a specific vertex, such as 0, and observe the order in which nodes are visited.

Home Tasks: Advanced Graph Operations

Home Task 1: Detecting a Cycle in an Undirected Graph

1. **Objective:** Practice DFS traversal to detect cycles in an undirected graph.
2. **Task:** Write a method isCyclic in the Graph class to detect if there is a cycle in an undirected graph using DFS.
    - Use a helper method to track visited nodes and parent nodes.
3. **Example Output:**
    - **Input Graph:** [0-1, 1-2, 2-0] (cycle present)
    - **Output:** true

Home Task 2: Finding the Shortest Path Using BFS

1. **Objective:** Use BFS traversal to find the shortest path in an unweighted graph.
2. **Task:** Write a method shortestPath that calculates the shortest path from a given start vertex to all other vertices using BFS. Return an array where the index represents the vertex and the value represents the shortest distance from the start vertex.
3. **Example Output:**
    - **Input Graph:** [0-1, 1-2, 0-2, 2-3], **Start Vertex:** 0
    - **Output:** [0, 1, 1, 2] (shortest distance from vertex 0 to all other vertices)

Home Task 3: Topological Sorting of a Directed Acyclic Graph (DAG)

1. **Objective:** Practice topological sorting, a linear ordering of vertices in a directed acyclic graph.
2. **Task:** Write a method topologicalSort in the Graph class to perform a topological sort using DFS. Return a list of vertices in topologically sorted order.
3. **Example Output:**
    - **Input Graph:** [5 -> 2, 5 -> 0, 4 -> 0, 4 -> 1, 2 -> 3, 3 -> 1]

- **Output:** [5, 4, 2, 3, 1, 0] (one possible topological order)

Submission Guidelines:

- Each task should be submitted as a separate Java file with comments explaining each part of the code.
- Attach screenshots of the output for each task.

Grading Rubric:

- **Lab Task:** Completion and understanding of graph representations and traversals (50 points).
- **Home Task 1 (Cycle Detection):** Correctly detects cycles in an undirected graph (20 points).
- **Home Task 2 (Shortest Path):** Correctly calculates the shortest path from a start vertex using BFS (15 points).
- **Home Task 3 (Topological Sort):** Correct topological order for a DAG (15 points).

This lab and these home tasks provide a comprehensive introduction to graphs, covering various representations, traversal techniques, and practical applications in tasks like cycle detection and topological sorting.

Complex Computing Activity

Smart City Traffic Management System Simulation

Objective

To design and implement a simulation of a **Smart City Traffic Management System** using advanced data structures and algorithms. The system will manage and optimize traffic flow in a simulated city with multiple intersections, roads, and vehicles. Students will learn to apply various data structures like graphs, heaps, and queues, alongside pathfinding algorithms and optimization techniques to create an efficient traffic management solution.

Problem Overview

Design a program that simulates the movement of vehicles across a city represented as a **graph**. Each node in the graph represents an **intersection**, and each edge represents a **road** connecting intersections. Vehicles must be directed to their destinations while minimizing overall traffic congestion and travel time.

Key Requirements:

1. **City Representation (Graph)**
    - Model the city as a graph where:
        - **Nodes** represent intersections.
        - **Edges** represent roads with properties such as length (distance) and traffic load (current number of vehicles).
    - Use an **adjacency list** or **adjacency matrix** to represent the graph.
2. **Vehicles**
    - Each vehicle has:
        - A unique identifier.
        - A start and destination intersection.
        - A preferred route (which can change dynamically based on traffic conditions).
    - Vehicles should move in discrete time steps, progressing from one node to another across edges in the graph.
3. **Traffic Lights and Signals**
    - Each intersection has a **traffic light** that controls the flow of vehicles from one direction at a time.

- Traffic lights cycle in a way that minimizes waiting time, adapting to current traffic conditions at each intersection.

4. **Pathfinding and Route Optimization**
   - Implement pathfinding algorithms (such as **Dijkstra's Algorithm** or **A*** algorithm) to calculate the shortest or fastest route from a vehicle's starting point to its destination.
   - Allow routes to be recalculated dynamically if traffic conditions change (e.g., a shorter or less congested route becomes available).
5. **Traffic Load and Congestion Management**
   - Track traffic load on each road segment, updating as vehicles move through.
   - Implement a congestion management system:
     - Use a **priority queue** to handle road segments with the highest congestion and allow the system to redirect traffic.
     - Set thresholds for congestion, after which vehicles are redirected to alternative routes.
     - Use heuristics or algorithms to balance load across roads, reducing bottlenecks.

Additional Requirements (Advanced)

To make the simulation more complex and realistic, include one or more of the following features:

1. **Emergency Vehicle Priority**:
   - Implement priority routing for emergency vehicles that overrides regular traffic rules to clear their path.
   - Emergency vehicles can "pause" regular vehicles at intersections to allow faster movement to their destinations.
2. **Weather and Time of Day Effects**:
   - Simulate effects of different weather conditions or time of day on traffic speed and visibility (e.g., slowing vehicles in rain).
   - Adjust pathfinding algorithms to factor in these conditions when calculating optimal routes.
3. **Adaptive Traffic Signal Optimization**:
   - Implement machine learning (optional) or rule-based algorithms to adjust traffic signal timings dynamically based on real-time traffic data to optimize flow.
4. **Vehicle-to-Infrastructure Communication**:
   - Implement a communication model where vehicles share information with nearby intersections to allow for preemptive adjustments in routes or traffic signals.
5. **Traffic Pattern Analysis and Reporting**:
   - Collect data on peak hours, commonly congested routes, and average travel times.
   - Generate a summary report or visualization for insights into traffic patterns.

Suggested Approach and Steps

Step 1: Design and Plan

1. **Define Classes and Data Structures**:
    - **Graph**: Use adjacency lists or matrices to represent intersections and roads.
    - **Vehicle**: Track each vehicle's ID, location, and destination.
    - **TrafficLight**: Manage state and timing for each intersection's traffic light.
    - **Priority Queue**: For handling traffic segments with high congestion levels.
2. **Define Methods and Functions**:
    - calculateRoute(): Pathfinding function for determining the optimal path.
    - updateTraffic(): Adjusts vehicle locations and updates congestion data.
    - adjustTrafficLights(): Changes light states at intersections based on traffic.
3. **Model Traffic and Time**:
    - Define a way to simulate time steps, allowing vehicles to move and traffic lights to change at each step.

Step 2: Implement Core Components

1. **Graph and Traffic Management System**:
    - Build the graph and initialize intersections, roads, and traffic lights.
    - Implement traffic flow logic, updating vehicle positions and handling congestions.
2. **Pathfinding and Adaptive Routing**:
    - Implement Dijkstra's or A* algorithm for initial routing.
    - Add adaptive routing to recalculate paths dynamically when congestion is detected.
3. **Traffic Lights**:
    - Code traffic lights to cycle through states.
    - Optimize their timing to balance waiting times and improve flow at each intersection.

Step 3: Simulate and Optimize

1. **Run the Simulation**:
    - Place a set number of vehicles on the city map with random starting points and destinations.
    - Run the simulation in time steps, updating each vehicle's position and adjusting routes as needed.
2. **Handle Edge Cases**:
    - Test for cases like high congestion on all routes, intersections with heavy inflow, or emergency vehicle prioritization.
3. **Optimize and Enhance**:
    - Fine-tune parameters for traffic light timing, rerouting frequency, and congestion thresholds.
    - Experiment with additional features (e.g., weather impact, adaptive signal optimization).

Assessment Criteria

1. **Correctness and Efficiency**:
   - Accurate implementation of the graph and pathfinding algorithms.
   - Efficient handling of data structures and minimization of computational overhead.
2. **Use of Data Structures**:
   - Proper use of graphs, priority queues, and other data structures for optimization.
   - Clear and modular code structure for each component.
3. **Handling of Complex Scenarios**:
   - Effective adaptive routing and congestion management.
   - Successful implementation of additional requirements (if attempted).
4. **Documentation and Code Quality**:
   - Clean, readable code with comments explaining each section.
   - Clear documentation of classes, functions, and algorithms used.
5. **Creativity and Additional Features**:
   - Successfully implemented advanced features, such as emergency routing or adaptive signals.
   - Creative approaches to problem-solving and system enhancements.

Learning Outcomes

By completing this activity, students will:

- Gain hands-on experience with graph structures, pathfinding, and priority queues.
- Learn how to manage complex, dynamic data in a real-world context.
- Improve their skills in algorithmic thinking, optimization, and simulation modeling.
- Understand the interplay between data structures and algorithms in solving practical, large-scale problems.