Q.1 Array and linked list implementation of list

Theory

Arrays and linked lists are fundamental data structures used in programming.

Arrays:

- Contiguous memory blocks storing elements of the same type.
- Efficient random access via indexing.
- Fixed size, making resizing expensive.

Linked Lists:

- Consist of nodes with data and a reference to the next node.
- Efficient insertion and deletion.
- No direct access by index, requiring sequential traversal.

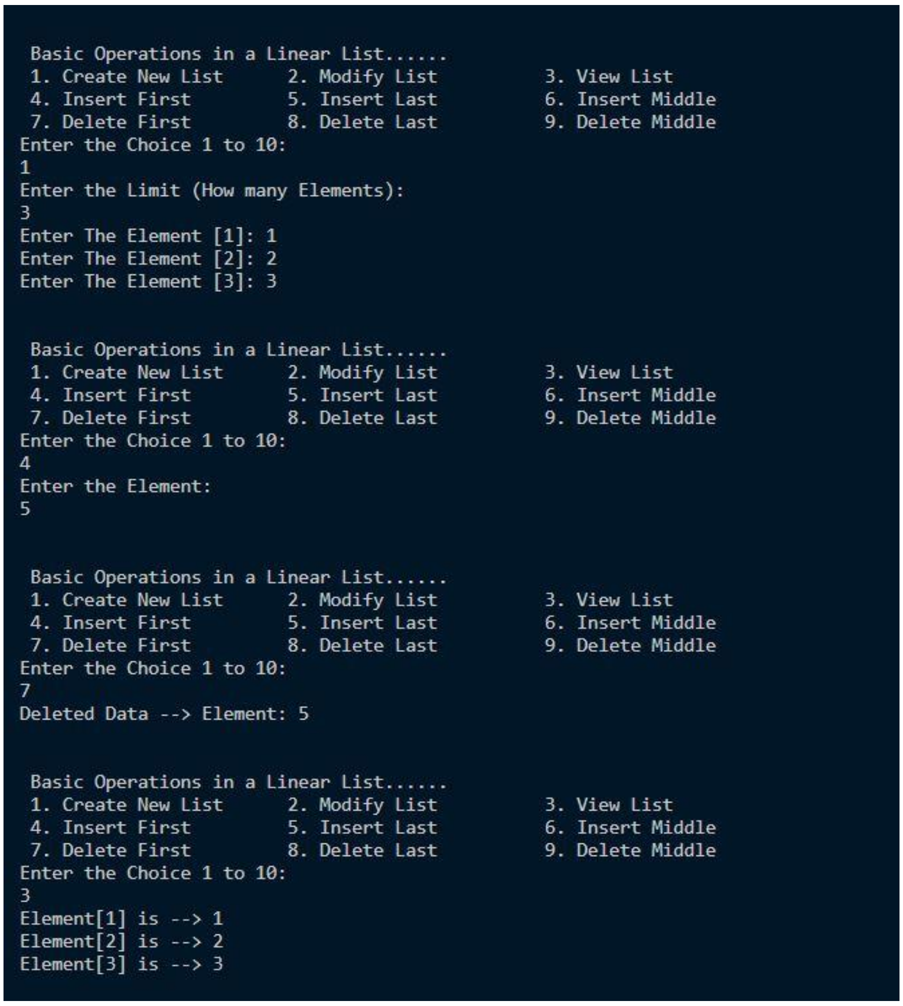
Use Cases:

- **Arrays**: Preferred for frequent random access or fixed-size collections.
- **Linked Lists**: Suitable for dynamic resizing, efficient insertion/deletion, and memory efficiency.

Java code:

```
import java.util.Scanner;

public class LinearListOperations {

    private static final int LIST_SIZE = 30;

    private static int[] element;

    private static int top = -1;


    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        while (true) {

            System.out.println("\n\n Basic Operations in a Linear List......");

            System.out.println(" 1. Create New List \t 2. Modify List \t 3. View List");

            System.out.println(" 4. Insert First \t 5. Insert Last \t 6. Insert Middle");

            System.out.println(" 7. Delete First \t 8. Delete Last \t 9. Delete Middle");

            System.out.println("Enter the Choice 1 to 10: ");

            int ch = scanner.nextInt();

            switch (ch) {

                case 1:

                    top = -1;

                    System.out.println("Enter the Limit (How many Elements):");
        int n = scanner.nextInt();
        element = new int[LIST_SIZE];
        for (int i = 0; i < n; i++) {
            System.out.print("Enter The Element [" + (i + 1) + "]: ");
            element[++top] = scanner.nextInt();
        }
        break;
    case 2:
        if (top == -1) {
            System.out.println("Linear List is Empty:");
            break;
        }
        System.out.println("Enter the Element for Modification:");
        int moddata = scanner.nextInt();
        int found = 0;
        for (int i = 0; i <= top; i++) {
            if (element[i] == moddata) {
                found = 1;
                System.out.println("Enter The New Element:");
                element[i] = scanner.nextInt();
                break;
            }
        }
        if (found == 0)
            System.out.println("Element " + moddata + " not found");
        break;
    case 3:
        if (top == -1)
            System.out.println("\n Linear List is Empty:");
        else if (top == LIST_SIZE - 1)
            System.out.println("\n Linear List is Full:");
        for (int i = 0; i <= top; i++)
            System.out.println("Element[" + (i + 1) + "] is --> " + element[i]);
        break;
    case 4:
        if (top == LIST_SIZE - 1) {
            System.out.println("Linear List is Full:");
            break;
        }
        top++;
        for (int i = top; i > 0; i--)
            element[i] = element[i - 1];
        System.out.println("Enter the Element:");
        element[0] = scanner.nextInt();
        break;
    case 5:
        if (top == LIST_SIZE - 1) {
            System.out.println("Linear List is Full:");
            break;
        }
        System.out.println("Enter the Element:");
        element[++top] = scanner.nextInt();
        break;
    case 6:
        if (top == LIST_SIZE - 1)
            System.out.println("Linear List is Full:");
        else if (top == -1)
            System.out.println("Linear List is Empty.");
        else {
            found = 0;
            System.out.println("Enter the Element after which the insertion is to be made:");
            int insdata = scanner.nextInt();
            for (int i = 0; i <= top; i++)
                if (element[i] == insdata) {
                    found = 1;
                    top++;
                    for (int j = top; j > i; j--)
                        element[j] = element[j - 1];
                    System.out.println("Enter the Element:");
                    element[i + 1] = scanner.nextInt();
                break;
            }
        if (found == 0)
            System.out.println("Element " + insdata + " Not Found");
    }
    break;
case 7:
    if (top == -1) {
        System.out.println("Linear List is Empty:");
        break;
    }
    System.out.println("Deleted Data --> Element: " + element[0]);
    top--;
    for (int i = 0; i <= top; i++)
        element[i] = element[i + 1];
    break;
case 8:
    if (top == -1)
        System.out.println("Linear List is Empty:");
    else
        System.out.println("Deleted Data --> Element: " + element[top--]);
    break;
case 9:
    if (top == -1) {
        System.out.println("Linear List is Empty:");
        break;
    }
    System.out.println("Enter the Element for Deletion:");
    int deldata = scanner.nextInt();
    found = 0;
    for (int i = 0; i <= top; i++)
        if (element[i] == deldata) {
            found = 1;
            System.out.println("Deleted Data --> Element: " + element[i]);
            top--;
```

```
                for (int j = i; j <= top; j++)
                    element[j] = element[j + 1];
                break;
            }
        if (found == 0)
            System.out.println("Element " + deldata + " Not Found");
        break;
    default:
        System.out.println("End Of Run Of Your Program.........");
        scanner.close();
        System.exit(0);
}}}}
```



```
 Basic Operations in a Linear List......
 1. Create New List      2. Modify List          3. View List
 4. Insert First         5. Insert Last          6. Insert Middle
 7. Delete First         8. Delete Last          9. Delete Middle
Enter the Choice 1 to 10:
1
Enter the Limit (How many Elements):
3
Enter The Element [1]: 1
Enter The Element [2]: 2
Enter The Element [3]: 3


 Basic Operations in a Linear List......
 1. Create New List      2. Modify List          3. View List
 4. Insert First         5. Insert Last          6. Insert Middle
 7. Delete First         8. Delete Last          9. Delete Middle
Enter the Choice 1 to 10:
4
Enter the Element:
5


 Basic Operations in a Linear List......
 1. Create New List      2. Modify List          3. View List
 4. Insert First         5. Insert Last          6. Insert Middle
 7. Delete First         8. Delete Last          9. Delete Middle
Enter the Choice 1 to 10:
7
Deleted Data --> Element: 5


 Basic Operations in a Linear List......
 1. Create New List      2. Modify List          3. View List
 4. Insert First         5. Insert Last          6. Insert Middle
 7. Delete First         8. Delete Last          9. Delete Middle
Enter the Choice 1 to 10:
3
Element[1] is --> 1
Element[2] is --> 2
Element[3] is --> 3
```

In Conclusion:

The choice between array and linked list implementations depends on application needs, such as access frequency, insertion, and deletion operations, and memory constraints. Arrays excel in scenarios requiring frequent index-based access due to their constant-time element retrieval, but resizing them can be costly. Linked lists offer dynamic memory allocation with efficient insertion and deletion, though they lack efficient random access and consume more memory due to pointer storage. Ultimately, the specific use case determines the most suitable data structure.

Q.2 Stack operations and Queue operations

1. Stack operations using Array

Theory:

- **Stack Operations**

A stack is a LIFO (Last-In-First-Out) data structure where the last element added is the first one to be removed. In Java, the `Stack` class is used to implement stacks.

Key Operations:

- **Push**: Adds an element to the top.
- **Pop**: Removes and returns the top element.
- **Peek**: Returns the top element without removing it.
- **IsEmpty**: Checks if the stack is empty.
- **Size**: Returns the number of elements.

- **Queue Operations**

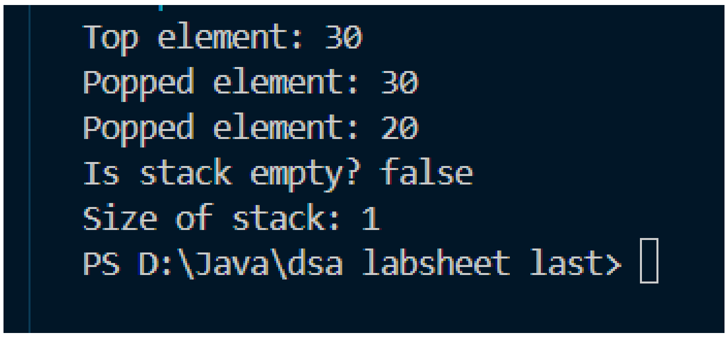
A queue is a FIFO (First-In-First-Out) data structure where the first element added is the first one to be removed. Java uses the `Queue` interface, commonly implemented by the `LinkedList` class.

Key Operations:

- **Add (Enqueue)**: Adds an element to the end.
- **Poll (Dequeue)**: Removes and returns the front element.
- **Peek**: Returns the front element without removing it.
- **IsEmpty**: Checks if the queue is empty.
- **Size**: Returns the number of elements.

Code using Java : stack operations

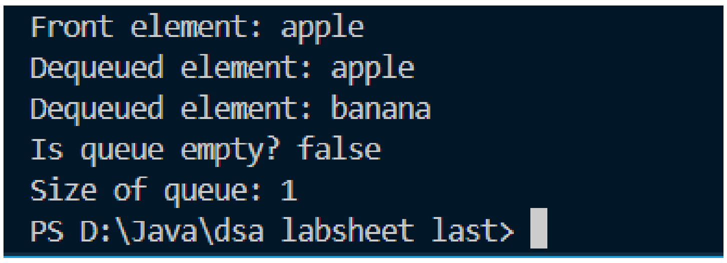
```
import java.util.Stack;

public class StackExample {

   public static void main(String[] args) {

      Stack<Integer> stack = new Stack<>();

      stack.push(10);

      stack.push(20);

      stack.push(30);

      System.out.println("Top element: " + stack.peek());

      System.out.println("Popped element: " + stack.pop());

      System.out.println("Popped element: " + stack.pop());

      System.out.println("Is stack empty? " + stack.isEmpty());

      System.out.println("Size of stack: " + stack.size());
}
}
```



```
Top element: 30
Popped element: 30
Popped element: 20
Is stack empty? false
Size of stack: 1
PS D:\Java\dsa labsheet last>
```

Code using Java : Queue operations

```
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.add("apple");
        queue.add("banana");
        queue.add("cherry");
        System.out.println("Front element: " + queue.peek());
        System.out.println("Dequeued element: " + queue.poll());
        System.out.println("Dequeued element: " + queue.poll());
        System.out.println("Is queue empty? " + queue.isEmpty());
        System.out.println("Size of queue: " + queue.size());
    }}
```



```
Front element: apple
Dequeued element: apple
Dequeued element: banana
Is queue empty? false
Size of queue: 1
PS D:\Java\dsa labsheet last>
```

Conclusion:

Stacks and queues are essential data structures in Java used for different purposes: **Stacks** follow a LIFO (Last-In-First-Out) approach, making them ideal for scenarios like reversing data or managing function calls, with operations like push(), pop(), and peek(). **Queues** follow a FIFO (First-In-First-Out) model, suitable for tasks like scheduling or buffering, with operations like add(), poll(), and peek(). Choosing between them depends on whether you need to process elements in reverse order or sequentially.

Q3. Recursion

Theory:

Recursion is a technique where a method calls itself to solve a problem by breaking it down into simpler subproblems. It involves two main parts: the **base case**, which stops the recursion, and the **recursive case**, which moves towards the base case.

Common Recursive Problems:

- **Factorial Calculation:** Computes the product of all positive integers up to a number. It multiplies the number by the factorial of the number minus one until it reaches zero.
- **Fibonacci Sequence:** Generates numbers where each term is the sum of the two preceding ones, starting with 0 and 1.
- **Tower of Hanoi:** A puzzle involving moving disks between rods following specific rules, solved by moving smaller groups of disks recursively.

Advantages:

- Simplifies complex problems by breaking them into smaller tasks.
- Useful for problems involving sequences, tree structures, and divide-and-conquer strategies.

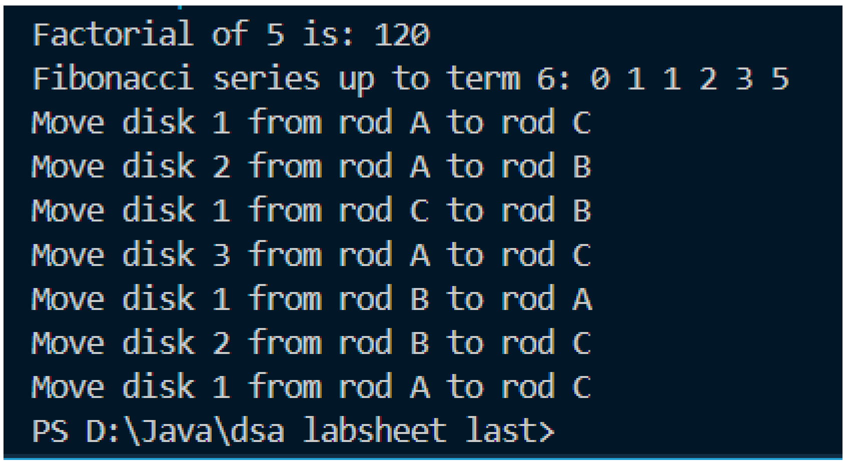
Disadvantages:

- Can use more memory and be less efficient due to repeated function calls.
- Risk of stack overflow if the recursion depth is too large.

Java code for recursion:

```
public class RecursionExamples {


  public static void main(String[] args) {
    System.out.println("Factorial of 5 is: " + factorial(5));


    int term = 6;
    System.out.print("Fibonacci series up to term " + term + ": ");
    for (int i = 0; i < term; i++) {
      System.out.print(fibonacci(i) + " ");
    }
    System.out.println();


    towerOfHanoi(3, 'A', 'C', 'B');
  }


  public static int factorial(int n) {
    if (n == 0)
```

```
            return 1;
        return n * factorial(n - 1);
    }

    public static int fibonacci(int n) {
        if (n <= 1)
            return n;
        return fibonacci(n - 1) + fibonacci(n - 2);
    }

    public static void towerOfHanoi(int n, char fromRod, char toRod, char auxRod) {
        if (n == 1) {
            System.out.println("Move disk 1 from rod " + fromRod + " to rod " + toRod);
            return;
        }
        towerOfHanoi(n - 1, fromRod, auxRod, toRod);
        System.out.println("Move disk " + n + " from rod " + fromRod + " to rod " + toRod);
        towerOfHanoi(n - 1, auxRod, toRod, fromRod);
    }
}
```



```
Factorial of 5 is: 120
Fibonacci series up to term 6: 0 1 1 2 3 5
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
PS D:\Java\dsa labsheet last>
```

Conclusion:

Recursion is a powerful technique that solves problems by breaking them into smaller subproblems and solving each recursively. It is used in scenarios like calculating factorials, generating Fibonacci sequences, and solving puzzles like the Tower of Hanoi. While recursion simplifies problem-solving and algorithm design, it can be memory-intensive and less efficient due to repeated function calls.

Q.4 Linked list implementation of Stack and Queue

Linked List is a dynamic data structure composed of nodes, each containing data and a reference to the next node. It is useful for implementing various abstract data types like Stacks and Queues.

Stack Implementation

A Stack is a Last-In-First-Out (LIFO) data structure where the most recently added element is the first to be removed. It supports two main operations:

- **Push:** Add an element to the top of the stack.
- **Pop:** Remove and return the element from the top of the stack.
- **Peek:** View the element at the top without removing it.
- **IsEmpty:** Check if the stack is empty.
- **Size:** Get the number of elements in the stack.

Linked List Operations:
In the linked list implementation of a stack:

- **Push:** Add elements to the beginning of the list.
- **Pop:** Remove elements from the beginning of the list.
- **Peek:** Access the first element of the list.

Advantages:

- **Dynamic Size:** Can grow or shrink as needed.
- **Efficient Operations:** Push, Pop, and Peek operations are performed in constant time.

Uses:

- Function call management
- Undo mechanisms in applications
- Expression evaluation

Queue Implementation

A Queue is a First-In-First-Out (FIFO) data structure where the first element added is the first to be removed. It supports two main operations:

- **Enqueue:** Add an element to the rear of the queue.
- **Dequeue:** Remove and return the element from the front of the queue.
- **Peek:** View the element at the front without removing it.
- **IsEmpty:** Check if the queue is empty.
- **Size:** Get the number of elements in the queue.

Linked List Operations:
In the linked list implementation of a queue:

- **Enqueue:** Add elements to the end of the list.
- **Dequeue:** Remove elements from the beginning of the list.
- **Peek:** Access the first element of the list.

Advantages:

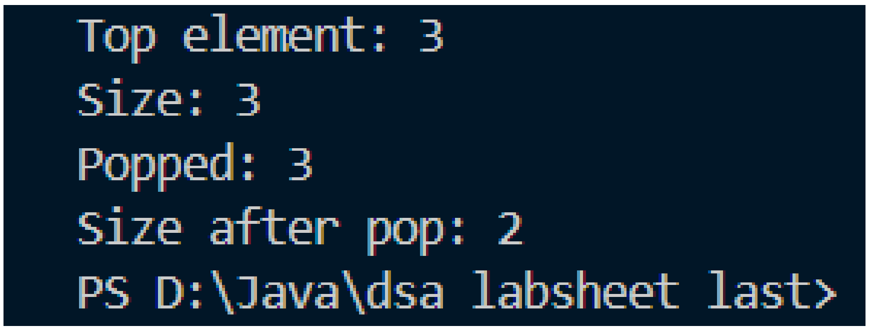
- **Dynamic Size:** Can grow or shrink as needed.
- **Efficient Operations:** Enqueue and Dequeue operations are performed in constant time.

Uses:

- Task scheduling
- Buffer management
- Breadth-first search in algorithms

- ## Java code: Linked list implementation of stack

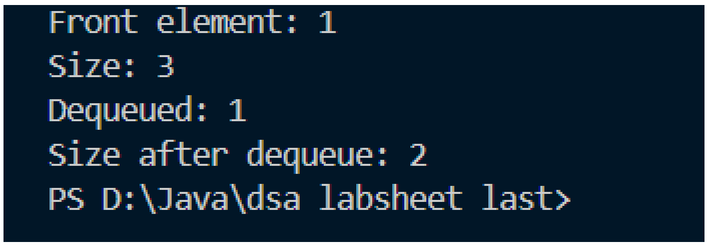
```
import java.util.LinkedList;


public class LinkedListStack<T> {
    private LinkedList<T> list = new LinkedList<>();


    public void push(T item) {
        list.addFirst(item);
    }


    public T pop() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return list.removeFirst();
    }
    public T peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return list.getFirst();
    }


    public boolean isEmpty() {
        return list.isEmpty();
    }


    public int size() {
        return list.size();
    }

    public static void main(String[] args) {
        LinkedListStack<Integer> stack = new LinkedListStack<>();
        stack.push(1);
        stack.push(2);
        stack.push(3);

        System.out.println("Top element: " + stack.peek()); // 3
        System.out.println("Size: " + stack.size()); // 3

        System.out.println("Popped: " + stack.pop()); // 3
        System.out.println("Size after pop: " + stack.size()); // 2
    }
}
```



```
Top element: 3
Size: 3
Popped: 3
Size after pop: 2
PS D:\Java\dsa labsheet last>
```

- **Java code: Linked list implementation of Queue**

```
import java.util.LinkedList;
public class LinkedListQueue<T> {
    private LinkedList<T> list = new LinkedList<>();
    public void enqueue(T item) {
        list.addLast(item);
    }
    public T dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return list.removeFirst();
    }
    public T peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return list.getFirst();
    }
    public boolean isEmpty() {
        return list.isEmpty();
    }
    public int size() {
        return list.size();
    }
    public static void main(String[] args) {
        LinkedListQueue<Integer> queue = new LinkedListQueue<>();
        queue.enqueue(1);
        queue.enqueue(2);
        queue.enqueue(3);
        System.out.println("Front element: " + queue.peek());
        System.out.println("Size: " + queue.size());
        System.out.println("Dequeued: " + queue.dequeue());

        System.out.println("Size after dequeue: " + queue.size());

    }

}
```



```
Front element: 1
Size: 3
Dequeued: 1
Size after dequeue: 2
PS D:\Java\dsa labsheet last>
```

Conclusion:

LinkedList implementations of Stacks and Queues offer dynamic and efficient solutions for managing collections of elements.

- **Stack:** Follows a Last-In-First-Out (LIFO) approach, with push and pop operations at the top. Ideal for recursion management and undo functionality.
- **Queue:** Follows a First-In-First-Out (FIFO) approach, with enqueue and dequeue operations at the rear and front. Suited for task scheduling and buffering.

Both data structures leverage the LinkedList class for dynamic resizing and efficient operations, making them versatile tools in algorithm design and system management.

Q5. Binary search tree

A Binary Search Tree (BST) is a binary tree where each node's left child has a smaller value, and the right child has a larger value.

Key Operations:

1. **Insert:**
   - Adds a value while preserving BST order.
   - Complexity: O(log n) average; O(n) worst case.
2. **Search:**
   - Checks if a value exists in the tree.
   - Complexity: O(log n) average; O(n) worst case.
3. **Delete:**
   - Removes a value and reorganizes the tree to maintain order.
   - Complexity: O(log n) average; O(n) worst case.

Applications: Efficient for sorting, searching, and maintaining dynamic data sets.

Operations:

1. Searching: Recursively traverse the tree to find the target value.
2. Insertion: Traverse the tree to find the appropriate position for the new node and add it while maintaining the ordering property.
3. Deletion: Identify the node to remove and rearrange the tree structure if necessary to maintain the ordering property.
4. Traversal: Visit all nodes in specific orders, such as inorder, preorder, and postorder, to process or display their values.

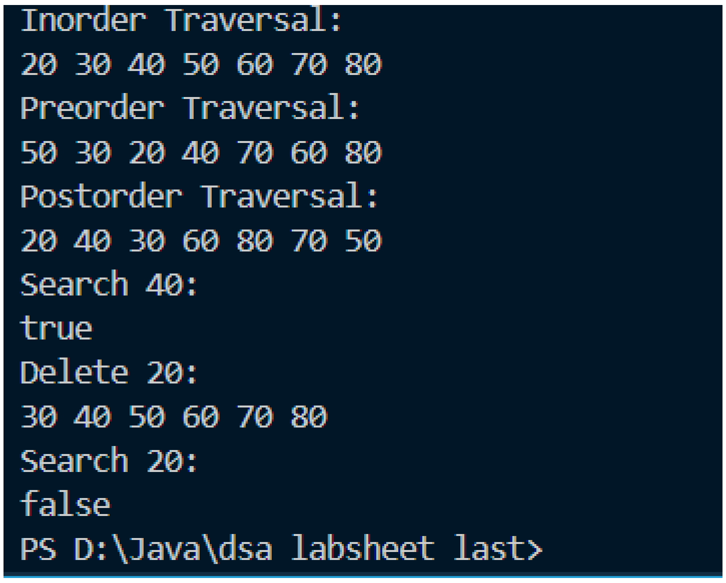
Java code:

```
public class BinarySearchTree<T extends Comparable<T>> {

   private Node<T> root;


   private static class Node<T> {

      T data;

      Node<T> left, right;


      Node(T data) {

         this.data = data;

         this.left = this.right = null;}}

   public void insert(T data) {

      root = insertRec(root, data); }
```

```
private Node<T> insertRec(Node<T> root, T data) {
    if (root == null) {
        root = new Node<>(data);
        return root;}
    if (data.compareTo(root.data) < 0) {
        root.left = insertRec(root.left, data);
    } else if (data.compareTo(root.data) > 0) {
        root.right = insertRec(root.right, data);}
    return root;
}
public boolean search(T data) {
    return searchRec(root, data) != null;
}
private Node<T> searchRec(Node<T> root, T data) {
    if (root == null || data.equals(root.data)) {
        return root;
    }
    if (data.compareTo(root.data) < 0) {
        return searchRec(root.left, data);
    } else {
        return searchRec(root.right, data);
    }
}
public void delete(T data) {
    root = deleteRec(root, data);
}
private Node<T> deleteRec(Node<T> root, T data) {
    if (root == null) {
        return null;}
    if (data.compareTo(root.data) < 0) {
        root.left = deleteRec(root.left, data);
    } else if (data.compareTo(root.data) > 0) {
```

```
root.right = deleteRec(root.right, data);
   } 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 T minValue(Node<T> root) {
   T minValue = root.data;
   while (root.left != null) {
      minValue = root.left.data;
      root = root.left;  }
   return minValue;
}
public void inorderTraversal() {
   inorderRec(root);
   System.out.println();    }
private void inorderRec(Node<T> root) {
   if (root != null) {
      inorderRec(root.left);
      System.out.print(root.data + " ");
      inorderRec(root.right);
   }    }
public void preorderTraversal() {
   preorderRec(root);
   System.out.println();
}    private void preorderRec(Node<T> root) {
    if (root != null) {
        System.out.print(root.data + " ");
        preorderRec(root.left);
        preorderRec(root.right);
    }    }
public void postorderTraversal() {
    postorderRec(root);
    System.out.println();
}
private void postorderRec(Node<T> root) {
    if (root != null) {
        postorderRec(root.left);
        postorderRec(root.right);
        System.out.print(root.data + " ");
    }    }
public static void main(String[] args) {
    BinarySearchTree<Integer> bst = new BinarySearchTree<>();
    bst.insert(50);        bst.insert(30);
    bst.insert(20);        bst.insert(40);
    bst.insert(70);        bst.insert(60);
    bst.insert(80);
    System.out.println("Inorder Traversal:");
    bst.inorderTraversal(); // 20 30 40 50 60 70 80
    System.out.println("Preorder Traversal:");
    bst.preorderTraversal(); // 50 30 20 40 70 60 80
    System.out.println("Postorder Traversal:");
    bst.postorderTraversal(); // 20 40 30 60 80 70 50
    System.out.println("Search 40:");
    System.out.println(bst.search(40)); // true
    System.out.println("Delete 20:");
    bst.delete(20);
    bst.inorderTraversal(); // 30 40 50 60 70 80
```

```
        System.out.println("Search 20:");
        System.out.println(bst.search(20)); // false
    }
}
```



```
Inorder Traversal:
20 30 40 50 60 70 80
Preorder Traversal:
50 30 20 40 70 60 80
Postorder Traversal:
20 40 30 60 80 70 50
Search 40:
true
Delete 20:
30 40 50 60 70 80
Search 20:
false
PS D:\Java\dsa labsheet last>
```

Conclusion:

Binary Search Trees (BSTs) offer a versatile and efficient way to manage and organize data. By maintaining a sorted structure, they facilitate fast search, insertion, and deletion operations, making them suitable for dynamic data sets. While their performance is optimal with balanced trees, it can degrade in skewed scenarios, leading to linear time complexities. Despite this, their inherent properties and ease of implementation make BSTs a fundamental data structure in computer science, widely applicable in various algorithms and systems requiring ordered data management.

Q5. Graph Representation

Theory:

Graph representation refers to the method or technique used to represent a graph data structure in computer memory. In a graph, vertices (nodes) are connected by edges (links), and the representation method determines how these vertices and edges are stored and organized in memory. Different representation techniques, such as adjacency lists, adjacency matrices, and incidence matrices, offer varying trade-offs in terms of memory usage, computational efficiency, and suitability for different types of graphs and algorithms. The choice of representation depends on factors such as the size and density of the graph, memory constraints, and the specific operations to be performed on the graph.

1. Adjacency List: This representation involves each vertex holding a list of adjacent vertices. It's advantageous for sparse graphs due to its memory efficiency. Traversal of the adjacency list enables swift access to neighboring vertices, facilitating various operations such as graph traversal and pathfinding algorithms.

2. Adjacency Matrix: Here, a 2D array signifies edge presence between vertices. Best suited for dense graphs, it offers constant-time edge checks. However, its memory usage increases quadratically with the number of vertices, making it less efficient for large sparse graphs.

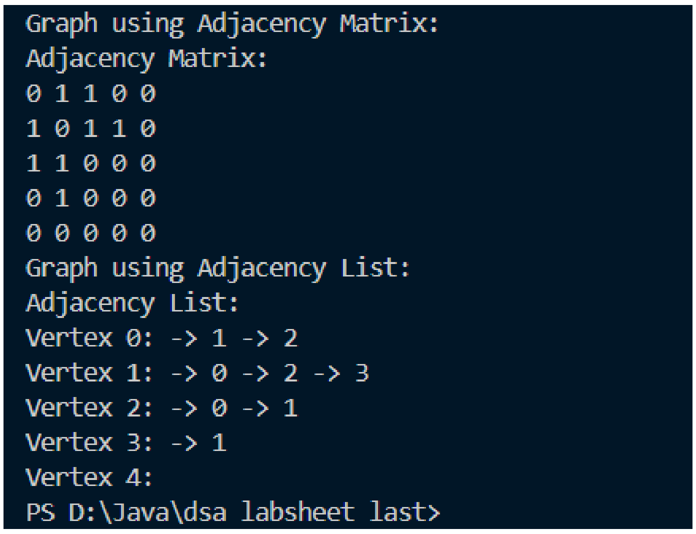
3. Incidence Matrix: Employing a 2D array where rows represent vertices and columns represent edges, this representation indicates vertex-edge connections. It's useful for graphs with attributes but is less common due to its higher memory consumption compared to other representations. Despite this, it offers a structured view of vertex-edge relationships, which can be beneficial in certain analytical contexts.

Java Code:

```
import java.util.LinkedList;
import java.util.List;

public class GraphRepresentation {

  // Adjacency Matrix Representation
  static class GraphAdjMatrix {
    private int[][] adjMatrix;
    private int numVertices;

    public GraphAdjMatrix(int numVertices) {
      this.numVertices = numVertices;
```

```
        adjMatrix = new int[numVertices][numVertices];
    }

    public void addEdge(int i, int j) {
        adjMatrix[i][j] = 1;
        adjMatrix[j][i] = 1; // For undirected graph
    }

    public void removeEdge(int i, int j) {
        adjMatrix[i][j] = 0;
        adjMatrix[j][i] = 0; // For undirected graph
    }

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

// Adjacency List Representation
static class GraphAdjList {
    private List<List<Integer>> adjList;
    private int numVertices;

    public GraphAdjList(int numVertices) {
        this.numVertices = numVertices;
        adjList = new LinkedList<>();
        for (int i = 0; i < numVertices; i++) {
            adjList.add(new LinkedList<>());
        }
    }

    public void addEdge(int src, int dest) {
        adjList.get(src).add(dest);
        adjList.get(dest).add(src); // For undirected graph
    }

    public void removeEdge(int src, int dest) {
        adjList.get(src).remove((Integer) dest);
        adjList.get(dest).remove((Integer) src); // For undirected graph
    }

    public void printGraph() {
        System.out.println("Adjacency List:");
        for (int i = 0; i < adjList.size(); i++) {
            System.out.print("Vertex " + i + ":");
            for (Integer vertex : adjList.get(i)) {
                System.out.print(" -> " + vertex);
            }
            System.out.println();
        }
    }
}

public static void main(String[] args) {
    int numVertices = 5;

    // Adjacency Matrix Example
    GraphAdjMatrix graphMatrix = new GraphAdjMatrix(numVertices);
    graphMatrix.addEdge(0, 1);
    graphMatrix.addEdge(0, 2);
    graphMatrix.addEdge(1, 2);
    graphMatrix.addEdge(1, 3);
    System.out.println("Graph using Adjacency Matrix:");
    graphMatrix.printGraph();
```

```
        // Adjacency List Example
        GraphAdjList graphList = new GraphAdjList(numVertices);
        graphList.addEdge(0, 1);
        graphList.addEdge(0, 2);
        graphList.addEdge(1, 2);
        graphList.addEdge(1, 3);
        System.out.println("Graph using Adjacency List:");
        graphList.printGraph();
    }
}
```



```
Graph using Adjacency Matrix:
Adjacency Matrix:
0 1 1 0 0
1 0 1 1 0
1 1 0 0 0
0 1 0 0 0
0 0 0 0 0
Graph using Adjacency List:
Adjacency List:
Vertex 0: -> 1 -> 2
Vertex 1: -> 0 -> 2 -> 3
Vertex 2: -> 0 -> 1
Vertex 3: -> 1
Vertex 4:
PS D:\Java\dsa labsheet last>
```

Conclusion:

In conclusion, graph representation is a crucial aspect of graph theory and computer science, enabling the efficient storage and manipulation of graphs in computer memory. By selecting the appropriate representation technique, such as adjacency lists, adjacency matrices, or incidence matrices, developers can optimize memory usage and computational efficiency while effectively capturing the relationships and connections within a graph. Understanding the strengths and limitations of each representation method is essential for designing efficient algorithms and solving various graph-related problems in diverse fields, including networking, social media analysis, logistics, and more.

Q.6 SPANNING TREE AND SHORTEST PATH ALGORITHMS

A. Spanning Tree

Theory:

● A spanning tree is a subset of Graph G, which has all the vertices covered with a minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

● By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree.

● A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

1) KRUSKAL'S ALGORITHM

THEORY

Kruskal's Algorithm is used to find the Minimum Spanning Tree **(**MST**)** of a connected, undirected graph. It works by:

1. **Sorting:** Ordering all edges by weight.
2. **Adding Edges:** Iteratively adding the smallest edge to the MST, ensuring no cycles are formed, until all vertices are connected.

Key Points:

- **Greedy Algorithm:** Chooses the smallest available edge at each step.
- **Union-Find:** A data structure is used to detect cycles.

Use Case: Efficient for sparse graphs and network design problems.

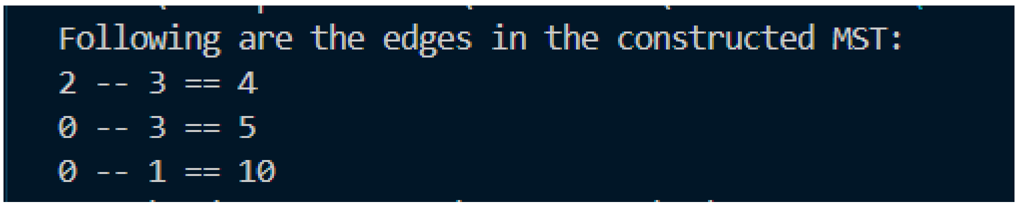
CODE USING JAVA

```java
import java.util.*;

// Class to represent an edge in the graph
class Edge implements Comparable<Edge> {
    int src, dest, weight;

    // Comparator to sort edges based on their weights
    public int compareTo(Edge compareEdge) {
        return this.weight - compareEdge.weight;
    }
};

// Class to represent a subset for union-find
class Subset {
    int parent, rank;
};

public class KruskalsAlgorithm {
    int V, E; // V-> no. of vertices & E->no.of edges
    Edge edge[]; // collection of all edges

    // Creates a graph with V vertices and E edges
    KruskalsAlgorithm(int v, int e) {
        V = v;
        E = e;
        edge = new Edge[E];
        for (int i = 0; i < e; ++i)
            edge[i] = new Edge();
    }
```

```
// A utility function to find set of an element i
// (uses path compression technique)
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are the same, then make one as root and increment its rank by one
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST() {
    Edge result[] = new Edge[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
int i = 0; // An index variable, used for sorted edges

for (i = 0; i < V; ++i)
    result[i] = new Edge();

// Step 1: Sort all the edges in non-decreasing order of their weight.
Arrays.sort(edge);

// Allocate memory for creating V subsets
Subset subsets[] = new Subset[V];
for (i = 0; i < V; ++i)
    subsets[i] = new Subset();

// Create V subsets with single elements
for (int v = 0; v < V; ++v) {
    subsets[v].parent = v;
    subsets[v].rank = 0;
}

i = 0; // Initialize index for the edges[] array

// Number of edges to be taken is equal to V-1
while (e < V - 1) {
    // Step 2: Pick the smallest edge. Increment index for the next iteration
    Edge next_edge = new Edge();
    next_edge = edge[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    // If including this edge doesn't cause a cycle, include it in the result and increment the index
of result for next edge
    if (x != y) {
        result[e++] = next_edge;
```

```
            Union(subsets, x, y);
        }
        // Else discard the next_edge
    }

    // Print the contents of result[] to display the built MST
    System.out.println("Following are the edges in the constructed MST:");
    for (i = 0; i < e; ++i)
        System.out.println(result[i].src + " -- " + result[i].dest + " == " + result[i].weight);
}

public static void main(String[] args) {
    /* Let us create following weighted graph
            10
        0--------1
        | \     |
       6|   5\   |15
        |     \ |
        2--------3
            4       */
    int V = 4; // Number of vertices in the graph
    int E = 5; // Number of edges in the graph
    KruskalsAlgorithm graph = new KruskalsAlgorithm(V, E);

    // add edge 0-1
    graph.edge[0].src = 0;
    graph.edge[0].dest = 1;
    graph.edge[0].weight = 10;

    // add edge 0-2
    graph.edge[1].src = 0;
    graph.edge[1].dest = 2;
    graph.edge[1].weight = 6;
```

```
        // add edge 0-3
        graph.edge[2].src = 0;
        graph.edge[2].dest = 3;
        graph.edge[2].weight = 5;

        // add edge 1-3
        graph.edge[3].src = 1;
        graph.edge[3].dest = 3;
        graph.edge[3].weight = 15;

        // add edge 2-3
        graph.edge[4].src = 2;
        graph.edge[4].dest = 3;
        graph.edge[4].weight = 4;

        // Function call
        graph.KruskalMST();
    }
}
```



```
Following are the edges in the constructed MST:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

CONCLUSION

Kruskal's algorithm, a fundamental approach in graph theory and optimization, efficiently finds the minimum spanning tree (MST) of a connected, weighted graph. Through a systematic process of adding edges in ascending order of weight while avoiding cycles, Kruskal's algorithm constructs an MST that spans all vertices with the least total edge weight. Its simplicity, effectiveness, and time complexity of O(E log E) or O(E log V) make it a preferred choice for many applications, including network design, circuit layout, and clustering analysis. With its ability to handle sparse graphs and its straightforward implementation, Kruskal's algorithm stands as a cornerstone in the arsenal of graph algorithms, facilitating optimal solutions in various domains.

B) Shortest-Path Problems:

Types, Single-Source Shortest path problem-Dijkestra’s Algorithm.

Shortest Path Problem

In data structures,

● Shortest path problem is a problem of finding the shortest path(s) between vertices of a given graph.

● Shortest path between two vertices is a path that has the least cost as compared to all other existing paths.

Shortest Path Algorithms

Shortest path algorithms are a family of algorithms used for solving the shortest path problem.

Applications

Shortest path algorithms have a wide range of applications such as in-

● Google Maps

● Road Networks

● Logistics Research

1. Dijkstra Algorithm

THEORY

● The Dijkstra Algorithm is a very famous greedy algorithm.

● It is used for solving the single source shortest path problem.

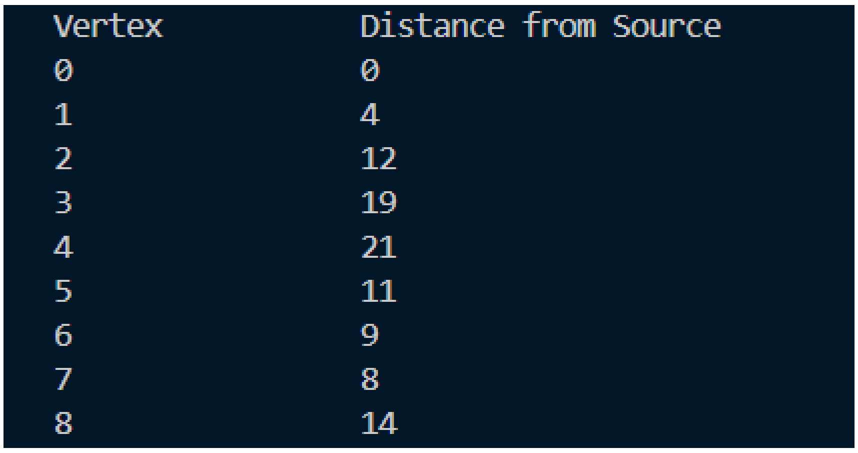
● It computes the shortest path from one particular source node to all other remaining nodes of graph.

CODE USING JAVA

```
import java.util.*;

class PrimAlgorithm {
```

```
// Number of vertices in the graph
private static final int V = 5;

// Function to find the vertex with minimum key value,
// from the set of vertices not yet included in MST
private int minKey(int[] key, boolean[] mstSet) {
    int min = Integer.MAX_VALUE, minIndex = -1;

    for (int v = 0; v < V; v++) {
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }

    return minIndex;
}

// Function to print the constructed MST stored in parent[]
private void printMST(int[] parent, int[][] graph) {
    System.out.println("Edge \tWeight");
    for (int i = 1; i < V; i++) {
        System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);
    }
}

// Function to construct and print MST for a graph represented
// using adjacency matrix representation        // Key values used to pick minimum weight edge in cut
    int[] key = new int[V];

    // To represent set of vertices not yet in
void primMST(int[][] graph) {
    // Array to store constructed MST
    int[] parent = new int[V];
```

```
cluded in MST
        boolean[] mstSet = new boolean[V];

        // Initialize all keys as INFINITE
        for (int i = 0; i < V; i++) {
            key[i] = Integer.MAX_VALUE;
            mstSet[i] = false;
        }

        // Always include first vertex in MST.
        // Make key 0 so that this vertex is picked as first vertex
        key[0] = 0;
        parent[0] = -1; // First node is always root of MST

        // The MST will have V vertices
        for (int count = 0; count < V - 1; count++) {
            // Pick the minimum key vertex from the set of vertices
            // not yet included in MST
            int u = minKey(key, mstSet);

            // Add the picked vertex to the MST Set
            mstSet[u] = true;

            // Update key value and parent index of the adjacent vertices
            // of the picked vertex. Consider only those vertices which are
            // not yet included in MST
            for (int v = 0; v < V; v++) {
                // graph[u][v] is non zero only for adjacent vertices of m
                // mstSet[v] is false for vertices not yet included in MST
                // Update the key only if graph[u][v] is smaller than key[v]
                if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {
                    parent[v] = u;
                    key[v] = graph[u][v];
                }
            }
        }

        // print the constructed MST
        printMST(parent, graph);
    }

    public static void main(String[] args) {
        /* Let us create the following graph
           2    3
        (0)--(1)--(2)
        |   / \   |
       6| 8/   \5 |7
        | /     \ |
        (3)-------(4)
             9          */
        PrimAlgorithm t = new PrimAlgorithm();
        int[][] graph = new int[][]{{0, 2, 0, 6, 0},
                                    {2, 0, 3, 8, 5},
                                    {0, 3, 0, 0, 7},
                                    {6, 8, 0, 0, 9},
                                    {0, 5, 7, 9, 0}};

        // Print the solution
        t.primMST(graph);
    }
}
```



```
Vertex          Distance from Source
0               0
1               4
2               12
3               19
4               21
5               11
6               9
7               8
8               14
```

CONCLUSION

In conclusion, Dijkstra's algorithm stands as a cornerstone in graph theory and algorithm design, offering an elegant solution for finding shortest paths in weighted graphs. Its simplicity of implementation and efficiency make it indispensable for a wide array of applications, from network routing to GPS navigation. While its limitations, such as the assumption of non-negative edge weights, must be acknowledged, its enduring impact on the field of algorithms and its practical utility remain undeniable. As such, Dijkstra's algorithm continues to serve as a fundamental tool for computer scientists and programmers alike, solving shortest path problems with precision and effectiveness.

Q.8 Sorting, Searching and Hashing Algorithm

1. Sorting

Theory:

The arrangement of data in a preferred order is called sorting in the data Structure . By sorting data, it is easier to search through it quickly and easily. The simplest example of sorting is a dictionary. There are different types of sorting. i.e.

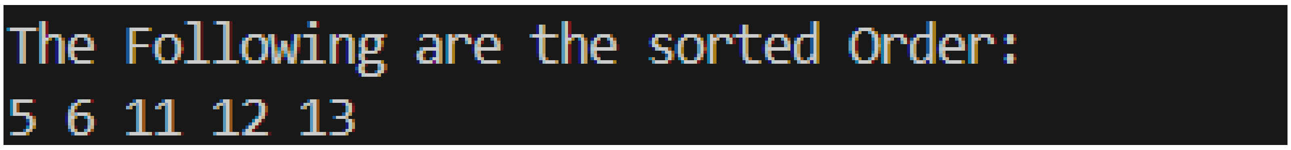
1. Bubble Sort: It's a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they're in the wrong order.
2. Insertion Sort: In Insertion Sorting, the dataset is virtually split into a sorted and an unsorted part, then the algorithm picks up the elements from the unsorted part and places them at the correct position in the sorted part.
3. Selection Sort: In Selection Sorting, we move along the data and select the smallest item, swap the selected item to the position 0,and so on.
4. Quick Sort: This sorting algorithm picks up a pivot element, then partitions the dataset into two sub-arrays, one sub-array is greater than the element and another sub-array is less than the element.
5. Merge Sort: It divides the dataset into smaller sub-arrays , sorts them individually, and then merges them to obtain the final sorted result.
6. Heap Sort: Heap Sorting involves building a Heap data structure from the given array and then utilizing the Heap to sort the array.
7. Radix Sort: It sorts the numbers by sorting each digit from left to right, resulting in the sorted data. Its time complexity is O(d*(n+b)).

1. Insertion sort

Java code:

```
public class InsertionSort {
  static void insertionSort(int arr[], int n) {
  int i, key, j;
  for (i = 1; i < n; i++) {
  key = arr[i];
  j = i - 1;
  while (j >= 0 && arr[j] > key) {
  arr[j + 1] = arr[j];
  j = j - 1;
 }
  arr[j + 1] = key;
  }   }    static void printArray(int arr[], int n) {
  for (int i = 0; i < n; i++)
  System.out.print(arr[i] + " ");
  System.out.println();    }
  public static void main(String args[]) {
  int arr[] = { 12, 11, 13, 5, 6 };
  int N = arr.length;
  insertionSort(arr, N);
  printArray(arr, N);
  }    }
```

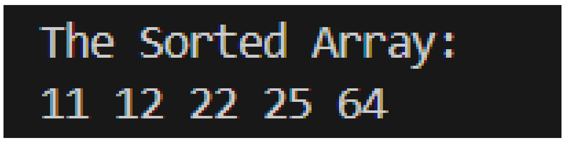
Output:



```
The Following are the sorted Order:
5 6 11 12 13
```

2. Selection sort

```
import java.util.Arrays;
public class SelectionSort {
// Function for Selection sort
static void selectionSort(int arr[], int n) {
int i, j, min_idx;
// One by one move boundary of
// unsorted subarray
for (i = 0; i < n - 1; i++) {
// Find the minimum element in
// unsorted array
min_idx = i;
for (j = i + 1; j < n; j++) {
if (arr[j] < arr[min_idx])
min_idx = j;
}
// Swap the found minimum element
// with the first element
if (min_idx != i) {
int temp = arr[min_idx];
arr[min_idx] = arr[i];
arr[i] = temp;}}}
// Function to print an array
static void printArray(int arr[], int size) {
for (int i = 0; i < size; i++) {
System.out.print(arr[i] + " ");
}
System.out.println();
}
// Driver program
public static void main(String[] args) {
int arr[] = { 64, 25, 12, 22, 11 };
int n = arr.length;
// Function Call
selectionSort(arr, n);
System.out.println("The Sorted Array: ");
printArray(arr, n);
}}
```

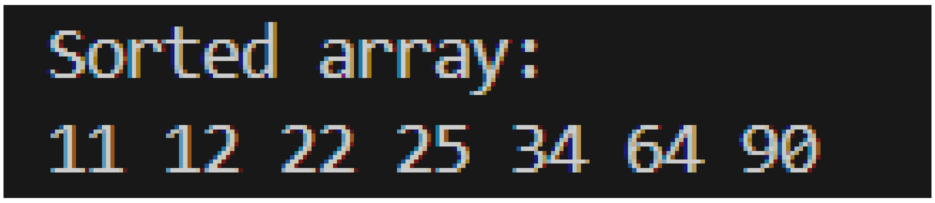


```
The Sorted Array:
11 12 22 25 64
```

3. Bubble sort

```
import java.util.Arrays;
public class BubbleSort {
// An optimized version of Bubble Sort
static void bubbleSort(int arr[], int n) {
int i, j;
boolean swapped;
for (i = 0; i < n - 1; i++) {
swapped = false;
for (j = 0; j < n - i - 1; j++) {
if (arr[j] > arr[j + 1]) {
// Swap elements if they are in the wrong order
int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
swapped = true;
}
}
// If no two elements were swapped by inner loop, then break
if (!swapped)
break;
}
}
// Function to print an array
static void printArray(int arr[], int size) {
for (int i = 0; i < size; i++)
System.out.print(arr[i] + " ");
}
// Driver program to test above functions
public static void main(String args[]) {
int arr[] = { 64, 34, 25, 12, 22, 11, 90 };
int N = arr.length;
bubbleSort(arr, N);
System.out.println("Sorted array:");
printArray(arr, N);
}}
```

Output:



```
Sorted array:
11 12 22 25 34 64 90
```

B. SEARCHING ALGORITHM

Theory :

Searching algorithms are vital tools in computer science for locating specific items within data collections. Three main types of searching algorithms exist:

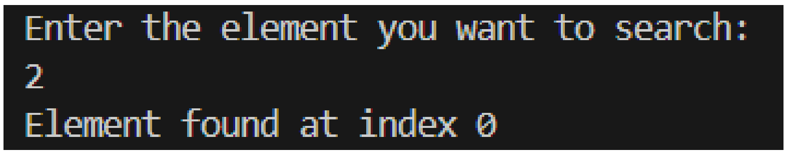
- Binary Search: Utilized in sorted arrays, binary search repeatedly divides the search interval in half, leveraging the sorted nature of the array to achieve a time complexity of O(log N).

- Interpolation Search: An enhancement over binary search, interpolation search is effective when sorted array values are uniformly distributed. It constructs new data points within the range of known data points, potentially leading to different search locations based on the value being sought.

1. Binary searching

```
import java.util.Scanner;
public class BinarySearch {

    // Function that returns index of
    // x if it is present in arr[l, r]
    int binarySearch(int arr[], int l,
                     int r, int x)
    {
        if (r >= l) {
            int mid = l + (r - l) / 2;

            // If the element is present
            // at the middle itself
            if (arr[mid] == x)
                return mid;

            // If element is smaller than
            // mid, then it can only be
            // present in left subarray
            if (arr[mid] > x)
                return binarySearch(arr, l,
                                    mid - 1, x);

            // Else the element can only be
            // present in right subarray
            return binarySearch(arr, mid + 1,
                                r, x);
        }

        // Reach here when element is
        // not present in array
        return -1;
    }

    // Driver Code
    public static void main(String args[])
    {

        // Create object of this class
        BinarySearch ob = new BinarySearch();
        Scanner sc=new Scanner(System.in);
        // Given array arr[]
        int arr[] = { 2, 3, 4, 10, 40 };
        int n = arr.length;
        System.out.println("Enter the element you want to search:");
        int x=sc.nextInt();

        // Function Call
        int result = ob.binarySearch(arr, 0,
                                     n - 1, x);

        if (result == -1)
            System.out.println("Element "
                               + "not present");
        else
            System.out.println("Element found"
                               + " at index "
                               + result);
    }
}
```

Output:

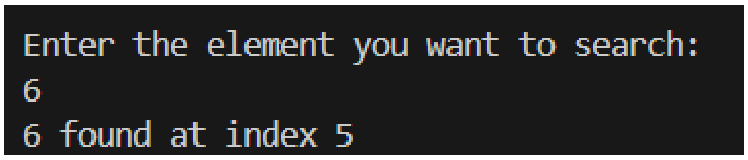


```
Enter the element you want to search:
2
Element found at index 0
```

2. Interpolation searching

```
import java.util.Scanner;
public class InterpolationSearch {
    public static int interpolationSearch(int[] arr, int target) {
        int low = 0;
        int high = arr.length - 1;

        while (low <= high && target >= arr[low] && target <= arr[high]) {
            // Calculate the position of the target element based on its value
            int pos = low + (((target - arr[low]) * (high - low)) / (arr[high] - arr[low]));
            // Check if the target element is at the calculated position
            if (arr[pos] == target) {
                return pos;
            }

            // If the target element is less than the element at the
            // calculated position, search the left half of the list
            if (arr[pos] > target) {
                high = pos - 1;
            } else {
                // If the target element is greater than the element
                // at the calculated position, search the right half of the list
                low = pos + 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
         Scanner sc=new Scanner(System.in);
         System.out.println("Enter the element you want to search:");
        int target=sc.nextInt();
        int index = interpolationSearch(arr, target);

        if (index == -1) {
            System.out.println(target + " not found in the list");
        } else {
            System.out.println(target + " found at index " + index);
        }
    }
}
```

Output:



```
Enter the element you want to search:
6
6 found at index 5
```

Conclusion:

In conclusion, linear search is a basic search algorithm that can be useful in certain situations, such as when the list of elements is small, or when the list is unsorted. Binary search is a more efficient search algorithm for large, sorted lists.

C.Hashing Algorithm

Theory:

A hashing algorithm converts input into a fixed-size output, serving as an index in an array or hash table. The hash function ensures determinism, always producing the same result for a given input. Different hashing techniques include:

- Linear Probing: Used to resolve collisions in hash tables by linearly searching for the next available location if the hashed memory location is already filled. The table is considered circular for seamless traversal.

- Quadratic Probing: Similar to linear probing but utilizes a quadratic sequence (h, h+1, h+4, h+9...) to search for available locations, aiming to distribute values more evenly.

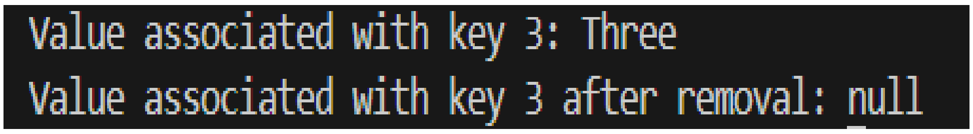
- Separate Chaining: Handles collisions by allowing multiple key-value pairs to be stored at the same index. Each index contains a linked list or another data structure holding the colliding pairs.

- Double Hashing: Resolves collisions by employing two hash functions to compute two different hash values for a key. The first function calculates the initial hash value, while the second determines the step size for the probing sequence, resulting in a lower collision rate compared to other methods.

1. Linear Probing

```
public class LinearProbingHashTable {
    private static final int DEFAULT_CAPACITY = 10;
    private static final double LOAD_FACTOR_THRESHOLD = 0.7;

    private Entry[] table;
    private int size;

    public LinearProbingHashTable() {
        this(DEFAULT_CAPACITY);
    }

    public LinearProbingHashTable(int initialCapacity) {
        table = new Entry[initialCapacity];
    size = 0;
}

private static class Entry {
    Object key;
    Object value;
    boolean deleted;

    Entry(Object key, Object value) {
        this.key = key;
        this.value = value;
        this.deleted = false;
    }
}

public void put(Object key, Object value) {
    if ((double) size / table.length > LOAD_FACTOR_THRESHOLD) {
        resize();
    }

    int index = hash(key);
    while (table[index] != null && !table[index].deleted) {
        if (table[index].key.equals(key)) {
            table[index].value = value;
            return;
        }
        index = (index + 1) % table.length; // Linear probing
    }
    table[index] = new Entry(key, value);
    size++;
}

public Object get(Object key) {
    int index = find(key);
    return index != -1 ? table[index].value : null;
}

public void remove(Object key) {
    int index = find(key);
    if (index != -1) {
        table[index].deleted = true;
        size--;
    }
}

private int find(Object key) {
    int index = hash(key);
    while (table[index] != null) {
        if (!table[index].deleted && table[index].key.equals(key)) {
            return index;
        }
        index = (index + 1) % table.length; // Linear probing
    }
    return -1;
}

private int hash(Object key) {
    return Math.abs(key.hashCode()) % table.length;
}

private void resize() {
    Entry[] oldTable = table;
    table = new Entry[table.length * 2];
    size = 0;

    for (Entry entry : oldTable) {
        if (entry != null && !entry.deleted) {
            put(entry.key, entry.value);
            }
        }
    }

    public static void main(String[] args) {
        LinearProbingHashTable hashTable = new LinearProbingHashTable();
        hashTable.put(1, "One");
        hashTable.put(2, "Two");
        hashTable.put(3, "Three");
        hashTable.put(4, "Four");
        hashTable.put(5, "Five");
        hashTable.put(6, "Six");
        hashTable.put(7, "Seven");

        System.out.println("Value associated with key 3: " + hashTable.get(3));
        hashTable.remove(3);
        System.out.println("Value associated with key 3 after removal: " + hashTable.get(3));
    }
}
```

Output:

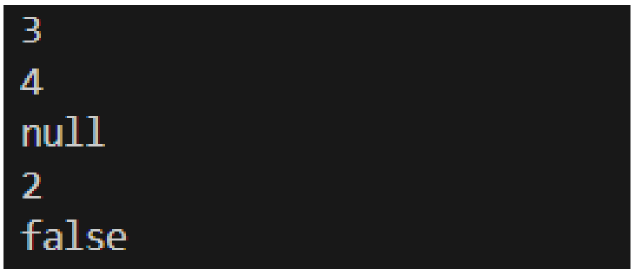


```
Value associated with key 3: Three
Value associated with key 3 after removal: null
```

2. Separate Chaining

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

// A node of chains
class HashNode<K, V> {
        K key;
        V value;
        final int hashCode;

        // Reference to next node
        HashNode<K, V> next;

        // Constructor
        public HashNode(K key, V value, int hashCode)
        {
                this.key = key;
                this.value = value;
                this.hashCode = hashCode;
        }
}

// Class to represent entire hash table
class Map<K, V> {
        // bucketArray is used to store array of chains
        private ArrayList<HashNode<K, V> > bucketArray;

        // Current capacity of array list
        private int numBuckets;

        // Current size of array list
        private int size;

        // Constructor (Initializes capacity, size and
        // empty chains.
        public Map()
        {
                bucketArray = new ArrayList<>();
                numBuckets = 10;
                size = 0;

                // Create empty chains
                for (int i = 0; i < numBuckets; i++)
                        bucketArray.add(null);
        }

        public int size() { return size; }
        public boolean isEmpty() { return size() == 0; }

        private final int hashCode (K key) {
                return Objects.hashCode(key);
        }

        // This implements hash function to find index
        // for a key
        private int getBucketIndex(K key)
        {
                int hashCode = hashCode(key);
                int index = hashCode % numBuckets;
                // key.hashCode() could be negative.
                index = index < 0 ? index * -1 : index;
                return index;
        }

        // Method to remove a given key
        public V remove(K key)
        {
                // Apply hash function to find index for given key
                int bucketIndex = getBucketIndex(key);
                int hashCode = hashCode(key);
                // Get head of chain
                HashNode<K, V> head = bucketArray.get(bucketIndex);

                // Search for key in its chain
                HashNode<K, V> prev = null;
                while (head != null) {
                    // If Key found
                    if (head.key.equals(key) && hashCode == head.hashCode)
                        break;

                    // Else keep moving in chain
                    prev = head;
                    head = head.next;
                }

                // If key was not there
                if (head == null)
                    return null;

                // Reduce size
                size--;

                // Remove key
                if (prev != null)
                    prev.next = head.next;
                else
                    bucketArray.set(bucketIndex, head.next);

                return head.value;
            }

            // Returns value for a key
            public V get(K key)
            {
                // Find head of chain for given key
                int bucketIndex = getBucketIndex(key);
                int hashCode = hashCode(key);

                HashNode<K, V> head = bucketArray.get(bucketIndex);
                // Search key in chain
                while (head != null) {
                        if (head.key.equals(key) && head.hashCode == hashCode)
                                return head.value;
                        head = head.next;
                }

                // If key not found
                return null;
        }

        // Adds a key value pair to hash
        public void add(K key, V value)
        {
                // Find head of chain for given key
                int bucketIndex = getBucketIndex(key);
                int hashCode = hashCode(key);
                HashNode<K, V> head = bucketArray.get(bucketIndex);

                // Check if key is already present
                while (head != null) {
                        if (head.key.equals(key) && head.hashCode == hashCode) {
                                head.value = value;
                                return;
                        }
                        head = head.next;
                }

                // Insert key in chain
                size++;
                head = bucketArray.get(bucketIndex);
                HashNode<K, V> newNode
                        = new HashNode<K, V>(key, value, hashCode);
                newNode.next = head;
                bucketArray.set(bucketIndex, newNode);

                // If load factor goes beyond threshold, then
                // double hash table size
                if ((1.0 * size) / numBuckets >= 0.7) {
                    ArrayList<HashNode<K, V> > temp = bucketArray;
                    bucketArray = new ArrayList<>();
                    numBuckets = 2 * numBuckets;
                    size = 0;
                    for (int i = 0; i < numBuckets; i++)
                        bucketArray.add(null);

                    for (HashNode<K, V> headNode : temp) {
                        while (headNode != null) {
                            add(headNode.key, headNode.value);
                            headNode = headNode.next;
                        }
                    }
                }
            }

            // Driver method to test Map class
            public static void main(String[] args)
            {
                Map<String, Integer> map = new Map<>();
                map.add("this", 1);
                map.add("coder", 2);
                map.add("this", 4);
                map.add("hi", 5);
                System.out.println(map.size());
                System.out.println(map.remove("this"));
                System.out.println(map.remove("this"));
                System.out.println(map.size());
                System.out.println(map.isEmpty());
                }
        }
```

Output:



```
3
4
null
2
false
```

Conclusion:

One common technique is open addressing, where collisions are resolved by probing the table for an empty slot. Linear probing sequentially searches for the next available slot, while quadratic probing uses a quadratic function to determine the next slot to check, reducing clustering. However, both methods can suffer from clustering issues. Another approach, double hashing, uses two hash functions to calculate the step size, which helps mitigate clustering and provides better distribution of values across the table. Separate chaining is another popular technique, where each slot in the hash table contains a linked list of key-value pairs that hash to the same index. This method is simple and efficient but may incur additional memory overhead. Each hashing technique has its advantages and disadvantages, and the choice of which to use depends on factors such as the data distribution, performance requirements, and memory constraints. Overall, understanding the characteristics of different hashing techniques is essential for designing efficient and reliable data structures and algorithms.