Data Structure

Q1. Create a class `Node` that contains a member 'info' to store a number, with 'left' referring to the left child and 'right' referring to the right child. Provide the necessary constructor. Additionally, create a method to insert a node into a binary search tree.
Q2. Write a program to add methods to the binary search tree created in Q1 for traversing the tree in pre-order, in-order, and post-order. Invoke the above methods for execution.

```
class Node {
      private int info;
      private Node left;
      private Node right;
      public Node(int v) {
            info=v;
            left=null;
            right=null;
      }
      public int getInfo() {
            return info;
      }
      public Node getleft() {
            return left;
      }
      public Node getright() {
            return right;
      }
      public void setInfo(int info) {
            this.info = info;
      }
      public void setleft(Node Child) {
            this.left = Child;
      }
      public void setright(Node Child) {
            this.right = Child;
      }
}

public class InsertBST {
      private Node root;

      public void InsertNode(int info) {
            root = InsertNode(root, info);
      }
      public Node InsertNode(Node node, int info) {
            if (node == null) {
                  node = new Node(info);
            }
            else {
                  if (node.getInfo() > info) {
                        node.setleft(InsertNode(node.getleft(),info));
                        }
                  else {
                        node.setright(InsertNode(node.getright(),info));
                        }
```

```
                }
            return node;
        }

        public void PrintPreOrder(Node node)/* pre order */
        {
            if (node != null) {
                System.out.print(" " + node.getInfo());
                PrintPreOrder(node.getleft());
                PrintPreOrder(node.getright());
            }
        }
        public void PrintInOrder(Node node)/*In order */
        {
            if (node != null) {
                PrintInOrder(node.getleft());
                System.out.print(" " + node.getInfo());
                PrintInOrder(node.getright());
            }
        }
        public void PrintPostOrder(Node node)/*Post order */
        {
            if (node != null) {
                PrintPostOrder(node.getleft());
                PrintPostOrder(node.getright());
                System.out.print(" " + node.getInfo());
            }
        }



        public static void main(String[] args) {
            InsertBST t = new InsertBST();
            System.out.println("Binary Search Tree:");
            t.InsertNode(40);
            t.InsertNode(20);
            t.InsertNode(30);
            t.InsertNode(60);
            t.InsertNode(70);
            t.InsertNode(10);
            t.InsertNode(50);
            System.out.print("\n"+"PreOrder:");
            t.PrintPreOrder(t.root);
            System.out.print("\n"+"InOrder:");
            t.PrintInOrder(t.root);
            System.out.print("\n"+"PostOrder:");
            t.PrintPostOrder(t.root);

        }

}
```

Q3. Create a class Country containing members for name and population, along with a constructor and necessary methods. Additionally, create a class BNode with a member 'info' to store a country object, 'left' to refer to the left child, and 'right' to refer to the right child. Provide the required constructor. Finally, create another class BST with a member 'root', along with the necessary constructor and a method to insert a node into the binary search tree.

Q4. Extend the BST created in Q3 by adding methods to traverse the tree in level order, find the node with the maximum population (find-max), and find the node with the minimum population (find-min). Invoke these methods for execution.

```java
import java.util.*;
class Country {
    private String name;
    private int population;

    public Country(String name, int population) {
        this.name = name;
        this.population = population;
    }

    public String getName() {
        return name;
    }

    public int getPopulation() {
        return population;
    }
}

class BNode {
    private Country info;
    private BNode left;
    private BNode right;

    public BNode(Country info) {
        this.info = info;
        left = null;
        right = null;
    }

    public Country getInfo() {
        return info;
    }

    public BNode getLeft() {
        return left;
    }

    public BNode getRight() {
        return right;
    }

    public void setLeft(BNode left) {
        this.left = left;
    }

    public void setRight(BNode right) {
        this.right = right;
    }
}

public class BSTCountry {
    private BNode root;

    public BSTCountry() {
        root = null;
    }

    public void insertNode(Country country) {
        root = insertNode(root, country);
    }

    private BNode insertNode(BNode node, Country country) {
        if (node == null) {
            node = new BNode(country);
        } else {
            if (country.getPopulation() < node.getInfo().getPopulation()) {
                node.setLeft(insertNode(node.getLeft(), country));
            } else {
                node.setRight(insertNode(node.getRight(), country));
            }
        }
        return node;
    }

    public void levelOrderTraversal() {
        if (root == null)
            return;

        Queue<BNode> queue = new LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
            BNode tempNode = queue.poll();
            System.out.println("Country: " + tempNode.getInfo().getName() + ", Pop-
ulation: " + tempNode.getInfo().getPopulation());

            if (tempNode.getLeft() != null)
                queue.add(tempNode.getLeft());

            if (tempNode.getRight() != null)
                queue.add(tempNode.getRight());
        }
    }

    public BNode findMaxPopulationNode() {
        BNode current = root;
        if (current == null)
            return null;

        while (current.getRight() != null)
            current = current.getRight();

        return current;
    }

    public BNode findMinPopulationNode() {
        BNode current = root;
        if (current == null)
            return null;

        while (current.getLeft() != null)
            current = current.getLeft();
        return current;
    }

    public static void main(String[] args) {
        BSTCountry bst = new BSTCountry();

        // Inserting nodes
        bst.insertNode(new Country("India", 140));
        bst.insertNode(new Country("China", 150));
        bst.insertNode(new Country("USA", 90));
        bst.insertNode(new Country("UK", 95));
        bst.insertNode(new Country("Pakistan", 50));
        bst.insertNode(new Country("Australia", 30));

        // Traversing in level order
        System.out.println("Level Order Traversal:");
        bst.levelOrderTraversal();

        // Finding node with maximum population
        BNode maxNode = bst.findMaxPopulationNode();
        System.out.println("Node with Maximum Population: " +
maxNode.getInfo().getName() + ", Population: " +
maxNode.getInfo().getPopulation());

        // Finding node with minimum population
        BNode minNode = bst.findMinPopulationNode();
        System.out.println("Node with Minimum Population: " + min-
Node.getInfo().getName() + ", Population: " + minNode.getInfo().getPopulation());
    }
}
```

Q5. Construct a binary search tree (BST) from the given array of elements: {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}. Include a method called 'CreateBinaryTree' to construct the binary search tree from a sorted array. This method takes an array of integers as input and constructs the tree recursively using a binary search algorithm.

```
class Node {
    protected int value;
    protected Node lChild;
    protected Node rChild;
    public Node(int v)  {
        value = v;
        lChild = null;
        rChild = null;
    }
    public int getValue() {
        return value;
    }
    public Node getlChild()  {
        return lChild;
    }
    public Node getrChild()  {
        return rChild;
    }
}

public class CreateBST {
    private static Node root;

    public static void PrintPreOrder(Node node) {
        if (node != null) {
            System.out.print(" " + node.getValue());
            PrintPreOrder(node.getlChild());
            PrintPreOrder(node.getrChild());
        }
    }
    public static void PrintInOrder(Node node) {
        if (node != null) {
            PrintInOrder(node.getlChild());
            System.out.print(" " + node.getValue());
            PrintInOrder(node.getrChild());
        }
    }
    public static void PrintPostOrder(Node node) {
        if (node != null) {
            PrintPostOrder(node.getlChild());
            PrintPostOrder(node.getrChild());
            System.out.print(" " + node.getValue());
        }
    }

    public void CreateBinaryTree(int[] arr) {
        root = CreateBinaryTree(arr, 0, arr.length - 1);
    }

    public Node CreateBinaryTree(int[] arr, int start, int end) {
        Node curr = null;
        if (start > end)
            return null;
        int mid = (start + end) / 2;
        curr = new Node(arr[mid]);
        curr.lChild = CreateBinaryTree(arr, start, mid - 1);
        curr.rChild = CreateBinaryTree(arr, mid + 1, end);
        return curr;
    }

    public static void main(String[] args) {
        CreateBST t = new CreateBST();
        int[] arr = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };
        System.out.println("Create BinaryTree:");
        t.CreateBinaryTree(arr);
        System.out.print("\nPreOrder:");
        PrintPreOrder(root);
        System.out.print("\nInOrder:");
        PrintInOrder(root);
        System.out.print("\nPostOrder:");
        PrintPostOrder(root);
    }
}
```

Q6. Determine if a given binary tree is a binary search tree. You will use an 'isBST()' method, which takes the maximum and minimum range of the values of the nodes.

```
class Node {
        private int value;
        private Node lChild;
        private Node rChild;
        public Node(int v) {
              value=v;
              lChild=null;
              rChild=null;
        }
        public int getValue() {
              return value;
        }
        public Node getlChild() {
              return lChild;
        }
        public Node getrChild() {
              return rChild;
        }
        public void setValue(int value) {
              this.value = value;
        }
        public void setlChild(Node Child) {
              this.lChild = Child;
        }
        public void setrChild(Node Child) {
              this.rChild = Child;
        }
  }

  public class BSTCheck {
        private Node root;

        public static void PrintPreOrder(Node node)
        {
              if (node != null) {
                    System.out.print(" " + node.getValue());
                    PrintPreOrder(node.getlChild());
                    PrintPreOrder(node.getrChild());
              }
        }

        public boolean isBST() {
              return isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
        }
        public boolean isBST(Node curr, int min, int max) {
              if (curr == null)
                    return true;
              if (curr.getValue() < min || curr.getValue() > max)
                    return false;
              return isBST(curr.getlChild(), min, curr.getValue()) &&
  isBST(curr.getrChild(), curr.getValue(), max);
        }

        public static void main(String[] args) {
              BSTCheck t = new BSTCheck();
              System.out.println("Tree: ");
              t.root = new Node(1);
            t.root.setlChild(new Node(2));
            t.root.setrChild(new Node(3));
            t.root.getlChild().setlChild(new Node(4));
            t.root.getlChild().setrChild(new Node(5));
            t.root.getrChild().setlChild(new Node(6));
            t.root.getrChild().setrChild(new Node(7));
            System.out.print("PreOrder:");
            PrintPreOrder(t.root);
              System.out.println("\n"+"isBST: "+t.isBST());
        }
  }
```

Q7. Remove node x from the binary search tree and reorganize the nodes to maintain its necessary properties.
There are three cases in the deletion process. Let us denote the node that needs to be deleted as x:
Case 1: Node x has no children.
Case 2: Node x has one child.
Case 3: Node x has two children.

```
class BNode {
      private int value;
      private BNode lChild;
      private BNode rChild;
      public BNode(int v) {
            value=v;
            lChild=null;
            rChild=null;
      }
      public int getValue() {
            return value;
      }
      public BNode getlChild() {
            return lChild;
      }
      public BNode getrChild() {
            return rChild;
      }
      public void setValue(int value) {
            this.value = value;
      }
      public void setlChild(BNode Child) {
            this.lChild = Child;
      }
      public void setrChild(BNode Child) {
            this.rChild = Child;
      }
}
public class DeleteBST {
      private BNode root;

      public void InsertNode(int value) {
            root = InsertNode(root, value);
      }
      public BNode InsertNode(BNode node, int value) {
            if (node == null) {
                  node = new BNode(value);
            }
            else {
                  if (node.getValue() > value) {
                        node.setlChild(InsertNode(node.getlChild(),value));
                        }
                else {
                        node.setrChild(InsertNode(node.getrChild(),value));
                        }
                }
            return node;
    }

    public void PrintInOrder(BNode node)/*In order */
    {
            if (node != null) {
                    PrintInOrder(node.getlChild());
                    System.out.print(" " + node.getValue());
                    PrintInOrder(node.getrChild());
            }
    }

    public static BNode FindMinNode(BNode curr) {
            BNode node = curr;
            if (node == null) {
                    return null;
            }
            while (node.getlChild() != null) {
                    node = node.getlChild();
            }
            return node;
    }
    public void DeleteNode(int value) {
            root = DeleteNode(root, value);
            System.out.print("\n"+"After Delete InOrder:");
            PrintInOrder(root);
    }
    public BNode DeleteNode(BNode node, int value) {
            BNode temp = null;
            if (node != null) {
                    if (node.getValue() == value) {
                            if (node.getlChild() == null && node.getrChild() == null) {
                                    return null;
                            } else {
                                    if (node.getlChild() == null) {
                                            temp = node.getrChild();
                                            return temp;
                                    }
                                    if (node.getrChild() == null) {
                                            temp = node.getlChild();
                                            return temp;
                                    }
                                    BNode minNode = FindMinNode(node.getrChild());
                                    int minValue = minNode.getValue();
                                    node.setValue(minValue);
                                    node.setrChild(DeleteNode(node.getrChild(), minVal-
ue));

                                    }
                            }
                    else {
                            if (value < node.getValue() ) {
                                    node.setlChild(DeleteNode(node.getlChild(), value));
                            } else {
                                    node.setrChild(DeleteNode(node.getrChild(), value));
                        }
                    }
                }
            return node;
            }

        public static void main(String[] args) {
            DeleteBST t = new DeleteBST();
            int[] arr = { 40, 20, 30, 60, 70, 10, 50, 5 };
            for(int val:arr)
                t.InsertNode(val);
            System.out.println("Binary Search Tree:");
            System.out.print("\n"+"InOrder:");
            t.PrintInOrder(t.root);
            System.out.print("\n"+"Delete node(70) has no child");
            t.DeleteNode(70);
            System.out.print("\n"+"Delete node(10) has one child");
            t.DeleteNode(10);
            System.out.print("\n"+"Delete node(20) has two children");
            t.DeleteNode(20);
        }
}
```

Q8. Write a program to implement the graph using adjacency matrix representation and adjacency list representation. Create methods to display the adjacency matrix and adjacency list of the graph.

```
import java.util.Arrays;
import java.util.LinkedList;

// Class for Graph Representation using Adjacency Matrix
class GraphAdjMatrix {
    int[][] adjMatrix;

    // Method to initialize the adjacency matrix
    public void initializeMatrix(int v) {
        adjMatrix = new int[v][v];
        for (int[] row : adjMatrix)
            Arrays.fill(row, 0);
    }

    // Method to add an edge to the adjacency matrix
    public void addEdge(int from, int to) {
        adjMatrix[from][to] = 1;
        adjMatrix[to][from] = 1;
    }

    // Method to display the adjacency matrix
    public void printAdjMatrix() {
        for (int i = 0; i < adjMatrix.length; i++) {
            for (int j = 0; j < adjMatrix[i].length; j++) {
                System.out.print(adjMatrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}

// Class for Graph Representation using Adjacency List
```

```
class GraphAdjList {
    LinkedList<Integer>[] adjList;

    // Method to initialize the adjacency list
    public void initializeList(int v) {
        adjList = new LinkedList[v];
        for (int i = 0; i < v; i++)
            adjList[i] = new LinkedList<>();
    }

    // Method to add an edge to the adjacency list
    public void addEdge(int from, int to) {
        adjList[from].add(to);
        adjList[to].add(from);
    }

    // Method to display the adjacency list
    public void displayAdjList() {
        for (int i = 0; i < adjList.length; i++) {
            System.out.print(i);
            for (int j : adjList[i]) {
                System.out.print(" -> " + j);
            }
            System.out.println();
        }
    }
}

public class GraphRepresentation {
    public static void main(String[] args) {
        // Adjacency Matrix Representation
        GraphAdjMatrix matrixGraph = new GraphAdjMatrix();
        matrixGraph.initializeMatrix(5);
        matrixGraph.addEdge(0, 1);
        matrixGraph.addEdge(0, 4);
        matrixGraph.addEdge(1, 2);
        matrixGraph.addEdge(2, 3);
        matrixGraph.addEdge(3, 4);
        System.out.println("Adjacency Matrix:");
        matrixGraph.printAdjMatrix();

        // Adjacency List Representation
        GraphAdjList listGraph = new GraphAdjList();
        listGraph.initializeList(5);
        listGraph.addEdge(0, 1);
        listGraph.addEdge(0, 4);
        listGraph.addEdge(1, 2);
        listGraph.addEdge(2, 3);
        listGraph.addEdge(3, 4);
        System.out.println("Adjacency List:");
        listGraph.displayAdjList();
    }
}
```

Q9. Create a class Graph with a linked list member to represent N number of vertices. Implement the required constructor. Add a method to the Graph class for reading a graph and storing it in an adjacency list representation.
Include a depth-first search (DFS) method in the Graph class to traverse the vertices of the graph, and a main method to invoke all the methods.

```java
import java.util.*;

class GraphDFS {
    private LinkedList<Integer>[] adj;
    private int vertices;

    // Constructor to initialize the graph with a given number of vertices
    public GraphDFS(int vertices) {
        this.vertices = vertices;
        initializeList(vertices);
    }

    // Method to initialize the adjacency list
    private void initializeList(int v) {
        adj = new LinkedList[v];
        for (int i = 0; i < v; i++) {
            adj[i] = new LinkedList<>();
        }
    }

    // Method to add an edge to the graph
    public void addEdge(int i, int j) {
        adj[i].add(j);
        adj[j].add(i);
    }

    // Method to read a graph (can be enhanced to read from input)
    public void readGraph(int[][] edges) {
        for (int[] edge : edges) {
            addEdge(edge[0], edge[1]);
        }
    }

    // Method to display the adjacency list representation of the graph
    public void display() {
        for (int i = 0; i < adj.length; i++) {
            System.out.print(i);
            for (int j : adj[i]) {
                System.out.print(" -> " + j);
            }
            System.out.println();
        }
    }

    // Recursive Depth-First Search traversal
    private void dfsRecursive(int vertex, boolean[] visited) {
        visited[vertex] = true; // Mark the current vertex as visited
        System.out.print(vertex + " "); // Print the current vertex

        // Visit all unvisited neighbors of the current vertex recursively
        for (int neighbor : adj[vertex]) {
            if (!visited[neighbor]) {
                dfsRecursive(neighbor, visited);
            }
        }
    }

    // Wrapper method for recursive DFS
    public void dfs(int start) {
        boolean[] visited = new boolean[adj.length]; // Initialize visited array
        dfsRecursive(start, visited); // Call the recursive DFS method
    }

    public static void main(String[] args) {
        // Initialize graph with 8 vertices
        GraphDFS graph = new GraphDFS(8);

        // Define edges of the graph
        int[][] edges = {
            {0, 1}, {0, 2}, {1, 3}, {1, 4}, {2, 5},
            {2, 6}, {3, 7}, {4, 7}, {5, 7}, {6, 7}
        };

        // Read and construct the graph
        graph.readGraph(edges);

        // Display the adjacency list
        System.out.println("Adjacency List:");
        graph.display();

        // Perform and display DFS traversal starting from vertex 0
        System.out.println("DFS Traversal:");
        graph.dfs(0);
    }
}
```

Q10. Write a Java program to traverse a graph using breadth-first search (BFS) and provide the final output of the code. (Use the ArrayDeque collection.)

```
import java.util.ArrayDeque;
import java.util.LinkedList;

public class BFS {
      LinkedList<Integer>[] adj;
      public void initializeList(int v) {
            adj = new LinkedList[v];
            for (int i = 0; i < v; i++)
                  adj[i] = new LinkedList<Integer>();
      }
      public void addEdge(int i, int j) {
            adj[i].add(j);
            adj[j].add(i);
      }
      public void display() {
            for(int i = 0; i < adj.length; i++) {
                  System.out.print(i);
                  for(int j = 0; j < adj[i].size(); j++)
                        System.out.print("->" + adj[i].get(j));
                  System.out.println(" ");
            }
      }
      public void bfs(int start)
      {
            ArrayDeque<Integer> queue=new ArrayDeque<>();
            int element;
            //Mark all the vertices as not visited (By default set as false)
            boolean visited[]=new boolean[8];
            visited[start]=true;
            queue.add(start);
            while(queue.size() != 0) {
                  //Dequeue a vertex from queue and print it
                  element=queue.remove();
                  System.out.print(element + " ");
                  //Get all adjacent vertices and mark visited
                  for(int i=0; i<adj[element].size(); i++) {
                        int n=adj[element].get(i);
                        if(visited[n] != true) {
                              queue.add(n);
                              visited[n]=true;
                        }
                  }
            }
      }
      public static void main(String[] args) {
            BFS list=new BFS();
            list.initializeList(8);
        list.addEdge(0, 1);
        list.addEdge(0, 2);
        list.addEdge(0, 3);
        list.addEdge(1, 4);
        list.addEdge(2, 5);
        list.addEdge(3, 6);
        list.addEdge(4, 7);
        list.addEdge(5, 7);
        list.addEdge(6, 7);
        list.display();
        System.out.println("BFS Traversal:");
        list.bfs(0);
      }
}
```