**1) Create a binary search tree (BST) of and perform following operations: i) Insert ii) Display inorder iii)Search a node iv) Find height of the tree v) level wise display iv) Delete v) Mirror**   
  
import java.util.\*;

public class BSTOperations {

static class Node {

int data;

Node left, right;

Node(int data) {

this.data = data;

}

}

// 1. Insert

public static Node insert(Node root, int val) {

if (root == null) return new Node(val);

if (val < root.data) root.left = insert(root.left, val);

else root.right = insert(root.right, val);

return root;

}

// 2. Inorder Traversal (Display)

public static void inorder(Node root) {

if (root == null) return;

inorder(root.left);

System.out.print(root.data + " ");

inorder(root.right);

}

// 3. Search

public static boolean search(Node root, int key) {

if (root == null) return false;

if (root.data == key) return true;

else if (key < root.data) return search(root.left, key);

else return search(root.right, key);

}

// 4. Height

public static int height(Node root) {

if (root == null) return 0;

return 1 + Math.max(height(root.left), height(root.right));

}

// 5. Level-order (BFS)

public static void levelOrder(Node root) {

if (root == null) return;

Queue<Node> q = new LinkedList<>();

q.add(root);

while (!q.isEmpty()) {

Node curr = q.poll();

System.out.print(curr.data + " ");

if (curr.left != null) q.add(curr.left);

if (curr.right != null) q.add(curr.right);

}

}

// 6. Delete a node

public static Node delete(Node root, int key) {

if (root == null) return null;

if (key < root.data) root.left = delete(root.left, key);

else if (key > root.data) root.right = delete(root.right, key);

else {

// Node with 0 or 1 child

if (root.left == null) return root.right;

if (root.right == null) return root.left;

// Node with 2 children: replace with inorder successor

Node successor = findMin(root.right);

root.data = successor.data;

root.right = delete(root.right, successor.data);

}

return root;

}

private static Node findMin(Node root) {

while (root.left != null) root = root.left;

return root;

}

// 7. Mirror

public static Node mirror(Node root) {

if (root == null) return null;

Node leftMirror = mirror(root.left);

Node rightMirror = mirror(root.right);

root.left = rightMirror;

root.right = leftMirror;

return root;

}

// Main method for demonstration

public static void main(String[] args) {

int[] values = {5, 3, 8, 1, 4, 7, 9};

Node root = null;

for (int val : values) {

root = insert(root, val);

}

System.out.print("Inorder: ");

inorder(root); // Sorted output

System.out.println();

System.out.print("Level Order: ");

levelOrder(root);

System.out.println();

System.out.println("Search 4: " + search(root, 4)); // true

System.out.println("Search 10: " + search(root, 10)); // false

System.out.println("Height: " + height(root)); // e.g., 3

System.out.println("Delete 3");

root = delete(root, 3);

System.out.print("Inorder after deletion: ");

inorder(root);

System.out.println();

System.out.println("Mirror the tree");

root = mirror(root);

System.out.print("Inorder of mirrored tree: ");

inorder(root);

}

}

**2) Create a graph and perform graph traversal using BFS and DFS.**

import java.util.\*;

public class GraphTraversal {

static class Graph {

private Map<Integer, List<Integer>> adj;

Graph() {

adj = new HashMap<>();

}

// Add edge (undirected)

public void addEdge(int u, int v) {

adj.computeIfAbsent(u, k -> new ArrayList<>()).add(v);

adj.computeIfAbsent(v, k -> new ArrayList<>()).add(u);

}

// BFS using Queue

public void bfs(int start) {

Set<Integer> visited = new HashSet<>();

Queue<Integer> queue = new LinkedList<>();

queue.add(start);

visited.add(start);

System.out.print("BFS: ");

while (!queue.isEmpty()) {

int current = queue.poll();

System.out.print(current + " ");

for (int neighbor : adj.getOrDefault(current, new ArrayList<>())) {

if (!visited.contains(neighbor)) {

visited.add(neighbor);

queue.add(neighbor);

}

}

}

System.out.println();

}

// DFS using Stack

public void dfs(int start) {

Set<Integer> visited = new HashSet<>();

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

stack.push(start);

System.out.print("DFS: ");

while (!stack.isEmpty()) {

int current = stack.pop();

if (!visited.contains(current)) {

visited.add(current);

System.out.print(current + " ");

// Push neighbors in reverse order for correct traversal

List<Integer> neighbors = adj.getOrDefault(current, new ArrayList<>());

Collections.reverse(neighbors); // Ensures left-to-right DFS order

for (int neighbor : neighbors) {

if (!visited.contains(neighbor)) {

stack.push(neighbor);

}

}

}

}

System.out.println();

}

}

public static void main(String[] args) {

Graph g = new Graph();

// Create graph

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

g.addEdge(2, 5);

g.addEdge(2, 6);

// Traversals

g.bfs(0); // BFS from node 0

g.dfs(0); // DFS from node 0 using Stack

}

}

**3) a) The internship is offered to students based on rank obtained in the second year of graduation. Create a suitable non-linear data structure to identify the next topper student for internship. (Create max-heap). b) Sort the student data in ascending order of grades.**  
import java.util.\*;

// Student class

class Student {

String name;

int grade; // Grade or rank score

Student(String name, int grade) {

this.name = name;

this.grade = grade;

}

}

public class InternshipSelection {

public static void main(String[] args) {

List<Student> students = Arrays.asList(

new Student("Alice", 92),

new Student("Bob", 85),

new Student("Charlie", 90),

new Student("David", 78),

new Student("Eva", 88)

);

// a) Max-Heap to get next topper

PriorityQueue<Student> maxHeap = new PriorityQueue<>(

(s1, s2) -> Integer.compare(s2.grade, s1.grade)

);

maxHeap.addAll(students);

System.out.println("Top 3 students for internship:");

for (int i = 0; i < 3 && !maxHeap.isEmpty(); i++) {

Student top = maxHeap.poll();

System.out.println((i+1) + ". " + top.name + " (Grade: " + top.grade + ")");

}

// b) Sort students in ascending order of grades

students.sort(Comparator.comparingInt(s -> s.grade));

System.out.println("\n Students sorted by grades (ascending):");

for (Student s : students) {

System.out.println(s.name + " - Grade: " + s.grade);

}

}

}

**4) Divide and Conquer: Implement a program to find minimum and maximum elements from a given list using Divide and Conquer strategy.**

public class MinMaxDivideConquer {

// Helper class to store min and max

static class Result {

int min;

int max;

Result(int min, int max) {

this.min = min;

this.max = max;

}

}

// Divide and Conquer method

public static Result findMinMax(int[] arr, int low, int high) {

// Only one element

if (low == high) {

return new Result(arr[low], arr[low]);

}

// Two elements

if (high == low + 1) {

int min = Math.min(arr[low], arr[high]);

int max = Math.max(arr[low], arr[high]);

return new Result(min, max);

}

// More than two elements

int mid = (low + high) / 2;

Result left = findMinMax(arr, low, mid);

Result right = findMinMax(arr, mid + 1, high);

int min = Math.min(left.min, right.min);

int max = Math.max(left.max, right.max);

return new Result(min, max);

}

public static void main(String[] args) {

int[] arr = {23, 45, 12, 67, 5, 90, 34};

Result result = findMinMax(arr, 0, arr.length - 1);

System.out.println("Minimum element: " + result.min);

System.out.println("Maximum element: " + result.max);

}

}

**5) Greedy approach: A business house has several offices in different countries; they want to lease phone lines to connect them with each other and the phone company charges different rent to connect different pairs of cities. Business house wants to connect all its offices with a minimum total cost. Represent using appropriate data structure. Apply Prim’s and Kruskal’s algorithm to find the minimum total cost.**

import java.util.\*;

class Connection implements Comparable<Connection> {

int city1, city2, cost;

Connection(int city1, int city2, int cost) {

this.city1 = city1;

this.city2 = city2;

this.cost = cost;

}

public int compareTo(Connection other) {

return this.cost - other.cost;

}

}

class KruskalAlgorithm {

int[] parent;

// Find root parent

int find(int node) {

if (parent[node] == node)

return node;

return parent[node] = find(parent[node]);

}

// Union of two sets

void union(int a, int b) {

parent[find(a)] = find(b);

}

// Kruskal's MST

void kruskal(List<Connection> connections, int totalCities) {

Collections.sort(connections); // sort by cost

parent = new int[totalCities];

for (int i = 0; i < totalCities; i++)

parent[i] = i;

int totalCost = 0;

System.out.println("\nKruskal's MST:");

for (Connection c : connections) {

if (find(c.city1) != find(c.city2)) {

System.out.println("Connect City " + c.city1 + " <--> City " + c.city2 + " = " + c.cost);

totalCost += c.cost;

union(c.city1, c.city2);

}

}

System.out.println("Total Minimum Cost (Kruskal): " + totalCost);

}

}

class PrimAlgorithm {

boolean[] visited;

// Prim's MST

void prim(List<List<Connection>> graph, int totalCities) {

visited = new boolean[totalCities];

PriorityQueue<Connection> pq = new PriorityQueue<>();

visited[0] = true;

pq.addAll(graph.get(0));

int totalCost = 0;

System.out.println("\nPrim's MST:");

while (!pq.isEmpty()) {

Connection c = pq.poll();

if (visited[c.city2])

continue;

System.out.println("Connect City " + c.city1 + " <--> City " + c.city2 + " = " + c.cost);

totalCost += c.cost;

visited[c.city2] = true;

for (Connection next : graph.get(c.city2)) {

if (!visited[next.city2])

pq.add(next);

}

}

System.out.println("Total Minimum Cost (Prim): " + totalCost);

}

}

public class CityPhoneNetwork {

public static void main(String[] args) {

int totalCities = 5;

// List of all connections between cities with cost

List<Connection> connections = Arrays.asList(

new Connection(0, 1, 10),

new Connection(0, 2, 20),

new Connection(1, 2, 30),

new Connection(1, 3, 25),

new Connection(2, 4, 40),

new Connection(3, 4, 15)

);

// Run Kruskal’s algorithm

KruskalAlgorithm kruskalObj = new KruskalAlgorithm();

kruskalObj.kruskal(connections, totalCities);

// Build adjacency list for Prim's algorithm

List<List<Connection>> graph = new ArrayList<>();

for (int i = 0; i < totalCities; i++)

graph.add(new ArrayList<>());

for (Connection c : connections) {

graph.get(c.city1).add(new Connection(c.city1, c.city2, c.cost));

graph.get(c.city2).add(new Connection(c.city2, c.city1, c.cost));

}

// Run Prim’s algorithm

PrimAlgorithm primObj = new PrimAlgorithm();

primObj.prim(graph, totalCities);

}

}

**6) Greedy and dynamic approach: a) Use the map of the area around the college as the graph. Identify the prominent landmarks as nodes and find minimum distance to various landmarks from the college as the source. Represent this graph using an adjacency matrix. Find the shortest path using Dijkstra’s algorithm. b) Write a program to implement the Bellman-Ford algorithm to find the shortest path from a single source to all other nodes in a graph with negative edge weights. Verify its results for a sample graph and compare it with Dijkstra’s algorithm.**  
  
a) import java.util.\*;

public class DijkstraAlgorithm {

public static void dijkstra(int[][] graph, int source) {

int V = graph.length;

int[] dist = new int[V];

boolean[] visited = new boolean[V];

Arrays.fill(dist, Integer.MAX\_VALUE);

dist[source] = 0;

for (int count = 0; count < V - 1; count++) {

int u = findMinDistance(dist, visited);

visited[u] = true;

for (int v = 0; v < V; v++) {

if (!visited[v] && graph[u][v] != 0 &&

dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

System.out.println("🔹 Dijkstra’s Shortest Distances from College (Node 0):");

for (int i = 0; i < V; i++) {

System.out.println("To Landmark " + i + " -> " + dist[i]);

}

}

public static int findMinDistance(int[] dist, boolean[] visited) {

int min = Integer.MAX\_VALUE, minIndex = -1;

for (int i = 0; i < dist.length; i++) {

if (!visited[i] && dist[i] <= min) {

min = dist[i];

minIndex = i;

}

}

return minIndex;

}

public static void main(String[] args) {

// Adjacency matrix of 5 landmarks (0 = college)

int[][] graph = {

{0, 10, 0, 30, 100},

{10, 0, 50, 0, 0},

{0, 50, 0, 20, 10},

{30, 0, 20, 0, 60},

{100, 0, 10, 60, 0}

};

dijkstra(graph, 0);

}

}

b) import java.util.\*;

public class BellmanFordAlgorithm {

static class Edge {

int src, dest, weight;

Edge(int s, int d, int w) {

src = s;

dest = d;

weight = w;

}

}

public static void bellmanFord(List<Edge> edges, int V, int source) {

int[] dist = new int[V];

Arrays.fill(dist, Integer.MAX\_VALUE);

dist[source] = 0;

for (int i = 1; i < V; i++) {

for (Edge e : edges) {

if (dist[e.src] != Integer.MAX\_VALUE && dist[e.src] + e.weight < dist[e.dest]) {

dist[e.dest] = dist[e.src] + e.weight;

}

}

}

// Check for negative weight cycles

for (Edge e : edges) {

if (dist[e.src] != Integer.MAX\_VALUE && dist[e.src] + e.weight < dist[e.dest]) {

System.out.println("Graph contains negative weight cycle!");

return;

}

}

System.out.println("\n🔹 Bellman-Ford Distances from College (Node 0):");

for (int i = 0; i < V; i++) {

System.out.println("To Landmark " + i + " -> " + dist[i]);

}

}

public static void main(String[] args) {

int V = 5; // number of landmarks

List<Edge> edges = new ArrayList<>();

// Sample graph with possible negative edge (but no negative cycle)

edges.add(new Edge(0, 1, 6));

edges.add(new Edge(0, 3, 7));

edges.add(new Edge(1, 2, 5));

edges.add(new Edge(1, 3, 8));

edges.add(new Edge(1, 4, -4));

edges.add(new Edge(2, 1, -2));

edges.add(new Edge(3, 2, -3));

edges.add(new Edge(3, 4, 9));

edges.add(new Edge(4, 0, 2));

edges.add(new Edge(4, 2, 7));

bellmanFord(edges, V, 0);

}

}  
  
**7) Backtracking: a) Solve Graph Coloring problem using backtracking approaches. b) N-Queens Problem: Write a recursive program to find the solution of placing N- queens on a chess board so that no queen takes each other.**

a) import java.util.\*;

public class GraphColoring {

private static boolean isSafe(int node, int color[], int[][] graph, int c) {

for (int i = 0; i < graph.length; i++) {

if (graph[node][i] == 1 && color[i] == c) {

return false; // Adjacent node has the same color

}

}

return true;

}

private static boolean graphColoringUtil(int[][] graph, int m, int color[], int node) {

if (node == graph.length) {

return true; // All nodes are colored

}

for (int c = 1; c <= m; c++) {

if (isSafe(node, color, graph, c)) {

color[node] = c; // Assign color to node

if (graphColoringUtil(graph, m, color, node + 1)) {

return true;

}

color[node] = 0; // Backtrack if not valid

}

}

return false; // No valid color assignment

}

public static boolean graphColoring(int[][] graph, int m) {

int[] color = new int[graph.length];

Arrays.fill(color, 0); // No color assigned initially

if (graphColoringUtil(graph, m, color, 0)) {

System.out.println("Solution: ");

for (int i = 0; i < color.length; i++) {

System.out.println("Node " + i + " is assigned color " + color[i]);

}

return true;

}

return false;

}

public static void main(String[] args) {

// Example graph represented as adjacency matrix

int[][] graph = {

{0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0}

};

int m = 3; // Number of colors

if (!graphColoring(graph, m)) {

System.out.println("Solution does not exist");

}

}

}

b) public class NQueens {

private static boolean isSafe(int board[][], int row, int col, int N) {

// Check the column

for (int i = 0; i < row; i++) {

if (board[i][col] == 1) {

return false; // Queen already present in the same column

}

}

// Check the upper-left diagonal

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {

if (board[i][j] == 1) {

return false; // Queen already present in the upper-left diagonal

}

}

// Check the upper-right diagonal

for (int i = row, j = col; i >= 0 && j < N; i--, j++) {

if (board[i][j] == 1) {

return false; // Queen already present in the upper-right diagonal

}

}

return true; // Safe position

}

private static boolean solveNQueensUtil(int board[][], int row, int N) {

if (row == N) {

return true; // All queens are placed

}

for (int col = 0; col < N; col++) {

if (isSafe(board, row, col, N)) {

board[row][col] = 1; // Place queen

if (solveNQueensUtil(board, row + 1, N)) {

return true;

}

board[row][col] = 0; // Backtrack if not safe

}

}

return false; // No solution found

}

public static void solveNQueens(int N) {

int[][] board = new int[N][N];

if (!solveNQueensUtil(board, 0, N)) {

System.out.println("Solution does not exist");

} else {

System.out.println("Solution:");

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

if (board[i][j] == 1) {

System.out.print("Q ");

} else {

System.out.print(". ");

}

}

System.out.println();

}

}

}

public static void main(String[] args) {

int N = 4; // Size of the board and number of queens

solveNQueens(N);

}

}  
  
**8) Branch and Bound: a) Write a program to solve the travelling salesman problem. Print the path and the cost. b) Implement branch and bound for the 0/1 Knapsack problem.**

**TSP**

**a)** import java.util.\*;

public class TravelingSalesmanProblem {

static final int INF = Integer.MAX\_VALUE;

int n;

int[][] dist;

int[] visited;

public TravelingSalesmanProblem(int[][] dist) {

this.n = dist.length;

this.dist = dist;

this.visited = new int[n];

Arrays.fill(this.visited, -1); // Mark all cities as unvisited

}

// Calculate the minimum path using Branch and Bound

public void tsp() {

int[] path = new int[n];

Arrays.fill(path, -1);

int ans = Integer.MAX\_VALUE;

// Start from city 0

visited[0] = 0; // The salesman starts from city 0

path[0] = 0;

ans = branchAndBound(path, 0, 1, 0, ans);

if (ans == INF) {

System.out.println("Solution does not exist");

} else {

System.out.println("Minimum cost path: ");

for (int i = 0; i < n; i++) {

System.out.print(path[i] + " ");

}

System.out.println("\nMinimum cost: " + ans);

}

}

// Branch and Bound recursive function to calculate the path and minimum cost

private int branchAndBound(int[] path, int currPos, int count, int cost, int ans) {

if (count == n) {

if (dist[currPos][0] != 0) { // If there's a way to return to the starting city

int totalCost = cost + dist[currPos][0];

if (totalCost < ans) {

ans = totalCost;

path[count] = 0; // Mark the path to return to the starting city

}

}

return ans;

}

for (int i = 0; i < n; i++) {

if (visited[i] == -1 && dist[currPos][i] != 0) {

visited[i] = 0;

path[count] = i;

ans = branchAndBound(path, i, count + 1, cost + dist[currPos][i], ans);

visited[i] = -1; // Backtrack

}

}

return ans;

}

public static void main(String[] args) {

// Sample distance matrix

int[][] dist = {

{0, 10, 15, 20, 25},

{10, 0, 35, 25, 30},

{15, 35, 0, 30, 5},

{20, 25, 30, 0, 15},

{25, 30, 5, 15, 0}

};

TravelingSalesmanProblem tsp = new TravelingSalesmanProblem(dist);

tsp.tsp();

}

}

**Knapsack**  
**b)** import java.util.\*;

public class KnapsackBranchBound {

static class Item {

int value, weight;

Item(int v, int w) {

value = v;

weight = w;

}

}

static class Node {

int level, profit, weight;

double bound;

Node(int l, int p, int w, double b) {

level = l;

profit = p;

weight = w;

bound = b;

}

}

public static double calculateBound(Node u, int n, int W, Item[] items) {

if (u.weight >= W) return 0;

double bound = u.profit;

int j = u.level + 1;

int totalWeight = u.weight;

while (j < n && totalWeight + items[j].weight <= W) {

totalWeight += items[j].weight;

bound += items[j].value;

j++;

}

if (j < n) {

bound += (W - totalWeight) \* items[j].value / items[j].weight;

}

return bound;

}

public static void knapsack(int W, Item[] items, int n) {

PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingDouble(o -> -o.bound));

Node u = new Node(-1, 0, 0, 0.0);

pq.add(u);

int maxProfit = 0;

while (!pq.isEmpty()) {

u = pq.poll();

if (u.level == n - 1) continue;

Node v = new Node(u.level + 1, u.profit + items[u.level + 1].value, u.weight + items[u.level + 1].weight, 0.0);

if (v.weight <= W && v.profit > maxProfit) {

maxProfit = v.profit;

}

v.bound = calculateBound(v, n, W, items);

if (v.bound > maxProfit) {

pq.add(v);

}

v = new Node(u.level + 1, u.profit, u.weight, 0.0);

v.bound = calculateBound(v, n, W, items);

if (v.bound > maxProfit) {

pq.add(v);

}

}

System.out.println("Maximum profit in 0/1 Knapsack is: " + maxProfit);

}

public static void main(String[] args) {

int W = 50; // Knapsack capacity

Item[] items = {

new Item(60, 10),

new Item(100, 20),

new Item(120, 30)

};

int n = items.length;

knapsack(W, items, n);

}

}