CSA0961 – JAVA

ASSIGNMENT – 10

1 . Create a generic method sortList that takes a list of comparable elements and sorts it.Demonstrate this method with a list of Stringsand a list of Integers.

PROGRAM :

```
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;

// Generic method to sort a list of Comparable elements
public class GenericSorter {

   // Generic method to sort a list
   public static <T extends Comparable<T>> void sortList(List<T> list) {
      Collections.sort(list);
   }

   public static void main(String[] args) {
      // Demonstrating with a list of Strings
      List<String> stringList = new ArrayList<>();
      stringList.add("Banana");
      stringList.add("Apple");
      stringList.add("Cherry");

      System.out.println("Before sorting (Strings): " + stringList);
      sortList(stringList);
      System.out.println("After sorting (Strings): " + stringList);

      // Demonstrating with a list of Integers
      List<Integer> integerList = new ArrayList<>();
      integerList.add(5);
      integerList.add(2);
      integerList.add(8);
      integerList.add(1);

      System.out.println("Before sorting (Integers): " + integerList);
      sortList(integerList);
      System.out.println("After sorting (Integers): " + integerList);
   }
}
```

OUTPUT :

```
Before sorting (Strings): [Banana, Apple, Cherry]
After sorting (Strings): [Apple, Banana, Cherry]
Before sorting (Integers): [5, 2, 8, 1]
After sorting (Integers): [1, 2, 5, 8]
```

2. Write a generic class TreeNode<T> representing a node in a tree with children. Implement methods to add children, traverse the tree(e.g., depth-first search), and find a node by value. Demonstrate thiswith a tree of Strings and Integers.

PROGRAM :

```
import java.util.ArrayList;

import java.util.List;


// Generic TreeNode class

class TreeNode<T> {

   private T value;

   private List<TreeNode<T>> children;


   // Constructor

   public TreeNode(T value) {

      this.value = value;

      this.children = new ArrayList<>();

   }


   // Method to add a child

   public void addChild(TreeNode<T> child) {

      children.add(child);

   }


   // Method to traverse the tree using Depth-First Search (DFS)

   public void traverse() {

      System.out.println(value);

      for (TreeNode<T> child : children) {

         child.traverse();

      }

   }


   // Method to find a node by value

   public TreeNode<T> findNodeByValue(T searchValue) {
```

```
        if (value.equals(searchValue)) {
            return this;
        }
        for (TreeNode<T> child : children) {
            TreeNode<T> result = child.findNodeByValue(searchValue);
            if (result != null) {
                return result;
            }
        }
        return null;
    }

    // Getters
    public T getValue() {
        return value;
    }

    public List<TreeNode<T>> getChildren() {
        return children;
    }
}

// Main class to demonstrate TreeNode functionality
public class TreeNodeDemo {
    public static void main(String[] args) {
        // Demonstrating with a tree of Strings
        TreeNode<String> rootString = new TreeNode<>("Root");
        TreeNode<String> child1String = new TreeNode<>("Child1");
        TreeNode<String> child2String = new TreeNode<>("Child2");
        TreeNode<String> grandChild1String = new TreeNode<>("GrandChild1");

        rootString.addChild(child1String);
```

```
        rootString.addChild(child2String);
        child1String.addChild(grandChild1String);

        System.out.println("String Tree Traversal:");
        rootString.traverse();

        TreeNode<String> foundNodeString = rootString.findNodeByValue("Child2");
        System.out.println("Found Node (String): " + (foundNodeString != null ?
foundNodeString.getValue() : "Not Found"));

        // Demonstrating with a tree of Integers
        TreeNode<Integer> rootInteger = new TreeNode<>(1);
        TreeNode<Integer> child1Integer = new TreeNode<>(2);
        TreeNode<Integer> child2Integer = new TreeNode<>(3);
        TreeNode<Integer> grandChild1Integer = new TreeNode<>(4);

        rootInteger.addChild(child1Integer);
        rootInteger.addChild(child2Integer);
        child1Integer.addChild(grandChild1Integer);

        System.out.println("\nInteger Tree Traversal:");
        rootInteger.traverse();

        TreeNode<Integer> foundNodeInteger = rootInteger.findNodeByValue(3);
        System.out.println("Found Node (Integer): " + (foundNodeInteger != null ?
foundNodeInteger.getValue() : "Not Found"));
    }
}
```

OUTPUT :

```
String Tree Traversal:
Root
Child1
GrandChild1
Child2
Found Node (String): Child2

Integer Tree Traversal:
1
2
4
3
Found Node (Integer): 3
```

3. Implement a generic class GenericPriorityQueue<T extendsComparable<T>> with methods like enqueue, dequeue, and peek.The elements should be dequeued in priority order. Demonstrate with Integer and String.

PROGRAM :

```
import java.util.PriorityQueue;


// Generic class for a priority queue

public class GenericPriorityQueue<T extends Comparable<T>> {

    private PriorityQueue<T> priorityQueue;


    // Constructor

    public GenericPriorityQueue() {

        this.priorityQueue = new PriorityQueue<>();

    }


    // Enqueue method to add elements to the queue

    public void enqueue(T element) {

        priorityQueue.offer(element);

    }


    // Dequeue method to remove and return the highest priority element

    public T dequeue() {

        return priorityQueue.poll();

    }
```

```
// Peek method to view the highest priority element without removing it
public T peek() {
    return priorityQueue.peek();
}

// Check if the queue is empty
public boolean isEmpty() {
    return priorityQueue.isEmpty();
}

// Main method to demonstrate the functionality
public static void main(String[] args) {
    // Demonstrating with Integer
    GenericPriorityQueue<Integer> intQueue = new GenericPriorityQueue<>();
    intQueue.enqueue(5);
    intQueue.enqueue(1);
    intQueue.enqueue(3);

    System.out.println("Integer PriorityQueue:");
    while (!intQueue.isEmpty()) {
        System.out.println("Peek: " + intQueue.peek());
        System.out.println("Dequeue: " + intQueue.dequeue());
    }

    // Demonstrating with String
    GenericPriorityQueue<String> strQueue = new GenericPriorityQueue<>();
    strQueue.enqueue("apple");
    strQueue.enqueue("banana");
    strQueue.enqueue("cherry");

    System.out.println("\nString PriorityQueue:");
    while (!strQueue.isEmpty()) {
```

```
            System.out.println("Peek: " + strQueue.peek());

            System.out.println("Dequeue: " + strQueue.dequeue());

        }

    }

}
```

OUTPUT :

```
Integer PriorityQueue:
Peek: 1
Dequeue: 1
Peek: 3
Dequeue: 3
Peek: 5
Dequeue: 5

String PriorityQueue:
Peek: apple
Dequeue: apple
Peek: banana
Dequeue: banana
Peek: cherry
Dequeue: cherry
```

4. Design a generic class Graph<T> with methods for adding nodes,adding edges, and performing graph traversals (e.g., BFS and DFS).Ensure that the graph can handle both directed and undirectedgraphs. Demonstrate with a graph of String nodes and another graphof Integer nodes.

PROGRAM :

```
import java.util.*;


public class Graph<T> {

    private final Map<T, List<T>> adjacencyList;

    private final boolean isDirected;


    // Constructor to initialize the graph

    public Graph(boolean isDirected) {

        this.adjacencyList = new HashMap<>();

        this.isDirected = isDirected;

    }


    // Method to add a node to the graph
```

```
public void addNode(T node) {
    adjacencyList.putIfAbsent(node, new ArrayList<>());
}

// Method to add an edge to the graph
public void addEdge(T from, T to) {
    adjacencyList.putIfAbsent(from, new ArrayList<>());
    adjacencyList.putIfAbsent(to, new ArrayList<>());
    adjacencyList.get(from).add(to);
    if (!isDirected) {
        adjacencyList.get(to).add(from);
    }
}

// Method to perform Breadth-First Search (BFS)
public void bfs(T start) {
    if (!adjacencyList.containsKey(start)) {
        System.out.println("Node not found.");
        return;
    }

    Set<T> visited = new HashSet<>();
    Queue<T> queue = new LinkedList<>();
    queue.add(start);
    visited.add(start);

    while (!queue.isEmpty()) {
        T node = queue.poll();
        System.out.print(node + " ");

        for (T neighbor : adjacencyList.get(node)) {
            if (!visited.contains(neighbor)) {
```

```
                visited.add(neighbor);
                queue.add(neighbor);
            }
        }
    }
    System.out.println();
}

// Method to perform Depth-First Search (DFS)
public void dfs(T start) {
    if (!adjacencyList.containsKey(start)) {
        System.out.println("Node not found.");
        return;
    }

    Set<T> visited = new HashSet<>();
    dfsUtil(start, visited);
    System.out.println();
}

private void dfsUtil(T node, Set<T> visited) {
    visited.add(node);
    System.out.print(node + " ");

    for (T neighbor : adjacencyList.get(node)) {
        if (!visited.contains(neighbor)) {
            dfsUtil(neighbor, visited);
        }
    }
}

// Method to print the graph
```

```java
public void printGraph() {
    for (Map.Entry<T, List<T>> entry : adjacencyList.entrySet()) {
        System.out.print(entry.getKey() + " -> ");
        for (T neighbor : entry.getValue()) {
            System.out.print(neighbor + " ");
        }
        System.out.println();
    }
}

// Main method to demonstrate the graph with String and Integer nodes
public static void main(String[] args) {
    // Graph with String nodes
    Graph<String> stringGraph = new Graph<>(false); // Undirected graph
    stringGraph.addNode("A");
    stringGraph.addNode("B");
    stringGraph.addNode("C");
    stringGraph.addEdge("A", "B");
    stringGraph.addEdge("A", "C");
    stringGraph.addEdge("B", "C");

    System.out.println("String Graph:");
    stringGraph.printGraph();
    System.out.print("BFS starting from A: ");
    stringGraph.bfs("A");
    System.out.print("DFS starting from A: ");
    stringGraph.dfs("A");

    // Graph with Integer nodes
    Graph<Integer> intGraph = new Graph<>(true); // Directed graph
    intGraph.addNode(1);
    intGraph.addNode(2);
```

```
        intGraph.addNode(3);
        intGraph.addEdge(1, 2);
        intGraph.addEdge(2, 3);
        intGraph.addEdge(1, 3);

        System.out.println("\nInteger Graph:");
        intGraph.printGraph();
        System.out.print("BFS starting from 1: ");
        intGraph.bfs(1);
        System.out.print("DFS starting from 1: ");
        intGraph.dfs(1);
    }
}
```

OUTPUT :

```
String Graph:
A -> B C
B -> A C
C -> A B
BFS starting from A: A B C
DFS starting from A: A B C

Integer Graph:
1 -> 2 3
2 -> 3
3 ->
BFS starting from 1: 1 2 3
DFS starting from 1: 1 2 3
```

5. Create a generic class Matrix<T extends Number> that represents amatrix and supports operations like addition, subtraction, andmultiplication of matrices. Ensure that the operations are type-safe and efficient. Demonstrate with matrices of Integer and Double.

PROGRAM :

```
public class Matrix<T extends Number> {
    private final int rows;
    private final int cols;
    private final T[][] data;
    private final Class<T> type;
```

```
@SuppressWarnings("unchecked")
public Matrix(int rows, int cols, Class<T> type) {
    this.rows = rows;
    this.cols = cols;
    this.type = type;
    this.data = (T[][]) new Number[rows][cols];
}

// Method to set the value at a specific position
public void set(int row, int col, T value) {
    data[row][col] = value;
}

// Method to get the value from a specific position
public T get(int row, int col) {
    return data[row][col];
}

// Matrix addition
public Matrix<T> add(Matrix<T> other) {
    if (this.rows != other.rows || this.cols != other.cols) {
        throw new IllegalArgumentException("Matrix dimensions must match for addition.");
    }
    Matrix<T> result = new Matrix<>(rows, cols, type);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result.set(i, j, addNumbers(this.get(i, j), other.get(i, j)));
        }
    }
    return result;
}
```

```java
// Matrix subtraction
public Matrix<T> subtract(Matrix<T> other) {
    if (this.rows != other.rows || this.cols != other.cols) {
        throw new IllegalArgumentException("Matrix dimensions must match for subtraction.");
    }
    Matrix<T> result = new Matrix<>(rows, cols, type);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result.set(i, j, subtractNumbers(this.get(i, j), other.get(i, j)));
        }
    }
    return result;
}

// Matrix multiplication
public Matrix<T> multiply(Matrix<T> other) {
    if (this.cols != other.rows) {
        throw new IllegalArgumentException("Matrix dimensions must match for multiplication.");
    }
    Matrix<T> result = new Matrix<>(this.rows, other.cols, type);
    for (int i = 0; i < this.rows; i++) {
        for (int j = 0; j < other.cols; j++) {
            T sum = type == Integer.class ? (T) Integer.valueOf(0) : (T) Double.valueOf(0);
            for (int k = 0; k < this.cols; k++) {
                sum = addNumbers(sum, multiplyNumbers(this.get(i, k), other.get(k, j)));
            }
            result.set(i, j, sum);
        }
    }
    return result;
}
```

```java
// Add two numbers
private T addNumbers(T a, T b) {
    if (type == Integer.class) {
        return (T) Integer.valueOf(a.intValue() + b.intValue());
    } else if (type == Double.class) {
        return (T) Double.valueOf(a.doubleValue() + b.doubleValue());
    }
    throw new UnsupportedOperationException("Type not supported for addition.");
}

// Subtract two numbers
private T subtractNumbers(T a, T b) {
    if (type == Integer.class) {
        return (T) Integer.valueOf(a.intValue() - b.intValue());
    } else if (type == Double.class) {
        return (T) Double.valueOf(a.doubleValue() - b.doubleValue());
    }
    throw new UnsupportedOperationException("Type not supported for subtraction.");
}

// Multiply two numbers
private T multiplyNumbers(T a, T b) {
    if (type == Integer.class) {
        return (T) Integer.valueOf(a.intValue() * b.intValue());
    } else if (type == Double.class) {
        return (T) Double.valueOf(a.doubleValue() * b.doubleValue());
    }
    throw new UnsupportedOperationException("Type not supported for multiplication.");
}

// Method to print the matrix
public void printMatrix() {
```

```
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            System.out.print(data[i][j] + "\t");
        }
        System.out.println();
    }
}

// Main method to demonstrate with Integer and Double matrices
public static void main(String[] args) {
    // Integer Matrix
    Matrix<Integer> intMatrix1 = new Matrix<>(2, 2, Integer.class);
    Matrix<Integer> intMatrix2 = new Matrix<>(2, 2, Integer.class);

    intMatrix1.set(0, 0, 1);
    intMatrix1.set(0, 1, 2);
    intMatrix1.set(1, 0, 3);
    intMatrix1.set(1, 1, 4);

    intMatrix2.set(0, 0, 5);
    intMatrix2.set(0, 1, 6);
    intMatrix2.set(1, 0, 7);
    intMatrix2.set(1, 1, 8);

    System.out.println("Integer Matrix 1:");
    intMatrix1.printMatrix();
    System.out.println("Integer Matrix 2:");
    intMatrix2.printMatrix();

    System.out.println("Addition Result:");
    Matrix<Integer> intAddResult = intMatrix1.add(intMatrix2);
    intAddResult.printMatrix();
```

```java
System.out.println("Subtraction Result:");
Matrix<Integer> intSubResult = intMatrix1.subtract(intMatrix2);
intSubResult.printMatrix();

System.out.println("Multiplication Result:");
Matrix<Integer> intMulResult = intMatrix1.multiply(intMatrix2);
intMulResult.printMatrix();

// Double Matrix
Matrix<Double> doubleMatrix1 = new Matrix<>(2, 2, Double.class);
Matrix<Double> doubleMatrix2 = new Matrix<>(2, 2, Double.class);

doubleMatrix1.set(0, 0, 1.1);
doubleMatrix1.set(0, 1, 2.2);
doubleMatrix1.set(1, 0, 3.3);
doubleMatrix1.set(1, 1, 4.4);

doubleMatrix2.set(0, 0, 5.5);
doubleMatrix2.set(0, 1, 6.6);
doubleMatrix2.set(1, 0, 7.7);
doubleMatrix2.set(1, 1, 8.8);

System.out.println("\nDouble Matrix 1:");
doubleMatrix1.printMatrix();
System.out.println("Double Matrix 2:");
doubleMatrix2.printMatrix();

System.out.println("Addition Result:");
Matrix<Double> doubleAddResult = doubleMatrix1.add(doubleMatrix2);
doubleAddResult.printMatrix();
```

```
        System.out.println("Subtraction Result:");
        Matrix<Double> doubleSubResult = doubleMatrix1.subtract(doubleMatrix2);
        doubleSubResult.printMatrix();

        System.out.println("Multiplication Result:");
        Matrix<Double> doubleMulResult = doubleMatrix1.multiply(doubleMatrix2);
        doubleMulResult.printMatrix();
    }
}
```

OUTPUT :

```
Integer Matrix 1:
1       2
3       4
Integer Matrix 2:
5       6
7       8
Addition Result:
6       8
10      12
Subtraction Result:
-4      -4
-4      -4
Multiplication Result:
19      22
43      50

Double Matrix 1:
1.1     2.2
3.3     4.4
Double Matrix 2:
5.5     6.6
7.7     8.8
Addition Result:
6.6     8.8
11.0    13.200000000000001
Subtraction Result:
-4.4    -4.3999999999999995
-4.4    -4.4
Multiplication Result:
22.990000000000002      26.620000000000005
52.03   60.5
```