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

A.
```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class GenericSorter {
    public static <T extends Comparable<T>> void sortList(List<T> list) {
        Collections.sort(list);
    }
    public static void main(String[] args) {
        List<String> stringList = new ArrayList<>();
        stringList.add("Apple");
        stringList.add("Orange");
        stringList.add("Banana");
        stringList.add("Mango");
        System.out.println("Before sorting (Strings): " + stringList);
        sortList(stringList);
        System.out.println("After sorting (Strings): " + stringList);
        List<Integer> intList = new ArrayList<>();
        intList.add(5);
        intList.add(3);
        intList.add(8);
        intList.add(1);
        System.out.println("Before sorting (Integers): " + intList);
        sortList(intList);
        System.out.println("After sorting (Integers): " + intList);
    }
}
```

Output:

```
Before sorting (Strings): [Apple, Orange, Banana, Mango]
After sorting (Strings): [Apple, Banana, Mango, Orange]
Before sorting (Integers): [5, 3, 8, 1]
After sorting (Integers): [1, 3, 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 this with a tree of Strings and Integers.

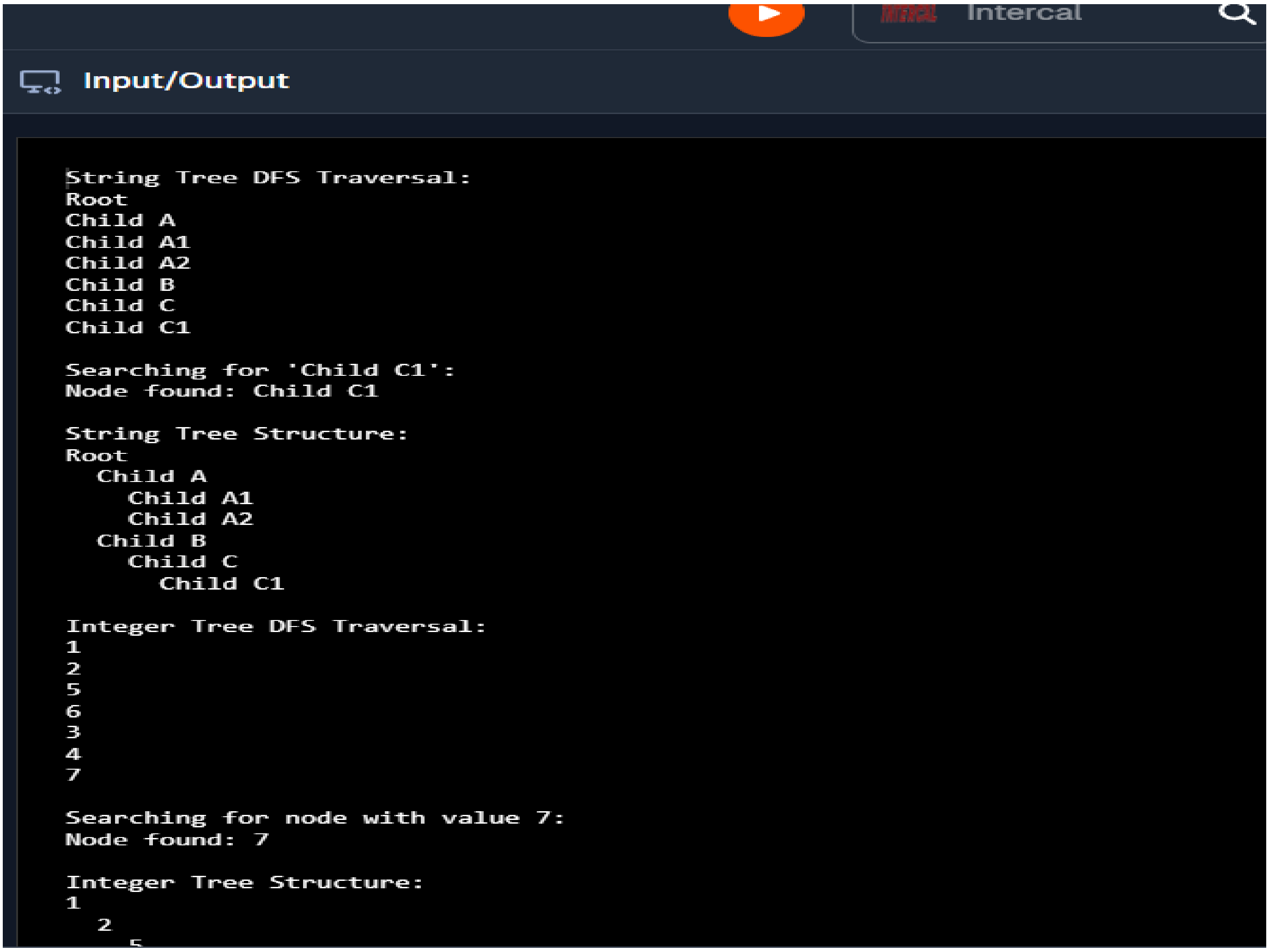
A.

```
import java.util.ArrayList;

import java.util.List;

public class TreeNode<T> {

    private T value;

    private List<TreeNode<T>> children;

    public TreeNode(T value) {

        this.value = value;

        this.children = new ArrayList<>();

    }

    public void addChild(TreeNode<T> child) {

        children.add(child);

    }

    public T getValue() {

        return value;

    }

    public List<TreeNode<T>> getChildren() {

        return children;

    }

    public void depthFirstTraversal() {

        System.out.println(value);

        for (TreeNode<T> child : children) {
```

```
        child.depthFirstTraversal();
    }
}
public TreeNode<T> findNode(T value) {
    if (this.value.equals(value)) {
        return this;
    }
    for (TreeNode<T> child : children) {
        TreeNode<T> result = child.findNode(value);
        if (result != null) {
            return result;
        }
    }
    return null; // Node not found
}
public void printTree(String prefix) {
    System.out.println(prefix + value);
    for (TreeNode<T> child : children) {
        child.printTree(prefix + "  ");
    }
}
public static void main(String[] args) {
    TreeNode<String> rootString = new TreeNode<>("Root");
    TreeNode<String> childA = new TreeNode<>("Child A");
    TreeNode<String> childB = new TreeNode<>("Child B");
    TreeNode<String> childC = new TreeNode<>("Child C");
    rootString.addChild(childA);
    rootString.addChild(childB);
    childA.addChild(new TreeNode<>("Child A1"));
    childA.addChild(new TreeNode<>("Child A2"));
    childB.addChild(childC);
```

```
        childC.addChild(new TreeNode<>("Child C1"));

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

        rootString.depthFirstTraversal();

        System.out.println("\nSearching for 'Child C1':");

        TreeNode<String> foundNode = rootString.findNode("Child C1");

        System.out.println(foundNode != null ? "Node found: " + foundNode.getValue() : "Node not found.");

        System.out.println("\nString Tree Structure:");

        rootString.printTree("");

        TreeNode<Integer> rootInteger = new TreeNode<>(1);

        TreeNode<Integer> intChildA = new TreeNode<>(2);

        TreeNode<Integer> intChildB = new TreeNode<>(3);

        TreeNode<Integer> intChildC = new TreeNode<>(4);

        rootInteger.addChild(intChildA);

        rootInteger.addChild(intChildB);

        intChildA.addChild(new TreeNode<>(5));

        intChildA.addChild(new TreeNode<>(6));

        intChildB.addChild(intChildC);

        intChildC.addChild(new TreeNode<>(7));

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

        rootInteger.depthFirstTraversal();

        System.out.println("\nSearching for node with value 7:");

        TreeNode<Integer> foundIntNode = rootInteger.findNode(7);

        System.out.println(foundIntNode != null ? "Node found: " + foundIntNode.getValue() : "Node not found.");

        System.out.println("\nInteger Tree Structure:");

        rootInteger.printTree("");

    }

}
```

Output:



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

A.

```
import java.util.PriorityQueue;
public class GenericPriorityQueue<T extends Comparable<T>> {
    private PriorityQueue<T> queue;
    public GenericPriorityQueue() {
        queue = new PriorityQueue<>();
    }
    public void enqueue(T element) {
        queue.offer(element);
```

```
}
public T dequeue() {
    return queue.poll();
}
public T peek() {
    return queue.peek();
}
public boolean isEmpty() {
    return queue.isEmpty();
}
public static void main(String[] args) {
    GenericPriorityQueue<Integer> intQueue = new GenericPriorityQueue<>();
    intQueue.enqueue(5);
    intQueue.enqueue(1);
    intQueue.enqueue(3);
    intQueue.enqueue(10);
    intQueue.enqueue(2);
    System.out.println("Integer Priority Queue:");
    while (!intQueue.isEmpty()) {
        System.out.println("Dequeue: " + intQueue.dequeue());
    }
    GenericPriorityQueue<String> stringQueue = new GenericPriorityQueue<>();
    stringQueue.enqueue("Apple");
    stringQueue.enqueue("Orange");
    stringQueue.enqueue("Banana");
    stringQueue.enqueue("Mango");
    stringQueue.enqueue("Peach");
    System.out.println("\nString Priority Queue:");
    while (!stringQueue.isEmpty()) {
        System.out.println("Dequeue: " + stringQueue.dequeue());
    }
```

```
    }
}
```

```
Output     Generated Files

Integer Priority Queue:
Dequeue: 1
Dequeue: 2
Dequeue: 3
Dequeue: 5
Dequeue: 10

String Priority Queue:
Dequeue: Apple
Dequeue: Banana
Dequeue: Mango
Dequeue: Orange
Dequeue: Peach
```

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 undirected graphs. Demonstrate with a graph of String nodes and another graph of Integer nodes.

A.
```
import java.util.*;
class Graph<T> {
    private Map<T, List<T>> adjList;
    private boolean isDirected;
    public Graph(boolean isDirected) {
        this.isDirected = isDirected;
        this.adjList = new HashMap<>();   }
    public void addNode(T node) {
        adjList.putIfAbsent(node, new ArrayList<>());    }
    public void addEdge(T from, T to) {
        addNode(from);
        addNode(to);
```

```
        adjList.get(from).add(to);
        if (!isDirected) {
            adjList.get(to).add(from);
        }   }
    public void bfs(T start) {
        Set<T> visited = new HashSet<>();
        Queue<T> queue = new LinkedList<>();
        visited.add(start);
        queue.add(start);
        while (!queue.isEmpty()) {
            T node = queue.poll();
            System.out.print(node + " ");
            for (T neighbor : adjList.get(node)) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    queue.add(neighbor);
                }     }      }
        System.out.println();   }
    public void dfs(T start) {
        Set<T> visited = new HashSet<>();
        dfsRecursive(start, visited);
        System.out.println();  }
    private void dfsRecursive(T node, Set<T> visited) {
        visited.add(node);
        System.out.print(node + " ");
        for (T neighbor : adjList.get(node)) {
            if (!visited.contains(neighbor)) {
                dfsRecursive(neighbor, visited);
            }   }   }}
public class GraphDemo {
    public static void main(String[] args) {
```

```
        Graph<String> stringGraph = new Graph<>(false);
        stringGraph.addEdge("A", "B");
        stringGraph.addEdge("A", "C");
        stringGraph.addEdge("B", "D");
        stringGraph.addEdge("C", "D");
        stringGraph.addEdge("D", "E");
        System.out.println("String Graph BFS Traversal from node 'A':");
        stringGraph.bfs("A");
        System.out.println("String Graph DFS Traversal from node 'A':");
        stringGraph.dfs("A");
        Graph<Integer> intGraph = new Graph<>(true);
        intGraph.addEdge(1, 2);
        intGraph.addEdge(1, 3);
        intGraph.addEdge(2, 4);
        intGraph.addEdge(3, 4);
        intGraph.addEdge(4, 5);
        System.out.println("\nInteger Graph BFS Traversal from node 1:");
        intGraph.bfs(1);
        System.out.println("Integer Graph DFS Traversal from node 1:");
        intGraph.dfs(1);
    }
}
```

```
Output     Generated Files

String Graph BFS Traversal from node 'A':
A B C D E
String Graph DFS Traversal from node 'A':
A B D C E

Integer Graph BFS Traversal from node 1:
1 2 3 4 5
Integer Graph DFS Traversal from node 1:
1 2 4 5 3
```

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

A.

```
import java.util.Arrays;
public class Matrix<T extends Number> {
   private T[][] data;
   private int rows;
   private int cols;
   @SuppressWarnings("unchecked")
   public Matrix(int rows, int cols) {
      this.rows = rows;
      this.cols = cols;
      this.data = (T[][]) new Number[rows][cols];
   }
   public void setValue(int row, int col, T value) {
      if (row < 0 || row >= rows || col < 0 || col >= cols) {
         throw new IndexOutOfBoundsException("Invalid index for matrix.");
      }
      data[row][col] = value;
```

```
    }
    public T getValue(int row, int col) {
        if (row < 0 || row >= rows || col < 0 || col >= cols) {
            throw new IndexOutOfBoundsException("Invalid index for matrix.");
        }
        return data[row][col];
    }
    public Matrix<T> add(Matrix<T> other) {
        checkDimensions(other);
        Matrix<T> result = new Matrix<>(rows, cols);
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result.setValue(i, j, (T) Double.valueOf(this.getValue(i, j).doubleValue() + other.getValue(i,
j).doubleValue()));
            } }
        return result;
    }
    public Matrix<T> subtract(Matrix<T> other) {
        checkDimensions(other);
        Matrix<T> result = new Matrix<>(rows, cols);
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result.setValue(i, j, (T) Double.valueOf(this.getValue(i, j).doubleValue() - other.getValue(i,
j).doubleValue()));      }    }
        return result;    }
    public Matrix<T> multiply(Matrix<T> other) {
        if (this.cols != other.rows) {
            throw new IllegalArgumentException("Matrix multiplication not possible: incompatible
dimensions.");    }
        Matrix<T> result = new Matrix<>(this.rows, other.cols);
        for (int i = 0; i < this.rows; i++) {
            for (int j = 0; j < other.cols; j++) {
```

```
            double sum = 0;
            for (int k = 0; k < this.cols; k++) {
                sum += this.getValue(i, k).doubleValue() * other.getValue(k, j).doubleValue();       }
            result.setValue(i, j, (T) Double.valueOf(sum));     }     }
        return result;
    }
    private void checkDimensions(Matrix<T> other) {
        if (this.rows != other.rows || this.cols != other.cols) {
            throw new IllegalArgumentException("Matrix dimensions must match for this operation.");
        }   }
    public String toString() {
        return Arrays.deepToString(data);
    }
    public static void main(String[] args) {
        try {
            Matrix<Integer> intMatrix1 = new Matrix<>(2, 2);
            intMatrix1.setValue(0, 0, 1);
            intMatrix1.setValue(0, 1, 2);
            intMatrix1.setValue(1, 0, 3);
            intMatrix1.setValue(1, 1, 4);
            Matrix<Integer> intMatrix2 = new Matrix<>(2, 2);
            intMatrix2.setValue(0, 0, 5);
            intMatrix2.setValue(0, 1, 6);
            intMatrix2.setValue(1, 0, 7);
            intMatrix2.setValue(1, 1, 8);
            Matrix<Integer> intResultAdd = intMatrix1.add(intMatrix2);
            System.out.println("Integer Matrix Addition:\n" + intResultAdd);
            Matrix<Double> doubleMatrix1 = new Matrix<>(2, 2);
            doubleMatrix1.setValue(0, 0, 1.5);
            doubleMatrix1.setValue(0, 1, 2.5);
            doubleMatrix1.setValue(1, 0, 3.5);
```

```
            doubleMatrix1.setValue(1, 1, 4.5);
            Matrix<Double> doubleMatrix2 = new Matrix<>(2, 2);
            doubleMatrix2.setValue(0, 0, 5.5);
            doubleMatrix2.setValue(0, 1, 6.5);
            doubleMatrix2.setValue(1, 0, 7.5);
            doubleMatrix2.setValue(1, 1, 8.5);

            Matrix<Double> doubleResultMultiply = doubleMatrix1.multiply(doubleMatrix2);
            System.out.println("Double Matrix Multiplication:\n" + doubleResultMultiply);
        } catch (Exception e) {
            System.err.println("An error occurred: " + e.getMessage());
        }
    }
}
```