Assignment – 12

Prakash Manikanta Irrinki
Prakashnaidu9494@gmail.com

Task 1:

Balanced Binary Tree Check:

Write a function to check if a given binary tree is balanced. A balanced tree is one where the height of two subtrees of any node never differs by more than one.

Program:

```
public class TreeNode {
	int element;
	TreeNode left;
	TreeNode rigth;

	public TreeNode(int x) {
		element = x;
	}
}
public class BinaryTree {
	public boolean isBalanced(TreeNode root) {
		return checkHeight(root) != -1;
	}
	private int checkHeight(TreeNode root) {
		if(root==null) {
			return 0;
		}
		int leftheight=checkHeight(root.left);
		if(leftheight == -1) {
			return -1;
		}
		int rightheight=checkHeight(root.rigth);
		if(rightheight == -1) {
			return -1;
		}
		if(Math.abs(leftheight - rightheight)>1) {
			return -1;
		}
		return Math.max(leftheight, rightheight)+1;
	}
	public static void main(String[] args) {
		TreeNode root=new TreeNode(3);
		root.left = new TreeNode(9);
		root.left.left=new TreeNode(5);
		root.rigth=new TreeNode(20);
		root.rigth.left=new TreeNode(15);
		root.rigth.rigth=new TreeNode(7);
		BinaryTree bt= new BinaryTree();
		System.out.println("Is the binary tree balance? "
			+bt.isBalanced(root));
	}
```

```
}
```

Output:

```
Is the binary tree balance? True
```

Task 2:

Trie for Prefix Checking:

Implement a trie data structure in C# that supports insertion of strings and provides a method to check if a given string is a prefix of any word in the trie.

Program:

```
public class TrieNode {
	TrieNode[] child;
	boolean isEndOfWord;
	TrieNode() {
		child = new TrieNode[26];
		isEndOfWord = false;
	}
}
public class TrieDS {

	private TrieNode root;

	public TrieDS() {
		root = new TrieNode();
	}
	public void insert(String word) {
		TrieNode node = root;
		for (char c : word.toCharArray()) {
			int index = c - 'a';
			if (node.child[index] == null) {
				node.child[index] = new TrieNode();
			}
			node = node.child[index];
		}
		node.isEndOfWord = true;
	}
	public boolean search(String word) {
		TrieNode node = root;
		for (char c : word.toCharArray()) {
			int index = c - 'a';
			if (node.child[index] == null) {
				return false;
			}
			node = node.child[index];
		}
		return node != null && node.isEndOfWord;
	}
	public boolean startsWith(String prefix) {
		TrieNode node = root;
		for (char c : prefix.toCharArray()) {
```

```
                int index = c - 'a';
                if (node.child[index] == null) {
                      return false;
                }
                node = node.child[index];
          }
          return true;
     }
     public static void main(String[] args) {
          TrieDS tds = new TrieDS();

          tds.insert("mani");
          tds.insert("teja");
          tds.insert("pavan");

          System.out.println("Is 'Man' a prifix? " +
tds.startsWith("man"));
          System.out.println("Is 'Teja' a prifix? " +
tds.startsWith("te"));
          System.out.println("Is 'vang' a prifix? " +
tds.startsWith("vang"));
     }
}
```

Output:

```
Is 'Man' a prifix? true
Is 'Teja' a prifix? true
Is 'vang' a prifix? False
```

Task 3:

Implementing Heap Operations:

Code a min-heap in C# with methods for insertion, deletion, and fetching the minimum element. Ensure that the heap property is maintained after each operation.

Program:

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

public class MinHeap {
    private List<Integer> heap;
    public MinHeap() {
        heap = new ArrayList<>();
    }
    public void insert(int value) {
        heap.add(value);
        heapifyUp(heap.size() - 1);
    }

    private void heapifyUp(int index) {
        int parentIndex = (index - 1) / 2;
```

```
        while (index > 0 && heap.get(index) < heap.get(parentIndex))
{
            swap(index, parentIndex);
            index = parentIndex;
            parentIndex = (index - 1) / 2;
        }
    }
    public int deleteMin() {
        if (isEmpty()) {
            throw new IllegalStateException("Heap is empty");
        }
        int min = heap.get(0);
        int lastElement = heap.remove(heap.size() - 1);
        if (!isEmpty()) {
            heap.set(0, lastElement);
            heapifyDown(0);
        }
        return min;
    }
    private void heapifyDown(int index) {
        int smallest = index;
        int leftChild = 2 * index + 1;
        int rightChild = 2 * index + 2;

        if (leftChild < heap.size() && heap.get(leftChild) <
heap.get(smallest)) {
            smallest = leftChild;
        }
        if (rightChild < heap.size() && heap.get(rightChild) <
heap.get(smallest)) {
            smallest = rightChild;
        }
        if (smallest != index) {
            swap(index, smallest);
            heapifyDown(smallest);
        }
    }
    public int getMin() {
        if (isEmpty()) {
            throw new IllegalStateException("Heap is empty");
        }
        return heap.get(0);
    }
    public boolean isEmpty() {
        return heap.isEmpty();
    }
    private void swap(int i, int j) {
        int temp = heap.get(i);
        heap.set(i, heap.get(j));
        heap.set(j, temp);
    }
```

```
    public static void main(String[] args) {
        MinHeap minHeap = new MinHeap();
        minHeap.insert(3);
        minHeap.insert(2);
        minHeap.insert(1);
        System.out.println("Minimum element: " + minHeap.getMin());
        minHeap.deleteMin();
        System.out.println("Minimum element after deletion: "
           + minHeap.getMin());
    }
}
```

Output:

```
Minimum element: 1
Minimum element after deletion: 2
```

Task 4:

Graph Edge Addition Validation:

Given a directed graph, write a function that adds an edge between two nodes and then checks if the graph still has no cycles. If a cycle is created, the edge should not be added.

Program:

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

public class Graph {
    private int V; // Number of vertices
    private List<List<Integer>> adj;
    public Graph(int V) {
        this.V = V;
        adj = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }
    }
    public void addEdge(int u, int v) {
        adj.get(u).add(v);
    }

    public boolean hasCycle() {
        boolean[] visited = new boolean[V];
        boolean[] recursionStack = new boolean[V];

        for (int i = 0; i < V; i++) {
            if (isCyclicUtil(i, visited, recursionStack)) {
                return true;
            }
        }
        return false;
    }
```

```
    private boolean isCyclicUtil(int v, boolean[] visited, boolean[]
                                 recursionStack) {
        if (recursionStack[v]) {
            return true;.
        }

        if (visited[v]) {
            return false;        }
        visited[v] = true;
        recursionStack[v] = true;

        List<Integer> neighbors = adj.get(v);
        for (Integer neighbor : neighbors) {
            if (isCyclicUtil(neighbor, visited, recursionStack)) {
                return true;
            }
        }
        recursionStack[v] = false; // Backtrack
        return false;
    }
    public boolean addEdgeAndCheckForCycle(int u, int v) {
        addEdge(u, v);
        if (hasCycle()) {
            adj.get(u).remove((Integer) v);
            return false;
        }
        return true;
    }
    public static void main(String[] args) {
        Graph graph = new Graph(4);
        graph.addEdge(0, 1);
        graph.addEdge(1, 2);
        graph.addEdge(2, 3);

        System.out.println("Adding edge from 3 to 0: "
                           + graph.addEdgeAndCheckForCycle(3, 0));
        System.out.println("Adding edge from 3 to 1: "
                           + graph.addEdgeAndCheckForCycle(3, 1));
    }
}
```

Output:

```
Adding edge from 3 to 0: false
Adding edge from 3 to 1: false
```

Task 5:

Breadth-First Search (BFS) Implementation:

For a given undirected graph, implement BFS to traverse the graph starting from a given node and print each node in the order it is visited.

Program:

```
public class BFS {
    public static void bfsTraversal(Map<Integer, List<Integer>>
graph, int startNode) {
        Set<Integer> visited = new HashSet<>();
        Queue<Integer> queue = new LinkedList<>();
        visited.add(startNode);
        queue.add(startNode);
        while (!queue.isEmpty()) {
            int currentNode = queue.poll();
            System.out.print(currentNode + " ");
            List<Integer> neighbors = graph.get(currentNode);
            if (neighbors != null) {
                for (int neighbor : neighbors) {
                       if (!visited.contains(neighbor)) {
                        visited.add(neighbor);
                        queue.add(neighbor);
                    }
                }
            }
        }
    }

    public static void main(String[] args) {
        Map<Integer, List<Integer>> graph = new HashMap<>();

        graph.put(0, Arrays.asList(1, 2));
        graph.put(1, Arrays.asList(0, 3, 4));
        graph.put(2, Arrays.asList(0, 5, 6));
        graph.put(3, Arrays.asList(1));
        graph.put(4, Arrays.asList(1));
        graph.put(5, Arrays.asList(2));
        graph.put(6, Arrays.asList(2));

        int startNode = 0;
        System.out.println("BFS traversal starting from node "
                                + startNode + ":");
        bfsTraversal(graph, startNode);
    }
}
```

Output:

```
BFS traversal starting from node 0:
0 1 2 3 4 5 6
```

Task 6:

Depth-First Search (DFS) Recursive

Write a recursive DFS function for a given undirected graph. The function should visit every node and print it out.

Program:

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
public class DepthFirstSearch {
    public void dfs(int node, boolean[] visited, Map<Integer,
List<Integer>> graph) {
        visited[node] = true;
        System.out.println(node);
        for (int neighbor : graph.get(node)) {
            if (!visited[neighbor]) {
                dfs(neighbor, visited, graph);
            }
        }
    }
    public void addEdge(Map<Integer, List<Integer>> graph, int u,
int v) {
        if (!graph.containsKey(u)) {
            graph.put(u, new ArrayList<>());
        }
        if (!graph.containsKey(v)) {
            graph.put(v, new ArrayList<>());
        }
        graph.get(u).add(v);
        graph.get(v).add(u); // Since the graph is undirected
    }
    public static void main(String[] args) {
        DepthFirstSearch dfs = new DepthFirstSearch();
        Map<Integer, List<Integer>> graph = new HashMap<>();
        dfs.addEdge(graph, 0, 1);
        dfs.addEdge(graph, 0, 2);
        dfs.addEdge(graph, 1, 2);
        dfs.addEdge(graph, 2, 0);
        dfs.addEdge(graph, 2, 3);
        dfs.addEdge(graph, 3, 3);
        boolean[] visited = new boolean[graph.size()];
        dfs.dfs(2, visited,graph);
    }
}
```

Output:

2
0
1
3