Day 7 and 8:

Task 1: Balanced Binary Tree Check

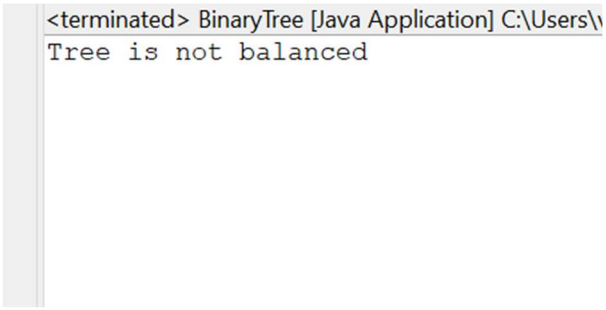
Write a function to check if a given binary tree is balanced. A balanced tree is one where the height of two subtrees of any node never differs by more than one.

SOLUTION :

```
package com.dsassignment_day_7_8;

class Node {

int data;

Node left, right;

Node(int item) {

data = item;

left = right = null;

}

}

public class BinaryTree {

class Height {

int height = 0;

}

boolean isBalanced(Node node, Height height) {

if (node == null)

return true;

Height lh = new Height(), rh = new Height();

if (!isBalanced(node.left, lh))

return false;

if (!isBalanced(node.right, rh))

return false;

height.height = Math.max(lh.height, rh.height) + 1;

if (Math.abs(lh.height - rh.height) <= 1)
```

```
return true;

return false;

}

public static void main(String args[]) {

BinaryTree tree = new BinaryTree();

Node root = new Node(1);

root.left = new Node(2);

root.right = new Node(3);

root.left.left = new Node(4);

root.left.right = new Node(5);

root.left.left.left = new Node(8);

BinaryTree.Height height = tree.new Height();

if (tree.isBalanced(root, height))

System.out.println("Tree is balanced");

else

System.out.println("Tree is not balanced");

}

}
```

OUTPUT :



Task 2: Trie for Prefix Checking (Yet to be completed in Class)

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

Task 3: Implementing Heap Operations

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

SOLUTION :

```
package com.dsassignment_day_7_8;

import java.util.ArrayList;

public class MinHeap {

private ArrayList<Integer> heap;

public MinHeap() {

heap = new ArrayList<>();

}

public void insert(int val) {

heap.add(val);

int index = heap.size() - 1;

while (index > 0) {

int parent = (index - 1) / 2;

if (heap.get(parent) <= heap.get(index)) {

break;

}

int temp = heap.get(parent);

heap.set(parent, heap.get(index));
```

```
heap.set(index, temp);
index = parent;
}
}
public int delete() {
if (heap.size() == 0) {
throw new IllegalStateException();
}
int removedVal = heap.get(0);
int lastVal = heap.remove(heap.size() - 1);
if (heap.size() > 0) {
heap.set(0, lastVal);
siftDown();
}
return removedVal;
}
public int getMin() {
if (heap.size() == 0) {
throw new IllegalStateException();
}
return heap.get(0);
}
public void printHeap() {
System.out.println(heap);
}
private void siftDown() {
int index = 0;
int leftChild = 2 * index + 1;
```

```
while (leftChild < heap.size()) {
int minIndex = leftChild;
int rightChild = leftChild + 1;
if (rightChild < heap.size()) {
if (heap.get(rightChild) < heap.get(leftChild)) {
minIndex = rightChild;
}
}
if (heap.get(index) <= heap.get(minIndex)) {
break;
}
int temp = heap.get(index);
heap.set(index, heap.get(minIndex));
heap.set(minIndex, temp);
index = minIndex;
leftChild = 2 * index + 1;
}
}
public static void main(String[] args) {
MinHeap minHeap = new MinHeap();
minHeap.insert(3);
minHeap.insert(2);
minHeap.insert(1);
System.out.println("Heap: ");
minHeap.printHeap(); // Outputs: [1, 3, 2]
System.out.println("Minimum: " + minHeap.getMin()); // Outputs: 1
System.out.println("Deleted: " + minHeap.delete()); // Outputs: 1
System.out.println("Heap: ");
```

```
minHeap.printHeap(); // Outputs: [2, 3]

System.out.println("Minimum: " + minHeap.getMin()); // Outputs: 2

}

}
```

OUTPUT :

```
Heap:
[1, 3, 2]
Minimum: 1
Deleted: 1
Heap:
[2, 3]
Minimum: 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.

SOLUTION :

```
package com.dsassignment_day_7_8;

import java.util.*;

public class DirectedGraph {

private int V;

private LinkedList<Integer> adj[];

public DirectedGraph(int v) {

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList();

}
```

```
public void addEdge(int v, int w) {

adj[v].add(w);

}

public boolean isCyclic() {

boolean[] visited = new boolean[V];

boolean[] recStack = new boolean[V];

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

if (isCyclicUtil(i, visited, recStack))

return true;

}

return false;

}

private boolean isCyclicUtil(int v, boolean[] visited, boolean[] recStack)
{

if (!visited[v]) {

visited[v] = true;

recStack[v] = true;

Iterator<Integer> it = adj[v].iterator();

while (it.hasNext()) {

int n = it.next();

if (!visited[n] && isCyclicUtil(n, visited, recStack))

return true;

else if (recStack[n])

return true;

}

}

recStack[v] = false;

return false;
```

```
}
public static void main(String args[]) {
DirectedGraph g = new DirectedGraph(4);
g.addEdge(0, 1);
g.addEdge(1, 2);
g.addEdge(2, 0);
int u = 1;
int v = 3;
System.out.println("Adding edge from " + u + " to " + v);
g.addEdge(u, v);
if (g.isCyclic())
System.out.println("Cycle detected! Edge not added.");
else {
System.out.println("Edge added successfully.");
System.out.println("Adjacency list after adding edge:");
for (int i = 0; i < g.V; ++i) {
System.out.print(i + " -> ");
for (Integer j : g.adj[i])
System.out.print(j + " ");
System.out.println();
}
}
}
}
```

OUTPUT :

```
Adding edge from 1 to 3
Cycle detected! Edge not added.
```

Task 5: Breadth-First Search (BFS) Implementation

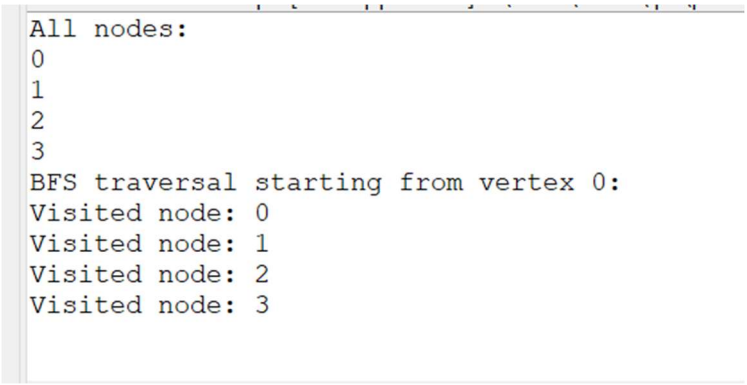
For a given undirected graph, implement BFS to traverse the graph starting from a given node and print each node in the order it is visited.

```
package com.dsassignment_day_7_8;

import java.util.*;

public class BFSGraph {

private int V;

private LinkedList<Integer> adj[];

public BFSGraph(int v) {

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList();

}
```

```
public void addEdge(int v, int w) {
adj[v].add(w);
adj[w].add(v);
}
public void BFS(int s) {
boolean visited[] = new boolean[V];
LinkedList<Integer> queue = new LinkedList<>();
visited[s] = true;
queue.add(s);
System.out.println("BFS traversal starting from vertex " + s + ":");
while (queue.size() != 0) {
s = queue.poll();
System.out.println("Visited node: " + s);
Iterator<Integer> i = adj[s].listIterator();
while (i.hasNext()) {
int n = i.next();
if (!visited[n]) {
visited[n] = true;
queue.add(n);
}
}
}
}
public static void main(String args[]) {
BFSGraph g = new BFSGraph(4);
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
```

```
g.addEdge(2, 3);

System.out.println("All nodes:");

for (int i = 0; i < 4; i++) {

System.out.println(i);

}

g.BFS(0);

}

}
```

Output :



```
All nodes:
0
1
2
3
BFS traversal starting from vertex 0:
Visited node: 0
Visited node: 1
Visited node: 2
Visited node: 3
```

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.

SOLUTION :

```
package com.dsassignment_day_7_8;
```

```
import java.util.*;

public class UndirectedGraph {

private int V;

private LinkedList<Integer> adj[];

public UndirectedGraph(int v) {

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList();

}

public void addEdge(int v, int w) {

adj[v].add(w);

adj[w].add(v);

}

private void DFSUtil(int v, boolean visited[]) {

visited[v] = true;

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

Iterator<Integer> i = adj[v].listIterator();

while (i.hasNext()) {

int n = i.next();

if (!visited[n])

DFSUtil(n, visited);

}

}

public void DFS(int v) {

boolean visited[] = new boolean[V];

DFSUtil(v, visited);

}
```

```
public static void main(String args[]) {

UndirectedGraph g = new UndirectedGraph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 3);

System.out.println("DFS traversal starting from vertex 0:");

g.DFS(0);

}

}
```

OUTPUT: