Day 7 and 8:

Task 1: Balanced Binary Tree Check

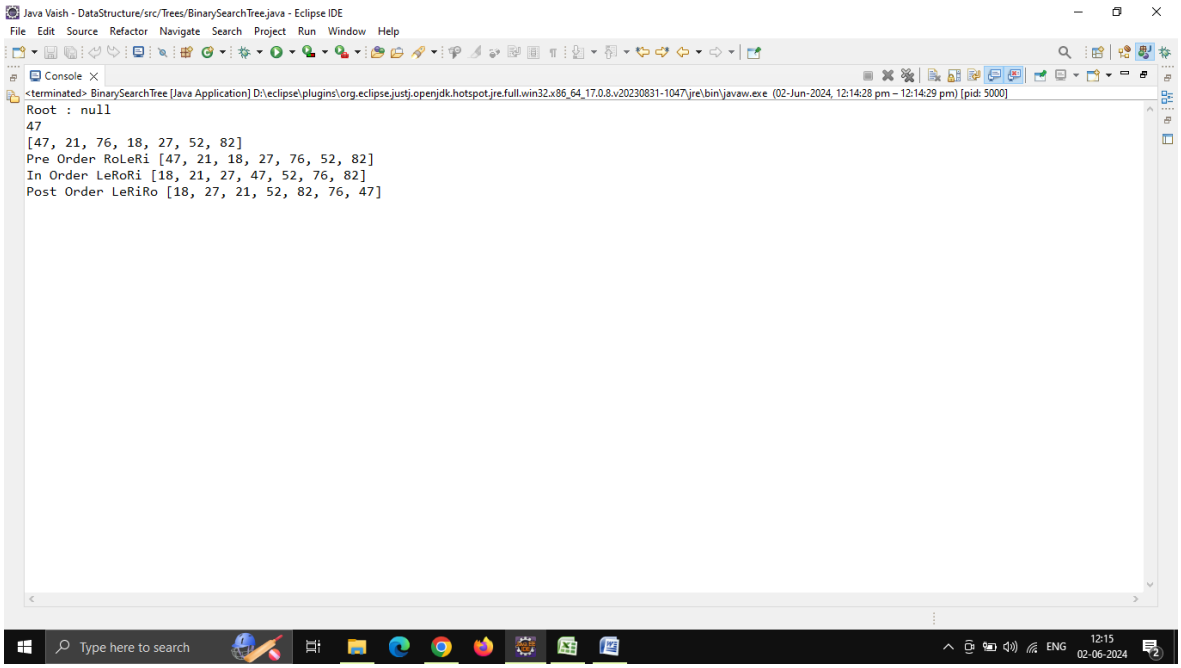
Write a function to check if a given binary tree is balanced. A balanced tree is one where the height of two subtrees of any node never differs by more than one.

```
package Trees;


import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class BinarySearchTree {

	public Node root;

	public static class Node {
		int value;
		Node left;
		Node right;

		public Node(int value) {
			super();
			this.value = value;
		}
	}

	public boolean insert(int value) {
		Node newNode = new Node(value);
		if (root == null) {
			root = newNode;
			return true;
		}
		Node temp = root;
		while (true) {
			if (newNode.value == temp.value)
				return false;
			if (newNode.value < temp.value) {
				if (temp.left == null) {
					temp.left = newNode;
					return true;
				}
				temp = temp.left;
			} else {
				if (temp.right == null) {
```

```
                        temp.right = newNode;
                        return true;
                    }
                    temp = temp.right;
                }
            }
        }

    public List<Integer> BFS() {

        Node currentNode = root;
        Queue<Node> queue = new LinkedList<>();
        List<Integer> results = new ArrayList();
        queue.add(currentNode);

        while (queue.size() > 0) {
            currentNode = queue.remove();
            results.add(currentNode.value);
            if (currentNode.left != null) {
                queue.add(currentNode.left);
            }
            if (currentNode.right != null) {
                queue.add(currentNode.right);
            }
        }

        return results;
    }

    public List<Integer> DFSPreOrder(){
        List<Integer> results = new ArrayList<>();
        class Traverse{
            public Traverse(Node currentNode){
                results.add(currentNode.value);
                if(currentNode.left != null) {
                    new Traverse(currentNode.left);
                }
                if(currentNode.right != null) {
                    new Traverse(currentNode.right);
                }

            }
        }
        new Traverse(root);
        return results;
    }
    public List<Integer> DFSInOrder(){
        List<Integer> results = new ArrayList<>();
        class Traverse{
            public Traverse(Node currentNode){
```

```
                        if(currentNode.left != null) {
                                new Traverse(currentNode.left);
                        }
                        results.add(currentNode.value);
                        if(currentNode.right != null) {
                                new Traverse(currentNode.right);
                        }

                }
        }
        new Traverse(root);
        return results;
}
public List<Integer> DFSPostOrder(){
        List<Integer> results = new ArrayList<>();
        class Traverse{
                public Traverse(Node currentNode){

                        if(currentNode.left != null) {
                                new Traverse(currentNode.left);
                        }
                        if(currentNode.right != null) {
                                new Traverse(currentNode.right);
                        }
                        results.add(currentNode.value);

                }
        }
        new Traverse(root);
        return results;
}

public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        System.out.println("Root : " + bst.root);

        bst.insert(47);
        bst.insert(21);
        bst.insert(76);
        bst.insert(18);
        bst.insert(27);
        bst.insert(52);
        bst.insert(82);

        System.out.println(bst.root.value);

        System.out.println(bst.BFS());

        System.out.println("Pre Order RoLeRi " + bst.DFSPreOrder());
        System.out.println("In Order LeRoRi " + bst.DFSInOrder());
        System.out.println("Post Order LeRiRo " +bst.DFSPostOrder());
```

```
    }
}
```

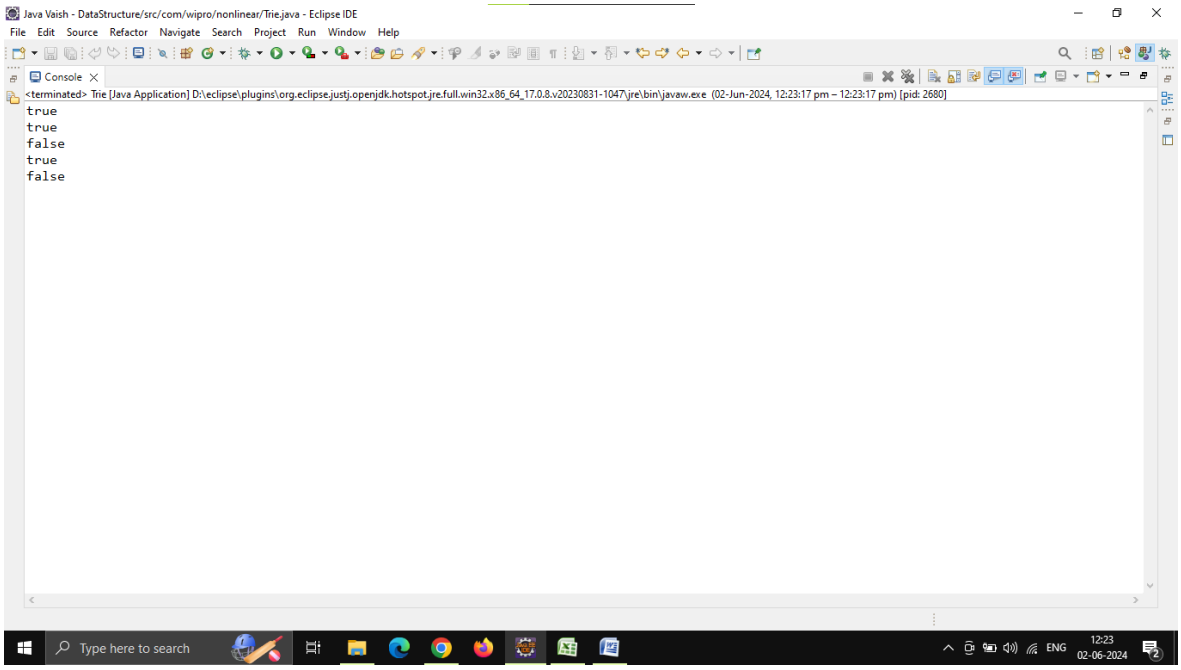

Java Vaish - DataStructure/src/Trees/BinarySearchTree.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Console
<terminated> BinarySearchTree [Java Application] D:\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.8.v20230831-1047\jre\bin\javaw.exe (02-Jun-2024, 12:14:28 pm – 12:14:29 pm) [pid: 5000]
Root : null
47
[47, 21, 76, 18, 27, 52, 82]
Pre Order RoLeRi [47, 21, 18, 27, 76, 52, 82]
In Order LeRoRi [18, 21, 27, 47, 52, 76, 82]
Post Order LeRiRo [18, 27, 21, 52, 82, 76, 47]
Type here to search
ENG
12:15
02-06-2024

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.

```
package com.assig.nonlinear;


import java.util.HashMap; import
java.util.Map;


class TrieNode {
    Map<Character, TrieNode> children;
    boolean isEndOfWord;


    public TrieNode() {
        children = new HashMap<>();
        isEndOfWord = false;
    }
}


public class Trie {
    private final TrieNode root;


    public Trie() {
        root = new TrieNode();
    }


    public void insert(String word) { TrieNode
        current = root;
        for (char c : word.toCharArray()) {
            current.children.putIfAbsent(c, new TrieNode()); current =
```

```
            current.children.get(c);
        }
        current.isEndOfWord = true;
    }

    public boolean isPrefix(String prefix) {
        TrieNode current = root;
        for (char c : prefix.toCharArray()) {
            if (!current.children.containsKey(c)) {
                return false;
            }
            current = current.children.get(c);
        }
        return true;
    }

    public static void main(String[] args) { Trie
        trie = new Trie(); trie.insert("apple");
        trie.insert("app"); trie.insert("application");
        trie.insert("banana");

        System.out.println(trie.isPrefix("app"));
        System.out.println(trie.isPrefix("ban"));
        System.out.println(trie.isPrefix("bat"));
        System.out.println(trie.isPrefix("appl"));
        System.out.println(trie.isPrefix("apx"));
    }
}
```



Task 3: Implementing Heap Operations

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

```
package com.ds.tree;


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


public class Heap {
	private List<Integer>heap;
	public Heap()
	{
		this.heap=new ArrayList<>();


	}


	public List<Integer> getheap()
```

```
{
        return new ArrayList<Integer>(heap);
}


public int lefrchild(int index)
{
        return (index*2)+2;

}

public int rightchild(int index)
{
        return (index*2)+2;

}
```

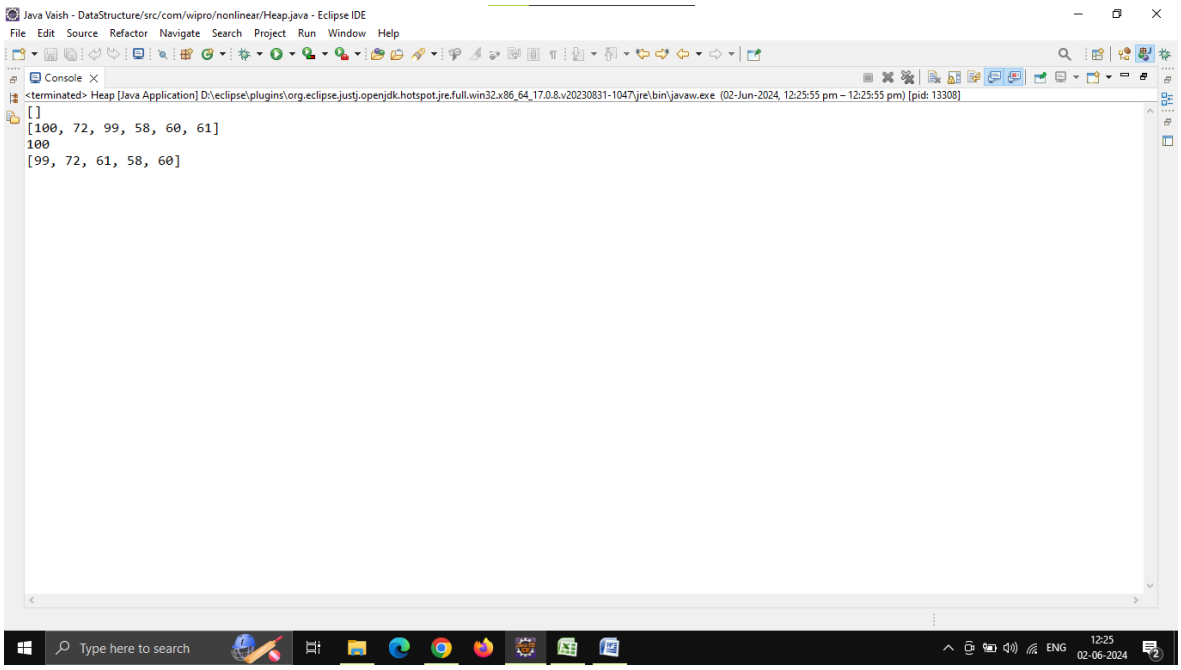
```
public int parent(int index)
{
	return (index-1)/2;

}

public void insert(int value)
{
	heap.add(value);
	int current=heap.size()-1;
	while(current > 0&&
heap.get(current)>heap.get(parent(current)))
	{
		swap(current,parent(current)); current=parent(current);
	}
}
private void swap(int index1, int index2) {
	// TODO Auto-generated method stub

	int temp=heap.get(index1); heap.set(index1,
	heap.get(index2)); heap.set(index2, temp);
```

```
}


public Integer remove()
{

        if(heap.size()==0)
        {
                return null;

        }
        if(heap.size()==1)
        {
                return heap.remove(0);

        }

        int maxvalue=heap.get(0);
        heap.set(0, heap.remove(heap.size()-1));
        sinkDown(0);
        return maxvalue;
```

```
}

private void sinkDown(int index) { int

        maxindex=index;
        int leftindex=lefrchild(index); int
        rightindex=rightchild(index);

if(leftindex<heap.size()&&heap.get(leftindex)>heap.get(maxind
ex))
        {
                maxindex=leftindex;

        }

if(rightindex<heap.size()&&heap.get(rightindex)>heap.get(maxi ndex))
        {
                maxindex=rightindex;

        }
```

```
            if(maxindex!=index)
            {
                swap(index, maxindex);
                index=maxindex;
            }
            // TODO Auto-generated method stub

        }


        public List<Integer> heapSort() {
//          List<Integer> sortedList = new ArrayList<>();
//          while (!heap.isEmpty()) {
//              sortedList.add(remove());
//          }

            Collections.sort(heap); return
            heap;
        }

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

            h=new Heap();
```

```
            System.out.println(h.getheap()); h.insert(99);
        h.insert(66);;
        h.insert(34);
        h.insert(44);
        h.insert(50);

        System.out.println(h.getheap());

        System.out.println("Removed Element is :-          "+h.remove());

        System.out.println(h.getheap()); System.out.println( "sorted
array"+h.heapSort());
        }

}
```



Task 4: Graph Edge Addition Validation

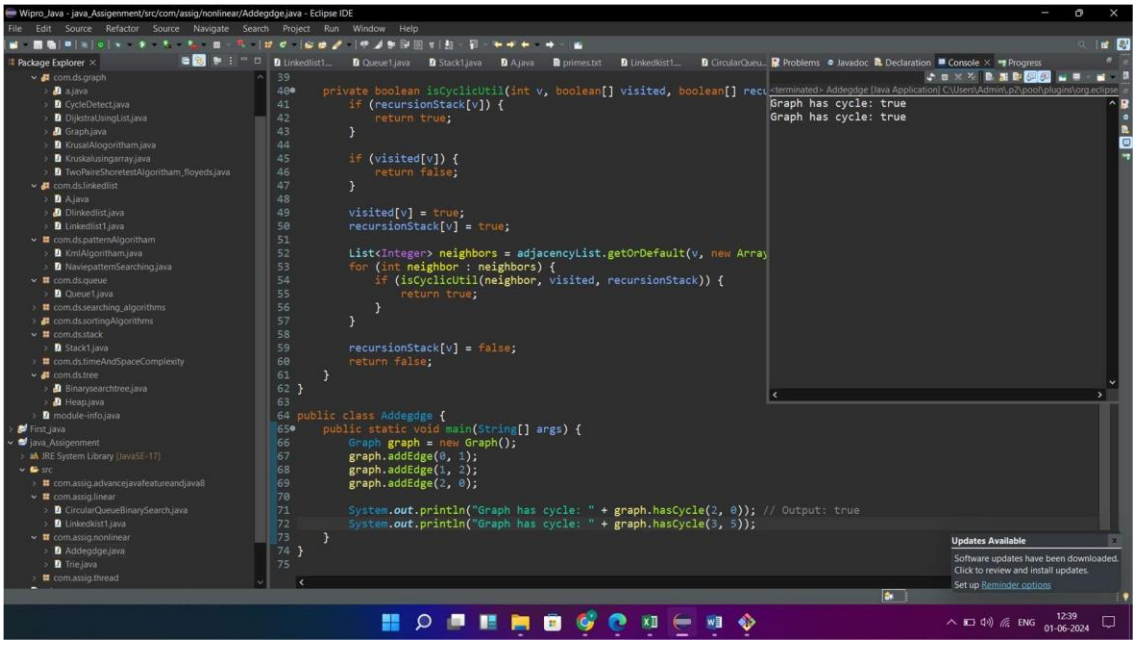
Given a directed graph, write a function that adds an edge between two nodes and then checks if the graph still has no cycles. If a cycle is created, the edge should not be added.

```
package com.assig.nonlinear;


import java.util.ArrayList;
import java.util.HashMap;
import java.util.List; import
java.util.Map;


class Graph {
    private final Map<Integer, List<Integer>> adjacencyList; public

    Graph() {

        adjacencyList = new HashMap<>();
    }


    public void addEdge(int from, int to) {
        if (!adjacencyList.containsKey(from)) { adjacencyList.put(from,
            new ArrayList<>());
        }
        adjacencyList.get(from).add(to);
    }


    public boolean hasCycle(int from, int to) { addEdge(from, to); //
        Add the edge temporarily
        boolean[] visited = new boolean[adjacencyList.size() + 1];
        boolean[] recursionStack = new boolean[adjacencyList.size() +
1];
```

```
for (int i : adjacencyList.keySet()) {
    if (!visited[i] && isCyclicUtil(i, visited, recursionStack)) {
        // Remove the temporarily added edge
        adjacencyList.get(from).remove(Integer.valueOf(to)); return
        true;
    }
}
```

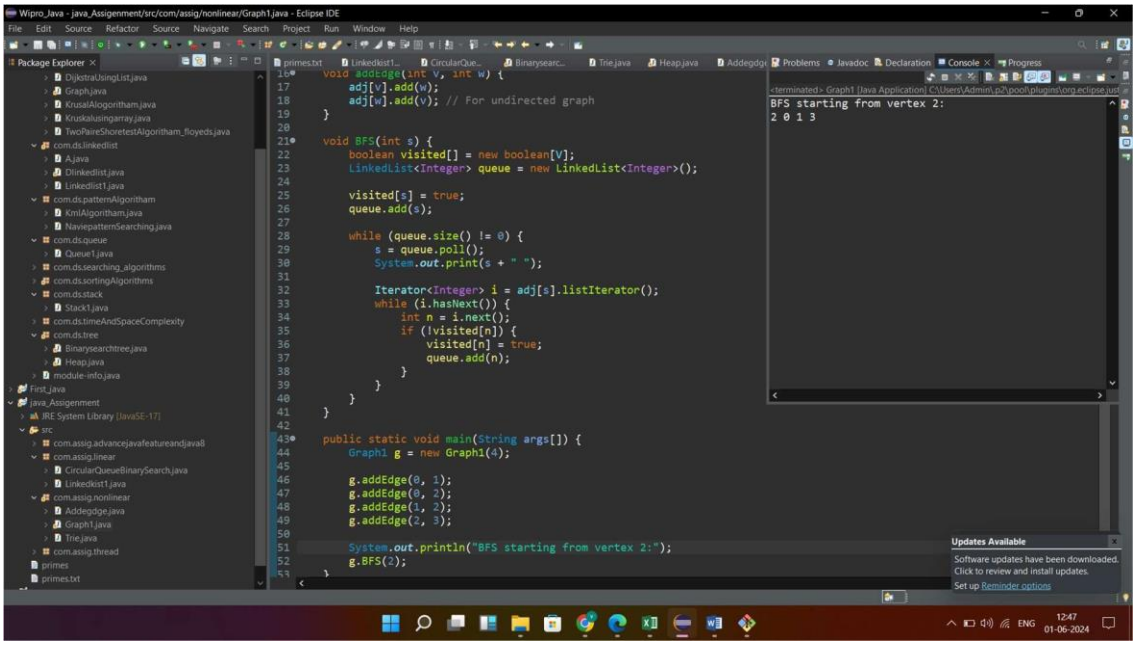
```
    // Remove the temporarily added edge
    adjacencyList.get(from).remove(Integer.valueOf(to)); return false;
}

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 = adjacencyList.getOrDefault(v, new ArrayList<>());
    for (int neighbor : neighbors) {
        if (isCyclicUtil(neighbor, visited, recursionStack)) { return true;
        }
```

```
        }

        recursionStack[v] = false; return
        false;
    }
}

public class Addegdge {
    public static void main(String[] args) { Graph
        graph = new Graph(); graph.addEdge(0,
        1);
        graph.addEdge(1, 2);
        graph.addEdge(2, 0);

        System.out.println("Graph has cycle: " + graph.hasCycle(2, 0)); // Output:
true
        System.out.println("Graph has cycle: " + graph.hasCycle(3, 5));
    }
}
```



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.



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.

```
import java.util.*;



public class Graph {

   private int V; // Number of vertices

   private LinkedList<Integer>[] adj; //

Adjacency list



   public Graph(int V) {
```

```
        this.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); // For undirected
graph
    }

    public void DFSUtil(int v, boolean[]
visited) {
        // Mark the current node as
visited and print it
        visited[v] = true;
        System.out.print(v + " ");

        // Recur for all the vertices
adjacent to this vertex
        for (int i : adj[v]) {
```

```
            if (!visited[i])
                DFSUtil(i, visited);
        }
    }

    public void DFS() {
        // Mark all the vertices as not
visited (default value is false)
        boolean[] visited = new
boolean[V];
        for (int i = 0; i < V; ++i) {
            if (!visited[i]) {
                DFSUtil(i, visited);
            }
        }
    }

    public static void main(String[] args)
{
        Graph graph = new Graph(5); //
Create a graph with 5 vertices
        graph.addEdge(0, 1);
```

```
        graph.addEdge(0, 2);

        graph.addEdge(1, 3);

        graph.addEdge(2, 4);



        System.out.println("Depth-First
Search traversal:");

        graph.DFS(); // Output: 0 1 3 2 4

    }

}
```