

Day 7 and 8: Manisha Assignment

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.

Explanation

1. TreeNode Class:

- A simple class to represent each node in the binary tree. Each node has a value ('val'), a left child ('left'), and a right child ('right').

2. BalancedBinaryTree Class:

- Contains the main logic to check if the binary tree is balanced.

3. checkHeight Method:

- This is a helper method that recursively checks the height of the tree.
 - If a subtree is unbalanced, it returns '-1'.
 - Otherwise, it returns the height of the tree.

4. isBalanced Method:

- This method uses 'checkHeight' to determine if the tree is balanced.
 - It returns 'true' if the tree is balanced and 'false' otherwise.

5. main Method:

- Constructs a sample balanced binary tree.
- Calls the 'isBalanced' method to check if the tree is balanced.
- Prints the result.

The screenshot shows four separate code editor windows in the Eclipse IDE, each containing a different part of the Java code for a balanced binary tree. The code is as follows:

```
1 package wipro;
2 class TreeNode {
3     int val;
4     TreeNode left;
5     TreeNode right;
6 }
7
8 TreeNode(int val) {
9     this.val = val;
10    left = right = null;
11 }
12
13 public class BalancedBinaryTree {
14
15     // Helper function to check the height of a tree
16     private int checkHeight(TreeNode root) {
17         if (root == null) {
18             return 0;
19         }
20
21         int leftHeight = checkHeight(root.left);
22         int rightHeight = checkHeight(root.right);
23
24         if (leftHeight == -1 || rightHeight == -1 || Math.abs(leftHeight - rightHeight) >= 2) {
25             return -1;
26         }
27
28         return Math.max(leftHeight, rightHeight) + 1;
29     }
30
31     // Function to check if the tree is balanced
32     public boolean isBalanced(TreeNode root) {
33         return checkHeight(root) != -1;
34     }
35
36     public static void main(String[] args) {
37         // Creating a sample balanced binary tree
38         TreeNode root = new TreeNode(100);
39         root.left = new TreeNode(98);
40         root.right = new TreeNode(78);
41         root.left.left = new TreeNode(89);
42         root.left.right = new TreeNode(89);
43         root.left.right.left = new TreeNode(100);
44         root.left.right.right = new TreeNode(99);
45
46         BalancedBinaryTree tree = new BalancedBinaryTree();
47         boolean result = tree.isBalanced(root);
48
49         System.out.println("Is the tree balanced? " + result);
50     }
51 }
```

Each code editor window has its own set of toolbars, menus, and status bars, typical of the Eclipse IDE environment.

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.

Explanation

1. TrieNode Class:

- Represents each node in the trie.
 - Contains an array of `TrieNode` references for each letter in the alphabet (26 for lowercase English letters).
 - A boolean `isEndOfWord` to mark the end of a word.

2. Trie Class:

- Contains the root node and methods to insert words and check for prefixes.

3. insert Method:

- Inserts a word into the trie by iterating through each character, creating new nodes as needed, and marking the end of the word.

4. isPrefix Method:

- Checks if a given string is a prefix by traversing the trie nodes according to the characters in the prefix.
 - Returns 'true' if all characters in the prefix are found in the trie, otherwise 'false'.

5. main Method:

- Demonstrates how to use the `Trie` class.
 - Inserts several words into the trie.
 - Tests the `isPrefix` method with different prefixes and prints the results.

```
1 class TrieNode {  
2     TrieNode[] children;  
3     boolean isEndOfWord;  
4 }  
5  
6 public class TrieNode {  
7     children = new TrieNode[26]; // Assuming only lowercase English letters  
8     isEndOfWord = false;  
9 }  
10  
11 public class Trie {  
12     private TrieNode root;  
13 }  
14  
15 public Trie() {  
16     root = new TrieNode();  
17 }  
18  
19 // Insert a word into the trie  
20 public void insert(String word) {  
21  
22     TrieNode node = root;  
23     for (char c : word.toCharArray()) {  
24         int index = c - 'a';  
25         if (node.children[index] == null) {  
26             node.children[index] = new TrieNode();  
27         }  
28         node = node.children[index];  
29     }  
30     node.isEndOfWord = true;  
31 }  
32  
33 // Check if a given prefix is in the trie  
34 public boolean isPrefix(String prefix) {  
35     TrieNode node = root;  
36     for (char c : prefix.toCharArray()) {  
37         int index = c - 'a';  
38         if (node.children[index] == null) {  
39             return false;  
40         }  
41         node = node.children[index];  
42     }  
43     return true;  
44 }  
45  
46 public static void main(String[] args) {  
47     Trie trie = new Trie();  
48  
49     // Insert words into the trie  
50     trie.insert("apple");  
51     trie.insert("app");  
52     trie.insert("banana");  
53     trie.insert("band");  
54     trie.insert("bandana");  
55  
56     // Test the isPrefix method  
57     System.out.println("Is 'app' a prefix? " + trie.isPrefix("app")); // true  
58     System.out.println("Is 'ban' a prefix? " + trie.isPrefix("ban")); // true  
59     System.out.println("Is 'band' a prefix? " + trie.isPrefix("band")); // true  
60     System.out.println("Is 'banda' a prefix? " + trie.isPrefix("banda")); // true  
61 }  
62 }
```

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.

Explanation

1. MinHeap Class:

- **Constructor:** Initializes an empty ArrayList to store heap elements.

- parent, leftChild, rightChild: Helper methods to calculate parent and child indices.

- **insert:** Adds a new value to the heap and maintains the heap property by comparing and swapping values up the tree.

- **deleteMin:** Removes and returns the minimum element (root) from the heap. Replaces the root with the last element and heapifies down to maintain the heap property.

- **getMin:** Returns the minimum element without removing it.

- **heapify:** Ensures the heap property is maintained by comparing parent and child nodes and swapping if necessary.

- **display:** Prints the elements of the heap.

2. Main Method:

- Demonstrates inserting elements into the heap, fetching the minimum element, and deleting the minimum element.

- Displays the heap after each operation.

The screenshots show the Eclipse IDE interface with three open code editors. The top editor contains the `MinHeap` class definition, the middle editor shows the `Main` class demonstrating insertions, and the bottom editor shows the `Main` class demonstrating deletions. The code is as follows:

```
MinHeap.java:
```

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

class MinHeap {
    private ArrayList<Integer> heap;
    public MinHeap() {
        heap = new ArrayList<>();
    }
    // Get the index of the parent of the node at index i
    private int parent(int i) {
        return (i - 1) / 2;
    }
    // Get the index of the left child of the node at index i
    private int leftChild(int i) {
        return 2 * i + 1;
    }
    // Get the index of the right child of the node at index i
    private int rightChild(int i) {
        return 2 * i + 2;
    }
    // Insert a new value into the heap
    public void insert(int value) {
        int index = heap.size();
        while (index >= 0 && heap.get(parent(index)) > heap.get(index)) {
            int temp = heap.get(index);
            heap.set(index, heap.get(parent(index)));
            heap.set(parent(index), temp);
            index = parent(index);
        }
    }
    // Remove and return the minimum element from the heap
    public int deleteMin() {
        if (heap.isEmpty()) {
            throw new IllegalStateException("Heap is empty");
        }
        int root = heap.get(0);
        int lastValue = heap.remove(heap.size() - 1);
        if (!heap.isEmpty()) {
            heap.set(0, lastValue);
            heapify(0);
        }
        return root;
    }
    // Print the heap
    public void display() {
        System.out.println("Heap after insertions:");
        minHeap.display();
        System.out.println("Minimum element: " + minHeap.getMin());
        System.out.println("Deleting minimum element: " + minHeap.deleteMin());
        System.out.println("Heap after deletion:");
        minHeap.display();
    }
}
```

```
Main.java:
```

```
public class Main {
    public static void main(String[] args) {
        MinHeap minHeap = new MinHeap();
        minHeap.insert(2);
        minHeap.insert(3);
        minHeap.insert(5);
        minHeap.insert(8);
        minHeap.insert(10);
        minHeap.insert(12);
        minHeap.insert(15);
        minHeap.insert(18);
        minHeap.insert(20);
        System.out.println("Heap after insertions:");
        minHeap.display();
        System.out.println("Minimum element: " + minHeap.getMin());
        System.out.println("Deleting minimum element: " + minHeap.deleteMin());
        System.out.println("Heap after deletion:");
        minHeap.display();
    }
}
```

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.

Explanation

1. DirectedGraph Class:

- adjList: A map representing the adjacency list of the graph.
- addNode: Adds a node to the adjacency list if it doesn't already exist.
- addEdge: Adds an edge between two nodes if it doesn't create a cycle. It first checks for potential cycles using the `createsCycle` method.
- createsCycle: Uses depth-first search (DFS) to check if adding an edge creates a cycle.
- dfs: A helper method for performing DFS. It checks if a cycle is detected by maintaining a recursion stack ('recStack').

2. Main Method:

- Creates an instance of 'DirectedGraph'.
- Adds edges between nodes.
- Attempts to add an edge that would create a cycle, demonstrating the cycle detection mechanism.

Running the Program in Eclipse

1. Create a New Java Project: In Eclipse, go to 'File -> New -> Java Project'. Name the project, e.g., 'DirectedGraphExample'.
2. Create a New Java Class: Right-click on the 'src' folder in your project, then go to 'New -> Class'. Name the class 'DirectedGraph' and include the `public static void main(String[] args)` method.
3. Copy and Paste the Code: Copy the provided code and paste it into your 'DirectedGraph' class.
4. Run the Program: Right-click on your 'DirectedGraph' class in the Eclipse project explorer, then select 'Run As -> Java Application'.

The code in the 'DirectedGraph.java' file is as follows:

```
import java.util.*;  
public class DirectedGraph {  
    private Map<Integer, List<Integer>> adjList;  
    public DirectedGraph() {  
        adjList = new HashMap<Integer, List<Integer>>();  
    }  
    public void addNode(int node) {  
        adjList.putIfAbsent(node, new ArrayList<Integer>());  
    }  
    public boolean addEdge(int from, int to) {  
        addNode(from);  
        addNode(to);  
        if (createsCycle(from, to)) {  
            System.out.println("Edge from " + from + " to " + to + " not added to avoid cycle.");  
            return false;  
        } else {  
            adjList.get(from).add(to);  
            System.out.println("Edge from " + from + " to " + to + " added.");  
            return true;  
        }  
    }  
    private boolean createsCycle(int start, int end) {  
        Set<Integer> visited = new HashSet<Integer>();  
        Set<Integer> recStack = new HashSet<Integer>();  
        return dfs(start, visited, recStack, end);  
    }  
    private boolean dfs(int node, Set<Integer> visited, Set<Integer> recStack, int target) {  
        if (recStack.contains(node)) {  
            return true;  
        }  
        if (visited.contains(node)) {  
            return false;  
        }  
        visited.add(node);  
        recStack.add(node);  
        for (int neighbor : adjList.get(node)) {  
            if (neighbor == target || dfs(neighbor, visited, recStack, target)) {  
                return true;  
            }  
        }  
        recStack.remove(node);  
        return false;  
    }  
    public static void main(String[] args) {  
        DirectedGraph graph = new DirectedGraph();  
        graph.addEdge(1, 2);  
        graph.addEdge(2, 3);  
        graph.addEdge(3, 4);  
        graph.addEdge(4, 5);  
        // attempt to add an edge that creates a cycle  
        graph.addEdge(2, 1); // This should not be added  
        graph.addEdge(4, 2);  
    }  
}
```

The run output shows the following log:

```
Edge from 1 to 2 added.  
Edge from 2 to 3 added.  
Edge from 3 to 4 added.  
Edge from 4 to 5 added.  
Edge from 5 to 2 added.  
Edge from 4 to 2 not added to avoid cycle.
```

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.

Explanation

1. Graph Class:

- **adjList**: A map representing the adjacency list of the graph.
- **addNode**: Adds a node to the adjacency list if it doesn't already exist.
- **addEdge**: Adds an undirected edge between two nodes.
- **bfs**: Implements the Breadth-First Search algorithm. It starts from the given node, visits all the adjacent nodes, and prints each node in the order it is visited.

2. Main Method:

- Creates an instance of 'Graph'.
- Adds edges between nodes to form an undirected graph.
- Calls the 'bfs' method to start the BFS traversal from node 1.

Running the Program in Eclipse

- Create a New Java Project:** In Eclipse, go to 'File -> New -> Java Project'. Name the project, e.g., 'GraphExample'.
- Create a New Java Class:** Right-click on the 'src' folder in your project, then go to 'New -> Class'. Name the class 'Graph' and include the 'public static void main(String[] args)' method.
- Copy and Paste the Code:** Copy the provided code and paste it into your 'Graph' class.
- Run the Program:** Right-click on your 'Graph' class in the Eclipse project explorer, then select 'Run As -> Java Application'.

The code implements a Breadth-First Search (BFS) algorithm. It starts from a given node and prints each node in the order it is visited. The output shows the traversal starting from node 1, visiting nodes 2, 3, 4, and 5.

```
import java.util.*;  
  
public class Graph {  
    private Map<Integer, List<Integer>> adjList;  
  
    public Graph() {  
        adjList = new HashMap<>();  
    }  
  
    public void addNode(int node) {  
        adjList.putIfAbsent(node, new ArrayList<>());  
    }  
  
    public void addEdge(int node1, int node2) {  
        addNode(node1);  
        addNode(node2);  
        adjList.get(node1).add(node2);  
        adjList.get(node2).add(node1); // Since the graph is undirected  
    }  
  
    public void bfs(int startNode) {  
        Set<Integer> visited = new HashSet<>();  
        Queue<Integer> queue = new LinkedList<>();  
  
        queue.add(startNode);  
  
        while (!queue.isEmpty()) {  
            int currentNode = queue.poll();  
            System.out.print(currentNode + " ");  
  
            for (int neighbor : adjList.get(currentNode)) {  
                if (!visited.contains(neighbor)) {  
                    visited.add(neighbor);  
                    queue.add(neighbor);  
                }  
            }  
        }  
    }  
}
```

BFS starting from node 1:
1 2 3 4 5 6

BFS starting from node 1:
1 2 3 4 5 6

BFS starting from node 1:
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.

Explanation

- Graph Initialization: The graph is initialized with 6 vertices.
- Adding Edge: Edges are added between the vertices to form the graph.
- DFS Traversal: Starting from vertex 0, the `dfs` function recursively visits each vertex, marking it as visited and printing it out.
- Visited Array: The `visited` array ensures that each vertex is visited only once, preventing infinite loops in the case of cycles in the graph.

Running the Program in Eclipse

1. Open Eclipse: Create a new Java project and a new Java class named 'GraphDFS'.
2. Copy the Code: Copy the provided code into the 'GraphDFS.java' file.
3. Run the Program: Right-click on the file and select 'Run As -> Java Application'.
4. View Output: The output will be displayed in the Eclipse console, showing the DFS traversal starting from vertex 0.

The image contains three vertically stacked screenshots of the Eclipse IDE interface. Each screenshot shows the Java code for a 'GraphDFS' class and its execution in the 'Console' tab.

Screenshot 1: The code defines a constructor that initializes the number of vertices and an adjacency list. It then adds edges between vertices 0, 1, 2, 3, and 4.

```
import java.util.ArrayList;
public class GraphDFS {
    private int numVertices;
    private List<List<Integer>> adjList;
    public GraphDFS(int vertices) {
        this.numVertices = vertices;
        adjList = new ArrayList<>(vertices);
        for (int i = 0; i < vertices; i++) {
            adjList.add(new ArrayList<>());
        }
    }
    public void addEdge(int src, int dest) {
        adjList.get(src).add(dest);
        adjList.get(dest).add(src); // As the graph is undirected
    }
}
```

Screenshot 2: The code includes a recursive DFS function that prints each visited vertex. The output shows the traversal starting from vertex 0, visiting vertices 0, 1, 3, 4, and 2.

```
private void dfs(int vertex, boolean[] visited) {
    visited[vertex] = true;
    System.out.print(vertex + " ");
    for (int neighbor : adjList.get(vertex)) {
        if (!visited[neighbor]) {
            dfs(neighbor, visited);
        }
    }
}
```

Screenshot 3: The main method creates a GraphDFS object with 6 vertices, adds edges, and performs the DFS traversal starting from vertex 0. The output shows the traversal order: 0, 1, 3, 4, 2, 5.

```
public static void main(String[] args) {
    GraphDFS graph = new GraphDFS(6);
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(1, 4);
    graph.addEdge(2, 5);
    System.out.println("DFS traversal starting from vertex 0:");
    graph.performDFS(0);
}
```