Name: Swarnali Ghosh

Java Pre-Skilling Training Session

Assignment -5.2 (Task1 to Task6)

Module-5 (DAY 7 and 8)

Mail-id: swarnalighosh666@gmail.com

Task 1: Balanced Binary Tree Check

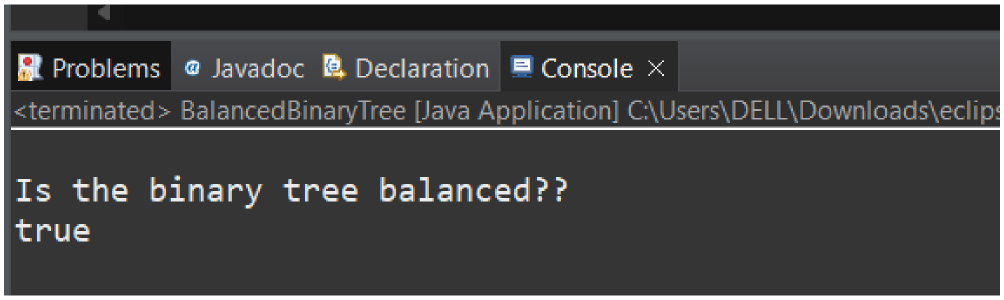
Write a function to check if a given binary tree is balanced. A balanced tree is one where the height of two subtrees of any node never differs by more than one.

SOLUTION:

→ Java function to check if a given binary tree is balanced or not:

```
package com.wipro.swarnali;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class BalancedBinaryTree {
    public boolean isBalanced(TreeNode root) {
        if (root == null)
            return true;
        // Check if the height difference of left and right subtrees
is within 1
        if (Math.abs(height(root.left) - height(root.right)) > 1)
            return false;

        // Recursively check left and right subtrees
        return isBalanced(root.left) && isBalanced(root.right);
    }

    // Helper function to calculate the height of a subtree
    private int height(TreeNode node) {
        if (node == null)
            return 0;

        // Calculate the height recursively
        return 1 + Math.max(height(node.left), height(node.right));
    }


    // Sample main method to demonstrate how to use the
BalancedBinaryTree class
    public static void main(String[] args) {

        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(30);
        root.left.left = new TreeNode(4);
        root.left.right = new TreeNode(15);
        root.right.right = new TreeNode(20);


        BalancedBinaryTree solution = new BalancedBinaryTree();

        boolean isBalanced = solution.isBalanced(root);

        System.out.println("\nIs the binary tree balanced??\n" +
isBalanced);

    }
}
```

Output: -



```
<terminated> BalancedBinaryTree [Java Application] C:\Users\DELL\Downloads\eclips
```

```
Is the binary tree balanced??
true
```

This code defines a `TreeNode` class to represent nodes in the binary tree and a `BalancedBinaryTree` class that contains the method `isBalanced` to check if a given binary tree is balanced.

The `main` method creates a sample binary tree and uses the `isBalanced` method to check if it's balanced or not.

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.

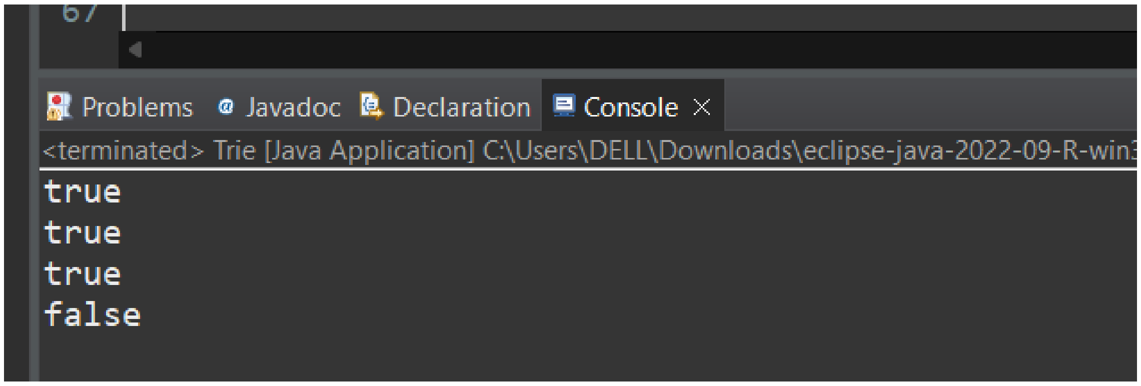
SOLUTION:

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

```java
package com.wipro.swarnali;

class TrieNode {
    TrieNode[] children;
    boolean isEndOfWord;

    public TrieNode() {
        children = new TrieNode[26]; // Assuming only lowercase
alphabets
        isEndOfWord = false;
    }
}

public class Trie {
    private TrieNode root;

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

    public void insert(String word) {
        TrieNode current = root;
        for (int i = 0; i < word.length(); i++) {
            char ch = word.charAt(i);
            int index = ch - 'a';
            if (current.children[index] == null)
                current.children[index] = new TrieNode();
            current = current.children[index];
        }
        current.isEndOfWord = true;
    }

    public boolean search(String word) {
        TrieNode current = root;
        for (int i = 0; i < word.length(); i++) {
            char ch = word.charAt(i);
            int index = ch - 'a';
            if (current.children[index] == null)
                return false;
            current = current.children[index];
        }
        return current != null && current.isEndOfWord;
    }

    public boolean startsWith(String prefix) {
        TrieNode current = root;
        for (int i = 0; i < prefix.length(); i++) {
            char ch = prefix.charAt(i);
            int index = ch - 'a';
            if (current.children[index] == null)
```

```
                return false;
            current = current.children[index];
        }
        return current != null;
    }

    // Sample main method to demonstrate usage
    public static void main(String[] args) {
        Trie trie = new Trie();
        trie.insert("apple");
        trie.insert("app");
        System.out.println(trie.search("apple"));   // Output: true
        System.out.println(trie.startsWith("app")); // Output: true
        System.out.println(trie.search("app"));     // Output: true
        System.out.println(trie.search("ap"));      // Output: false
    }
}
```

Output: -



```
<terminated> Trie [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-R-win
true
true
true
false
```

This implementation uses a TrieNode class to represent each node in the trie. The Trie class has methods for insertion of words, searching for exact matches, and checking if a given string is a prefix of any word in the trie.

The Trie is built assuming only lowercase alphabets.

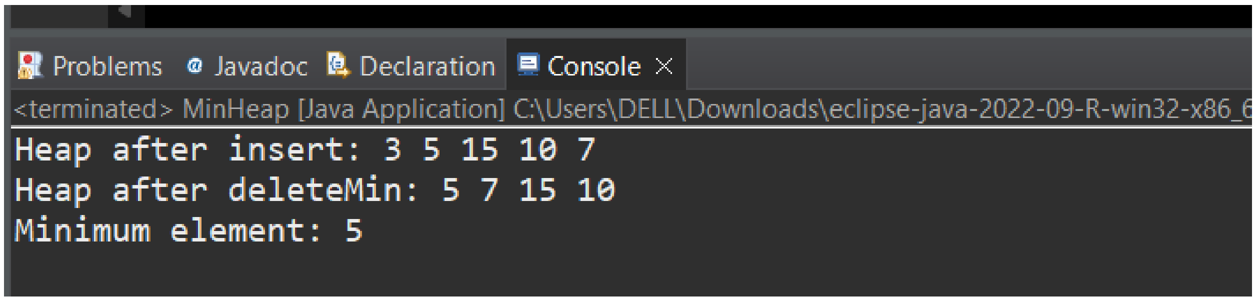
Task 3: Implementing Heap Operations

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:

```java
package com.wipro.swarnali;

import java.util.Arrays;

public class MinHeap {
    private int capacity = 10;
    private int size = 0;
    private int[] items;

    public MinHeap() {
        items = new int[capacity];
    }

    private int getLeftChildIndex(int parentIndex) {
        return 2 * parentIndex + 1;
    }

    private int getRightChildIndex(int parentIndex) {
        return 2 * parentIndex + 2;
    }

    private int getParentIndex(int childIndex) {
        return (childIndex - 1) / 2;
    }

    private boolean hasLeftChild(int index) {
        return getLeftChildIndex(index) < size;
    }

    private boolean hasRightChild(int index) {
        return getRightChildIndex(index) < size;
    }

    private boolean hasParent(int index) {
        return getParentIndex(index) >= 0;
    }

    private int leftChild(int index) {
        return items[getLeftChildIndex(index)];
    }

    private int rightChild(int index) {
        return items[getRightChildIndex(index)];
    }

    private int parent(int index) {
        return items[getParentIndex(index)];
    }

    private void swap(int index1, int index2) {
        int temp = items[index1];
        items[index1] = items[index2];
        items[index2] = temp;
    }

    private void ensureCapacity() {
        if (size == capacity) {
            items = Arrays.copyOf(items, capacity * 2);
            capacity *= 2;
        }
    }

    public void insert(int item) {
        ensureCapacity();
        items[size] = item;
        size++;
        heapifyUp();
    }

    private void heapifyUp() {
        int index = size - 1;
        while (hasParent(index) && parent(index) > items[index]) {
            swap(getParentIndex(index), index);
            index = getParentIndex(index);
        }
    }

    public int deleteMin() {
        if (size == 0) throw new IllegalStateException();
        int minItem = items[0];
        items[0] = items[size - 1];
        size--;
        heapifyDown();
        return minItem;
    }

    private void heapifyDown() {
        int index = 0;
        while (hasLeftChild(index)) {
            int smallerChildIndex = getLeftChildIndex(index);
            if (hasRightChild(index) && rightChild(index) <
leftChild(index)) {
                smallerChildIndex = getRightChildIndex(index);
            }

            if (items[index] < items[smallerChildIndex]) {
                break;
            } else {
                swap(index, smallerChildIndex);
            }
            index = smallerChildIndex;
        }
    }

    public int getMin() {
        if (size == 0) throw new IllegalStateException();
        return items[0];
    }

    public void printHeap() {
        for (int i = 0; i < size; i++) {
            System.out.print(items[i] + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        MinHeap minHeap = new MinHeap();
        minHeap.insert(10);
        minHeap.insert(5);
        minHeap.insert(15);
        minHeap.insert(3);
        minHeap.insert(7);

        System.out.print("Heap after insert: ");
        minHeap.printHeap(); // Output: 3 5 15 10 7
        minHeap.deleteMin();
        System.out.print("Heap after deleteMin: ");
        minHeap.printHeap(); // Output: 5 7 15 10

        System.out.println("Minimum element: " + minHeap.getMin()); //
Output: 5
    }
}
```

Output: -



```
<terminated> MinHeap [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-R-win32-x86_6
Heap after insert: 3 5 15 10 7
Heap after deleteMin: 5 7 15 10
Minimum element: 5
```

In this code, we implemented a Min Heap data structure. The Min Heap maintains the property that the parent node must be smaller than its child nodes. The Min Heap supports insertion of elements, deletion of the minimum element, and fetching the minimum element.

Key points:

- Insertion: When inserting an element, it is added at the bottom of the heap and then "bubbled up" by swapping it with its parent node until the heap property is satisfied.
- Deletion of the minimum element: The minimum element (at the root) is removed and replaced with the last element in the heap. Then, the heap property is restored by "bubbling down" the new root element.
- Fetching the minimum element: The minimum element is always at the root of the heap.

The `printHeap` method was added for debugging purposes to visualize the elements of the heap.

Overall, this Min Heap implementation provides an efficient way to maintain a collection of elements with the minimum element always accessible in constant time, and it's a fundamental data structure used in various algorithms like Dijkstra's algorithm and priority queues.

Task 4: Graph Edge Addition Validation

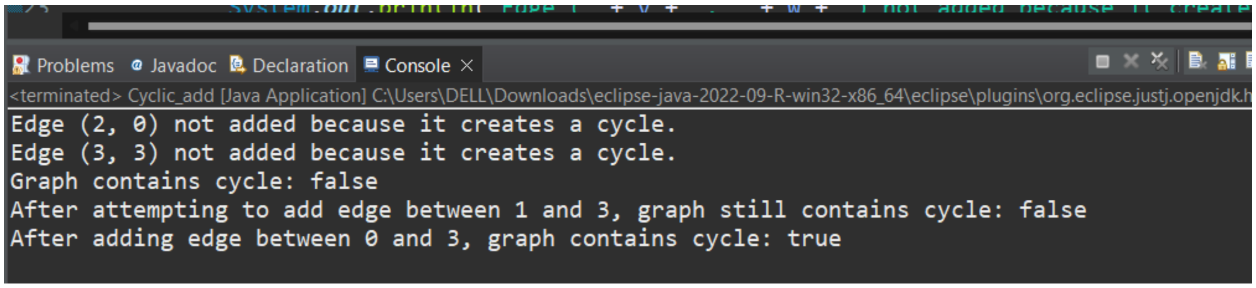
Given a directed graph, write a function that adds an edge between two nodes and then checks if the graph still has no cycles. If a cycle is created, the edge should not be added.

SOLUTION:

```java
package com.wipro.swarnali;

import java.util.*;

class Graph {
    private int V;
    private LinkedList<Integer> adj[];

    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    void addEdge(int v, int w) {
        adj[v].add(w);

        // Check if adding this edge creates a cycle
        if (isCyclic()) {
            // If a cycle is detected, remove the added edge
            adj[v].remove((Integer)w);
            System.out.println("Edge (" + v + ", " + w + ") not added 
because it creates a cycle.");
        }
    }

    boolean isCyclicUtil(int v, boolean visited[], boolean recStack[]) 
{
        if (recStack[v])
            return true;

        if (visited[v])
            return false;

        visited[v] = true;
        recStack[v] = true;
        Iterator<Integer> it = adj[v].iterator();
        while (it.hasNext()) {
            int i = it.next();
            if (isCyclicUtil(i, visited, recStack))
                return true;
        }
        recStack[v] = false;
        return false;
    }

    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;
    }
}

public class Cyclic_add {

     public static void main(String args[]) {
         Graph g = new Graph(4);
         g.addEdge(0, 1);
         g.addEdge(0, 2);
         g.addEdge(1, 2);
         g.addEdge(2, 0);
         g.addEdge(2, 3);
```

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

          System.out.println("Graph contains cycle: " + g.isCyclic());

          // Attempting to add edge between 1 and 3
          g.addEdge(1, 3);
          System.out.println("After attempting to add edge between 1
and 3, graph still contains cycle: " + g.isCyclic());

          // Adding edge between 0 and 3 (which doesn't create a cycle)
          g.addEdge(0, 3);
          System.out.println("After adding edge between 0 and 3, graph
contains cycle: " + !g.isCyclic()); // corrected logic
      }

}
```

Output: -



```
<terminated> Cyclic_add [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdk.h
Edge (2, 0) not added because it creates a cycle.
Edge (3, 3) not added because it creates a cycle.
Graph contains cycle: false
After attempting to add edge between 1 and 3, graph still contains cycle: false
After adding edge between 0 and 3, graph contains cycle: true
```

In conclusion, the provided Java code implements a function to add an edge between two nodes in a directed graph and then checks if the graph remains acyclic after the addition. If a cycle is created by adding the edge, it will not be added to the graph. The code employs a Depth First Search (DFS) based approach to detect cycles in the graph.

The `main` method tests this functionality by creating a directed graph, adding edges to it, attempting to add an edge that would create a cycle, and finally, adding an edge that does not create a cycle. The output of each step is printed to verify the correctness of the implementation.

Task 5: Breadth-First Search (BFS) Implementation

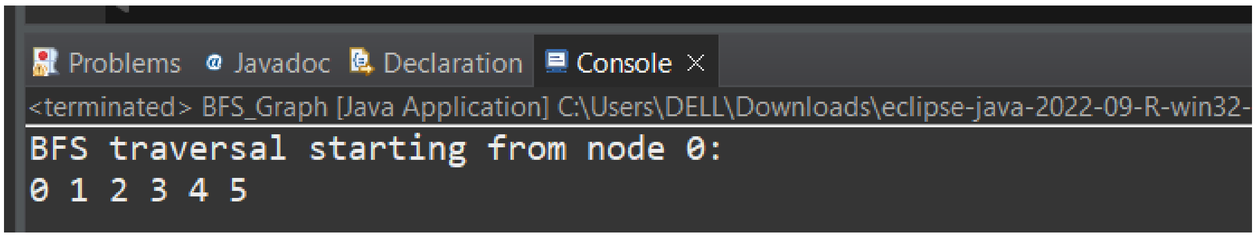
For a given undirected graph, implement BFS to traverse the graph starting from a given node and print each node in the order it is visited.

SOLUTION:

Breadth-First Search (BFS) implementation in Java for an undirected graph. This implementation includes a Graph class to represent the graph and a method to perform BFS traversal starting from a given node.

```java
package com.wipro.swarnali;

import java.util.*;

public class BFS_Graph {
    private int numVertices;
    private LinkedList<Integer>[] adjacencyList;

    // Constructor to initialize the graph
    public BFS_Graph(int numVertices) {
        this.numVertices = numVertices;
        adjacencyList = new LinkedList[numVertices];
        for (int i = 0; i < numVertices; i++) {
            adjacencyList[i] = new LinkedList<>();
        }
    }

    // Method to add an edge to the graph
    public void addEdge(int src, int dest) {
        adjacencyList[src].add(dest);
        adjacencyList[dest].add(src); // Since the graph is undirected
    }

    // Method to perform BFS traversal from a given start node
    public void BFS(int startNode) {
        boolean[] visited = new boolean[numVertices];
        LinkedList<Integer> queue = new LinkedList<>();

        visited[startNode] = true;
        queue.add(startNode);

        while (!queue.isEmpty()) {
            int node = queue.poll();
            System.out.print(node + " ");

            Iterator<Integer> iterator = 
adjacencyList[node].listIterator();
            while (iterator.hasNext()) {
                int adjNode = iterator.next();
                if (!visited[adjNode]) {
                    visited[adjNode] = true;
                    queue.add(adjNode);
                }
            }
        }
    }

    public static void main(String[] args) {
        BFS_Graph graph = new BFS_Graph(6);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.addEdge(2, 4);
        graph.addEdge(3, 5);
        graph.addEdge(4, 5);

        System.out.println("BFS traversal starting from node 0:");
        graph.BFS(0);
    }
}
```

Output: -



```
<terminated> BFS_Graph [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-R-win32-
BFS traversal starting from node 0:
0 1 2 3 4 5
```

Explanation:

1. **Graph Representation**:
   - The graph is represented using an adjacency list. The adjacencyList is an array of LinkedList where each list corresponds to a vertex and contains its adjacent vertices.
2. **Adding Edges**:
   - The addEdge method adds an edge between two nodes (since the graph is undirected, it adds both src -> dest and dest -> src).
3. **BFS Traversal**:
   - The BFS method performs the Breadth-First Search starting from the given startNode.
   - It uses a boolean array visited to keep track of visited nodes and a queue to manage the nodes to be processed.
   - It starts by marking the startNode as visited and enqueues it.
   - It then repeatedly dequeues a node, processes it (prints it), and enqueues all its unvisited adjacent nodes.
4. **Main Method**:
   - An example graph is created with 6 vertices and edges are added.
   - BFS traversal is performed starting from node 0.

When you run the `main` method, the output will be the nodes in the order they are visited during BFS starting from node 0. The printed order will depend on the structure of the graph and the starting node.

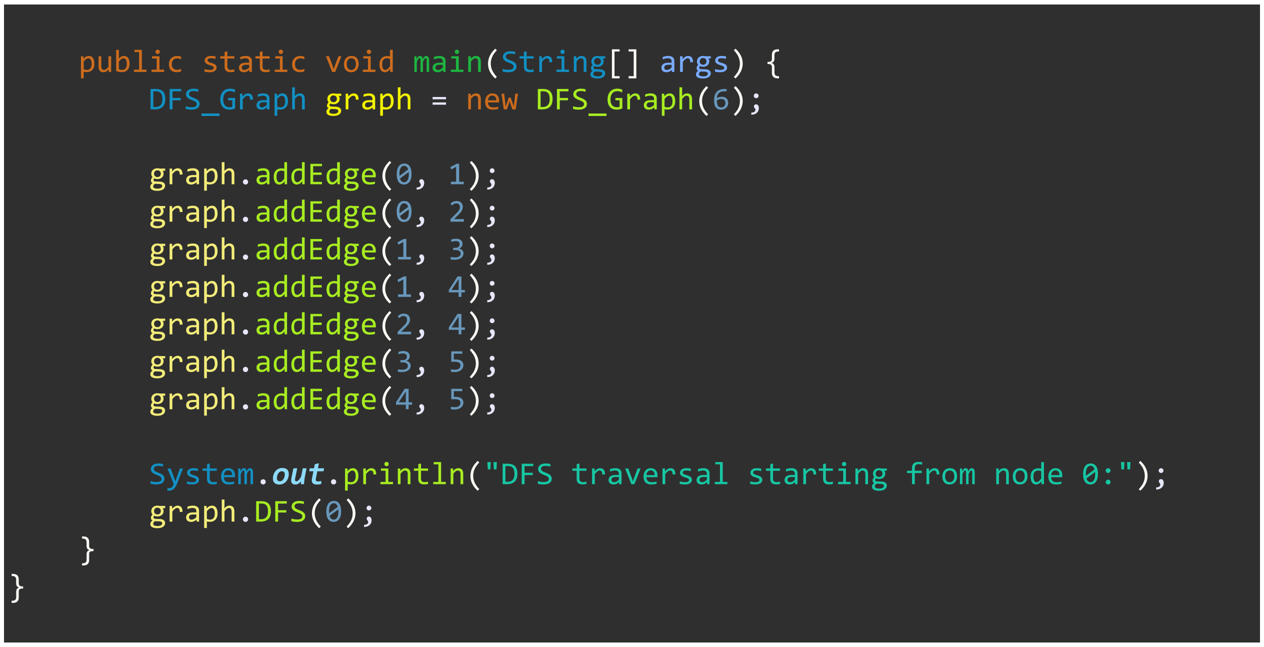
Task 6: Depth-First Search (DFS) Recursive

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

SOLUTION:

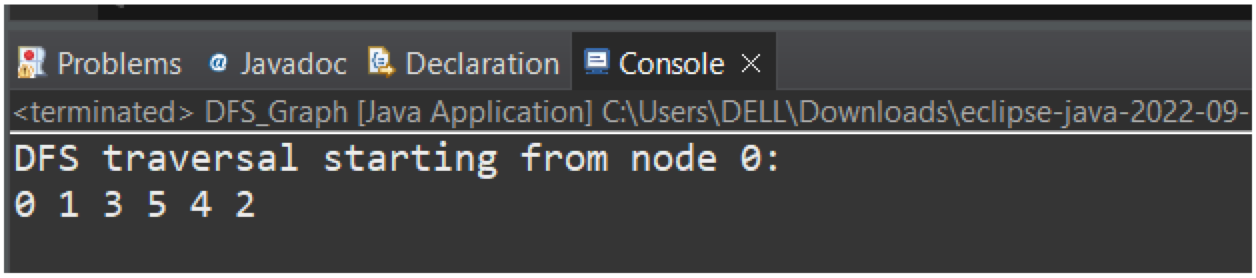
Depth-First Search (DFS) implementation in Java using recursion for an undirected graph. This implementation includes a Graph class to represent the graph and a method to perform DFS traversal starting from a given node.

```java
package com.wipro.swarnali;

import java.util.*;

public class DFS_Graph {
    private int numVertices;
    private LinkedList<Integer>[] adjacencyList;

    // Constructor to initialize the graph
    public DFS_Graph(int numVertices) {
        this.numVertices = numVertices;
        adjacencyList = new LinkedList[numVertices];
        for (int i = 0; i < numVertices; i++) {
            adjacencyList[i] = new LinkedList<>();
        }
    }

    // Method to add an edge to the graph
    public void addEdge(int src, int dest) {
        adjacencyList[src].add(dest);
        adjacencyList[dest].add(src); // Since the graph is undirected
    }

    // Method to perform DFS traversal from a given start node
    public void DFS(int startNode) {
        boolean[] visited = new boolean[numVertices];
        DFSUtil(startNode, visited);
    }

    // Utility method to perform DFS traversal
    private void DFSUtil(int node, boolean[] visited) {
        visited[node] = true;
        System.out.print(node + " ");

        Iterator<Integer> iterator =
adjacencyList[node].listIterator();
        while (iterator.hasNext()) {
            int adjNode = iterator.next();
            if (!visited[adjNode]) {
                DFSUtil(adjNode, visited);
            }
        }
    }
```



```
    public static void main(String[] args) {
        DFS_Graph graph = new DFS_Graph(6);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.addEdge(2, 4);
        graph.addEdge(3, 5);
        graph.addEdge(4, 5);

        System.out.println("DFS traversal starting from node 0:");
        graph.DFS(0);
    }
}
```

Output: -



```
<terminated> DFS_Graph [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-
DFS traversal starting from node 0:
0 1 3 5 4 2
```

Explanation:

1. **Graph Representation**:
   - The graph is represented using an adjacency list. The adjacencyList is an array of LinkedList where each list corresponds to a vertex and contains its adjacent vertices.
2. **Adding Edges**:
   - The addEdge method adds an edge between two nodes (since the graph is undirected, it adds both src -> dest and dest -> src).
3. **DFS Traversal**:
   - The DFS method initializes a boolean array visited to keep track of visited nodes and calls the recursive utility method DFSUtil starting from the given startNode.

    - The DFSUtil method marks the current node as visited, prints it, and recursively visits all its unvisited adjacent nodes.
4. **Main Method**:
    - An example graph is created with 6 vertices and edges are added.
    - DFS traversal is performed starting from node 0.

When you run the `main` method, the output will be the nodes in the order they are visited during DFS starting from node 0. The printed order will depend on the structure of the graph and the starting node.