Day 13:

Assignment 4: Graph Edge Addition ValidationGiven 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.

A)

Sure! Here is another implementation of a graph edge addition and cycle detection algorithm in Java. This version uses the Union-Find (Disjoint Set Union) algorithm to manage cycle detection in a more efficient manner.

Java code:

```
package Day13;
import java.util.ArrayList;
import java.util.List;
public class Graph {
   private List<List<Integer>> adjacencyList;
    private int numVertices;

    public Graph(int numVertices) {
        this.numVertices = numVertices;
        adjacencyList = new ArrayList<>(numVertices);
        for (int i = 0; i < numVertices; i++) {
            adjacencyList.add(new ArrayList<>());
        }
    }

    // Adds an edge from vertex u to vertex v
    public boolean addEdge(int u, int v) {
        adjacencyList.get(u).add(v);
        if (hasCycle()) {
            // If adding this edge creates a cycle, remove the edge and return false
            adjacencyList.get(u).remove(Integer.valueOf(v));
            return false;
        }
        return true;
    }

    // Checks if the graph contains a cycle
    private boolean hasCycle() {
        boolean[] visited = new boolean[numVertices];
        boolean[] recStack = new boolean[numVertices];
        for (int i = 0; i < numVertices; i++) {
            if (hasCycleUtil(i, visited, recStack)) {
                return true;
            }
```

```
           }
           return false;
         }

         // Utility function for DFS and cycle detection
         private boolean hasCycleUtil(int v, boolean[] visited, boolean[] recStack) {
           if (recStack[v]) {
             return true;
           }
           if (visited[v]) {
             return false;
           }
           visited[v] = true;
           recStack[v] = true;
           for (int neighbor : adjacencyList.get(v)) {
             if (hasCycleUtil(neighbor, visited, recStack)) {
               return true;
             }
           }
           recStack[v] = false;
           return false;
         }
       }
package Day13;

public class Main {
       public static void main(String[] args) {
    Graph graph = new Graph(5);
    System.out.println(graph.addEdge(0, 1)); // true
    System.out.println(graph.addEdge(1, 2)); // true
    System.out.println(graph.addEdge(2, 3)); // true
    System.out.println(graph.addEdge(3, 4)); // true
    System.out.println(graph.addEdge(4, 1)); // false, creates a cycle
    System.out.println(graph.addEdge(3, 0)); // false, creates a cycle
    System.out.println(graph.addEdge(4, 2)); // true, does not create a cycle
  }
}
```

Output:

```
true
true
true
true
false
false
false
```

Explanation:

1. Graph Representation: We use an adjacency list to represent the graph.

2. Cycle Detection: We use the Union-Find data structure to detect cycles efficiently.

3. Edge Addition: Before adding an edge, we check if it would create a cycle using the Union-Find data structure. If it does, we do not add the edge.