Data Structure and Algorithm Practicum
Graph

Name
Muhammad Baihaqi Aulia Asy'ari

NIM
2241720145

Class
1I

Department
Information Technology

Study Program
D4 Informatics Engineering

1.1 Learning Objective

After doing this practicum, students are able to:

1. Understand graph models;
2. Create and declare graph algorithm structure;
3. Apply graph's basic algorithm in several case studies.

1.2 Practicum 1

1.2.1 Steps

In practicum 1, Graph will be implemented using Linked Lists to represent adjacency graphs. Please do the practical steps as follows.

1. Make a class **Node**, and class Linked Lists in accordance with the **Double Linked Lists** practicum.
2. Add a **Graph** class that will store methods in the graph as well as the main () method.

```
public class Graph {

}
```

3. In the **Graph** class, add the **vertex** attribute of type integer and **list** [] of type LinkedList.

```
public class Graph {
    int vertex;
    LinkedList[] list;
}
```

4. Add a default constructor to initialize the vertex variable and add a loop for the number of vertices according to the number of length arrays that have been determined.

```
public Graph(int vertex) {
    this.vertex = vertex;
    list = new LinkedList[vertex];
    for (int i = 0; i < vertex; i++) {
        list[i] = new LinkedList();
    }
}
```

5. Add the **addEdge ()** method. If the directed graph will be created, then only the first row will be run. If the graph is not directed, run all lines in the **addEdge ()** method

```
public void addEdge(int source, int destination) {
    list[source].addFirst(destination);
    list[destination].addFirst(source);
}
```

6. Add the **degree ()** method to display the number of degrees in the vertex. Within this method also distinguishes which statements are used for directed graphs or undirected graphs. Execution only as needed.

```
public void degree(int source) throws Exception {
    System.out.println("degree vertex " + source + " : " +
    ↪  list[source].size());

    int k = 0, totalIn = 0, totalOut = 0;
    for (int i = 0; i < vertex; i++) {
        for (int j = 0; j < list[i].size(); j++) {
            if (list[i].get(j) == source) {
                ++totalIn;
            }
        }
        for (int j = 0; j < list[source].size(); j++) {
            list[source].get(j);
            k = j;
        }
        totalOut = k;
    }

    System.out.println("Indegree dari vertex " + source + " : " +
    ↪  totalIn);
    System.out.println("Outdegree dari vertex " + source + " : " +
    ↪  totalOut);
    System.out.println("degree vertex " + source + " : " + (totalIn +
    ↪  totalOut));
}
```

7. Add the **removeEdge ()** method. This method will delete the path in a graph. Therefore, it takes 2 parameters to delete the path, namely source and destination.

```
public void removeEdge(int source, int destination) throws Exception {
    for (int i = 0; i < vertex; i++) {
        if (i == destination) {
            list[source].remove(destination);
        }
    }
}
```

8. Add the **removeAllEdges ()** method to delete all vertices in the graph.

```
    public void removeAllEdge() {
        for (int i = 0; i < vertex; i++) {
            list[i].clear();
        }
        System.out.println("Graph Successfully Cleared");
    }
```

9. Add the **printGraph ()** method to record the updated graph.

```
    public void printGraph() {
        for (int i = 0; i < vertex; i++) {
            if (list[i].size() > 0) {
                System.out.print("Vertex " + i + " terhubung dengan: ");
                for (int j = 0; j < list[i].size(); j++) {
                    System.out.print(list[i].get(j) + " ");
                }
                System.out.println();
            }
        }
        System.out.println();
    }
```

10. Compile and run the **main ()** method in the **Graph** class to add a few edges to the graph, then display. After that, remove the results using the main () method call. Note: degree must be adjusted to the type of graph that has been created (directed / undirected).

11. Observe the results of the running.

12. Add the **removeEdge ()** method call according to the code snippet below to the main () method. Then display the graph.

13. Observe the results of the running.

14. Try deleting another track! Observe the results!

1.2.2 Verification of Practicum Results

Verify the results of your program code compilation with the following image.

The results of running in step 11

```
PS D:\Kuliah\Smt 2\Algoritma dan Struktur Data\Praktikum\Week 15\Graph>  & 'C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'D:\Kuliah\Smt 2\Algoritma dan Struktur Data\Praktikum\Week 15\Graph\bin' 'Graph'
Vertex 1 terhubung dengan: 4 3 2 0
Vertex 2 terhubung dengan: 3 1
Vertex 3 terhubung dengan: 0 4 2 1
Vertex 4 terhubung dengan: 3 1 0

degree vertex 2 : 2
Indegree dari vertex 2 : 2
Outdegree dari vertex 2 : 1
degree vertex 2 : 3
```

The results of running in step 13

```
PS D:\Kuliah\Smt 2\Algoritma dan Struktur Data\Praktikum\Week 15\Graph> d:; cd 'd:\Kuliah\Smt 2\Algoritma dan Struktur Data\Praktikum\Week 15\Graph'; & 'C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'D:\Kuliah\Smt 2\Algoritma dan Struktur Data\Praktikum\Week 15\Graph\bin' 'Graph'
Vertex 0 terhubung dengan: 3 4 1
Vertex 1 terhubung dengan: 4 3 2 0
Vertex 2 terhubung dengan: 3 1
Vertex 3 terhubung dengan: 0 4 2 1
Vertex 4 terhubung dengan: 3 1 0

degree vertex 2 : 2
Indegree dari vertex 2 : 2
Outdegree dari vertex 2 : 1
degree vertex 2 : 3
Vertex 0 terhubung dengan: 3 4 1
Vertex 1 terhubung dengan: 4 3 0
Vertex 2 terhubung dengan: 3 1
Vertex 3 terhubung dengan: 0 4 2 1
Vertex 4 terhubung dengan: 3 1 0
```

1.2.3 Questions

1. Mention 3 kinds of algorithm that uses Graph fundamental,what's the use of those ?

2. In class Graph, there is an array with LinkedList data type, LinkedList list[]. What's the aim of this?
3. What is the reason of calling method addFirst() to add data, instead of calling other add methods in Linked list when using method addEdge in class Graph?
4. How do we detect prev pointer when we are about to remove an edge of a graph?
5. Why in practicum 1.2, the 12th step is to remove path that is not the first path to produce the wrong output? What's the solution?

   ```
   graph.removeEdge(1,3);
   graph.printGraph();
   ```

1.3 Practicum 2

Please do the following steps in practicum 2, then verify the results. After that answer the questions related to the practicum that you have done. Implementation of Graphs with a Matrix

1.3.1 Steps

In practicum 2, Graph will be implemented using a matrix to represent graph adjacency. Please do the practical steps as follows.

1. Practicum graph part 2 uses a 2-dimensional array as graph representation. Make **graphArray** class in which there are **vertices** and **twoD-array arrays**!

   ```
   public class GraphArray {
       private final int vertices;
       private final int[][] twoD_array;
   }
   ```

2. Make the **graphArray** constructor as follows!

   ```
   public GraphArray(int v) {
       vertices = v;
       twoD_array = new int[vertices + 1][vertices + 1];
   }
   ```

3. To make a path a **makeEdge ()** method is made as follows.

   ```
   public void makeEdge(int to, int from, int edge) {
       try {
           twoD_array[to][from] = edge;
       } catch (ArrayIndexOutOfBoundsException index) {
           System.out.println("Vertex tidak ada");
   ```

```
    }
}
```

4. To display a path requires creating the following **getEdge ()** method.

```
public int getEdge(int to, int from) {
    try {
        return twoD_array[to][from];
    } catch (ArrayIndexOutOfBoundsException index) {
        System.out.println("Vertex tidak ada");
    }
    return -1;
}
```

5. Then make the **main ()** method as follows.

```
public static void main(String[] args) {
    int v, e, count = 1, to = 0, from = 0;
    Scanner sc = new Scanner(System.in);
    GraphArray graph;
    try {
        System.out.println("Masukkan jumlah vertices: ");
        v = sc.nextInt();
        System.out.println("Masukkan jumlah edges: ");
        e = sc.nextInt();

        graph = new GraphArray(v);

        System.out.println("Masukkan edges: <to> <from>");
        while (count <= e) {
            to = sc.nextInt();
            from = sc.nextInt();

            graph.makeEdge(to, from, 1);
            count++;
        }
        System.out.println("Array 2D sebagai representasi graph sbb:
        ↪  ");
        System.out.print("  ");
        for (int i = 1; i <= v; i++) {
            System.out.print(i + " ");
        }
        System.out.println();

        for (int i = 1; i <= v; i++) {
            System.out.print(i + " ");
```

```
                for (int j = 1; j <= v; j++) {
                    System.out.print(graph.getEdge(i, j) + " ");
                }
                System.out.println();
            }
        } catch (Exception E) {
            System.out.println("Error. silahkan cek kembali\n" +
             ↪  E.getMessage());
        }
        sc.close();
    }
```

6. Run the **graphArray** class and observe the results!

1.3.2 Result

```
PS D:\Kuliah\Smt 2\Algoritma dan Struktur Data\Praktikum\Week 15\Graph>  &
 ↪  'C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe'
 ↪  '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'D:\Kuliah\Smt
 ↪  2\Algoritma dan Struktur Data\Praktikum\Week 15\Graph\bin' 'GraphArray'
Masukkan jumlah vertices:
5
Masukkan jumlah edges:
6
Masukkan edges: <to> <from>
1 2
1 5
2 3
2 4
2 5
3 4
Array 2D sebagai representasi graph sbb:
  1 2 3 4 5
1 0 1 0 0 1
2 0 0 1 1 1
3 0 0 0 1 0
4 0 0 0 0 0
5 0 0 0 0 0
```

1.3.3 Pertanyaan Percobaan

1. What is the degree difference between directed and undirected graphs?

2. In the graph implementation using adjacency matrix. Why does the number of vertices have to be added to 1 in the following array index?

```java
public graphArray(int v) {
    vertices = v;
    twoD_array = new int[vertices + 1][vertices + 1];
}
```

3. What is the use of the **getEdge()** method?
4. What kind of graph were implemented on practicum 1.3 ?
5. Why does the main method use try-catch Exception?

1.4 Tugas Praktikum

1. Convert the path in 1.2 as an input !

```java
public static void main(String[] args) throws Exception {
    // Graph graph = new Graph(6);
    // graph.addEdge(0, 1);
    // graph.addEdge(0, 4);
    // graph.addEdge(1, 2);
    // graph.addEdge(1, 3);
    // graph.addEdge(1, 4);
    // graph.addEdge(2, 3);
    // graph.addEdge(3, 4);
    // graph.addEdge(3, 0);
    // graph.printGraph();
    // graph.degree(2);

    // graph.removeEdge(1, 2);
    // graph.printGraph();

    Scanner sc = new Scanner(System.in);

    System.out.print("Insert vertex amount: ");
    int vertexCount = sc.nextInt();

    Graph graph = new Graph(vertexCount);

    System.out.println("Insert vertex: <to> <from>");
    for (int i = 0; i < vertexCount; i++) {
        graph.addEdge(sc.nextInt(), sc.nextInt());
    }
    graph.printGraph();
    graph.degree(2);
    sc.close();
}
```

2. Add method **graphType** with boolean as its return type to differentiate which graph is directed or undirected graph. Then update all the method that relates to **graphType()** (only runs the statement based on the graph type) in practicum 1.2

```
public boolean graphType() throws Exception {
    int totalIn = 0, totalOut = 0;
    for (int i = 0; i < vertex; i++) {
        for (int j = 0; j < vertex; j++) {
            for (int k = 0; k < list[j].size(); k++) {
                if (list[j].get(k) == i) {
                    totalIn++;
                }
            }
            for (int k = 0; k < list[i].size(); k++) {
                if (list[i].get(k) == j) {
                    totalOut++;
                }
            }
        }
    }
    return (totalIn != totalOut);
}
```

3. Modify method **removeEdge()** in practicum 1.2 so that it won't give the wrong path other than the initial path as an output !

```
public void removeEdge(int source, int destination) throws Exception {
    // for (int i = 0; i < vertex; i++) {
    //     if (i == destination) {
    //         list[source].remove(destination);
    //     }
    // }

    int sourceIndex = list[destination].search(source);
    int destinationIndex = list[source].search(destination);
    list[source].remove(destinationIndex);
    list[destination].remove(sourceIndex);
}
```

4. Convert vertex's data type in the graph of practicum 1.2. and 1.3 from integer to generic data type so that it can accepts all basic data type in Java programming language! For example, if the initial vertex are 0,1,2,3, dst. Then the next will

be in form of region name, like Malang, Surabaya, Gresik, Bandung, dst.
LinkedList.java

```
public class DoubleLinkedList<TData> {
    Node<TData> head;
    int size;

    public DoubleLinkedList() {
        head = null;
        size = 0;
    }

    boolean isEmpty() {
        return size == 0;
    }

    void addFirst(TData item) {
        if (isEmpty()) {
            head = new Node<>(null, item, null);
        } else {
            Node<TData> newNode = new Node<>(null, item, head);
            head.prev = newNode;
            head = newNode;
        }
        size++;
    }

    int size() {
        return size;
    }

    void clear() {
        head = null;
        size = 0;
    }

    void removeFirst() throws Exception {
        if (isEmpty()) {
            throw new Exception("Linked list is still empty, cannot
             ↪  remove");
        }

        if (size == 1) {
            removeLast();
            return;
        }

        head = head.next;
        head = null;
        size--;
    }

    void removeLast() throws Exception {
        if (isEmpty()) {
            throw new Exception("Linked list is still empty, cannot
            ↪  remove");
        }

        if (head.next == null) {
            head = null;
        } else {
            Node<TData> current = head;
            while (current.next.next != null) {
                current = current.next;
            }
            current.next = null;
        }
        size--;
    }

    void remove(int index) throws Exception {
        if (isEmpty() || index >= size) {
            throw new Exception("Index value is out of bound");
        }

        if (index == 0) {
            removeFirst();
            return;
        }

        Node<TData> current = head;
        int i = 0;
        while (i < index - 1) {
            current = current.next;
            i++;
        }
        current.next = current.next.next;
        size--;
    }
    TData get(int index) throws Exception {
        if (isEmpty()) {
            throw new Exception("Linked list is still empty");
        }

        Node<TData> tmp = head;
        for (int i = 0; i < index; i++) {
            tmp = tmp.next;
        }

        return tmp.data;
    }

    int search(TData data) {
        if (isEmpty()) return -1;

        Node<TData> current = head;
        int i = 0;
        while (current != null) {
            if (current.data == data) return i;
            i++;
            current = current.next;
        }

        return -1;
    }
}
```

Graph.java

```
public class Graph<TData> {
    int vertex;
    HashMap<TData, DoubleLinkedList<TData>> list;

    public Graph(int vertex) {
        this.vertex = vertex;
        list = new HashMap<>();
    }

    public void addEdge(TData source, TData destination) {
        list.putIfAbsent(source, new DoubleLinkedList<>());
        list.putIfAbsent(destination, new DoubleLinkedList<>());

        list.get(source).addFirst(destination);
        list.get(destination).addFirst(source);
    }

    public void degree(TData source) throws Exception {
        System.out.println("degree vertex " + source + " : " +
        ↪  list.get(source).size());

        int totalIn = 0, totalOut = 0;
        for (TData key : list.keySet()) {
            for (int j = 0; j < list.get(key).size(); j++) {
                if (Objects.equals(list.get(key).get(j), source)) {
                    totalIn++;
                }
            }
            for (int j = 0; j < list.get(source).size(); j++) {
                if (Objects.equals(list.get(source).get(j), key)) {
                    totalOut++;
                }
            }
        }


        System.out.println("Indegree from vertex " + source + " : " +
        ↪  totalIn);
        System.out.println("Outdegree from vertex " + source + " : " +
        ↪  totalOut);
        System.out.println("Degree from vertex " + source + " : " +
        ↪  (totalIn + totalOut));
    }

    public void removeEdge(TData source, TData destination) throws
    ↪  Exception {
        int destinationIndex = list.get(source).search(destination);
        int sourceIndex = list.get(destination).search(source);
        list.get(source).remove(destinationIndex);
        list.get(destination).remove(sourceIndex);
    }

    public void printGraph() throws Exception {
        for (TData key : list.keySet()) {
            if (list.get(key).size() > 0) {
                System.out.print("Vertex " + key + " connected with :
                ↪  ");
                for (int j = 0; j < list.get(key).size(); j++) {
                    System.out.print(list.get(key).get(j) + " ");
                }
                System.out.println();
            }
        }
        System.out.println();
    }

    public boolean graphType() throws Exception {
        int totalIn = 0, totalOut = 0;
        for (TData key : list.keySet()) {
            for (int j = 0; j < list.get(key).size(); j++) {
                if (Objects.equals(list.get(key).get(j), key)) {
                    totalIn++;
                }
            }
            for (int j = 0; j < list.get(key).size(); j++) {
                if (Objects.equals(list.get(key).get(j), key)) {
                    totalOut++;
                }
            }
        }
        return (totalIn != totalOut);
    }

    public static void main(String[] args) throws Exception {
        Graph<Integer> graph = new Graph<>(6);

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

        graph.printGraph();
        graph.removeEdge(0, 1);

        graph.printGraph();
        graph.degree(2);

        System.out.println("Is directed graph: " + graph.graphType());
    }
}
```