Data Structure and Algorithm Practicum
Graph

Name
Dicha Zelianivan Arkana

NIM
2241720002

Class
1i

Department
Information Technology

Study Program
D4 Informatics Engineering

1 Questions

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

   - A* Algorithm: Path finding in video game
   - Dijkstra Algorithm: Google maps route finding
   - Greedy Algorithm: Simple form of path finding algorithm

2. In class Graph, there is an array with LinkedList data type `LinkedList list[]`. What's the aim of this?

   This is used to keep the adjacency array so that we can traverse the graph.

3. What is the reason of calling method `addFirst()` to add data, instead of calling other add methods in LinkedList when using method `addEdge` in class Graph?

   Because we want to add the node in the beginning of the list instead of at the end of the list. The adjacency list starts from the beginning instead of the last.

4. How do we detect prev pointer when we are about to remove an edge of a graph?

   We don't need to handle that manually because the linked list class already does that for us by checking if the previous pointer is null or not.

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?

   mbuh

2 Questions

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

   Undirected graph will have the same amount of in-degree and out-degree while the directed graph usually have different amount of in-degree and out-degree. This is because in directed graph there will be a directed edge between two nodes moving in one direction, causing the adjacency list to have a different amount of in-degree and out-degree.

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?

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

   Because we want to use a 1-based index for the adjacency matrix instead of a 0-based index.

3. What is the use of `getEdge()` method?

   To get the edge between two nodes in the graph.

4. What kind of graph were implemented on practicum 1.3?

   Directed graph

5. Why does the main method use try-catch Exception?

   Because there is a method in the graph class that will throw an exception if the input is not valid.

3 Assignments

1. Convert the path in 1.2 as an input!

   ```
   public static void main(String[] args) throws Exception {
       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);
   }
   ```

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

```java
// returns true when it's directed
public boolean graphType() throws Exception {
    int totalIn = 0, totalOut = 0;
    for (int source = 0; source < vertex; source++) {
        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++) {
                if (list[source].get(j) == i) {
                    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!

```java
public void removeEdge(int source, int destination) throws Exception {
    int destinationIndex = list[source].search(destination);
    int sourceIndex = list[destination].search(source);
    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, etc. Then the next will be in form of region name, like Malang, Surabaya, Gresik, Bandung, etc.

   - **Generic version of DoubleLinkedList**

```
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;
        }
    }
```

- **Generic version of Graph**

```
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) {
        // setup list
        list.putIfAbsent(source, new DoubleLinkedList<>());
        list.putIfAbsent(destination, new DoubleLinkedList<>());

        // add edge
        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());

        // degree undirected graph
        // in-degree
        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))
    }

    // remove in every adjacency array entries
    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();
    }

    // returns true when it's directed
    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 {
        Scanner sc = new Scanner(System.in);

        System.out.print("Insert vertex amount: ");

        int vertexCount = sc.nextInt();
        Graph<Integer> graph = new Graph<>(vertexCount);

        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());
    }
}
```