Week 15

Subject	Data Structure and Algorithm
Lecturer	Imam Fahrur Rozi ST. MT.
Type	Assignment
Semester	Semester 2
Time	@June 15, 2023
Files & Media	

Jobsheet 15

quesitons 1

1. BFS (breadth-first search), to determine the shortest paths and minimum spanning trees

   DFS (depth-first search), used to find between two vertices, detect cycles in a graph

   shortest path, used to find directions to travel from one location to another in mapping software

2. the aim for `list[]` is that to keep the adjacency array, so we can traverse the graph
3. by using **`addFirst()`**, we will add the node in the beginning of the list instead on the end of the list
4. we don’t need to handle it manually, because the linked list class already does that for us by checking if the previous pointer is null or not
5. uhhhh the question is inunderstando me not know what do that mean

questions 2

1. undirected graph have the same amount of indegree and outdegree while the directed graph usually have different amount of indegree and outdegree. directed graph will be a directed edge between two nodes moving in one direction, causing the adjacency list to have a different amount of indegree and outdegree
2. because we want to use a 1-based index for the adjacency matrix instead of 0-based index
3. to get the edge between two nodes
4. directed graph
5. because there is a method in the graph class that will throw an exception if the input is not valid

assignments

1. code

```
package JB15.Prac;

import java.util.Scanner;

public class Main
{
    static Scanner sc = new Scanner(System.in);
    public static void main(String[] args) throws Exception
    {
        System.out.print("Insert amount of Vertices: ");
        int input = sc.nextInt();
        Graph graph = new Graph(input);
//        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();
        int menu, source, destination;
        do
        {
            System.out.println("1. Add Edge");
            System.out.println("2. Remove Edge");
            System.out.println("3. Print Graph");
            System.out.println("4. Degree");
            System.out.println("5. Remove All Edges");
            System.out.println("6. Exit");
            menu = sc.nextInt();
            switch (menu) {
                case 1:
                    System.out.println("Add Edge");
                    System.out.print("Source: ");
                    source = sc.nextInt();
                    System.out.print("Destination: ");
                    destination = sc.nextInt();
                    graph.addEdge(source, destination);
                    break;
                case 2:
                    System.out.println("Remove Edge");
                    System.out.print("Source: ");
                    source = sc.nextInt();
                    System.out.print("Destination: ");
                    destination = sc.nextInt();
                    graph.removeEdge(source, destination);
                    break;
                case 3:
                    System.out.println("Graph");
                    graph.printGraph();
                    break;
                case 4:
                    System.out.print("Degree: ");
                    graph.degree(sc.nextInt());
                    break;
                case 5:
                    graph.removeAllEdges();
                    System.out.println("All Edges are removed");
                    break;
                case 0:
                    System.out.println("Thank you for using");
                    System.exit(0);
                    break;
                default:
                    System.out.println("Please select menu correctly!");
            }
        }
        while (menu != 0);
    }
}
```

2. code

```
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 (list.get(key).get(j) == key) totalIn++;
            }
            for (int j = 0; j < list.get(key).size(); j++)
            {
                if (list.get(key).get(j) == key) totalOut++;
            }
        }
        return (totalIn != totalOut);
    }
```

3. code

```
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);
    }
```

4. generic version of `DoubleLinkedList`

```
package JB15.Asg;

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<TData>(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, can't 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, can't 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

```
package JB15.Asg;

import java.util.HashMap;
import java.util.Objects;

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

    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);
    }

    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));
    }

    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);
    }

    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();
    }

    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 (list.get(key).get(j) == key) totalIn++;
            }
            for (int j = 0; j < list.get(key).size(); j++)
            {
                if (list.get(key).get(j) == key) totalOut++;
            }
        }
        return (totalIn != totalOut);
    }
}
```