CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

NON-LINEAR DATA STRUCTURES AND ALGORITHMS.

ASSIGNMENT 1: Non-Linear Data Structures: Trees & Graphs

Part 1 : Operations on Binary Search Trees. **(Week 4)**

1. Background.

The Abstract Data Type (ADT) myBinarySearchTrees<T1, T2> allows to store nodes with the format (T1 key, T2 value). Whereas T2 can be any datatype (e.g., an Integer, a String, a myPlayer, etc.), the datatype T1 is constrained to have a total order relationship ≤ (i.e., two elements T1 elem_i and T1 elem_j can be compared and sorted).

The ADT myBinarySearchTrees<T1, T2> supports the set of operations that we have seen in the lectures, and it is specified in the interface myBinarySearchTree.java.

1. //public myBinarySearchTree<T1, T2> create_from_binary_search_node(
   myBinarySearchNode<T1, T2> n);

2. public boolean my_is_empty();
3. public myBinarySearchNode<T1, T2> my_root();
4. public myBinarySearchTree<T1, T2> my_left_tree() throws myException;
5. public myBinarySearchTree<T1, T2> my_right_tree() throws myException;

6. public myBinarySearchNode<T1,T2> my_find(T1 key);
7. public myBinarySearchTree<T1, T2> my_insert(T1 key, T2 info);
8. public myBinarySearchTree<T1, T2> my_remove(T1 key);

9. public int my_length();
10. public int my_node_count();
11. public int my_leaf_count();
12. public myList<T2> my_inorder();
13. public myList<T2> my_preorder();
14. public myList<T2> my_postorder();
15. public myBinarySearchNode<T1, T2> my_maximum() throws myException;
16. public myBinarySearchNode<T1, T2> my_minimum() throws myException;

All the operations are implemented in the class **myBinarySearchTreeImpl.java.**

- myMain.java: This class tests the functionality of the trees. Please, have a detailed look and also run it to test if your code is right. The first thing that you must do, is to **draw in your notebook** the trees that are defined in this file.

2. Goal of the Assignment.

In this assignment the ADT myBinarySearchTrees<T1, T2> has been extended with 4 new operations:

17. public int my_count_at_level(int level);
    This operation receives as an input the level of the tree we are looking for, and returns the amount of nodes placed on that level.

18. public boolean my_is_balanced();
    A binary tree is balanced when the length of its two subtrees (left subtree and right subtree) do not differ in more than 1 unit, and the proper left subtree and right subtree are balanced trees as well).
    This operation returns if the tree is balanced or not.

19. public int my_count_smaller_nodes(T1 key);
    This operation receives as an input a key, and returns the amount of nodes in the tree with smaller key values.

20. public int my_find_node_at_level(T1 key);
    This operation receives as an input a key, and returns the level were the node is located.

Exercise: Implement the 4 methods in the class **myBinarySearchTreeImpl.java** using recursion. *Note that the methods that have a key as input, must use the advantages of binary search trees by exploring only the right subtree or the left subtree in case that the other subtree can be discarded. Note: You can reuse any of the other 16 defined operations if you want.*

TIP: Look for the comments "TO-DO" in the code. Where you find a "TO-DO", you have to implement yourself this part of the program.

Part 2 & Part 3 : ADT Graph (Week 5 & Week 6)

BACKGROUND.

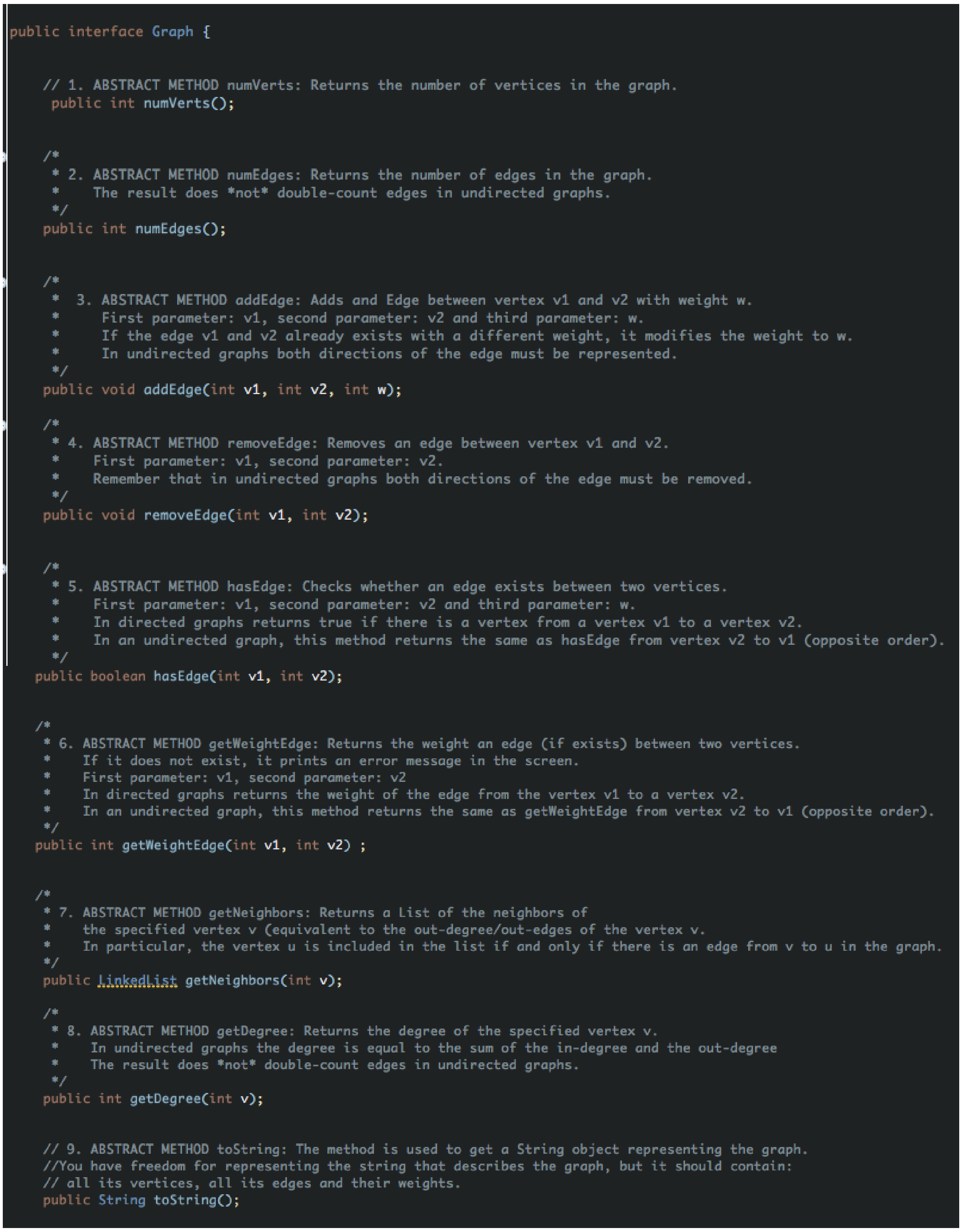
In the lectures we have already studied the ADT Graph, with its characteristics, representations types, etc. Please, check the slides of the lectures regarding graphs. In this assignment, you will have to specify the ADT Graph with the following characteristics:

- Encompassing directed and undirected weighted graphs with integer weights.
- The vertices are represented by natural numbers (e.g. 0,1,2,3...), and remain the same after the creation of an instance of a Graph. Therefore, no vertex can be added or removed after the creation of the graph instance.
- The same occurs with the characteristic of being directed or not. A graph cannot change this characteristic after its creation.
- Self-loops are allowed.

This assignment consists in two parts, corresponding to the two implementations of the interface of the ADT Graph (see Figure 1): with Adjacency matrix and with Adjacency Lists. Please, check the slides of the lectures regarding these two representations of graphs.

Furthermore, you should pay special attention to:

- Among others, as attributes of the class you should define an attribute for keeping the count of the edges that currently your graph contains. Also, another attribute that specifies if the graph is directed or the opposite, undirected.
- Any method that takes one or more vertex IDs as arguments should print an error message if any input ID is out of bounds.
- Check the slides of the lecture of graphs. Many of the doubts that you might have are explained in the Graphs representation section.



```java
public interface Graph {

    // 1. ABSTRACT METHOD numVerts: Returns the number of vertices in the graph.
     public int numVerts();

    /*
     * 2. ABSTRACT METHOD numEdges: Returns the number of edges in the graph.
     *    The result does *not* double-count edges in undirected graphs.
     */
    public int numEdges();

    /*
     *  3. ABSTRACT METHOD addEdge: Adds and Edge between vertex v1 and v2 with weight w.
     *     First parameter: v1, second parameter: v2 and third parameter: w.
     *     If the edge v1 and v2 already exists with a different weight, it modifies the weight to w.
     *     In undirected graphs both directions of the edge must be represented.
     */
    public void addEdge(int v1, int v2, int w);

    /*
     * 4. ABSTRACT METHOD removeEdge: Removes an edge between vertex v1 and v2.
     *    First parameter: v1, second parameter: v2.
     *    Remember that in undirected graphs both directions of the edge must be removed.
     */
    public void removeEdge(int v1, int v2);

    /*
     * 5. ABSTRACT METHOD hasEdge: Checks whether an edge exists between two vertices.
     *    First parameter: v1, second parameter: v2 and third parameter: w.
     *    In directed graphs returns true if there is a vertex from a vertex v1 to a vertex v2.
     *    In an undirected graph, this method returns the same as hasEdge from vertex v2 to v1 (opposite order).
     */
   public boolean hasEdge(int v1, int v2);

   /*
    * 6. ABSTRACT METHOD getWeightEdge: Returns the weight an edge (if exists) between two vertices.
    *    If it does not exist, it prints an error message in the screen.
    *    First parameter: v1, second parameter: v2
    *    In directed graphs returns the weight of the edge from the vertex v1 to a vertex v2.
    *    In an undirected graph, this method returns the same as getWeightEdge from vertex v2 to v1 (opposite order).
    */
   public int getWeightEdge(int v1, int v2) ;

   /*
    * 7. ABSTRACT METHOD getNeighbors: Returns a List of the neighbors of
    *    the specified vertex v (equivalent to the out-degree/out-edges of the vertex v.
    *    In particular, the vertex u is included in the list if and only if there is an edge from v to u in the graph.
    */
    public LinkedList getNeighbors(int v);

    /*
     * 8. ABSTRACT METHOD getDegree: Returns the degree of the specified vertex v.
     *    In undirected graphs the degree is equal to the sum of the in-degree and the out-degree
     *    The result does *not* double-count edges in undirected graphs.
     */
    public int getDegree(int v);

    // 9. ABSTRACT METHOD toString: The method is used to get a String object representing the graph.
    //You have freedom for representing the string that describes the graph, but it should contain:
    // all its vertices, all its edges and their weights.
    public String toString();
```

Figure 1.- interface of the ADTGraph: Graph.java

The folder **/src** contains the following files:

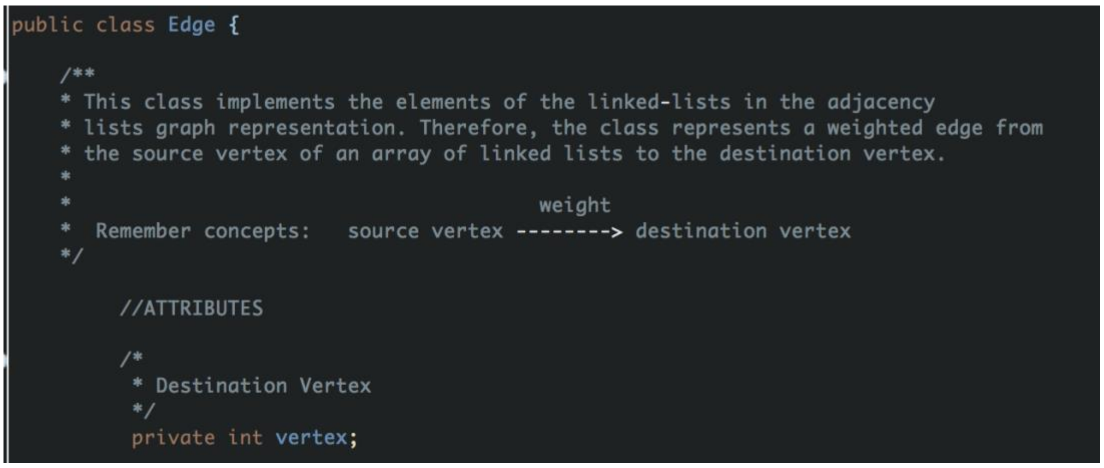
- **<u>MainGraphs.java:</u>** This class tests the functionality of the graphs. Please, have a detailed look and also run it to test if your code is right. The first thing that you must do, is to **draw in your notebook** the graphs that are defined in this file. Also, you must draw in the notebook the corresponding adjacency matrix representation and adjacency lists representation. Both for directed and undirected graphs. Therefore, at the end you have to test the 4 scenarios of mainGraphs.java.

- **Graph.java:** This interface specifies the ADT Graph. Check this file before implementing the methods, since all the functionalities that your ADT Graph must have, are explained in this file.
- **GraphAdjMatrix.java:** This class implements all the operations of the ADT Graph by representing it as an adjacency matrix. Therefore, it uses as data structure a 2D array.
- **GraphAdjList.java:** This class implements all the operations of the ADT Graph by representing it as an adjacency list. Therefore, it uses as data structure an array of linked lists. It has one linked-list for every vertex i of the graph. The linked-list of a vertex i contains objects of type Edge (this class is already implemented), representing the adjacent/neighbour vertices of the vertex i. For directed graphs, there is an edge from vertex i to vertex j then there must be an object of type Edge representing such relationship in the adjacency list of node i. For undirected graphs, if there is an edge between vertex i and vertex j, then must be two objects of type Edge representing such relationship in the adjacency lists of ***both*** vertex i and vertex j. The edges are not sorted; they contain the adjacent nodes in ***order*** of edge insertion.

  TIP: You can use the Java Library LinkedList. Read the documentation of this library or any other that you might need in the future from the following website https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html. In Eclipse, press "." after creating an object for seeing all the methods implemented.

- **Edge.java:** This class implements elements of the linked-lists in the adjacency lists graph representation. Therefore, the class represents a weighted edge from the source vertex of an array of linked lists to the destination vertex. It contains two attributes: the vertex and the weight of the corresponding edge. See Figure 2.

```
                                          weight
Remember concepts:   source vertex --------> destination vertex
```

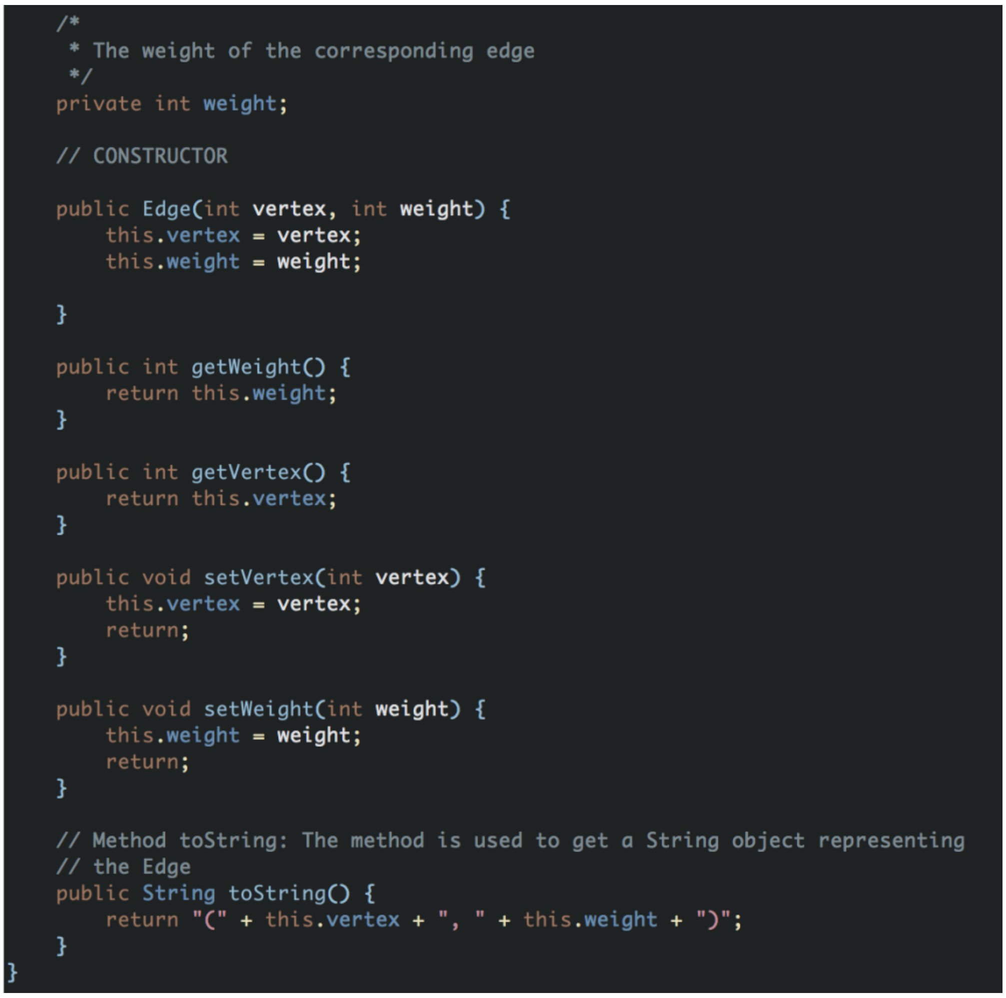


```
public class Edge {

    /**
    * This class implements the elements of the linked-lists in the adjacency
    * lists graph representation. Therefore, the class represents a weighted edge from
    * the source vertex of an array of linked lists to the destination vertex.
    *
    *                                        weight
    *  Remember concepts:   source vertex --------> destination vertex
    */

        //ATTRIBUTES

        /*
         * Destination Vertex
         */
         private int vertex;
```



```java
    /*
     * The weight of the corresponding edge
     */
    private int weight;

    // CONSTRUCTOR

    public Edge(int vertex, int weight) {
        this.vertex = vertex;
        this.weight = weight;

    }

    public int getWeight() {
        return this.weight;
    }

    public int getVertex() {
        return this.vertex;
    }

    public void setVertex(int vertex) {
        this.vertex = vertex;
        return;
    }

    public void setWeight(int weight) {
        this.weight = weight;
        return;
    }

    // Method toString: The method is used to get a String object representing
    // the Edge
    public String toString() {
        return "(" + this.vertex + ", " + this.weight + ")";
    }
}
```

Figure 2.- Edge.java

IMPORTANT: You can not use ArrayLists in this assignment. If you do not use the class Edge for the Adjacency List, it means that you are not implementing properly the Adjacency list, therefore you will get 0 marks in part 3.

Part 2: Adjacency Matrix (Week 5)

Adjacency Matrix Implementation of the ADT Graph: GraphAdjMatrix.java

Part 3-EXTRA: Adjacency List (Week 6)

Adjacency Lists implementation of the ADT Graph: **GraphAdjList.java**

TIP: Look for the comments "TO-DO" in the code. Where you find a "TO-DO", you have to implement yourself this part of the program.

MARK BREAKDOWN.

Assignment 1: 25 marks.

Part 1: (11 marks)

1. public int my_count_at_level(int level) → 2.5 marks
2. public boolean my_is_balanced()→ 2.5 marks
3. public int my_count_smaller_nodes(T1 key) → 3 marks
4. public int my_find_node_at_level(T1 key) → 3 marks

Part 2: (7 marks)

1. attributes, constructor and toString → 1 marks
2. addEdge, removeEdge → 2 marks
3. hasEdge and getWeightEdge → 2 marks
4. getNeighbours and getDegree→ 2 marks

Part 3- EXTRA: (7 marks)

1. attributes, constructor and toString → 1 marks
2. addEdge, removeEdge → 2 marks
3. hasEdge and getWeightEdge → 2 marks
4. getNeighbours and getDegree→ 2 marks

To evaluate the assignment I will run it over some tests (do not forget to run the main files and check that the outputs are correct!). Remember that for graphs, the tests involve directed and undirected graphs and the methods should work properly for both types of graphs. Remember also that for trees the methods should take advantage of the efficiency that binary search trees provide in terms of ordering. And, as always, remember to print the error messages. The results are based on the performance of your code over these tests.

For each function, there are 4 possible scenarios:

A. The function passes all the test, it is efficient and prints error messages → 100% of marks.
B. The function does not pass all tests by small/medium mistakes→ 90% of marks – 10% marks (depending how small is the mistake).
C. The function does not pass all tests by big mistakes, it does not compile or it generates and exception (makes the program crash) or the function was not attempted → 0% of marks.
D. The function works well but you do not pass the demo → 5% of marks.

IMPORTANT: I will use a source code plagiarism detection tool. In case of detecting a copy (for at least one method) between some students, all these students get 0% marks for the whole assignment. Including the student that originally coded it.

SUBMISSION DETAILS.

Deadline.

Sunday 11:30.pm of week 6.

Submission Details.

Please upload to Blackboard **ONLY** the following files:

- **myBinarySearchTreeImpl.java**
- **GraphAdjMatrix.java**
- **GraphAdjList.java**

IMPORTANT: Do not ZIP the files.

Lab Demo.

A brief individual interview about the assignment will take place on our lab session on the lab of week 7. **The demo is mandatory for the assignment to be evaluated.**
Please, let me know in the lab if you are willing to do the demo earlier. (With the corresponding uploading of the full assignment to the blackboard). Once the demo is done, there will be no possibility of re-evaluating later. The evaluation will be done with the exact code presented at this moment. Therefore, my advice is to take this option only if you are 100% sure that your assignment is already completed.

If you cannot attend (for important reasons) to the demo, you must talk with me and we will do another appointment to the demo no later than scheduled demo.