Assessed Exercise Week 5
The exercise has 2 assessed tasks: Task A and Task B

Task A

Implement all methods up to the method `public void AddEdge(T from, T to)` (included). We will use two classes, *GraphNode* and *Graph*. Use the completed classes to implement an application with GUI (Windows Forms Application) which allow:
- to insert a node into a graph (given the ID which you can assume to be a *char*)
- to insert a directed edge between two nodes (given their IDs)
- to display the total number of nodes in the graph and the total number of edges in the graph.

Before you start to complete the task please read the "hints to complete the task" below.

Open a new project (console application for C#) called **Graph**

Add two new generic classes **GraphNode** and **Graph**.

Starting codes for Generic Class: GraphNode

```
public class GraphNode<T>
    {
       private T id; // data stored in the node ("id" of the  node).
       private LinkedList<T> adjList; // adjacency list of the node
                                        //Use LinkedList from C#

        // constructor
        public GraphNode(T id)
         {
          this.id = id;
          adjList = new LinkedList<T>();
         }

        // set and get the data stored in the node
        public T ID
         {
          // to be completed
         }


        //add a directed edge from "this" node to the node "to"
        public void AddEdge(GraphNode<T> to)
         {
           //to be completed.
          //hint: add the id of node "to" to the adjacency list
         }

        // returns the adjacent list of the node
        public LinkedList<T> GetAdjList()
         {
          //to be completed.
         // hint: returns the adjacency list
         }
      }
```

Starting codes for Generic Class: Graph

```
public class Graph<T> where T : IComparable
      {

         // list of all the nodes in the graph. Use LinkedList from C#
         private LinkedList<GraphNode<T>> nodes;


         // constructor – set nodes to new empty list
          public Graph()
          {
            nodes = new LinkedList<GraphNode<T>>();
          }


         // only returns true if the graph's list of nodes is empty
         public bool IsEmptyGraph()
         {
           // to be completed
         }


         // returns the total number of nodes present in the graph
         public int NumNodesGraph()
         {
           // to be completed

           // hint: how can you get the list of all nodes in the graph ?
           // once you have that you can count (and return) how many elements are
        in the list

         }


         // returns the total number of edges present in the graph
         public int NumEdgesGraph()
         {

           // to be completed


          foreach (GraphNode<T> n in nodes)
           {
           // hint: this loop allows to run over all the nodes present in the
        graph.

          // to get the total number of edges in the graph:
          // you need to count how many outgoing edges each node has and then
        return the sum obtained considering all nodes in the graph

           // how can you get the number of outgoing edges for each node ? (i.e.,
        use the adjacency list of the node)
          }



         }
```

```
            // only returns true if node is present in the graph
            public bool ContainsGraph(GraphNode<T> node)
             {
             // to be completed. Hint: Search through the full list of nodes (search
of the node is based on the id)

              if (n.ID.CompareTo(node.ID) == 0)
                  return true;

             }


            // only returns true if nodes “from “ and “to” are adjacent

             public bool IsAdjacent(GraphNode<T> from, GraphNode<T> to)
             {
             // to be completed
            //Hint: Find the node “from” in the list of nodes and then search its
            adjList to see if there is node “to”

             }


             // add a new node (with this “id”) to the list of nodes of the graph
             public void AddNode(T id)
             {
               // to be completed

             }

             //returns the node with this id
            public GraphNode<T> GetNodeByID(T id)
            {
              // to be completed
              //Hint: Search through the list of nodes for a node with this id
            }



          // Add a directed edge between the node with id "from" and the node with id
          “to”
            public void AddEdge(T from, T to)
            {
              // to be completed
             //Hint: Find the node with id “from” in the list of nodes and then
             //use the GraphNode method to add an edge to the node with id “to”
            }


          } //end class
```

Hints to complete the task: On Moodle you can find videos (in “Graphs”) that discuss the basic ideas (graphs, adjacency list and adjacency matrix) necessary to solve this task. These should be sufficient to complete this task. Notice that in this implementation the graph is represented using an adjacency list implemented using the built-in LinkedList from C#.
We are also going to discuss and implement many of these methods in the webinar – you can also refer to the slides of the webinar and to the recorded webinar A on Graphs when you want to complete the methods.

Task B

Implement an application in C# (with a user interface) which allows users to

- Insert an airport by adding its code (assume the airport code is a *string*, see Figure 1 below)
- Insert / remove a direct connection (i.e., direct flight) between two airports given their codes (see Figure 1 below)
- Display the codes of the airports that can be reached from a *starting airport* by using **direct or connecting** flights (allow the user to insert the starting airport).
- Display the codes of the airports from which is possible to reach **all** the other airports by using **direct or connecting** flights.

The user interface should be a GUI (use a Windows Forms application). As one of the tests to show in your screencast, use the application you have implemented to identify the airports from which is possible to reach all the others in Figure 1 (below) using direct or connecting flights.

Hints to complete the task: In the recorded materials on Moodle (in "Graphs") you can find videos on graph traversals (DFS, BFS) and mother vertex which are the concepts needed to solve this task. We will cover graph traversals in the webinars so please consider also the webinar slides and the recorded webinar when completing this task. To implement the application you will need to use and extend the classes Graph and GraphNode implemented in TaskA.
For the removal of a direct connection (i.e., directed edge), see how you have done the AddEdge method (T from, T to); the removal should be quite similar.
For the third point ("Display the codes of the airports that can be reached…") use the idea of graph traversal (implement either DFS or BFS, see the starting code below, that you need to add in the class Graph).
*For the last point ("*Display the codes of the airports *from ..") use the idea of mother vertex of a graph (look at the corresponding video on Moodle in "Graphs"). You can add in the class Graph a method which returns the id's of the nodes that are mother vertices of the graph (see the starting code below). To get the mother vertices of a graph use one of the two graph traversals that you have implemented.*

```
//Perform a DFS traversal starting at the node with id "startID"
//leaving a list of visited id's in the visited list.

            public void DepthFirstTraverse(T startID, ref List<T> visited)
            {
             LinkedList<T> adj;
             Stack<T> toVisit = new Stack<T>();

             GraphNode<T> current  = new GraphNode<T>(startID);

             toVisit.Push(startID);

             while (toVisit.Count != 0)
                {
              //to be completed. Hint: get current node to the list of visited nodes
             and add its adjacent nodes (only those not already visited) to toVist

                }
              }
```

```
 //Perform a BFS traversal starting at the node with id "startID"
//leaving a list of visited id's in the visited list.

              public void BreadthFirstTraverse(T startID, ref List<T> visited)
              {
               LinkedList<T> adj;
               Queue<T> toVisit = new Queue<T>();
               GraphNode<T> current = new GraphNode<T>(startID);

               toVisit.Enqueue(startID);

               while (toVisit.Count != 0)
                 {
               //to be completed. Hint: get current node to the list of visited nodes
              and add its adjacent nodes (only those not already visited) to toVisit
                 }
               }

// Return a list which contains the id's of the nodes that are the mother vertices of
the graph. For example, return the list ['A',C','B'] if the nodes with id's = A, C and
B are mother vertices. Hint: use the BFS (or DFS) visit to implement the method

             public List<T> mothervertex()
               {
               //to be completed.
               }
```

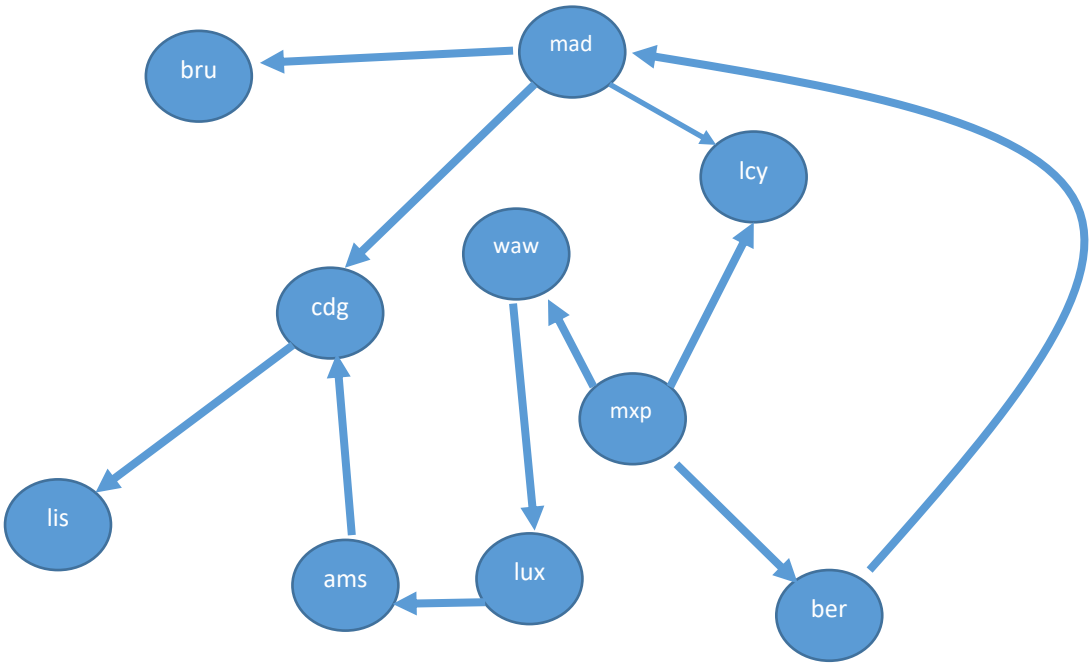

Figure 1

*The graph shows a set of airports (identified by their **code**, which is a string) and the established **direct** connections (i.e., direct flights). A direct connection (i.e., direct flight) between the two airports is denoted by a **directed** edge. e.g., there is a direct flight going from cdg to lis, but there is no direct flight going from lis to cdg.*
Airport codes (cgd, lis, etc..) are strings. You can see that in the shown Figure from most airports is not possible to reach all the others even by using connecting flights. For example, starting from lux (i.e., starting airport) one can only reach (by direct or connecting flights) ams, cdg, lis. As another example, starting from waw one can reach (by direct or connecting flights) lux, ams, cdg, lis.