Practical Task 8

Due date: 09:00pm Friday, September 15, 2018

(Very strict, late submission is not allowed)

General Instructions

1. You may need to read Chapter 11 of SIT232 Workbook available in Resources → Course materials → SIT232 Workbook to remind / study Generics and their application in object-oriented programming. You may also need to read Chapter 6 of SIT232 Workbook about Polymorphism and Interfaces as you need excellent understanding of these topics to progress well through the practical tasks of the course. Make sure you are proficient with them as they form a basis to design and develop programming modules in this and all the subsequent tasks. You may find other important topics required to complete the task, like exceptions handling, in other chapters of the workbook.
2. Read Sections 14.1 and 14.2.2 of Chapter 14 of the SIT221 course book "Data structures and algorithms in Java" (2014) by M. Goodrich, R. Tamassia, and M. Goldwasser. You may access the book on-line for free from the reading list application in CloudDeakin available in Resources → Course materials → Course Book: Data structures and algorithms in Java. Furthermore, you may explore chapters 10 of the SIT221 Workbook available in CloudDeakin in Resources → Course materials → SIT221 Workbook to learn some other implementation issues of a graph.
3. This time, your task is to implement a graph structure that can be used to represent relationships that may exist between pairs of objects. A graph is a collection of vertices and edges connecting the vertices pairwise. Normally, nodes relate to real-life objects that can be characterized by a generic data type, say T. Edges reflect relationships described by another generic data type, say K. Thus, the expected outcome of this task is a generic class, parametrized by two data types, realizing the adjacency list representation of a graph.
4. Download the zip file called "Practical Tasks 8 (template)" attached to this task. It contains a template for the program that you need to work on. Open the solution file in Microsoft Visual Studio. There are two connected projects inside: DataStructures_Algorithms and Runner. There are currently no files in the Week08 subfolder and the objective of the practical task is to add several new classes and interfaces. There are no constraints on how to design and implement the new modules internally, but their exterior must meet all the requirements of the task in terms of functionality accessible by a user.

   The second project has Runner08_Task1.cs, which is new. It implements IRunner interface and acts as a main method in case of this practical task by means of the Run(string[] args) method. Make sure that you set the right runner class in Program.cs. You may modify this class as you need as its main purpose is to test the computer-based representation of a graph for its correctness. However, it has already a good structure to check the expected methods and classes. Therefore, you should first explore the content of the method and see whether it satisfies your goals in terms of testing. Some lines are commented out to make the current version compilable, so assure that you make them active as you progress with the coding part. In general, you should use this class and its method to thoroughly test your code. Your aim here is to cover all potential logical issues and runtime errors. This (testing) part of the task is as important as writing the code. We will later on replace your runner by our special versions in order to test functionality of your code for common mistakes.

Objectives

1. Learn representation and computer implementation of a graph structure.

Specification

Main task

A graph is a way to represent relationships that exist between pairs of objects. Formally, a graph G is defined as a set V of vertices (also called nodes) and a collection E of edges (also called arcs) connecting the vertices from V pairwise. In this practical task, we deal with representation of a ***directed*** graph, where every edge (u, v) is said to be directed from u to v so that (u, v) is ordered, with u preceding v. The two nodes joined by an edge are called ***endpoints*** of the edge. And if an edge is directed, its first endpoint is its ***origin*** (also called a source) and the other is the ***destination*** (also called a target) of the edge. Two nodes u and v are said to be ***adjacent*** if there is an edge whose end nodes are u and v. An edge is said to be ***incident*** to a node if the node is one of the edge's endpoints. The ***outgoing*** edges of a node are the directed edges whose origin is that node. The ***incoming*** (or ingoing) edges of a node are the directed edges whose destination is that node. The ***degree*** of a node v, denoted as $\mathrm{deg}(v)$, is the number of incident edges of v. The ***in-degree*** and ***out-degree*** of a node v are the number of the incoming and outgoing edges of v, and are denoted as $\mathrm{indeg}(v)$ and $\mathrm{outdeg}(v)$, respectively.

The objective of this practical task is to construct an ***adjacency list***, a complex data structure for representing a graph. It must maintain a **collection** to store the nodes of the graph and, for each node, provide a separate **list** containing those edges that are incident to the node. This organization allows us to efficiently find all edges incident to a given node. The idea of the adjacency list structure consists in supporting direct access to the incident edges (and thus to the adjacent nodes) of each node. Specifically, in the case of a directed graph, it maintains **two collections** $\mathrm{out}(v)$ and $\mathrm{in}(v)$ of outgoing and incoming edges incident to v. Thus, we require that the primary structure for an adjacency list maintain the collection V of nodes in a way so that we can locate the secondary structures $\mathrm{out}(v)$ and $\mathrm{in}(v)$ for a given node v in $O(1)$ time. This could be done by using a doubly linked list to represent the set V, with each node maintaining direct references to its $\mathrm{out}(v)$ and $\mathrm{in}(v)$ incidence collections, also implemented via doubly linked lists. The primary benefit of an adjacency list is that the collections $\mathrm{out}(v)$ and $\mathrm{in}(v)$ contain exactly those edges that should be reported by the methods OutgoingEdges(v) and IngoingEdges(v). Therefore, we can implement both methods by iterating the edges of $\mathrm{out}(v)$ in $O(\mathrm{outdeg}(v))$ and that of $\mathrm{in}(v)$ in $O(\mathrm{indeg}(v))$ time. This is the best possible outcome for any graph representation.

The outline of this practical task is as follows. *Firstly*, nodes and edges are seen as building blocks of a graph. In order to hide the details of their implementation from a user, two interfaces must be designed. The first interface called INode<T> defines the exterior of the subsequent Node class. Its goal is to record and retrieve the generic data of type T associated with a particular node. Clearly, this functionality seems limited, which is mainly caused by simplicity and the reduced set of graph-based operations implied by this task. However, in real practice, you will likely find the functionality of this interface to be much broader. A similar interface called IEdge<T,K> is to be developed for the subsequent Edge class characterizing an edge. Its purpose is to get and save the generic data of type K associated with the edge based on two instances of the Node class implementing the INode<T> generic interface. The user can perform reading and writing operations on the data of the edge as well as access the related nodes. *Secondly*, two private classes, designated here as Node and Edge, are to be completed within the main generic class Graph<T,K>. The both classes are internal intentionally as their structure must not be exposed to the user, who can communicate with these entities only by means of instances casted to INode<T> and IEdge<T,K> interfaces. Therefore, all operations on the nodes and edges of the graph are available only through the methods and properties of the Graph<T,K>, which is responsible for their correctness as well as communication with underlying objects.

Task 1. Implementation of the INode<T> and IEdge<T,K> interfaces

Before starting the work, remind that an interface only declares methods and properties stating their signatures and must not implement them. The specific code for each of them must be later on provided by classes implementing the interface. Now, in Week08 subfolder, create a new generic **public** interface called "INode<T>" and define the following property to be accessible by a user.

Data	Property. Gets or sets the data of type T associated with a particular node in a graph.

Subsequently, add to the project another generic **public** interface called "IEdge<T,K>". Similarly, declare the properties as the specification requires below. Note that **From** and **To** are both read-only. This is crucial as users must not be able to alter edges externally to the Graph<T,K> class; that is, they must have no way to manipulate the nodes of an edge. Otherwise, the structure of a graph might be corrupted. Furthermore, **T** and **K** are generally two different data types parametrizing the IEdge<T,K> since the type T corresponds to the data used as part of a node and K relates to the data type relevant to an edge. For example, nodes may designate cities and be represented as strings, while edges can measure distances between the cities and be integers.

Data	Property. Gets or sets the data of type K associated with a particular edge in a graph.
From	Property. Gets the reference of type INode<T> to the source node incident to the current edge implementing the IEdge<T,K>.
To	Property. Gets the reference of type INode<T> to the target node incident to the current edge implementing the IEdge<T,K>.

Task 2. Implementation of the Node and Edge classes

To succeed with this assignment, you must develop a new generic class called “Graph<T, K>”. However, you first have to focus on the node and edge-related data structures implementing them both as generic private classes (so they must be built-in) inside the larger Graph<T,K> class. Therefore, in Week08 subfolder, create the Graph<T,K> class and inside it declare two **private** classes called “Node” and “Edge” implementing the INode<T> and IEdge<T,K> interfaces, respectively. Ensure that they provide specified functionality necessary to support various operations on the Graph<T,K>.

The Node class must meet the following specification.

Node(T data)	Initializes a new instance of the Node class implementing the INode<T> interface and records the supplied data. The both doubly linked list based collections of ingoing and outgoing edges are initialized as empty. This constructor is an $O(1)$ operation.
Data	Property. Gets or sets the data of type T associated with the current Node. This property results from the INode<T> interface.
ListNode	Property. Gets or sets the reference to the doubly linked list node of type LinkedListNode<Node>, which stores the current instance of the Node class. Note that the LinkedListNode<TYPE> is a built-in class of .NET Framework.
Edge[] GetOutgoingEdges()	Retrieves a collection of outgoing edges incident to the current Node object. If the out-degree of the node is zero, the resulting array is empty (not null). This method is an $O(\mathrm{n})$ operation, where n is the out-degree of the node.
Edge[] GetIngoingEdges()	Retrieves a collection of ingoing edges incident to the current Node object. If the in-degree of the node is zero, the resulting array is empty (not null). This method is an $O(\mathrm{n})$ operation, where n is the in-degree of the node.
void AddOutgoingEdge(Edge edge)	Adds a new edge to the collection of outgoing edges for which the current node serves as a source. This method is an $O(1)$ operation.
void AddIngoingEdge(Edge edge)	Adds a new edge to the collection of ingoing edges for which the current node serves as a target. This method is an $O(1)$ operation.
void DeleteOutgoingEdge(Edge edge)	Removes the specified edge from the collection of outgoing edges associated with the current Node. This method is an $O(1)$ operation.
void DeleteIngoingEdge(Edge edge)	Removes the specified edge from the collection of ingoing edges associated with the current Node. This method is an $O(1)$ operation.
Edge Find(Node to)	Finds and returns the directed edge from the collection of outgoing edges such that the edge originates in the current Node and ends in the node specified as ‘to’. If the edge does not exist, the method returns null. This method is an $O(\mathrm{n})$ operation, where n is the out-degree of the node.
string ToString()	Returns a string that represents the current Node. ToString() is the major formatting method in the .NET Framework. It converts an object to its string representation so that it is suitable for display.

It is important to understand that every instance of the Node class will be a part of an internal doubly linked list managed by the Graph<T,K>. Thus, every graph node constitutes a node of the doubly linked list realized in this assignment via the built-in .NET Framework class LinkedList<TYPE>, where TYPE is actually replaced by the Node. Therefore, from the perspective of runtime complexity, it is crucial that nodes are aware of the doubly linked list's nodes that hold them. This mechanism is required to support $O(1)$ time deletion of a node from a graph. To achieve this, we must have a reference to the LinkedListNode<Node> associated with the graph's Node. Because of this, the Node class contains the ListNode property that must be updated by the Graph<T,K> once the node has been attached to the underlying LinkedList<Node> of the Graph<T,K>. Furthermore, note that some methods of the LinkedList<TYPE> .NET Framework class, such as AddFirst() or AddLast(), actually return LinkedListNode<TYPE> references as an output. You should then only assign the reference to the relevant variable (property).

Now, proceed with the Edge class and ensure that it meets the following functionality requirements.

Edge(Node from, Node to, K data)	Initializes a new instance of the Edge class implementing the IEdge<T,K> interface and records the supplied data. It saves the specified reference to the source (target) node named as 'from' ("to"). This constructor is an $O(1)$ operation.
From	Property. Gets the reference to the source node implementing the INode<T> that is incident to the current Edge implementing the IEdge<T,K> interface. This property results from the IEdge<T,K>.
To	Property. Gets the reference to the target node implementing the INode<T> that is incident to the current Edge implementing the IEdge<T,K> interface. This property results from the IEdge<T,K>.
Data	Property. Gets or sets the value of type K as a data associated with the current Edge. This property results from the IEdge<T,K>.
LinkedListNode<Edge> ListNodeFrom	Property. Gets or sets the reference to the doubly linked list node of type LinkedListNode<Edge> that stores the current instance of the Edge class as an outgoing edge.
LinkedListNode<Edge> ListNodeTo	Property. Gets or sets the reference to the doubly linked list node of type LinkedListNode<Edge> that stores the current instance of the Edge class as an ingoing edge.
string ToString()	Returns a string that represents the current Edge. ToString() is the major formatting method in the .NET Framework. It converts an object to its string representation so that it is suitable for display.

Similarly to nodes in the Graph<T,K>, edges are stored as elements of a doubly linked list. However, such doubly linked lists are maintained by separate nodes rather the unique Graph<T,K>. In this sense, one may see the graph structure as a collection of nodes, each node owns two affiliated collections: one to store the ingoing and another to store the outgoing edges. Both Node and Edge type collections in the scope of this practical task should be LinkedList<TYPE> built-in .NET Framework class objects. Again, to support $O(1)$ time edge deletion, every edge must refer to the LinkedListNode<Edge> that actually stores the edge while being a part of one of the doubly linked lists. If the edge originates in node 'from', then its ListNodeFrom must keep the reference to the LinkedListNode<Edge> in the doubly linked list that represents the outgoing edges of node 'from'. Clearly, this doubly linked list must belong to 'from'. Likewise, if an edge ends in node 'to', then its ListNodeTo property must refer to the LinkedListNode<Edge> in the doubly linked list that represents the ingoing edges of node 'to'. Then this doubly linked list belongs to 'to'. The AddOutgoingEdge(Edge edge) and AddIngoingEdge(Edge edge) methods of the Node must update the references of the given edge accordingly once the edge is attached to one of its incidence collections. Obviously, the properties are to be used in DeleteOutgoingEdge(Edge edge) and DeleteIngoingEdge(Edge edge) methods carried out by the Node.

Task 3. Implementation of the Graph class

Finally, complete the Graph<T,K> class meeting the following requirements.

int NumVertices	Property. Gets the number of nodes contained in the Graph<T,K>. Retrieving the value of this property is an $O(1)$ operation.
int NumEdges	Property. Gets the number of edges contained in the Graph<T,K>. Retrieving the value of this property is an $O(1)$ operation.
INode<T> AddNode(T data)	Adds a new node containing the specified data to the Graph<T,K>. Returns the reference of type INode<T> pointing to the newly added node. This method is an $O(1)$ operation.
IEdge<T, K> AddEdge(INode<T> from, INode<T> to, K data)	Adds a new edge containing the specified data and connecting the two given nodes in the Graph<T,K>. The node named as 'from' ('to') serves as a source (target) of the edge. Returns the reference of type IEdge<T,K> pointing to the newly added edge. This method is an $O(1)$ operation.
void DeleteNode(INode<T> node)	Removes the specified node and all the incident edges from the Graph<T,K>. This method is an $O(\mathrm{n})$ operation, where n is the degree (the in-degree plus the out-degree) of the node.
void DeleteEdge(INode<T> from, INode<T> to)	Removes the edge specified by the two nodes from the Graph<T,K>. The method assumes that edge starts in 'from' and ends in 'to'. This method is an $O(1)$ operation.
DeleteEdge(IEdge<T, K> edge)	Removes the specified edge from the Graph<T,K>. This method is an $O(1)$ operation.
INode<T>[] GetNodes()	Retrieves a collection of nodes currently presented in the Graph<T,K>. The nodes are casted to the INode<T> interface type. This method is an $O(\mathrm{n})$ operation.
IEdge<T,K>[] GetOutgoingEdges(INode<T> node)	Retrieves a collection of outgoing edges incident to the specified node in the Graph<T,K>. If the out-degree of the node is zero, the resulting array is empty (not null). The edges are casted to the IEdge<T,K> interface type. This method is an $O(\mathrm{n})$ operation, where n is the out-degree of the node.
IEdge<T,K>[] GetIngoingEdges(INode<T> node)	Retrieves a collection of ingoing edges incident to the specified node in the Graph<T,K>. If the indegree of the node is zero, the resulting array is empty (not null). The edges are casted to the IEdge<T,K> interface type. This method is an $O(\mathrm{n})$ operation, where n is the indegree of the node.

IEdge<T, K> Find(INode<T> from, INode<T> to)	Finds the directed edge (casted to IEdge<T,K> interface type) originating in node 'from' and ending in 'to' in the Graph<T,K>. If the edge does not exist, the method returns null. This method performs a linear search; therefore, this method is an $O(\mathrm{n})$ operation, where n is the out-degree of the node.
bool Contains(INode<T> from, INode<T> to)	Determines whether the directed edge originating in node 'from' and ending in 'to' exists in the Graph<T,K>. It returns true if the edge is found; otherwise, false. This method performs a linear search; therefore, this method is an $O(\mathrm{n})$ operation, where n is the out-degree of the node.
string ToString()	Returns a string that represents the current Graph<T,K>. ToString() is the major formatting method in the .NET Framework. It converts an object to its string representation so that it is suitable for display.

Although we have a number of methods in the Graph<T,K>, many of them can be delegated to the similar methods of the underlying Node and Edge classes, and therefore require just a line of code. Note that you may always cast interface-type variables to the instances of particular classes. Therefore, input arguments can be converted to objects to extend functionality accessible within the class methods. For example, INode<T> from, can be casted to Node _from by applying

Node _from = (Node) from.

You may then run any methods that are available for the Node, but are hidden with respect to the INode<T> type variable. Note that the reversed casting from Node to INode<T> type is done automatically as it complies with the principle of polymorphism.

Testing your code

This section displays the printout produced by the given Runner based on the official solution. Note that you will likely get different printout as the ToString() methods written by you might differ from those in the official solution. This is clearly valid and allowed. This printout is given to facilitate testing of your code for potential logical errors. It demonstrates the correct logic rather than the expected printout in terms of text and alignment.

```
:: Execute command: graph.AddNode("Sydney")
:: Execute command: graph.AddNode("Melbourne")
:: Execute command: graph.AddEdge(sydney, melbourne, 878)

Printing the content of the graph...
  --> [ Sydney ]  --> Sydney-Melbourne:878
Sydney-Melbourne:878  --> [ Melbourne ]  -->


:: Execute command: graph.AddNode("Adelaide")
:: Execute command: graph.AddEdge(melbourne, adelaide, 726)
:: Execute command: graph.AddEdge(adelaide, melbourne, 726)

Printing the content of the graph...
  --> [ Sydney ]  --> Sydney-Melbourne:878
Sydney-Melbourne:878 | Adelaide-Melbourne:726   --> [ Melbourne ]   --> Melbourne-
Adelaide:726
Melbourne-Adelaide:726  --> [ Adelaide ]  --> Adelaide-Melbourne:726
```

```
graph.Contains(melbourne, adelaide): correct
graph.Contains(sydney, adelaide): correct
graph.Find(sydney, adelaide): does not exist (correct)
graph.Find(sydney, melbourne): Sydney-Melbourne:878
graph.NumVertices: 3 (correct)
graph.NumEdges: 3 (correct)
:: Execute command: graph.DeleteEdge(sydney, melbourne)

Printing the content of the graph...
  --> [ Sydney ]  -->
Adelaide-Melbourne:726  --> [ Melbourne ]  --> Melbourne-Adelaide:726
Melbourne-Adelaide:726  --> [ Adelaide ]  --> Adelaide-Melbourne:726


graph.NumVertices: 3 (correct)
graph.NumEdges: 2 (correct)
:: Execute command: graph.DeleteEdge(adelaide_melbourne)

Printing the content of the graph...
  --> [ Sydney ]  -->
  --> [ Melbourne ]  --> Melbourne-Adelaide:726
Melbourne-Adelaide:726  --> [ Adelaide ]  -->


:: Execute command: graph.AddEdge(sydney, melbourne, 878)
:: Execute command: graph.AddEdge(melbourne, darwin, 3741)
:: Execute command: graph.AddEdge(melbourne, perth, 3406)
:: Execute command: graph.AddEdge(perth, darwin, 4041)

Printing the content of the graph...
  --> [ Sydney ]  --> Sydney-Melbourne:878
Sydney-Melbourne:878   --> [ Melbourne ]   --> Melbourne-Adelaide:726 | Melbourne-
Darwin:3741 | Melbourne-Perth:3406
Melbourne-Adelaide:726  --> [ Adelaide ]  -->
Melbourne-Darwin:3741 | Perth-Darwin:4041  --> [ Darwin ]  -->
Melbourne-Perth:3406  --> [ Perth ]  --> Perth-Darwin:4041


:: Execute command: graph.DeleteNode(melbourne)

Printing the content of the graph...
  --> [ Sydney ]  -->
  --> [ Adelaide ]  -->
Perth-Darwin:4041  --> [ Darwin ]  -->
  --> [ Perth ]  --> Perth-Darwin:4041


graph.NumVertices: 4 (correct)
graph.NumEdges: 1 (correct)

Printing the nodes of the graph...
Node 0:   --> [ Sydney ]  -->
Node 1:   --> [ Adelaide ]  -->
Node 2: Perth-Darwin:4041  --> [ Darwin ]  -->
Node 3:   --> [ Perth ]  --> Perth-Darwin:4041

Printing the outgoing nodes of Perth...
Edge 0: Perth-Darwin:4041

Printing the ingoing nodes of Darwin...
Edge 0: Perth-Darwin:4041
```

Submission instructions

Submit your program code as an answer to the main task via the CloudDeakin System by the deadline. You should zip your INode.cs, IEdge.cs, and Graph.cs files only and name the zip file as PracticalTask8.zip. Please, **make sure the file names are correct**. You must not submit the whole MS Visual Studio project / solution files.

Marking scheme

You will get 1 mark for outstanding work, 0.5 mark only for reasonable effort, and zero for unsatisfactory work or any compilation errors.