Practical Task 4

Due date: 09:00pm Friday, August 17, 2018

(Very strict, late submission is not allowed)

General Instructions

1. Save your time, read instructions!
2. Make sure that Microsoft Visual Studio 2015 is installed on your machine. Community version can be found on the SIT221 course web-page in CloudDeakin in Resources → Course materials → Visual Studio Community 2015 Installation Package. Your submission will be tested in this environment later by the markers.

   If you use Mac, you may install Visual Studio for Mac from https://e5.onthehub.com/WebStore/ProductsByMajorVersionList.aspx?ws=a8de52ee-a68b-e011-969d-0030487d8897&vsro=8, or use Xamarin Studio.

   If you are an on-campus student, this is your responsibility to check that your program is compilable and runs perfectly not just on your personal machine. You may always test your program in a classroom prior to submission to make sure that everything works properly in the MS Visual Studio 2015 environment. Markers generally have no obligations to work extra to make your program runnable; hence verify your code carefully before submission.
3. 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.
4. This time, your task is to realize a double linked list data structure. To facilitate your learning, study the functionality and content of the LinkedList<T> class existing in Microsoft .NET Framework https://msdn.microsoft.com/en-au/library/he2s3bh7(v=vs.110).aspx. Your objective is to implement a data structure like the LinkedList<T>, which you may also use to collect and store given elements of generic type T. Functionality of your class should in general be similar to the LinkedList<T>. Therefore, when you have some doubts on how particular methods should act, you may always address to the LinkedList<T> class and study its behaviour first. Please, check carefully its constructors, properties, and methods and more importantly read all the remarks and related examples as they explain complexity of particular operations. This knowledge is crucial for you as a programmer.
5. Read Sections 2.2 and 2.3 of Chapter 2 of SIT221 Workbook available in CloudDeakin in Resources → Course materials → SIT221 Workbook to learn about single linked and double linked lists. Alternatively, you may explore chapters 3.2 and 3.4 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.

6. Read Section 4.5 of Chapter 4 of the SIT221 Workbook. You should use this material to get familiar with implementation of IEnumerable and IEnumerator interfaces which allow to apply foreach command to generic data collections. You should further read the article available in https://programmingwithmosh.com/csharp/ienumerable-and-ienumerator/ to understand the Iterator OOP design pattern utilised to organise access to the elements of a collection. This material complements the discussion on the Iterator pattern, which you will find in part 3 of this practical task. Both the chapter of the workbook and the referred article provide examples that may help you to build a mechanism supporting the foreach command for your data structure.
7. Download the zip file called “Practical Task 4 (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 Week04 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 Runner04_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 implementation of the double linked list and related interfaces. However, it has already a good structure to check the double linked list data structure that you are required to realize. 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 make sure you make them active as you progress with implementation of different methods of the data structure. 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 implementation of a Double Linked List

2. Study the Iterator object-oriented programming design pattern

Specification

Main task

Task 1. Implementation of a node data structure

Start this assignment with creating a new generic class representing a node, which, being an atomic data structure itself, serves as a building block for a linked list. A linked list is a linear collection of data elements, whose order is not given by their physical positions in memory, for example like in arrays. Instead, each element points to the next (and the previous) one. It is a data structure consisting of a set of nodes which together represent a sequence. Generally, a node of a double linked list consists of a data record that holds valuable information to be stored and two auxiliary pointers referring to the preceding and succeeding nodes in the ordered sequence of nodes constituting the linked list. The two pointers allow to navigate back and forth between two adjacent nodes. In the case of a simpler single linked list, just one pointer is used to refer to the next node. However, traverse is then possible in one direction only, from the head of the linked list to its end. Furthermore, some other utility attributes can assist with associating a node to one of linked lists presented simultaneously in the runtime environment of a single program. They protect the linked lists from acquiring a node that is already a part of another linked list, and therefore may corrupt the logic and/or integrity of the both data structures.

In the Week04 subfolder, create a new class called “Node” capable to store a value of generic type T, and ensure that it provides the following functionality to a user.

Node(T value)	Initializes a new instance of the Node<T> class, containing the specified value. The Owner, Next, and Previous properties are set to null.
Value	Property. Gets or sets the value of type T contained in the node.
Owner	Property. Gets the DoubleLinkedList<T> that the Node<T> belongs to, or null if the Node<T> is not linked.
Next	Property. Gets a reference to the next node in the DoubleLinkedList<T>, or null if the current node is the last element (property Last) of the DoubleLinkedList<T>.
Previous	Property. Gets a reference to the previous node in the DoubleLinkedList<T>, or null if the current node is the first element (property First) of the DoubleLinkedList<T>.
string ToString()	Returns a string that represents the current Node<T>. 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.

Essentially, Node<T> is an analogue of the LinkedListNode<T> sealed class, which is a part of Microsoft .NET Framework. You may compare their functionality following the link

https://msdn.microsoft.com/en-au/library/ahf4c754(v=vs.110).aspx

Exploring the link, you may find out that the LinkedListNode<T> class does not support set methods for properties like Owner, Next, and Previous as they are unavailable to a final user. However, when developing a class for the double linked list in task 2, you will realise that such

set methods are absolutely necessary to implement functionality of the data structure. Obviously, we want to link nodes to each other in order to obtain their ordered sequence. Therefore, a mechanism to set at least the next and the previous nodes must exist. Indeed, the mentioned methods are only unavailable to you as a potential user of the LinkedListNode<T>; the picture differs for the developers of this class and the associated LinkedList<T> class. The methods were declared as internal (recall that this is an access modifier), and thus are accessible only within files in the same assembly. This hides these internal methods from users beyond the Microsoft .NET Framework library. A clear reason for this is that a user must not be able to alter nodes' links as the sequential structure of a linked list might be destroyed; that leads to logical and runtime errors. Thus, the data structure takes entire control of assigning proper links to keep its integrity. Because of this, developers (but not users) can see the complete structure and functionality of the classes as what they do is assumed to be valid and correct.

In this practical task, you will proceed in a similar fashion. You need to add a set method for each of the properties (except Value) and declare them as internal. Later, you should be able to notice that it is impossible to modify these properties outside of the DataStructures_Algorithms project (acting as their assembly). For example, as a simple test, nullifying the owner of a node from the Run() method of the Runner04_Task1 class must cause a compilation error as the latter belongs to another assembly. If you compile the program solution, the DataStructures_Algorithms project referred from the Runner (the active StartUp project) will become a DLL library available on your hard disk in the "..\Runner\bin\Debug" subfolder of your project. The Runner project itself will be compiled as an executable file available in the same directory of your hard disk.

Now, complete the implementation of Node<T> with regard to its properties and move forward to realization of the DoubleLinkedList<T> class.

Task 2. Implementation of the Double Linked List

In this part of the practical session, you are asked to implement a new generic class called "DoubleLinkedList" similar to the LinkedList<T> collection, which is a part of Microsoft .NET Framework available in

https://msdn.microsoft.com/en-au/library/he2s3bh7(v=vs.110).aspx

An object of the DoubleLinkedList<T> class must maintain arbitrary number of data elements as a sequence of nodes and provide the following functionality to a user.

DoubleLinkedList()	Initializes a new instance of the DoubleLinkedList<T> class that is empty. If the DoubleLinkedList <T> is empty, the First and Last properties contain null. This constructor is an O(1) operation.
Count	Property. Gets the number of nodes actually contained in the DoubleLinkedList<T>. Retrieving the value of this property is an O(1) operation.
First	Property. Gets the first node of the DoubleLinkedList<T>. If the DoubleLinkedList<T> is empty, the First property contains null. Retrieving the value of this property is an O(1) operation.

Last	Property. Gets the last node of the DoubleLinkedList<T>. If the DoubleLinkedList<T> is empty, the Last property contains null. Retrieving the value of this property is an O(1) operation.
Node<T> AddFirst(T value)	Adds a new node containing the specified value at the start of the DoubleLinkedList<T>. Returns the new Node<T> containing the value. If the DoubleLinkedList<T> is empty, the new node becomes the First and the Last. This method is an O(1) operation.
void AddFirst(Node <T> node)	Adds the specified new node at the start of the DoubleLinkedList<T>. If the DoubleLinkedList<T> is empty, the new node becomes the First and the Last. This method is an O(1) operation. If node is null, it throws the ArgumentNullException. If node already belongs to one of the double linked lists, the method throws the InvalidOperationException.
Node<T> AddLast(T value)	Adds a new node containing the specified value at the end of the DoubleLinkedList<T>. Returns the new Node<T> containing the value. If the DoubleLinkedList<T> is empty, the new node becomes the First and the Last. This method is an O(1) operation.
void AddLast(Node<T> node)	Adds the specified new node at the end of the DoubleLinkedList<T>. If the DoubleLinkedList<T> is empty, the new node becomes the First and the Last. This method is an O(1) operation. If node is null, it throws the ArgumentNullException. If node already belongs to one of the double linked lists, the method throws the InvalidOperationException.
Node<T> AddBefore(Node <T> node, T value)	Adds a new node containing the specified value before the specified existing node in the DoubleLinkedList<T>. It returns the new Node<T> containing value. This method is an O(1) operation. If node is null, it throws the ArgumentNullException. If node is not in the current DoubleLinkedList<T>, the method throws the InvalidOperationException.
void AddBefore(Node <T> node, Node <T> newNode)	Adds the specified new node before the specified existing node in the DoubleLinkedList<T>. This method is an O(1) operation. If node or (and) newNode is (are) null, it throws the ArgumentNullException. If node is not in the current DoubleLinkedList<T> or new node already belongs to one of the double linked lists, the method throws the InvalidOperationException.
Node<T> AddAfter(Node <T> node, T value)	Adds a new node containing the specified value after the specified existing node in the DoubleLinkedList<T>. It returns the new Node<T> containing value. This method is an O(1) operation. If node is null, it throws the ArgumentNullException. If node is not in the current DoubleLinkedList<T>, the method throws the InvalidOperationException.
void AddAfter(Node <T> node, Node <T> newNode)	Adds the specified new node after the specified existing node in the DoubleLinkedList<T>. This method is an O(1) operation. If node or (and) newNode is (are) null, it throws the ArgumentNullException. If node is not in the current DoubleLinkedList<T> or new node already belongs to one of the double linked lists, the method throws the InvalidOperationException.

void Clear()	Removes all nodes from the DoubleLinkedList<T>. Count is set to zero, and references to other objects from elements of the collection are also released. First and Last are set to null. This method is an O(n) operation, where n is Count.
Node<T> Find(T value)	Finds the first node that contains the specified value. The method returns the first Node<T> that contains the value, if found; otherwise, null. The DoubleLinkedList<T> is searched forward starting at First and ending at Last. This method performs a linear search; therefore, this method is an O(n) operation, where n is Count.
Node<T> FindLast(T value)	Finds the last node that contains the specified value. The method returns the last Node<T> that contains the value, if found; otherwise, null. The DoubleLinkedList<T> is searched backward starting at Last and ending at First. This method performs a linear search; therefore, this method is an O(n) operation, where n is Count.
bool Contains(T value)	Determines whether a value is in the DoubleLinkedList<T>. It returns true if value is found in the DoubleLinkedList<T>; otherwise, false. This method performs a linear search; therefore, this method is an O(n) operation, where n is Count.
void Remove(Node <T> node)	Removes the specified node from the DoubleLinkedList<T>. This method is an O(1) operation. If node is null, it throws the ArgumentNullException. If node is not in the current DoubleLinkedList<T>, the method throws the InvalidOperationException.
bool Remove(T value)	Removes the first occurrence of the specified value from the DoubleLinkedList<T>. It returns false if value was not found in the original DoubleLinkedList<T>; otherwise, true. This method performs a linear search; therefore, this method is an O(n) operation, where n is Count.
void RemoveFirst()	Removes the node at the start of the DoubleLinkedList<T>. This method is an O(1) operation. If the DoubleLinkedList<T> is empty, it throws InvalidOperationException.
void RemoveLast()	Removes the node at the end of the DoubleLinkedList<T>. This method is an O(1) operation. If the DoubleLinkedList<T> is empty, it throws InvalidOperationException.
string ToString()	Returns a string that represents the current DoubleLinkedList<T>. 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.

We suggest you to encode all the methods and properties in the order they appear in the table as you will then likely use some already prepared methods to complete others. This should improve code reusability and save your time. Do not implement ToString() immediately as it normally needs a ***foreach*** block, which is the topic of the next two tasks. Therefore, postpone your work on this method till the end of task 4.

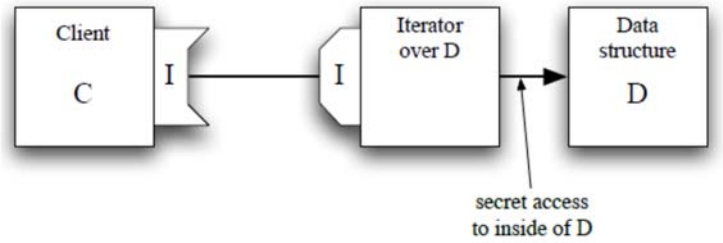
Task 3. The Iterator Pattern

The Iterator pattern provides a simple way for a program to access all the elements of a data structure without exposing its underlying representation. To understand the idea behind this OOP design pattern, imagine a situation where you are checking a set of books in the University library for a particular solution for one of the SIT221 assignments. The conversation between you and the library staff may look like this:

- You: “Can you give me the books one at a time?”
- Library: “Ok.”
- Company: “Are there any more books?”
- Library: “Yes.”
- Company: “Please give me the next book.”
- Library: “Here it is...”

This dialogue is repeated from line three until eventually the library answers “No” to the question, and the task is finished. Clearly, you have no idea what order the books will be brought out, nor do you have any idea where in the library they might have been stored. Getting the books is a problem for the library to solve. Equally clearly, the library has no idea what strategy you are actually using to check for the solution. Checking the books is your problem to solve.

The Iterator pattern neatly separates the task of getting the data that is to be processed (which is the responsibility of the data store) from the problem of processing the data (which is the responsibility of someone outside the data store). It is often used with the collection-type classes, such as Stack, Queue, List, Tree, etc. An application program will store a large amount of data in a collection, and then must process all the data in some way (for example print it or search for items that match a particular property). The application program can process all the data, without knowing how it is actually stored in the collection, by asking the collection to return an iterator that works with it. The iterator allows the application program to retrieve the data items one at a time, in some order (determined by the iterator).



The Iterator pattern can be visualised by the diagram presented above. Here, the internal content of the data structure D is unavailable to Client C, which mainly relies on the interface I implemented by a special class designed to enumerate elements of D. We call this special class as an Iterator over D. In the case of C# application, the generic interface I used by the client is one named as IEnumerator<T>. There is actually one more generic interface named as IEnumerable<T>, which returns an instance of IEnumerator<T> when that is requested by a user. IEnumerable<T> is to be implemented by a data structure, which produces a new instance of the iterator every time the user needs to sequentially extract the elements it stores inside. Clearly, the user may access the iterator via the public methods of the IEnumerator<T>

interface. Depending on the OOP design of the data structure, the class realizing the iterator can be private (built-in inside the data structure's class) or external one. The first case is discussed in

https://programmingwithmosh.com/csharp/ienumerable-and-ienumerator/

and we strongly recommend you to read this article as it will help you with implementation of your iterator for the double linked list. In fact, there is a clear reason to keep the class private as it may directly access the hidden attributes of the data structure. This makes the iterator strongly coupled with implementation of the data structure. In your task, however, you need to create a separate class implementing the IEnumerator<T> as the class will access only interface-specified read-only methods and properties of DoubleLinkedList<T>. Thus, the secret access point (i.e. the connecting mechanism) mentioned in the provided diagram and essential to the internal implementation of the class occurs to be publicly available.

Remember, so that the DoubleLinkedList<T> class can be enumerated, it must implement IEnumerable<T>. This means that you can then use a ***foreach*** block to iterate over DoubleLinkedList<T>. In C#, all collections are enumerable because they implement the IEnumerable interface. Typically, the structure of the program using the iterator pattern follows the scheme:

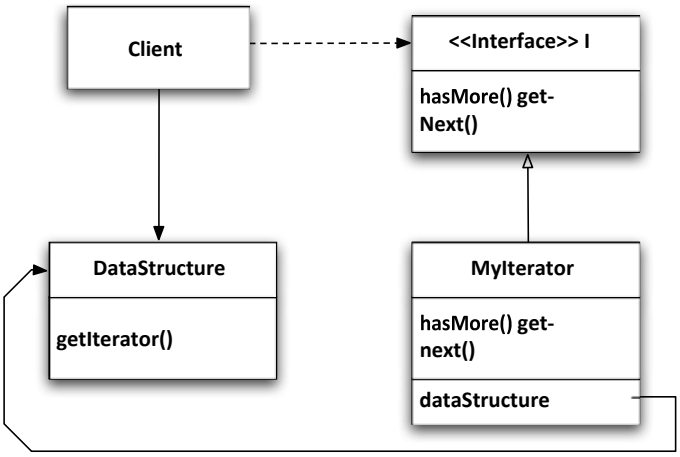
```
var enumerator = list.GetEnumerator();
while (enumerator.MoveNext())
{
      … process enumerator.Current
}
```

And with the facilities introduced in C# language, this program can now be written like this:

```
foreach (var item in list)
{
      … process the item
}
```

In fact, the C# compiler translates this program into code that looks like the earlier program shown above.

Finally, the following UML diagram for the typical Iterator pattern should help you to understand how different classes and interfaces are connected into one logic module. It follows the idea presented in the diagram above.



Task 4. Implementation of the Iterator Pattern

Your first step in this part is to create a new public generic class called "ListEnumerator<T>" and place it into the Week04 subfolder of the DataStructures_Algorithms project. This class must implement the generic interface IEnumerator<T>, which in its turn enforces you to define the following methods and properties:

Current	Property. Gets the element in the collection at the current position of the enumerator.
bool MoveNext()	Advances the enumerator to the next element of the collection.
void Reset()	Sets the enumerator to its initial position, which is before the first element in the collection.
void Dispose()	Performs application-defined tasks associated with freeing, releasing, or resetting unmanaged resources.

Study this interface and explore its associated methods (especially focusing on the remarks and examples) following the link

https://msdn.microsoft.com/en-au/library/78dfe2yb(v=vs.110).aspx

Your second step is encode the listed functions accessing the public methods supported by the DoubleLinkedList<T> class. ListEnumerator<T> will need a constructor, and a good idea is to pass an instance of the DoubleLinkedList<T> class as an input argument. ListEnumerator<T> should store the reference to the instance inside and may then access the members of attached linked list when required. Furthermore, you may keep the body of the Dispose() method empty. Since you are working with the generic version of the IEnumerator interface, you will have to define two properties: T Current (the generic one) and object IEnumerator.Current (the non-generic version). You may provide the same code for the both cases.

The last step is to extend the DoubleLinkedList<T> making it implementing the IEnumerable<T> interface. Again, because of the generic version of the interface, this will force you to define two functions:

- ***IEnumerator<T> GetEnumerator()*** (the generic one), and
- ***IEnumerator IEnumerable.GetEnumerator()*** (the non-generic version).

The reason there are two methods is because IEnumerable<T> inherits the IEnumerable interface so you are seeing the generic method from IEnumerable<T> and the non-generic method from IEnumerable. You only need to ensure that the first method returns

```
new ListEnumerator<T>(this);
```

Then the second method may simply call the first one in a single line of code as follows:

```
return GetEnumerator();
```

Finalize your work on the both interfaces and their functions and proceed with completion of the ***ToString()*** method for DoubleLinkedList<T>.

You should now be able to use a ***foreach*** statement to process elements one by one and return a string. The format of the string returned by the ***ToString()*** method is [a,b,c,d], where a, b, c, and d are string values returned by the corresponding ***ToString()*** method of data type T. Note that this is a good time to test the implementation of your iterator, so make sure that ***ToString()*** does work as expected. You should be mainly concerned about the border cases when the iterator starts and completes enumeration of the elements as these situations often cause potential runtime errors. However, make sure that there are no logical errors either, for example no elements are skipped or checked multiple times. If your iterator is correct, then ***foreach*** block should operate smoothly.

Submission instructions

Submit your program code as an answer to the main task via the CloudDeakin System by the deadline. You should zip your DoubleLinkedList.cs, ListEnumerator.cs, and Node.cs files only and name the zip file as PracticalTask4.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.