Practical Task 5

Due date: 09:00pm Friday, August 24, 2018

(Very strict, late submission is not allowed)

General Instructions

1. 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.

2. 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.

3. This time, your task is to realize a stack and a queue, the two core data structures. To facilitate your learning, study the functionality and content of the both data structures available at https://msdn.microsoft.com/en-au/library/3278tedw(v=vs.110).aspx and https://msdn.microsoft.com/en-au/library/7977ey2c(v=vs.110).aspx as they are given by Microsoft .NET Framework. Your objective is to implement similar data structures and functionality of your classes should in general be similar to the library classes. Therefore, when you have some doubts on how particular methods should act, you may always address to the .NET Framework classes and study their behaviour first. Please, check carefully their 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.

4. Read Section 5.3 and Chapter 6 of SIT221 Workbook available in CloudDeakin in Resources → Course materials → SIT221 Workbook to learn about the stack and the queue. Alternatively, you may explore chapter 6 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.

5. Download the zip file called "Practical Task 5 (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. During the practical task you will need 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 two new runners: Runner05_Task1.cs and Runner05_Task2.cs. They implement IRunner interface and act as main methods in case of this practical task by means of their Run(string[] args) methods. Make sure that you set the right runner class in Program.cs. You may modify these classes as you need as their main purpose is to test the implementation of the data structures and related interfaces. However, they have already some structure to check the stack and the queue that you are required to realize. Therefore, you should first explore the content of these methods and see whether they 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. During testing, make sure you 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 runners by our special versions in order to test functionality of your code for common mistakes.

Objectives

1. Learn implementation of a stack and a queue, the two core data structures

2. Revise the Iterator object-oriented programming design pattern

Specification

Main task

Task 1. Implementation of the Stack data structure

The first task of the practical is to create a new generic class called "MyStack", which is to be acting as the Stack<T> collection that is a part of Microsoft .NET Framework available at

https://msdn.microsoft.com/en-au/library/3278tedw(v=vs.110).aspx

The new class must be placed into the Week05 subfolder. An instance of the MyStack<T> class must maintain arbitrary number of data elements of type T representing the stack data structure. The implementation of MyStack<T> must base on the Vector<T> class, which you have completed earlier within the practical tasks 1-3.

Stacks are useful when you need temporary storage for information; that is, when you might want to discard an element after retrieving its value. Note that this data structure realizes LIFO (Last-In-First-Out) principle that enforces one to add to and extract elements from the top (i.e. the end) of a collection. Therefore, one can only access information in the order reversed to that it is stored in the collection. Because of this fact, a common use for stacks is to preserve variable states during calls to other procedures. Three main operations can be performed on a stack and its elements:

- Push inserts an element at the top of the stack.
- Pop removes an element from the top of the stack.
- Peek returns an element that is at the top of the stack, but does not remove it from the stack.

Because you are to employ the Vector<T> class, the capacity of MyStack<T> is that one of the underlying vector. Note that as elements are added to a stack, its capacity is automatically increased as required by reallocating the internal vector. Similarly to the vector, the stack has a Count property. If Count is less than the capacity of the stack, Push is an $O(1)$ operation. If the capacity needs to be increased to accommodate a new element, Push becomes an $O(n)$ operation, where n is Count. Pop is an $O(1)$ operation.

Now proceed with the implementation of the new class and ensure that it provides the following functionality to a user.

MyStack()	Initializes a new instance of the MyStack<T> class that is empty. This constructor is an $O(1)$ operation.
Count	Gets the number of elements contained in the MyStack<T>. Retrieving the value of this property is an $O(1)$ operation.
void Clear()	Removes all objects from the MyStack<T>. Count is set to zero, and references to other objects from elements of the collection are also released. This method is an $O(\mathrm{n})$ operation, where n is Count.
bool Contains(T item)	Determines whether an element is in the MyStack<T>. Returns true if item is found in the MyStack<T>; otherwise, false. This method performs a linear search; therefore, this method is an $O(\mathrm{n})$ operation, where n is Count.
T Peek()	Returns the object at the top of the MyStack<T> without removing it. This method is similar to the Pop method, but Peek does not modify the MyStack<T>. The method returns InvalidOperationException if the MyStack<T> is empty. This method is an $O(1)$ operation.
T Pop()	Removes and returns the object at the top of the MyStack<T>. The method throws InvalidOperationException if the MyStack<T> is empty. This method is similar to the Peek method, but Pop does modify the MyStack<T>. This method is an $O(1)$ operation.
void Push(T item)	Inserts an object at the top of the MyStack<T>. If Count is less than the capacity of the stack, Push is an $O(1)$ operation. If the capacity needs to be increased to accommodate the new element, Push becomes an $O(\mathrm{n})$ operation, where n is Count.

string ToString()	Returns a string that represents the current MyStack<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.

Furthermore, the MyStack<T> class must be enumerable; that is, it must implement the IEnumerable<T> interface and have an associated iterator supporting traversal of elements in the data structure. The latter must be a generic iterator class implementing the IEnumerator<T> interface. To succeed with the task, you must realize the Iterator OOP design pattern similar to one you have already done for the DoubleLinkedList<T> in the Practical Task 4. To cope with the Iterator, remind that the class presenting it can be either private (built-in inside the data structure's class) or external. Although, the Practical Task 4 dealt with the second variant, this time you must implement it via a new **private** (internal) class within the MyStack<T> class similar to how it is discussed in

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

This version allows the iterator class to directly access the underlying data structure of MyStack<T>; that is, an instance of the Vector<T> class running inside MyStack<T>. Note that this makes the iterator strongly coupled with the implementation of the data structure.

Also remember, so that the MyStack <T> class can be enumerated, it must implement IEnumerable<T>. This means that you can then use a ***foreach*** block to iterate over the MyStack<T>. Again, this interface will force you to define two functions:

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

You only need to ensure that the first method returns an instance of the private iterator class, for example

```
return new StackEnumerator<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 the MyStack<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. If your iterator is correct, then ***foreach*** block should operate smoothly.

Task 2. Application of the Stack data structure: Reverse Polish Notation

Reverse Polish Notation (RPN) is a mathematical notation in which every operator follows all of its operands. In other words, it is a way of expressing arithmetic expressions that avoids the use of brackets to define priorities for evaluation of operators. In ordinary notation, one might write

$$(3 + 5) * (7 – 2)$$

and the brackets tell us that we have to add 3 to 5, then subtract 2 from 7, and multiply the two results together. In RPN, the numbers and operators are listed one after another, and an operator always acts on the most recent numbers in the list. The numbers can be thought of as forming a stack, like a pile of plates. The most recent number goes on the top of the stack. An operator takes the appropriate number of arguments from the top of the stack and replaces them by the result of the operation. In the Reverse Polish Notation, the above expression would be

$$3\ 5 + 7\ 2 – *$$

Reading from left to right, this is interpreted as follows:

- Push 3 onto the stack.
- Push 5 onto the stack. Reading from the top, the stack now contains (5, 3).
- Apply the + operation: take the top two numbers off the stack, add them together, and put the result back on the stack. The stack now contains just the number 8.
- Push 7 onto the stack.
- Push 2 onto the stack. It now contains (2, 7, 8).
- Apply the – operation: take the top two numbers off the stack, subtract the top one from the one below, and put the result back on the stack. The stack now contains (5, 8).
- Apply the * operation: take the top two numbers off the stack, multiply them together, and put the result back on the stack. The stack now contains just the number 40.

In the given template, you will find the RPNCalculator class, where Add() method has been already written and introduced in the code computing the RPN. This is a very simple version of RPN Calculator with one stack operands (numbers). You will notice that in the constructor it receives a vector called “expression”. Your task is to complete this class with regard to the three methods: Subtract, Multiply, and Divide. You need to test then not only the RPNCalculator class, but mainly correctness of the MyStack<T>. Finally, in the submission form of this practical task, you **must briefly yet clearly explain** how this algorithm works, how it stores intermediate results of the arithmetic operations, and how this all leads to the final result. The provided Runner05_Task1 class will help you with one of test cases for the RPNCalculator, but clearly you should produce some other instances to check your solution well.

Task 3. Implementation of the Queue data structure

Your last task is to create, in the Week05 subfolder, a new generic class called "MyQueue", which is to work similarly to the Queue<T> collection that is a part of Microsoft .NET Framework available at

https://msdn.microsoft.com/en-au/library/7977ey2c(v=vs.110).aspx

An instance of the MyQueue<T> class must maintain arbitrary number of data elements of type T representing the queue data structure. The implementation of MyQueue <T> must base on the DoubleLinkedList<T> class, which you have completed earlier within the practical task 4. Note that this data structure realizes FIFO (First -In-First-Out) principle that enforces one to add to and extract elements from the head of a collection. Therefore, one can only access information in the order in which it is stored in the collection.

Now proceed with the implementation of the new class and ensure that it provides the following functionality to a user.

MyQueue()	Initializes a new instance of the MyQueue<T> class that is empty. This constructor is an $O(1)$ operation.
Count	Gets the number of elements contained in the MyQueue<T>. Retrieving the value of this property is an $O(1)$ operation.
void Clear()	Removes all objects from the MyQueue<T>. Count is set to zero, and references to other objects from elements of the collection are also released. This method is an $O(\mathrm{n})$ operation, where n is Count.
bool Contains(T item)	Determines whether an element is in the MyQueue<T>. Returns true if item is found in the MyQueue<T>; otherwise, false. This method performs a linear search; therefore, this method is an $O(\mathrm{n})$ operation, where n is Count.
T Dequeue()	Removes and returns the object at the beginning of the MyQueue<T>. This method throws InvalidOperationException if the MyQueue<T> is empty. This method is similar to the Peek method, but Peek does not modify the MyQueue<T>. This method is an $O(1)$ operation.
void Enqueue(T item)	Adds an object to the end of the MyQueue<T>. This method is an $O(1)$ operation.
T Peek()	Returns the object at the beginning of the MyQueue<T> without removing it. It throws InvalidOperationException if the MyQueue<T> is empty. This method is similar to the DeMyQueue method, but Peek does not modify the MyQueue<T>. This method is an $O(1)$ operation.
string ToString()	Returns a string that represents the current MyQueue<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.

By analogy with the MyStack<T> class, the MyQueue<T> must be enumerable. Make sure it implements the IEnumerable<T> interface and has a private class inside the MyQueue<T> acting as an iterator. Again, the iterator must be a generic class implementing the IEnumerator<T> interface. Encode the ToString() method using a foreach statement to process elements one by one to return a string. Check the correctness of the ToString() to ensure that the enumeration works fine. The format of the ToString() method is the same as one for the MyStack<T>.

Additional Task

In the submission form of this practical task, briefly explain (in ~150 words) pros and cons of vector and linked list based implementations of the stack and the queue data structures.

Submission instructions

Submit your program code as an answer to the main task via the CloudDeakin System by the deadline. You should zip your MyStack.cs, MyQueue.cs, and RPNCalculator.cs files only and name the zip file as PracticalTask5.zip. Please, **make sure the file names are correct**. You must not submit the whole MS Visual Studio project / solution files.

Note that the exercises from the section of **additional tasks will be marked**, too. You must attempt them to get full scores.

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.