Practical Task 1

Due date: 09:00pm Friday, July 20, 2018

(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. Read Chapter 1 of SIT221 Workbook available in CloudDeakin in Resources → Course materials → SIT221 Workbook. It provides motivation part for the practical and will give you general understanding of the task and issues related to the use of arrays in practice.

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. Download the zip file called “Practical Task 1 (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. The former already has Vector.cs file in Week01 subfolder. You need to modify this class file according to the specification. You are allowed to change it as you want unless you meet all the requirements in terms of functionality and signatures of requested methods and properties. The later project has Runner01.cs file whose Run(string[] args) method acts as a main method in case of this practical task. You are free to modify it as you need as its main purpose is to test the implementation of the vector class. You should use this class and this method to thoroughly test the vector class aiming on covering all potential logical issues and runtime errors. This (testing) part of the task is as important as

writing the vector class. We will later on replace this file by our special version in order to test functionality of your vector class for common mistakes.

In addition, the Runner project has the folder called "Data", which contains test data for the first practical task. This is a dataset of three files: 1H.txt, 1T.txt, and 1M.txt with 100, 1000 and, 10,000 integer values, respectively. The Runner reads the data via the argument of the command line. You may change parameter of the command line when debugging in MS Visual Studio 2015 following the order Project → DataStructures_Algorithms Properties → Debug → Start Options → Command line arguments. You then need to replace the line by a new one, for example "..\..\Data\Week01\1T.txt". This line sets the path to a data file.

5. Study well the functionality and content of the List<T> class existing in Microsoft .NET Framework https://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx. Your task is to implement a data structure similar to the List, where you also **must** use a simple array as internal data structure to collect and store given elements of generic type T. Functionality of your class should in general be similar to the List<T>. Therefore, when you have some doubts on how particular methods should act, you may always address to the List<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.

Objectives

1. Review key object-oriented programming concepts

2. Relate complexity analysis concepts to practice

3. Review basic file I/O operations

Specification

Main task.

In this practical session, you are asked to implement a new generic class called ***Vector*** similar to the List<T> collection, which is a part of Microsoft .NET Framework. An object of this class must maintain arbitrary number of data elements and provide the following functionality to a user.

Vector()	Initializes a new instance of the Vector<T> class that is empty and has the default initial capacity, say 10 elements. The capacity of a Vector<T> is the number of elements that the Vector<T> can hold. As elements are added to a Vector<T>, the capacity is automatically increased as required by reallocating the internal array. This constructor is an O(1) operation.

Vector(int capacity)	Initializes a new instance of the Vector<T> class that is empty and has the specified initial capacity. It accepts the number of elements that the new list can initially store as an input parameter. If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the Vector<T>. This constructor is an O(*n*) operation, where *n* is capacity.
Capacity	Property. Gets or sets the total number of elements that the internal data structure can hold without resizing. Capacity is the number of elements that the Vector<T> can store before resizing is required, whereas Count is the number of elements that are actually in the Vector<T>. Capacity is always greater than or equal to Count. If Count exceeds Capacity while adding elements, the capacity is increased by automatically reallocating the internal array before copying the old elements and adding the new elements. Retrieving the value of this property is an O(1) operation; setting the property is an O(*n*) operation, where *n* is the new capacity. It throws ArgumentException when a new capacity value is invalid.
Count	Property. Gets the number of elements contained in the Vector<T>. Count is the number of elements that are actually in the Vector<T>. Retrieving the value of this property is an O(1) operation.
void Add(T item)	Adds an object to the end of the Vector<T>. If Count already equals Capacity, the capacity of the Vector<T> is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added. If Count is less than Capacity, this method is an O(1) operation. If the capacity needs to be increased to accommodate the new element, this method becomes an O(n) operation, where n is Count.
void Insert(int index, T item)	Inserts an element into the Vector <T> at the specified index. If Count already equals Capacity, the capacity of the Vector<T> is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added. If index is equal to Count, item is added to the end of Vector<T>. This method is an O(n) operation, where n is Count. It throws IndexOutOfRangeException when the index value is invalid.

void Clear()	Removes all elements from the Vector<T>. Count is set to 0, and references to other objects from elements of the collection are also released. Capacity remains unchanged. This method is an O(n) operation, where n is Count.
int IndexOf(T item)	Searches for the specified object and returns the zero-based index of the first occurrence within the entire data structure. This method performs a linear search; therefore, this method is an O(n) operation. It returns the zero-based index of the first occurrence of item within the entire Vector<T>, if found; otherwise, –1.
bool Contains(T item)	Determines whether an element is in the Vector<T>. This method performs a linear search; therefore, this method is an O(n) operation, where n is Count.
bool Remove(T item)	Removes the first occurrence of a specific object from the Vector<T>. It returns *true* if item is successfully removed; otherwise, *false*. This method also returns *false* if item was not found in the Vector<T>. This method performs a linear search; therefore, it is an O(n) operation, where n is Count.
int RemoveAll(T item)	This method should delete all occurrences of an element from a Vector object. Returns the number of elements removed from the Vector<T>. This method performs a linear search; therefore, this method is an O(n) operation, where n is Count.
void RemoveAt(int index)	Removes the element at the specified index of the Vector<T>. When you call RemoveAt to remove an item, the remaining items in the list are renumbered to replace the removed item. For example, if you remove the item at index 3, the item at index 4 is moved to the 3rd position. In addition, the number of items in the vector (as represented by the Count property) is reduced by 1. This method is an O(n) operation, where n is (Count - index). It throws IndexOutOfRangeException when the index value is invalid.
T Max()	This method returns the maximum value in a generic sequence. If the source sequence is empty, it returns value which is default for the type T.
T Min()	This method returns the minimum value in a generic sequence. If the source sequence is empty, it returns value which is default for the type T.
T This[int index]	Returns the element at a specified index in a sequence. It accepts as an argument the zero-based index of the element to retrieve. It throws IndexOutOfRangeException when the index value is invalid.

string ToString()	Returns a string that represents the current object. 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.

For simplicity, you may assume that the value of ***capacity*** property can only be increased. Therefore, ***ArgumentException*** is thrown every time when one tries to set value strictly less than the current one.

In order to implement Min() and Max() methods properly, assume (and realize this in your code) that the data type T implements interface ***IEnumerable<T>***. This assumption allows you to extract ***Comparer<T>*** object via the ***Default*** property of ***Comparer<T>*** class as follows:

```
Comparer<T> comparer = Comparer<T>.Default;
```

The ***comparer*** then allows you to call its ***Compare()*** method to compare two variables of type T. This is useful when you need to determine whether one variable is greater than another. To read more about the ***Default*** property, check the reference

https://msdn.microsoft.com/en-us/library/cfttsh47(v=vs.110).aspx.

Furthermore, explore related articles in MSDN service. In addition, read about the default value for type T to complete these two methods. See for example

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/default-value-expressions

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.

Additional Tasks.

1. What is the best and worst case running time for ***IndexOf(…)***, ***Max()***, ***Min()*** methods? How the run time changes between the three datasets 1H, 1T, and 10M?
2. Should O(1) complexity for the ***Max()*** and ***Min()*** operations be achieved, how the implementation of the ***Vector*** class should be changed?
3. In folder DataStructures_Algorithms/Utils there is a Data Serializer class that we use to load data from text file, save data to text file, serialize and deserialize your ***Vector*** class. Review the class, and make sure you understand it.

Submission instructions

Submit you program code as an answer to the main task via the CloudDeakin System by the deadline. You should zip your Vector.cs file only and name the zip file as Vector.zip. Please, **make sure the file name is correct**. You must not submit the whole MS Visual Studio project / solution files. You must answer additional questions 1 and 2 and enter your answers into the submission textbox in the CloudDeakin system. Your answers should be concise.

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.