Practical Task 1.1

(Pass Task)

Submission deadline: Monday, March 17
Discussion deadline: Friday, April 4

Instructions

The objective of your first task is to implement a generic data structure Vector<T> that is to be similar to the collection class List<T> of the Microsoft .NET Framework. You can explore the functionality of the List<T> class at

https://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx

A Vector (also known as a List) is a simple data structure used to store any number of elements in a single dimension. Essentially, Vectors are like arrays, except for a few differences. Firstly, they are not a built-in language construct and instead need to be directly implemented. Secondly, while all arrays remain static and their sizes can only be set upon creation, Vectors can be resized during the run time. They also provide broad functionality including such basic operations as accessing, recording, and deleting elements from the data collection. However, it is important to understand that the implementation of any Vector class is based on the use of a simple array, which is the internal data structure wrapped by the Vector class. Thus, the Vector class manipulates the internal array giving the illusion that the user deals with a dynamic data collection that enables adding new and deleting existing elements.

Recall that 'T' within the angle brackets (< >) of the Vector<T> class refers to a generic type, which in practice can be substituted by any specific data type, including bool, int, string, an array, or a class. To comprehend the following instructions and succeed with this task, you should have a strong understanding of generic classes and methods as has been covered earlier in SIT232 unit on Object-Oriented Programming and revised in the first lecture of this unit. Thus, in case you accidentally missed the lecture, we strongly recommend you start with watching its recording as the example for the Stack<T> class considered there should help you with the implementation of your Vector<T> class.

The following steps indicate what you need to do.

1. Download the two C# source code files attached to this task. These files serve as a template for the program that you need to develop. Create a new C# project and import the files. Your newly built project should compile and work without errors, although it will initially fail most of provided test cases.
   - **Vector.cs** file contains the partially completed Vector<T> class that you will need to finish by adding remaining methods. Therefore, explore this code thoroughly as the missing methods can reuse some of the existing methods or may appear similar to them in terms of implementation. For more information see Appendix A of this task.
   - **Tester.cs** file is where you will find the Main method to be used as the starting point of your program. It also contains a series of tests that will verify if your Vector class works correctly. When first running your program, these tests will fail. Once you have completed the task, all these tests will report success. See Appendix B for an example of what the program will output when working correctly.
2. You must complete the Vector<T> implementation by providing the following functionality.
   - **void Insert(int index, T item)**

     Inserts the given **item** into the data structure at the specified **index**. If **Count** equals **Capacity** (i.e., the currently allocated space to store data elements is full), the capacity of the Vector<T> must be increased by reallocating the internal array with copying all existing elements to the new larger array before the new element is added[1].

[1] For simplicity, you may assume that the value of **Capacity** can only be increased.

  - If the specified **index** is equal to **Count**, the item is added to the end of the Vector<T>.
  - If the specified **index** is outside of the range of the current Vector, this method throws an exception of the IndexOutOfRangeException class.

– **void Clear( )**

  Sets **Count** to 0 and removes all elements from the Vector<T> without changing its capacity.

– **bool Contains(T item)**

  Determines whether the specified **item** is in the data collection. It returns 'true' when the **item** is presented, and 'false' otherwise.

– **bool Remove(T item)**

  Removes the first occurrence of the specified **item** from the data collection. It returns 'true' if the **item** is successfully removed. This method returns 'false' if the **item** is not found in the Vector<T>.

– **void RemoveAt(int index)**

  Removes the element at the specified **index** of the Vector<T>.

  - This method renumbers the items remaining in the list to close the gap caused by deletion of the respective item. In addition, the number of items in the data collection (as represented by **Count**) is reduced by 1. For example, if one removes the item at index 3, the item at index 4 is moved to the 3rd position. Such move is reiterated for each element in the collection, including the last element.
  - If the **index**'s value is invalid, this method throws an exception of the IndexOutOfRangeException class.

– **String ToString( )**

  Returns a string that represents the current object.

  - Being the major formatting method in the Microsoft .NET Framework, this method converts an object to its string representation so that it is suitable for display. The Object class, which is the base class for all C# classes, provides a default implementation of this method. However, this simply generates a fully qualified type name and does not provide information about the object's contents. To provide a specific implementation of the ToString() method, one needs to override it in the respective user's class.
  - Here, the resulting string must be a comma separated list of each of the elements in the Vector[2]. The sequence must start and end with square brackets ([ ]). For example, [a,b,c,d] is the valid string representation for four characters: 'a', 'b', 'c', and 'd'.

3. Keep in mind that you may reuse some of the existing methods (e.g., IndexOf, RemoveAt, or Contains) to simplify your implementation and ensure its correctness.
4. As you progress through the implementation, you should start using the Tester class to thoroughly test the Vector<T> aiming on the coverage of all potential logical issues and runtime errors. This (testing) part of the task is as important as writing the vector class. The given version of the testing class covers only some basic cases. Therefore, you should extend it with extra cases to make sure that your vector class is checked against other potential mistakes.
5. In this task, **you are not allowed** to use any library functions such as the methods provided by the LINQ library. For example, when copying an array, you must implement a loop that does this rather than delegate this job to a method outside of your own code.

Further Notes

– Read Chapter 1 of SIT221 Workbook available in CloudDeakin in Content → Learning Resources → SIT221 Workbook. It should give you a good understanding of the task and issues related to the use of arrays in practice.

[2] StringBuilder is a C# class that will help you construct a String from your Vector. See the following link for details: https://docs.microsoft.com/en-us/dotnet/api/system.text.stringbuilder?view=netcore-3.1

- If you still struggle with such OOP concept as Generics, you may wish to read Chapter 11 of SIT232 Workbook available in Content → Learning Resources → SIT232 Workbook. You may also have to read Chapter 6 of SIT232 Workbook about Polymorphism and Interfaces as you will need an excellent understanding of these topics in order to progress well through the practical tasks of the unit. Make sure that you are proficient with them as they form a basis to design and develop programming modules in this and all subsequent tasks. In other chapters of the workbook, you may find more important topics required to complete the task, for example, the material on exceptions handling.
- We will test your code using the .NET Core 6.0. You are free to use any IDE (for example, Visual Studio Code), though we recommend you work from Microsoft Visual Studio 2022 due to its simple installation process and good support of debugging. You can find the necessary instructions and installation links by navigating in CloudDeakin to Content → Learning Resources → Software.

Submission Instructions and Marking Process

To get your task completed, you must finish the following steps strictly on time.

- Make sure your programs implement the required functionality. They must compile, have no runtime errors, and pass all test cases of the task. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your programs thoroughly before submission. Think about potential errors where your programs might fail.
- **Submit** the expected code files as a solution to the task via OnTrack submission system.
- Once your solution is accepted by your tutor, you will be invited to **continue its discussion and answer relevant theoretical questions through a face-to-face interview**. Specifically, you will need to meet with the tutor to demonstrate and discuss the solution in one of the dedicated practical sessions (run online via MS Teams for online students and on-campus for students who selected to join classes at Burwood\Geelong). Please, come prepared so that the class time is used efficiently and fairly for all students in it. Be on time with respect to the specified discussion deadline.

  You will also need to **answer all additional questions** that your tutor may ask you. Questions will cover the lecture notes; so, attending (or watching) the lectures should help you with this **compulsory** discussion part. You should start the discussion as soon as possible as if your answers are wrong, you may have to pass another round, still before the deadline. Use available attempts properly.

Note that we will not accept your solution after **the submission deadline** and will not discuss it after **the discussion deadline**. If you fail one of the deadlines, you fail the task, and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Remember that this is your responsibility to keep track of your progress in the unit that includes checking which tasks have been marked as completed in the OnTrack system by your marking tutor, and which are still to be finalised. When grading your achievements at the end of the unit, we will solely rely on the records of the OnTrack system and feedback provided by your tutor about your overall progress and the quality of your solutions.

Appendix A: The Contents of the Original Vector<T> class

The following is a list of the properties and methods already provided to you in the original Vector<T> class attached to this task.

- int **Capacity**

  Property. Gets the maximum number of elements that the data structure can hold without resizing.

- int **Count**

  Property. Gets the number of elements physically contained in the Vector<T>.

  - This is the number of elements that are actually stored in the Vector<T>; it is different to **Capacity** that bounds the value of **Count**. Therefore, the value of **Count** is always less than or equal to the value of **Capacity**. If **Count** exceeds **Capacity** at the time of adding a new element, **Capacity** must be increased by copying the elements stored in the internal array to a new array of a larger size before adding the new element.

- **Vector( )**

  Constructor. Initializes a new instance of the Vector<T> class that is empty and has **Capacity** set to a default value, say 10 elements.

- **Vector(int capacity)**

  Constructor. Initializes a new instance of the Vector<T> class that is empty and has **Capacity** set to the specified value given as the input parameter. It is worth to estimate the maximum number of elements that one wants to store in the data collection as this eliminates the need to perform a number of resizing operations while adding new elements to it.

- **void Add(T item)**

  Adds the specified **item** to the end of the Vector<T>. At this time, if **Count** equals **Capacity**, the capacity of the Vector<T> must be increased by reallocating the internal array with all existing elements copied to a new larger array before the new element is added.

- **int IndexOf(T item)**

  Searches for the specified **item** and returns the zero-based index of the first occurrence within the entire data structure. It returns the zero-based index of the item, if found; otherwise, it returns –1.

- **T this[int index]**

  Returns the element at the specified **index** in the collection. As an input argument, it accepts a zero-based **index** of the element to retrieve. If the value of the **index** is invalid, this method throws an exception of the IndexOutOfRangeException class.

Appendix B: Expected Printout

The following provides an example of the output generated from the testing module (Tester.cs) once you have correctly implemented all methods of the Vector<T> class.

```
Test A: Create a new vector by calling 'Vector<int> vector = new Vector<int>(50);'
 :: SUCCESS


Test B: Add a sequence of numbers 2, 6, 8, 5, 5, 1, 8, 5, 3, 5
 :: SUCCESS


Test C: Remove number 3, 7, and then 6
 :: SUCCESS


Test D: Insert number 50 at index 6, then number 0 at index 0, then number 60 at index 'vector.Count-1', then number 70 at index
'vector.Count'
 :: SUCCESS


Test E: Insert number -1 at index 'vector.Count+1'
 :: SUCCESS


Test F: Remove number at index 4, then number index 0, and then number at index 'vector.Count-1'
 :: SUCCESS


Test G: Remove number at index 'vector.Count'
 :: SUCCESS


Test H: Run a sequence of operations:
vector.Contains(1);
```

```
 :: SUCCESS
vector.Contains(2);
 :: SUCCESS
vector.Contains(4);
 :: SUCCESS
vector.Add(4); vector.Contains(4);
 :: SUCCESS

Test I: Print the content of the vector via calling vector.ToString();
[2,8,5,1,8,50,5,60,5,4]
 :: SUCCESS

Test J: Clear the content of the vector via calling vector.Clear();
 :: SUCCESS

 ------------------- SUMMARY -------------------
Tests passed: ABCDEFGHIJ
```