Practical Task 4.1

(Credit Task)

Submission deadline: 11:59 pm Sunday, April 10
Discussion deadline: 11:59 pm Sunday, May 1

Task Objective

The aim of this task is to practice recursion in both the searching of an array and when iterating over a data structure. This task has two stages, the first is to implement the binary search algorithm and the second is to implement the Iterator design pattern.

Background: Binary Search

Binary Search is a simple and highly efficient search algorithm that many people derive themselves as a child when playing the number guessing game. The algorithm requires that a list of numbers be already sorted. It then uses recursion to split the search space in half with each iteration. Therefore, the algorithm uses the divide and conquer paradigm. The algorithm simply involves looking at the item in the middle of the array and comparing this with item being searched for. If the item we seek does not match the central item, then you recursively search the half of the array that the item must lie. As this algorithm halves the search space each time it results in a O(log n) search. This is clearly better than the linear search with O(n) that we looked at in the first week.

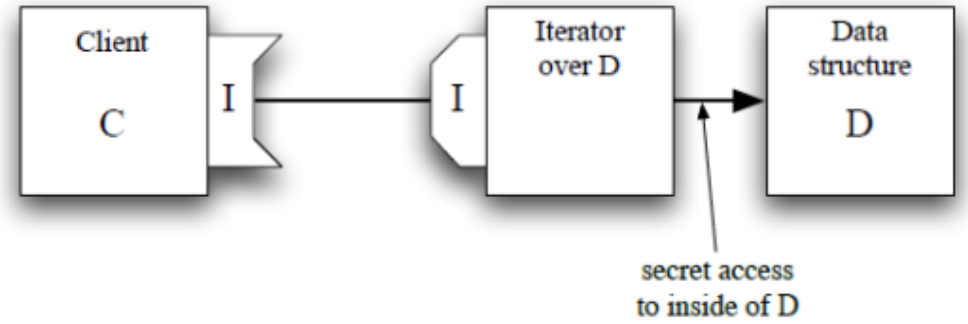
Background: Iterator Design Pattern

The purpose of this task is to learn implementation of the *Iterator* object-oriented programming design pattern. The Iterator design pattern enables a simple sequential access to the elements of a data structure without exposing its underlying representation. To grasp the conceptual model of this pattern, imagine a situation where you are successively requesting a collection of books from a library. The conversation between you and a librarian may be as follows:

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

This dialogue is repeated from line 3 until the librarian answers “No” to your question, and the enumeration is finished. Clearly, you have no idea what order the books will be brought out, nor you know where and how in the library they have been stored. Getting the books is the responsibility of the library. The library also has no idea what specifically you are looking for, thus checking the books is your task.

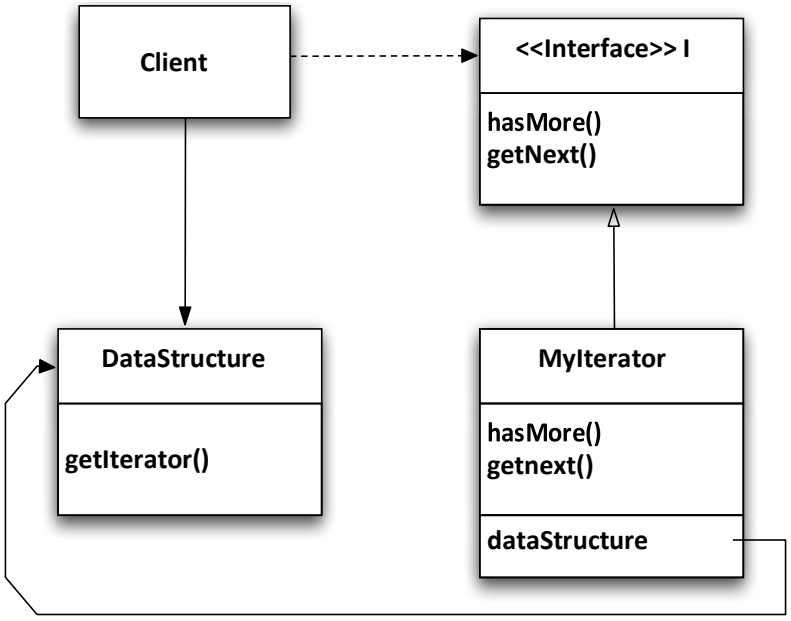
The Iterator pattern neatly separates the task of getting data (which is the responsibility of a data store) from processing the data (which is the user’s responsibility). It is often used with collection-type classes, e.g. List, Linked List, Stack, and Queue. By means of the pattern, applications can store a large amount of data in a collection and process it without knowing how the collection stores it internally. To get access to the elements, an application just needs to request the iterator implemented for the data structure. Then the iterator retrieves the elements, one at a time, as the application asks for the next one. The order in which elements are returned is determined by the iterator. The Iterator pattern can be visualised via the following diagram.



Here, the internal content of the data structure D remains invisible to the client C, who just expects the interface I. The interface I, which is known as IEnumerator<T> within the .NET Framework, realizes the functionality (methods) required to enumerate elements of D. Because the interface I only prescribes a set of methods to a class, one still has to develop a special class that implements the interface. This class is referred as an Iterator over D. Note that the IEnumerator<T> is generic and the expected data type T must match that of the elements in the related data structure.

Indeed, to get access, the client C must request an instance of the Iterator class from the data structure D. To provide it to the client, the data structure must implement another generic interface, the IEnumerable<T>. This interface enforces the data structure to implement the GetEnumerator public method that returns an instance of the Iterator. In practice, the GetEnumerator produces a new instance of the Iterator class every time the client requests an access to the elements that it stores inside. Usually, the Iterator class is *private* (i.e. built-in inside the data structure's class). There is a strong reason to keep the class private (*nested*) as it may directly access the private (hidden from the client) attributes of the data structure. This makes the iterator strongly coupled with its data structure.

In C#, all collections are enumerable because they implement the IEnumerable interface. This enables the use of *foreach statement* to iterate over a generic collection. The UML class diagram depicted below complements the aforementioned diagram and describes the typical structure of the Iterator pattern.



Now, when you have got some insights about the Iterator pattern and its implementation details, proceed with the task and realize the pattern for the Vector<T> class.

Task Details

Using the Vector class you implemented as part of Task 1.1P work through the following steps to complete the task:

1. Explore the program code attached to this task. Create a new Microsoft Visual Studio project and import the Vector.cs file, or alternatively, extend the project inherited from the previous tasks by copying the missing code from the enclosed template for the Vector<T> class. Import the Tester.cs file to the project to access the prepared Main method important for the purpose of debugging and testing the required algorithmic solution.
2. In the first part of this task, you must further extend the Vector<T> class to enable searching an element in a sequence of generic data elements using the recursive version of Binary Search. You must add the following functionality to the class:
   - **int BinarySearch(T item)**

     Searches within the entire sorted Vector<T> for an element using the default comparer and returns the zero-based index of the element, if item is found; otherwise, a negative number (e.g. -1). This method uses the default comparer Comparer<T>.Default for type T to determine the order of the elements in the Vector<T>.
   - **int BinarySearch(T item, IComparer<T> comparer)**

     Searches within the entire sorted Vector<T> for an element using the specified comparer and returns the zero-based index of the element, if item is found; otherwise, a negative number (e.g. -1).

   Note that you are not allowed to delegate the binary search operation to the Array class or any other collection classes. You must implement the Binary Search algorithm from scratch. To learn the similarity with the standard List<T> class of .NET Framework, explore the material available at

   https://msdn.microsoft.com/en-us/library/w4e7fxsh(v=vs.110).aspx

   and focus on provided remarks and complexity issues.
3. In the second part you will implement the Iterator Design Pattern. We strongly recommend you explore the link below and follow the tutorial. It should help you with the implementation issues of the iterator that you need to construct.

   https://programmingwithmosh.com/csharp/ienumerable-and-ienumerator/
4. Check the link below to read about the IEnumerator<T> interface and its associated methods. Focus on the remarks and examples.

   https://msdn.microsoft.com/en-au/library/78dfe2yb(v=vs.110).aspx
5. Inside the Vector<T> class, develop the Iterator, a new private (nested) class, so that it implements the generic IEnumerator<T> interface. Here, you may need to find out how to write a nested class. The IEnumerator<T> will ask 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.

6. Note that the Iterator class needs a constructor, and a good idea is to pass the *data* array of the Vector<T> as an argument so that the Iterator can record it as a *private reference* and access later to get particular elements. Furthermore, because the IEnumerator<T> is a generic interface, you will have to define two properties:
   - **T Current** and
   - **object IEnumerator.Current**

   The latter is required for compatibility of the Iterator class with earlier C# code that may not support generics. However, you may write exactly the same code inside the *get* method of the both properties. Finally, you may keep the body of the Dispose method empty.
7. As the last step, you will need to alter your Vector<T> class to indicate to the client programs that it can be enumerated by using the IEnumerable<T> interface and provide the required methods. These methods have not been included in the code provided. Additionally you will need to connect the GetEnumerator method you implement in your Vector<T> to the underlying Iterator class. The code of the method must create and return a new instance of the Iterator.
8. As you progress with the implementation of the Binary Search and Iterator pattern, you should start using the Tester class in order to test your code for potential logical issues and runtime errors. This (testing) part of the task is as important as coding. You may wish to extend it with extra test cases to be sure that your solution is checked against other potential mistakes. To enable the tests, remember to uncomment the corresponding code lines. Before you proceed with searching, make sure that your data is sorted according to the dedicated comparer.
9. As a credit task it is important to ensure your submission is professional. Ensure you remove debug code, correctly format, use formal naming convention consistently and comment your code. Submissions will not be accepted that are not of a professional quality.

Expected Printout

Appendix A provides an example printout for your program. If you are getting a different printout then your implementation is not ready for submission. Please ensure your program prints out Success for all tests before submission.

Further Notes

- Explore Chapter 7 of SIT221 Workbook available in CloudDeakin in Resources → Additional Course Resources → Resources on Algorithms and Data Structures → SIT221 Workbook. It starts with general explanation of algorithm complexity and describes Bubble Sort, Insertion Sort, and Selection Sort algorithms. Study the provided examples and follow the pseudocodes as you progress with coding of the algorithms.
- The implementation of the Insertion Sort algorithm is also detailed in Chapter 3.1.2 of the course book "Data Structures and Algorithms in Java" by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser (2014). You may access this book on-line for free from the reading list application in CloudDeakin available in Resources → Additional Course Resources → Resources on Algorithms and Data Structures → Course Book: Data structures and algorithms in Java.
- If you still struggle with such OOP concepts as Generics and their application, you may wish to read Chapter 11 of SIT232 Workbook available in Resources → Additional Course Resources → Resources on Object-Oriented Programming. You may also have 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 unit. Make sure that 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.

- We will test your code in Microsoft Visual Studio 2017. Find the instructions to install the community version of Microsoft Visual Studio 2017 available on the SIT221 unit web-page in CloudDeakin at Resources → Additional Course Resources → Software → Visual Studio Community 2017. You are free to use another IDE if you prefer that, e.g. Visual Studio Code. But we recommend you to take a chance to learn this environment.

Marking Process and Discussion

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

1. Work on your task either during your allocated lab time or during your own study time.
2. Once the task is complete you should make sure that your program implements all the required functionality, is compliable, and has no runtime errors. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your program thoroughly before submission. Think about potential errors where your program might fail. Note we can sometime use test cases that are different to those provided so verify you have checked it more thoroughly than just using the test program provided.
3. Submit your solution as an answer to the task via the OnTrack submission system. This first submission must be prior to the submission "S" deadline indicated in the unit guide and in OnTrack.
4. If your task has been assessed as requiring a "Redo" or "Resubmit" then you should prepare a new submission. You will have 1 (7 day) calendar week from the day you receive the assessment from the tutor. This usually will mean you should revise the lecture, the readings indicated, and read the unit discussion list for suggestions. After your submission has been corrected and providing it is still before the due deadline you can resubmit.
5. If your task has been assessed as correct, either after step 3 or 4, you can "discuss" with your tutor. This first discussion must occur prior to the discussion "D".
6. Meet with your tutor to demonstrate/discuss your submission. Be on time with respect to the specified discussion deadline.
7. The tutor will ask you both theoretical and practical questions. Questions are likely to cover lecture notes, so attending (or watching) lectures should help you with this compulsory interview part. The tutor will tick off the task as complete, only if you provide a satisfactory answer to these questions.
8. If you cannot answer the questions satisfactorily your task will remain on discussion and you will need to study the topic during the week and have a second discussion the following week.
9. Please note, due to the number of students and time constraints tutors will only be expected to mark and/or discuss your task twice. After this it will be marked as a "Exceeded Feedback".
10. Note that we will not check 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 through the unit.
11. Final note, A "Fail" or "Exceeded Feedback" grade on a task does not mean you have failed the unit. It simply means that you have not demonstrated your understanding of that task through OnTrack. Similarly failing the redemption quiz also does not mean you have failed the unit. You can replace a task with a task from a higher grade.

Appendix A

This section displays the printout produced by the attached Tester class, specifically by its *Main* method. It is based on our solution. The printout is provided here to help with testing your code for potential logical errors. It demonstrates the correct logic rather than an expected printout in terms of text and alignment.

```
Test A: Apply BinarySearch searching for number 333 to array of integer numbers sorted in AscendingIntComparer:
Resulting order: [100,120,122,175,213,236,263,299,312,333,511,596,722,724,752,772,780,958,966,995]
 :: SUCCESS

Test B: Apply BinarySearch searching for number 99 to array of integer numbers sorted in AscendingIntComparer:
Resulting order: [100,120,122,175,213,236,263,299,312,333,511,596,722,724,752,772,780,958,966,995]
 :: SUCCESS

Test C: Apply BinarySearch searching for number 996 to array of integer numbers sorted in AscendingIntComparer:
Resulting order: [100,120,122,175,213,236,263,299,312,333,511,596,722,724,752,772,780,958,966,995]
 :: SUCCESS

Test D: Apply BinarySearch searching for number 333 to array of integer numbers sorted in DescendingIntComparer:
Resulting order: [995,966,958,780,772,752,724,722,596,511,333,312,299,263,236,213,175,122,120,100]
 :: SUCCESS

Test E: Apply BinarySearch searching for number 994 to array of integer numbers sorted in DescendingIntComparer:
Resulting order: [995,966,958,780,772,752,724,722,596,511,333,312,299,263,236,213,175,122,120,100]
 :: SUCCESS

Test F: Apply BinarySearch searching for number 101 to array of integer numbers sorted in DescendingIntComparer:
Resulting order: [995,966,958,780,772,752,724,722,596,511,333,312,299,263,236,213,175,122,120,100]
 :: SUCCESS

Test G: Run a sequence of operations:
Create a new vector by calling 'Vector<int> vector = new Vector<int>(5);'
 :: SUCCESS
Add a sequence of numbers 2, 6, 8, 5, 5, 1, 8, 5, 3, 5, 7, 1, 4, 9
 :: SUCCESS

Test H: Run a sequence of operations:
Check whether the interface IEnumerable<T> is implemented for the Vector<T> class
 :: SUCCESS
Check whether GetEnumerator() method is implemented
 :: SUCCESS

Test I: Run a sequence of operations:
Return the Enumerator of the Vector<T> and check whether it implements IEnumerator<T>
Check the initial value of Current of the Enumerator
```

```
Check the value of Current of the Enumerator after MoveNext() operation
 :: SUCCESS

Test J: Check the content of the Vector<int> by traversing it via 'foreach' statement
 :: SUCCESS

Test K: Run a sequence of operations:
Create a new vector of Student objects by calling 'Vector<Student> students = new Vector<Student>();'
Add student with record: 0[Vicky]
Add student with record: 1[Cindy]
Add student with record: 2[Tom]
Add student with record: 3[Simon]
Add student with record: 4[Richard]
Add student with record: 5[Vicky]
Add student with record: 6[Tom]
Add student with record: 7[Elicia]
Add student with record: 8[Richard]
Add student with record: 9[Cindy]
Add student with record: 10[Vicky]
Add student with record: 11[Guy]
Add student with record: 12[Richard]
Add student with record: 13[Michael]
Print the vector of students via students.ToString();
[0[Vicky],1[Cindy],2[Tom],3[Simon],4[Richard],5[Vicky],6[Tom],7[Elicia],8[Richard],9[Cindy],10[Vicky],11[Guy],12[Richard]
,13[Michael]]
 :: SUCCESS

Test L: Check the content of the Vector<Student> by traversing it via 'foreach' statement
 :: SUCCESS

 ---------- SUMMARY ----------
Tests passed: ABCDEFGHIJKL
```