بسم الله الرحمن الرحيم

Dot Net - Advanced C#

Assignment 03 - Report on Collections in C#

1. Introduction

In the context of .NET, the Collections framework provides various types of classes and interfaces for handling groups of objects, such as arrays, lists, queues, stacks, dictionaries, and more. These collections can store data of various types, and some are designed to be flexible, while others are specialized for specific tasks.

2. Contents

Note:

- ✓ **O(1):** Fast and independent of the size of the data structure.
- ✓ **O(n):** Slower and depends on the size of the data structure, increasing linearly as the size grows.

2.1 NonGeneric Collections ArrayList

- ❖ **Structure**

- Dynamic array that can hold objects of any type.
- Stores elements contiguously in memory.

- **Time Complexity**
  - Access: O(1).
  - Search: O(n).
  - Insert/Delete (end): O(1), Insert/Delete (specific position): O(n).

- **Business Case**
  - Suitable for scenarios where dynamic resizing is needed, and data type safety is not a primary concern.

Example:

```
class Program {
   static void Main() {
          ArrayList arrayList = new ArrayList();
          Console.WriteLine($"Count Of ArrayList ={arrayList.Count}");
          //Count=> Actuall Number Of Elements In Arraylist
          Console.WriteLine($"Count Of ArrayList ={arrayList.Capacity}");
          //Capacity =>Numbers Of Elements That ArrayList Can Hold
          arrayList.Add(1);
          //adding the First Element To The List The Capacity is
          //Increased To _defaultCapacity //4
          //New Array Created At Heap With Size 4
          //Console.WriteLine("After adding Frist Element");
          //Console.WriteLine($"Count Of ArrayList ={arrayList.Count}");//1
```

```
 //Console.WriteLine($"Count Of ArrayList
={arrayList.Capacity}");//4

 //arrayList.Add(2);

 //arrayList.AddRange(new int[]{3, 4});

 //Console.WriteLine($"Count Of ArrayList ={arrayList.Count}");


 //Console.WriteLine($"Count Of ArrayList ={arrayList.Capacity}");

 //arrayList.Add(5);

 ////Creatw a New Array At Heap With Double Size =8

 ////Take Old Element And Adding New Element => To The New
Array

 //Console.WriteLine($"Count Of ArrayList ={arrayList.Count}");//5

 //Console.WriteLine($"Count Of ArrayList
={arrayList.Capacity}");//8


 //ArrayList arrayList02 = new ArrayList(5) { 1, 2, 3, 4, 5 };

 //Console.WriteLine($"Count Of ArrayList ={arrayList02.Count}");

 ////Count=> Actuall Number Of Elements In Arraylist

 //Console.WriteLine($"Count Of ArrayList
={arrayList02.Capacity}");

 //arrayList02.Add(6);

 //Console.WriteLine("After Adding The Six Element");

 //Console.WriteLine($"Count Of ArrayList ={arrayList02.Count}");
```

```
//Console.WriteLine($"Count Of ArrayList
={arrayList02.Capacity}");

//------------------------------------------------------------

//4 int=>4*4 =16 byte [unused Bytes]

//Frese || Delete || Deallocate Unused Bytes

//arrayList02.TrimToSize();

//Console.WriteLine($"Count Of ArrayList
={arrayList02.Count}");//6


//Console.WriteLine($"Count Of ArrayList
={arrayList02.Capacity}");//6

//create A New Array At Heap With Size = Count Of Elements

//Take Old Element + New Elements In New Array

//Old Array Now Is Unreachable


//--------------------

//arrayList02.Add(7);//Casting From Value Type To Rference Type

////Boxing

//arrayList02.Add(8);

//arrayList02.Add("Mohammed");

////arrayList=>Hetrogenous List

////Compiler Can Not Enforce Type Safty At Compilation
```

```
////foreach (int Number in arrayList02)
////{
////    Console.WriteLine(Number);
////}


//int SumResult = SumArrayList(arrayList02);
//Console.WriteLine($"The Result is {SumResult}");
}
```

2.2 List

- **Structure**
  - Generic list providing type safety.
  - Backed by a dynamic array.
- **Time Complexity**
  - Access: O(1).
  - Search: O(n).
  - Insert/Delete (end): O(1), Insert/Delete (specific position): O(n)

- **Business Case**
  - Ideal for type-safe operations with frequent access and dynamic resizing.

Example:

```
class Program {
  static void Main() {
```

```
                List<int> numbers = new List<int>();
                numbers.Add(1);
                numbers.Add(2);
                numbers.Add(3);
               foreach (int Number in numbers)
                {
                  Console.WriteLine(Number);

                }
}
}
```

2.3 Generic Collections - List (Methods)

- **Structure**
  - Generic list providing type safety.
  - Backed by a dynamic array.
- **Time Complexity**
  - Access: O(1).
  - Search: O(n).
  - Insert/Delete (end): O(1), Insert/Delete (specific position): O(n)

Business Case

- Ideal for type-safe operations with frequent access and dynamic resizing.

Example:

```
List<int> Numbers02 = new List<int>() { 1,2,3,1};

//Numbers02.Add(1);//Add One Element

//Numbers02.AddRange(new int[] { 2,3});//Add Range Of Element

//Numbers02.Insert(3,4);//Insert Element Specific Index In List

//Numbers02.InsertRange(4,new int[] {5,6});

// Numbers02.Clear();//Remove All Element In List

//Console.WriteLine(Numbers02.BinarySearch(2));//Print  Number of Potitation

//Console.WriteLine(Numbers02.Contains(9));//return bool t if the number is exist

Console.WriteLine(Numbers02.Capacity);

Console.WriteLine(Numbers02.EnsureCapacity(8));//Increase Capatcity But Chech Ild Capatciy First

Console.WriteLine(Numbers02.IndexOf(1));

Console.WriteLine(Numbers02.LastIndexOf(1));

//foreach (int Number in Numbers02)

//{

//    Console.WriteLine(Number);

//}
```

2.4 Generic Collections – LinkedList

- **Structure:**

- Doubly linked nodes where each node contains data and pointers to the next and previous nodes.

- **Time Complexity:**
  - Access: O(n)
  - Insert/Delete: O(1) for pointer adjustments.
- **Business Case:**
  - Best for scenarios involving frequent insertions and deletions.

Example:

```
class Program {
   static void Main() {
      LinkedList<string> list = new LinkedList<string>();
      list.AddLast("Eshag");
          list.AddFirst("Mohammed");
      list.AddLast("Mester");
      LinkedListNode<string> node = list.Find("Mester");
      list.AddAfter(node, "Eshag");
      list.AddBefore(node, "Mohammed");
      list.Remove("Mester");
      list.RemoveFirst();
      list.RemoveLast();
      Console.WriteLine("Contains 'Mester': " + list.Contains("Mester"));
      Console.WriteLine("First Element: " + list.First.Value);
      Console.WriteLine("Last Element: " + list.Last.Value);
```

```
        // Uncomment this line if you want to clear the list.
        // list.Clear();
        Console.WriteLine("\nRemaining elements in the LinkedList:");
        foreach (var item in list) {
            Console.WriteLine(item);
        }
    }
}
```

2.4 Generic Collections - Stack

- **Structure:**
  - Last-In-First-Out (LIFO) structure.
- **Time Complexity:**
  - Push/Pop: O(1)
  - Search: O(n)
- **Business Case:**
  - Used for recursion, undo operations, and syntax parsing.
- **Example:**

```
class Program {
    static void Main() {
        Stack<int> stack = new Stack<int>();
```

```
        stack.Push(115);
        stack.Push(185);
        stack.Push(132);
        Console.WriteLine("Stack after Push operations: " + string.Join(", ", stack));
        Console.WriteLine("Popped: " + stack.Pop());
        Console.WriteLine("Stack after Pop: " + string.Join(", ", stack));
        Console.WriteLine("Peek (Top element): " + stack.Peek());
        Console.WriteLine("Contains 20: " + stack.Contains(20));
        Console.WriteLine("Contains 50: " + stack.Contains(50));

        int[] stackArray = stack.ToArray();
        Console.WriteLine("Stack converted to Array: " + string.Join(", ",
stackArray));

        stack.Clear();
        Console.WriteLine("Stack after Clear: " + string.Join(", ", stack));
    }
}
```

2.5 Generic Collections - Queue

- **Structure:**
  - First-In-First-Out (FIFO) structure.
- **Time Complexity:**
  - Enqueue/Dequeue: O(1)

- Access/Search: O(n)

- **Business Case:**
  - Useful in scheduling algorithms, breadth-first search, and buffering.

Example:

```
class Program {
   static void Main() {
        Queue<string> queue = new Queue<string>();


      queue.Enqueue("Code");
      queue.Enqueue("First");
      queue.Enqueue("In Egypt");
      Console.WriteLine("Queue after Enqueue operations: " + string.Join(", ", queue)).
      Console.WriteLine("Dequeued: " + queue.Dequeue());
      Console.WriteLine("Queue after Dequeue: " + string.Join(", ", queue));


      Console.WriteLine("Peek (First element): " + queue.Peek());


      Console.WriteLine("Contains 'Code': " + queue.Contains("Code "));
      Console.WriteLine("Contains 'Route': " + queue.Contains("Route"));
      string[] queueArray = queue.ToArray();
      Console.WriteLine("Queue converted to Array: " + string.Join(", ", queueArray));
```

```
        queue.Clear();
        Console.WriteLine("Queue after Clear: " + string.Join(", ", queue));
    }
}
```

3. Comparison of Collections

Aspect	ArrayList	List	LinkedList	Stack	Queue
Structure	Resizable array	Resizable type-safe array	Doubly linked list	LIFO	FIFO
Access Time	O(1)	O(1)	O(n)	O(n)	O(n)
Insertion Time	O(1) at end, O(n) elsewhere	O(1) at end, O(n) elsewhere	O(1)	O(1)	O(1)
Deletion Time	O(n)	O(n)	O(1)	O(1)	O(1)
Use Cases	Dynamic resizing without type safety	Type-safe dynamic resizing	Frequent insert/delete ops	Recursion, undo operations	Scheduling, BFS, Buffering

4. Conclusion

Each collection in C# offers distinct advantages. ArrayList provides flexibility but lacks type safety. List combines type safety and performance for general purposes. LinkedList excels in frequent insertions/deletions, while Stack and Queue cater to specialized LIFO and FIFO workflows. The choice depends on specific application requirements.