What is Dictionary in C#

In C#, Dictionary is a generic collection that is generally used to store key/value pairs. The working of Dictionary is quite similar to the non-generic hashtable. The advantage of a Dictionary is, it is a generic type. Dictionary is defined under System.Collection.Generic namespace. It is dynamic in nature means the size of the dictionary grows according to the need.

Dictionary<TKey, TValue>(): This constructor is used to create an instance of the Dictionary<TKey, TValue> class that is empty, has the default initial capacity, and uses the default equality comparison for the key type as follows:

Let's see an example of a generic Dictionary<TKey, TValue> class that stores elements using Add() method and iterates elements using for-each loop. Here, we are using KeyValuePair class to get the key and value.

```
using System;
using System.Collections.Generic;

public class DictionaryExample
{
   public static void Main(string[] args)
   {
      Dictionary<string, string> names = new Dictionary<string, string>();
      names.Add("1","Sonoo");
      names.Add("2","Peter");
      names.Add("3","James");
      names.Add("4","Ratan");
      names.Add("5","Irfan");

      foreach (KeyValuePair<string, string> kv in names)
      {
         Console.WriteLine(kv.Key+" "+kv.Value);
      }
   }
}
```

How to remove elements from the Dictionary?

In Dictionary, you are allowed to remove elements from the Dictionary. Dictionary<TKey, TValue> class provides two different methods to remove elements, and the methods are:

- Clear: This method removes all keys and values from the Dictionary<TKey, TValue>.
- Remove: This method removes the value with the specified key from the Dictionary<TKey, TValue>.

```
using System;
using System.Collections.Generic;

class GFG {

        // Main Method
    static public void Main() {

        // Creating a dictionary
        // using Dictionary<TKey,TValue> class
        Dictionary<int, string> My_dict =
                    new Dictionary<int, string>();

          // Adding key/value pairs in the
          // Dictionary Using Add() method
          My_dict.Add(1123, "Welcome");
          My_dict.Add(1124, "to");
          My_dict.Add(1125, "GeeksforGeeks");

          // Before Remove() method
          foreach(KeyValuePair<int, string> ele in My_dict)
          {
              Console.WriteLine("{0} and {1}",
                          ele.Key, ele.Value);
          }
          Console.WriteLine();
```

```
            // Using Remove() method
            My_dict.Remove(1123);

            // After Remove() method
            foreach(KeyValuePair<int, string> ele in My_dict)
            {
                Console.WriteLine("{0} and {1}",
                              ele.Key, ele.Value);
            }
            Console.WriteLine();


            // Using Clear() method
            My_dict.Clear();

            Console.WriteLine("Total number of key/value "+
             "pairs present in My_dict:{0}", My_dict.Count);

    }
}
```

Output:

```
1123 and Welcome
1124 and to
1125 and GeeksforGeeks

1124 and to
1125 and GeeksforGeeks

Total number of key/value pairs present in My_dict:0
```

Sets in C#

Sets in C# is a HashSet. HashSet in C# eliminates duplicate strings or elements in an array. In C#, it is an optimized set collection

Characteristics of HashSet Class:

- The HashSet<**T**> class provides high-performance set operations. A set is a collection that contains no duplicate elements, and whose elements are in no particular order.
- The capacity of a HashSet<**T**> object is the number of elements that the object can hold.
- A HashSet<**T**> object's capacity automatically increases as elements are added to the object.
- A HashSet<**T**> collection is not sorted and cannot contain duplicate elements.
- HashSet<**T**> provides many mathematical set operations, such as set addition (unions) and set subtraction.

To declare HashSet –

```
var h = new HashSet<string>(arr1);
```

Above, we have set the already declared array arr1 in the HashSet.

How to remove elements from the HashSet?

In HashSet, you are allowed to remove elements from the HashSet. HashSet<T> class provides three different methods to remove elements and the methods are:

- **Remove(T):** This method is used to remove the specified element from a HashSet object.

- **RemoveWhere(Predicate):** This method is used to remove all elements that match the conditions defined by the specified predicate from a HashSet collection.
- **Clear:** This method is used to remove all elements from a HashSet object.

```
using System;
using System.Collections.Generic;

class GFG {

    // Main Method
    static public void Main()
    {

        // Creating HashSet
        // Using HashSet class
        HashSet<string> myhash = new HashSet<string>();

        // Add the elements in HashSet
        // Using Add method
        myhash.Add("C");
        myhash.Add("C++");
        myhash.Add("C#");
        myhash.Add("Java");
        myhash.Add("Ruby");

        // Before using Remove method
        Console.WriteLine("Total number of elements present (Before Remov
                          " in myhash: {0}", myhash.Count);
```

```
    // Before using Remove method
    Console.WriteLine("Total number of elements present (Before Removal)"
                      " in myhash: {0}", myhash.Count);

    // Remove element from HashSet
    // Using Remove method
    myhash.Remove("Ruby");

    // After using Remove method
    Console.WriteLine("Total number of elements present (After Removal)"+
                      " in myhash: {0}", myhash.Count);

    // Remove all elements from HashSet
    // Using Clear method
    myhash.Clear();
    Console.WriteLine("Total number of elements present"+
                       " in myhash:{0}", myhash.Count);
}
```

Output:

```
Total number of elements present in myhash: 5
Total number of elements present in myhash: 4
Total number of elements present in myhash:0
```

Queue in C#

Queue represents a ***first-in, first-out*** collection of objects. It is used when you need first-in, first-out access to items. When you add an item to the list, it is called **enqueue**, and when you remove an item, it is called **dequeue**. This class comes under **System. Collections** namespace and implements *ICollection, IEnumerable, and ICloneable* interfaces.

Characteristics of Queue Class:

- **Enqueue** adds an element to the end of the Queue.
- **Dequeue** removes the oldest element from the start of the Queue.
- **Peek** returns the oldest element that is at the start of the Queue but does not remove it from the Queue.
- The **capacity** of a Queue is the number of elements the Queue can hold.
- As elements are added to a Queue, the capacity is automatically increased as required by reallocating the internal array.
- Queue accepts **null** as a valid value for reference types and allows duplicate elements.

```
using System;
using System.Collections;

class GFG {

    // Driver code
    public static void Main()
    {

        // Creating a Queue
        Queue myQueue = new Queue();

        // Inserting the elements into the Queue
        myQueue.Enqueue("one");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);

        myQueue.Enqueue("two");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");
```

```
        Console.WriteLine(myQueue.Count);

        myQueue.Enqueue("three");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);

        myQueue.Enqueue("four");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);

        myQueue.Enqueue("five");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);
```

```
        myQueue.Enqueue("six");

        // Displaying the count of elements
        // contained in the Queue
        Console.Write("Total number of elements in the Queue are : ");

        Console.WriteLine(myQueue.Count);
    }
}
```

Output:

```
Total number of elements in the Queue are : 1
Total number of elements in the Queue are : 2
Total number of elements in the Queue are : 3
Total number of elements in the Queue are : 4
Total number of elements in the Queue are : 5
Total number of elements in the Queue are : 6
```

Stack in C#

Stack is a special type of collection that stores elements in LIFO style (Last In First Out). C# includes the generic Stack<T> and non-generic Stack collection classes. It is recommended to use the generic Stack<T> collection. Stack is useful to store temporary data in LIFO style, and you might want to delete an element after retrieving its value.

Stack<T> Characteristics

- Stack<T> is the Last In First Out collection.
- It comes under System.Collection.Generic namespace.

- Stack<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
- Elements can be added using the Push() method. Cannot use collection-initializer syntax.
- Elements can be retrieved using the Pop() and the Peek() methods. It does not support an indexer.

Creating a Stack

You can create an object of the Stack<T> by specifying a type parameter for the type of elements it can store. The following example creates and adds elements in the Stack<T> using the Push() method. Stack allows null (for reference types) and duplicate values.

```
Stack<int> myStack = new Stack<int>();
myStack.Push(1);
myStack.Push(2);
myStack.Push(3);
myStack.Push(4);

foreach (var item in myStack)
     Console.Write(item + ","); //prints 4,3,2,1,
```

You can also create a Stack from an array, as shown below.

```
int[] arr = new int[]{ 1, 2, 3, 4};
Stack<int> myStack = new Stack<int>(arr);

foreach (var item in myStack)
     Console.Write(item + ","); //prints 4,3,2,1,
```

Stack<T> Properties and Methods:

Property	Usage
Count	Returns the total count of elements in the Stack.

Method	Usage
Push(T)	Inserts an item at the top of the stack.
Peek()	Returns the top item from the stack.
Pop()	Removes and returns items from the top of the stack.
Contains(T)	Checks whether an item exists in the stack or not.
Clear()	Removes all items from the stack.
