C# Collection

We have learned about an array in the previous section. C# also includes specialized classes that hold many values or objects in a specific series, that are called 'collection'.

There are two types of collections available in C#:

non-generic collections and

[generic collections](https://www.tutorialsteacher.com/csharp/csharp-generic-collections). We will learn about non-generic collections in this section.

The *System.Collections* namespace includes the interfaces and classes for the non-generic collections. The following diagram illustrates the hierarchy of the interfaces and classes for the non-generic collections.

IEnumerator: The [IEnumerator](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerator?view=netframework-4.7.2) interface supports a simple iteration over a non-generic collection. It includes methods and property which can be implemented to support easy iteration using foreach loop.

IEnumerable: The [IEnumerable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable?view=netframework-4.7.2) interface includes GetEnumerator() method which returns an object of IEnumerator.

So, all the built-in collection classes and custom collection classes must implement IEnumerator and IEnumerable interfaces for easy iteration using foreach loop.

ICollection: The [ICollection](https://docs.microsoft.com/en-us/dotnet/api/system.collections.icollection?view=netframework-4.7.2) interface is the base interface for all the collections that defines sizes, enumerators, and synchronization methods for all non-generic collections. The Queue and Stack collection implement ICollection inferface.

IList: The [IList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ilist?view=netframework-4.7.2) interface includes properties and methods to add, insert, remove elements in the collection and also individual element can be accessed by index. The ArrayList and BitArray collections implement IList interface.

IDictionary: The [IDictionary](https://docs.microsoft.com/en-us/dotnet/api/system.collections.idictionary?view=netframework-4.7.2) interface represents a non-generic collection of key/value pairs. The Hashtable and SortedList implement IDictionary interface and so they store key/value pairs.

As you can see from the diagram, ArrayList, BitArray, Hashtable, SortedList, Queue, and Stack collections implement different interfaces and so, they are used for the different purposes.

| Non-generic Collections | Usage |
| --- | --- |
| [ArrayList](https://www.tutorialsteacher.com/csharp/csharp-arraylist) | ArrayList stores objects of any type like an array. However, there is no need to specify the size of the ArrayList like with an array as it grows automatically. |
| [SortedList](https://www.tutorialsteacher.com/csharp/csharp-sortedlist) | SortedList stores key and value pairs. It automatically arranges elements in ascending order of key by default. C# includes both, generic and non-generic SortedList collection. |
| [Stack](https://www.tutorialsteacher.com/csharp/csharp-stack) | Stack stores the values in LIFO style (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. C# includes both, generic and non-generic Stack. |
| [Queue](https://www.tutorialsteacher.com/csharp/csharp-queue) | Queue stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. C# includes generic and non-generic Queue. |
| [Hashtable](https://www.tutorialsteacher.com/csharp/csharp-hashtable) | Hashtable stores key and value pairs. It retrieves the values by comparing the hash value of the keys. |
| BitArray | BitArray manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0). |

# C# - ArrayList

ArrayList is a non-generic type of collection in C#. It can contain elements of any data types. It is similar to an [array](https://www.tutorialsteacher.com/csharp/array-csharp), except that it grows automatically as you add items in it. Unlike an array, you don't need to specify the size of ArrayList.

| Methods | Description |
| --- | --- |
| [Add()/AddRange()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#add) | Add() method adds single elements at the end of ArrayList. AddRange() method adds all the elements from the specified collection into ArrayList. |
| [Insert()/InsertRange()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#insert) | Insert() method insert a single elements at the specified index in ArrayList. InsertRange() method insert all the elements of the specified collection starting from specified index in ArrayList. |
| [Remove()/RemoveRange()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#remove) | Remove() method removes the specified element from the ArrayList. RemoveRange() method removes a range of elements from the ArrayList. |
| [RemoveAt()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#removeat) | Removes the element at the specified index from the ArrayList. |
| [Sort()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#sort) | Sorts entire elements of the ArrayList. |
| [Reverse()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#sort) | Reverses the order of the elements in the entire ArrayList. |
| [Contains](https://www.tutorialsteacher.com/csharp/csharp-arraylist#contains) | Checks whether specified element exists in the ArrayList or not. Returns true if exists otherwise false. |
| Clear | Removes all the elements in ArrayList. |
| CopyTo | Copies all the elements or range of elements to compitible Array. |
| GetRange | Returns specified number of elements from specified index from ArrayList. |
| IndexOf | Search specified element and returns zero based index if found. Returns -1 if element not found. |
| ToArray | Returns compitible array from an ArrayList. |

# C# - SortedList

The SortedList collection stores key-value pairs in the ascending order of key by default. SortedList class implements IDictionary & ICollection interfaces, so elements can be accessed both by key and index.

C# includes two types of SortedList, [generic SortedList](https://www.tutorialsteacher.com/csharp/csharp-generic-sortedlist) and non-generic SortedList. Here, we will learn about non-generic SortedList.

The following diagram illustrates the non-generic SortedList hierarchy.

C# SortedList

## Important Properties and Methods of SortedList

| Property | Description |
| --- | --- |
| Capacity | Gets or sets the number of elements that the SortedList instance can store. |
| Count | Gets the number of elements actually contained in the SortedList. |
| IsFixedSize | Gets a value indicating whether the SortedList has a fixed size. |
| IsReadOnly | Gets a value indicating whether the SortedList is read-only. |
| Item | Gets or sets the element at the specified key in the SortedList. |
| Keys | Get list of keys of SortedList. |
| Values | Get list of values in SortedList. |

| Method | Description |
| --- | --- |
| Add(object key, object value) | Add key-value pairs into SortedList. |
| Remove(object key) | Removes element with the specified key. |
| RemoveAt(int index) | Removes element at the specified index. |
| Contains(object key) | Checks whether specified key exists in SortedList. |
| Clear() | Removes all the elements from SortedList. |
| GetByIndex(int index) | Returns the value by index stored in internal array |
| GetKey(int index) | Returns the key stored at specified index in internal array |
| IndexOfKey(object key) | Returns an index of specified key stored in internal array |
| IndexOfValue(object value) | Returns an index of specified value stored in internal array |

# C# - Hashtable

C# includes Hashtable collection in *System.Collections* namespace, which is similar to generic [Dictionary](https://www.tutorialsteacher.com/csharp/csharp-dictionary) collection. The Hashtable collection stores key-value pairs. It optimizes lookups by computing the hash code of each key and stores it in a different bucket internally and then matches the hash code of the specified key at the time of accessing values.

## Important Propertis and Methods of Hashtable

| Property | Description |
| --- | --- |
| Count | Gets the total count of key/value pairs in the Hashtable. |
| IsReadOnly | Gets boolean value indicating whether the Hashtable is read-only. |
| Item | Gets or sets the value associated with the specified key. |
| Keys | Gets an ICollection of keys in the Hashtable. |
| Values | Gets an ICollection of values in the Hashtable. |

| Methods | Usage |
| --- | --- |
| [Add](https://www.tutorialsteacher.com/csharp/csharp-hashtable#add) | Adds an item with a key and value into the hashtable. |
| [Remove](https://www.tutorialsteacher.com/csharp/csharp-hashtable#remove) | Removes the item with the specified key from the hashtable. |
| Clear | Removes all the items from the hashtable. |
| [Contains](https://www.tutorialsteacher.com/csharp/csharp-hashtable#contains) | Checks whether the hashtable contains a specific key. |
| [ContainsKey](https://www.tutorialsteacher.com/csharp/csharp-hashtable#contains) | Checks whether the hashtable contains a specific key. |
| [ContainsValue](https://www.tutorialsteacher.com/csharp/csharp-hashtable#contains) | Checks whether the hashtable contains a specific value. |
| GetHash | Returns the hash code for the specified key. |

## Add key-value into Hashtable

The Add() method adds an item with a key and value into the Hashtable. Key and value can be of any data type. Key cannot be null whereas value can be null.

Add() Signature: *void Add(object key, object value);*

Example: Add()

Hashtable ht = new Hashtable();

ht.Add(1, "One");

ht.Add(2, "Two");

ht.Add(3, "Three");

ht.Add(4, "Four");

ht.Add(5, null);

ht.Add("Fv", "Five");

ht.Add(8.5F, 8.5);

# C# - Stack

C# includes a special type of collection which stores elements in LIFO style (Last In First Out). C# includes a generic and non-generic Stack. Here, you are going to learn about the non-generic stack.

Stack allows null value and also duplicate values. It provides a Push() method to add a value and Pop() or Peek() methods to retrieve values.

## mportant Properties and Methods of Stack:

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

| Method | Usage |
| --- | --- |
| [Push](https://www.tutorialsteacher.com/csharp/csharp-stack#push) | Inserts an item at the top of the stack. |
| [Peek](https://www.tutorialsteacher.com/csharp/csharp-stack#peek) | Returns the top item from the stack. |
| [Pop](https://www.tutorialsteacher.com/csharp/csharp-stack#pop) | Removes and returns items from the top of the stack. |
| [Contains](https://www.tutorialsteacher.com/csharp/csharp-stack#contains) | Checks whether an item exists in the stack or not. |
| [Clear](https://www.tutorialsteacher.com/csharp/csharp-stack#clear) | Removes all items from the stack. |

## Add Values into Stack

The Push() method adds values into the Stack. It allows value of any datatype.

Push() method signature:*void Push(object obj);*

Example: Push()

Stack myStack = new Stack();

myStack.Push("Hello!!");

myStack.Push(null);

myStack.Push(1);

myStack.Push(2);

myStack.Push(3);

myStack.Push(4);

myStack.Push(5);

## Accessing Stack Elements

You can retrieve stack elements by various ways. Use a foreach statement to iterate the Stack collection and get all the elements in LIFO style.

Example: Access Stack

Stack myStack = new Stack();

myStack.Push("Hello!!");

myStack.Push(null);

myStack.Push(1);

myStack.Push(2);

myStack.Push(3);

myStack.Push(4);

myStack.Push(5);

foreach (var itm in myStack)

Console.Write(itm);

# C# - Queue

C# includes a Queue collection class in the *System.Collection* namespace. Queue stores the elements in FIFO style (First In First Out), exactly opposite of the [Stack](https://www.tutorialsteacher.com/csharp/csharp-stack) collection. It contains the elements in the order they were added.

Queue collection allows multiple null and duplicate values. Use the Enqueue() method to add values and the Dequeue() method to retrieve the values from the Queue.

## Important Properties and Methods of Queue:

| Property | Usage |
| --- | --- |
| Count | Returns the total count of elements in the Queue. |

| Method | Usage |
| --- | --- |
| [Enqueue](https://www.tutorialsteacher.com/csharp/csharp-queue#enqueue) | Adds an item into the queue. |
| [Dequeue](https://www.tutorialsteacher.com/csharp/csharp-queue#dequeue) | Removes and returns an item from the beginning of the queue. |
| [Peek](https://www.tutorialsteacher.com/csharp/csharp-queue#peek) | Returns an first item from the queue |
| [Contains](https://www.tutorialsteacher.com/csharp/csharp-queue#contains) | Checks whether an item is in the queue or not |
| [Clear](https://www.tutorialsteacher.com/csharp/csharp-queue#clear) | Removes all the items from the queue. |
| TrimToSize | Sets the capacity of the queue to the actual number of items in the queue. |

## Add elements in Queue

Queue is a non-generic collection. So you can add elements of any datatype into a queue using the Enqueue() method.

Enqueue() signature: *void Enqueue(object obj)*

Example: Enqueue()

Queue queue = new Queue();

queue.Enqueue(3);

queue.Enqueue(2);

queue.Enqueue(1);

queue.Enqueue("Four");

## Access Queue

Dequeue() method is used to retrieve the top most element in a queue collection. Dequeue() removes and returns a first element from a queue because the queue stores elements in FIFO order. Calling Dequeue() method on empty queue will throw InvalidOperation exception. So always check that the total count of a queue is greater than zero before calling the Dequeue() method on a queue.

Dequeue() method signature: *object Dequeue()*

Example: Dequeue()

Queue queue = new Queue();

queue.Enqueue(3);

queue.Enqueue(2);

queue.Enqueue(1);

queue.Enqueue("Four");

Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);

while (queue.Count > 0)

Console.WriteLine(queue.Dequeue());

Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);

# Generics in C#

Generics introduced in C# 2.0. Generics allow you to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at compile time.

A generic class can be defined using angle brackets <>. For example, the following is a simple generic class with a generic member variable, generic method and property.

C# - Generic Collection

You have learned about the [collection](https://www.tutorialsteacher.com/csharp/csharp-collection) in the previous section, e.g. [ArrayList](https://www.tutorialsteacher.com/csharp/csharp-arraylist), BitArray, [SortedList](https://www.tutorialsteacher.com/csharp/csharp-sortedlist), [Queue](https://www.tutorialsteacher.com/csharp/csharp-queue), [Stack](https://www.tutorialsteacher.com/csharp/csharp-stack) and [Hashtable](https://www.tutorialsteacher.com/csharp/csharp-hashtable). These types of collections can store any type of items. For example, ArrayList can store items of different data types:

Example: C# ArrayList Collection

ArrayList arList = new ArrayList();

arList.Add(1);

arList.Add("Two");

arList.Add(true);

arList.Add(100.45);

arList.Add(DateTime.Now);

The limitation of these collections is that while retrieving items, you need to cast into the appropriate data type, otherwise the program will throw a runtime exception. It also affects on performance, because of boxing and unboxing.

To overcome this problem, C# includes generic collection classes in the ***System.Collections.Generic*** namespace.

The following are widely used generic collections:

| Generic Collections | Description |
| --- | --- |
| [List<T>](https://www.tutorialsteacher.com/csharp/csharp-list) | Generic List<T> contains elements of specified type. It grows automatically as you add elements in it. |
| [Dictionary<TKey,TValue>](https://www.tutorialsteacher.com/csharp/csharp-dictionary) | Dictionary<TKey,TValue> contains key-value pairs. |
| [SortedList<TKey,TValue>](https://www.tutorialsteacher.com/csharp/csharp-generic-sortedlist) | SortedList stores key and value pairs. It automatically adds the elements in ascending order of key by default. |
| Hashset<T> | Hashset<T> contains non-duplicate elements. It eliminates duplicate elements. |
| Queue<T> | Queue<T> stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. |
| Stack<T> | Stack<T> stores the values as LIFO (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. |

A generic collection gets all the benefit of generics. It doesn't need to do boxing and unboxing while storing or retrieving items and so performance is improved.

# C# - List<T>

You have already learned about ArrayList in the previous section. An ArrayList resizes automatically as it grows. The List<T> collection is the same as an ArrayList except that List<T> is a generic collection whereas ArrayList is a non-generic collection.

The following diagram illustrates the List<T> hierarchy.

| Property | Usage |
| --- | --- |
| Items | Gets or sets the element at the specified index |
| Count | Returns the total number of elements exists in the List<T> |

| Method | Usage |
| --- | --- |
| Add | Adds an element at the end of a List<T>. |
| AddRange | Adds elements of the specified collection at the end of a List<T>. |
| BinarySearch | Search the element and returns an index of the element. |
| Clear | Removes all the elements from a List<T>. |
| Contains | Checks whether the speciied element exists or not in a List<T>. |
| Find | Finds the first element based on the specified predicate function. |
| Foreach | Iterates through a List<T>. |
| Insert | Inserts an element at the specified index in a List<T>. |
| InsertRange | Inserts elements of another collection at the specified index. |
| Remove | Removes the first occurence of the specified element. |
| RemoveAt | Removes the element at the specified index. |
| RemoveRange | Removes all the elements that match with the supplied predicate function. |
| Sort | Sorts all the elements. |
| TrimExcess | Sets the capacity to the actual number of elements. |
| TrueForAll | Determines whether every element in theÂ List<T> matches the conditions defined by the specified predicate. |

## Add Elements into List

Use the IList.Add() method to add an element into a List collection. The following example adds int value into a List<T> of *int* type.

Add() signature: *void Add(T item)*

Example: Adding elements into List

IList<int> intList = new List<int>();

intList.Add(10);

intList.Add(20);

intList.Add(30);

intList.Add(40);

IList<string> strList = new List<string>();

strList.Add("one");

strList.Add("two");

strList.Add("three");

strList.Add("four");

strList.Add("four");

strList.Add(null);

strList.Add(null);

IList<Student> studentList = new List<Student>();

studentList.Add(new Student());

studentList.Add(new Student());

studentList.Add(new Student());

# C# - Dictionary<TKey, TValue>

The Dictionary<TKey, TValue> collection in C# is same as English dictionary. English dictionary is a collection of words and their definitions, often listed alphabetically in one or more specific languages. In the same way, the Dictionary in C# is a collection of Keys and Values, where key is like word and value is like definition.

The Dictionary<TKey, TValue> class is a generic collection class in the System.Collection.Generics namespace. TKey denotes the type of key and TValue is the type of TValue.

A Dictionary object can be assigned to a variable of IDictionary<Tkey, TValue> or Dictionary<TKey, Tvalue> class.

Example: Dictionary Initialization

IDictionary<int, string> dict = new Dictionary<int, string>();

//or

Dictionary<int, string> dict = new Dictionary<int, string>();

In the above example, we have specified types of key and value while declaring a dictionary object. An int is a type of key and string is a type of value that will be stored into a dictionary object named dict. You can use any valid C# data type for keys and values.

It is recommended to program to the interface rather than to the class. So, use IDictionary<TKey, TValue> type variable to initialize a dictionary object.

 Note:

Dictionary cannot include duplicate or null keys, where as values can be duplicated or set as null. Keys must be unique otherwise it will throw a runtime exception.

## Important Properties and Methods of IDictionary

| Property | Description |
| --- | --- |
| Count | Gets the total number of elements exists in the Dictionary<TKey,TValue>. |
| IsReadOnly | Returns a boolean indicating whether the Dictionary<TKey,TValue> is read-only. |
| Item | Gets or sets the element with the specified key in the Dictionary<TKey,TValue>. |
| Keys | Returns collection of keys of Dictionary<TKey,TValue>. |
| Values | Returns collection of values in Dictionary<TKey,TValue>. |

| Method | Description |
| --- | --- |
| Add | Adds an item to the Dictionary collection. |
| Add | Add key-value pairs in Dictionary<TKey, TValue> collection. |
| Remove | Removes the first occurrence of specified item from the Dictionary<TKey, TValue>. |
| Remove | Removes the element with the specified key. |
| ContainsKey | Checks whether the specified key exists in Dictionary<TKey, TValue>. |
| ContainsValue | Checks whether the specified key exists in Dictionary<TKey, TValue>. |
| Clear | Removes all the elements from Dictionary<TKey, TValue>. |
| TryGetValue | Returns true and assigns the value with specified key, if key does not exists then return false. |

## Add Elements into Dictionary

Use Add() method to add the key-value pair in dictionary.

Add() Signature: *void Add(TKey, Tvalue)*

Example: Add elements in dictionary

IDictionary<int, string> dict = new Dictionary<int, string>();

dict.Add(1,"One");

dict.Add(2,"Two");

dict.Add(3,"Three");

Custom Comparer

public class Student

{

public int StudentID { get; set; }

public string StudentName { get; set; }

}

class StudentDictionaryComparer : IEqualityComparer<KeyValuePair<int,Student>>

{

public bool Equals(KeyValuePair<int, Student> x, KeyValuePair<int, Student> y)

{

if (x.Key == y.Key && (x.Value.StudentID == y.Value.StudentID) && (x.Value.StudentName == y.Value.StudentName))

return true;

return false;

}

public int GetHashCode(KeyValuePair<int, Student> obj)

{

return obj.Key.GetHashCode();

}

}

class Program

{

static void Main(string[] args)

{

IDictionary<int, Student> studentDict = new Dictionary<int, Student>()

{

{ 1, new Student(){ StudentID =1, StudentName = "Bill"}},

{ 2, new Student(){ StudentID =2, StudentName = "Steve"}},

{ 3, new Student(){ StudentID =3, StudentName = "Ram"}}

};

Student std = new Student(){ StudentID = 1, StudentName = "Bill"};

KeyValuePair<int, Student> elementToFind = new KeyValuePair<int, Student>(1, std);

bool result = studentDict.Contains(elementToFind, new StudentDictionaryComparer()); // returns true

Console.WriteLine(result);

}

}

# C# - SortedList<TKey, TValue>

The generic SortedList SortedList<TKey, TValue> represents a collection of key-value pairs that are sorted by key based on associated [IComparer<T>](https://msdn.microsoft.com/en-us/library/8ehhxeaf(v=vs.110).aspx). A SortedList collection stores key and value pairs in ascending order of key by default.

C# includes two type of SortedList, generic SortedList and [non-generic SortedList](https://www.tutorialsteacher.com/csharp/csharp-sortedlist). Generic SortedList denotes with angel bracket: SortedList<TKey,TValue> where TKey is for type of key and TValue is for type of value. Non-generic type do not specify the type of key and values.

## Important Properties and Methods of Generic SortedList

| Property | Description |
| --- | --- |
| Capacity | Gets or sets the number of elements that the SortedList<TKey,TValue> can store. |
| Count | Gets the total number of elements exists in the SortedList<TKey,TValue>. |
| IsReadOnly | Returns a boolean indicating whether the SortedList<TKey,TValue> is read-only. |
| Item | Gets or sets the element with the specified key in the SortedList<TKey,TValue>. |
| Keys | Get list of keys of SortedList<TKey,TValue>. |
| Values | Get list of values in SortedList<TKey,TValue>. |

| Method | Description |
| --- | --- |
| Add | Add key-value pairs into SortedList<TKey, TValue>. |
| Remove | Removes element with the specified key. |
| RemoveAt | Removes element at the specified index. |
| ContainsKey | Checks whether the specified key exists in SortedList<TKey, TValue>. |
| ContainsValue | Checks whether the specified key exists in SortedList<TKey, TValue>. |
| Clear | Removes all the elements from SortedList<TKey, TValue>. |
| IndexOfKey | Returns an index of specified key stored in internal array of SortedList<TKey, TValue>. |
| IndexOfValue | Returns an index of specified value stored in internal array of SortedList<TKey, TValue> |
| TryGetValue | Returns true and assigns the value with specified key, if key does not exists then return false. |

## Add Elements into SortedList

Use the Add() method to add key value pairs into a SortedList. The key cannot be null, but the value can be null. Also, the datatype of key and value must be same as specified, otherwise it will give compile time error.

Add() method signature: *void Add(TKey key, TValue value)*

The following example shows how to add key-value pair in the generic SortedList collection.

Example:Add Elements into SortedList<TKey, TValue>

SortedList<int,string> sortedList1 = new SortedList<int,string>();

sortedList1.Add(3, "Three");

sortedList1.Add(4, "Four");

sortedList1.Add(1, "One");

sortedList1.Add(5, "Five");

sortedList1.Add(2, "Two");

SortedList<string,int> sortedList2 = new SortedList<string,int>();

sortedList2.Add("one", 1);

sortedList2.Add("two", 2);

sortedList2.Add("three", 3);

sortedList2.Add("four", 4);

// Compile time error: cannot convert from <null> to <int>

// sortedList2.Add("Five", null);

SortedList<double,int?> sortedList3 = new SortedList<double,int?>();

sortedList3.Add(1.5, 100);

sortedList3.Add(3.5, 200);

sortedList3.Add(2.4, 300);

sortedList3.Add(2.3, null);

sortedList3.Add(1.1, null);

# C# HashSet<T>

C# HashSet class can be used to store, remove or view elements. It does not store duplicate elements. It is suggested to use HashSet class if you have to store only unique elements. It is found in System.Collections.Generic namespace.

### C# HashSet<T> example

Let's see an example of generic HashSet<T> class that stores elements using Add() method and iterates elements using for-each loop.

1. **using** System;
2. **using** System.Collections.Generic;
4. **public** **class** HashSetExample
5. {
6. **public** **static** **void** Main(**string**[] args)
7. {
8. // Create a set of strings
9. var names = **new** HashSet<**string**>();
10. names.Add("Sonoo");
11. names.Add("Ankit");
12. names.Add("Peter");
13. names.Add("Irfan");
14. names.Add("Ankit");//will not be added
16. // Iterate HashSet elements using foreach loop
17. **foreach** (var name **in** names)
18. {
19. Console.WriteLine(name);
20. }
21. }
22. }

# C# SortedSet<T>

C# SortedSet class can be used to store, remove or view elements. It maintains ascending order and does not store duplicate elements. It is suggested to use SortedSet class if you have to store unique elements and maintain ascending order. It is found in System.Collections.Generic namespace.

### C# SortedSet<T> example

Let's see an example of generic SortedSet<T> class that stores elements using Add() method and iterates elements using for-each loop.

1. **using** System;
2. **using** System.Collections.Generic;
4. **public** **class** SortedSetExample
5. {
6. **public** **static** **void** Main(**string**[] args)
7. {
8. // Create a set of strings
9. var names = **new** SortedSet<**string**>();
10. names.Add("Sonoo");
11. names.Add("Ankit");
12. names.Add("Peter");
13. names.Add("Irfan");
14. names.Add("Ankit");//will not be added
16. // Iterate SortedSet elements using foreach loop
17. **foreach** (var name **in** names)
18. {
19. Console.WriteLine(name);
20. }
21. }
22. }

# C# LinkedList<T>

C# LinkedList<T> class uses the concept of linked list. It allows us to insert and delete elements fastly. It can have duplicate elements. It is found in System.Collections.Generic namespace.

It allows us to add and remove element at before or last index.

### C# LinkedList<T> example

Let's see an example of generic LinkedList<T> class that stores elements using AddLast() and AddFirst() methods and iterates elements using for-each loop.

1. **using** System;
2. **using** System.Collections.Generic;
4. **public** **class** LinkedListExample
5. {
6. **public** **static** **void** Main(**string**[] args)
7. {
8. // Create a list of strings
9. var names = **new** LinkedList<**string**>();
10. names.AddLast("Sonoo Jaiswal");
11. names.AddLast("Ankit");
12. names.AddLast("Peter");
13. names.AddLast("Irfan");
14. names.AddFirst("John");//added to first index
16. // Iterate list element using foreach loop
17. **foreach** (var name **in** names)
18. {
19. Console.WriteLine(name);
20. }
21. }
22. }

# C# SortedDictionary<TKey, TValue>

C# SortedDictionary<TKey, TValue> class uses the concept of hashtable. It stores values on the basis of key. It contains unique keys and maintains ascending order on the basis of key. By the help of key, we can easily search or remove elements. It is found in System.Collections.Generic namespace.

### C# SortedDictionary<TKey, TValue> example

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

1. **using** System;
2. **using** System.Collections.Generic;
4. **public** **class** SortedDictionaryExample
5. {
6. **public** **static** **void** Main(**string**[] args)
7. {
8. SortedDictionary<**string**, **string**> names = **new** SortedDictionary<**string**, **string**>();
9. names.Add("1","Sonoo");
10. names.Add("4","Peter");
11. names.Add("5","James");
12. names.Add("3","Ratan");
13. names.Add("2","Irfan");
14. **foreach** (KeyValuePair<**string**, **string**> kv **in** names)
15. {
16. Console.WriteLine(kv.Key+" "+kv.Value);
17. }
18. }
19. }