**nCOLLECTIONS**

Collections are types that holds a set of related data object together and treat them as single object on which you can perform operation like searching and iterating over each item.

Arrays are declared with a specific size of array. So, whenever we want to change the size of array, then we can copy the array into new increased array.After that old array is destroyed. Or we can also use the Array.resize(copying old values to new one with increased size). Array is fixed size. Performing resizing is a tedious task.

In array, we cant add a new value in middle of array and deleting or removing values from the middle of array.

So, to overcome these problems in an array, we have the collections also called as dynamic arrays.

In collections, we have auto resizing, and it is possible to insert or delete values from the middle of the array.

The collections introduces in 1.0 are non generic collections: they are stack, queue, hashtable,Arraylist,sorted list.

The advantages of these collections is that these can be used directly, but in traditional languages like C we have to define stack, queue and all and then use them.

Difference between array and arraylist

1. Array is fixed length, while array list is variable length
2. We cant insert or delete items in array while in arraylist it is possible.

We have three types of namespaces

1. System.Collection

* ArrayList
* Queue
* Stack
* HashTable

1. System.Collection.Generic

* List
* LinkedList
* SortedList
* Queue
* Stack
* HashSet
* SortedSet
* Dictionary
* SortedDictionary

1. System.Collection.Concurrent

Concurrent collections in C# are designed to be thread-safe, allowing multiple threads to access and modify the collection concurrently without the need for external synchronization. These collections are part of the System.Collections.Concurrent namespace.

* BlockingCollection
* ConcurrentBag
* ConcurrentStack
* ConcurrentQueue
* ConcurrentDictionary

ArrayList

How the autoresizing is performed in arraylist

Initially when an arraylist is created its capacity is 0, by default. But when we add an element, then the capacity becomes to 4, after the four elements are added to the list and when we try to add the 5th element then as there is no room, again new room will be created for it with extra size and then the capacity be doubled.

Initially(c=0)->1 element added->capacity becomes 4->After capacity is filled and 5th element is added-> then becomes 8-> after that becomes 16(capacity gets doubled everytime).

We can also give our own value of size of the list,

List<T> n=new List<T>(10);

Now, it will consider 10 itself, and doubles the capacity by starting with 10.

If we wont give value, the compiler will start with 0.

Methods:

1. Add ( object value)

ArrayList list = new ArrayList();

list.Add(1);

list.Add("Hello");

1. AddRange(ICollection c)

Adds range of elements at the end

ArrayList list = new ArrayList();

list.AddRange(new int[] { 1, 2, 3 });

1. Clear()

Removes all elements from list

ArrayList list = new ArrayList();

list.Add(1);

list.Add(2);

list.Clear(); // Clears all elements

1. Contains(Object value)

Checks if a value is present in list and return true or false

ArrayList list = new ArrayList();

list.Add(1);

bool containsOne = list.Contains(1); // true

1. IndexOf(object value) and LastIndexOf(object value):

Returns the index of the first (or last) occurrence of a specific object in the ArrayList.

ArrayList list = new ArrayList() { 1, 2, 3, 1, 4 };

int firstIndex = list.IndexOf(1); // 0

int lastIndex = list.LastIndexOf(1); // 3

1. Remove(object obj) and RemoveAt(int index):

Removes the first occurrence of a specific object (or an element at a specific index) from the ArrayList.

ArrayList list = new ArrayList() { 1, 2, 3 };

list.Remove(2); // Removes the element 2

list.RemoveAt(0); // Removes the element at index 0 (1)

1. RemoveRange(int index, int count):

Removes a range of elements from the ArrayList.

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

list.RemoveRange(1, 3); // Removes elements from index 1 to 3

1. ToArray()

Copies the elements to a new array

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

int[] array = numbers.ToArray();

Properties:

1. Count

Gets the total number of elements in collection

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

int count = numbers.Count; // 3

1. Capacity

Gets or sets the total capacity of the collection without resizing.

List<int> numbers = new List<int>();

numbers.Capacity = 10;

List

List is generic, while ArrayList is non-generic.

Allows duplicate elements and accepts null as valid value for reference types.

If the size of the list equals the capacity of the list, then the capacity of list is increased automatically by relocating the internal array. The existing elements will be copied to the new array before addition of new element into list.

Elements in list are not sorted.

Accessed by zero based index.

Linked list

Represents doubly linked list

1. AddFirst(T value):

Adds a new node with the specified value at the beginning of the LinkedList<T>.

LinkedList<int> numbers = new LinkedList<int>();

numbers.AddFirst(1);

1. AddLast(T value)

Adds a new node with specified value at the end of the linked list

1. AddBefore(LinkedListNode<T> node, T value)

Adds the specified value before the specified node.

1. AddAfter(LinkedListNode<T> node, T value)

Adds the specified value after the specified node.

1. Remove(T value):

Removes the first occurrence of the specified value from linked list

1. RemoveFirst():

Removes the first node of the linked list

1. RemoveLast()

Removes the last node of the linked list.

1. Find(T value):

Finds the first node that contains the specified value.

1. Contains(T value):

Determines whether the linked list contains the specified value.

Returns true or false

Properties:

1. First:

Returns the first node of the linked list

1. Last:

Returns the last node of the linked list.

1. Count:

Identifies the count of the linked list.

Stack<t>

LIFO principle

Methods:

1. Push( T item)

Pushes the specified element at the end(top of stack)

1. Pop()

Pops the element at the top of stack.

1. Peek()

Returns the element at the top of stack.

1. Clear()

Clears the stack

1. Contains(t item)

Determines a specified element is present in the stack or not, Returns true or false.

Properties:

1. Count

Stack.Count

Gets the number of elements in the stack.

Queue<t>

FIFO principle

Methods:

1. Enqueue(T item)

Adds the element at the end of the queue

1. Dequeue()

Deleted the element from the start of the queue

1. Peek()

Returns the element at the beginning of the queue without deleting it.

1. Clear()

Deletes the elements from the queue.

1. Contains(T item)

Determines if the specified item is present in the queue, returns a Boolean value.

Properties:

1. Count

Gets the number of elements in the queue.

HashTable

Key, value combination.

Has the feature of auto resizing.

Key is user-defined.

A Hashtable is a collection of key/value pairs that are arranged based on the hash code of the key.

We wont get the same order of elements as inserted because internally it uses hashcode and has a hashing algo for storing values. Because it fetches data with hashcode so it is faster to retrieve based on hashcode.

Every class has by default four methods in it:

GetHashCode, equals, getType, ToString

// C# program to illustrate a hashtable

using System;

using System.Collections;

class Hahtable\_Example{

// Main method

static public void Main()

{

// Create a hashtable

// Using Hashtable class

Hashtable my\_hashtable = new Hashtable();

// Adding key/value pair in the hashtable

// Using Add() method

my\_hashtable.Add("A1", "Welcome");

my\_hashtable.Add("A2", "to");

my\_hashtable.Add("A3", "GeeksforGeeks");

foreach(DictionaryEntry element in my\_hashtable)

{

Console.WriteLine("Key:- {0} and Value:- {1} ",

element.Key, element.Value);

}}}

Dictionary<Tkey,TValue>

// C# program to illustrate Dictionary

using System;

using System.Collections.Generic;

class Dictionary\_example{

// Main Method

static public void Main()

{

// Creating a dictionary

// using Dictionary<TKey, TValue> class

Dictionary<string, string> My\_dict =

new Dictionary<string, string>();

// Adding key/value pairs in the Dictionary

// Using Add() method

My\_dict.Add("a.01", "C");

My\_dict.Add("a.02", "C++");

My\_dict.Add("a.03", "C#");

foreach(KeyValuePair<string, string> element in My\_dict)

{

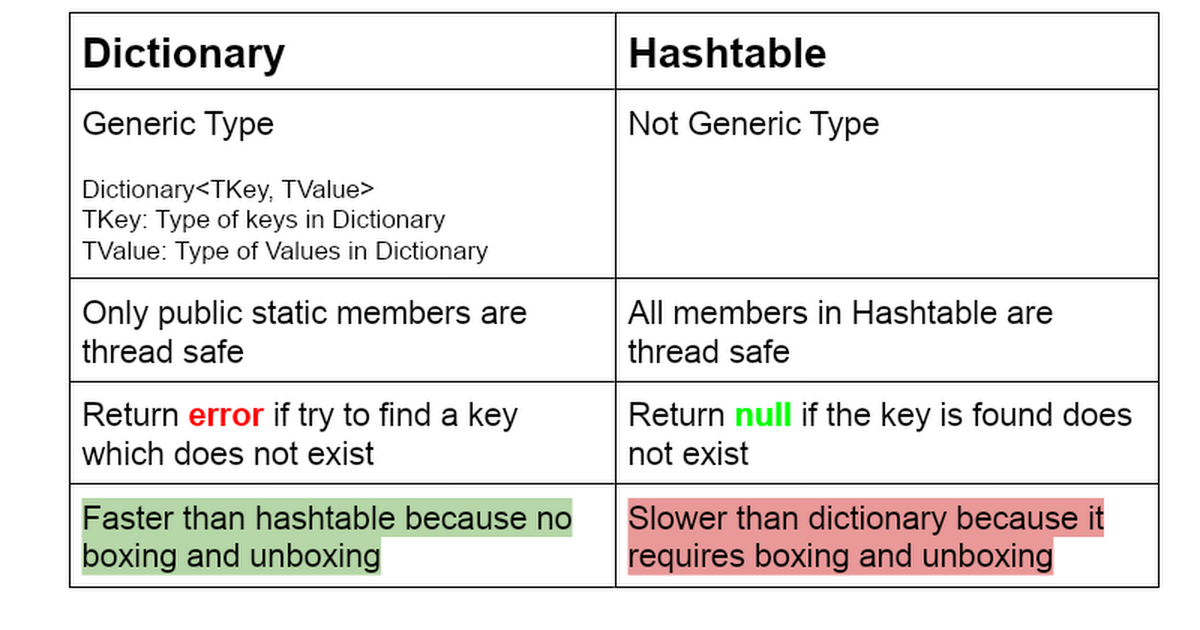
Console.WriteLine("Key:- {0} and Value:- {1}",

element.Key, element.Value);

}

}

}



**Sorted List**

Collection of key/value pairs which are sorted based on Keys.

By default, it sorts based on the ascending order.

A SortedList object internally maintains two arrays to store the elements of the list, i.e, one array for the keys and another array for the associated values.

a key cannot be null, but a value can be.

capacity of a SortedList object is the number of key/value pairs it can hold.

Cant store duplicate values.

you can store values of the same type and of the different types due to the non-generic collection. If you use a generic SortedList in your program, then it is necessary that the type of the values should be the same.

Keys are of same type.

DictionaryEntry, which is used for non-generic SortedList.

When using a generic SortedList, you should iterate over it using KeyValuePair<TKey, TValue>

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

keyValuePairs.Add(1, "A");

keyValuePairs.Add(5, "E");

keyValuePairs.Add(3, "C");

keyValuePairs.Add(26, "Z");

keyValuePairs.Add(24, "X");

Console.WriteLine(keyValuePairs[1]);

keyValuePairs.Remove(1);

keyValuePairs.RemoveAt(0); // after sorting it will remove the entry at the particular index

foreach (KeyValuePair<int,string> entry in keyValuePairs)

{

Console.WriteLine($"Key:{entry.Key}, Value:{entry.Value}");

}

**Sorted dictionary**

SortedDictionary maintains the keys in sorted order, and it uses a binary search tree internally to achieve efficient lookups. It provides a balance between the benefits of a sorted order and efficient key-based operations.

It is generic only.

SortedDictionary<int, string> mySortedDictionary = new SortedDictionary<int, string>();

mySortedDictionary.Add(3, "Three");

mySortedDictionary.Add(1, "One");

mySortedDictionary.Add(5, "Five");

Console.WriteLine("Value of key 3: " + mySortedDictionary[3]);

foreach (KeyValuePair<int, string> entry in mySortedDictionary)

{

Console.WriteLine($"Key: {entry.Key}, Value: {entry.Value}");

}

Difference between SortedList and SortedDictionary

|  |  |
| --- | --- |
| Sorted List | Sorted Dictionary |
| Internally uses array | Internally uses binary search tree |
| As elements are stored in contiguous order, it is memory efficient. | Insertions and deletions are easily performed as stored in binary search tree |
| Lower memory overhead | Higher memory overhead |
| suitable when memory efficiency is a concern, and the collection is relatively static (few insertions and deletions). | Preferred when you need efficient lookups and plan to perform frequent insertions and deletions. |

**Hash Set**

Unordered collection of unique elements.

Generic type collection.

Used when we want to prevent duplicate entries.

Performance is better than list.

Order of elements is not defined.

Cant sort the elements.

It provides many mathematical set operations, such as intersection, union, and difference.

A HashSet is a dynamic collection means the size of the HashSet is automatically increased when the new elements are added.

In HashSet, you can only store the same type of elements.

HashSet<int> myHashSet = new HashSet<int>();

myHashSet.Add(1);

myHashSet.Add(2);

myHashSet.Add(3);

myHashSet.Add(2); //duplicate element ignored

myHashSet.Remove(3);

bool containsElement = myHashSet.Contains(1);

Console.WriteLine($"HashSet contains 1: {containsElement}");

foreach (int element in myHashSet)

{

Console.WriteLine(element);

}

**Sorted set**

a collection class that represents a set of unique elements sorted in ascending order.

SortedSet provides efficient performance for operations such as adding, removing, and looking up elements, all of which have logarithmic time complexity.

SortedSet<int> mySortedSet = new SortedSet<int>();

mySortedSet.Add(5);

mySortedSet.Add(2);

mySortedSet.Add(8);

mySortedSet.Add(2); //duplicate element is ignored

mySortedSet.Remove(5);

bool containsElement = mySortedSet.Contains(2);

Console.WriteLine($"SortedSet contains 2: {containsElement}");

foreach (int element in mySortedSet)

{

Console.WriteLine(element);

}