Collections in C#

By Tonmoy Biswas

Introduction

What is collection?

- A data structure used to store and manage group of related objects.
- Dynamic storage.
- Built-in methods for sorting, searching, filtering.

Example:

Non-Generic: ArrayList, Hashtable.

Generic: List<T>, Dictionary<TKey, TValue>, HashSet<T> etc.

Why Use Collections over Array?

- Fixed sized
- No dynamic operations like add or remove.
- Not suitable for real-time applications.

```
// can't add extra elements
double[] scores = new double[3] { 87, 75, 91 };
```

ArrayList

- Non-Generic
- Store elements as object
- Type-unsafe
- Box/Unboxing for value types.
- Explicit casting for reference types.
- Slower.

```
ArrayList arr = new ArrayList() { 1, "two", 3.0, true };
int x = (int)arr[0]; // Unboxing
```

Methods & Properties:

- Add(object obj) ; TC: O(1)
- Insert(int idx, object obj) ; TC: O(n)
- Remove(object obj) ; TC: O(n)
- RemoveAt(int idx) ; TC: O(n)
- Clear() ; TC: O(n)
- Count ; TC: O(1)
- IndexOf(object obj) ; TC: O(n)
- LastIndexOf(object obj) ; TC: O(n)

List<T>

- Represents a strongly typed list of objects.
- Resizable array.
- Internally use array for storing elements.
- Indexed based accessing.
- Methods for searching, sorting, adding, removing etc.

Applications:

- Used to store and manipulate query results.
- Helps deserialize and manipulate JSON responses from APIs.

```
List<int> numbers = new List<int>();
numbers.Add(10);
numbers.Add(20);
```

List<T> Initialization Using Collection Initializer

Used to initialize the elements at the time of list object creation.

```
List<int> numbers = new List<int>() { 10, 20, 30, 40 };

Console.WriteLine(string.Join(", ", numbers));
Console.WriteLine($"Number of elements in numbers: {numbers.Count}");
```

Output:

```
10, 20, 30, 40
Number of elements in numbers: 4
```

List<T> Properties

Count: Returns the number of element present in the list. (Read-only)
Capacity: The total number of elements the list can hold without resizing. (Read-only)

```
List<int> arr = new List<int>() { 10, 20 };
Console.WriteLine($"Count : {arr.Count}");
Console.WriteLine($"Capacity : {arr.Capacity}");
Console.WriteLine("-------------------------------");
arr.Add(30); arr.Add(40);
Console.WriteLine($"Count : {arr.Count}");
Console.WriteLine($"Capacity : {arr.Capacity}");
Console.WriteLine("-------------------------------");
arr.Add(50);
Console.WriteLine($"Count : {arr.Count}");
Console.WriteLine($"Capacity : {arr.Capacity}");
```

```
Output:
Count : 2
Capacity : 4
-------------------------------
Count : 4
Capacity : 4
-------------------------------
Count : 5
Capacity : 8
```

List<T> Methods

Add (T ele) :

- Add element at the end of list.
- TC: O(1)

```
arr.Add(10);
arr.Add(20);
arr.Add(30); // arr=[10, 20, 30]
```

Remove (T ele) :

- Remove specified element.
- Internally use Equals(T ele)
- TC: O(n)

```
List<int> arr = new List<int>() { 10, 20, 30 };
arr.Remove(20);
Console.WriteLine(string.Join(", ", arr));// 10, 30
```

Insert (int idx,T ele) :

- Add element at specified index in list.
- TC: O(n)

```
arr.Add(30); // arr=[10, 20, 30]
arr.Insert(1, 40); //arr=[10, 40, 20, 30]
```

RemoveAt (int idx) :

- Remove element at specified index.
- TC: O(n)

```
List<int> arr = new List<int>() { 10, 20, 30 };
arr.RemoveAt(0);
Console.WriteLine(string.Join(", ", arr));// 20, 30
```

List<T> Methods

Contains(T ele) :

- Returns bool indicating present or not.
- Internally use Equals(T ele)
- TC: O(n)

```
List<int> arr = new List<int>() { 10, 20, 30 };
Console.WriteLine(arr.Contains(10)); //True
Console.WriteLine(arr.Contains(50)); //False
```

First (Func<bool,T>) :

- Returns first element that matches condition, else throw exception.
- TC: O(n)

```
List<int> arr = new List<int>() { 10, 20, 30 };
arr.First(x=>x==20); //return 20
arr.First(x=>x==50); //throw exception
```

FirstOrDefault (Func<bool,T>) :

- Returns first element that matches condition, else return default.
- TC: O(n)

```
List<int> arr = new List<int>() { 10, 20, 30 };
arr.FirstOrDefault(x=>x==20); //return 20
arr.FirstOrDefault(x=>x==50); //return 0
```

Iterating over List<T>

Using For Loop & Index:

```
List<int> arr = new List<int>() { 1, 2, 3, 4, 5 };
for(int i = 0; i < arr.Count; i++)
{
    Console.Write($"{arr[i]} ");
}
```

```
Output:
1 2 3 4 5
```

Using Foreach Loop:

```
List<int> arr = new List<int>() { 1, 2, 3, 4, 5 };
foreach (int n in arr)
{
    Console.Write($"{n} ");
}
```

```
Output:
1 2 3 4 5
```

List<T>.Sort()

- Sort the existing list.
- Use Introsort algorithm (combination of Quicksort, Heapsort & Insertion Sort).
- TC: O(nlog n)

```
List<int> arr = new List<int>() { 3, 5, 2, 7, 1};
arr.Sort();// ascending order
Console.WriteLine(string.Join(", ", arr));
```

```
Output:
1, 2, 3, 5, 7
```

List<T>.Sort() with Custom class

```
internal class Student
{
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public string Name { get; set; }
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public int Roll { get; set; }
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public double Score { get; set; }
}
```

```
List<Student> students = new List<Student>
{
    new Student { Name = "Rahul", Roll = 1, Score = 85.5 },
    new Student { Name = "Vikram", Roll = 3, Score = 78.4 },
    new Student { Name = "Ananya", Roll = 2, Score = 91.2 },
    new Student { Name = "Sourav", Roll = 5, Score = 82.0 },
    new Student { Name = "Megha", Roll = 4, Score = 88.9 }
};
```

```
students.Sort();
```

```
Unhandled Exception: System.InvalidOperationException: Failed to compare two
 elements in the array. ---> System.ArgumentException: At least one object m
ust implement IComparable.
```

List<T>.Sort() with Custom class

- Sort() method internally call CompareTo(T other) method.

```
internal class Student: IComparable<Student>
{
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public string Name { get; set; }
    13 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public int Roll { get; set; }
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public double Score { get; set; }
    0 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public int CompareTo(Student other)
    {
        //return Score.CompareTo(other.Score);
        return Roll.CompareTo(other.Roll);
        //return Name.CompareTo(other.Name);
    }
}
```

```
students.Sort();
foreach (var student in students)
{
    Console.WriteLine(student);
}
```

```
Name: Rahul, Roll: 1, Score: 85.5
Name: Ananya, Roll: 2, Score: 91.2
Name: Vikram, Roll: 3, Score: 78.4
Name: Megha, Roll: 4, Score: 88.9
Name: Sourav, Roll: 5, Score: 82
```

List<T>.Sort(Comparison) with Custom class

- Sort(Comparison<T> comparison) method internally call the specified comparison rule.
- Comparison rules dominant.
- Not mandatory to implement IComparable<T>.

```
internal class Student: IComparable<Student>
{
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public string Name { get; set; }
    13 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public int Roll { get; set; }
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public double Score { get; set; }
    0 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public int CompareTo(Student other)
    {
        //return Score.CompareTo(other.Score);
        return Roll.CompareTo(other.Roll);
        //return Name.CompareTo(other.Name);
    }
}
```

```
students.Sort((a,b)=> b.Score.CompareTo(a.Score) );
foreach (var student in students)
{
    Console.WriteLine(student);
}
```

```
Name: Ananya, Roll: 2, Score: 91.2
Name: Megha, Roll: 4, Score: 88.9
Name: Rahul, Roll: 1, Score: 85.5
Name: Sourav, Roll: 5, Score: 82
Name: Vikram, Roll: 3, Score: 78.4
```

List<T>.Contains(T obj) with Custom class

```
internal class Student
{
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public string Name { get; set; }
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public int Roll { get; set; }
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public double Score { get; set; }
}
```

```
List<Student> students = new List<Student>
{
    new Student { Name = "Rahul", Roll = 1, Score = 85.5 },
    new Student { Name = "Vikram", Roll = 3, Score = 78.4 },
    new Student { Name = "Ananya", Roll = 2, Score = 91.2 },
    new Student { Name = "Sourav", Roll = 5, Score = 82.0 },
    new Student { Name = "Megha", Roll = 4, Score = 88.9 }
};
```

```
Console.WriteLine(students.Contains(new Student { Name = "Vikram", Roll = 3, Score = 78.4 }));
```

```
Output:
False
```

```
Console.WriteLine(students.Contains(students[1]));
```

```
Output:
True
```

List<T>.Contains(T obj) with Custom class

- Contains(T obj) internally called Equals(object other) on each element.

```
internal class Student
{
    12 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public string Name { get; set; }
    14 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public int Roll { get; set; }
    12 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public double Score { get; set; }
    0 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public override bool Equals(object obj)
    {
        return Roll.Equals((obj as Student).Roll);
    }
    0 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public override string ToString()
    {
        return $"Name: {Name}, Roll: {Roll}, Score: {Score}";
    }
}
```

```
List<Student> students = new List<Student>
{
    new Student { Name = "Rahul", Roll = 1, Score = 85.5 },
    new Student { Name = "Vikram", Roll = 3, Score = 78.4 },
    new Student { Name = "Ananya", Roll = 2, Score = 91.2 },
    new Student { Name = "Sourav", Roll = 5, Score = 82.0 },
    new Student { Name = "Megha", Roll = 4, Score = 88.9 }
};
```

```
Console.WriteLine(students.Contains(new Student { Name = "Vikram", Roll = 3, Score = 78.4 }));
```

```
Output:
True
```

```
Console.WriteLine(students.Contains(students[1]));
```

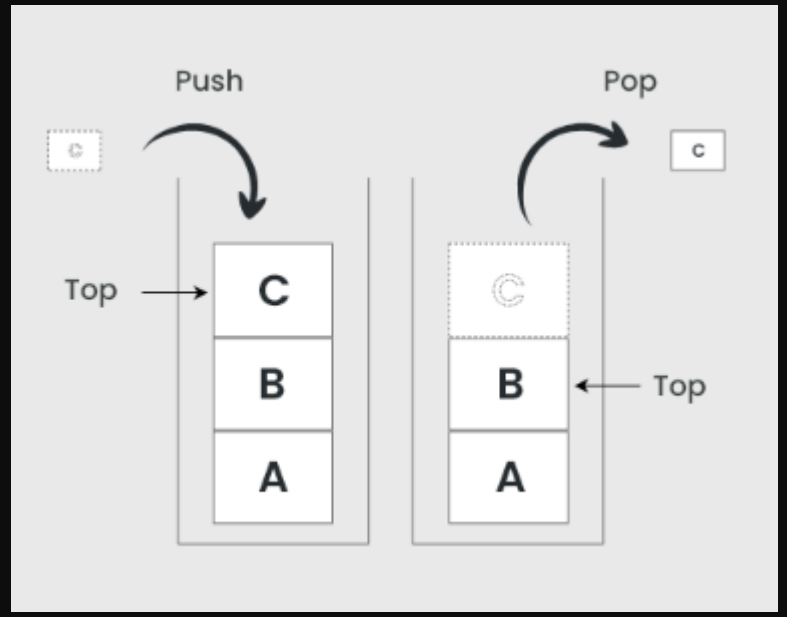
```
Output:
True
```

Array vs ArrayList vs List<T>

Feature	Array	ArrayList	List<T>
Type Safety	(strongly typed)	(object type)	(generic, strongly typed)
Performance	Fast	Slower (boxing/unboxing)	Fast (type-safe)
Flexibility	Fixed size	Dynamic	Dynamic
Usage Example	`int[] arr = new int[3];`	`ArrayList list = new ArrayList();`	`List<int> list = new List<int>();`

Stack<T>

- Last-In-First-Out collection.
- Used in text editors for undo operation.



Stack<T> Methods

Push(T ele) :
- Add element at the top of stack.
- TC: O(1)

Pop() :
- Remove top element and returns it.
- TC: O(1)

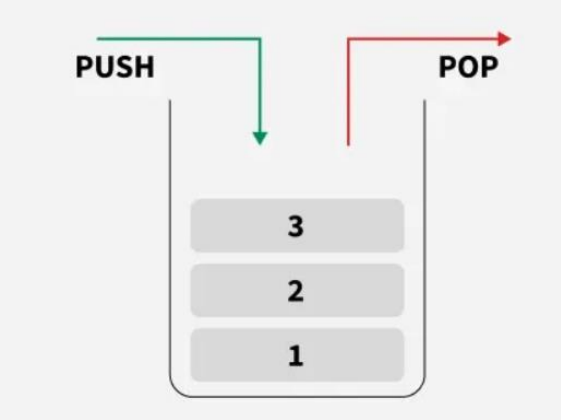
Peek() :
- Return top element.
- TC: O(1)

```
// Create a new stack
Stack<int> s = new Stack<int>();
// Push elements onto the stack
s.Push(1);
s.Push(2);
s.Push(3);
s.Push(4);
Console.WriteLine($"Top element: {s.Peek()}");
// Pop elements from the stack
while (s.Count > 0)
{
    Console.WriteLine(s.Pop());
}
```

Output:

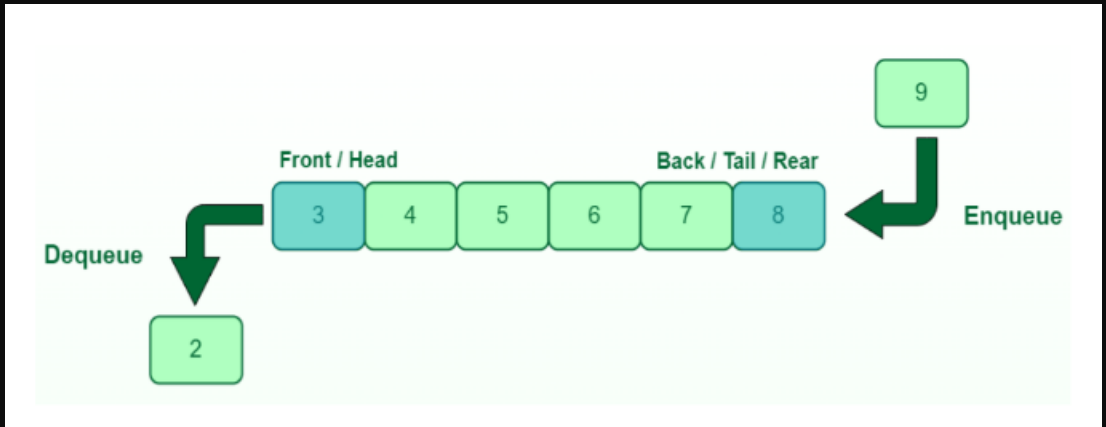
```
Top element: 4
4
3
2
1
```

Memory Organization:



Queue<T>

- First-In-First-Out collection.
- Used in printer queues, background jobs, task pipelines.



Queue<T> Methods

Enqueue(T ele) :

- Add element at the back of the queue.
- TC: O(1)

Dequeue() :

- Remove front element and returns it.
- TC: O(1)

Peek() :

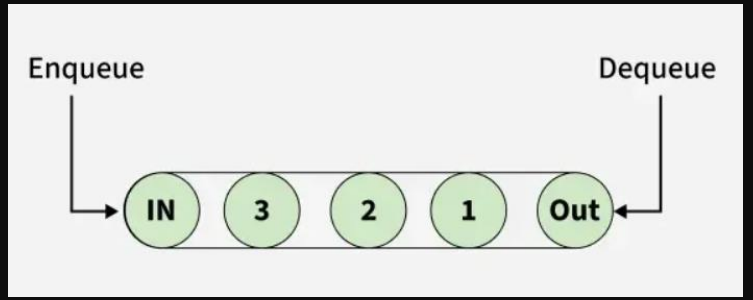
- Return front element.
- TC: O(1)

```
// Create a new queue
Queue<int> q = new Queue<int>();
// Enqueue elements into the queue
q.Enqueue(1);
q.Enqueue(2);
q.Enqueue(3);
Console.WriteLine($"Front Element: {q.Peek()}");
// Dequeue elements from the queue
while (q.Count > 0)
{
    Console.WriteLine(q.Dequeue());
}
```

Output:

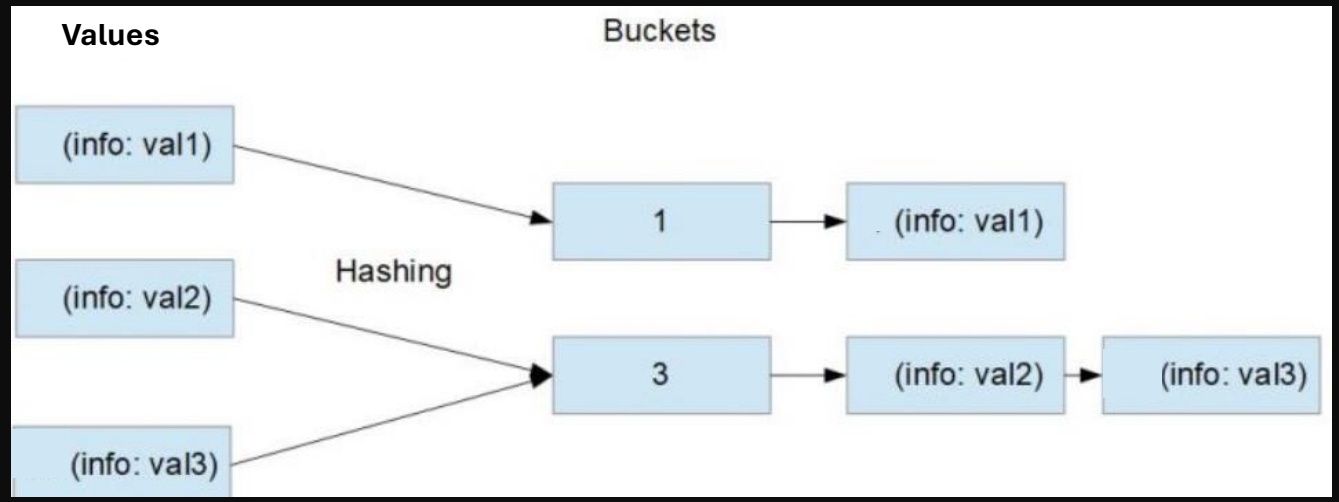
```
Front Element: 1
1
2
3
```

Memory Organization:



HashSet<T>

- Collection of unique elements.
- Hashing for fast lookups.
- Insertion order not preserved.
- Duplicate values not allowed.



HashSet<T> Methods

Add(T ele) :

- Add element into set.
- Return ture, if item added.
- Return false, if it already exist.
- TC: O(1)

```
HashSet<int> s = new HashSet<int>();
Console.WriteLine(s.Add(1));
Console.WriteLine(s.Add(2));
Console.WriteLine(s.Add(3));
Console.WriteLine(s.Add(2));
Console.WriteLine(string.Join(", ", s));
```

```
Output:
True
True
True
False
1, 2, 3
```

HashSet<T> Methods

Remove() :

- Remove element from set.
- Return ture, if item found and removed.
- Return false otherwise.
- TC: O(1)

```
HashSet<int> s = new HashSet<int>() { 1, 2, 3 };
Console.WriteLine(s.Remove(2));
Console.WriteLine(s.Remove(4));
Console.WriteLine(string.Join(", ", s));
```

```
Output:
True
False
1, 3
```

HashSet<T> Methods

Contains(T ele) :

- Return ture, if the item present.
- Return false otherwise.
- TC: O(1)

```
HashSet<int> s = new HashSet<int>() { 1, 2, 3 };
Console.WriteLine(s.Contains(2));
Console.WriteLine(s.Contains(4));
Console.WriteLine(string.Join(", ", s));
```

```
Output:
True
False
1, 2, 3
```

HashSet<T> - Set Operation

UnionWith() :

- Combines elements from another set.

```
var set1 = new HashSet<int> { 1, 2 };
var set2 = new HashSet<int> { 2, 3 };
set1.UnionWith(set2); // set1 = {1, 2, 3}
```

IntersectWith() :

- Keeps only common elements.

```
set1 = new HashSet<int> { 1, 2 };
set2 = new HashSet<int> { 2, 3 };
set1.IntersectWith(set2); // set1 = {2}
```

HashSet<T> - Set Operation

ExceptWith() :

- Removes elements that exist in another set.

```
set1 = new HashSet<int> { 1, 2, 3 };
set2 = new HashSet<int> { 2 };
set1.ExceptWith(set2); // set1 = {1, 3}
```

HashSet<T> with Custom class

```
internal class Student
{
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public string Name { get; set; }
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public int Roll { get; set; }
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public double Score { get; set; }
}
```

```
HashSet<Student> students = new HashSet<Student>()
{
    new Student { Name = "Rahul", Roll = 1, Score = 85.5 },
    new Student { Name = "Vikram", Roll = 3, Score = 78.4 },
    new Student { Name = "Ananya", Roll = 2, Score = 91.2 },
    new Student { Name = "Sourav", Roll = 5, Score = 82.0 },
    new Student { Name = "Megha", Roll = 4, Score = 88.9 }
};
```

```
students.Add(new Student { Name = "Vikram", Roll = 3, Score = 78.4 });
Console.WriteLine(string.Join("\n", students));
Console.WriteLine(students.Contains(new Student { Name = "Vikram", Roll = 3, Score = 78.4 }));
```

```
Output:
Name: Rahul, Roll: 1, Score: 85.5
Name: Vikram, Roll: 3, Score: 78.4
Name: Ananya, Roll: 2, Score: 91.2
Name: Sourav, Roll: 5, Score: 82
Name: Megha, Roll: 4, Score: 88.9
Name: Vikram, Roll: 3, Score: 78.4
False
```

Internally use GetHashCode() & Equals() methods.

HashSet<T> with Custom class

```
internal class Student
{
    13 references | Tonmoy Biswas, 2 days ago | 1 author, 1 change
    public string Name { get; set; }
    16 references | Tonmoy Biswas, 2 days ago | 1 author, 1 change
    public int Roll { get; set; }
    13 references | Tonmoy Biswas, 2 days ago | 1 author, 1 change
    public double Score { get; set; }
    0 references | Tonmoy Biswas, 2 days ago | 1 author, 1 change
    public override bool Equals(object obj)
    {
        return Roll.Equals((obj as Student).Roll);
    }
    0 references | Tonmoy Biswas, 2 days ago | 1 author, 1 change
    public override int GetHashCode()
    {
        return Roll.GetHashCode();
    }
```

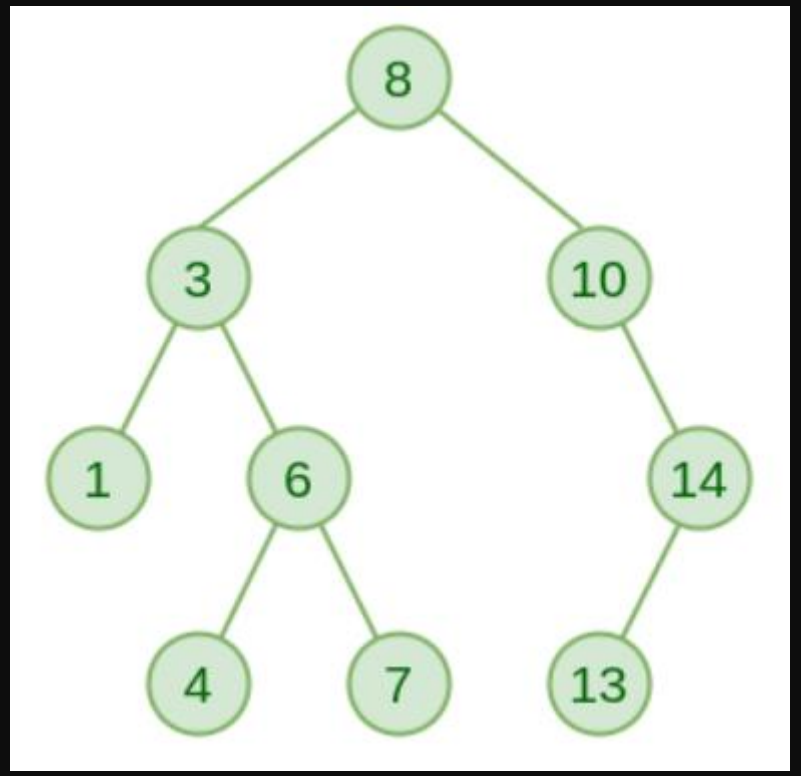
```
HashSet<Student> students = new HashSet<Student>()
{
    new Student { Name = "Rahul", Roll = 1, Score = 85.5 },
    new Student { Name = "Vikram", Roll = 3, Score = 78.4 },
    new Student { Name = "Ananya", Roll = 2, Score = 91.2 },
    new Student { Name = "Sourav", Roll = 5, Score = 82.0 },
    new Student { Name = "Megha", Roll = 4, Score = 88.9 }
};
```

```
students.Add(new Student { Name = "Vikram", Roll = 3, Score = 78.4 });
Console.WriteLine(string.Join("\n", students));
Console.WriteLine(students.Contains(new Student { Name = "Vikram", Roll = 3, Score = 78.4 }));
```

```
Output:
Name: Rahul, Roll: 1, Score: 85.5
Name: Vikram, Roll: 3, Score: 78.4
Name: Ananya, Roll: 2, Score: 91.2
Name: Sourav, Roll: 5, Score: 82
Name: Megha, Roll: 4, Score: 88.9
True
```

SortedSet<T>

- Collection of unique elements.
- Elements are stored in ascending ordered.
- Duplicate values not allowed.
- Self-balancing BST used (Specially Red-Black Tree).



SortedSet<T> Methods

- All methods presents in HastSet<T>.
- Add(); TC: O(log n)
- Remove(); TC: O(log n)
- Contains(); TC: O(log n)

SortedSet<T> with Custom class

```
internal class Student
{
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public string Name { get; set; }
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public int Roll { get; set; }
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public double Score { get; set; }
}
```

```
SortedSet<Student> students = new SortedSet<Student>()
{
    new Student { Name = "Rahul", Roll = 1, Score = 85.5 },
    new Student { Name = "Vikram", Roll = 3, Score = 78.4 },
    new Student { Name = "Ananya", Roll = 2, Score = 91.2 },
    new Student { Name = "Sourav", Roll = 5, Score = 82.0 },
    new Student { Name = "Megha", Roll = 4, Score = 88.9 }
};
```

Output:

```
Output:

Unhandled Exception: System.ArgumentException: At least one object must implement IComparable.
```

Internally use CompareTo() method.

SortedSet<T> with Custom class

```
internal class Student: IComparable<Student>
{
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public string Name { get; set; }
    13 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public int Roll { get; set; }
    11 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public double Score { get; set; }
    0 references | Tonmoy Biswas, 1 day ago | 1 author, 1 change
    public int CompareTo(Student other)
    {
        //return Score.CompareTo(other.Score);
        return Roll.CompareTo(other.Roll);
        //return Name.CompareTo(other.Name);
    }
}
```

```
SortedSet<Student> students = new SortedSet<Student>()
{
    new Student { Name = "Rahul", Roll = 1, Score = 85.5 },
    new Student { Name = "Vikram", Roll = 3, Score = 78.4 },
    new Student { Name = "Ananya", Roll = 2, Score = 91.2 },
    new Student { Name = "Sourav", Roll = 5, Score = 82.0 },
    new Student { Name = "Megha", Roll = 4, Score = 88.9 }
};
```

```
students.Add(new Student { Name = "Vikram", Roll = 3, Score = 78.4 });
Console.WriteLine(string.Join("\n", students));
Console.WriteLine(students.Contains(new Student { Name = "Vikram", Roll = 3, Score = 78.4 }));
```

```
Output:
Name: Rahul, Roll: 1, Score: 85.5
Name: Ananya, Roll: 2, Score: 91.2
Name: Vikram, Roll: 3, Score: 78.4
Name: Megha, Roll: 4, Score: 88.9
Name: Sourav, Roll: 5, Score: 82
True
```

SortedSet<T> with Custom class

- Use implementation of IComparer<T>.
- Implementation of IComparer<T> rules dominant.
- Not mandatory to implement IComparable<T>.

```
class StudentComparer : IComparer<Student>
{
    0 references | 0 changes | 0 authors, 0 changes
    public int Compare(Student x, Student y)
    {
        return x.Roll - y.Roll;
        //return x.Roll.CompareTo(y.Roll);
    }
}
```

```
SortedSet<Student> students = new SortedSet<Student>(new StudentComparer())
{
    new Student { Name = "Rahul", Roll = 1, Score = 85.5 },
    new Student { Name = "Vikram", Roll = 3, Score = 78.4 },
    new Student { Name = "Ananya", Roll = 2, Score = 91.2 },
    new Student { Name = "Sourav", Roll = 5, Score = 82.0 },
    new Student { Name = "Megha", Roll = 4, Score = 88.9 }
};
```

```
students.Add(new Student { Name = "Vikram", Roll = 3, Score = 78.4 });
Console.WriteLine(string.Join("\n", students));
Console.WriteLine(students.Contains(new Student { Name = "Vikram", Roll = 3, Score = 78.4 }));
```

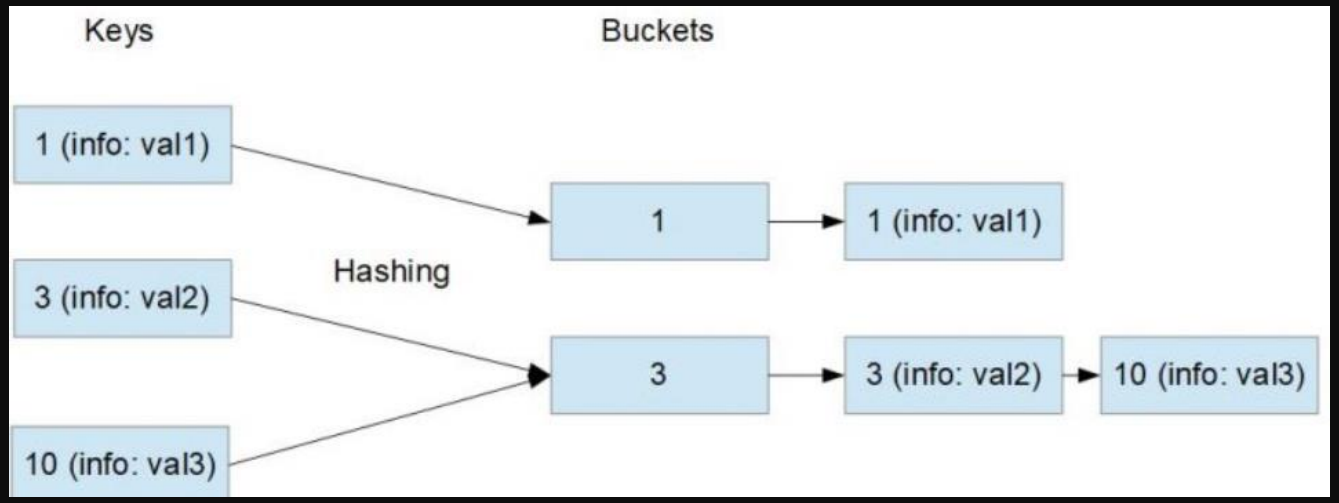
```
Output:
Name: Rahul, Roll: 1, Score: 85.5
Name: Ananya, Roll: 2, Score: 91.2
Name: Vikram, Roll: 3, Score: 78.4
Name: Megha, Roll: 4, Score: 88.9
Name: Sourav, Roll: 5, Score: 82
True
```

HashSet<T> vs SortedSet<T>

Feature	HashSet	SortedSet
Underlying Data Structure	Hash table	Balanced binary search tree (Red-Black Tree)
Ordering	Unordered	Maintains sorted order
Performance (Contains)	Average: O(1)	Average: O(log n)
Internal Comparison Method	Uses `GetHashCode()` and `Equals()` of `T`	Uses `Comparer<T>.Compare(x, y)`
Duplicates Allowed	No	No
Custom Comparison Logic	Override `Equals()` and `GetHashCode()`	Provide custom `IComparer<T>`
Use Case	Fast lookups, uniqueness checks	Sorted data, range queries, ordered traversal

Dictionary<TKey,TValue>

- Stores key-value pairs.
- Fast lookups by key.
- Duplicate keys not allowed.
- Duplicate values allowed.
- Used for key-based data caching.
- Use GetHashCode() & Equals() for key comparison.



Dictionary<TKey,TValue> Methods

Add(TKey key,TValue value) :

- Add key-value pair in Dictionary.
- Throws exception if key already exist.
- TC: O(1)

```
Dictionary<string, int> quantity = new Dictionary<string, int>();
quantity.Add("apple", 3);
quantity.Add("orange", 5);
```

Dictionary<TKey,TValue> Methods

Remove(TKey key) :

- Remove key-value pair from dictionary based on key.
- Return true, if key found and removed.
- Return false, otherwise.
- TC: O(1)

```
Dictionary<string, int> quantity = new Dictionary<string, int>()
{
    {"apple", 3 },
    {"orange", 5 }
};
Console.WriteLine(quantity.Remove("apple"));
Console.WriteLine(quantity.Remove("banana"));
foreach (KeyValuePair<string, int> p in quantity)
{
    Console.WriteLine($"Key: {p.Key}, Value: {p.Value}");
}
```

```
Output:
True
False
Key: orange, Value: 5
```

Dictionary<TKey,TValue> Methods

ContainsKey(TKey key) :

- Return true, if key present.
- Return false, otherwise.
- TC: O(1)

```
Dictionary<string, int> quantity = new Dictionary<string, int>()
{
    {"apple", 3 },
    {"orange", 5 }
};
Console.WriteLine(quantity.ContainsKey("apple"));
Console.WriteLine(quantity.ContainsKey("banana"));
foreach (KeyValuePair<string, int> p in quantity)
{
    Console.WriteLine($"Key: {p.Key}, Value: {p.Value}");
}
```

```
Output:
True
False
Key: apple, Value: 3
Key: orange, Value: 5
```

Dictionary<TKey,TValue> Methods

ContainsValue(TValue value) :

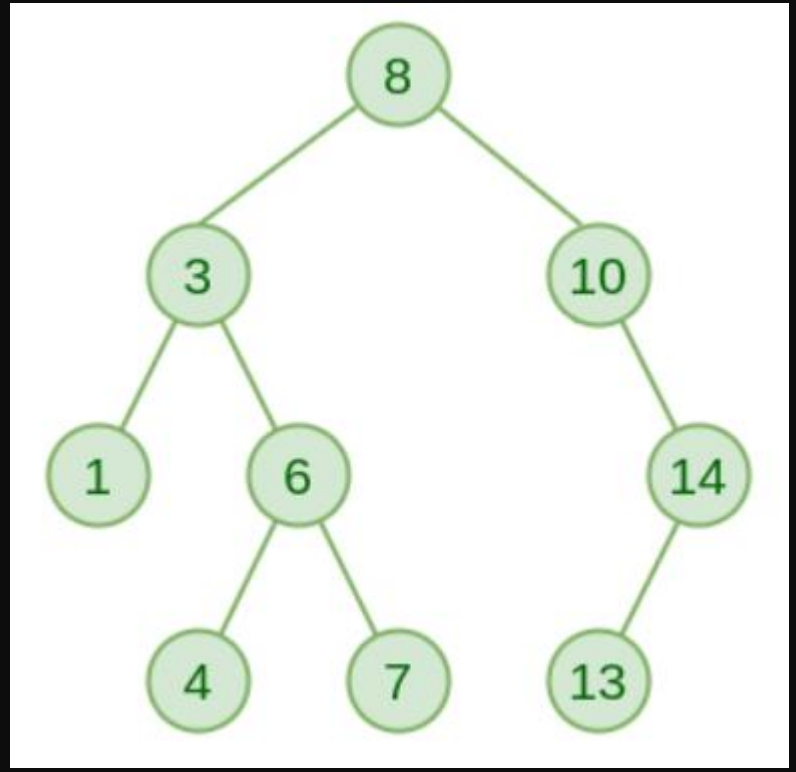
- Return true, if value present.
- Return false, otherwise.
- TC: O(n)

```
Dictionary<string, int> quantity = new Dictionary<string, int>()
{
    {"apple", 3 },
    {"orange", 5 }
};
Console.WriteLine(quantity.ContainsValue(5));
Console.WriteLine(quantity.ContainsValue(9));
foreach (KeyValuePair<string, int> p in quantity)
{
    Console.WriteLine($"Key: {p.Key}, Value: {p.Value}");
}
```

```
Output:
True
False
Key: apple, Value: 3
Key: orange, Value: 5
```

SortedDictionay<TKey,TValue>

- Stores key-value pairs.
- Underlying Datasturcture: BST (Red-Black Tree)
- Duplicate keys not allowed.
- Duplicate values allowed.
- Use CompareTo() for key comparison.



SortedDictionay<TKey,TValue> Methods

- All methods presents in Dictionary<TKey,TValue>.
- Add(); TC: O(log n)
- Remove(); TC: O(log n)
- ContainsKey(); TC: O(log n)
- ContainsValue(); TC: O(n)

SortedList<TKey,TValue>

- Stores key-value pairs.
- Ordered by key.
- Internally use two parallel array. Array of Keys & Array of Values.
- Duplicate keys not allowed.
- Duplicate values allowed.
- Use CompareTo() for key comparison.

SortedList<TKey,TValue> Methods

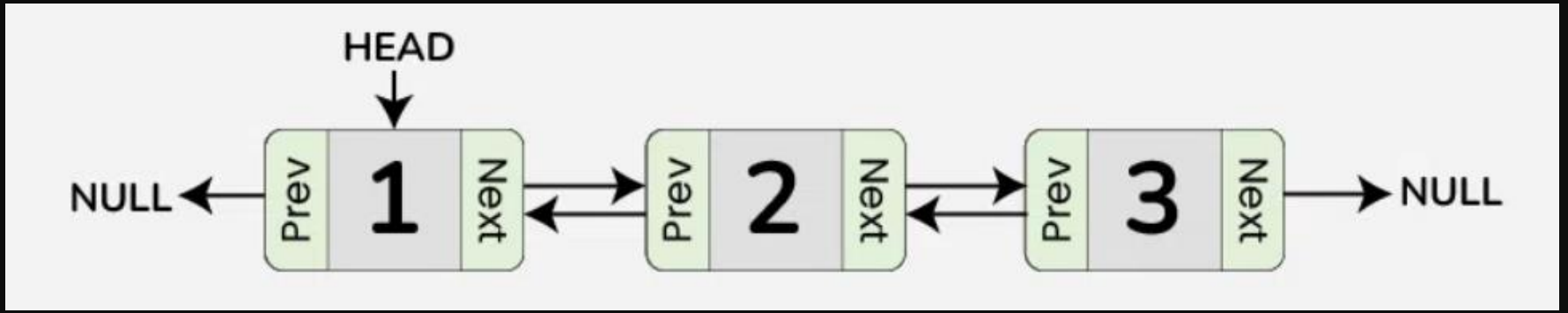
- All methods presents in Dictionary<TKey,TValue>.
- Add(); TC: O(n)
- Remove(); TC: O(n)
- ContainsKey(); TC: O(log n)
- ContainsValue(); TC: O(n)

Dictionary vs SortedDictionary vs SortedList

Feature	Dictionary	SortedDictionary	SortedList
Ordering	No	Yes	Yes
Internal Structure	Hash Table	Red-Black Tree	Array
Lookup Performance	O(1) avg	O(log n)	O(log n)
Insert Performance	O(1) avg	O(log n)	O(n)
Memory Efficiency	High	Moderate	High (small datasets)
Custom Comparer	EqualityComparer	Comparer	Comparer

LinkedList<T>

- Linear and Non-contiguous.
- Implemented as Doubly Linked List.
- Each node contains value and reference(pointer) .
- Duplicate values allowed.



LinkedList<T> Methods and Properties

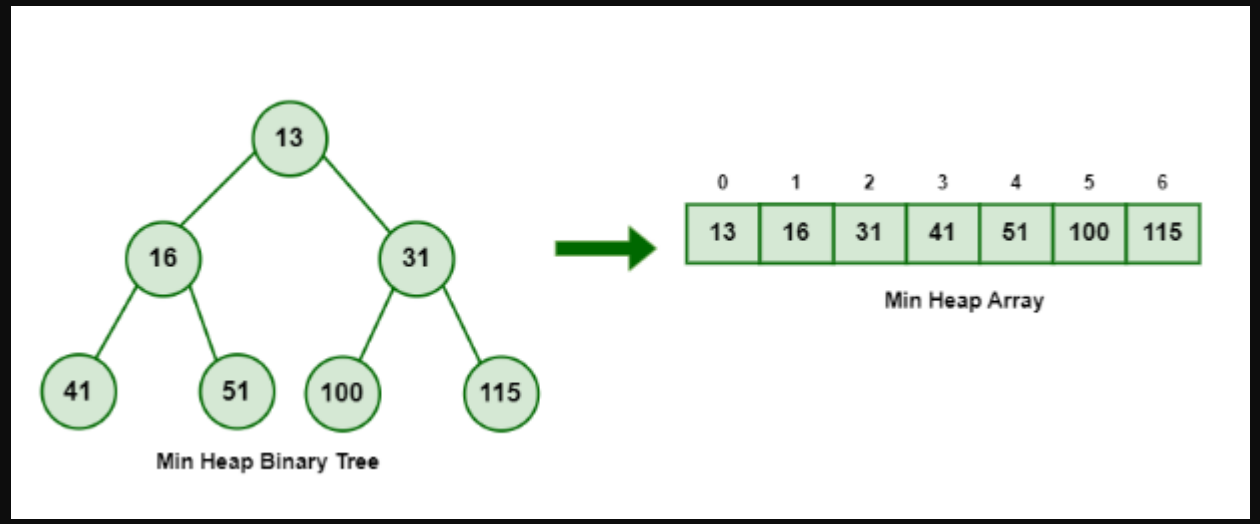
- AddFirst(), AddLast(); TC: O(1)
- RemoveFirst(), RemoveLast(); TC: O(1)
- First, Last; TC: O(1)

```
LinkedList<int> list = new LinkedList<int>();
list.AddLast(1); // 1
list.AddFirst(2); // 2, 1
list.AddLast(3); // 2, 1, 3
list.AddFirst(4); // 4, 2, 1, 3
Console.WriteLine(string.Join(", ", list));
Console.WriteLine($"First Element: {list.First.Value}");
Console.WriteLine($"Last Element: {list.Last.Value}");
list.RemoveFirst(); // 2, 1, 3
list.RemoveLast(); // 2, 1
Console.WriteLine(string.Join(", ", list));
```

```
Output:
4, 2, 1, 3
First Element: 4
Last Element: 3
2, 1
```

PriorityQueue<TValue,TPriority>

- Non-Linear.
- Implemented using Min-Heap data structure.
- Elements are removed based on priority.
- Duplicate values and priorities are allowed.
- Introduced in .NET 6.



PriorityQueue<TValue,TPriority> Methods

- Enqueue(TValue value, TPriority priority); TC: O(log n)
- TryDequeue(out TValue value, out TPriority priority); TC: O(log n)
- TryPeek(out TValue value, out TPriority priority); TC: O(1)

```
PriorityQueue<string, int> pq = new PriorityQueue<string, int>();
pq.Enqueue("Task 3", 3);
pq.Enqueue("Task 2", 2);
pq.Enqueue("Task 4", 3);
pq.Enqueue("Task 1", 1);
if (pq.TryPeek(out string task, out int priority))
{
    Console.WriteLine($"Peek Element: {task}, Priority: {priority}");
}
Console.WriteLine("--------------------");
while (pq.Count > 0)
{
    pq.TryDequeue(out  task, out  priority);
    Console.WriteLine($"Dequeued: {task} with priority {priority}");
}
```

```
Output:
Peek Element: Task 1, Priority: 1
--------------------
Dequeued: Task 1 with priority 1
Dequeued: Task 2 with priority 2
Dequeued: Task 4 with priority 3
Dequeued: Task 3 with priority 3
```

THANK YOU