.NET Collections

C# provides a rich set of collection support to manage and manipulate data efficiently.

ArrayList,
BitArray,
Hastable

Collections

Generic Collections

Concurrent Collections

Immutable Collections

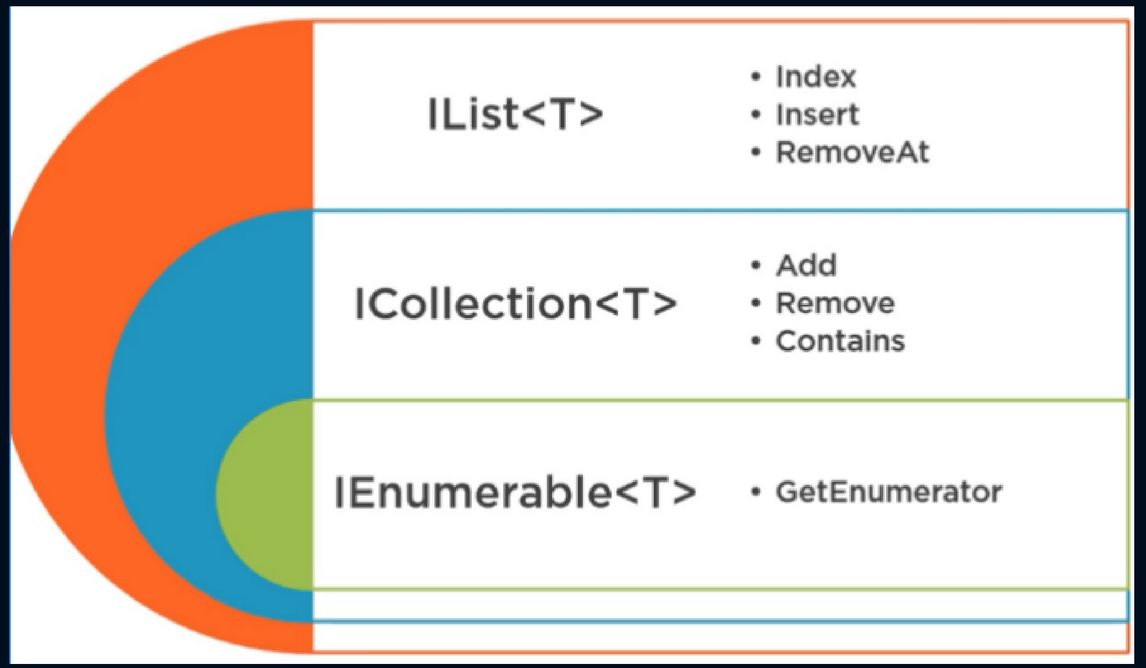
Specialized Collections

IEnumerable

IEnumerable : represents a generic sequence of elements that can be enumerated. It defines a single method, GetEnumerator(), which returns an IEnumerator<T> object for iterating over the collection.
IEnumerator: provides methods for iterating over a collection of elements. It defines properties like Current to get the current element in the collection, and methods like MoveNext() to move to the next element and Reset() to reset the enumerator.

These interfaces provide abstractions for working with collections in a generic and type-safe manner.

Generic collection interfaces implementation



IList, ICollection, IEnumerable

Dictionary

Represents a collection of key-value pairs, where each key is unique within the dictionary. It is part of the System.Collections.Generic namespace and provides efficient lookup and retrieval of values based on their corresponding keys.
Has fast lookups, key based operations, suitable for applications that require efficient data access and retrieval.

Dictionary<TKey, TValue> - A collection of key-value pairs that provides fast lookup of values based on keys.

Dictionary.cs

```
Dictionary<string, int> personAges = new Dictionary<string, int>();
personAges.Add("Ali",63);
personAges.Add("Fatima", 18);
personAges.Add("Hasan", 48);
```

Main top Dictionary<TKey, TValue> methods

Add(TKey, TValue): Adds the specified key-value pair to the dictionary.
ContainsKey(TKey): Determines whether the dictionary contains the specified key.
ContainsValue(TValue): Determines whether the dictionary contains the specified value.
Remove(TKey): Removes the element with the specified key from the dictionary.
TryGetValue(TKey, out TValue): Gets the value associated with the specified key.
Keys: Gets a collection of the keys in the dictionary.
Values: Gets a collection of the values in the dictionary.

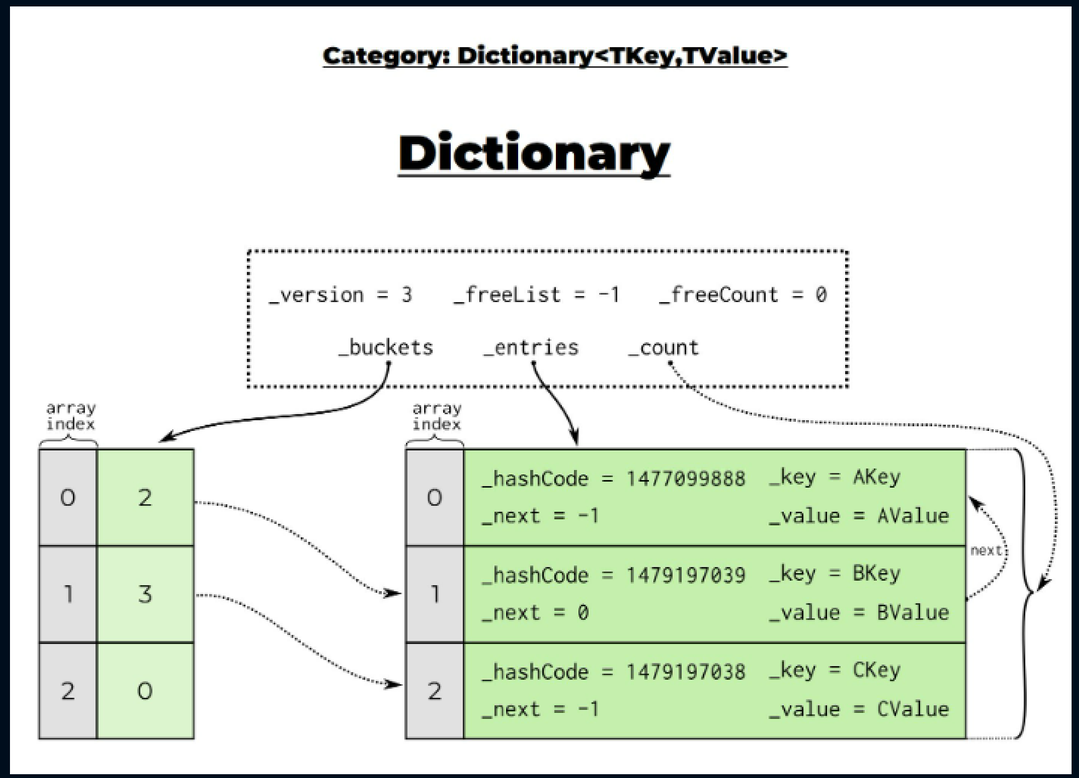
Stores key-value pairs

Fast lookup and insertion of elements

Slower element access due to hashing collisions

Tip: Use Dictionary<TKey, TValue> when you have a collection of key-value pairs and you need to perform operations like add, remove, and search frequently based on the key.

Lists



How Dictionary wotks

HashSet

Ignores values when adding, It is designed from the outset as a collection of unique values, very scalable and efficient at enforcing uniqueness. Don't have keys, don't support lookup.
Supports set operations like Union, intersection and difference

The HashSet<T> class is a collection of **unique** objects that **do not have a specific order**. It is useful when you want to ensure that the elements in the collection are unique.

HashSet.cs

```
HashSet<string> attendees = new HashSet<string>();
attendees.Add("John");
attendees.Add("Ali");
attendees.Add("John"); // Will not be added since John is already in the set
```

Main top HashSet<T> methods

Add(T item): Adds an item to the HashSet if it does not already exist.

UnionWith(IEnumerable<T> other): Modifies the current HashSet object to contain all elements that are present in itself, the specified collection, or both.

IntersectWith(IEnumerable<T> other): Modifies the current HashSet object to contain only elements that are present in both itself and the specified collection.

ExceptWith(IEnumerable<T> other): Modifies the current HashSet object to contain only elements that are present in itself and not in the specified collection.

Stores unique elements in no particular order

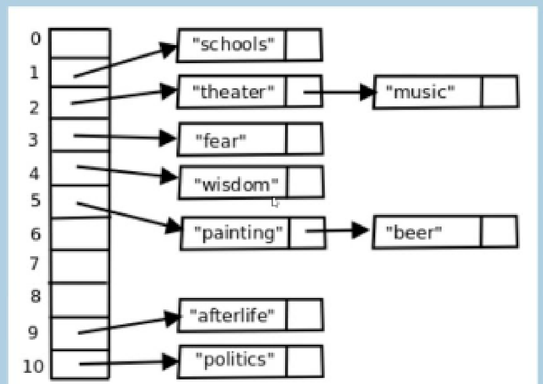
Fast lookup, insertion, and deletion operations

Not allow duplicate value

Tip: Use HashSet<T> when you have a collection of unique objects and you need to perform operations like add, remove, and search frequently.

Lists

- HashSet collection contains a group of elements of unique values stored at respective indexes.
- Full Path: System.Collections.Generic.HashSet
- Process of adding an element:
  - Generate index based on the value. Ex: index = hash code % count
  - Add the element (value) next to the linked list at the generated index.



HashSet explained

Stack

We use Stack when we want the last added element to be the first one to go. A practical implementation in the software would be an undo-redo operation in a common editor, where we can use a stack to store all the performed operations by a user and then use it to retrieve them from the most recently performed action to the oldest one.

The Stack<T> class is a collection of objects arranged in a **last-in, first-out (LIFO) order**. It is useful when you want to process elements in the **reverse** of the order in which they were added.

Stack.cs

```
Stack<string> history = new Stack<string>();
history.Push("Homepage");
history.Push("Contact Us");
history.Push("About Us");
```

Main top Stack<T> methods

Push: Adds an item to the top of the stack.

Pop: Removes and returns the item at the top of the stack.

Peek: Returns the item at the top of the stack without removing it.

CopyTo: Copies the stack elements to an array.

GetEnumerator: Returns an enumerator for iterating through the stack elements.

Managing data that follows the LIFO technique

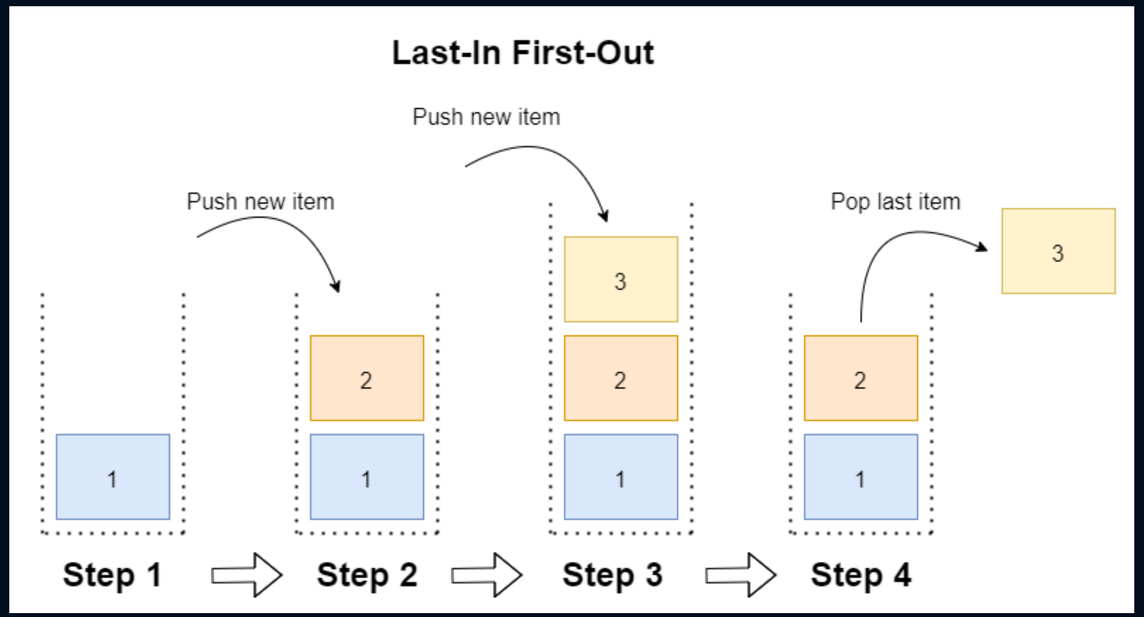
Fast insertion at the top and removal from the top

Stack memory is of limited size

Random accessing is not possible

Tip: when you have a collection of objects that needs to follow a LIFO (Last-In-First-Out) pattern, and you need to perform operations like Push and Pop frequently.

Lists



Stack collection

Queue

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It represents a collection of elements where new elements are added to the back (enqueue) and existing elements are removed from the front (dequeue)

The Queue<T> class is a collection of objects arranged in a **first-in, first-out** (**FIFO**) order. It is useful when you want to process elements in a **specific order**.

Queue.cs

```
Queue<string> orders = new Queue<string>();
orders.Enqueue("Pizza");
orders.Enqueue("QormeSabzi");
orders.Enqueue("Pasta");
```

Main top Queue<T> methods

Enqueue(T): adds an element to the end of the queue.

Dequeue(): removes and returns the element at the beginning of the queue.

Peek(): returns the element at the beginning of the queue without removing it.

TrimExcess(): reduces the capacity of the queue to match the number of elements.

FIFO (First In, First Out) data structure

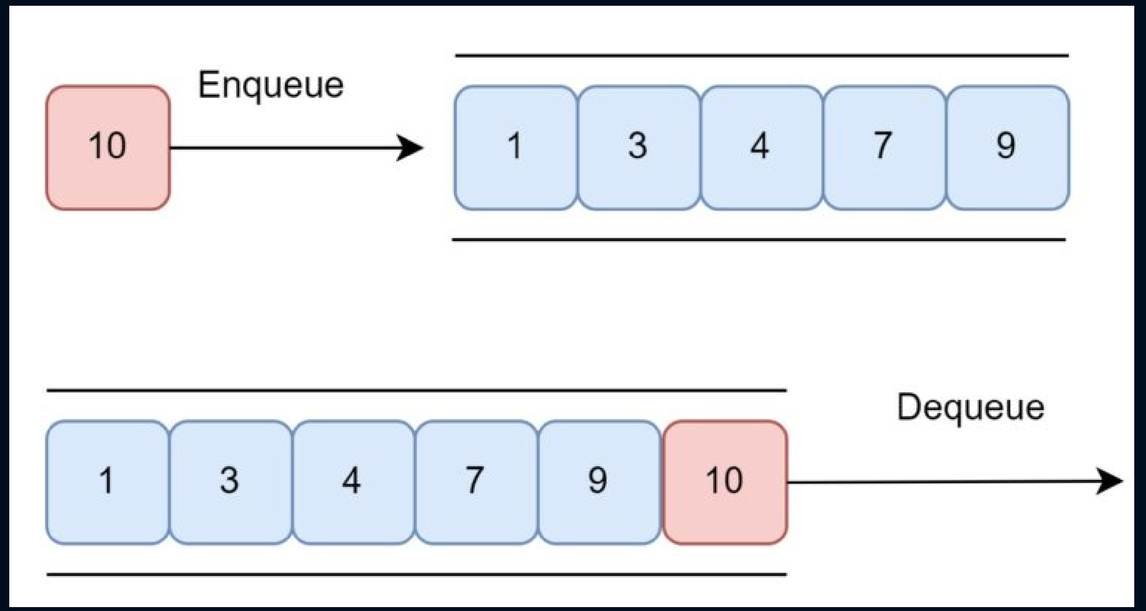
Fast insertion at the end and removal from the front

Limited Space

insertion and deletion of elements from the middle are time consuming

Tip: Use Queue<T> when you have a collection of objects that needs to follow a FIFO (First-In-First-Out) pattern, and you need to perform operations like Enqueue and Dequeue frequently.

Lists



Queue collection

Some performance arguments using collections

Mutable	Amortized	Worst Case	Immutable	Complexity
`Stack<T>.Push`	O(1)	O(`n`)	`ImmutableStack<T>.Push`	O(1)
`Queue<T>.Enqueue`	O(1)	O(`n`)	`ImmutableQueue<T>.Enqueue`	O(1)
`List<T>.Add`	O(1)	O(`n`)	`ImmutableList<T>.Add`	O(log `n`)
`List<T>.Item[Int32]`	O(1)	O(1)	`ImmutableList<T>.Item[Int32]`	O(log `n`)
`List<T>.Enumerator`	O(`n`)	O(`n`)	`ImmutableList<T>.Enumerator`	O(`n`)
`HashSet<T>.Add`, lookup	O(1)	O(`n`)	`ImmutableHashSet<T>.Add`	O(log `n`)
`SortedSet<T>.Add`	O(log `n`)	O(`n`)	`ImmutableSortedSet<T>.Add`	O(log `n`)
`Dictionary<T>.Add`	O(1)	O(`n`)	`ImmutableDictionary<T>.Add`	O(log `n`)
`Dictionary<T>` lookup	O(1)	O(1) – or strictly O(`n`)	`ImmutableDictionary<T>` lookup	O(log `n`)
`SortedDictionary<T>.Add`	O(log `n`)	O(`n` log `n`)	`ImmutableSortedDictionary<T>.Add`	O(log `n`)

More information on the official Microsoft Docs

When to use what

I want to...	Generic collection options	Non-generic collection options	Thread-safe or immutable collection options
Store items as key/value pairs for quick look-up by key	Dictionary<TKey,TValue>	Hashtable (A collection of key/value pairs that are organized based on the hash code of the key.)	ConcurrentDictionary<TKey,TValue> ReadOnlyDictionary<TKey,TValue> ImmutableDictionary<TKey,TValue>
Access items by index	List<T>	Array ArrayList	ImmutableList<T> ImmutableArray
Use items first-in-first-out (FIFO)	Queue<T>	Queue	ConcurrentQueue<T> ImmutableQueue<T>
Use data Last-In-First-Out (LIFO)	Stack<T>	Stack	ConcurrentStack<T> ImmutableStack<T>
Access items sequentially	LinkedList<T>	No recommendation	No recommendation
Receive notifications when items are removed or added to the collection. (implements INotifyPropertyChanged and INotifyCollectionChanged)	ObservableCollection<T>	No recommendation	No recommendation
A sorted collection	SortedList<TKey,TValue>	SortedList	ImmutableSortedDictionary<TKey,TValue> ImmutableSortedSet<T>
A set for mathematical functions	HashSet<T> SortedSet<T>	No recommendation	ImmutableHashSet<T> ImmutableSortedSet<T>

Collection Use Cases