100+ C# Collections Interview Q&A

Follow: https://www.linkedin.com/in/sandeeppal/

1. What is the difference between generic and non-generic collections in C#?

Answer:
Generic collections are type-safe and defined using generics (`<T>`), allowing you to specify the type of elements they hold. This ensures compile-time type checking and eliminates the need for casting.

Non-generic collections, on the other hand, store elements as objects (`object` type), requiring boxing/unboxing for value types and casting for reference types.

Example:

```
// Generic collection
List<int> numbers = new List<int>();
numbers.Add(10); // No boxing, type-safe

// Non-generic collection
ArrayList list = new ArrayList();
list.Add(10); // Boxing occurs
int value = (int)list[0]; // Needs casting
```

Real-world use case:
In applications dealing with specific data types (e.g., a list of customer IDs), generic collections are ideal. Non-generic collections may be used in legacy systems or when dealing with multiple data types.

---

2. What are some advantages of using generic collections over non-generic collections?

Answer:
Generic collections offer several advantages:

- **Type Safety:** Compile-time checking prevents runtime errors.
- **Performance:** Avoids boxing/unboxing of value types.
- **Code Clarity:** Cleaner code without explicit casting.

- **Reusability:** Generic code can work with any data type.

Example:

```
List<string> names = new List<string>();
names.Add("Alice"); // Valid
// names.Add(123); // Compile-time error
```

Real-world use case:
When managing a list of product names or order numbers, using `List<string>` or `List<int>` ensures that invalid data types are caught at compile time.

---

3. What is the use of the IEnumerable<T> interface in C# collections?

Answer:
`IEnumerable<T>` is the base interface for all generic collections that can be enumerated (looped over). It allows the use of `foreach` loops and LINQ queries.

It defines a single method:

```
IEnumerator<T> GetEnumerator();
```

Example:

```
List<string> items = new List<string> { "A", "B", "C" };
foreach (string item in items) // IEnumerable<string> in action
{
    Console.WriteLine(item);
}
```

Real-world use case:
When reading product data from a list or querying a database, `IEnumerable<T>` allows deferred execution and efficient data processing using LINQ.

---

4. What does ICollection<T> provide, and how does it differ from IEnumerable<T>?

Answer:
`ICollection<T>` extends `IEnumerable<T>` and adds features like:

- Counting (`Count` property)
- Adding and removing items (`Add`, `Remove`)
- Checking for existence (`Contains`)

Difference:

- `IEnumerable<T>` is **read-only and forward-only** iteration.
- `ICollection<T>` adds **modification capabilities**.

Example:

```
ICollection<int> numbers = new List<int>();
numbers.Add(5);
numbers.Remove(5);
Console.WriteLine(numbers.Count);
```

Real-world use case:
Use `ICollection<T>` when you need to manipulate the collection (add/remove items), such as managing an in-memory cart of products in a shopping application.

---

5. What is the role of the IList<T> interface in collections?

Answer:
`IList<T>` extends `ICollection<T>` and allows:

- Indexed access (like arrays)

- Inserting and removing at specific positions

Example:

```
IList<string> fruits = new List<string>();
fruits.Add("Apple");
fruits.Insert(0, "Banana"); // Insert at index 0
Console.WriteLine(fruits[1]); // Access by index
```

Real-world use case:
Use `IList<T>` when order matters and you need to access, insert, or remove elements at specific positions, like reordering tasks in a to-do list.

---

6. What is the purpose of IReadOnlyCollection<T> and IReadOnlyList<T> interfaces?

Answer:
These interfaces provide **read-only** access to collections:

- `IReadOnlyCollection<T>` provides `Count` and enumeration.
- `IReadOnlyList<T>` provides index-based access without modification.

Example:

```
IReadOnlyList<int> ids = new List<int> { 1, 2, 3 };
Console.WriteLine(ids[0]); // Access element
// ids[0] = 10; // Compile-time error – read-only
```

Real-world use case:
Useful in APIs where you want to expose collection data but prevent consumers from modifying it—e.g., returning a list of supported currencies from a service.

---

7. How do non-generic collections like ArrayList and Hashtable differ from their generic counterparts (List<T>, Dictionary<TKey, TValue>)?

Answer:

Feature	Non-Generic	Generic
Type Safety	No	Yes
Performance	Slower (boxing/unboxing)	Faster
Casting	Required	Not required
Syntax	Less readable	Clean and type-specific

Examples:

```
// Non-generic
ArrayList arr = new ArrayList();
arr.Add(1);
arr.Add("text"); // Allowed, but risky

// Generic
List<int> list = new List<int>();
list.Add(1);
// list.Add("text"); // Compile-time error

// Hashtable vs Dictionary
Hashtable ht = new Hashtable();
ht["id"] = 101;

Dictionary<string, int> dict = new Dictionary<string, int>();
dict["id"] = 101;
```

Real-world use case:
Generic collections are recommended for new development due to safety and performance. Non-generic collections are often found in older legacy systems.

8. What is the difference between IEnumerable<T> and ICollection<T> in C#?

Answer:

Feature	IEnumerable<T>	ICollection<T>
Read-only	Yes	No
Modification	Not supported	Supports Add, Remove, Clear
Count property	No	Yes (Count)

Example:

```
IEnumerable<string> names = new List<string> { "A", "B" };
// names.Add("C"); // Error

ICollection<string> nameCollection = new List<string> { "A", "B" };
nameCollection.Add("C"); // Allowed
```

Real-world scenario:
Use IEnumerable<T> for read-only, query-focused tasks like LINQ. Use ICollection<T> when managing and modifying the collection.

---

9. What is the difference between List<T> and ArrayList in C#?

Answer:

Feature	List<T>	ArrayList
Generic	Yes	No
Type Safety	Compile-time	Runtime (casting required)
Performance	Better (no boxing)	Slower for value types (boxing)

Example:

```
List<int> list = new List<int>();
list.Add(10); // Type-safe

ArrayList arrayList = new ArrayList();
arrayList.Add(10);
int num = (int)arrayList[0]; // Casting required
```

Best Practice:
Always prefer `List<T>` in new code. Use `ArrayList` only when working with legacy systems.

---

10. How do you iterate over a collection in C#?

Answer:
You can iterate using:

1. `foreach` loop (most common)

```
List<string> fruits = new List<string> { "Apple", "Banana" };
foreach (var fruit in fruits)
{
    Console.WriteLine(fruit);
}
```

2. `for` loop (when using index)

```
for (int i = 0; i < fruits.Count; i++)
{
    Console.WriteLine(fruits[i]);
}
```

3. `LINQ`

```
fruits.Where(f =>
f.StartsWith("A")).ToList().ForEach(Console.WriteLine);
```

Real-world tip:
Use `foreach` for simplicity, `for` when you need index, and `LINQ` for filtering and transforming data.

📘 C# List<T> – Interview Questions & Answers

1. What is a `List<T>` in C# and when should you use it?

Answer:
`List<T>` is a **generic collection** in C# that represents a dynamically sized list of elements. It resides in the `System.Collections.Generic` namespace and grows or shrinks as needed.

Use it when:

- You need a **resizable** array-like structure
- You want to perform frequent **insertions, deletions, and searches**

Example:

```
List<string> names = new List<string>();
names.Add("Alice");
```

---

2. How do you add elements to a `List<T>` in C#?

Answer:
You can use the `Add()` or `AddRange()` method.

Example:

```
List<int> numbers = new List<int>();
numbers.Add(10); // Add single item
numbers.AddRange(new int[] { 20, 30 }); // Add multiple
```

---

3. How do you remove elements from a `List<T>` in C#?

Answer:
Use methods like:

- `Remove(item)` – removes first occurrence
- `RemoveAt(index)` – removes by index
- `RemoveAll(predicate)` – removes all matching a condition
- `Clear()` – removes all items

Example:

```
numbers.Remove(10);
numbers.RemoveAt(0);
numbers.RemoveAll(x => x > 100);
numbers.Clear();
```

---

4. What is the time complexity of accessing an element in a `List<T>`?

Answer:
Accessing an element by index is **O(1)** (constant time) — same as arrays.

Example:

```
int first = numbers[0]; // O(1)
```

5. What methods does the `List<T>` class provide to search for an element?

Answer:

- `Contains(item)`
- `IndexOf(item)`
- `Find(predicate)`
- `FindAll(predicate)`
- `Exists(predicate)`
- `BinarySearch(item)` (for sorted lists)

Example:

```
bool hasItem = numbers.Contains(10);
int index = numbers.IndexOf(10);
var result = numbers.Find(x => x > 50);
```

---

6. What is the difference between `Add()` and `AddRange()` in a `List<T>`?

Answer:

Method	Purpose
`Add()`	Adds **one** item
`AddRange()`	Adds **multiple** items (collection)

Example:

```
list.Add(1); // One item
```

```
list.AddRange(new[] { 2, 3, 4 }); // Multiple
```

---

7. What does `Insert()` do in a `List<T>` and how is it different from `Add()`?

Answer:

- `Insert(index, item)` adds an item at a specific index.
- `Add(item)` adds to the **end** of the list.

Example:

```
list.Insert(0, 99); // Add at beginning
list.Add(100);      // Add at end
```

---

8. How can you sort a `List<T>` in C#?

Answer:
Use the `Sort()` method or provide a custom comparer.

Example:

```
list.Sort(); // Default sort (ascending)
list.Sort((a, b) => b.CompareTo(a)); // Descending
```

---

9. How do you reverse the elements of a `List<T>`?

Answer:
Use the `Reverse()` method.

Example:

```
list.Reverse();
```

Use case: Reversing order of recent messages or results.

---

10. How do you clear a `List<T>`?

Answer:
Use the `Clear()` method to remove all elements.

Example:

```
list.Clear();
```

---

11. What is the difference between `Find()` and `FindAll()` in `List<T>`?

Answer:

Method	Returns
`Find()`	First match
`FindAll()`	All matches in a new list

Example:

```
var firstEven = list.Find(x => x % 2 == 0);
var allEvens = list.FindAll(x => x % 2 == 0);
```

---

12. How does the `Contains()` method work in a `List<T>`?

Answer:
`Contains(item)` checks whether the item exists in the list. It uses **Equals()** internally.

Example:

```
bool exists = list.Contains(10);
```

Note: For custom objects, override `Equals()` and `GetHashCode()`.

---

13. What is the `Capacity` property in a `List<T>`?

Answer:
`Capacity` is the **number of elements the list can hold** before resizing. It is greater than or equal to `Count`.

Example:

```
list.Capacity = 100; // Optional performance tuning
```

---

14. What is the difference between `Count` and `Capacity` in `List<T>`?

Answer:

Property	Meaning
`Count`	Number of elements currently in the list
`Capacity`	Total allocated slots (memory reserved)

Example:

```
Console.WriteLine($"Count: {list.Count}, Capacity:
{list.Capacity}");
```

---

15. How do you check if a `List<T>` contains a duplicate element?

Answer:
Use `GroupBy`, `Distinct`, or nested loops.

Example:

```
bool hasDuplicates = list.Count != list.Distinct().Count();
```

---

16. How would you convert a List<T> into an array in C#?

Answer:
Use `ToArray()` method.

Example:

```
int[] array = list.ToArray();
```

Useful when interfacing with APIs that require arrays.

---

17. How do you remove duplicates from a List<T>?

Answer:

```
list = list.Distinct().ToList();
```

For custom objects, override `Equals()` and `GetHashCode()`.

---

18. How do you get the index of an element in a List<T>?

Answer:
Use `IndexOf(item)` or `FindIndex(predicate)`.

Example:

```
int index = list.IndexOf(10);
int indexByCondition = list.FindIndex(x => x > 100);
```

📘 C# Dictionary<TKey, TValue> – Interview Questions & Answers

---

1. What is a Dictionary<TKey, TValue> in C#?

Answer:
A Dictionary<TKey, TValue> is a **generic collection** in C# that stores data as **key-value pairs**. It provides **fast lookup**, **addition**, and **removal** of values based on their keys.

- Keys must be **unique**
- Keys must be **non-null** (for reference types)
- Values can be **null** if the type allows it

Example:

```
Dictionary<string, int> ages = new Dictionary<string, int>();
ages.Add("Alice", 30);
ages.Add("Bob", 25);
```

Use case: Storing user profiles by ID, or mapping product names to prices.

---

2. How do you add key-value pairs to a Dictionary?

Answer:
Use the `Add()` method or the indexer `[]`:

```
// Using Add()
dictionary.Add("key1", "value1");

// Using indexer
dictionary["key2"] = "value2";
```

Note: `Add()` throws an exception if the key already exists, while indexer will **overwrite** the value.

---

3. How do you remove a key-value pair from a Dictionary?

Answer:
Use the `Remove(key)` method:

```
dictionary.Remove("key1");
```

Returns `true` if the key was found and removed, `false` otherwise.

---

4. How do you check if a Dictionary contains a specific key or value?

Answer:

- `ContainsKey(key)` – checks for key existence
- `ContainsValue(value)` – checks for value

Example:

```
dictionary.ContainsKey("Alice");    // true/false
dictionary.ContainsValue(30);       // true/false
```

---

5. What is the time complexity of searching for a key in a Dictionary?

Answer:
The average time complexity is **O(1)** (constant time), thanks to **hash-based indexing**.

However, in worst-case scenarios (rare), it can degrade to **O(n)**.

---

6. What is the difference between `TryGetValue()` and indexer access in a Dictionary?

Answer:

Feature	`TryGetValue()`	Indexer (`dictionary[key]`)
Safe?	Yes – avoids exception	No – throws if key doesn't exist
Returns	Boolean (and output value)	Direct value
Use case	When unsure if key exists	When key is guaranteed to exist

Example:

```
if (dictionary.TryGetValue("Bob", out int age)) {
    Console.WriteLine(age);
}

// dictionary["Unknown"]; // throws KeyNotFoundException if missing
```

---

7. How do you get all keys and values from a `Dictionary<TKey, TValue>`?

Answer:

- `dictionary.Keys` – returns a collection of all keys
- `dictionary.Values` – returns a collection of all values

Example:

```
foreach (var key in dictionary.Keys)
    Console.WriteLine(key);

foreach (var value in dictionary.Values)
```

```
    Console.WriteLine(value);
```

---

8. What happens when you try to insert a duplicate key into a Dictionary?

Answer:

- Using `Add()` will throw a `System.ArgumentException`
- Using the indexer (`dictionary[key] = value`) will **overwrite** the existing value

Example:

```
dictionary.Add("John", 25);
dictionary.Add("John", 30); // Exception

dictionary["John"] = 30; // Overwrites the value safely
```

---

9. How do you iterate over a `Dictionary<TKey, TValue>`?

Answer:
Use a `foreach` loop with `KeyValuePair<TKey, TValue>`:

```
foreach (KeyValuePair<string, int> pair in dictionary)
{
    Console.WriteLine($"Key: {pair.Key}, Value: {pair.Value}");
}
```

Or use deconstruction (C# 7+):

```
foreach (var (key, value) in dictionary)
{
    Console.WriteLine($"{key} = {value}");
}
```

---

10. Can a Dictionary in C# have a null key? Why or why not?

Answer:

- For **reference types**, `null` keys are **not allowed** — adding one throws an `ArgumentNullException`.
- For **value types**, keys must be non-nullable (like `int`, `Guid`).

This restriction ensures the integrity of hashing, which is used internally by the dictionary.

---

11. What is the significance of the `KeyValuePair<TKey, TValue>` structure in Dictionary?

Answer:
`KeyValuePair<TKey, TValue>` represents a **single item** in a dictionary — a key-value pair.

Used in:

- Iteration
- LINQ queries
- Return values from dictionary enumerators

Example:

```
foreach (KeyValuePair<string, int> entry in dictionary)
{
    Console.WriteLine($"Key: {entry.Key}, Value: {entry.Value}");
}
```

---

12. How can you remove all items from a Dictionary?

Answer:
Use the `Clear()` method to remove all key-value pairs.

```
dictionary.Clear();
```

This resets the dictionary to an empty state.

📘 C# Queue<T> – Interview Questions & Answers

---

1. What is a Queue<T> in C#?

Answer:
A `Queue<T>` is a **generic collection** in C# that stores elements in a **First-In, First-Out (FIFO)** order.

- The first item added is the first to be removed.
- It is part of the `System.Collections.Generic` namespace.

Use case:
Modeling real-world queues — e.g., print jobs, task scheduling, or customer service systems.

Example:

```
Queue<string> orders = new Queue<string>();
orders.Enqueue("Order1");
orders.Enqueue("Order2");
```

---

2. How does a Queue<T> work internally?

Answer:
Internally, Queue<T> uses a **circular array** to efficiently manage memory and operations.

- **Head pointer** marks the front (next item to be dequeued).
- **Tail pointer** marks where the next item will be enqueued.
- Automatically resizes when capacity is exceeded.

This implementation ensures **constant time** operations for enqueue and dequeue.

---

3. What methods does the Queue<T> class provide to add or remove elements?

Answer:

Operation	Method	Description
Add	Enqueue()	Adds an item to the end of the queue
Remove	Dequeue()	Removes and returns the item at the front
Peek	Peek()	Returns the front item without removing it

Example:

```
Queue<int> queue = new Queue<int>();
queue.Enqueue(1);         // Add
int front = queue.Dequeue(); // Remove
```

---

4. What is the time complexity for the operations Enqueue() and Dequeue() in a Queue<T>?

Answer:

- `Enqueue()` → O(1) average case
- `Dequeue()` → O(1) average case

Due to the internal circular array and pointer arithmetic, both operations are highly efficient unless resizing is needed (which is O(n), but infrequent).

---

5. How do you check the first element in a `Queue<T>` without removing it?

Answer:
Use the `Peek()` method to view the front element without removing it.

Example:

```
Queue<string> tasks = new Queue<string>();
tasks.Enqueue("Task1");
string nextTask = tasks.Peek(); // Returns "Task1", does not remove it
```

Useful when you want to see what's next without modifying the queue.

---

6. How would you clear a `Queue<T>`?

Answer:
Use the `Clear()` method to remove all elements.

Example:

```
tasks.Clear();
```

After calling `Clear()`, the queue is empty (`Count == 0`).

---

7. How would you iterate through the elements of a `Queue<T>`?

Answer:
Use a `foreach` loop. Iteration does **not** modify the queue.

Example:

```
foreach (var order in orders)
{
    Console.WriteLine(order);
}
```

You can also use `.ToArray()` if needed:

```
string[] items = orders.ToArray();
```

---

8. How do you peek at the front element of a `Queue<T>` without dequeuing it?

Answer:
Again, use the `Peek()` method:

```
var front = queue.Peek();
```

Difference from `Dequeue()`:

- `Peek()` returns the front element **without** removing it.
- `Dequeue()` returns and **removes** the front element.

📘 C# Stack<T> – Interview Questions & Answers

---

1. What is a Stack<T> in C#?

Answer:
A Stack<T> is a **generic collection** in C# that stores elements in a **Last-In, First-Out (LIFO)** order.

- The last element **added** is the first one **removed**
- Belongs to System.Collections.Generic

Real-world use case:
Undo operations, browser history, expression evaluation.

Example:

```
Stack<int> numbers = new Stack<int>();
numbers.Push(10);
numbers.Push(20); // Top of the stack
```

---

2. How does a Stack<T> work internally?

Answer:
Internally, Stack<T> uses an **array-based dynamic storage** system:

- When capacity is exceeded, the internal array **resizes automatically** (typically doubles)
- The **top** of the stack is managed with a private index pointer

This structure provides **fast push and pop** operations (constant time on average).

---

3. What methods are available in a Stack<T> for adding and removing elements?

Answer:

Method	Description
`Push()`	Adds an element to the top
`Pop()`	Removes and returns the top element
`Peek()`	Returns top element without removing it
`Clear()`	Removes all elements

Example:

```
stack.Push(100);        // Add
int top = stack.Pop(); // Remove and return top
```

---

4. What is the time complexity for `Push()` and `Pop()` operations in a `Stack<T>`?

Answer:

- `Push()` → O(1) average, O(n) worst-case (if resizing needed)
- `Pop()` → O(1)

These operations are fast and efficient due to the internal array structure.

---

5. How do you access the top element in a `Stack<T>` without removing it?

Answer:
Use the `Peek()` method.

Example:

```
int top = stack.Peek();
```

This is useful when you just want to **inspect** the top element without altering the stack.

---

6. What is the purpose of the `Peek()` method in a `Stack<T>`?

Answer:
`Peek()` returns the **top element** without removing it. It's helpful for:

- Conditional checks
- Previewing what's next
- Preventing accidental removal

Example:

```
if (stack.Count > 0)
{
    var current = stack.Peek();
}
```

Throws `InvalidOperationException` if the stack is empty.

---

7. How do you check whether a `Stack<T>` is empty?

Answer:
Use the `Count` property.

```
if (stack.Count == 0)
{
    Console.WriteLine("Stack is empty");
}
```

Unlike some languages, C# stacks do not provide an `IsEmpty` property.

---

8. How do you clear a `Stack<T>`?

Answer:
Use the `Clear()` method to remove all elements.

```
stack.Clear();
```

After this, `Count` becomes 0, and the internal array is reset.

---

9. How would you iterate through a `Stack<T>`?

Answer:
Use a `foreach` loop, which iterates from **top to bottom** (LIFO order).

Example:

```
foreach (var item in stack)
{
    Console.WriteLine(item);
}
```

This does **not** modify the stack — it's read-only iteration.

📘 C# HashSet<T> – Interview Questions & Answers

---

1. What is a `HashSet<T>` in C# and when would you use it?

Answer:
`HashSet<T>` is a **generic collection** that stores **unique elements** with no particular order. It is optimized for **fast lookups**, **additions**, and **deletions**.

Use case:
When you want to store a collection of unique items without duplicates, such as user IDs, tags, or keywords.

```
HashSet<int> uniqueNumbers = new HashSet<int> { 1, 2, 3 };
uniqueNumbers.Add(4);
```

---

2. How does the HashSet<T> differ from a List<T> or Dictionary<TKey, TValue>?

Answer:

Feature	HashSet<T>	List<T>	Dictionary<TKey, TValue>
Allows duplicates?	No	Yes	Keys: No; Values: Yes
Order	No guaranteed order	Maintains insertion order	No guaranteed order
Lookup speed	O(1) average	O(n)	O(1) average for keys
Key-value pairs	No (only values)	No	Yes (key-value pairs)

---

3. What is the time complexity for adding, removing, or searching for an element in a HashSet<T>?

Answer:
All these operations generally have **O(1)** average time complexity due to the underlying **hash table** structure.

---

4. How do you remove duplicates from a collection using HashSet<T>?

Answer:
Add all elements from the collection to a HashSet<T>, which automatically removes duplicates.

```
List<int> numbers = new List<int> { 1, 2, 2, 3, 3, 4 };
HashSet<int> uniqueNumbers = new HashSet<int>(numbers);
```

Now, uniqueNumbers contains only unique values: {1, 2, 3, 4}.

---

5. Can a HashSet<T> store duplicate values?

Answer:
No, HashSet<T> **does not allow duplicates**. Attempting to add a duplicate value will return false and not change the set.

```
bool added = uniqueNumbers.Add(2); // returns false because 2
already exists
```

---

6. How do you perform a union or intersection between two HashSet<T> objects?

Answer:

Union: Combines all unique elements from both sets

```
set1.UnionWith(set2);
```

-

Intersection: Keeps only elements present in both sets

```
set1.IntersectWith(set2);
```

-

Example:

```
HashSet<int> set1 = new HashSet<int> { 1, 2, 3 };
HashSet<int> set2 = new HashSet<int> { 3, 4, 5 };

set1.UnionWith(set2);       // set1 = {1, 2, 3, 4, 5}
```

```
set1.IntersectWith(set2);   // set1 = {3, 4, 5}
```

---

7. What is the difference between `Add()` and `Contains()` in a `HashSet<T>`?

Answer:

Method	Purpose	Return Value
`Add(item)`	Adds item if not already present	`true` if added, `false` if duplicate
`Contains(item)`	Checks if item exists in the set	`true` if found, `false` otherwise

---

8. How would you iterate over a `HashSet<T>`?

Answer:
Use a `foreach` loop; the iteration order is **not guaranteed**.

```
foreach (var item in uniqueNumbers)
{
    Console.WriteLine(item);
}
```

📘 C# LinkedList<T> – Interview Questions & Answers

---

1. What is a `LinkedList<T>` in C#?

Answer:
`LinkedList<T>` is a **doubly linked list** collection in C#. It stores elements as nodes, where each node contains the data and references to the **previous** and **next** nodes.

- Allows efficient insertions and deletions anywhere in the list.
- Does **not** support indexed access like `List<T>`.

Example:

```
LinkedList<int> numbers = new LinkedList<int>();
numbers.AddLast(10);
numbers.AddLast(20);
```

---

2. How do you add elements to the start or end of a `LinkedList<T>`?

Answer:

- Use `AddFirst(value)` to add at the **start**.
- Use `AddLast(value)` to add at the **end**.

```
numbers.AddFirst(5);  // Adds 5 at the beginning
numbers.AddLast(30);  // Adds 30 at the end
```

---

3. How would you remove an element from a `LinkedList<T>`?

Answer:
Use `Remove(value)` to remove the first occurrence of the specified value, or `RemoveFirst()` / `RemoveLast()` to remove from the start or end respectively.

```
numbers.Remove(10);        // Removes the first node with value 10
numbers.RemoveFirst();     // Removes the first node
numbers.RemoveLast();      // Removes the last node
```

---

4. How do you iterate through a LinkedList<T>?

Answer:
Use a `foreach` loop to traverse the linked list from start to end.

```
foreach (var num in numbers)
{
    Console.WriteLine(num);
}
```

---

5. What is the time complexity of adding or removing elements in a LinkedList<T>?

Answer:

- Adding or removing at the **start or end**: **O(1)**
- Adding or removing at an **arbitrary position** (if you already have the node reference): **O(1)**
- Searching for a node by value: **O(n)**, because traversal is required

---

6. How does a LinkedList<T> compare to a List<T> in terms of performance?

Answer:

Operation	LinkedList<T>	List<T>
Indexed access	O(n) (no indexing)	O(1) (direct access)
Add/Remove at start/end	O(1)	O(n) (start), O(1) (end)
Add/Remove in middle	O(1) (with node ref)	O(n) (shifts elements)

Memory overhead	Higher (extra pointers)	Lower (array storage)

Summary:
Use `LinkedList<T>` when you need fast insertions/deletions anywhere and don't require indexed access. Use `List<T>` for fast random access and better memory efficiency.

📘 C# SortedList<TKey, TValue> – Interview Questions & Answers

---

1. What is a `SortedList<TKey, TValue>` in C#?

Answer:
`SortedList<TKey, TValue>` is a collection of key-value pairs that **maintains the elements sorted by keys**.

- Implements both `IDictionary<TKey, TValue>` and `IList<KeyValuePair<TKey, TValue>>`.
- Keys are automatically sorted in **ascending order** based on their natural comparer or a provided comparer.

Example:

```
SortedList<int, string> sortedList = new SortedList<int, string>();
sortedList.Add(3, "Three");
sortedList.Add(1, "One");
sortedList.Add(2, "Two");
```

The elements are stored sorted by key: 1, 2, 3.

---

2. How does a `SortedList<TKey, TValue>` differ from a `Dictionary<TKey, TValue>`?

Feature	SortedList<TKey, TValue>	Dictionary<TKey, TValue>
Order	Maintains keys in sorted order	No guaranteed order
Internal storage	Uses two arrays (keys & values)	Uses a hash table
Lookup complexity	O(log n) (binary search)	O(1) average
Insertion complexity	O(n) (due to shifting elements)	O(1) average
Memory overhead	Lower (arrays)	Higher (hash buckets, overhead)

---

3. How do you add, remove, or search for elements in a `SortedList<TKey, TValue>`?

Add:

```
sortedList.Add(4, "Four");
```

Remove:

```
sortedList.Remove(2); // Remove element with key 2
```

Search (by key):

```
bool exists = sortedList.ContainsKey(3);
string value = sortedList[3]; // Access value by key
```

---

4. What is the time complexity of searching for a key in a `SortedList<TKey, TValue>`?

Answer:
Searching by key uses **binary search**, so the time complexity is **O(log n)**.

---

5. How would you iterate through a `SortedList<TKey, TValue>`?

Answer:
You can use a `foreach` loop over `KeyValuePair<TKey, TValue>` elements, which iterates in sorted key order:

```
foreach (var kvp in sortedList)
{
    Console.WriteLine($"Key: {kvp.Key}, Value: {kvp.Value}");
}
```

You can also iterate over keys or values separately:

```
foreach (var key in sortedList.Keys) { /* ... */ }
foreach (var value in sortedList.Values) { /* ... */ }
```

📘 C# SortedSet<T> – Interview Questions & Answers

---

1. What is a `SortedSet<T>` in C#?

Answer:
`SortedSet<T>` is a collection that stores **unique elements** in **sorted order**.

- Implements a **self-balancing binary search tree** (usually a Red-Black Tree).
- Automatically maintains elements in **ascending sorted order**.
- Provides set operations like union, intersection, and difference.

Example:

```
SortedSet<int> sortedSet = new SortedSet<int> { 5, 1, 3 };
sortedSet.Add(2);  // Sorted order maintained: {1, 2, 3, 5}
```

---

2. How does a `SortedSet<T>` differ from a `HashSet<T>`?

Feature	SortedSet<T>	HashSet<T>
Ordering	Maintains sorted order	No guaranteed order
Implementation	Balanced binary search tree	Hash table
Lookup complexity	O(log n)	O(1) average
Memory overhead	Higher (tree nodes)	Lower (hash buckets)
Use case	When sorted data or range queries needed	Fast insertion and lookup without ordering

---

3. What are the common use cases for a `SortedSet<T>`?

Answer:

- When you need a **collection of unique elements** in sorted order.
- Performing **range queries** or retrieving elements in sorted order.
- Implementing **mathematical set operations** efficiently.
- Examples:
  - Leaderboards
  - Scheduling tasks sorted by priority
  - Auto-complete suggestions sorted alphabetically

---

4. How do you perform operations like union, intersection, and difference with `SortedSet<T>`?

Answer:

Operation	Method	Description
Union	`UnionWith()`	Adds all elements from another set
Intersection	`IntersectWith()`	Keeps only elements present in both sets
Difference	`ExceptWith()`	Removes elements found in another set

Example:

```
SortedSet<int> set1 = new SortedSet<int> { 1, 2, 3 };
SortedSet<int> set2 = new SortedSet<int> { 3, 4, 5 };

set1.UnionWith(set2);        // {1, 2, 3, 4, 5}
set1.IntersectWith(set2);    // {3, 4, 5} (if applied on the original set1)
set1.ExceptWith(set2);       // {1, 2} (if applied on the original set1)
```

---

5. How do you iterate through a `SortedSet<T>`?

Answer:
Use a `foreach` loop which iterates over the elements in **sorted ascending order**:

```
foreach (var item in sortedSet)
{
    Console.WriteLine(item);
}
```

---

6. What is the time complexity for common operations like `Add()`, `Remove()`, and `Contains()` in `SortedSet<T>`?

Answer:
Due to the underlying balanced tree structure, these operations have **O(log n)** time complexity.

📘 C# Collection Initializers & LINQ – Interview Questions & Answers

---

1. How do you initialize a collection using collection initializers in C#?

Answer:
Collection initializers allow you to create and populate a collection in a concise way at the time of declaration.

Example:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
Dictionary<string, int> ages = new Dictionary<string, int>
{
    { "Alice", 30 },
    { "Bob", 25 }
};
```

This syntax internally calls the collection's `Add()` method for each element.

---

2. How do you add elements to a collection using LINQ in C#?

Answer:
LINQ itself doesn't modify collections directly but **produces new collections** based on queries.

You typically combine LINQ with collection methods to add elements, for example:

```
var evenNumbers = new List<int> { 2, 4, 6 };
var allNumbers = new List<int> { 1, 2, 3, 4, 5, 6 };

var combined = allNumbers.Where(n => n % 2 == 0).ToList(); //
Filters even numbers
```

If you want to add LINQ results to a collection:

```
List<int> filteredNumbers = allNumbers.Where(n => n % 2 ==
0).ToList();
```

---

3. How can you perform a LINQ query on a collection to filter, sort, or group elements?

Answer:

- **Filter:** Use `Where()` to select elements based on a condition.
- **Sort:** Use `OrderBy()` or `OrderByDescending()`.
- **Group:** Use `GroupBy()` to group elements by a key.

Example:

```
var products = new List<Product> { ... };

// Filter products with price > 100
var expensiveProducts = products.Where(p => p.Price > 100);

// Sort products by name
var sortedProducts = products.OrderBy(p => p.Name);

// Group products by category
var groupedProducts = products.GroupBy(p => p.Category);
```

---

4. What is the benefit of using LINQ to manipulate collections in C#?

Answer:

- **Concise and readable code:** LINQ makes querying collections clear and expressive.
- **Declarative style:** You focus on *what* to retrieve, not *how*.
- **Powerful operations:** Filtering, sorting, grouping, joining, projecting, and more.
- **Deferred execution:** Queries execute only when results are needed, improving performance.
- **Strongly typed:** Compile-time checking and IntelliSense support.

---

5. How can you convert a collection to a different type using LINQ?

Answer:

Use methods like `ToList()`, `ToArray()`, or `ToDictionary()` to convert LINQ query results to different collection types.

Examples:

```
var numbers = new int[] { 1, 2, 3, 4, 5 };

// Convert to List<int>
List<int> numberList = numbers.ToList();

// Convert to array
int[] numberArray = numberList.ToArray();

// Convert to dictionary (key = number, value = square)
Dictionary<int, int> numberDict = numbers.ToDictionary(n => n, n =>
n * n);
```

📘 C# Thread-Safe Collections – Interview Questions & Answers

---

1. What is a `ConcurrentDictionary<TKey, TValue>` in C#?

Answer:
`ConcurrentDictionary<TKey, TValue>` is a thread-safe dictionary designed for **concurrent access** by multiple threads without needing external synchronization (locks).

- Supports atomic operations like adding, updating, and removing items.
- Useful in multi-threaded scenarios where data integrity and performance are critical.

Example:

```
ConcurrentDictionary<int, string> concurrentDict = new
ConcurrentDictionary<int, string>();
concurrentDict.TryAdd(1, "One");
concurrentDict.TryUpdate(1, "Uno", "One");
```

---

2. How does `ConcurrentQueue<T>` differ from a regular `Queue<T>`?

Answer:

Feature	`ConcurrentQueue<T>`	`Queue<T>`
Thread safety	Designed for concurrent access	Not thread-safe; requires locks
Locking mechanism	Internal lock-free or fine-grained locking	No internal synchronization
Suitable for	Multi-threaded producer-consumer patterns	Single-threaded scenarios or external synchronization

`ConcurrentQueue<T>` allows safe enqueueing and dequeueing by multiple threads simultaneously without corrupting the data.

---

3. How do you ensure thread safety when accessing collections in a multi-threaded environment?

Answer:

- **Use thread-safe collections** provided by .NET (`ConcurrentDictionary`, `ConcurrentQueue`, `BlockingCollection`, etc.).
- **Use synchronization primitives** like `lock`, `Mutex`, `Semaphore`, or `ReaderWriterLock` around critical sections when using non-thread-safe collections.
- Avoid shared mutable state or design the program to minimize contention.
- Use **immutable collections** when possible to eliminate synchronization.

---

4. What is the difference between `BlockingCollection<T>` and a regular `Collection<T>`?

Answer:

Feature	`BlockingCollection<T>`	`Collection<T>`
Thread safety	Thread-safe for adding and taking items	Not thread-safe
Blocking behavior	Supports blocking and bounding (waits when empty/full)	No blocking behavior
Use case	Producer-consumer scenarios	General-purpose collection
Additional features	Supports bounded capacity and cancellation	Basic collection

`BlockingCollection<T>` wraps around other thread-safe collections and provides blocking and bounding capabilities, ideal for producer-consumer queues.

📘 C# Collections: Performance & Memory Considerations – Interview Q&A

---

1. How do you measure the performance of collection operations in C#?

Answer:

- Use the **`Stopwatch`** class from `System.Diagnostics` to time operations accurately.
- Profile your code using tools like **Visual Studio Profiler**, **dotTrace**, or **PerfView** for deeper insights.
- Measure specific operations like add, remove, search, or iteration by running them multiple times and averaging results.

Example:

```
var stopwatch = Stopwatch.StartNew();
list.Add(1000);
stopwatch.Stop();
Console.WriteLine($"Add operation took {stopwatch.ElapsedTicks}
ticks");
```

---

2. How does the memory usage of a `List<T>` compare to a `LinkedList<T>`?

Answer:

- `List<T>` uses a **contiguous array** internally, so memory is compact and cache-friendly.
- `LinkedList<T>` stores elements in nodes with **extra pointers** (`Next` and `Previous`), leading to more memory overhead.
- Therefore, `List<T>` generally has **lower memory usage** and better cache performance than `LinkedList<T>`, especially for large collections.

---

3. What are some performance trade-offs when choosing between different collections?

Answer:

Scenario	Best Choice	Trade-offs
Fast indexed access	`List<T>`	Slower inserts/removes in the middle
Frequent insert/delete at ends	`LinkedList<T>`	No fast indexed access; higher memory use
Fast lookups by key	`Dictionary<TKey, TValue>`	No sorted order
Sorted key-value pairs	`SortedList<TKey, TValue>` or `SortedDictionary<TKey, TValue>`	Slower inserts vs Dictionary
Thread-safe multi-thread use	`ConcurrentDictionary` / `ConcurrentQueue`	Slight overhead for synchronization

---

4. How can you optimize the memory usage of a collection in C#?

Answer:

- **Pre-allocate capacity** when you know the expected size (e.g., `new List<T>(capacity)`) to avoid frequent resizing.
- Use **value types** or structs when appropriate to reduce reference overhead.
- Choose collections with **lower overhead** for your use case (e.g., `List<T>` instead of `LinkedList<T>` if indexing is needed).
- Use **immutable collections** or pooling to minimize allocations.
- Avoid unnecessary boxing/unboxing by using generic collections instead of non-generic.
- Regularly **trim** collections if supported (e.g., `List<T>.TrimExcess()`).

📘 C# Advanced Collection Topics – Interview Questions & Answers

---

1. How would you implement a custom collection in C#?

Answer:
To implement a custom collection:

- Derive from existing base classes like `Collection<T>`, `List<T>`, or implement interfaces such as `ICollection<T>`, `IEnumerable<T>`, or `IList<T>`.
- Override or implement necessary methods like `Add()`, `Remove()`, `GetEnumerator()`, and indexers.
- Provide custom behavior, validation, or constraints as needed.

Example:

```
public class MyCustomCollection<T> : Collection<T>
{
    protected override void InsertItem(int index, T item)
```

```
    {
        // Custom validation
        if (item == null) throw new
ArgumentNullException(nameof(item));
        base.InsertItem(index, item);
    }
}
```

---

2. What is a `Collection<T>` class in C# and how does it differ from other collections?

Answer:

- `Collection<T>` is a **base class** for creating custom collections.
- It wraps an `IList<T>` internally and provides virtual methods for insert, remove, and clear, allowing easy customization.
- Unlike `List<T>`, which is optimized for performance, `Collection<T>` focuses on **extensibility**.
- Useful when you want to create a collection with customized behaviors (e.g., validation, event firing).

---

3. What is the role of `IComparer<T>` and `IEqualityComparer<T>` in sorting and comparing collections?

Answer:

- `IComparer<T>` defines a method `Compare(T x, T y)` for **custom sorting** logic (used in sorting operations like `Sort()`).
- `IEqualityComparer<T>` defines methods `Equals(T x, T y)` and `GetHashCode(T obj)` to determine **equality and hashing** (used in dictionaries,

sets, etc.).

- They allow collections to customize how items are compared or checked for equality, beyond default implementations.

---

4. How do you perform deep cloning or deep copying of a collection in C#?

Answer:

- Deep cloning copies the collection and all objects inside it recursively.
- Ways to deep clone:
    - Implement `ICloneable` in your objects with deep clone logic.
    - Use serialization (binary, XML, JSON) to serialize and deserialize objects.
    - Manually create new instances of each item.

Example (manual):

```
List<MyClass> DeepClone(List<MyClass> original)
{
    return original.Select(item => item.Clone()).ToList();
}
```

Note: `MyClass` must implement a `Clone()` method that performs deep copy.

---

5. How would you implement a generic collection in C# for a custom object?

Answer:

- Define your custom object class.

- Create a collection class that holds objects of that type using generics or directly.

Example:

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class EmployeeCollection : Collection<Employee>
{
    // You can add custom methods specific to Employee collection
here
}
```

Or simply use `List<Employee>` directly for flexibility.

---

6. How do you use a `SortedList<T>` to store items in a specific order in C#?

Answer:

- Actually, `SortedList<TKey, TValue>` stores key-value pairs sorted by keys.
- To store items in a specific order, use the key to represent the sorting criteria.
- Keys must be unique and implement `IComparable` or provide a custom `IComparer`.

Example:

```
SortedList<int, string> sortedList = new SortedList<int, string>();
sortedList.Add(10, "Ten");
sortedList.Add(5, "Five");
sortedList.Add(20, "Twenty");
```

```
// Items automatically sorted by keys: 5, 10, 20
```

If you want to sort by custom criteria, implement an `IComparer` and pass it to the `SortedList` constructor.