**What is Data Structure (and why)**

It is a way of organizing all data items and their relationships to each other inside the program in order to deal with them. It affects the design of both structural and functional aspects of the program. It is how you organize, manage and store data for efficiency reasons.

It is not only used for organizing data. It also used for processing, retrieving and storing data.

There are different basic and advanced types of data structures that are sued in almost every program or software system that has been developed. So we must have a good knowledge of data structures.

They are an integral part of computers used for the arrangement of data in memory. They are essential and responsible for organizing processing, accessing and storing data efficiently. But this is not all. Various types of data structures have their own characteristics, features, applications, advantages and disadvantages.

**Calcification of Data Structures**

**Primitive (Basic):**

Is generally a basic structure that is usually built into the language and directly operated upon the machine instructions, such as Integer, Float, Char, Pointer, etc...

**NON-Primitive (Advanced):**

**Linear, Non-Linear:**

* Complex/Sophisticated data structures derived for primitive DS.
* Emphasize on structuring of group of homogenous (same type) or heterogenous (different type) data items.
* The design of an efficient data structure must take operations to be performed on data structure.

**Linear vs Non-Linear Data Structures:**

* **Linear:**
  + Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements is called a linear data structure.
* **Non-Linear:**
  + Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can’t traverse all the elements in a single run only.
  + Example of this data structure are Tree, Graph, etc.

**Static vs Non-Static:**

* **Static:**
  + It has a fixed memory size. It is easier to access the elements in a static data structure.
  + Example of this one is an array.
* **Dynamic:**
  + In this one the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
  + Examples are Stack, Queue, etc.

**Operation on Data Structures:**

Create, Update, Search, Select, Sorting, Merging, Destroy or Delete.

**Boxing**

It is the process of converting a value type to a reference type. This involves wrapping a value type like (int, float, char) in an object or any interface type implemented by this value type.

**Example:**

using System;

class Program

{

    static void Main()

    {

        int valType = 10;

        object objType = valType; // Boxing

        Console.WriteLine("Value Type: " + valType);

        Console.WriteLine("Object Type: " + objType);

    }

}

**Expected Output:**

Value Type: 10

Object Type: 10

**Output Explanation:**

The output demonstrates the boxing process where valType, an integer (value type), is boxed into objType (object type). Both display the same value, but objType is a reference type stored in the heap.

**Conclusion:**

Boxing is a fundamental concept in C#, allowing value types to be treated as objects. While it necessary in certain scenarios, developers should be aware of its performance.

**Unboxing**

It is the reverse process of Boxing, where the value type is extracted from the object. It’s crucial to ensure the type being Unboxed matches the type of the object.

**Example:**

using System;

class Program

{

    static void Main()

    {

        int valType = 10;

        object objType = valType; // Boxing

        int unboxedValType = (int)objType; // Unboxing

        Console.WriteLine("Unboxed Value: " + unboxedValType);

    }

}

**Output:**

Unboxed Value: 10

Explanation:

The program demonstrates Unboxing, where the value 10 is retrieved from objType (the boxed object) and stored back in UnboxedValType, a value type.

**Key Point**

Unboxing requires the exact data type match, otherwise, it results in InvalidCastException.

**Conclusion:**

Unboxing is a critical operation in C# that retrieves values from object. Proper type matching is essential for successful unboxing.

**Introduction to Collections**

**What are Collections:**

* Collections are data structures used to store and organize groups of related objects in memory.
* They are sophisticated ways to store and manage data in C#. they offer more flexibility and functionality compared to basic array types.
* They allow for dynamic memory allocation, meaning the size of the collection can grow or shrink as needed.

**Why Use Collections:**

* They are used to store, retrieve, manipulate and communicate aggregate data.
* They provide efficient ways to handle large amount of data with built-int methods for common tasks.
* Collection provide efficient ways to manipulate and manage data, making programming tasks easier and more efficient.

**Common operations on collections:** Adding, removing, modifying, and accessing elements.

**Exploring Types of Collections:**

* The System.Collections and System.Collection.Generic name spaces.
  + These namespaces include various collection types. System.Collections contains non-generic collections.

**Common Collections types in C#:**

1. **List<T>:** A List<T> is a collection of objects that can be accessed by index. It functions like a dynamic array, which can automatically resize as needed. It’s versatile and suitable for sorting and manipulating a list of objects of a specific types.
2. **SortedList<TKey, TValue>:** It is a collection that maintain its elements in sorted order. It’s a combination of an array and hash table, providing fast lookups as well as maintaining a sorted order.
3. **Dictionary<TKey, TValue>:** This collection stores key-value pairs. It enables fast retrieval of values based on keys, making it ideal for situations where you need to access elements quickly and uniquely, like lookup table.
4. **HashSet<T>:** It stores a set of unique elements. It’s useful for operations that requires uniqueness for each element and is efficient in performing set operations like union or intersection.
5. **Stack<T>:** It represents a Last-In-First-Out (LIFO) structure. It’s perfect for scenarios that require reverse order processing, such as undo mechanism in applications.
6. **Queue<T>:** Representing First-In-First-Out (FIFO) structure, it is great for tasks where you need to process items in the order they where added, like task scheduling.
7. **linkedList<T>:** This is a doubly linked list, where each element points to both its previous and next element. It allows for efficient insertions and deletions at any point in the list.
8. **ObservableCollection<T>:** this collection is used primarily in data binding, typically in UI context. It notifies listeners of dynamic changes, like when items get added, removed or the whole list is refreshed.
9. **CurrentDictionary<TKey, TValue>:** A thread safe version of dictionary, this one is designed for concurrent access. It’s useful in multi-threading applications where different threads need to add or remove items simultaneously.
10. **BitArray:** this one manages compact array of bit values, which are represented as Booleans. It’s used in scenarios where you need to store bits but don’t need the overhead of a Boolean array.

Each of this collection types in C# serves specific purpose and choosing the right type depends on the requirements of the application or the specific problem you’re solving.

**Conclusion:**

In this lesson, we explored the fundamental of collections in C#, including various collection types and their characteristics. By understanding collection, you will be able to efficiently mange and manipulate data in your applications.

**Generic vs Non-Generic Collections**

1. **Generic Collections:**

**What are Generic Collections?**

* They are part of the System.Collections.Generic namespace.
* Generics allow us to create reusable code that can work with different types.
* Generics introduce the concept of type parameters to collections, making them more flexible and type-safe.
* They allow the collections to store any data type and prevent runtime type errors.
* They allow you to specify the type of objects they store, for example List<int>.
* They offer type safety, better performance, and reduced need for boxing/unboxing.

Advantages of Generic Collections

* Type Safety: They store elements of a specified type, reducing runtime errors.
* Performance: No need for boxing/unboxing of value types, which improves performance.
* Reduce Memory Overhead: They directly store elements without converting them to object type.
* Code Reusability: Avoid code duplication by creating generic algorithms and data structures.

**Key Generic Collections**

1. List<T>: A list of elements that can be accessed by index.
2. Dictionary<TKey, TValue>: A collection of key-value pairs.
3. Queue<T>: A first-in, first-out (FIFO) collection of objects.
4. Stack<T>: A last-in, first-out (LIFO) collection of objects.
5. HashSet<T>: A collection of unique and unordered elements.
6. LinkedList<T>: A double – linked list.
7. SortedSet<T>: A collection of objects that maintains order.
8. SortedDictionary<TKey, TValue>: A dictionary with sorted keys.
9. SortedList<TKey, TValue>: Similar to SortedDictionary but with different performance characteristics.
10. ConcurrentDictionary<TKey, TValue>: A thread-safe dictionary used in concurrent scenarios.
11. BlockingCollection<T>: Provides blocking and bounding capabilities for thread-safe collections.
12. ConcurrentBag<T>: An unordered collection of objects suitable for concurrent scenarios.
13. ConcurrentQueue<T>: A thread-safe FIFO collection.
14. ConcurrentStack<T>: A thread-safe LIFO collection.
15. **Non – Generic Collections**

**What are Non – Generic Collections?**

* Non – Generic collections are part of the System.Collections namespace.
* They store elements as object types, allowing them to hold any data type.
* They require boxing/unboxing for value types.

**Disadvantages of Non – Generic?**

* Type Unsafe: Can store any type of object, leading to runtime errors.
* Performance Overhead: Boxing/Unboxing of value types impacts performance.
* Memory Overhead: Storing value types as object consumes more memory.

**Key Non – Generic Collections:**

1. ArrayList: A dynamically resizable collection.
2. Hashtable: A collection of key-value pairs organized based on the hash code of the key.
3. Queue: A first-in, first-out (FIFO) collection.
4. Stack: A last-in, first-out (LIFO) collection.
5. SortedList: A collection of key-value pairs that are sorted by the keys and are accessible by key and index.
6. BitArray: Manages compact array of bit values, which are represented as Booleans.
7. HybridDictionary: Implements IDictionary using ListDictionary while the collection is small, and then switching to Hashtable as the collection grows.
8. ListDictionary: A simple, small dictionary implemented as a singly linked list.
9. NameValueCollection: Represents a collection of associated string keys and string values that can be accessed either with the key or with the index.
10. OrderedDictionary: A collection of key-value pairs that are accessible by the key or index.
11. StringCollection: A collection of strings.
12. StringDictionary: A collection of associated string keys and string values with a hash table implementation.

**Conclusion:**

Understanding the distinction between generic and non-generic collections is crucial for selecting the right type of collection in C#. while generic collections are preferred for their type safety and performance benefits, non – generic collections can still be useful in scenarios requiring heterogenous data storage.

These collections offer a wide range of functionalities and characteristics, making them suitable for various scenarios in programming. The choice between generic and non-generic collections typically depends on factors like type safety, performance requirement and specific use cases.

**What is List**

It is a generic collection class in the .NET Framework. It’s used to store a collection of objects of the same type. Unlike array List is dynamic, meaning it can automatically resize as needed.

**Key Concepts:**

* Generic Collection: T in List<T> is a type parameter, meaning that you can create a list of any type (e.g., List<int>, List<string>, List<CutomeType>).
* Dynamic Sizing: Automatically resize itself, offering more flexibility that traditional arrays.
* Zero Based Index: Like arrays, lists use Zero-Based indexing.
* Strongly Typed: Ensures type safety. You can’t add an int to a list of strings.
* Capacity and Count:
  + Count: Is the number of elements actually in the list.
  + Capacity: Is the number of elements the list can store before resizing.
* Thread Safety: It is not thread-safe. For thread – safe collections, consider using ConcurrentBag or other collections.

**Conclusion:**

It is a versatile and powerful collection class, suitable for a wide range of applications. Its dynamic nature, coupled with the powerful features provided makes it a go-to choice for storing and manipulating collections of objects.

**Working with List:**

Sample code in the folder inside the List solution there is a project named list.

**Code explanation:**

1. Initializing: A List<int> named numbers is created to store integers.
2. Adding Elements: The Add method is used to add elements to the list.
3. Count Property: This one is used to get the total number of elements in the list.
4. Index-Based Access: Elements in the list are accessed using their indices.
5. Modifying Elements: The value of an element at a specific index is modified.

**Conclusion:**

This lesson demonstrated the basic using of List<int> in C#. understanding how to create lists, add items, access and modify elements, and utilize properties like Count are fundamental skills for working with collections.

**Inserting Elements into a List:**

In this lesson, we’ll delve into the process of inserting elements into List.

**Code Example:**

Inside the List solution -> InsertingElements (Project).

**Code Explanation:**

1. Adding an Element at the End:
   1. The Add method append an element to the end of the list.
2. Inserting an Element at a Specific Position:
   1. The Insert method inserts an element at the specified index.
3. Inserting Multiple Elements:
   1. The InsertRange method allows inserting multiple elements from another collection at a specified index.

**Conclusion:**

This lesson covered various methods for inserting elements into a List. Understanding how to add elements at specified position or multiple elements at once allows for more sophisticated list manipulation and is essential for effective programming.

Time and Space Complexity:

* Insert:
  + Worst Case: when inserting at the beginning, needs to shift all existing elements O(n).
  + Best Case: when inserting at the end O (1).
  + Only needs a constant extra space.
* insertRange:
  + O(n + m).
  + The n is the number of existing elements that need to be shifted.
  + The m is the number of elements being inserted
  + Space complexity O(m) size of the collection being inserted.

Both operations may trigger array resizing if capacity is exceeded. Inserting at the end is more efficient than at the beginning.

For frequent insertion at the beginning, consider using LinkedList.

Internal array resizing can occasionally make the operation more expensive.

**Remove Items from List:**

**Code Example:**

Inside the List solution -> RemoveItems (Project).

**Code Explanation:**

1. Removing Items by Value:
   1. The Remove method removes the first occurrence of a specified object from the list.
   2. Worst case time complexity is O(n).
2. Removing Item by Index:
   1. The RemoveAt method removes item at a specified index.
   2. Worst case time complexity is O(n).
3. Removing Multiple Items:
   1. The RemoveAll method removes all the elements that matches the conditions defined by the specified predicate.
   2. Worst case time complexity is O(n).
4. Removing range of items:
   1. The RemoveRange remove the items from the start index to the end of range index (last item not included in the range).
   2. Worst case time complexity is O(n).

All remove operations require shifting remaining elements, however,

RemoveAt is faster than remove when you know the index.

Note: Remove operation don’t reduce the capacity of the list.

**Looping Through a List:**

**Code Example:**

Inside the List solution -> LoopingThrough (Project).

**Code Explanation:**

List.ForEach: A method provided by the list class that takes an action (in this case, a lambda expression) and applies it to each element in the list.

**Conclusion:**

Looping through lists is a common operation in C#. this lesson covered three primary methods to iterate over a list using a for loop, foreach loop and the List.ForEach method. Understanding these methos is crucial for performing operations on each element.

**Aggregating List Data Using LINQ:**

In this lesson, we focus on using Language Integrated Query (LINQ) to perform aggregation operations on the list. LINQ is a powerful feature in .NET that provides a convenient and efficient way to query and manipulate data in collections.

**Code Example:**

Inside the List solution -> LinqAggregation (Project).

**Filtering Data with LINQ:**

In this lesson we will delve into LINQ for filtering data within a list. LINQ provides a flexible and powerful way to query collections. We will explore various filtering techniques using the given list.

**Code Example:**

Inside the List solution -> LinqAggregation (Project).

**Concluson:**

This lesson demonstrates the versatility of LINQ for filtering data. With LINQ complex queries can be executed with concise readable code, making it an invaluable tool for data manipulation.

**Sorting List:**

**Code Example:**

Inside the List solution -> SortingList (Project).

**Explanation:**

List.Sort (): Uses introspective sort (hybrid of quicksort, heapsort and insertion sort) time and space complexity O (n log n).

List.Reverse (): Swaps element from both ends toward the middle, time complexity O(n).

OrderBy and OrderByDescending: time complexity O(n log n), space complexity O(n) – creates a new sequence rather than sorting in place.

Key Differences:

1. In-place vs new sequence:
   1. List.Sort: modifies the original collection.
   2. LINQ’s OrderBy: Returns a new sequence without modifying the original.
2. Stability:
   1. LINQ’s OrderBy: Is stable (preserve order of equal elements).
   2. List.Sort: are not guaranteed to be stable.
3. Deferred Execution:
   1. LINQ operation uses deferred execution (only evaluated when enumerable).
   2. Direct sort methods execute immediately.
4. Memory usage:
   1. LINQ methods generally use more memory as they create new sequences.
   2. Direct sort methods modify in – place, using less memory.

For performance – critical code with large datasets, in-place sorting is typically more efficient, while LINQ offers more flexibility and cleaner syntax for complex sorting scenarios.

**Conclusion:**

This lesson showed various methods to sort a List, ranging from the straightforward Sort method to more complex custom sorting logic. Understanding these sorting techniques is crucial for data manipulation and presentation in software development.

**Contains, Exists, Find, FindAll, and Any:**

* Contains: Check if the list contains specific element.
* Exists: Checks if the any element in the list matches a specified condition.
* Find: Finds the first element the matches a condition. If no match is found, it returns the default value for the type.
* FindAll: Retrieves all elements that matches a specific condition.
* Any: Checks if any of the elements in the list satisfy a given condition. It’s similar to Exists but is a LINQ method.

Code Example:

Inside the List solution -> BuiltInFunctions (Project).

**Conclusion:**

This lesson provides an overview of various methods for querying List. These methods are crucial for effective data manipulation and querying within collections.

**Working with a list of objects:**

**Code example:**

Inside the List solution -> CustomObjects (Project).

**Converting a List to an Array:**

This conversion is a common operation in programming, especially when you need to pass list data to a method that only accepts arrays of when interfacing with APIs or libraries that require array inputs.

**Code Example:**

Inside the List solution -> ListAndArray (Project).

**Converting Array to List:**

This operation is commonly needed when you’re working with APIs that return arrays, or when you need the dynamic features of a list after starting with a fixed-size array.

**Code Example:**

Inside the List solution -> ListAndArray (Project).

**Explanation:**

Using List constructor

One of the most straightforward method to convert an array to a list Is using the list’s constructor the accepts array.

**Exists vs Any:**

Both of them are methods used for collection manipulation, typically with lists or arrays. However, they are associated with different classes and serve slightly different purposes:

**Exists:**

* Is a method provided by Lists class.
* It takes a predicate delegate as an argument and returns a Boolean value indicating whether any element in the list satisfies the condition specified by the predicate.

**Example:**

List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

bool exists = numbers.Exists(n => n > 3); // Returns true because there is at least one element greater than 3

**Any:**

* It is a LINQ extension method available for any collection implementing the IEnumerable<T> interface, including arrays, lists, dictionaries, etc.
* It also takes a predicate delegate and returns a Boolean value indicating whether any element in the collection satisfies the condition specified by the predicate.

**Example:**

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

bool any = numbers.Any(n => n > 3); // Returns true because there is at least one element greater than 3

**key Differences:**

* Exists is specified to List and is available directly on instances of List.
* Any is a LINQ extension method available for any collection implementing IEnumerable<T>.
* Both method serve similar purposes, but Exists is more specialized for lists, while Any is more versatile and can be used with any enumerable collection.
* Exists is more efficient than Any for lists because it directly operates on the list without the overhead of LINQ . however, for collections other than lists, Any is often the only option.

**Hash Table**

**What is Hashtable:**

It is a data structure that you can use to store data in key-value format with direct access to its items in constant time.

Hash tables are said to be associative, which means that for each key, data occurs at most once. Hash tables let us implement things like phone books or dictionaries in them, we store the association between a value (like a dictionary definition of the word ‘chair’) and its key (the word ‘chair’ itself).

We can use hash tables to store, retrieve and delete data uniquely based on their unique key.

A **Hashtable**, also known as a hash map, isa data structure that implements an associative array abstract data type, a structure that can map keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. Ideally, the hash function will assign each key to a unique bucket, but most hash table designs assume that hash collisions (two keys that are different but have the same hash value) are inevitable and must be accommodated in some way.

**Here are the key features of hashtables:**

* Efficient Access: Hashtables provide very efficient average time complexity for insert, delete and search operations, ideally in O(1) time, which means the time to perform these operation is constant and does not grow with the size of the data.
* Dynamic Resizing: To maintain efficient operations and a good load factor (the ratio of the entries to the number of buckets), hashtables may dynamically resize. Thid involves creating a larger array and rehashing all existing entries into the new array.
* Use Cases: Hashtables are widely used in many computer applications, including database indexing, caching, symbol tables in compilers and implementing associative arrays in programming languages.

In summary, hashtables are powerful data structures for efficiently managing key-value pairs, allowing for quick data retrieval, addition and removal.

**Introduction:**

It is a collection that store key-value pairs, organized based on the hash code of the key. It resides in the System.Collections namespace and is designed for scenarios where quick searches, additions and deletions are crucial. Unlike generic collections, Hashtable allows for keys and values of any type, adding versatility but requiring careful handling of data types.

**Key Features:**

* Non-Generic: Operates on object of any type, requiring casting when retrieving elements.
* Efficient Lookups: Utilizes hash codes for keys, optimizing search operations.
* Uniqueness: Keys must be unique, though values may repeat.
* Order: Does not maintain a predictable order of stored elements.

**Conclusion:**

Hashtable is a powerful, if somewhat dated, collection type in C# that excels in scenarios requiring quick access to elements by key. While newer generic collection like Dictionary<TKey, TValue> offer type safety and potentially better performance, understanding how to use Hashtable is still valuable, especially for working with legacy code or APIs the require it.

**Working with Hashtable Basic Operations:**

To use a Hashtable, start by adding using System.Collections at the beginning of your file. Here's how to declare and populate it:

Hashtable myHashtable = new Hashtable();

myHashtable.Add("key1", "value1");

myHashtable.Add("key2", 100); // Mixed value types allowed

myHashtable.Add("key1", "value3");

**Accessing Elements**

Retrieve elements using their keys, remembering to cast the result:

string value = (string)myHashtable["key1"];

Console.WriteLine(value); // Expected: value1

**Modifying and Removing**

Change values directly or remove them with Remove:

myHashtable["key1"] = "newValue1"; // Update

myHashtable.Remove("key2"); // Delete

**Iteration**

Loop over the collection with foreach:

foreach (DictionaryEntry entry in myHashtable)

{

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

}

**Conclusion:**

Hashtable is a powerful, if somewhat dated, collection type that excels in scenarios requiring quick access to elements by key. While newer generic collection like Dictionary<TKey, TValue> offer type safety and potentially better performance, understanding how to use Hashtable is still valuable, especially for working with legacy code or APIs the require it.

**Dictionary**

It is a collection of key-value pairs that provides fast retrieval based on the key. It is part of the System.Collections.Generic namespace and is widely used in situations where quick lookups are necessary.

**Introduction to Dictionary:**

* Key – Value Pairs: Stores data as pairs of keys and values. Each key must be unique.
* Fast Lookups: Provide very efficient retrieval of values based on keys.
* Generic Collection: Allows specifying types for both keys and values.
* Dictionary is like a MAP in C++.

**Conclusion:**

It is a powerful and efficient collection for storing and retrieving data based on keys. It is essential in scenarios where quick data access and retrieval are critical.

**Dictionary vs Hashtable:**

In C#, both Dictionary and Hashtable are collection types used to store key-value pairs. However, they are designed to cater to different needs and scenarios based on their features and implementations. Understanding the differences between them is crucial for choosing the appropriate collection type for a given situation.

Dictionary:

Dictionary<TKey, TValue> is generic collection introduced in .NET 2.0. it resides in the System.Collections.Generics namespace and provides fast lookups to manage collections of keys and values. The key features of Dictionary include:

* Generic: Allows for type – safe data storage, ensuring that both keys and values are a specified type which helps to prevent runtime errors and eliminates the need for casting when retrieving values.
* Performance: Offers fast access to elements based on keys. The performance of searching for a key is close to O(1), making it highly efficient for lookups.
* Order: Doesn’t guarantee the order of elements. The order in which elements are returned during enumeration may not match the order in which they were inserted.
* Thread Safe: Not thread – safe. If multiple threads access it concurrently, you must implement you own synchronization mechanism.

**Hashtable:**

It is part of the System.Collections namespace, is a non-generic collection available since .NET 1.0. it can store keys and values of any types because it works with the object type. Key characteristics of Hashtable include:

* Non – Generic: Keys and values are of type object, which means they can store any data type. This flexibility comes at the cost of type safety, as it requires casting when retrieving values and increase the chance of runtime errors.
* Performance: It also provides fast access to elements. How ever the need for boxing and unboxing when working with values types can affect performance.
* Order: Doesn’t maintain the order of stored elements, similar to Dictionary.
* Thread Safety: Provides some thread safety features, such as synchronized (thread-safe) wrappers obtained through the Hashtable.Synchronized method. However, for full thread safety with multiple writers, external synchronization is recommended.

**Comparison Summary**

* **Type Safety**: Dictionary is strongly typed, whereas Hashtable requires casting for non-object types.
* **Performance**: Both provide fast lookups, but Dictionary can be more performant due to type safety and the lack of boxing/unboxing for value types.
* **Version Compatibility**: Hashtable is available from the first version of .NET, making it suitable for legacy applications. Dictionary was introduced later and is preferred for new development due to its generic nature.
* **Thread** **Safety**: Hashtable offers basic thread safety features, but neither collection is fully thread-safe for concurrent modifications without external synchronization.

**Choosing Between Dictionary and Hashtable:**

* Used Dictionary when you need strong type safety, better performance with value types, and are working with .NET 2.0 or later.
* Consider Hashtable if you are maintaining legacy code or need a collection that accepts key and values of any type without specifying their data types upfront.

Int modern .NET applications, Dictionary is generally preferred due to its type safety and performance advantages. However, understanding Hashtable is still valuable for working with existing codebases that use it.

**Working with Dictionary:**

**Code Example:**

Inside the HashTables solution -> Dictionaries (Project).

**TryGetValue Method:**

This method is essential for safely retrieving values from dictionary, avoiding exceptions that occur when trying to access key that may not exist.

**Understanding TryGetValue:**

It is a method designed to safely retrieve a value from a dictionary based on a key. It returns a Boolean indicating whether the key was found, and if so, assigns the corresponding value to an out parameter.

**Code Example:**

Inside the HashTables solution -> TryGetValueMethod (Project).

**Code Explanation:**

In the example, TryGetValue checks for “Apple” in the dictionary. If found, appleQuantity is set to the quantity of apples, and the method returns true. If not found, it returns false, and the else block is executed.

**The Advantages of TryGetValue:**

* Safety: Prevents exceptions if a key is not found, unlike directly accessing the value by key.
* Efficiency: Checks existence and retrieve the value in a single operation.
* Clarity: Makes the intent of safe retrieval clear, enhancing code readability.

**Conclusion:**understanding and using TryGetValue method is crucial for safely and efficiently working with dictionaries. This methos enhance error handling and code clarity, making it a best practice when retrieving values from dictionaries.

**Utilizing LINQ with Dictionaries:**

**Code Example:**

Inside the HashTables solution -> LinqWithDictionaries (Project).

**Conclusion:**

Using LINQ with dictionaries opens up world of possibilities for querying and manipulating data. It allows for concise and readable code, making operations like filtering, sorting, transforming and aggregation data straightforward and efficient.

**Advance LINQ with Dictionaries:**

This lesson will delve deeper into LINQ capabilities when working with dictionaries. We’ll focus on advanced operations such as grouping and combining various LINQ queries to perform more complex data manipulations.

**Introduction to Advanced LINQ with Dictionaries:**

LINQ provides advance capabilities that go beyond simple transformation, filtering and sorting. Operations like grouping can provide significant insights into the data structure.

**Code Example:**

Inside the HashTables solution -> AdvancedLinq (Project).

**Grouping Items with LINQ:**

GroupBy is a powerful method used to group items in a collection based on specified key.

**Combining LINQ queries:**

LINQ queries can be combined to perform filtering, sorting and transformation in a single statement.

**Conclusion:**

Advanced LINQ queries with dictionaries can significantly enhance data querying capabilities. Using operations like GroupBy and combining multiple queries allows for efficient and powerful data manipulation, crucial for complex data processing tasks.

**Hash Set**

**What is HashSet:**

* It is a collection class in the System.collections.Generic namespace designed to store unique elements.
* Uniqueness: The primary feature of HashSet<T> is that is automatically ensure all elements are unique.
* No Indexing: Unlike lists, HashSet<T> doesn’t maintain the order of its elements and does not support indexing.
* Generic: It is a generic collection meaning it can store any type of object.

**Conclusion:**

HashSet is a powerful collection for storing unique elements. It is particularly useful when you need to ensure no duplicates, perform set operations and when the order of elements is not concern.

Remember, HashSet does not support indexing, so if you need to access element by index, consider using other collections like lists.

**Working with HashSet:**

**Code Example:**

Inside the HashSet solution -> HashSetIntro (Project).

**Code Explanation:**

The duplicated records in the example will be ignored due to its duplication.

**Checking for Existence in HashSet:**

**Code Example:**

Inside the HashSet solution -> CheckForExistence (Project).

**Code Explanation:**

The Contains method takes the specified element and check its present in the HashSet and returns a Boolean expression.

**Removing Elements:**

**Code Example:**

Inside the HashSet solution -> RemoveItems (Project).

**Code Explanation:**

The Remove method removes the specified element, and the Clear method clears the HashSet completely.

**Using HashSet to Remove Duplicates:**

**Code Example:**

Inside the HashSet solution -> RemoveDuplicates (Project).

**Code Explanation:**

When initializing a HashSet and assign the array that contains the duplicate numbers the HashSet automatically takes the unique numbers and remove any duplicates (it works with any data type).

**Using HashSet with LINQ:**

**Code Example:**

Inside the HashSet solution -> HashSetAndLinq (Project).

Combining LINQ with HashSet enhances the capability of HashSet by providing more complex operations like filtering, searching and sorting if applicable.

**Conclusion:**

This lesson demonstrated the versatility of combining LINQ with HashSet. It shows how easily we can query an manipulate string data stored in a HashSet using LINQ methods.

The practical example illustrates the use of LINQ for handling string or any data type within the HashSet, providing a clear understanding for how to apply these techniques in real-world programming scenarios.

**Union Operations with HashSet:**

**Code Example:**

Inside the HashSet solution -> UnionOperation (Project).

**Code Explanation:**

The UnionWith combine only unique elements of two sets.

**Intersection Operation:**

**Code Example:**

Inside the HashSet solution -> UnionOperation (Project).

**Code Explanation:**

The IntersectWith method get only the common elements between two sets.

**Difference Operation:**

**Code Example:**

Inside the HashSet solution -> DifferenceOperation (Project).

**Code Explanation:**

The ExceptWith method removes the common elements between two sets from the corresponding set.

**Symmetric Difference Operation:**

**Code Example:**

Inside the HashSet solution -> DifferenceOperation (Project).

**Code Explanation:**

The SymmetricExceptWith removes the common elements and merges the two sets to the corresponding set.

**Using SetEquals:**

**Code Example:**

Inside the HashSet solution -> ComparingHashSets (Project).

**Objective:**

Determines if two sets are containing the same elements;

**Using IsSubSetOf:**

**Code Example:**

Inside the HashSet solution -> ComparingHashSets (Project).

**Objective:**

Determines if a set is subset (part of) another.

**Using IsSuperSetOf:**

**Code Example:**

Inside the HashSet solution -> ComparingHashSets (Project).

**Objective:**

Determines if a set is super set (contains the elements) of another.

And finally overlaps method determines if to sets overlaps or not.

**Sorted List**

It is a collection that stores key-value pairs, stored by the key. It is part of the System.Collections (non-generic) and System.Collections.Generic namespace.

Understanding how to use SortedList is important for scenarios where you need a dictionary-like collection with sorting by default.

**Characteristics of SortedList:**

* Automatically Sorted: The elements in a SortedList are sorted by the key as soon as they are added.
* Key-Value Pairs: Similar to a dictionary, it stores elements as key-value pairs.
* Unique Keys: Keys must be unique, and an exception in thrown if a duplicate key is added.
* Slower for addition and faster for search because it uses binary search algorithm.

**Conclusion:**

It is a useful collection for scenarios where automatic sorting of elements is required. Understanding when to use SortedList over other collections like Dictionary or List is crucial for efficient data management in your application.

**SortedList vs List:**

They are two different types of collections that serve different purposes and have different characteristics.

**List<T>:**

* Type: A generic collection that stores elements in a linear fashion.
* Ordering: The elements in a List are ordered based on how they are added or inserted. You can manually sort the list using the Sort() method.
* Performance: Adding elements to a List is fast, especially at the end. However, inserting or removing elements in the middle or beginning of the list can be slower because it may require shifting elements.
* Use Cases: Use List when you need simple, flexible collection to add, remove and access elements in no particular order, or when you control the order of elements manually.

**SortedList<TKey, TValue>:**

* Type: A generic collection that stores key-value pairs sorted by keys. It is a combination of an array and a HashTable.
* Ordering: The elements in a SortedList are automatically by the key. You can’t insert elements at a specific position as their position is determined by the key.
* Performance: Adding, removing and accessing elements can be fast if the collection in not large, as it uses binary search to find keys. However, the performance can degrade as the collection grows due to the cost of maintaining order.
* Use Case: Use SortedList when you need a collection of key-value pairs that must be sorted by key and you frequently need to search elements by key.

**Summary:**

* Purpose: List<T> is used for a simple list of items, whereas SortedList<TKey, TValue> is used for sorted key-value pairs.
* Ordering: List<T> maintains the order of elements as the are added, while SortedList<TKey, TValue> sorts elements by key.
* Performance: List<T> is generally faster for adding/removing at the end, SortedList<TKey, TValue> maintains sorted order, which can affect performance during addition/removals.
* Use Cases: List<T> for simplicity and when order is controlled manually or not important. Choose SortedList<TKey, TValue> when you need automatic sorting by keys and efficient key-based lookups.

Each collection type designed for specific scenarios, so the choice between List<T> and SortedList<TKey, TValue> depends on your specific requirement regarding ordering, performance and the nature of operations you’ll be performing on the collection.

**Code Example:**

Inside the SortedList (solution).

**LINQ with SortedList:**

**Code Example:**

Inside the SortedList (solution) -> LinqWithSortedList (project)

**Code Explanation:**

Section 1: Introduction to SortedList<TKey, TValue>

* Concept: It stores key-value pairs stored by the key, blending dictionary and array features.

Section 2: Initializing a SortedList.

* Enhanced code example: Populate s SortedList with keys and values, including comments for clarity.

Section 3: Querying SortedList using LINQ

* Query Expression Syntax: Use SQL-like syntax for filtering elements where keys are greater than 1, with comments explaining the process.
* Method Syntax: Achieve similar filtering using method syntax, illustrating its flexibility for chaining operations.

Section 4: Advanced Filtering Techniques:

* Specific Key Value Filtering: Demonstrates how to filter elements with keys greater than a specific value, showcasing LINQ’s power for complex queries.

**Conclusion:**

Through this enhanced lesson, you now understand how to use LINQ with SortedList<TKey, TValue>, including initializing the list, applying various LINQ queries, and interpreting the expected results. This knowledge enables you to manipulate and query sorted collections efficiently.

Armed with these capabilities, you’re well-equipped to handle complex data manipulation and querying tasks, enhancing data processing and analysis functionalities.

**Advance LINQ with SortedList:**

**Code Example:**

Inside the SortedList (solution) -> AdvancedLinq (project).

**Code Explanation:**

* Initializing: The SortedList is initialized with integer keys and string values, where each value is a fruit name. the list automatically sorts these fruits based on their integer keys.
* Grouping Operation: the LINQ GroupBy method is used to group fruit names by the length of the name. this showcases how to apply complex querying operations such as grouping on a SortedList, leveraging the GroupBy method to organize data based on a common characteristic.

**Conclusion:**

This lesson has demonstrated an advanced used case of LINQ with SortedList by focusing on the GroupBy operation to organize data based on a shared attribute. Through this example, we’ve seen how LINQ can extend the functionality of SortedList, allowing for sophisticated data manipulation and querying techniques such as grouping by the length of string values. This approach can be applied to various data processing scenarios, showcasing the versatility and power of LINQ in handling complex data structures and queries.

**Advance Complex Objects Operations:**

**Objective:**

This lesson demonstrates how to use LINQ to filter, sort, group, and select data from a SortedList containing complex objects. The operations include filtering and grouping employees by department, sorting them by salary in descending order, and selecting specific information. It demonstrates LINQ’s versatility in processing and querying complex data structures efficiently.

**Code Example:**

Inside the SortedList (solution) -> ComplexObjectsOperations (project).

**Sorted Set**

**Characteristics of SortedSet:**

1. Stores unique elements in sorted order.
2. Provides fast search, insertion, and removal operations.
3. Automatically maintains sorted order as elements are added or removed.
4. Does not allow duplicate elements.

**Advantages of Using SortedSet:**

1. Ensures elements are always stored in sorted order, facilitating efficient traversal and manipulation.
2. Provides fast lookup, insertion, and removal operations compared to other collection types.
3. Suitable for scenarios where maintaining sorted order and uniqueness of elements are essential.
4. Simplicity: the API of SortedSet is simpler compared to SortedList, as it deals with single elements rather than key-value pairs.

**Conclusion:**SortedSet is a useful collection for storing unique elements in sorted order. It provides efficient search, insertion and removal operations, making it suitable for a wide range of scenarios where maintaining sorted order is essential. By understanding the characteristics and advantages of SortedSet, developers can leverage it effectively in their programs.

**Introduction to Working with SortedSet:**

**Code Example:**

Inside the SortedSet (solution) ->SortedSetIntro (project).

**Code Explanation:**

* In the code example, we create a SortedSet of integers and add element to it.
* We demonstrate how to iterate through the elements of the SortedSet using foreach loop.
* We check if a specific element exists in the SortedSet using the Contains method.
* We remove an element from the SortedSet using the Remove method.

**LINQ with SortedSet Example:**

**Code Example:**

Inside the SortedList (solution) -> LinqWithSortedSet (project).

**Code Explanation:**

In this example, LINQ is used to filter elements greater than 2, calculate the sum of all elements, find the maximum and minimum elements, and sort the set in descending order.

And the other example, the Where LINQ method filters the SortedSet to find even numbers to its square.

**Union, Intersection, Difference, Subset, and Superset operations using SortedSet**

**Code Example:**

Inside the SortedList (solution) -> ManyOperationsOnSortedSet (project).

**Sorted Dictionary**

Sorted dictionary and sorted list are two commonly used data structures for maintaining a collection of key-value pairs sorted by keys. While they serve similar purpose, they have distinct characteristics that make them suitable for different scenarios.

**Sorted Dictionary:**

* It is a generic collection class in C# that represents a collection of key-value pairs sorted by keys.
* It is implemented as a binary search tree, which ensures that the keys are always sorted in ascending order.
* It offers efficient key-based operations like adding, removing and searching for elements.
* It provides O(log n) complexity for most operations, making it suitable for scenarios where efficient searching and insertion are required.

**Sorted List:**

* It is another generic collection class in C# that represents a collection of key-value pairs sorted by keys.
* It is implemented as an array of key-value pairs, sorted keys using an internal binary search algorithm.
* SortedList offers efficient indexed access to elements, similar to arrays, with O(log n) complexity for searching and insertion operations.
* However, it may incur overhead when elements are inserted or removed, as it may require shifting elements to maintain the sorted order.

**Difference Between SortedDictionary and SortedList**

* Implementation:
  + SortedDictionary: Implemented as a binary search tree.
  + SortedList: Implemented as array of key-value pairs.
* Performance Characteristics:
  + SortedDictionary offers efficient key-based operations with O(log n) complexity.
  + SortedList provides efficient indexed access with O(log n) complexity for searching and insertion but may incur overhead during insertion or removal.
* Memory Usage:
  + SortedDictionary typically consumes more memory due to its tree structure.
  + SortedList may have better memory efficiency, especially for large collections.

In terms of raw performance, the efficiency of SortedDictionary and SortedList can depend on the specific operations you perform and the size of the collection. Here is a breakdown:

* Insertion and Removal:
  + SortedList: Insertions and removals may require shifting elements in the underlying array to maintain the sorted order. This operation has a time complexity of O(n) in the worst-case scenario because it may involve copying elements.
  + SortedDictionary: Insertion and removals have a time complexity of O(log n) due to the underlying binary search tree structure. This makes SortedDictionary more efficient for these operations, especially for larger collections.
* Search:
  + Both data structures offer efficient search operations. SortedList uses binary search O(log n) for indexed access, while SortedDictionary also has O(log n) complexity for searching by key.
* Memory Usage:
  + SortedList: Typically consumes less memory compared to SortedDictionary because it uses an array structure to store elements.
  + SortedDictionary: May consume more memory due to the overhead of maintaining a binary search tree.
* Index-Based Access:
  + SortedList: Provides efficient indexed access similar to arrays with O(log n) complexity.
  + SortedDictionary: Does not support index-based access directly, you must access elements by their keys, which may involve searching.

Considering these factors, if your application involves frequent insertions and removals with a relatively small collections size or if memory efficiency is a concern, SortedList might be a better choice. On the other hand, if you require efficient search operations, especially with larger collections, or if memory usage is not primary concern, SortedDictionary could be more suitable.

Ultimately, the choice between the two of them should be based on your specific use case, considering factors like the size of the collection, the frequency of a different operations, and memory constraints.