Assign 04 : Advanced OOPs

Question 01 : C# Collection Types: In-Depth Overview

C# provides a variety of collection types, each designed for different use cases. Some collections are optimized for fast lookups, others for sorting, and some for ensuring uniqueness of elements. In this overview, we'll explore several important collection types, how they function internally, and the scenarios in which you might choose one over the other.

1. Hashtable (System.Collections)

   A Hashtable is a collection that stores key-value pairs. The key is unique, and the value is associated with the key. This collection is part of the non-generic System.Collections namespace.

   Mechanism of hashtable

   a. Buckets: A Hashtable uses an array of "buckets" to store the key-value pairs.
   b. Hashing: A hash function is applied to the key, and based on the hash value, the data is placed in the corresponding bucket.
   c. Collision Handling: If two keys hash to the same bucket, the Hashtable stores multiple key-value pairs in the same bucket (chaining).

   Example of hashtable

```
Hashtable hashtable = new Hashtable();
hashtable["apple"] = 1;
hashtable["banana"] = 2;
hashtable["cherry"] = 3;

Console.WriteLine(hashtable["banana"]); // Output: 2
```

   Key Takeaways:

   - Key-value pairs.
   - Average O(1) lookup time.
   - Unordered collection.
   - Non-generic: Can store any data type

   .

2. Dictionary<TKey, TValue> (System.Collections.Generic)

   The Dictionary<TKey, TValue> is a more modern, generic version of the Hashtable. It is part of the System.Collections.Generic namespace, providing better type safety and performance.

   Mechanism of Dictionary

   a. Hash Table: Like the Hashtable, it uses a hash table for fast lookups.
   b. Collision Resolution: Uses open addressing (probing) to resolve hash collisions.
   c. Type Safety: The key-value types are strongly typed (e.g., Dictionary<string, int>).

   Example

```
Dictionary<string, int> dictionary = new Dictionary<string, int>();
dictionary["apple"] = 10;
dictionary["banana"] = 20;
dictionary["cherry"] = 30;

Console.WriteLine(dictionary["banana"]); // Output: 20
```

   Key Takeaways:

   - Strongly typed (both keys and values).
   - Fast O(1) lookups.
   - Unordered collection.
   - More efficient than Hashtable due to improved memory and access patterns.

Assign 04 : Advanced OOPs

3. SortedDictionary<TKey, TValue> (System.Collections.Generic)
   The SortedDictionary<TKey, TValue> is a collection that behaves like a dictionary, but it maintains keys in sorted order.
   Mechanism of Sorted Dictionary:
   a. Red-Black Tree: Internally, it uses a balanced binary search tree (BST) to maintain order.
   b. O(log n) Operations: Both insertions and lookups take O(log n) time due to the tree structure.

   Example

```
SortedDictionary<string, int> sortedDict = new SortedDictionary<string, int>();
sortedDict["banana"] = 10;
sortedDict["apple"] = 20;
sortedDict["cherry"] = 30;

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

   Key Takeaways:
   - keys. O(log n) for lookups and insertions.
   - Useful when maintaining a sorted order is important.
   - Slower than Dictionary for lookups due to the use of a tree.

4. SortedList<TKey, TValue> (System.Collections.Generic)
   The SortedList<TKey, TValue> is similar to a SortedDictionary, but it uses an array-based implementation instead of a binary search tree.
   Mechanism of sorted List:
   a. Arrays for Keys and Values: Internally, the collection stores the keys and values in two separate arrays.
   b. Binary Search: When looking up a key, a binary search is performed, which takes O(log n) time.
   c. O(n) Insertions: Insertions and deletions are slower compared to a SortedDictionary due to the need to rearrange the arrays.

   Example

```
SortedList<string, int> sortedList = new SortedList<string, int>();
sortedList["banana"] = 10;
sortedList["apple"] = 20;
sortedList["cherry"] = 30;

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

   Key Takeaways:
   1. Sorted keys.
   2. O(log n) lookups.
   3. Slower insertions and deletions compared to SortedDictionary (O(n)).
   4. Array-based implementation.

5. HashSet<T> (System.Collections.Generic)
   The HashSet<T> is a collection designed to store unique elements. It does not allow duplicates, ensuring that only one occurrence of each element is stored.
   Mechanism of HashSet
   a. Hash Table: Internally, a hash table is used to store elements.
   b. Uniqueness: Duplicate elements are automatically ignored.

c. O(1) Operations: Operations like adding, removing, and checking for membership take constant time on average.

Example

```
HashSet<int> numbers = new HashSet<int>() { 1, 2, 3, 3, 4 };
numbers.Add(5);
numbers.Add(3); // Duplicate, ignored

foreach (int num in numbers)
{
    Console.WriteLine(num);  // Output: 1, 2, 3, 4, 5
}
```

Key Takeaways:

- Unique elements only.
- Fast O(1) operations.
- Unordered collection.

6. SortedSet<T> (System.Collections.Generic)

The SortedSet<T> is a variation of the HashSet<T> that keeps elements in sorted order.

Mechanism of SortedSet:

a. Binary Search Tree: Internally, a binary search tree (BST) is used to keep elements sorted.

b. O(log n) Insertions and Deletions: Insertions and deletions take O(log n) due to the tree structure.

c. Sorted Order: The collection maintains elements in ascending order

Example

```
SortedSet<int> numbers = new SortedSet<int>() { 5, 3, 1, 2, 4 };
numbers.Add(6);
numbers.Add(3); // Duplicate, ignored

foreach (int num in numbers)
{
    Console.WriteLine(num);  // Output: 1, 2, 3, 4, 5, 6
}
```

Key Takeaways:

- Unique elements sorted in ascending order.
- O(log n) insertions and deletions.
- Sorted collection, ideal when order matters.