Below is a detailed overview of the most commonly used collections in C# for Competitive Programming, along with all the methods associated with each.

### 1. List<T> (Dynamic Array)

List<T> is a dynamic array collection that allows adding, removing, and accessing elements by index.

#### Key Features:

* **Resizable**: Automatically resizes as elements are added or removed.
* **Zero-based index**: Access elements by index.

#### Commonly Used Methods:

| **Method** | **Description** |
| --- | --- |
| **Add(T item)** | Adds the item to the end of the list. |
| **AddRange(IEnumerable<T> collection)** | Adds the elements of the specified collection to the end of the list. |
| **Insert(int index, T item)** | Inserts an item into the list at the specified index. |
| **InsertRange(int index, IEnumerable<T> collection)** | Inserts the elements of the specified collection starting at the specified index. |
| **Remove(T item)** | Removes the first occurrence of a specific object from the list. |
| **RemoveAt(int index)** | Removes the element at the specified index. |
| **RemoveRange(int index, int count)** | Removes a range of elements from the list, starting from the specified index. |
| **Clear()** | Removes all elements from the list. |
| **Contains(T item)** | Returns true if the list contains the specified item. |
| **IndexOf(T item)** | Returns the index of the first occurrence of the item, or -1 if not found. |
| **LastIndexOf(T item)** | Returns the index of the last occurrence of the item, or -1 if not found. |
| **Count** | Gets the number of elements in the list. |
| **Sort()** | Sorts the list in ascending order. |
| **Reverse()** | Reverses the order of the elements in the list. |
| **BinarySearch(T item)** | Searches a sorted list for the specified object using binary search. |
| **ToArray()** | Copies the elements of the list to a new array. |
| **ForEach(Action<T> action)** | Performs the specified action on each element of the list. |
| **Find(Predicate<T> match)** | Searches for an element that matches the conditions defined by the specified predicate. |
| **FindAll(Predicate<T> match)** | Retrieves all elements that match the conditions defined by the specified predicate. |
| **FindIndex(Predicate<T> match)** | Returns the index of the first occurrence of an element that matches the conditions. |
| **FindLastIndex(Predicate<T> match)** | Returns the index of the last occurrence of an element that matches the conditions. |
| **Exists(Predicate<T> match)** | Determines if any elements match the conditions defined by the specified predicate. |
| **ConvertAll<TOutput>(Converter<T, TOutput> converter)** | Converts the elements to another type using the specified converter. |

#### Time Complexity:

* **Access by index**: O(1)
* **Add element (end)**: O(1) (Amortized)
* **Insert/remove element**: O(n)

### 2. Dictionary<TKey, TValue> (Hash Map)

Dictionary<TKey, TValue> is a collection of key-value pairs, where each key must be unique.

#### Key Features:

* **Key-based lookups**: Provides fast lookups by key.
* **Unique keys**: No duplicate keys are allowed.

#### Commonly Used Methods:

| **Method** | **Description** |
| --- | --- |
| **Add(TKey key, TValue value)** | Adds the specified key and value to the dictionary. |
| **Remove(TKey key)** | Removes the value with the specified key from the dictionary. |
| **ContainsKey(TKey key)** | Returns true if the dictionary contains the specified key. |
| **ContainsValue(TValue value)** | Returns true if the dictionary contains the specified value. |
| **TryGetValue(TKey key, out TValue value)** | Attempts to get the value associated with the specified key. Returns false if the key is not found. |
| **Keys** | Returns a collection of keys in the dictionary. |
| **Values** | Returns a collection of values in the dictionary. |
| **Clear()** | Removes all keys and values from the dictionary. |
| **Count** | Gets the number of key-value pairs in the dictionary. |
| **Indexer** | Gets or sets the value associated with the specified key (dict[key] = value). |

#### Time Complexity:

* **Access by key**: O(1) on average.
* **Insertion/removal**: O(1) on average.

### 3. HashSet<T> (Set Collection)

HashSet<T> stores a collection of unique elements and provides fast lookups.

#### Key Features:

* **Unique elements**: No duplicates are allowed.
* **Unordered**: Elements are not stored in a specific order.

#### Commonly Used Methods:

| **Method** | **Description** | |
| --- | --- | --- |
| **Add(T item)** | | Adds the specified item to the set. Returns true if the item was added, false if it already exists. |
| **Remove(T item)** | | Removes the specified item from the set. Returns true if the item was successfully removed. |
| **Contains(T item)** | | Returns true if the set contains the specified item. |
| **Clear()** | | Removes all elements from the set. |
| **Count** | | Gets the number of elements in the set. |
| **UnionWith(IEnumerable<T> other)** | | Adds all elements of the specified collection to the current set. |
| **IntersectWith(IEnumerable<T> other)** | | Modifies the current set so that it contains only elements that are also in the specified collection. |
| **ExceptWith(IEnumerable<T> other)** | | Removes all elements in the current set that are also in the specified collection. |
| **IsSubsetOf(IEnumerable<T> other)** | | Returns true if the current set is a subset of the specified collection. |
| **IsSupersetOf(IEnumerable<T> other)** | | Returns true if the current set is a superset of the specified collection. |

#### Time Complexity:

* **Add/remove/check element**: O(1) on average.

### 4. Queue<T> (FIFO - First In First Out)

Queue<T> is used to store elements in a first-in-first-out order.

#### Commonly Used Methods:

| **Method** | **Description** |
| --- | --- |
| **Enqueue(T item)** | Adds an item to the end of the queue. |
| **Dequeue()** | Removes and returns the item at the front of the queue. |
| **Peek()** | Returns the item at the front of the queue without removing it. |
| **Contains(T item)** | Returns true if the queue contains the specified item. |
| **Clear()** | Removes all elements from the queue. |
| **Count** | Gets the number of elements in the queue. |

#### Time Complexity:

* **Enqueue/Dequeue**: O(1)

### 5. Stack<T> (LIFO - Last In First Out)

Stack<T> is used to store elements in a last-in-first-out order.

#### Commonly Used Methods:

| **Method** | **Description** |
| --- | --- |
| **Push(T item)** | Adds an item to the top of the stack. |
| **Pop()** | Removes and returns the item at the top of the stack. |
| **Peek()** | Returns the item at the top of the stack without removing it. |
| **Contains(T item)** | Returns true if the stack contains the specified item. |
| **Clear()** | Removes all elements from the stack. |
| **Count** | Gets the number of elements in the stack. |

#### Time Complexity:

* **Push/Pop**: O(1)

### Summary of Most Used Collections:

* **List<T>: Used for dynamic arrays when the size is unknown or changes.**
* **Dictionary<TKey, TValue>: Key-value pairs for fast lookups.**
* **HashSet<T>: Collection of unique items.**
* **Queue<T>: First-in-first-out operations.**
* **Stack<T>: Last-in-first-out operations.**

### 1. List<T> (Dynamic Array)

#### Best Suited For:

* **Random access to elements** when you know the index.
* **Ordered collections** where elements may need to be accessed by their index.
* **Frequent appending** to the end of the collection.

#### Operations:

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| **Access by index** | **O(1)** | **Direct indexing like an array.** |
| **Add (to end)** | **O(1) (Amortized)** | **Amortized O(1) as resizing may occur occasionally.** |
| **Insert (at index)** | **O(n)** | **Inserts at the start or middle requires shifting elements, increasing time.** |
| **Remove by index** | **O(n)** | **Shifting required after removal.** |
| **Search (Contains)** | **O(n)** | **Linear search is required.** |
| **Space Complexity** | **O(n)** | **Storage of elements, plus resizing overhead.** |

#### When to Use:

* When **random access** is needed (e.g., accessing an element at index i).
* When **appending** elements at the end frequently.
* Not ideal for frequent **insertions** and **deletions** at the start or middle.

### 2. Dictionary<TKey, TValue> (Hash Map)

#### Best Suited For:

* **Key-based lookups** for fast retrieval.
* **Mapping relationships** between key-value pairs (e.g., counting frequencies).

#### Operations:

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| **Insert/Update by key** | **O(1) (Amortized)** | **Hashing allows for constant-time insertions/updates.** |
| **Access by key** | **O(1) (Amortized)** | **Direct access through the key using hashing.** |
| **Remove by key** | **O(1) (Amortized)** | **Constant-time removal by key.** |
| **Search (ContainsKey)** | **O(1)** | **Constant time search for key existence.** |
| **Iteration over values** | **O(n)** | **Iterating over keys or values takes linear time.** |
| **Space Complexity** | **O(n)** | **Storage for keys and values, plus additional overhead for hashing and handling collisions.** |

#### When to Use:

* When you need **fast lookups** by a key.
* Ideal for **caching** or **frequency counting**.
* **Not ordered: Use SortedDictionary for sorted keys.**

### 3. HashSet<T> (Set)

#### Best Suited For:

* **Unique collections of elements with fast lookups.**
* **Checking set membership or performing set operations like union, intersection, etc.**

#### Operations:

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| **Insert** | **O(1)** | **Fast insertion if no collisions occur in hashing.** |
| **Remove** | **O(1)** | **Constant-time removal using the element’s hash.** |
| **Contains** | **O(1)** | **Check for membership in constant time.** |
| **UnionWith/IntersectWith** | **O(n)** | **Set operations like union and intersection work in linear time.** |
| **Space Complexity** | **O(n)** | **Space for unique elements, plus hashing overhead.** |

#### When to Use:

* When you want **unique elements** and need to quickly check whether an element is in the set.
* Use for problems involving **membership tests** (e.g., checking if a number has already been processed).
* **Not ideal for** **ordered collections**; use **SortedSet** if ordering is needed.

### 4. Queue<T> (FIFO - First In First Out)

#### Best Suited For:

* **FIFO (First-In-First-Out)** operations, such as processing items in the order they were added.
* Scenarios like **BFS (Breadth-First Search)** or task scheduling.

#### Operations:

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| **Enqueue (add)** | **O(1)** | **Fast insertion at the end of the queue.** |
| **Dequeue (remove)** | **O(1)** | **Fast removal from the front of the queue.** |
| **Peek (view front)** | **O(1)** | **Constant time access to the front element.** |
| **Contains** | **O(n)** | **Searching the entire queue requires linear time.** |
| **Space Complexity** | **O(n)** | **Space for the elements in the queue.** |

#### When to Use:

* Ideal for **BFS** in graph algorithms or **task scheduling**.
* Use when you need to process items in the exact order they were added.

### 5. Stack<T> (LIFO - Last In First Out)

#### Best Suited For:

* **LIFO (Last-In-First-Out)** operations like **DFS (Depth-First Search)**.
* Scenarios like **backtracking**, expression evaluation, or **recursive** **algorithms**.

#### Operations:

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| **Push (add)** | **O(1)** | **Fast insertion at the top of the stack.** |
| **Pop (remove)** | **O(1)** | **Fast removal from the top of the stack.** |
| **Peek (view top)** | **O(1)** | **Constant time access to the top element.** |
| **Contains** | **O(n)** | **Searching the entire stack requires linear time.** |
| **Space Complexity** | **O(n)** | **Space for the elements in the stack.** |

#### When to Use:

* Ideal for **DFS** or when you need **LIFO** behavior (e.g., undo operations).
* Efficient for backtracking algorithms and evaluating expressions in compilers.

### 6. SortedDictionary<TKey, TValue> (Ordered Dictionary)

#### Best Suited For:

* Key-value pairs where the **keys need to be sorted**.
* **Range queries** or any scenario where the elements must remain in order.

#### Operations:

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| **Insert** | **O(log n)** | **Logarithmic insertion due to maintaining the order of the keys.** |
| **Access by key** | **O(log n)** | **Logarithmic access due to sorted order.** |
| **Remove** | **O(log n)** | **Logarithmic removal.** |
| **ContainsKey** | **O(log n)** | **Logarithmic search for keys.** |
| **Space Complexity** | **O(n)** | **Space for keys and values, plus overhead for maintaining the order.** |

#### When to Use:

* **Use when you need a sorted collection of key-value pairs.**
* **Ideal for range queries and scenarios where data must stay ordered (e.g., a leaderboard).**

### 7. SortedSet<T> (Sorted Set)

#### Best Suited For:

* A collection of **unique elements** that must be sorted.
* Efficient for scenarios that involve range queries or ordered traversals.

#### Operations:

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| **Add** | **O(log n)** | **Logarithmic insertion to maintain the sorted order of elements.** |
| **Remove** | **O(log n)** | **Logarithmic removal.** |
| **Contains** | **O(log n)** | **Logarithmic search due to sorted order.** |
| **Space Complexity** | **O(n)** | **Space for unique elements, plus additional overhead for maintaining the sorted order.** |

#### When to Use:

* Use when you need a **unique, sorted collection** (e.g., for maintaining a sorted set of unique numbers).
* Ideal for problems involving **range queries** or ordered iteration.

### Comparison of Collections for Competitive Programming:

| **Collection** | **Best for** | **Time Complexity (Key Operations)** | **Space Complexity** |
| --- | --- | --- | --- |
| **List<T>** | **Random access, appending** | **Access O(1), Insert/Remove O(n)** | **O(n)** |
| **Dictionary<TKey, TValue>** | **Fast lookups by key** | **Insert/Access O(1)** | **O(n)** |
| **HashSet<T>** | **Unique elements, fast membership check** | **Insert/Contains O(1)** | **O(n)** |
| **Queue<T>** | **FIFO operations, BFS** | **Enqueue/Dequeue O(1)** | **O(n)** |
| **Stack<T>** | **LIFO operations, DFS** | **Push/Pop O(1)** | **O(n)** |
| **SortedDictionary<TKey, TValue>** | **Ordered key-value pairs, range queries** | **Insert/Access O(log n)** | **O(n)** |
| **SortedSet<T>** | **Unique, ordered elements** | **Add/Contains O(log n)** | **O(n)** |

### Conclusion:

* For **fast random access**, use List<T>.
* For **fast lookups by key**, use Dictionary<TKey, TValue>.
* For **unique elements**, use HashSet<T> (unordered) or SortedSet<T> (ordered).
* For **FIFO** operations, use Queue<T>.
* For **LIFO** operations, use Stack<T>.
* For **sorted collections**, use SortedDictionary<TKey, TValue> or SortedSet<T> depending on the need for key-value mapping or unique elements.

4o