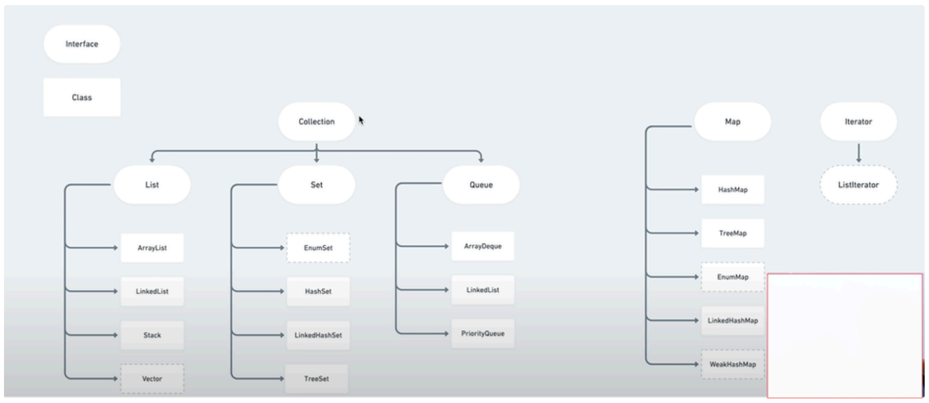
Collection Framework



Collection framework

Collection:-

Above the Fig, the collection is the interface on the top of the hierarchy:- The root interface in the collection hierarchy. A collection represents a group of objects known as its elements. Some collections allow duplicate elements, and others do not. Some are ordered, and others are unordered

List:-

List is also an interface.

ArrayList:-

1. ArrayList implements the List interface.
2. It provides a resizable array, which can grow as needed when elements are added. Unlike arrays, which have a fixed size, an `ArrayList` automatically adjusts its capacity when elements are added or removed.
3. **Dynamic Array**:
   - `ArrayList` is essentially a dynamic array. Unlike a regular array, which has a fixed size, an `ArrayList` can grow or shrink as elements are added or removed.
4. **Index-Based Access**:
   - `ArrayList` allows for fast, index-based access to elements. You can retrieve, update, or remove elements by their index in constant time, O(1).
5. **Insertion Order**:
   - Elements in an `ArrayList` are maintained in the order they were inserted. This means that when you iterate over an `ArrayList`, you will see the elements in the same order they were added.
6. **Resizable**:
   - When you add elements beyond the current capacity of the `ArrayList`, it automatically resizes itself. This resizing typically involves creating a new array with a larger capacity and copying the old elements to the new array.
7. **Null Values**:
   - `ArrayList` can store null elements.
8. **Non-Synchronized**:By default, `ArrayList` is not synchronized, meaning it is not thread-safe. If multiple threads access an `ArrayList` concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally.
9. Methods:-
   add(element) --add at last Complexity--> `get(int index)` : O(1)
   `add(int index, E element)` : O(n)
   `set(int index, E element)` : O(1)
   `indexOf(Object o)` : O(n)
   `remove(int index)` : O(n)
   remove(Integer.valueOf(element))-- This method accepts the element in object form.O(n)
   `lastIndexOf(Object o)` : O(n) → Returns the index of the last occurrence of the specified element in the list. This also involves a linear search, so the complexity is O(n).
   `contains(Object o)` : O(n)
   `size()` : O(1)
   `isEmpty()` : O(1)
   `trimToSize()` : O(n)
   `ensureCapacity(int minCapacity)` : O(n)
   `removeAll(Collection<?> c)` : O(n * m)
   `clear()` : O(n)

   addAll(list)-- suppose you have newlist you need to add all the list elements to newlist then you will call newlist.addAll(list);
10. Time Complexity
    - **Access by Index**: O(1)
    - **Add at End**: O(1) amortized

- **Remove by Index**: O(n)
- **Search (contains)**: O(n)
- **Insertion/Deletion at arbitrary positions**: O(n)
- set operation = 0(1)

11. When to Use `ArrayList`
    - **Random Access**: If you need fast random access to elements (i.e., frequent get operations).
    - **Size Variability**: If the number of elements is expected to grow or shrink frequently.
    - **Order Preservation**: When you need to maintain the order of elements.

    `ArrayList` is a versatile and commonly used collection class in Java that provides dynamic array functionality with easy-to-use methods for adding, removing, and accessing elements.

Stack:-

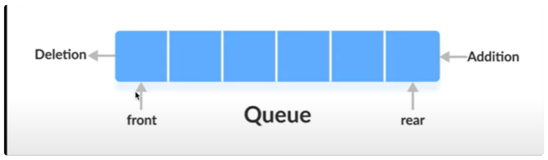
1. class represents a last-in, first-out (LIFO) stack of objects.

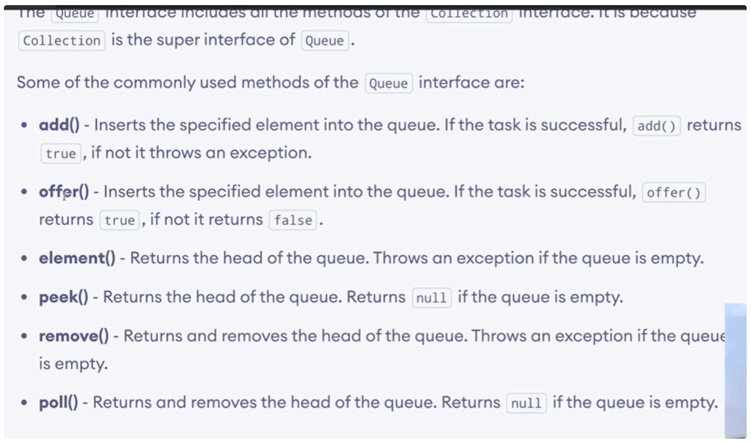
Time Complexity of Stack Operations

1. `push(E item)` : O(1) push on the top
    - Adding an element to the top of the stack is a constant-time operation.
2. `pop()` : O(1) :- remove top element
    - Removing the top element from the stack is also a constant-time operation.
3. `peek()` : O(1): will give top
    - Viewing the top element without removing it is a constant-time operation.
4. `empty()` : O(1)
    - Checking whether the stack is empty is a constant-time operation.
5. `search(Object o)` : O(n)
    - Searching for an element in the stack involves scanning through the elements, making the time complexity O(n), where n is the number of elements in the stack.

Queue:-

It follows the First-In-First-Out (FIFO) principle, where elements are added at the end of the queue and removed from the front.




The `Queue` interface includes all the methods of the `Collection` interface. It is because `Collection` is the super interface of `Queue`.

Some of the commonly used methods of the `Queue` interface are:

- **add()** - Inserts the specified element into the queue. If the task is successful, `add()` returns `true`, if not it throws an exception.
- **offer()** - Inserts the specified element into the queue. If the task is successful, `offer()` returns `true`, if not it returns `false`.
- **element()** - Returns the head of the queue. Throws an exception if the queue is empty.
- **peek()** - Returns the head of the queue. Returns `null` if the queue is empty.
- **remove()** - Returns and removes the head of the queue. Throws an exception if the queue is empty.
- **poll()** - Returns and removes the head of the queue. Returns `null` if the queue is empty.

Time Complexity Summary for Common `Queue` Implementations:

- `add(E e)` / `offer(E e)` : O(1) for `LinkedList` and `ArrayDeque` ; O(log n) for `PriorityQueue` .
- `remove()` / `poll()` : O(1) for `LinkedList` and `ArrayDeque` ; O(log n) for `PriorityQueue` .
- `peek()` / `element()` : O(1) for all.

PriorityQueue:

It is a very important data structure for implementing the min heap and max heap.

`PriorityQueue` is a part of the Java Collections Framework and is used to hold elements in a priority-based order. Unlike a regular `Queue,` which operates in a FIFO (First-In-First-Out) manner, a `PriorityQueue` orders its elements based on their natural ordering or according to a `Comparator` provided at the queue's construction time.

Common Operations:

1. `add(E e)` / `offer(E e)` :
   - Inserts the specified element into the priority queue.
   - The time complexity is **O(log n)**.
2. `poll()` :
   - Retrieves and removes the head of the queue, i.e., the element with the highest priority.
   - The time complexity is **O(log n)**.
3. `peek()` :
   - Retrieves, but does not remove, the head of the queue.
   - The time complexity is **O(1)**.
4. `remove(Object o)` :
   - Removes a single instance of the specified element from the queue.
   - The time complexity is **O(n)** because it may require a linear search.
5. `size()` :
   - Returns the number of elements in the queue.
   - The time complexity is **O(1)**.
6. `isEmpty()` :
   - Returns `true` if the queue contains no elements.
   - The time complexity is **O(1)**.

Use Cases:

- **Task Scheduling**: Assign priorities to tasks and always process the highest-priority task first.
- **Dijkstra's Algorithm**: Often used to implement the priority queue in Dijkstra's shortest path algorithm.
- **Huffman Coding**: To build a Huffman tree by always combining the two least frequent elements.

Time Complexity Summary:

- **Insertion**: O(log n)
- **Deletion of Head**: O(log n)
- **Peek**: O(1)
- **Iteration**: O(n), but order is not guaranteed

ArrayDeque:-

`ArrayDeque` is a resizable array implementation of the `Deque` interface in Java. It stands for "Array Double-Ended Queue" and is part of the Java Collections Framework. This data structure allows elements to be added or removed from both ends (head and tail) efficiently.

It can be used in the sliding window technique, where it can be used to check the tail and head.

Key Characteristics:-

- **No Capacity Limit**:
  - Unlike `ArrayBlockingQueue`, `ArrayDeque` has no fixed capacity. It can grow dynamically as needed.
- **Null Elements**:
  - `ArrayDeque` does not allow `null` elements. This is to avoid ambiguity in cases where `null` might be used to signal the end of the deque.
- **Memory Efficiency**:
  - `ArrayDeque` is more memory-efficient than `LinkedList` because it does not need to store pointers for the next and previous elements.

Common Operations and Time Complexity:

1. **Adding Elements**:
   - `addFirst(E e)` / `offerFirst(E e)` : Inserts the element at the front of the deque.
   - `addLast(E e)` / `offerLast(E e)` : Inserts the element at the end of the deque.
   - **Time Complexity**: O(1) for both operations.
2. **Removing Elements**:
   - `removeFirst()` / `pollFirst()` : Removes and returns the element at the front.
   - `removeLast()` / `pollLast()` : Removes and returns the element at the end.
   - **Time Complexity**: O(1) for both operations.
3. **Accessing Elements**:
   - `getFirst()` / `peekFirst()` : Retrieves, but does not remove, the element at the front.
   - `getLast()` / `peekLast()` : Retrieves, but does not remove, the element at the end.
   - **Time Complexity**: O(1) for both operations.

4. **Stack Operations**:
   - `push(E e)`: Equivalent to `addFirst(E e)`; pushes an element onto the stack represented by the deque.
   - `pop()`: Equivalent to `removeFirst()`; pops an element from the stack represented by the deque.
   - **Time Complexity**: O(1) for both operations.
5. **Iteration**:
   - Iterating through an `ArrayDeque` is O(n), where n is the number of elements in the deque.
6. **Size and Emptiness**:
   - `size()`: Returns the number of elements in the deque. O(1).
   - `isEmpty()`: Checks if the deque is empty. O(1).
7. **ArrayDeque vs. LinkedList**:
   - `ArrayDeque` is generally faster than `LinkedList` when used as a queue or stack, because `LinkedList` involves extra memory overhead for storing links and can be slower due to the need to traverse nodes.
8. **ArrayDeque vs. Stack**:
   - `ArrayDeque` is preferred over `Stack` for stack operations because `Stack` is synchronized and has higher overhead.
9. **ArrayDeque vs. ArrayList**:
   - While both are backed by arrays, `ArrayDeque` is more efficient for operations at both ends of the deque, whereas `ArrayList` is optimized for random access and operations primarily at the end.

Limitations:

- **Not Thread-Safe**: `ArrayDeque` is not thread-safe, so if you need a thread-safe deque, consider using `ConcurrentLinkedDeque` or wrapping `ArrayDeque` with `Collections.synchronizedDeque`.

Set:-

Java, a `Set` is a collection that contains no duplicate elements. It is part of the Java Collections Framework and is defined by the `Set` interface. The `Set` interface extends the `Collection` interface and is implemented by various classes like `HashSet`, `LinkedHashSet`, and `TreeSet`

HashSet:-

- **No Duplicates**:
- **Unordered Collection**:
- **Backed by a Hash Table**:
  - `HashSet` is implemented using a hash table. This allows for efficient operations like add, remove, and contains, which typically have an average time complexity of O(1).
- **Allows One**
- **Not Thread-Safe**:

Common Operations and Time Complexity:

1. **Adding an Element**:
   - `add(E e)`: Adds the specified element to the set if it is not already present.
   - **Time Complexity**: O(1) on average; O(n) in the worst case due to hash collisions.
2. **Removing an Element**:
   - `remove(Object o)`: Removes the specified element from the set if it is present.
   - **Time Complexity**: O(1) on average; O(n) in the worst case due to hash collisions.
3. **Checking if an Element Exists**:
   - `contains(Object o)`: Returns `true` if the set contains the specified element.
   - **Time Complexity**: O(1) on average; O(n) in the worst case due to hash collisions.
4. **Iterating Over Elements**:
   - `iterator()`: Returns an iterator over the elements in the set.
   - **Time Complexity**: O(n), where n is the number of elements in the set.
5. **Size of the Set**:
   - `size()`: Returns the number of elements in the set.
   - **Time Complexity**: O(1).
6. **Clearing the Set**:
   - `clear()`: Removes all elements from the set.
   - **Time Complexity**: O(1).

LinkedHashSet

This means it maintains the uniqueness of elements while also preserving their insertion order.

Everything else is similar to HashSet.

TreeSet:-

Binary Search tree + Set.

It is a sorted set; all other functions and properties are the same as HashSet and LinkedHashset.

Time complexity is greater than that of Hashset and linkedHashSetO(1). Here, time complexity, mostly for all operations, is long.

Comparison with Other Collections:

- `TreeSet` vs `HashSet`:
  - `TreeSet` maintains elements in sorted order, whereas `HashSet` does not maintain any order.
  - `TreeSet` has a higher time complexity (O(log n)) for basic operations compared to `HashSet`'s O(1) average time complexity.

Map:

Map Is not a part of collection API, `Map` is an interface in the Java Collections Framework that represents a collection of key-value pairs.

HashMap

Common Methods in `HashMap`:

1. `put(K key, V value)`:
   - Associates the specified value with the specified key. If the key already exists, the old value is replaced.
   - **Time Complexity**: O(1) average, O(n) in the worst case due to hash collisions.
2. `get(Object key)`:
   - Returns the value associated with the specified key or `null` if the key does not exist.
   - **Time Complexity**: O(1) average, O(n) in the worst case.
3. `remove(Object key)`:
   - Removes the key-value pair for the specified key.
   - **Time Complexity**: O(1) average, O(n) in the worst case.
4. `containsKey(Object key)`:
   - Returns `true` if the `HashMap` contains a mapping for the specified key.
   - **Time Complexity**: O(1) average, O(n) in the worst case.
5. `containsValue(Object value)`:
   - Returns `true` if one or more keys map to the specified value.
   - **Time Complexity**: O(n).
6. `size()`:
   - Returns the number of key-value pairs in the `HashMap`.
   - **Time Complexity**: O(1).
7. `keySet()`:
   - Returns a `Set` view of the keys contained in the `HashMap`.
   - **Time Complexity**: O(1).
8. `values()`:
   - Returns a `Collection` view of the values contained in the `HashMap`.
   - **Time Complexity**: O(1).
9. `entrySet()`:
   - Returns a `Set` view of the key-value pairs (entries) contained in the `HashMap`.
   - **Time Complexity**: O(1)

Time Complexity of Common Operations:

- **Insertion (`put`)**: O(1) average, O(n) worst-case (due to collisions).
- **Lookup (`get`)**: O(1) average, O(n) worst-case (due to collisions).
- **Deletion (`remove`)**: O(1) average, O(n) worst-case (due to collisions).
- **Iteration**: O(n) for iterating through all entries.