Time Complexity

Usually, **when we talk about time complexity, we refer to Big-O notation**. Simply put, the notation describes how the time to perform the algorithm grows with the size of the input.

Let us see the time complexity with respect to each collection framework we have seen

1. ArrayList

So, let's first focus on the time complexity of the common operations, at a high level:

- **add()** – takes O(1) time
- **add(index, element)** – in average runs in O(n) time
- **get()** – is always a constant time O(1) operation
- **remove()** – runs in linear O(n) time. We have to iterate the entire array to find the element qualifying for removal
- **indexOf()** – also runs in linear time. It iterates through the internal array and checking each element one by one. So the time complexity for this operation always requires O(n) time
- **contains()** – implementation is based on indexOf(). So it will also run in O(n) time

2. LinkedList

LinkedList is a linear data structure which consists of nodes holding a data field and a reference to another node.

Let's present the average estimate of the time we need to perform some basic operations:

- **add()** – supports O(1) constant-time insertion at any position
- **get()** – searching for an element takes O(n) time

- **remove()** – removing an element also takes O(1) operation, as we provide the position of the element
- **contains()** – also has O(n) time complexity

3. PriorityQueue

In Java programming, Java Priority Queue is implemented using Heap Data Structures and Heap

- O(log(n)) time complexity to **insert and delete** element.
- Offer() and add() methods are used to insert the element in the in the priority queue java program.
- Poll() and remove() is used to delete the element from the queue.
- Element retrieval methods i.e. **peek() and element(),** that are used to retrieve elements from the head of the queue is constant time i.e. O(1).
- **contains(Object)** method that is used to check if a particular element is present in the queue, have leaner time complexity i.e. O(n).
- Time complexity for the methods offer & **poll** is O(log(n)) and for the **peek()** it is Constant time O(1) of java priority queue.

4. TreeSet

Let's present the average estimate of the time we need to perform some basic operations:

- **add() –** O(log n) with a base 2 is the time complexity to ass an element.
- **contains()-**O(log n) with a base 2 is the time complexity to search for an element.
- **next()-** O(log n) with base 2.

In collections we basically have List, Set, Queue, Map. Below is the table that represents the time complexity for each of them.

List: **A list is an ordered collection of elements.**

	Add	Remove	Get	Contains	Data Structure
ArrayList	O(1)	O(n)	O(1)	O(n)	Array
LinkedList	O(1)	O(1)	O(n)	O(n)	Linked List
CopyonWriteArrayList	O(n)	O(n)	O(1)	O(n)	Array

Set: **A collection that contains no duplicate elements.**

	Add	Contains	Next	Data Structure
HashSet	O(1)	O(1)	O(h/n)	Hash Table
LinkedHashSet	O(1)	O(1)	O(1)	Hash Table + Linked List
EnumSet	O(1)	O(1)	O(1)	Bit Vector
TreeSet	O(log n)	O(log n)	O(log n)	Red-black tree
CopyonWriteArraySet	O(n)	O(n)	O(1)	Array
ConcurrentSkipList	O(log n)	O(log n)	O(1)	Skip List

Queue: **A collection designed for holding elements prior to processing**

	Offer	Peak	Poll	Size	Data Structure
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
LinkedList	O(1)	O(1)	O(1)	O(1)	Array
ArrayDequeue	O(1)	O(1)	O(1)	O(1)	Linked List
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)	Linked List
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)	Array
PriorirityBlockingQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
SynchronousQueue	O(1)	O(1)	O(1)	O(1)	None!
DelayQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)	Linked List

Map: **An object that makes keys to values. A map cannot duplicate heys, each key can map to at most one value.**

	Get	ContainsKey	Next	Data Structure
HashMap	O(1)	O(1)	O(h / n)	Hash Table
LinkedHashMap	O(1)	O(1)	O(1)	Hash Table + Linked List
IdentityHashMap	O(1)	O(1)	O(h / n)	Array
WeakHashMap	O(1)	O(1)	O(h / n)	Hash Table
EnumMap	O(1)	O(1)	O(1)	Array
TreeMap	O(log n)	O(log n)	O(log n)	Red-black tree
ConcurrentHashMap	O(1)	O(1)	O(h / n)	Hash Tables
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	Skip List