Time & Space Complexity

Time Complexity:

Measures the execution time of an algorithm with respect to the input size.

1. **O(1) – Constant Time:** The algorithm’s execution time doesn’t depend on the input size. The operation takes the same amount of time regardless of input size.
2. **O(log n) - Logarithmic Time:** These complexities usually apply to algorithms that divide problems in half every time.
3. **O(n) – Linear Time:** Execution time grows linearly with the input size (e.g., iterating through an array).
4. **O(n^2) – Quadratic Time:** Execution time grows quadratically (e.g., nested loops).

Space Complexity:

Space complexity measures **the amount of memory an algorithm uses** relative to the input size.

1. **O(1) - Constant space:** The algorithm uses a fixed amount of memory regardless of input size.
2. **O(n) - Linear space:** Memory usage grows linearly with the input size.

A screenshot of a computer

Description automatically generated

A close-up of a chart

Description automatically generated

LIST FAMILY

A screenshot of a computer

Description automatically generated

**Insertion:**

**ArrayList and Vector:** O(1) (amortized) when adding to the end, but O(n) when resizing is needed or inserting in the middle.

**LinkedList and Stack:** Always O(1) when inserting at the head or tail.

**Search:**

**For all list collections, searching is linear (O(n)) as they don't use hashing or binary search by default.**

**Deletion:**

**LinkedList:** O(1) for deleting the first or last element since pointers are adjusted.

**ArrayList and Vector:** O(n) due to shifting elements after deletion.

**Indexing:**

**ArrayList and Vector:** O(1) because of direct array access.

**LinkedList and Stack:** O(n) because traversal is required.

Tip: Use ArrayList for frequent random access and LinkedList for frequent insertions/deletions.

Summary:

* For most use cases**, ArrayList** is excellent due to its performance for **random access and iteration**.
* Use **LinkedList** for frequent **insertions/deletions** at the **start/middle.**
* Avoid Vector unless you need legacy thread-safety.
* Prefer **Deque (e.g., ArrayDeque) over Stack for LIFO requirements**.

If unsure, start with ArrayList, as it is the most commonly used due to its balance of performance and simplicity.

A screenshot of a computer

Description automatically generatedHere is the **space complexity chart** for the List implementations in the Java Collections Framework:

1. **ArrayList**:
   * Space grows linearly with the number of elements (O(n)).
   * Resizing doubles the internal array, but unused capacity may lead to overhead temporarily.
2. **LinkedList**:
   * Each node stores data and two pointers (next and previous).
   * Additional memory overhead due to pointers, making it less space-efficient compared to ArrayList.
3. **Vector**:
   * Similar to ArrayList in memory usage (O(n)).
   * Thread-safety introduces slight overhead due to synchronization locks.
4. **Stack**:
   * Inherits Vector's characteristics, resulting in O(n) complexity.
   * Ideal for small-scale LIFO operations where memory overhead isn't critical.

**Which is Best for Space Efficiency?**

* **ArrayList** is the most space-efficient in most cases because it doesn't have the pointer overhead of LinkedList.
* Avoid **LinkedList** when memory usage is a concern, especially for large datasets.

SET FAMILY

A screenshot of a computer

Description automatically generated

1. **HashSet**:
   * Average-case performance is O(1) for all operations.
   * Worst-case complexity can degrade to O(n) if there are many hash collisions.
2. **LinkedHashSet**:
   * Same as HashSet but uses a doubly-linked list internally to maintain insertion order.
3. **TreeSet**:
   * Provides O(log n) for all operations due to its underlying Red-Black Tree implementation.
   * Suitable when sorted order is needed.
4. **EnumSet**:
   * Extremely efficient for enum types; operations are constant time (O(1)).
   * Not applicable for non-enum data types.
5. **CopyOnWriteArraySet**:
   * Inefficient for frequent modifications as it creates a new copy of the backing array for every update.
   * Best for scenarios with **few writes but frequent reads** (thread-safe).

**Which is Best?**

* **HashSet** is excellent for most use cases due to its O(1) average performance.
* Use **TreeSet** if sorted elements are needed.
* Opt for **LinkedHashSet** if you need predictable iteration order.
* Consider **EnumSet** for enums and **CopyOnWriteArraySet** for thread-safe, read-heavy use cases.

===============================================================

A screenshot of a computer

Description automatically generated

1. **HashSet**:
   * Space grows linearly with the number of elements (O(n)).
   * Some additional memory is used for hash buckets and internal HashMap entries.
2. **LinkedHashSet**:
   * Slightly more memory-intensive than HashSet due to the doubly linked list used for maintaining insertion order.
3. **TreeSet**:
   * Linear space complexity (O(n)).
   * Each element in the Red-Black Tree requires additional memory for pointers to child nodes and parent nodes.
4. **EnumSet**:
   * Extremely space-efficient because it uses a bit vector representation.
   * Space is limited to either 32 bits (for enums with <= 32 constants) or 64 bits (for enums with <= 64 constants).
5. **CopyOnWriteArraySet**:
   * Requires O(n) space for storing elements in an array.
   * High memory usage in scenarios with frequent modifications due to array copying.

**Which is Best for Space Efficiency?**

* **EnumSet** is the most space-efficient when working with enums, as it uses compact bit vectors.
* **HashSet** is a good balance of performance and space efficiency for general-purpose sets.
* Avoid **LinkedHashSet** and **CopyOnWriteArraySet** if memory overhead is a concern.

=================================================================

QUEUE FAMILY

A screenshot of a chart

Description automatically generated

1. **PriorityQueue**:
   * Uses a **heap** internally, which means **O(log n)** time complexity for both insertion and removal.
   * **Access** (peek) is O(1) because the highest-priority element is at the top.
2. **LinkedList (Queue)**:
   * Implements a **doubly linked list**, so both **insertion** and **removal** at the head or tail are **O(1)**.
   * **Access** (peek) is also **O(1)**, but it is not ideal for random access or indexing.
3. **ArrayDeque (Queue)**:
   * Uses a **resizable array** and is very efficient for insertion and removal at both ends. Hence, both **insertion** and **removal** are **O(1)**.
   * Access is also **O(1)**, but less efficient compared to a simple array due to resizing.
4. **ConcurrentLinkedQueue**:
   * **Non-blocking** and **lock-free** queue designed for concurrent use, with **O(1)** operations for insertion, removal, and access.
5. **DelayQueue**:
   * A specialized queue based on priority ordering, where elements are removed based on their delay time. **Insertion** and **removal** are both **O(log n)** due to the underlying **priority queue**.
6. **BlockingQueue**:
   * Both **insertion** and **removal** are **O(1)**.
   * It is used for scenarios where threads need to wait for elements to become available (e.g., **LinkedBlockingQueue** and **ArrayBlockingQueue**). Operations block when the queue is empty or full, respectively.

**Space Complexity:**

* **O(n)** for all queues in this family, where **n** is the number of elements in the queue.
* For **PriorityQueue**, the space complexity is due to the underlying heap storage.
* For **ArrayDeque** and **LinkedList**, it's due to the array or linked nodes, respectively.

**Which One to Choose?**

* **ArrayDeque** is generally the best choice for most use cases when you need an efficient, flexible, and non-blocking queue implementation.
* **PriorityQueue** is ideal when you need to prioritize elements based on a custom comparator.
* **LinkedList** is best for simple queues where frequent insertion and removal at both ends are required.
* **BlockingQueue** is the go-to option for thread-safe, blocking operations in multi-threaded environments.

=============================

MAP FAMILY

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. **HashMap**:
   * **Insertion**, **search**, and **deletion** are **O(1)** on average because it uses a hash table.
   * The **space complexity** is **O(n)**, where **n** is the number of key-value pairs.
   * Performance can degrade to **O(n)** in the worst case if there are many hash collisions.
2. **LinkedHashMap**:
   * Same time complexities as **HashMap** for basic operations.
   * The added overhead for maintaining the insertion order uses **O(n)** extra space for the linked list of entries.
   * Offers predictable iteration order based on insertion order.
3. **TreeMap**:
   * **Insertion**, **search**, and **deletion** are **O(log n)** because it uses a Red-Black tree.
   * **Space complexity** is **O(n)** due to the need for tree pointers.
   * Provides a sorted map, where the keys are sorted according to their natural ordering or a specified comparator.
4. **ConcurrentHashMap**:
   * Similar to **HashMap** for basic operations, but it is **thread-safe**.
   * Uses a **lock striping** mechanism for efficient concurrency.
   * Good for use cases where multiple threads need to access and modify the map concurrently.
   * **O(n)** space complexity due to the underlying storage structure.
5. **WeakHashMap**:
   * Works similarly to **HashMap**, but its keys are stored as **weak references**.
   * This allows the garbage collector to remove entries when the key is no longer in use, reducing memory usage in certain situations.
   * **O(n)** space complexity, but the actual memory footprint can be lower due to garbage collection behavior.
6. **IdentityHashMap**:
   * Works similarly to **HashMap**, but it uses **reference equality** (==) instead of **equals()** for key comparison.
   * This is useful in scenarios where you need to compare object references rather than their content.
   * **O(n)** space complexity.
7. **Hashtable**:
   * Legacy synchronized map, similar to **HashMap** but with synchronization overhead for thread safety.
   * The **time complexity** for operations is **O(1)** on average, but performance is slower compared to other maps due to synchronization.
8. **TreeMap (Custom Comparator)**:
   * Same as **TreeMap**, but it allows specifying a **custom comparator** to define the ordering of keys.

**Which Map to Choose?**

* **HashMap** is typically the best choice for most use cases due to its performance and flexibility.
* Use **LinkedHashMap** if you need predictable iteration order (insertion order).
* Choose **TreeMap** if you need a sorted map and the overhead of **O(log n)** is acceptable.
* Use **ConcurrentHashMap** when thread safety is required in a concurrent environment.
* Use **WeakHashMap** if you need to manage memory-sensitive mappings where keys can be garbage collected.
* **IdentityHashMap** is best for cases where you need to compare keys by reference equality rather than value equality.

==================================================================

ARRAY

A screenshot of a computer

Description automatically generated

1. **Access (indexing)**:
   * Direct access to an element at a specific index is a constant-time operation, **O(1)**. However, the **space complexity** is **O(n)** because the array stores **n** elements.
2. **Search**:
   * In **linear search**, you need to iterate over each element to find the target value, leading to **O(n)** time complexity.
   * The **space complexity** remains **O(n)** due to the array storage.
3. **Insertion**:
   * Inserting an element in a fixed-size array usually requires shifting elements to accommodate the new one, resulting in **O(n)** time complexity.
   * For **dynamic arrays** like ArrayList, insertion might involve resizing, which requires **O(n)** time for copying the array to a new, larger one.
   * The space complexity is **O(n)**, as the array needs to store all its elements.
4. **Deletion**:
   * Deletion in an array often requires shifting the remaining elements to close the gap, leading to **O(n)** time complexity.
   * The space complexity is still **O(n)**, as the array holds the same number of elements even after deletion.
5. **Resize (Dynamic Array)**:
   * In **dynamic arrays** like ArrayList, resizing happens when the array reaches its capacity limit. The resizing process involves allocating a new array with a larger size and copying elements from the old array, which takes **O(n)** time.
   * The space complexity is **O(n)**, reflecting the total space occupied by the array.
6. **Appending (Dynamic Array)**:
   * For dynamic arrays, appending is typically **O(1)**, but occasionally, resizing is needed, which takes **O(n)** time.
   * The space complexity is **O(n)**, as the array must hold all its elements.

**Summary:**

* **Time Complexity** for common operations:
  + **Access**: **O(1)**
  + **Search**: **O(n)**
  + **Insertion**: **O(n)** (due to shifting elements)
  + **Deletion**: **O(n)** (due to shifting elements)
  + **Resize (Dynamic array)**: **O(n)** (when resizing happens)
  + **Appending**: **O(1)** amortized (except during resizing)
* **Space Complexity** is typically **O(n)**, as the array stores **n** elements.

This makes arrays highly efficient for random access and appending (in dynamic arrays), but less efficient for insertions and deletions where elements need to be shifted.

A white and black text on a white background

Description automatically generated

**Conclusion:**

* **For fast lookups** and key-based storage, **HashMap** and **ConcurrentHashMap** are ideal choices due to their **constant-time** performance in most operations.
* **For maintaining insertion order** with efficient access and updates, **LinkedHashMap** is a great choice.
* **For managing priorities**, **PriorityQueue** offers a highly efficient solution with logarithmic time complexity for insertion and removal.
* **For random access and dynamic resizing**, **ArrayList** excels in providing fast access and amortized constant-time insertion.

These data structures offer a balance between **time complexity** and **space complexity**, making them excellent choices for different use cases in Java development.

**Why HashSet Wasn't Initially Included:**

While **HashSet** is highly efficient in terms of **time complexity (O(1) for add, remove, and contains)**, it was **not included in the original list** for the following reasons:

1. **Lack of Order**:
   * A HashSet does not maintain any order of its elements. If your use case requires maintaining insertion order, like in the case of LinkedHashMap or ArrayList, then HashSet would not be an ideal choice.
   * In cases where you need to keep track of the insertion order or need ordered traversal, a data structure like LinkedHashSet (which maintains insertion order) or TreeSet (which keeps elements in sorted order) would be better.
2. **No Built-in Priority Management**:
   * Unlike PriorityQueue, which allows you to manage and retrieve elements based on priority (e.g., min or max), HashSet simply stores elements without considering any order or priority. It’s highly efficient for membership testing but lacks features for priority-based sorting.

**Inclusion of HashSet in the Top 5:**

If we expand the list to include **unordered collections** that offer **efficient membership tests** and **element insertion/deletion**, then **HashSet** would definitely deserve to be part of the top 5 for its **excellent time complexity (O(1) for most operations)** and **space complexity (O(n))**.

Here’s how it could fit in the list:

A screenshot of a computer screen

Description automatically generated

**Conclusion:**

**HashSet** is an excellent data structure for **efficient membership testing** and **element management** with no duplicates. It's especially well-suited for applications like **set operations (union, intersection, etc.)**, **caching**, or **filtering unique elements**. It would be included in the **top 5** for its **constant-time operations** (on average) and **linear space complexity** for storing elements.

However, it is excluded in cases where **order** or **priority management** is necessary.