Data Structures Cheat Sheet

1. Arrays

A fixed-size, ordered collection of elements of the same type.

Pros:

- Fast access and update (O(1) time complexity).
- Memory-efficient for known, fixed-size data.

Cons:

- Fixed size; cannot dynamically grow or shrink.
- Costly insertions and deletions (O(n) time complexity) except at the end.

2. ArrayList

A resizable array implementation of the List interface.

Pros:

- Dynamic size; can grow and shrink as needed.
- Fast random access (O(1) time complexity).

Cons:

- Higher memory overhead due to dynamic resizing.

- Costly insertions and deletions (O(n) time complexity) for elements other than at the end.

Real-Life Example: Managing a dynamic to-do list.

3. LinkedList

A doubly linked list implementation of the List and Deque interfaces.

Pros:

- Efficient insertions and deletions (O(1) time complexity) at the beginning or end.
- No fixed size; can dynamically grow and shrink.

Cons:

- Higher memory usage due to storage of pointers.
- No random access: accessing an element requires traversal (O(n) time complexity).

Real-Life Example: Maintaining browser history.

4. HashMap

A hash table-based implementation of the Map interface.

Pros:

- Fast insertions, deletions, and lookups (O(1) average time complexity).
- Flexible keys and values.

Cons:

- No guaranteed order of elements.
- Can be memory-intensive due to hash buckets and handling collisions.

Real-Life Example: Storing an employee directory with IDs as keys.

5. HashSet

A hash table-based implementation of the Set interface, ensuring no duplicate elements.

Pros:

- Fast insertions, deletions, and lookups (O(1) average time complexity).
- Ensures unique elements.

Cons:

- No guaranteed order of elements.
- Memory usage can be high due to hash buckets.

Real-Life Example: Ensuring unique usernames in a system.

6. Stack

A LIFO (Last In, First Out) data structure.

Pros:

- Simple API for push, pop, and peek operations.
- Efficient for LIFO access patterns.

Cons:

- Limited to LIFO operations.
- No random access: must pop elements to access others.

Real-Life Example: Implementing an undo feature in a text editor.

7. Queue

A FIFO (First In, First Out) data structure.

Pros:

- Simple API for enqueue and dequeue operations.
- Efficient for FIFO access patterns.

Cons:

- Limited to FIFO operations.
- No random access: must dequeue elements to access others.

Real-Life Example: Managing customer service call queues.

8. PriorityQueue

A special queue where elements are ordered by their priority.

Pros:

- Automatically orders elements by priority.
- Efficient for priority-based task scheduling.

Cons:

- No random access: only the highest priority element is accessible.
- Insertion and removal are O(log n) time complexity.

Real-Life Example: Scheduling tasks based on priority.

9. TreeMap

A Red-Black tree-based implementation of the Map interface, ensuring sorted order.

Pros:

- Maintains sorted order of keys.
- Efficient for range queries and ordered traversal.

Cons:

- Slower insertions, deletions, and lookups (O(log n) time complexity) compared to hash-based structures.
- Higher memory usage due to tree structure.

Real-Life Example: Storing and accessing books sorted by Id.

10. LinkedHashMap

A hash table and linked list implementation of the Map interface, maintaining insertion order.

Pros:

- Maintains insertion order of elements.
- Efficient for implementing LRU caches.

Cons:

- Higher memory usage compared to HashMap due to linked list structure.
- Slightly slower than HashMap for insertions and lookups.

Real-Life Example: Implementing an LRU (Least Recently Used) cache.

http://www.tutorialspoint.com/java/java_collections.htm

https://docs.oracle.com/javase/tutorial/collections/