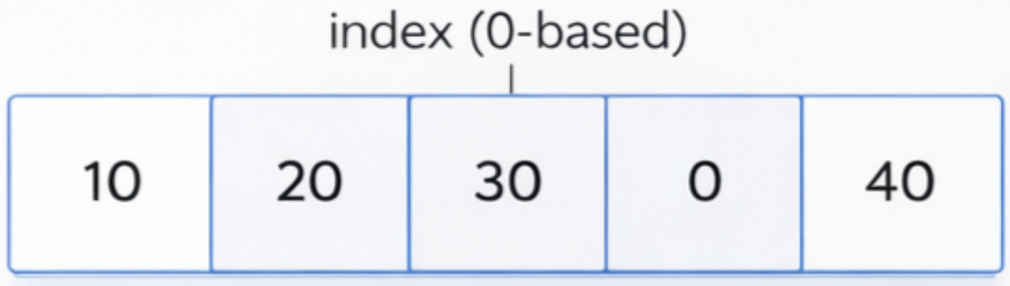
Java Collections Internals & Trade-offs

How they really work, when to use them, and their performance trade-offs

Array



What it is ?

A fixed-size, indexed data structure that holds elements of the same type in a contiguous manner.

When to use ?

- Size is known and fixed
- Need fast random access
- Want low memory overhead
- Performance-critical or low-level logic

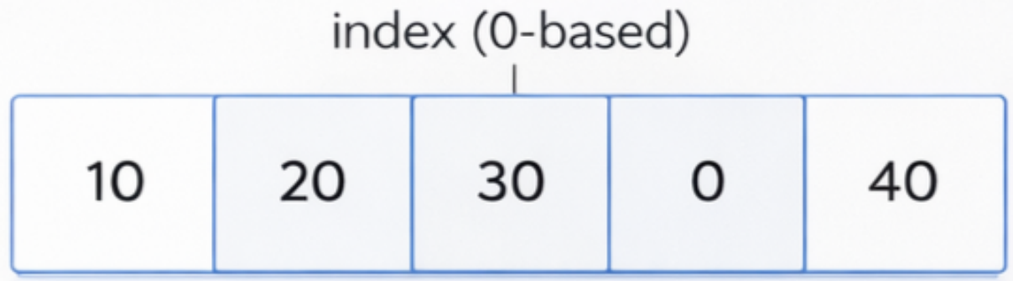
⁄ Avoid when:

- Frequent insertions or deletions
- Dynamic resizing is needed

Time Complexity

Operation	Complexity
Access (by index)	**O(1)**
Insert / Delete (not end)	**O(n)**
Search (unsorted)	**O(n)**
Search (sorted)	**O(log n)**

ArrayList



What it is ?

A **resizable array**-backed implementation of the List interface that maintains insertion order and allows duplicate elements.

When to use ?

- Need fast random access
- List size is dynamic
- More reads than inserts/deletes in the middle
- Most general-purpose List use cases

/ Avoid when:

- Heavy insertions/deletions in the middle
- Strict thread-safety is required

Time Complexity

Operation	Complexity
Access (by index)	**O(1)**
Insert (at end)	**O(1) amortized**
Insert / Delete (not end)	**O(n)**
Search (contains)	**O(n)**

LinkedList

What it is ?

A list where each element is stored as a **node**, and nodes are connected using links instead of indexes.

When to use ?

- Frequent insertions or deletions (especially at beginning or middle)
- Order matters
- Access pattern is mostly sequential, not random

⁄ Avoid when:

- Need fast random access
- Heavy get(index) operations

Time Complexity

Operation	Complexity
Access (by index)	**O(n)**
Insert / Delete	**O(1)**
Search (contains)	**O(n)**

Stack

What it is ?

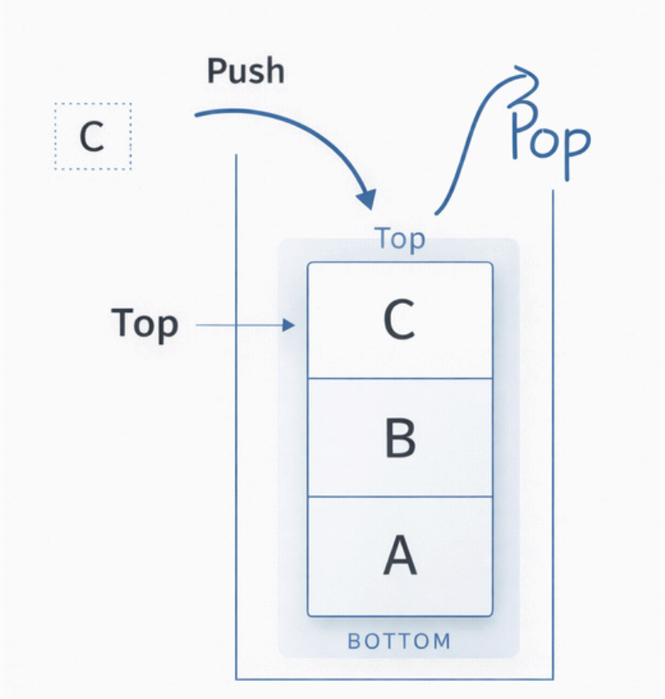
A linear data structure that follows **Last In, First Out** (LIFO) .

When to use ?

- Need reverse order processing
- Function calls / recursion handling
- Undo / redo operations
- Expression evaluation, parsing
- Backtracking problems

⁄ Avoid when:

- Random access is required
- **FIFO** behavior is needed



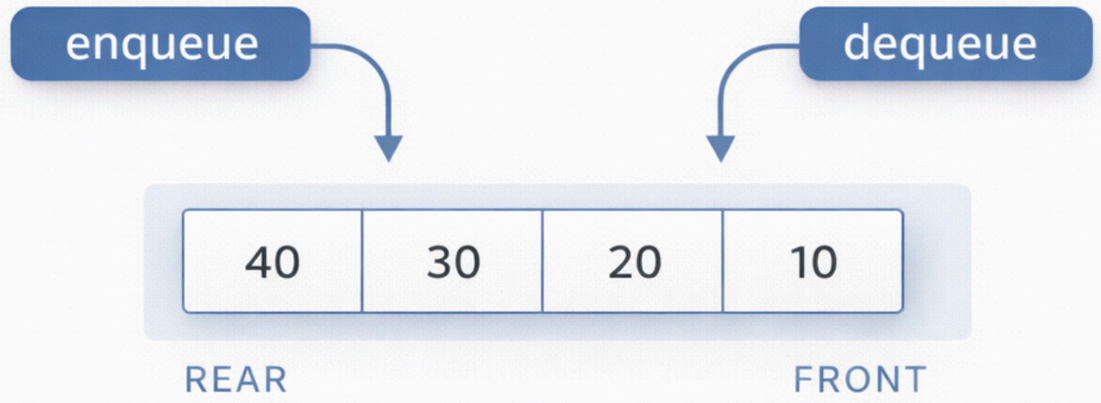
Time Complexity

Operation	Complexity
Access (by index)	**O(n)**
Pus / Pop / Peek	**O(1)**
Search (contains)	**O(n)**

Queue

What it is ?

A linear data structure that follows **First In, First Out** (FIFO)



When to use ?

- Order of processing must be preserved
- Task scheduling
- Request handling (producer–consumer)
- Buffers, messaging systems
- BFS (Breadth First Search)

⁄ Avoid when:

- Need **LIFO** behavior
- Random access is required

Time Complexity

Operation	Complexity
Access (by index)	**O(n)**
Offer / Poll / Peek	**O(1)**
Search (contains)	**O(n)**

PriorityQueue

What it is ?

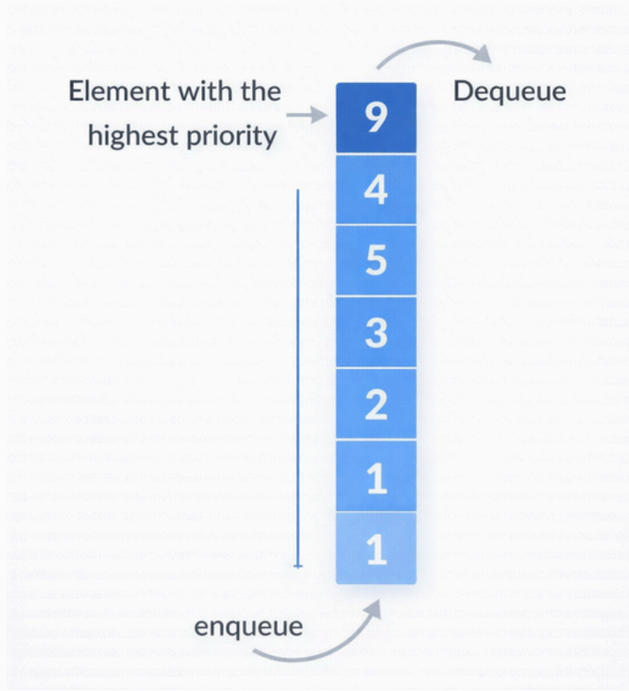
A queue where elements are processed based on **priority**, not insertion order.

When to use ?

- Always need highest / lowest priority element first
- Scheduling tasks
- Top-K problems
- Dijkstra algorithms
- Load balancing

⁄ Avoid when:

- Strict FIFO order is required
- You need to iterate in sorted order (PQ does not guarantee that)



Time Complexity

Operation	Complexity
Insert (offer)	**O(log n)**
Remove (poll)	**O(log n)**
Peek	**O(1)**
Search (contains)	**O(n)**

HashMap

What it is ?

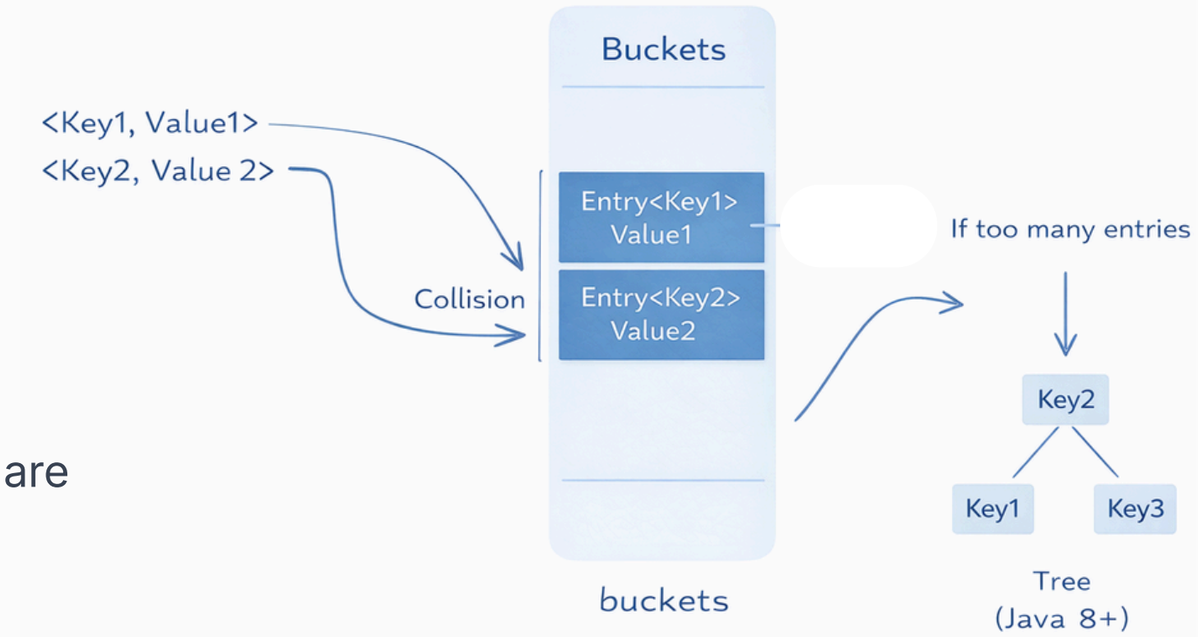
A **key-value** based data structure where keys are unique and values are accessed using those keys, not indexes.

When to use ?

- Need fast lookups by key
- No ordering requirement
- Caching
- Counting / frequency maps
- Mapping one object to another

⁄ Avoid when:

- Order matters
- Sorted keys are required
- Thread-safety is mandatory (without extra handling), prefer ConcurrentHashMap



Time Complexity

Operation	Complexity
Put/Get/Remove(Average case)	**O(1)**
Put/Get/Remove(Worst case)	**O(log n)**
Search (by value)	**O(n)**

LinkedHashMap

What it is ?

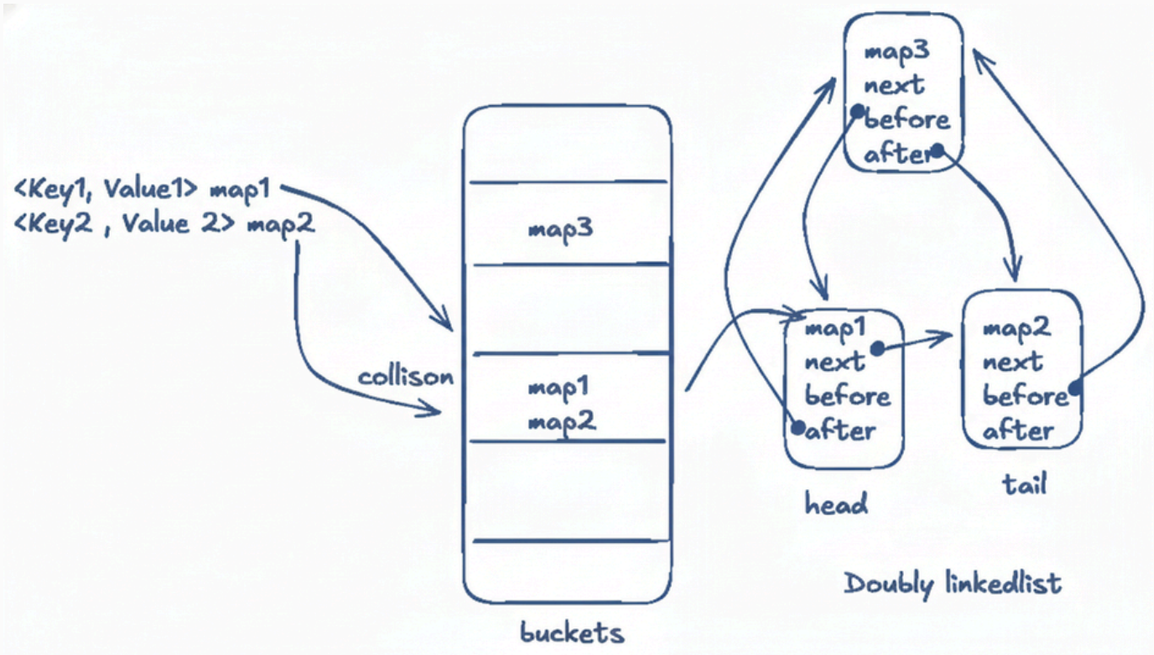
A HashMap that maintains order using a doubly linked list in addition to hashing.

When to use ?

- Order matters
- Need predictable iteration
- Implementing LRU cache
- Read-heavy maps where ordering adds value

Avoid when:

- Order is irrelevant
- You want minimal memory overhead



Time Complexity

Operation	Complexity
Put/Get/Remove(Average case)	**O(1)**
Put/Get/Remove(Worst case)	**O(log n)**
Search (by value)	**O(n)**

TreeMap

What it is ?

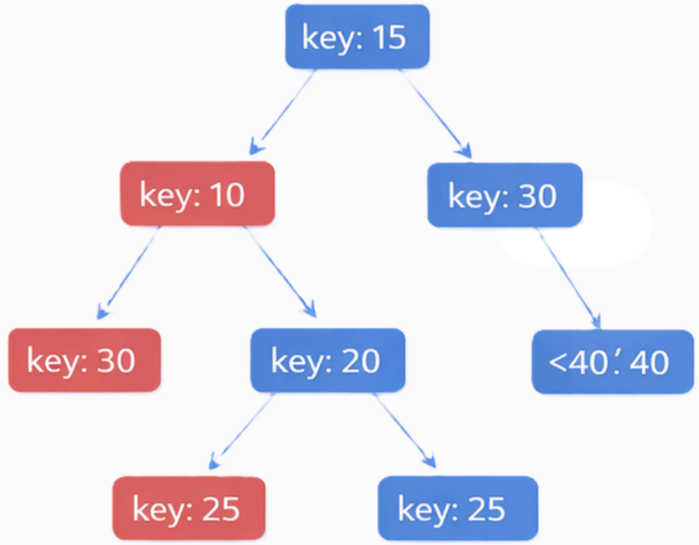
A Map implementation that stores keys in a sorted order.

When to use ?

- You need sorted keys
- Range queries (headMap, tailMap, subMap)
- Ordered iteration matters more than speed

/ Avoid when:

- You need constant-time lookups
- Order is irrelevant



Red-Black Tree

Time Complexity

Operation	Complexity
Put/Get/Remove(Worst case/Average case)	**O(log n)**
Search (by value)	**O(n)**

HashSet

What it is ?

A Set implementation that stores unique elements with no guaranteed order, internally backed by HashMap.

Internally Use HashMap

When to use ?

- Need uniqueness
- Fast contains checks
- Removing duplicates from a collection
- Membership tests

/ Avoid when:

- Order matters, prefer LinkedHashSet
- Sorted elements are required, prefer TreeSet

,

Time Complexity

Operation	Complexity
Put/Get/Remove(Average case)	**O(1)**
Put/Get/Remove(Worst case)	**O(log n)**
Search (by value)	**O(n)**