1. Array

- **Access**: O(1) - Direct access to any index.
- **Search**: O(n) - Linear search needed for unsorted arrays.
- **Insertion**:
    - End: O(1) (amortized for dynamic arrays)
    - Arbitrary position: O(n) due to shifting.
- **Deletion**: O(n) - Shifting required for elements after the deleted element.

2. Linked List

- **Access**: O(n) - Linear search needed, as no direct access.
- **Search**: O(n) - Linear search.
- **Insertion**:
    - Beginning or end: O(1)
    - Arbitrary position: O(n) (finding position takes O(n))
- **Deletion**:
    - Beginning: O(1)
    - Arbitrary position: O(n)

3. Stack (LIFO)

- **Push (Insert)**: O(1) - Constant time for adding to the top.
- **Pop (Remove)**: O(1) - Constant time for removing from the top.
- **Peek (Top element)**: O(1)

4. Queue (FIFO)

- **Enqueue (Insert)**: O(1) - Constant time to add at the end.
- **Dequeue (Remove)**: O(1) - Constant time to remove from the front.
- **Peek (Front element)**: O(1)

5. Hash Table

- **Access/Search**: Average O(1), Worst-case O(n) - Worst case occurs with poor hash functions (all elements in one bucket).
- **Insertion**: Average O(1), Worst-case O(n) - Similar reasons to access/search.
- **Deletion**: Average O(1), Worst-case O(n)

6. Binary Search Tree (BST)

- **Access/Search**: Average O(log n), Worst-case O(n) - Degenerates to O(n) for unbalanced trees.
- **Insertion**: Average O(log n), Worst-case O(n)
- **Deletion**: Average O(log n), Worst-case O(n)

7. AVL Tree (Self-Balancing BST)

- **Access/Search**: O(log n) - Always balanced.
- **Insertion**: O(log n) - Rebalancing may be needed.
- **Deletion**: O(log n) - Rebalancing may be needed.

8. Red-Black Tree (Self-Balancing BST)

- **Access/Search**: O(log n)
- **Insertion**: O(log n)
- **Deletion**: O(log n)

9. Heap (Binary Heap)

- **Access (Min/Max):** O(1) - Constant access to root (min/max).
- **Insertion**: O(log n) - Maintains heap property.
- **Deletion (Min/Max)**: O(log n) - Removing root and re-heapifying.

10. Graph (Adjacency List Representation)

- **Add Vertex**: O(1)
- **Add Edge**: O(1)
- **Remove Vertex**: O(V + E) - Requires updating all edges associated with the vertex.
- **Remove Edge**: O(E) - Finding the edge in the adjacency list.
- **Search (DFS/BFS)**: O(V + E)

STL containers:

1. Vector (std::vector)

- **Access (operator[]): O(1) - Direct access by index.**

- **Insertion:**
  - **End: O(1) amortized (due to resizing, which occasionally takes O(n))**
  - **Arbitrary position: O(n) - Elements must be shifted.**
- **Deletion:**
  - **End: O(1)**
  - **Arbitrary position: O(n) - Shifting required.**
- **Search (find): O(n) - Linear search unless sorted.**

2. Deque (std::deque)

- **Access (operator[]): O(1) - Direct access by index.**
- **Insertion/Deletion:**
  - **Front/End: O(1) - Efficient for both ends.**
  - **Arbitrary position: O(n) - Shifting may be required.**
- **Search (find): O(n)**

3. List (std::list) (Doubly Linked List)

- **Access: O(n) - No random access.**
- **Insertion/Deletion:**
  - **Beginning, end, or middle (with iterator): O(1)**
- **Search (find): O(n) - Linear search.**

4. Forward List (std::forward_list) (Singly Linked List)

- **Access: O(n)**
- **Insertion/Deletion:**
  - **Beginning or with iterator: O(1)**
  - **End: O(n) (no tail pointer for singly linked list)**
- **Search (find): O(n)**

5. Set (std::set) and Multiset (std::multiset) (Balanced Binary Search Tree)

- **Access: No direct access by index.**
- **Insertion: O(log n)**
- **Deletion: O(log n)**
- **Search: O(log n)**

- **Traversal: O(n) - Elements are sorted.**

6. Unordered Set (std::unordered_set) and Unordered Multiset (std::unordered_multiset) (Hash Table)

- **Access: No direct access by index.**
- **Insertion: Average O(1), Worst-case O(n) (due to collisions).**
- **Deletion: Average O(1), Worst-case O(n)**
- **Search: Average O(1), Worst-case O(n)**
- **Traversal: O(n) - No specific order.**

7. Map (std::map) and Multimap (std::multimap) (Balanced Binary Search Tree)

- **Access (operator[]): O(log n)**
- **Insertion: O(log n)**
- **Deletion: O(log n)**
- **Search: O(log n)**
- **Traversal: O(n) - Elements are sorted by key.**

8. Unordered Map (std::unordered_map) and Unordered Multimap (std::unordered_multimap) (Hash Table)

- **Access (operator[]): Average O(1), Worst-case O(n)**
- **Insertion: Average O(1), Worst-case O(n)**
- **Deletion: Average O(1), Worst-case O(n)**
- **Search: Average O(1), Worst-case O(n)**
- **Traversal: O(n) - No specific order.**

9. Stack (std::stack) (Typically implemented with std::deque or std::vector)

- **Push: O(1)**
- **Pop: O(1)**
- **Top: O(1)**

10. Queue (std::queue) (Typically implemented with std::deque)

- **Enqueue (Push): O(1)**
- **Dequeue (Pop): O(1)**
- **Front: O(1)**

- **Back: O(1)**

11. Priority Queue (std::priority_queue) (Heap)

- **Push: O(log n)**
- **Pop: O(log n) - Removes the highest/lowest priority element.**
- **Top: O(1)**

--

Time complexity of Sorting algorithms:

1. Bubble Sort

- **Best Case: O(n) - When the array is already sorted.**
- **Average Case: O(n^2) - Due to nested loops.**
- **Worst Case: O(n^2) - When the array is in reverse order.**
- **Space Complexity: O(1) - In-place sorting.**

2. Selection Sort

- **Best, Average, Worst Case: O(n^2) - Selection process is the same regardless of initial order.**
- **Space Complexity: O(1) - In-place sorting.**

3. Insertion Sort

- **Best Case: O(n) - When the array is already sorted.**
- **Average Case: O(n^2) - When elements are in random order.**
- **Worst Case: O(n^2) - When the array is in reverse order.**
- **Space Complexity: O(1) - In-place sorting.**

4. Merge Sort

- **Best, Average, Worst Case: O(n log n) - Recursively divides the array in half and merges.**
- **Space Complexity: O(n) - Requires auxiliary space for merging.**

5. Quick Sort

- **Best Case: O(n log n) - Partitioning splits array evenly.**
- **Average Case: O(n log n) - Good balance of partition splits.**

- **Worst Case: $O(n^2)$ - Occurs when pivot repeatedly picks smallest or largest element (like already sorted data if pivot choice is poor).**
- **Space Complexity: O(log n) - In-place, though recursion adds to call stack.**

6. Heap Sort

- **Best, Average, Worst Case: O(n log n) - Heapifying and extracting the max/min.**
- **Space Complexity: O(1) - In-place sorting.**

7. Counting Sort (only for integers within a range)

- **Best, Average, Worst Case: O(n + k) - k is the range of input values.**
- **Space Complexity: O(n + k) - Requires auxiliary space for count array.**

8. Radix Sort (used with Counting Sort for each digit level)

- **Best, Average, Worst Case: O(d * (n + k)) - d is number of digits, and k is range.**
- **Space Complexity: O(n + k) - Needs auxiliary space for counting.**

9. Bucket Sort (best for uniformly distributed data)

- **Best Case: O(n) - When data is uniformly distributed.**
- **Average Case: O(n + k) - k is the number of buckets.**
- **Worst Case: $O(n^2)$ - When all elements are in the same bucket.**
- **Space Complexity: O(n + k) - For buckets and auxiliary space.**

10. Tim Sort (hybrid of Merge Sort and Insertion Sort; used in Python and Java)

- **Best Case: O(n) - Already sorted array.**
- **Average, Worst Case: O(n log n) - Efficiently handles real-world data.**
- **Space Complexity: O(n) - Uses auxiliary space for merging.**

Summary Table

Sorting Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)

Sorting Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(n + k)
Radix Sort	O(d(n + k))	O(d(n + k))	O(d(n + k))	O(n + k)
Bucket Sort	O(n)	O(n + k)	$O(n^2)$	O(n + k)
Tim Sort	O(n)	O(n log n)	O(n log n)	O(n)