C++ DATA STRUCTURE COMPLEXITIES

DS	Access	Find	Ins/Del	Common Funcs
Array	$O(1)$	$O(n)$	N/A	`[]`
Vector	$O(1)$	$O(n)$	$O(n)^*$	`[]`, `at`, `push_back` `insert`, `erase`
List (DLL)	$O(n)$	$O(n)$	$O(1)$	`push_back/front` `insert`, `erase`
Fwd List	$O(n)$	$O(n)$	$O(1)$	`push_front`
Deque	$O(1)$	$O(n)$	$O(n)^*$	`push_back/front`
Stack	$O(1)$	-	$O(1)$	`push`, `pop`, `top`
Queue	$O(1)$	-	$O(1)$	`push`, `pop`, `front`
Set	-	$O(\lg n)$	$O(\lg n)$	`insert`, `find`, `erase`
Unord Set	-	$O(1)^\dagger$	$O(1)^\dagger$	`insert`, `find`, `erase`
Map	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	`[]`, `insert`, `find`
Unord Map	$O(1)^\dagger$	$O(1)^\dagger$	$O(1)^\dagger$	`[]`, `insert`, `find`

** $O(1)$ at end (vector) or ends (deque). † Average case; worst $O(n)$.*

LINKED LISTS (LL)

Singly LL: Node contains `data` and `next`. **Doubly LL:** Node contains `data`, `next`, and `prev`.

Operations

- **Traversal:** $O(n)$. Start at head, loop until `nullptr`.
- **Front Insert (SLL/DLL):** $O(1)$. New node points to head, head points to new.
- **Tail Insert:** $O(1)$ if tail pointer exists, else $O(n)$.
- **Middle Insert:** $O(n)$ to find spot, $O(1)$ to rewire.

Pointer Logic (Insert Node `N` after `P`)

- **SLL:** `N->next = P->next; P->next = N;`
- **DLL:** `N->next = P->next; N->prev = P;` `P->next->prev = N;` (if `P->next` exists) `P->next = N;`

Comparison vs Array/Vector

- **Memory:** LL has overhead (pointers) per node. Vector is contiguous.
- **Access:** LL is sequential $O(n)$. Vector is random $O(1)$.
- **Resizing:** LL is dynamic (no copy needed). Vector doubles capacity & copies elements when full.

STACKS & QUEUES

Stack (LIFO): Implemented via Vector, List, or Array.

- **Array:** Checks capacity. `top_index++`. Potential overflow.
- **List:** Push/Pop at head ($O(1)$). No overflow unless OOM.
- **Applications:** Function call stack, recursion, parsing.

Queue (FIFO):

- **List:** Enqueue at tail, Dequeue at head. Need tail pointer.
- **Circular Array:** Use `front` and `size/count`.
- **Wrap-around:** `index = (index + 1) % capacity`.
- **Full/Empty:** Hard to distinguish if only using `front/rear`; use `size` variable.

ITERATORS

Object allowing traversal of container elements.

- **Usage:** `for(auto it = c.begin(); it != c.end(); ++it)`

Categories

- **Random Access:** `+`, `-`, `[]`, `<`, `>`. (Vector, Deque, Array).
- **Bidirectional:** `++`, `--`. (List, Set, Map).
- **Forward:** `++` only. (Forward List, Unordered Map/Set).

Invalidation

- **Vector:** Insertion/Resize invalidates *all* iterators (memory shift).
- **List/Map/Set:** Insertion/Deletion only invalidates iterators to the *specific* node removed.

SETS & MAPS

Ordered (std::set/map):

- **Structure:** Balanced BST (Red-Black Tree).
- **Order:** Keys sorted by `<`.
- **Iterating:** In-order traversal produces sorted output.

Unordered (std::unordered_set/map):

- **Structure:** Hash Table with chaining/buckets.
- **Order:** Undefined. Depends on hash.
- **Reqs:** Key needs `hash` function and `==` operator.

Map Specifics:

- Stores `std::pair<Key, Value>`.
- `map[key]` creates default element if key missing.
- `map.find(key)` returns iterator to end if missing.
- Keys are `const` (cannot modify key in place).
- No duplicate keys (use `multimap` for duplicates).

SORTING & SEARCHING

Algo	Best	Avg	Worst	Space
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge	$N \lg N$	$N \lg N$	$N \lg N$	$O(n)$
Quick	$N \lg N$	$N \lg N$	$O(n^2)$	$O(\lg n)$
Bin Search	$O(1)$	$O(\lg n)$	$O(\lg n)$	$O(1)$

Algorithm Mechanics

- **Selection:** Find min, swap with current index. Constant writes.
- **Insertion:** Take element, shift sorted left-side to make room. Good for nearly sorted.
- **Bubble:** Swap adjacent if wrong order. Largest bubbles to end. Stop if no swaps in pass.
- **Merge:** Recursive split, merge sorted halves. Stable.
- **Quick:** Pivot partition. Small left, large right. Unstable. Worst case: bad pivot (sorted array).

Searching

- **Linear:** $O(n)$. Any data.
- **Binary:** $O(\lg n)$. Requires **Sorted Random Access** data.

FUNCTIONAL C++

Function Pointers

Stores address of a function. Decays from func name.

```
// Return Type (*PtrName)(Args...)
double (*funcPtr)(int) = &myFunc;
funcPtr(5); // Call
```

Functors (Function Objects)

Class that overloads `operator()`.

- **Pros:** Can hold state (member variables).
- **Usage:** Pass to STL algos (e.g., sort comparator).

```
struct AddX {
  int x;
  AddX(int val) : x(val) {}
  int operator()(int y) { return x + y; }
};
```

Lambdas

Anonymous inline functions. **Syntax:** `[capture](params) -> return { body }`

- `[]` : No capture.
- `[=]` : Capture all local vars by value (copy).
- `[&]` : Capture all local vars by reference.
- `[x, &y]` : x by value, y by ref.

Dangling Ref: Capturing local var by ref in a lambda that outlives the scope causes undefined behavior.

RECURSION

- **Base Case:** Stops recursion (e.g., `if(n==0) return;`).
- **Recursive Step:** Calls self with smaller input.
- **Stack Overflow:** Too many frames (missing base case).
- **Tail Recursion:** Recursive call is last action (optimizable).

PYTHON

Basics

- **Variables:** Names bound to objects (References), not buckets.
- **Rebinding:** `x = 5` changes what x points to, doesn't modify object 5.
- **Comparisons:**
  - `==` checks value equality (content).
  - `is` checks identity (memory address).

Mutability

- **Immutable:** int, float, string, tuple. Modifying creates NEW object.
- **Mutable:** list, dict, set. Modified in-place.
- **Aliasing:** `b = a` (if a is list). Modifying `b` affects `a`.

Argument Passing

Pass-by-Object-Reference:

- Func gets reference to object.
- If object is mutable, changes persist outside func.
- If object is immutable, func creates local copy on write.

Containers

- **List:** `[1, 2]`. Ordered, mutable, duplicates ok.
- **Tuple:** `(1, 2)`. Ordered, immutable. Use for dict keys.
- **Set:** {`1, 2`}. Unordered, unique elements. $O(1)$ lookups.
- **Dict:** {`'k': v`}. Key-value. Keys must be immutable (hashable). Insertion-ordered (modern Python).

Slicing & Copying

- **Slicing:** `L[start:stop:step]`. Returns new list (shallow copy).
- **Shallow Copy:** `L2 = list(L1)`. Copies references. Nested objects shared.
- **Deep Copy:** `copy.deepcopy(L)`. Copies everything recursively.

OO & Special Methods

- `__init__`: Constructor.
- `__str__`: String representation (for print).
- `__eq__`: Defines `==` behavior.
- **Inheritance:** `class Child(Parent):`.
- `super().__init__()`: Call parent logic.