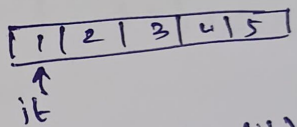
C++ STL

↓

Standard Template Library.

- Containers
  - Sequence Constrains → Arrn, deque, list, Vector.
  - Associative " → Set, map
  - Unordered " → unordered_map, set.
  - Container Adaptors. → Stack, queue, priority-queue
- Iterators
- Algorithms
  - Searching → lower, upper_bound, binary_Search.
  - Sorting →
  - Min/Max...
- Functors
  - greater<int>()
  - less<int>()

Vector := (Dynamic / Resize)



cout << *(it) ← print Value

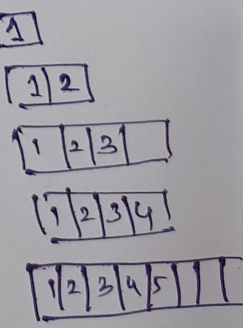
cout << it ← print address not Value

```
Vector<int> vec;  // empty Vector
Vector<int> vec1 = {1,2}; // initialize.
Vector<int> vec2 (3,10); // size=3, Value=10;
Vector<int> vec3(vec2); // copy Constructor.
```

→ Size & Capacity → if no space left it double's itself.

{1, 2, 3, 4, 5}

Capacity.



→ push_back / pop_back.

→ emplace_back.

→ at() and [] → index Value.

→ front() / back().

→ erase, insert, clear, empty.

{1, 2, 3, 4, 5}

vec.insert(vec.begin()+1, 100);

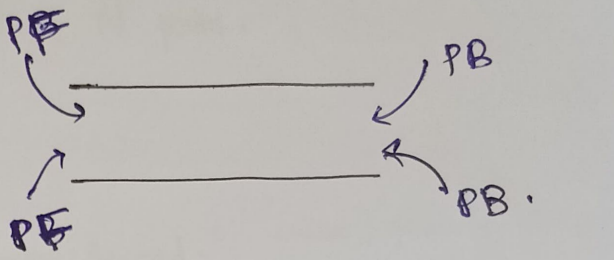
vec = {1, 100, 2, 3, 4, 5}.

List := (Doubly LinkedList)

→ Allow insertion & deletion from both side.

→ No Index Access (no l[0])

```
list<int> l = {1,2,3}
```



→ push_back, push_front.

→ emplace_back, emplace_front.

→ pop_back, pop_front.

→ size

→ erase, clear, insert.

→ front, back, begin, end.

↓ Accessing front & Back

↳ iterating.

• Insert At Any Position.

```
l = {1,2,3,4,5,6}
auto it = l.begin();
//advance(it,3)
```

move iterator to 3rd index (0-based).

```
//l.insert(it,100)
l = {1,2,3,100,4,5,6}
```

insert 100 before 4

Deque := (Double Ended Queue).

→ Allow insertion & deletion from both side.

→ Index Access. (unlike List).

```
deque<int> d = {1,2,3}
```

→ push_front, push_back | emplace

→ pop_back, pop_front.

→ front, back.

→ erase, insert, begin, end.

→ size, empty, clear.

→ [] and at().

→ Same As List

Pair :=

→ String as Key-Value.

→ When using vector / list / deque of pairs.

```
pair<int, int> p1 = {3,5};
pair<char, int> p2 = {'a', 1};
```

p2.first ↑ — p2.second ←

```
vector<pair<int, int>> vec = { {1,2}, {2,3}, {3,4} };
```

→ push_back() → vec.push_back({4,5});

→ emplace_back() → vec.push_back(4,5)

NON - SEQUENCE CONSTRAINS

Stack := (LIFO)

→ Reversing things (string, array).

→ Valid Paranthesis Problem

→ DFS

→ Stack is not Iterable (No begin(), end()).

→ push() → adds elem to TOP

→ emplace() → in-place.

→ top() → return the elem at TOP.

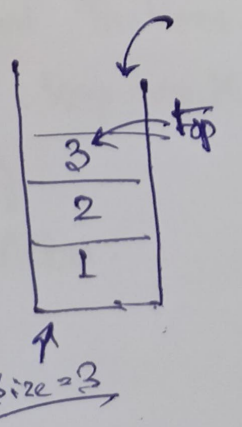
→ pop() → removes the TOP elem

→ size()

→ empty()

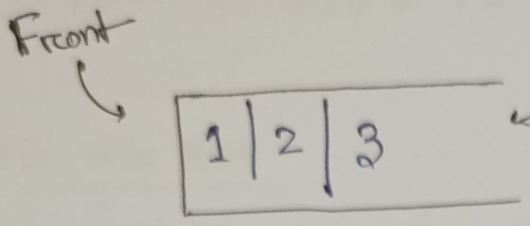
→ swap()

```
Stack<int> S;
S.push(1) // 1
S.push(2) // 2 1
S.push(3) // 3 2 1
```



Queue :— (FIFO).

→ BFS (lvl order) in Trees/Graphs
→ Task Scheduling, CPU Scheduling.
→ Producer - Consumer Problems.
→ No Iteration (No begin(), end())



push() ← insert at Rear

→ push() → adds elem at the BACK
→ emplace() → elems in-place at BACK
→ front() → elems at the FRONT (first inserted).
→ pop() → removes FRONT elem.
→ size()
→ empty()
→ swap().

```
queue<int> q;
q.push(1); // 1
q.push(2); // 1 2
q.push(3); // 1 2 3

q.front() // 1
q.POP() // 2 3
```

Priority_queue :— (MAX HEAP By Default).

→ Stores elems such that the largest elems always stays at the top.
→ When u always need the largest/smallest elems. quickly.
→ Dijkstra's Algo. (shortest Path).
→ K largest/smallest Problems.
→ Streaming data (keep top K elem)
→ Task Scheduling
→ Duplicate Allow.

```
priority_queue <int> pq; // MAX HEAP
                            Default

pq.push(5); // 5
pq.push(3); // 5 3
pq.push(10); // 10 5 3
pq.push(4); // 10 5 4 3
```

→ push() → insert elems.

→ emplace() → faster insertion

→ top() → returns current maxi

→ pop() → removes maximum

→ size()

→ empty()

For MIN HEAP

→ FUNCTORS (Function Object)

```
priority_queue <int, vector<int>, greater<int>> minpq; // MIN HEAP
```

↳ makes smallest elems highest priority.

if we use Pair

```
priority_queue <pair<int,int>> pairpq;

pairpq.push({2,100});
pairpq.emplace(1,100); // in-place.
```

here max-heap

Based on first elem → output:— (2,200), (1,100)

Map :=

→ map<K,V> stores key-value pairs in sorted order (ascending)

→ Keys are Unique. ✓

→ Balanced BST.

map<string, int> m; ⟶ Key wise sorting.

//Operator [] inserts OR Updates a Key.

```
m["tv"] = 100;
m["laptop"] = 100;
m["headphones"] = 50;
m["tablet"] = 120;
m["watch"] = 50;
```

→ Output:- (Keys Sorted Alphabetically)

camera → 25
headphones → 50
laptop → 100
tablet → 120
watch → 50.

→ m.insert({"camera", 25});

→ m.emplace("mic", 25);

→ m.size()
→ m.clear()
→ m.empty().

→ m.erase("watch"); //remove by key.

→ m.find("tv");

→ m.count("camera");
↳ if found return 1.
otherwise 0.

→ m["tv"]
↳ returns the Value = 100.

Unordered Map :=

→ unordered_map <K, V>
→ no specific order (random order)
→ Keys must be unique.
→ Fastest avg time O(1)

unordered_map <String, int> um;

Same As Map
But Random Order.

Multi Map := (Sorted but allows duplicate keys).

→ multimap <K, V>
→ Stores key in sorted order.
→ Allow multiple values for the same key.
→ insert : O(log n).
→ find() returns iterator to the first matching key.

multimap <String, int> mm;

~~mm.push~~ mm["tv"] = 100;
mm["tv"] = 200;
mm["tv"] = 300;
mm.erase(mm.find("tv")); // erasing only the first occurrence of "tv".

↳ output

tv → 200
tv → 300

Set :- (Balanced BST)

→ Stores ~~quote~~ Unique Values in Sorted order (ascending).

```
Set <int> S;
```

- insert, emplace
- count
- erase
- find
- size, empty, erase

→ you need fast search / insert / delete

→ No Indexing O(log n).

```
Set <int> s;

s.insert (1);
s.insert (2);
s.insert (1); // duplicates ignored.
s.insert (3);
s.insert (4);
```

output :- 1 2 3 4

→ s.size() // 4 (Coz duplicate ignored).

→ s.count(3); // checks if exists 1 else 0.

```
→ auto it = s.find(3);
  if(it != s.end())
     cout << 3 found;
  else
     cout << 3 not found;
```

Multi Set :-

```
multiset <int> S;
```

→ Allows duplicate Value.

→ print all duplicate Values also.

→ Also give total size including duplicates.

s. 1 2 2 3 3 3 4 5

s.size() // 8

Unordered Set :-

→ No upper & lower-bound Concept

→ random order like unordered_map.

→ Duplicates ignored.

Lower Bound :—

Set stores unique & sorted values (BST).

→ Returns iterators to the FIRST elem $\geq x$.

→ works in $O(\log n)$

→ if no elem exists → returns s.end().

Set : 1, 2, 3, 5, 6, 7.

s.lower_bound(4); // 4 not found so first elem greater than 4 is 5, so output is 5. If 4 was present than Output 4.

s.lower_bound(10); // s.end().

Upper Bound :—

a	a	b	b	b	c	c	d
0	1	2	3	4	5	6	7

upper_bound('b') → greater than Key. Means ans is 5 (c). it can't be 'b'.

Set : 1, 2, 3, 5, 6

*(s.upper_bound(4)) // 5. → first elem $> x$.

Algorithms

Sorting :—

```
int arr[] = {1, 8, 5, 3};
int n = 4;
```

⇒ `Sort(arr, arr+n)` // Sort in ascending Order
// 1, 3, 5, 8

⇒ `Sort(arr, arr+n, greater<int>())` // Sort in descending order.
using greater<int>().
// 8, 5, 3, 1.

```
Vector<pair<int, int>> vec = {{3,1}, {2,1}, {7,1}, {5,2}};
```

⇒ `Sort(v.begin(), v.end());` // 2 1 (Sorted by first)
3 1
5 2
7 1

Custom Operator For PAIR

Sort by second elem
// [] it's a lamda (like mini function without a name)

```
auto comparator = [](const pair<int,int>& p1, const pair<int,int>& p2)
{
    return p1.second < p2.second; // Sort by second value.
};
```

⇒ `Sort(vec.begin(), vec.end(), comparator);`

output :— 3 1
2 1
7 1
5 2

Other Algos :—

- reverse() → when u need to flip array/vector quickly.
- next_permutation() → Solving permutation-based Prob (strings, numbers).
- max_element() / min_element() → To find global max/min.
- binary_search() → binary search.
- __builtin_popcount() → Bit manipulation, DP, set-bit problems.