C++ STL

(A) Containers

(1) Vector (dynamic in nature)

Ways to initialize vector

- vector <int> vec;
- vector <int> vec = {1, 2};
- vector <int> vec (3, 10) ⇒ | 10 | 10 | 10 |
  - size, value
- vector <int> vec2(vec1)

Functions in vector

- size & capacity

  vec.push_back(1) ⇒ | 1 | ⇒ vec.size() ⇒ 1

  vec.push_back(2) ⇒ | 1 | 2 | ⇒ vec.size() ⇒ 2

  vec.push_back(3) ⇒ | 1 | 2 | 3 | ⇒ vec.size() ⇒ 3

  ⇒ vec.capacity() ⇒ 4

- push_back & pop_back
  - end mein add karega, end se delete
- emplace_back
- at() or []

  | 1 | 2 | 3 | 4 | 5 | 6 |
  | 0 | 1 | 2 | 3 | 4 | 5 |

  vec[2] ⇒ 3

  vec.at(2) ⇒ 3

- front & back

  | 1 | 2 | 3 | 4 | 5 | 6 |
  | 0 | 1 | 2 | 3 | 4 | 5 |

  vec.front() ⇒ 1

  vec.back() ⇒ 6

- erase

  | 1 | 2 | 3 | 4 | 5 | 6 |
  | 0 | 1 | 2 | 3 | 4 | 5 |

  vec.erase(vec.begin())

  vec.erase(vec.begin()+2)

  vec.erase[start, end]
  - inclusive, exclusive

  erase changes size
  not capacity

- insert

  | 1 | 2 | 3 | 4 | 5 | 6 |
  | 0 | 1 | 2 | 3 | 4 | 5 |

  vec.insert(v.begin()+2, 100)

  | 1 | 2 | 100 | 3 | 4 | 5 | 6 |
  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

- clear

  size zero hoga not capacity

- empty

  returns 1 if empty
  0 if not

Iterators

1	2	3	4	5	

vec.begin — vec.end

*(vec.begin()) ⇒ 1

dereferencing first value

doesn't point at last index but at a garbage value

```
vector<int>::iterator it;
for(it=vec.begin(); it!=vec.end(); it++){
    cout << *(it) << endl;
}
```

forward goes like this

1 2 3 4 5

	1	2	3	4	5

vec.rend — vec.rbegin

no need to initialize iterator/define

```
for (auto it=vec.rbegin(); it!=vec.rend(); it++){
    cout << *(it) << endl;
}
```

5 4 3 2 1

(2) List (doubly linked list)

list <int> l

- push_back, push_front
- pop_back, pop_front
- emplace_back, emplace_front

l.push_back(1)

| 1 |

l.push_back(2)

| 1 | 2 |

l.push_front(3)

| 3 | 1 | 2 |

l.push_front(5)

| 5 | 3 | 1 | 2 |

(3) Deque (Double ended queue)

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

- push_back, push_front
- pop_back, pop_front
- emplace_back, emplace_front

* List & Deque difference

List ⇒ DLL ⇒ can't access randomly

Deque ⇒ dynamic array ⇒ can access element randomly

(4) Pair

pair <int, int> p = {3, 5};

pair <char, int> p = {'a', 1};

can access

p.first ⇒ 'a'

p.second ⇒ 1

pair <int, pair <char, int>> p = {1, {'a', 3}};

p.first ⇒ 1

p.second.first ⇒ 'a'

p.second.second = 3

* Diff between push_back & emplace_back

vec.push_back({4, 5}) ⇒ assumes you already have a vector and just inserts

vec.emplace_back(4, 5) ⇒ in place objects create

(5) Stack (LIFO)

stack<int> s

| 3 |
| 2 |
| 1 |

- push, emplace
- top ⇒ 3
- pop (removes top) 3 2 1
- empty (returns 1 if empty 0 if not)
- size
- swap (swaps s1 ke elements in s2)

(6) Queue (FIFO)

queue<int> q

front — rear

| 1 | 2 | 3 |

- push, emplace
- front ⇒ 1
- pop ⇒ removes from front 1 2 3
- size
- empty
- swap

(7) Priority Queue

priority_queue<int> q

priority_queue<int, vector<int>, greater<int>> q

functor

this will give reverse order priority queue (ulta) ascending order

if pq.push(5)
(2)
(10)
(4)

⇒

| 10 |
| 5 |
| 4 |
| 3 |

sorted order

- push, emplace → O(logn)
- top → O(1)
- pop → O(logn)
- size
- empty

(8) Map

(key, value)

map <string, int> m

m[key] = value // to insert, change new value

unique

key	value
"tv"	100
"laptop"	50
"headphone"	100

if we print this then alphabetically sorted order

mein aate hai values

∴ headphones then laptop then tv

- insert, emplace
- count (returns no of keys)
- erase
- find
- size, empty

(i) Multimap

→ can store multiple keys (duplicates)

multimap<string, int> m

(ii) Unordered map

→ will not store in sorted order

unordered_map<string, int> m

sirf isko TC is O(1)

baaki sab kaa O(logn)

(9) Set

set <int> s

sirf unique values & sorted

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

1 2
3 4
5 6

ignores duplicates

sab ka TC ⇒ O(logn)

- lower_bound (4) ⇒ 4 if 4 exists, else 5 if didn't exists
- upper_bound (4) ⇒ 5

(i) Multiset

→ can store multiple keys (duplicates)

multiset <int> s

(ii) Unordered set

→ will not store in sorted order

unordered_set <int> s

(B) Algorithms

(1) Sorting

sort(arr, arr+n) ⇒ ascending

sort(arr, arr+n, greater<int>()) ⇒ descending order

sort(v.begin(), v.end())

eg: vector <pair <int, int>> vec = {{3, 1}, {2, 3}, {7, 1}, {5, 2}}

sort(vec.begin(), vec.end())

⇒ 2, 1
3, 1
5, 2
7, 1

sorted on first value

We can build our custom logic if we want to sort on second value

we need to define custom comparator

make a bool function which will return true if p1.second < p2.second then do

sort(vec.begin(), vec.end(), comparator)

⇒ 3 1
2 1
7 1
5 2

(2) Reverse

reverse(vec.begin(), vec.end())

5 4 3 2 1

reverse(vec.begin()+1, vec.begin()+2)

1 3 2 4 5

(3) Next permutation

next_permutation(vec.begin(), vec.end())

s = "abc"

abc
acb ✓
bac
bca
cab
cba

prev perm bhi bata hai

(4) Swap, min, max

(5) max & min element

max_element(vec.begin(), vec.end())

min_element(vec.begin(), vec.end())

(6) Binary search

binary_search(vec.begin(), vec.end(), target)

(7) Count set bits

int n = 15

4 bytes → 32 bits

returns kitne 1s in bit format

__builtin_popcount() → int

__builtin_popcountl() → long int

__builtin_popcountll() → long long int