STL: Standard temperory Library.

We Use to pre-define Containers; No Need to write long Codes for it

#include<bits/stdc++.h>.

using namespace std;

to Input We use cin>>a

to Output " " cout<<a.

std :: cin >> a;

std :: cout<<a;
No space

Algorithms Funct^n's

Containers Iterators

Pairs : Part of Utility library

Syntax: pair<int,int> P = {1,3}; First, second.

↳, can be different.

cout << p.first << " " << p.second; for<<end

Nested Pairs

③ pair<int, pair<int,int>> p = {1,{4,3}};

Similar to structures. Dot-Operator. Initialisation.

cout << p.second.first.

Usage of Array in Pair; ↳ Can be treated ass an array; 0 1 2

pair<int,int> arr[] = {{1,3}, {2,5}, {4,3}};

cout << arr[1].first;

Containers!

Vectors! A Container which is dynamic memory allocation. in Nature

↳ we can add space

funct's:

Syntax! Vector<int> v; ↳ creates An Empty Container. {}

Nogap

funct's!

v.push_back(1); ⇒ {1}.

v.emplace_back(2); ⇒ {1,2}

↳ Faster than Push back.

STL: Standard temperory Library.

We Use to pre-define Containers: → No Need to write long codes for it

#include<bits/stdc++.h>.

using namespace std;

To Input We use cin>>a

To Output " " cout<<a.

std::cin>>a;

std::cout<<a; (No space)

Algorithms Funct^n's

Containers Iterators

Pairs: Part of Utility library

Syntax: pair<int,int> P = {1,3}; (First, second)

↳ Can be different.

cout<< p.first <<" " << p.second; (Cout<<end)

③ pair<int, pair<int,int>> p = {1, {4,3}}; Nested Pairs.

Similar to structures. Dot-Operator. Initialisation.

cout << p.second.first.

Usage of Array in Pair; ↳ Can be treated ass an array.

pair<int,int> arr[] = {{1,3}, {2,5}, {4,3}.};

cout << arr[i].first;

0 1 2

Containers:

Vectors: A Container which is dynamic memory allocation. in Nature

↳ We can add space

Funct^n's:

Syntax: Vector<int> v; (No gap) → creates An Empty Container. {}

Funct^n's:

v.push_back(1); ⇒ {1}.

v.emplace_back(2); ⇒ {1,2}

↳ Faster than Push back.

We Can define a vector of pair data type.

It * v.push_back ({1,2}); => {{1,2}};
v.emplace_back (2,3); => {{1,2}, 2,3}};
↳ No Need Curli braces.

Syntax: (Pre-Initialisation).

vector<int> v(5,100); => {100,100,100,100,100}
↳size ↳ Number to be filled

" , v(5); => {, , , , ,} or Compiler
↳ May be '0' or garbage based

vector<int> v1(v); ↳ v1 = {100,100,100,100,100}

v.push_back(1); => {100,100,100,100,100,1}

To Access Elements in the Vector:

Iterators

Syntax: vector<int>::iterator it = v.begin(); v = {100, 100, 2, 3, 5}.
↳ Type of Iterator
it → v[0] (memory)

=> cout << *(it) << " "; => (100).

it = it + 2; (2).

cout << *(it) << " "; => (2).

Types of Iterator

Iterators

- v.end(); => {, , , ,}
- v.rend();
- v.rbegin();

Iterator similar to pointer.

We Can see in the form of Array;

cout << v[0] (or) v.at[0];

Element at last Position : v.back(); {, , , }

it → it++

Use of Loop Using Iterator:

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

→ Initialisation Inside loop

(or)

```
for(auto it:v){
    cout<<it<<" ";
}
```

→ Acc to data data type is Automatically Assigned.

for each loop

Deletion in an Vector:

{10,20,30}.

v.erase(v.begin+1); ⇒ {10,30}. reshuffled.

→ Iterator to erase One Data

v.erase(start, End Address); {10,20,30,40,50}
0 1 2 3 4

↳ [start, end) ⇒ {10,30,50}

Takes More time

v.insert(v.begin()+1, 20); ⇒ {10,20,20,30}

v.insert(v.begin()+1, 4, 10); ⇒ {10,10,10,10,10,20,30}.

↳ Location ↳ No. of elements Element

v.insert(v.begin(), v1.begin(), v1.end());

v1 = {40,50}, v = {10,10,10} → starting & Ending Address.

⇒ v = {40,50,10,10,10}.

v.size(); Size of Vector = No. of Elements (not Bytes).

v.pop_back(); deletes last element.

{} ⇐ v.clear(); : Entire Vector is erased

v1.swap(v2); ⇒ v1 ↔ v2

t = v1; v1 = v2; v2 = t;

v.empty();

If v = {}

v = {1,} or Anything else F.

.clear(); : Erases whole Container.

v.count(initial, final, element) → returns int

↑ Address ↑ ↳ No. of repetitions.

List: It gives you front operations As Well.

→ Dynamic in Nature

Syntax : `list<int> l;`

```
l.push_back(3);        {3}
l.emplace_back(4);     {3,4}
l.push_front(5);       {5,3,4}
l.emplace_front();
```

`deque<int> dq;`

push_back

dq = {1,2,3}

`dq.emplace_front(6);` {6,1,2,3}

pop_front : removing 1st element from front

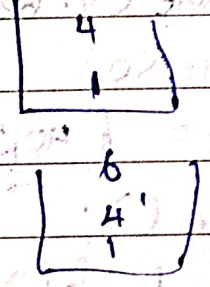
`dq.back();` : Last element

`dq.front();` : front (1st)

Stack: LIFO : Last in first Out

Syntax : `stack<int> st;`

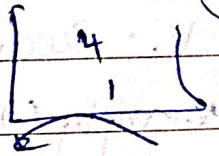
```
st.push(1);
st.push(4);
st.emplace(6)
```



{6,4,1}

⇒ `cout << st.top();` → (6)

`st.pop();` → Deletion of 6



`cout<< st.top();` → (4)

`st.swap(s2);` → st ⇌ s2

t = s2

s2 = st

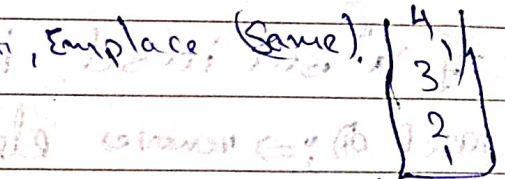
st = t

We can access size & last input element

Queue: First In First Out

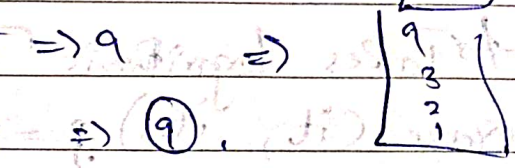
queue<int> q;

Push, Emplace (Same).



q.back() += 5;

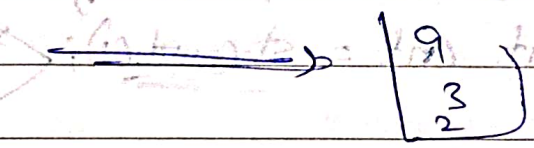
→ 4 // 4+5 ⇒ 9 ⇒



q.front = 1st element

cout << q.back(); ⇒ (9).

q.pop();



Priority Queue: High Value at the top.

Syntax: priority_queue <int> pq;

pq.push(5); ⇒ {5}

" " (2); ⇒ {5,2}

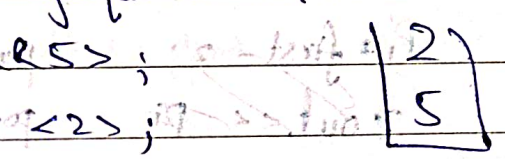
" " (8) ⇒ {8,5,2}.

} General (Max heap).

syntax for Min Wise; priority_queue<int, vector<int>, greater<int>> pq;

pq.push(5);

" " (2);



Min Heap

Time Complexity:

push → log n

v.size(); top → O(1) ?

pop → log n

Set: Stores in Sorted, Order. Unique,

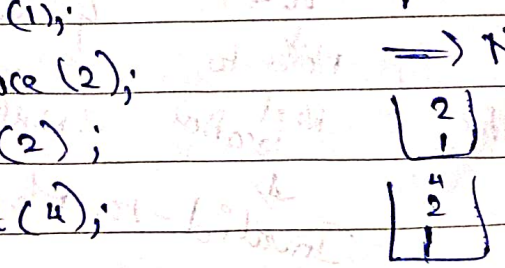
Syntax: set <int> st;

st.insert(1);

st.emplace(2);

st.insert(2);

st.insert(4);

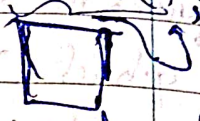


⇒ No it won't store

auto it = st.find(3);

→ It returns Iterator which points to 3.

If 'a' is'nt in set it returns st.end();



st.erase(it); (or) st.erase(a); => removes element a

→ Takes logarithmic time. (Address elements)

st.erase(it, it+1); => [it, it+1) erase.

int cnt = st.count(1); → 1 if exists
→ 0 if doesn't exist

auto it = st.lower_bound(2); } Based on
= st.upper_bound(3); } Binary Search

Pairs:

Syntax: pair<int, string> p; } Initialisation
p = make_pair(2, "abc"); }

We can Compare pairs.

p1 = p;

p1.first = 3; } push_back, pop_back } Time Complexity: O(1)

cout << p1;

Nesting in Vectors:

We can use vectors & pairs as parameters in v.push_back();

→ we use temp vectors for taking in Nested Vectors. But we No Need that in Array of Vectors.

~~it ++~~ it++ → Moves to Next Iterator. | it+1 → Moves to Next location } No difference in vectors Next Iterator = Next location

Difference

→ Invalid to Maps & sets.

v[i].push_back(n);

→ Doesn't exist for 1st One.

for Nested Vector & pair.

→ (*it).first & (*it). 2nd Dot

(or) (it→first) Arrow.

reference (C++)

Maps: Data structure. (Red black trees) → w.r.t key.

key → value. Can be Anything. Stored in Sorted Ordered & Unique way.

int string

1 abc Non-Contiguous stored.

Syntax: map <__, __> m;

Keys: Containers which can be compared directly.

m[1] = "abc"; Time Complexity: O(log(n)).

m[3] = "567";

m.insert({4, bcd});

Ex $S_1 < S_2$. cout << (it→first) << (it→second);

(or) (*it).first (*it).second

⇒ m[6]; By default It maps to Null string for int, float, double ⇒ '0'

If we reinitialise any mapping again it doesn't take it it Store Once again it Only changes the value(maped) of that mapping.

→ to access an element in Map Time Complexity: O(log n)

m.find(3); → returns Iterator of 3 (or) m.end();

→ Time Complexity: O(log(n)). n; Map size

m.erase(); Works On Iterator & key.

m.clear(); Clear whole Map.

Date :

When we Insert keys it Compares with previous keys & get Stored in Sorted order → so O(log n) + Comparision of keys.

⇒ ~~S~~ s.size() * O(log(n)).

Difference b/w Maps & UnOrdered Maps:

Inbuilt implementation → Maps : Red black trees
→ Unordered Maps : Hash Tables

O(log(n)) Map ← O(1) unordered Map → Time Complexity
→ Avg

Valid Key data type.

Unordered Map:

Syntax: unOrdered_map <int, string> m;

m.find(), m.erase, m.end() : O(1). Time compl

Hash function?

Valid Keys → Maps : Any data type
→ Unordered Map : Only Certain data type

int, float, double, string(✓).

Based on hash funct"

Multimap: multimap < , > m;

→ Similar to map: red black trees.

we can also insert Duplicates ⇐ No UNIQUE Order multiple time allo

Ex: map<int, vector<string>>;

Sets: Maps with Only Keys.

To access elements We use .find( );

* Erase ( ) it or element as input → Only one element

Clear ( ) → Total Container clear.

Internal implentation: Red Black trees.

erase (it) : Time Complexity: O(1).

Date :

unordered Set: → Unique. Input No two input same stored ~~hash~~ tables.
Only one of that

Syntax: unordered_set < > S; Internal implementation

3 differences. ← Similar to Unordered Maps

Multiset:

multiset < > S;

* S.Erase("abc"); all duplicates get erased
* S.erase(it); Value only type on that iterator get erased.

Similarly for Set → Pair Comparision
1st - 1st element then 2nd - 2nd element
→ which one low that is 1st kept.

Maps: m[{f, l}]
pair as key.

⟹ When we want to sort both key and value we use map & set.

Balanced Parenthesis : (( ))

~~Bob~~ → Using stack (push)
Pop if we get close Parenthesis & add if we had open parenthesis.

In built Sorting funct^n: (Introsort)

Syntax: sort(a , a+n);
initial, final address

1st → Quick Sort
2nd → Heap(?) Sort
3rd → Insertion Sort
order
} Mixture of Sort

Comparator funct^n:

→ comparator funct^n.
↳ used for vector pair due to complexity in normal Code.

```
bool ___ ( int a, int b ) {
    if (a > b) return true;
    return false.
}
sort ( a.begin (), a.end (), should_i_swap);
```

If we need to swap return False.

Due to present in sort funct^n → Comparator funct^n behavoir changes

greater Comparator funct^n is inbuilt funct^n
↳ To get in decreasing Order.

Date : _____

Upper bound & Lower bound Use when the data type (container) is in Sorted Order

↓ Given 'a' finds a element Just greater than a.

↓ Give 'a' It finds a(or) Just greater than a.

→ To decrease time Complexity.

→ returns an Iterator.

Syntax: *ptr = lower_bound (initial, final, element)

addresses → ints/strings/int etc.

" : upper_bound (" , " , element).

Time Complexity : log(n). for vectors etc.

Identifier, belong to map, set.

We use. lower_bound (" , ") ; Only for sets & Maps.

for Map : it takes based on Keys.

---

max/min_element (initial, final address) = [initial, final)

→ returns an iterator for given in which min value is present.

accumulate (initial, final address, int) = [initial, final)

→ initial sum.

→ returns sum of all elements of vector v.

find (initial, final Address, element).

exists return iterator of pointing

else " " of v.end().

Date :

reverse (initial, final address); reverses vector

Lambda funct^n's: all_of none_of any_of returns T or F

for lambda funct^n Syntax : [] (arguments) {return ---; } (2). → Input funct.

for all_of!

And Operator all_of (v.begin, v.end(), [] (int x {return x>0;}) → returns 0,1

→ every element should satisfy every element should satisfy.

OR operator any_of (v.begin, v.end(), [] {int x {r---}}); → returns 0, 1.

→ if any one element satisfies. returns 1

!(OR) none_of (v.begin, v.end(), [] ---); → is 0, 1

if all false returns 1.