DSA

C++

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a;
    return 0;
}
```

→ this has all the libraries so we don't need to add 1 by 1.

add the std meaning (without this we have to write std everytime)

This returns value 0

```
void print(){
    cout << " raj ";
}
```

← doesn't return anything unless print called.

STL

divided into 4 parts :-

i) Algorithms
ii) Containers
iii) Functions
iv) Iterators.

pair (part of utility library)

stores a couple of datas in a variable.

Eg
```
pair<int, int> p = {1, 3};
cout << p.first << " " << p.second;
```

Eg
```
pair<int, pair<int, int>> p = {1, {3, 4}};
cout << p.first << " " << p.second.second << " " << p.second.first;
```

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

cout << arr[1].second;

↳ 5.

Containers

1) Vectors

To ~~declare~~ store 5 values, we would have created arr[5], where we store 5 values.
But to add a value, we would need to change it to arr[6] but arrays are const. in size. So, no modification

So, in vector we can change.

Eg vector <int> v; → this creates an empty container.

v.push_back(1); → creates a space and adds one into it.

v.emplace_back(2); → dynamically creates a space & adds 2.

similar but emplace_back is faster.

To define vector of pair datatype.

Eg vector<pair<int, int>> vec;

To take a pair in vector,
For push_back, → u need to write it in {1,2}
But
emplace_back automatically assumes them as a pair.

So,
v.push_back({1,2});
v.emplace_back(1,2);

To declare a container for a same no. but n no. of times,

vector<int> v(5,100); → {100,100,100,100,100}

vector<int> v1(5); {0,0,0,0,0}

To copy v1 into v2,
vector<int> v2(v1);

Access elements of a Vector :-

1) v[0] or v.at(0) → same as arrays
↳ mostly used

2) iterator → points to the memory address.

vector<int>::iterator it = v.begin();
↓
this creates a memory address not an element

{20, 10, 15, 6, 7}

Eg → vector<int>:: iterator it v.begin();
it++;
cout << *it << " ";

begin points out to 20

& then it++ makes it to 10

→ vector<int>::iterator it = v.end();
it--;

{10, 20, 30, 40}

points to element after 40

→ vector<int>::iterator it = v.rend();

{10, 20, 30, 40}

points to element before 10

→ vector<int>:: iterator it = v.begin();

it++;

{10, 20, 30, 40}

3) {10, 20, 30}

v.back();

How to print the entire vector :-

1)
```
for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
    cout << *(it) << " ";
}
```

Short form — auto → automatically assigns a datatype to a variable acc. to the data.

2)
```
int a = 5;
auto a = 5;        a → int.

for (auto it = v.begin(); it != v.end(); it++) {
    cout << *(it) << " ";
}
```

3)
```
for (auto it : v) {
    cout << it << " ";
}
```
→ this doesn't declare the vector v. This just declare the type of it.

Output 10 20 30

Delete an element :-

1) `v.erase(v.begin() + 1);`

Provide the address of element

{10, 20, 30, 40}
↑ delete this
{10, 30, 40}

2) .erase (starting address, end address after the element)

v.erase (v.begin() + 2, v.begin() + 4);

{10, 20, 12, 23, 35}

deleted

{10, 20, 35}

Insert function :-

vector <int> v (2, 100); // {100, 100}

v.insert (v.begin(), 300); // {300, 100, 100}

v.insert (v.begin() + 1, 2, 10); // {300, 10, 10, 100, 100}

posi — how many — what

How to insert a vector into another vector :-

vector <int> copy (2, 50); // {50, 50}

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

// {50, 50, 300, 10, 10, 100, 100}

v.size(); // size of a vector / no. of elements.

v.pop_back(); // {10,20} → {10}

v1 → {10,20}
v2 → {30,40}
v1.swap(v2); v1 → {30,40}
v2 → {10,20}

v.clear(); //erases entire vector.

v.empty(); // if vector has 1 or more → False
no element → True.

List
Similar to vector but with options to work on the front elements.

list<int> ls;
ls.push_back(2); // {2}
ls.emplace_back(4); // {2,4}

ls.push_front(5); // {5,2,4};
ls.emplace_front(6); // {6,5,2,4};

push_front → first creates a new object then passes it to the function.
emplace_front → passes the constructor arguments for the object directly to the function.

Rest functions similar to vector → begin, end, rend, rbegin, clear, insert, size, swap

Deque

Container similar to vector.

```
deque <int> dq;
dq.push_back(1);          // {1}

dq.emplace_back(2);       // {1, 2}
dq.push_front(4);         // {4, 1, 2}
dq.emplace_front(3);      // {3, 4, 1, 2}

dq.pop_back();            // {3, 4, 1}
dq.pop_front();           // {4, 1}

dq.back();
dq.front();
```

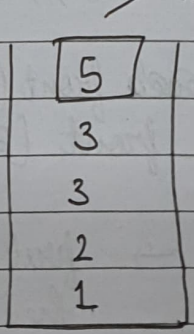
Rest same as vector → begin, end, rend, rbegin, clear, insert, size, swap.

Stack

LIFO → last in first out.

```
stack <int> st;
st.push(1);
st.push(2);
st.push(3);
st.push(3);
st.emplace(5);

cout << st.top(5);        // prints 5
```



* st[2] is invalid → there's no indexing in stacks.

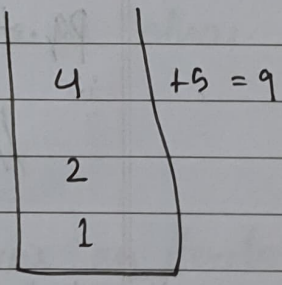
In, stack there are only 3 generic function → push, pop, top.

```
st.pop();        // new st → {3, 3, 2, 1}
st.top();        // 3
now st.size();   // 4
st.empty();      // false.
stack<int> st1, st2;
st1.swap(st2);
```

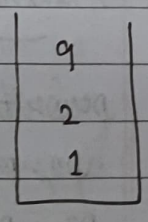
Complexity of stack → Everything happens in const. time
O(1)

Queue (O(1) → happens in const. time)
FIFO → first in first out

```
queue<int> q;
q.push(1);      // {1}
q.push(2);      // {1,2}
q.emplace(4);   // {1,2,4}
```




```
q.back() += 5
q.back();    // 9.
q.front();   // 1.
q.pop();     // {2,9}
q.front();   // 2.
```



// size, swap, empty same as stack

Priority Queue :-

The maximum or largest element stays at the top.

using character, largest character stays at top
strings, lexicographic largest string at top.

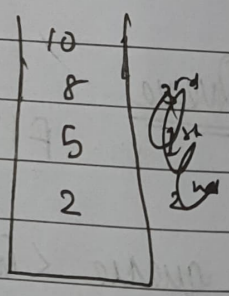
```
priority_queue <int> pq;

pq.push(5);        // {5}
pq.push(2);        // {5,2}
pq.push(8);        // {8,5,2}
pq.emplace(10);    // {10,8,5,2}

pq.top();          // 10
pq.pop();          // {8,5,2}

pq.top();          // 8

// size, swap, empty function same as other
```



* Minimum Heap :-

```
priority_queue <int, vector<int>, greater<int>> pq;
pq.push(5);        // {5}
pq.push(2);        // {2,5}
   "   "(8);       // {2,5,8}
       (10);       // {2,5,8,10}

pq.top();          // 2
```

Time complexity → Priority Queue

push - log n
pop - log n
top - O(1)

Set (Stores in a tree format)
Stores everything in a sorted order. & stores unique.

```
set<int> st;
st.insert(1);      // {1}
st.emplace(2);     // {1,2}
st.insert(2);      // {1,2}        → set is unique no element repeated
st.insert(4);      // {1,2,4}
st.insert(3);      // {1,2,3,4} → sorted.
```

begin, rend, end, rbegin, size, swap, empty same as others.

{1, 2, 3, 4, 5}

```
auto it = st.find(3);   // returns an iterator that points to 3
auto it = st.find(6);   // as 6 not found so, returns st.end();
                        // meaning points to element after 6.
st.erase(5);            → (takes logarithmic time)
                        // deletes 5 & maintains sorted order
st.erase(iterator);     → (takes constant time)
   ↳ or address
```

ds, set is unique.

```
int cnt = st.count(1);
```

exists → 1
not exists → 0

To erase elements.

{1, 2, 3, 4, 5}

```
auto it1 = st.find(2);
auto it2 = st.find(4);

st.erase(it1, it2);   // {1, 4, 5}
```

* lower_bound & upper_bound works in same way as in vector.

```
auto it = st.lower_bound(2);
auto it = st.upper_bound(3);
```

lower_bound points to the immediate greater element if element is not found

a[] = {1, 4, 5, 6, 9, 9}

~~int index~~ lower_bound(a, a+n, 4)

~~greater element after 4~~ points to 4 as 4 present

to get the position/index.

```
int index = lower_bound(a, a+n, 4) - a;
// 2
```

int ind = lower_bound(a, a+n, 7) - a ; // 4 → to 9 as not present
" " " " (a, a+n, 10) - a ; // 6.

Syntax
int index = lower_bound (a.begin(), a.end(), a)
- a.begin();

Upper_bound
Always gives an iterator pointing to next bigger element.

a[] = {1, 2, 3, 4, 5, 6}

int ind = upper_bound (a, a+n, 4) - a ;
// iterator point to 5

For 7 // iterator pointing to element after 6
// 6.

In set,
everything happens in logarithmic time complexity
log(n)

Multiset (Obeys Sorted) (Not Unique)

Everything same as set
but stores duplicate element also.

multiset < int > ms ;
ms.insert (1); // {1}
" {1, 1}
" {1, 1, 1}

ms.erase(1); // all 1s erased

int cnt = ms.count(1);

// only erase a single element.
ms.erase(ms.find(1)); // first occurrence of 1.

// ~~multiple element~~
ms.erase(ms.find(1), ms.find(1)+2);

{1, *, 1}
{1, 1}

or ms.erase(start, right after end).

Unordered set

Unique, but not sorted.
In most case, time complexity is O(1).

** lower_bound & upper_bound doesn't work.
* rest same as set.
* has better complexity → O(1)
except when collision happens.
Rarely, when high amount of data.
~~Only~~, Once in a million maybe.

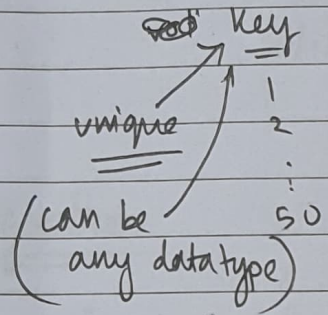
Map → stores unique keys in sorted order.

roll no.

Eg raj → 28
→ 25
→ 31.

how to store all of them

Map assumes roll no. as key & name as value




```
map <int, int > mpp;
```
key, value

```
map <int , pair <int , int >> mpp ;
```
key, value

```
map < pair <int, int > , int > mpp ;
```
key, value

```
mpp [1] = 2 ;
```
{1, 2}
k v

```
mpp.emplace ({3, 1});
```
{3, 1}
k v

```
mpp.insert ({2, 4});
```
{2, 4}

```
mpp [{2, 3}] = 10 ;
```
{{2, 3}, 10}
k v

Map
{1, 2}
{2, 4}
{3, 1}
sorted keys

How to traverse in a map :-

```
for (auto it : map) {
        cout << it.first << "  " << it.second << endl;
    }
```

[{1,2}, {2,4}, {3,1}]

it → 1st it → 2nd

1	2
2	4
3	1

```
cout << mpp[1];    // 2.
cout << mpp[5];    // null or 0.
```

To find address/iterator

```
auto it = mpp.find(3);
cout << *(it).second;    // to access the value.
```

* → gives element.

```
auto it = mpp.find(5);   # → mpp.end()
```

```
auto it = mpp.lower_bound(2);
auto it = mpp.upper_bound(3);   // Syntax
```

// erase, swap, empty, size same as others.

Multimap

Same as map.
But can store duplicate keys.
Sorted.

mpp[key] can't be used here.

Unordered Map

Same as set, Unique
Not sorted.

Algorithms

To sort,

a[] → {1, 5, 3, 2}

sort (a, a+n); } can sort any container except map.

1st posi → last position

sort (v.begin(), v.end()); //vector.

To sort a particular portion :-

{1, 3, 5, 2}

↳ to sort this

sort (a+2, a+4);

To sort in descending order :-

```
sort (a, a+n, greater <int>);
```

To create a custom sorting :-

Eg

```
pair <int, int> a[] = {{1,2}, {2,1}, {4,1}};

// sort acc. to 2nd element.
// if same, sort acc to 1st element
            but in descending order

sort (a, a+n, comp);
      ↓    ↓    ↳ self-written comparator
    start  end      ↳ boolean //

bool comp (pair <int, int> p1, pair <int, int> p2) {
    if (p1.second < p2.second) return True;
    if (p1.second > p2.second) return False;
      // same
    if (p1.first > p2.first) return True;
    return False;
}
```

__builtin_popcount();

```
int num = 7;
int cnt = __builtin_popcount();
```

7 in binary is 111

So, 00000 ... 111 (32 bit)

__builtin_popcount() returns no. of 1's.

So, 3.

for 6 → 2.

```
long long num = 16572564219;
int cnt = __builtin_popcountll();
```

next_permutations();

```
string s = "231";
do {
    cout << s << endl;
} while (next_permutations(s.begin(), s.end()));
int maxi = *max_element(a, a+n);
}
```

Prints = 231
312

For = 123
132
213
231

max_element ( ) ;

int maxi = *max_element(a, a+n);

~~or * max_element~~

min_element ()

If 231, next_permutation

If u want all permutations.

add sort(s.begin(), s.end());

after string s = "";