#include <bits/stdc++.h>

using namespace std;

namespace raj

{

    int val = 20;

}

struct node

{

    string s;

    int num;

    double d;

    char x;

    node(string s\_, int num\_, double d\_, char x\_)

    {

        s = s\_;

        num = num\_;

        d = d\_;

        x = x\_;

    }

};

int main()

{

    cout << raj::val << endl;

    node value = node("suresh", 100, 10.00, '\0');

    return 0;

}

/\*

Types of STL Container in C++

In C++, there are generally 3 kinds of STL containers:-

1. Sequential Containers

2. Associative Containers

3. Unordered Associative Containers

\*/

/\*

Types of Sequential Containers:-

1.Array

2.Vector

3.Deque

4.List

5.Forward List

\*/

1. Array - :

    SYNTAX of array container : array<object\_type, array\_size> array\_name;

Example : -array<int, 3> val = {1, 2, 3};

------- fill() function ----------

    This method assigns the given value to every element of the array.

    Example :

    array<int, 5> myarray;

    myarray.fill(10); -> {10,10,10,10,10}

------- at() function ------------

This method returns value in the array at the given index. If the given index is greater than the array size, out\_of\_range exception is thrown.

 array<int,10> array1 = {1,2,3,4,5,6,7,8,9};

    cout << array1.at(2)     // prints 3

    cout << array1.at(4)     // prints 5

    To print entire array  :-

    for(int i=0;i<10;i++){

        cout<<array1.at(i)<<" ";

    }

-------- front() function --------

This method returns the first element in the array.

array\_name.front();

-------- back() function ----------

This method returns the last element in the array. The point to note here is that if the array is not completely filled, back() will return the rightmost element in the array.

array\_name.back();

-------- swap() function ---------

This method swaps the content of two arrays of same type and same size. It swaps index wise, thus element of index i of first array will be swapped with the element of index i of the second array.

    array<int,8> a = {1,2,3,4,5,6,7,8};

    array<int,8> b = {8,7,6,5,4,3,2,1};

    a.swap(b)  // swaps array a and b

-------- empty() function --------

This method can be used to check whether the array is empty or not.

Syntax: array\_name.empty(), returns true if array is empty else return false.

--------- size() function ---------

This method returns the number of element present in the array.

array\_name.size();

-------- max\_size() function --------

This method returns the maximum size of the array.

array\_name.max\_size();

-------------------------- Iterators -------------------------------------

1. begin(), cbegin()    returns an iterator to the beginning

2 .end(), cend()        returns an iterator to the end

3. rbegin()             returns a reverse iterator to the beginning

4. rend()               returns a reverse iterator to the end

Example -

array<int,5> arr = {1,2,3,4,5};

for(auto it: arr.begin();it!=arr.end();it++){

    cout<< \*it <<" ";

}

output - 1 2 3 4 5

for(auto it: arr.end()-1;it>=arr.begin();it--){

    cout<< \*it <<" ";

}

output - 5 4 3 2 1

for(auto it: arr.rbegin();it!=arr.rend();it++){

    cout<< \*it <<" ";

}

output - 5 4 3 2 1

for(auto it: arr){  // for each loop

    cout<< it << " ";

}

-------------------------VECTOR-----------------------------

Vectors in C++ are sequence containers representing arrays that can change their size during runtime. They use contiguous storage locations for their elements just as efficiently as in arrays.

SYNTAX :- vector< object\_type > vector\_name;

Example - vector<int> vect; //create a blank vector

          vector<string> v {"Pankaj" ,"The" ,"Java" ,"Coder"};

          vector<string> v(4 , "test");   //Initialize a vector with one element a certain number of times

          vector<int> vec1(4,0); -> {0,0,0,0}

          vector<int> vec2(4,10); -> {10,10,10,10}

          copy the entire vec2 to vec3

          vector<int> vec3(vec2);

          vector<int> vec3(vec2.begin(), vec2.end());

-------- push\_back() function ----------

It is used for inserting an element at the end of the vector.

vector<int>  v;

    v.push\_back(1);  //insert 1 at the back of v

    v.push\_back(2);  //insert 2 at the back of v

    v.push\_back(4);  //insert 3 at the back of v

    push\_back() and emplace\_back() are identical but

    emplace\_back() takes lesser time than push\_back

--------- insert() function -----------

insert(itr, element) method inserts the element in vector before the position pointed by iterator itr.

--------- pop\_back() function ------------

pop\_back() is used to remove the last element from the vector. It reduces the size of the vector by one.

vector<int> v1 {10,20,30,40};

    v1.pop\_back();   // remove 40 from the v1

--------- erase() function ---------

erase(itr\_pos) removes the element pointed by the iterator itr\_pos. erase method can also be overloaded with an extra iterator specifying the end point of the range to be removed, i.e erase(itr\_start, itr\_end).

vecto<int>v1 {10,20,30,40};

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

    v.erase(it);   //removes first element from the vector

    v.erase(v1.begin(), v1.end() - 2 )

    /\*removes all the elements except last two \*/

    for(it = v.begin(); it != v.end(); it++)

    {

        cout << \*it <<" ";   // for printing the vector

    }

------- clear() function ----------

This method clears the whole vector, removes all the elements from the vector but do not delete the vector.

SYNTAX: clear()

For a vector v, v.clear() will clear it, but not delete it.

------- front() and back() functions -----------

vector\_name.front() retuns the element at the front of the vector (i.e. leftmost element).

While vector\_name.back() returns the element at the back of the vector (i.e. rightmost element).

------- at() function -----------

vector\_name.at(i) returns the element at ith index in the vector vector\_name.

-------- swap() function ----------

This method interchanges value of two vectors.

v1.swap(v2), this will swap the values of the two vectors.

------ To Defining 2D vector -------

vector<vector<int>> vec;

vector<int> v1;

v1.push\_back(1);

v1.push\_back(2);

vector<int> v2;

v2.push\_back(10);

v2.push\_back(23);

v2.push\_back(24);

vector<int> v3;

v3.push\_back(11);

v3.push\_back(12);

vec.push\_back(v1);

vec.push\_back(v2);

vec.push\_back(v3);

// it is vector itself

    for(auto vctr: vec) {

        for(auto it: vctr) {

            cout << it << " ";

        }

        cout << endl;

    }

    for(int i = 0;i<vec.size();i++) {

        for(int j = 0;j<vec[i].size();j++) {

            cout <<  vec[i][j] << " ";

        }

        cout << endl;

    }

    // define 10 x 20

    vector<vector<int>> vec(10, vector<int> (20, 0));

    vec.push\_back(vector<int>(20, 0));

    cout << vec.size() << endl; // 11 prints

    vec[2].push\_back(1);

    vector<int> arr[4];

    arr[1].push\_back(0);

    // 10 x 20 x 30 // int arr[10][20][30]

    vector<vector<vector<int>>> vec(10, vector<vector<int>> vec(20, vector<int> (30, 0));)

------------------------- SET -----------------------------------

Sets are containers that store unique elements following a specific order.

Syntax: set<datatype> setname;

Example:-

set<int> val; // defining an empty set

set<int> val = {6, 10, 5, 1}; // defining a set with values

Properties:

Storing order : The set stores the elements in sorted order.

Values Characteristics : All the elements in a set have unique values.

Values Nature : The value of the element cannot be modified once it is added to the set, though it is possible to remove and then add the modified value of that element. Thus, the values are immutable.

Search Technique : Sets follow the Binary search tree implementation.

Arranging order : The values in a set are unindexed.

Some Basic Functions Related To Sets

begin() :- Returns an iterator to the first element of the set .

end() :- Returns an iterator to the theoretical element that follows last element in the set .

Empty() :- check wheather the set is empty or not .

max\_size() :-Returns the maximum number of element that set can hold .

Size() :- Returns the number of elements in the set .

To store the elements in decreasing order in the set.

By default, elements are stored in ascending sorted order in the set. To store the elements in descending order, we have to use the given declaration syntax.

    // declaration syntax of the set to store the elements in decreasing sorted order.

     set<int,greater<int>> s;

------- insert() function --------

 Syntax:

set\_name.insert(element);

Example-

set<int> mySet;

   mySet.insert(10);

   mySet.insert(20);

   mySet.insert(30);

   mySet.insert(40);

   mySet.insert(50);

------- emplace() function -------------

Syntax:

setname.emplace(value);

Example-

 set<int> myset{};

    myset.emplace(1);

    myset.emplace(2);

    myset.emplace(3);

    myset.emplace(4);

    myset.emplace(5);

------- find() --------

Returns an iterator pointing to the element, if the element is found else return an iterator

pointing to the end of the set.

Syntax:

set\_name.find( key ) ;

------- erase() ----------

this method is used to delete the elements from the set.

Syntax:

set\_name.erase( starting\_iterator , ending\_iterator ) ;

                    or

set\_name.erase(key); delete the specific key from the set.

--------- Unordered\_Set ---------------

Unordered\_set can stores elements in any order, with no specified order for storing elements. It generally depends upon the machine that you are using.

Stores unique elements, no duplicates allowed.

unordered\_set uses the hash table for storing the element.

Note: All other properties are the same as that of the set.

--------------------MULTISET------------------

Multisets in C++ are containers that are very similar to sets.

Unlike sets, multisets can store duplicate elements in a sorted manner.

The elements inside the multiset cannot be changed, once they are added to the multiset, they can only be inserted or deleted. A multiset is present in #include<set> header file.

The elements inside the multiset can be accessed using iterators.

Syntax:

multiset <data\_type> name = { initial\_values };

Note: By default the multiset stores values in non-decreasing order. To store the values in non-increasing order, we use an inbuilt comparator function

multiset <data\_type, greater <data\_type>> name;

Example -

//initializes a multiset of size 0 which stores integer values arranged in non-decreasing order

multiset <int> s;

//initializes a multiset having initial values as 10,20,30

multiset <int> s = { 10, 20, 30 };

//initializes a multiset of size 0 which stores integer values arranged in non-increasing order

multiset <int, greater <int>> s;

--------mutliset function()---------------

begin(): Returns an iterator to the first element of the multiset.

Parameters: None

Return type: iterator

end(): Returns an iterator to the element past the last element of the multiset.

Parameters: None

Return type: iterator

size(): It tells us the size of the multiset.

Parameters: None

Return type: integer - total number of elements in the multiset

insert(element): Inserts an element in the multiset.    // or emplace(element)

Time Complexity: O(logN) where N is the size of the multiset

Parameters: the element to be inserted

Return type: void

erase(value) or erase(start\_iterator,end\_iterator): Delete elements from the multiset.

Time Complexity: O(logN) where N is the size of the multiset

Parameters: the value to be removed or iterators pointing to the position between which the value needs to be deleted

Return type: void

find(element): Returns an iterator pointing to the element, if the element is found else returns an iterator pointing to the end of the multiset.

Parameters: the element which needs to be found

Return type: iterator

clear(): It deletes all the elements from the multiset

Parameters: None

Return type: void

empty(): It tells us whether the multiset is empty or not.

Parameters: None

Return type: Boolean, true if a multiset is empty else false

------------------------------MAP Container-------------------------------------

\* Maps are used to replicate associative arrays.

\* Maps contain sorted key-value pair, in which each key is unique and cannot be changed, and it can be inserted or deleted      but cannot be altered.

\* Value associated with keys can be altered. We can search, remove and insert in a map within O(n) time complexity.

\* maps always arrange its keys in sorted order. In case the keys are of string type, they are sorted lexicographically.

Syntax -

map<key\_type , value\_type> map\_name;

Example -

 map<int,int> m{ {1,2} , {2,3} , {3,4} };

    /\* creates a map m with keys 1,2,3 and

        their corresponding values 2,3,4 \*/

    //printing map

    for(auto it : m){

        cout<<it.first << " " << it.second << endl;

    }

    for(auto it = m.begin(); it!=m.end();it++) {

        cout << it->first << " " << it->second << endl;

    }

    map<string,int> map1;

    /\*  creates a map with keys of type character and

      values of type integer \*/

    map1["abc"]=100;    // inserts key = "abc" with value = 100

    map1["b"]=200;      // inserts key = "b" with value = 200

    map1["c"]=300;      // inserts key = "c" with value = 300

    map1["def"]=400;    // inserts key = "def" with value = 400

    map1.emplace("def",400); // inserts key = "def" with value = 400

    map<char,int> map2 (map1.begin(), map1.end());

    /\* creates a map map2 which have entries copied

        from map1.begin() to map1.end() \*/

    map<char,int> map3 (m);

    /\* creates map map3 which is a copy of map m \*/

----------- Member Functions of Map ---------------

at() and [ ]

Both at and [ ] are used for accessing the elements in the map. The only difference between them is that at throws an exception if the accessed key is not present in the map, on the other hand operator [ ] inserts the key in the map if the key is not present already in the map.

Example -

map<int,string> m{ {1,”nikhilesh”} , {2,”shrikant”} , {3,”ashish”} };

    cout << m.at(1) ;  // prints value associated with key 1 ,i.e nikhilesh

    cout << m.at(2) ;  // prints value associated with key 2 ,i.e shrikant

    /\* note that the parameters in the above at() are the keys not the index \*/

    cout << m[3] ; // prints value associated with key 3 , i.e ashish

    m.at(1) = "vikas";   // changes the value associated with key 1 to vikas

    m[2] = "navneet";   // changes the value associated with key 2 to navneet

    m[4] = "doodrah";

    /\* since there is no key with value 4 in the map,

        it insert a key-value pair in map with key=4 and value = doodrah \*/

    m.at(5) = "umeshwa";

    /\* since there is no key with value 5 in the map ,

     it throws an exception  \*/

----- empty(), size() and max\_size() -------

empty() returns boolean true if the map is empty, else it returns Boolean false.

size() returns number of entries in the map, an entry consist of a key and a value.

max\_size() returns the upper bound of the entries that a map can contain (maximum possible entries) based on the memory allocated to the map.

------ insert() and insert\_or\_assign() -------------

insert() is used to insert entries in the map.

Since keys are unique in a map, it first checks that whether the given key is already present in the map or not, if it is present the entry is not inserted in the map and the iterator to the existing key is returned otherwise new entry is inserted in the map.

There are two variations of insert():

insert(pair) : In this variation, a pair of key and value is inserted in the map. The inserted pair is always inserted at the appropriate position as keys are arranged in sorted order.

insert(start\_itr , end\_itr): This variation inserts the entries in range defined by start\_itr and end\_itr of another map.

The insert\_or\_assing() works exactly as insert() except that if the given key is already present in the map then its value is modified.

--------- erase() and clear() ----------

erase() removes the entry from the map pointed by the iterator (which is passed as parameter), however if we want to remove all the elements from the map, we can use clear(), it clears the map and sets its size to 0.

There are two variations of erase :

erase(iterator\_itr) : This removes entry from the map pointed by iterator iterator\_itr, reducing the size of map by 1.

erase(start\_iterator, end\_iterator) : It removes the elements in range specified by the start\_iterator and end\_iterator.

// does not stores in any order

    unordered\_map<int,int> mpp;

    // unordered\_map<pair<int,int>,int> mpp; xxxxxx not possible

    // o(1) in almost all cases

    // o(n) in the worst case, where n is the container size

    multimap<string, int> mpp;

    mpp.emplace("raj", 2);

    mpp.emplace("raj", 5);

--------------------------- PAIR Template ----------------------------------

SYNTAX:-

pair<T1,T2>  pair1, pair2 ;

The above code creates two pairs, namely pair1 and pair2, both having first object of type T1 and second object of type T2.

Here are some function for pair template :

Operator = : assign values to a pair.

swap : swaps the contents of the pair.

make\_pair() : create and returns a pair having objects defined by parameter list.

Operators( == , != , > , < , <= , >= ) : lexicographically compares two pairs.

Example -

 pair<int,int> pair1, pair3;    //creats pair of integers

   pair<int,string> pair2;    // creates pair of an integer an a string

   pair1 = make\_pair(1, 2);     // insert 1 and 2 to the pair1

   pair2 = make\_pair(1, "Studytonight") // insert 1 and "Studytonight" in pair2

   pair3 = make\_pair(2, 4)

   cout<< pair1.first << endl;  // prints 1, 1 being 1st element of pair1

   cout<< pair2.second << endl; // prints Studytonight

   pair< pair<int,int>, int> pr = {{1,2}, 2};

    cout << pr.first.second << endl;

    pair<pair<int,int>, pair<int,int>> pr = {{1,2},{2, 4}};

    cout << pr.first.first; -> 1

    cout << pr.second.second; -> 4

--------------------- STACK Container------------------------

The stack container is used to replicate stacks in c++, insertion and deletion is always performed at the top of the stack.

Syntax :-

stack<object\_type> stack\_name;

-------------- push() function ---------------

push() is used to insert the element in the stack, the elements are inserted at the top of the stack.

Example -

 stack<int> s;   // creates an empty stack of integer s

    s.push(2);   // pushes 2 in the stack  , now top =2

    s.push(3);   // pushes 3 in the stack  , now top =3

----------- pop() function ------------

This method is used to removes single element from the stack. It reduces the size of the stack by 1.

The element removed is always the topmost element of the stack (most recently added element).

The pop() method does not return anything.

// deleted the entire stack

    while(!st.empty()) {

        st.pop();

    }

---------- top() function --------------

This method returns the topmost element of the stack.

Note that this method returns the element, not removes it, unlike pop().

SYNTAX: top()

--------- size() and empty() functions -----------

size() returns the number of elements present in the stack, whereas empty() checks if the stack is empty or not.

empty returns true if the stack is empty else false is returned.

Example -

stack<int> s;

    // pushing elements into stack

    s.push(2);

    s.push(3);

    s.push(4);

    cout << s.top();   // prints 4, as 4 is the topmost element

    cout << s.size();  // prints 3, as there are 3 elements in

---------------------------------- QUEUE Container ---------------------------------------

The queue container is used to replicate queue in C++, insertion always takes place at the back of the queue and deletion is always performed at the front of the queue.

Syntax:

queue < object\_type >  queue\_name;

-------- push() function -----------

push() is used to insert the element in the queue. The element is inserted at the back or rear of the queue.

Example -

 queue <int> q;   // creates an empty queue of integer q

    q.push(2);   // pushes 2 in the queue  , now front = back = 2

    q.push(3);   // pushes 3 in the queue  , now front = 2 , and back = 3

-------- pop() function -------------

This method removes single element from the front of the queue and therefore reduces its size by 1.

The element removed is the element that was entered first.

The pop() does not return anything.

Example -

queue <int> q;   // creates an empty queue of integer q

    q.push(2);   // pushes 2 in the queue  , now front = back = 2

    q.push(3);   // pushes 3 in the queue  , now front = 2 , and back = 3

    q.pop() ;  // removes 2 from the stack , front = 3

--------- front() and back() functions -------------

front() returns the front element of the queue whereas back() returns the element at the back of the queue.

----------------------------------- PRIORITY QUEUE ---------------------------------------------

priority\_queue is just like a normal queue except the element removed from the queue is always the greatest among all the elements in the queue, thus this container is usually used to replicate Max Heap in C++. Elements can be inserted at any order and it have O(log(n)) time complexity for insertion.

Syntax:

priority\_queue<int> pq;

------- push() function ---------

This method inserts an element in the priority\_queue.

The insertion of the elements have time complexity of logarithmic time.

Example-

  priority\_queue<int> pq1;

    pq1.push(30);  // inserts 30 to pq1 , now top = 30

    pq1.push(40);  // inserts 40 to pq1 , now top = 40 ( maxinmum element)

    pq1.push(90);  // inserts 90 to pq1 , now top = 90

    pq1.push(60);   // inserts 60 to pq1 , top still is 90

--------- pop() function ------------

This method removes the topmost element from the priority\_queue (greatest element) ,reducing the size of the priority queue by 1.

Example -

    pq1.pop();  // removes 90 ( greatest element in the queue

------- top() function -----------

This method returns the element at the top of the priority\_queue which is the greatest element present in the queue.

------- empty() and size() functions ----------

size() returns the number of element present in the priority \_queue, whereas empty() returns Boolean true if the priority\_queue is empty else Boolean false is returned.

--------- swap() function --------------

This method swaps the elements of two priority\_queue.

------------------------------------ DEQUE Container ------------------------------------------

Deque is a shorthand for doubly ended queue. Deque allows fast insertion and deletion at both ends of the queue. Although we can also use vector container for the insertion and deletion at both of its ends, but insertion and deletion at the front of the array is costlier than at the back, in case of deque but deque are more complex internally.

Syntax:

deque< object\_type > deque\_name;

--------- push\_back(), push\_front() and insert() functions ----------------

push\_back(element e) inserts an element e at the back of the deque.

push\_front(element e) inserts the element e at the front of the deque.

insert() method has three variations :

insert(iterator i, element e) : Inserts element e at the position pointed by iterator i in the deque.

insert(iterator i, int count, element e) : Inserts element e, count number of times from the position pointed by iterator i.

insert(iterator i, iterator first, iterator last) : Inserts the element in the range [first,last] at the position pointed by iterator i in deque.

Example -

int a[] = { 1,5,8,9,3 };

    deque<int> dq(a, a+5);

    /\* creates s deque with elements 1,5,8,9,3  \*/

    dq.push\_back(10);

    /\* now dq is : 1,5,8,9,3,10 \*/

    dq.push\_front(20);

    /\* now dq is : 20,1,5,8,9,3,10  \*/

    deque<int>::iterator i;

    i=dq.begin()+2;

    /\* i points to 3rd element in dq \*/

    dq.insert(i,15);

    /\* now dq 20,1,15,5,8,9,3,10  \*/

    int a[]={7,7,7,7};

    d1.insert(dq.begin() , a , a+4 );

    /\* now dq is 7,7,7,7,20,1,15,5,8,9,3,10  \*/

------- pop\_back() and pop\_front() functions ---------

pop\_back() removes an element from the back of the deque.

pop\_front() removes an element from the front of the deque, both decreasing the size of the deque by one.

Example -

int a[] = { 1,5,8,9,3,5,6,4 };

    deque<int> dq(a,a+8);

    /\* creates s deque with elements 1,5,8,9,3,5,6,4  \*/

    dq.pop\_back();

    /\* removes an element from the back \*/

    /\* now the deque dq is : 1,5,8,9,3,5,6 \*/

    dq.pop\_front();

    /\* now dq is : 1,5,8,9,3,5,6  \*/

-------------------------------- LIST Container -------------------------------------

Array and Vector are contiguous containers, i.e they store their data on continuous memory, thus the insert operation at the middle of vector/array is very costly (in terms of number of operaton and process time) because we have to shift all the elements, linked list overcome this problem. Linked list can be implemented by using the list container.

Syntax:

list<int> l;  /\* Creates a new empty linked list l \*/

list<int> l{1,2,3};

----------- insert() function---------------

This method, as the name suggests, inserts an element at specific position, in a list.

insert(iterator, element) : inserts element in the list before the position pointed by the iterator.

insert(iterator, count, element) : inserts element in the list before the position pointed by the iterator, count number of times.

insert(iterator, start\_iterator, end\_iterator): insert the element pointed by start\_iterator to the element pointed by end\_iterator before the position pointed by iterator

Example -

list<int> l = {1,2,3,4,5};

    list<int>::iterator it = l.begin();

    l.insert (it+1, 100);    // insert 100 before 2 position

    /\* now the list is 1 100 2 3 4 5 \*/

    list<int> new\_l = {10,20,30,40};   // new list

    new\_l.insert (new\_l.begin(), l.begin(), l.end());

    /\*

        insert elements from beginning of list l to end of list l

        before 1 position in list new\_l \*/

    /\* now the list new\_l is 1 100 2 3 4 5 10 20 30 40 \*/

    l.insert(l.begin(), 5, 10);  // insert 10 before beginning 5 times

    /\* now l is 10 10 10 10 10 1 100 2 3 4 5 \*/

------------- push\_back() and push\_front() functions-----------------

push\_back(element) method is used to push elements into a list from the back.

push\_front(element) method is used to push elements into a list from the front.

Example -

list<int> l{1,2,3,4,5};

    l.push\_back(6);

    l.push\_back(7);

    /\* now the list becomes 1,2,3,4,5,6,7 \*/

    l.push\_front(8);

    l.push\_front(9);

    /\* now the list becomes 9,8,1,2,3,4,5,6,7 \*/

------- pop\_back() and pop\_front() functions -------------

pop\_front() removes first element from the start of the list.

pop\_back() removes first element from the end of the list.

Example -

 list<int> l{1,2,3,4,5};

    l.pop\_back()();

    /\* now the list becomes 1,2,3,4 \*/

    l.pop\_front()();

    /\* now the list becomes 2,3,4 \*/

------- front() and back() function -------------

front() is used to get the first element of the list from the start.

back() is used to get the first element of the list from the back.

--------- reverse() function -----------------

This method can be used to reverse a list completely.

 list<int> l{1,2,3,4,5};

    reverse();

    /\* now the list becomes 5,4,3,2,1 \*/

---------- sort() function ---------------

sort() method sorts the given list. It does not create new sorted list but changes the position of elements within an existing list to sort it.

sort() : sorts the elements of the list in ascending order, the element of the list should by numeric for this function.

Example -

 list<int> list1 = {2,4,5,6,1,3};

    list1.sort();

    /\* list1 is now 1 2 3 4 5 6 \*/

-------- splice() function ----------------

splice() method transfers the elements from one list to another.

There are three versions of splice :

splice(iterator, list\_name) : Transfers complete list list\_name at position pointed by the iterator.

splice(iterator, list\_name, iterator\_pos) : Transfer elements pointed by iterator\_pos from list\_name at position pointed by iterator.

splice(iterator, list\_name, itr\_start, itr\_end) : Transfer range specified by itr\_start and itr\_end from list\_name at position pointed by iterator.

Example -

list<int> list1 = {1,2,3,4};

    list<int> list2 = {5,6,7,8};

    list<int>::iterator it;

    it = list1.begin();

    ++it;   //pointing to second position

    list1.splice(it, list2);

    /\* transfer all elements of list2 at position 2 in list1 \*/

    /\* now list1 is 1 5 6 7 8 2 3 4 and list2 is empty \*/

    list2.splice(list2.begin(), list1, it);

    /\* transfer element pointed by it in list1 to the beginning of list2 \*/

    /\* list2 is now 5 and list1 is 1 6 7 8 2 3 4\*/

---------- merge() function -------------

Merges two sorted list. It is mandatory that both the list should be sorted first.

merge() merges the two list such that each element is placed at its proper position in the resulting list.

Syntax- list1.merge(list2)

Example -

list<int> list1 = {1,3,5,7,9};

    list<int> list2 = {2,4,6,8,10};

    /\* both the lists are sorted. In case they are not ,

    first they should be sorted by sort function() \*/

    list1.merge(list2);

    /\* list list1 is now 1,2,3,4,5,6,7,8,9,10  \*/

    cout << list1.size() << endl;    // prints 10

----------------------------------- Bitset ----------------------------------------------------

A bitset stores bits (elements with only two possible values: 0 or 1, true or false, ...).

Member functions

1   bitset::all()

Tests whether all bits from bitset are set or not.

2   bitset::any()

Tests whether at least one bit from bitset is set or not.

3   bitset::count()

Count number of set bits from bitset.

4   bitset::flip() all bits

Toggles all bits from bitset.

5   bitset::flip() single bit

Toggles single bit from bitset.

6   bitset::none()

Tests whether all bits are unset or not.

7   bitset::operator[] bool version

Returns the value of bit at position pos.

8   bitset::operator[] reference version

Returns the reference of bit at position pos.

9   bitset::reset() all bits

Reset all bits of bitset to zero.

10  bitset::reset(2) single bit  // reset a particular index

Reset single bit of bitset to zero.

11  bitset::set() all bits

Set all bits from bitset to one.

12  bitset::set() single bit

Set single bit from bitset to either one or zero.

13  bitset::size()

Reports the size of the bitset.

14  bitset::test()

Tests whether Nth bit is set or not.

15  bitset::to\_string()

Converts bitset object to string object.

16  bitset::to\_ullong()

Convert bitset to unsigned long long.

17  bitset::to\_ulong()

Convert bitset to unsigned long.

------------------------------------------------------------------------------------------------------------

------------------------------------------- Algorithms -----------------------------------------------------

------------------------------------------------------------------------------------------------------------

--------------Sorting Algorithms in STL------------------------

------------ sort() method ------------

1. sort(start\_iterator, end\_iterator ) : sorts the range defined by iterators start\_iterator and end\_iterator in ascending order.

2.sort(start\_iterator, end\_iterator, compare\_function) : this also sorts the given range but you can define how the sorting should be done by compare\_function.

Example -

bool compare\_function(int i, int j)

{

   return i > j;    // return 1 if i>j else 0

}

bool compare\_string(string i, string j)

{

  return (i.size() < j.size());

}

    int arr[5] = {1,5,8,4,2};

    sort(arr , arr+5);    // sorts arr[0] to arr[4] in ascending order

    /\* now the arr is 1,2,4,5,8  \*/

    vector<int> v1;

    v1.push\_back(8);

    v1.push\_back(4);

    v1.push\_back(5);

    v1.push\_back(1);

    /\* now the vector v1 is 8,4,5,1 \*/

    vector<int>::iterator i, j;

    i = v1.begin();   // i now points to beginning of the vector v1

    j = v1.end();     // j now points to end of the vector v1

    sort(i,j);      //sort(v1.begin() , v1.end() ) can also be used

    /\* now the vector v1 is 1,4,5,8 \*/

    /\* use of compare\_function \*/

    int a2[] = { 4,3,6,5,6,8,4,3,6 };

    sort(a2,a2+9,compare\_function);  // sorts a2 in descending order

    /\* here we have used compare\_function which uses operator(>),

    that result into sorting in descending order \*/

    /\* compare\_function is also used to sort non-numeric elements such as\*/

    string s[]={"a" , "abc", "ab" , "abcde"};

    sort(s,s+4,compare\_string);

    /\* now s is "a","ab","abc","abcde"  \*/

------------ partial\_sort() method -------------------

1. partial\_sort(start, middle, end ) : sorts the range from start to end in such a way that the elements before middle are in ascending order and are the smallest elements in the range.

2. partial\_sort(start, middle, end, compare\_function) : sorts the range from start to end in such a way that the elements before middle are sorted with the help of compare\_function and are the smallest elements in the range.

Example -

int a[] = {9,8,7,6,5,4,3,2,1};

    partial\_sort(a, a+4, a+9);

    /\* now a is 1,2,3,4,9,8,7,6,5  \*/

    int b[] = {1,5,6,2,4,8,9,3,7};

    /\* sorts b such that first 4 elements are the greatest elements

    in the array and are in descending order \*/

    partial\_sort(b, b+4, b+9);

    /\* now b is  9,8,7,6,1,2,4,3,5 \*/

----------- is\_sorted() method ----------

1. is\_sorted(start\_iterator, end\_iterator) : Checks the range defined by iterators start\_iterator and end\_iterator in ascending order.

2. is\_sorted(start\_iterator, end\_iterator, compare\_function) : It also checks the given range but you can define how the sorting must be done.

--------------------------------------Binary Search-------------------------------------------

1. binary\_search(first, last, value): this version returns true if there is an element present, satisfying the condition (!(a < value) &&!(value < a)) in the given range, i.e from first to last, in other words, operator(<) is used to check the equality of the two elements.

2. binary\_search(first, last, value, compare\_function) : this version return true if there is an element present in the given range, i.e from first to the last.

Example -

bool compare\_string\_by\_length (string i,string j)

{

    return (i.size() == j.size());

}

 int inputs[] = {7,8,4,1,6,5,9,4};

    vector<int> v(inputs, inputs+8);

    cout<<binary\_search(v.begin() , v.end() , 7 );  //prints 1 , Boolean true

    cout<<binary\_search(v.begin() , v.end() , 217); //prints 0 , Boolean false

    /\* compare\_function can be used to search

    non numeric elements based on their properties \*/

    string s[] = { "test" , "abcdf" , "efghijkl" , "pop" };

    cout<<binary\_search(s, s+4, "nickt" , compare\_string\_by\_length);

    /\* search for the string in s which have same length as of "nicky" \*/

---------------------------Upper Bound and Lower Bound Search--------------------------------------

upper\_bound() returns an iterator to the elements in the given range which does not compare greater than the given value.

The range given should be already sorted for upper\_bound() to work properly.

In other words it returns an iterator to the upper bound of the given element in the given sorted range.

Example-

int input[] = {1,2,2,3,4,4,5,6,7,8,10,45};

    vector<int> v(input, input+12);

    vector<int>::iterator it1 , it2;

    it1 = upper\_bound(v.begin(), v.end(), 6);

    /\* points to eight element in v \*/

    it2 = upper\_bound(v.begin(), v.end(), 4);

    /\* points to seventh element in v \*/

------- lower\_bound() method ----------

lower\_bound() returns an iterator to the elements in the given range which does no compare less than the given value.

The range given should be already sorted for lower\_bound() to work properly.

In other words it returns an iterator to the lower bound of the given element in the given sorted range.

Example-

int input[] = {1,2,2,3,4,4,5,6,7,8,10,45};

    vector<int> v(input,input+12);

    vector<int>::iterator it1 , it2;

    it1 = lower\_bound(v.begin(), v.end(), 4);

    /\* points to fifth element in v \*/

    it2 = lower\_bound (v.begin(), v.end(), 10);

    /\* points to second last element in v \*/

-----------------------------------Non Modifying Algorithms-----------------------------------------

------------ count() method -------------

count() returns the number of elements in the given range that are equal to given value.

count(first ,last ,value) : This will return number of the element in range defined by iterators first and last ( excluded ) which are equal ( == ) the value .

Example -

int values[] = {5,1,6,9,10,1,12,5,5,5,1,8,9,7,46};

    int count\_5 = count(values, values+15, 5);

    /\* now count\_5 is equal to 4 \*/

    vector<int> v(values, values+15);

    int count\_1 = count(v.begin(), v.end(), 1);

    /\* now count\_1 is equal to  \*/

-------- equal() method ------------

equal() compares the elements in two ranges, if all the elements in one range compares equal to their corresponding elements in other range, Boolean true is returned, else Boolean false is returned.

equal(first1, last1, first2) : This function compare for the equality of elements in the range pointed by first1 and last1(excluded) to the range with starting position first2. If all elements are equal , true is returned else false.

equal(first1 ,last1 ,first2 ,cmp\_function) : Here cmp\_function is used to decide how to check the equality of two elements, it is useful for non-numeric elements like strings and objects.

-------------------------------------------------------------------------

// Next Permutation

    // string s = "abc"

    // all permutations are as follows:

    // abc

    // acb

    // bac

    // bca

    // cab

    // cba

    // s = "bca"

    bool res = next\_permutation(s.begin(), s.end());

    // s = "cba"

    bool res = next\_permutation(s.begin(), s.end());

    // if I give you any random string s = "bca"

    // i want you to print all the permutations

    string s = "bca";

    sort(s.begin(), s.end()); // this makes the string as "abc"

    do {

        cout << s << endl;

    } while(next\_permutation(s.begin(), s.end()));

    int arr[] = {1, 6, 5};

    int n = 3;

    sort(arr, arr + n); // this makes the array as {1, 5, 6}

    do {

        for(int i = 0;i<n;i++) cout << arr[i] << " ";

        cout << endl;

    } while(next\_permutation(arr, arr+n)) ;

    // prev permutation

    bool res = prev\_permutation(s.begin(), s.end());

    // COMPARATOR

    sort(arr, arr+n); // sorts everything in ascending order

    sort(arr, arr+n, comp);

    // descending

    sort(arr, arr+n, comp);

    // greater<int> is an inbuilt comparator

    // which works only if you wanna do this in descending

    sort(arr, arr+n, greater<int>);