V – all methods

L

D  
A  
F

S – push, pop, top, empty, size (use while loop to iterate)

P – no front(), top, push, pop, size, swap

Q – front(), no top, push, pop, size, swap

Set – insert(no push\_back)

Multiset- same

Map - same

Multimap - same

#include <iostream>

#include <stack>

using namespace std;

int main() {

stack<int> stack;

stack.push(21);

stack.push(22);

stack.push(24);

stack.push(25);

while (!stack.empty()) {

cout << stack.top() <<" ";

stack.pop();

}

}

Set – insert

Mset – insert, find, erase,

Map

Mmap

STL’s

**STL has four components**

* Algorithms
* Containers
* Functions
* Iterators

**Algorithms**

The header algorithm defines a collection of functions especially designed to be used on ranges of elements.They act on containers and provide means for various operations for STL’S

the contents of the containers.

* Algorithm
  + [Sorting](https://www.geeksforgeeks.org/sort-algorithms-the-c-standard-template-library-stl/)
  + [Searching](https://www.geeksforgeeks.org/binary-search-algorithms-the-c-standard-template-library-stl/)
  + [Important STL Algorithms](https://www.geeksforgeeks.org/c-magicians-stl-algorithms/)
  + [Useful Array algorithms](https://www.geeksforgeeks.org/useful-array-algorithms-in-c-stl/)
  + [Partition Operations](https://www.geeksforgeeks.org/stdpartition-in-c-stl/)
* Numeric
  + [valarray class](https://www.geeksforgeeks.org/std-valarray-class-c/)

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

VLDAF ::

* Sequence Containers: implement data structures which can be accessed in a sequential manner.
  + [vector](https://www.geeksforgeeks.org/vector-in-cpp-stl/)
  + [list](https://www.geeksforgeeks.org/list-cpp-stl/)
  + [deque](https://www.geeksforgeeks.org/deque-cpp-stl/)
  + [arrays](https://www.geeksforgeeks.org/array-class-c/)
  + [forward\_list](https://www.geeksforgeeks.org/forward-list-c-set-1-introduction-important-functions/)( Introduced in C++11)
* Container Adaptors : provide a different interface for sequential containers.
  + [queue](https://www.geeksforgeeks.org/queue-cpp-stl/)
  + [priority\_queue](https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/)
  + [stack](https://www.geeksforgeeks.org/stack-in-cpp-stl/)
* Associative Containers : implement sorted data structures that can be quickly searched (O(log n) complexity).
  + [set](https://www.geeksforgeeks.org/set-in-cpp-stl/)
  + [multiset](https://www.geeksforgeeks.org/multiset-in-cpp-stl/)
  + [map](https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/)
  + [multimap](https://www.geeksforgeeks.org/multimap-associative-containers-the-c-standard-template-library-stl/)
* Unordered Associative Containers : implement unordered data structures that can be quickly searched
  + [unordered\_set](https://www.geeksforgeeks.org/unordered_set-in-cpp-stl/) (Introduced in C++11)
  + [unordered\_multiset](https://www.geeksforgeeks.org/unordered_multiset-and-its-uses/) (Introduced in C++11)
  + [unordered\_map](https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/) (Introduced in C++11)
  + [unordered\_multimap](https://www.geeksforgeeks.org/unordered_multimap-and-its-application/) (Introduced in C++11)

**Functions**

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

* [Functors](https://www.geeksforgeeks.org/functors-in-cpp/)

**Iterators**

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

* [Iterators](https://www.geeksforgeeks.org/iterators-c-stl/)

**Utility Library**

Defined in header <utility>.

* [pair](https://www.geeksforgeeks.org/pair-in-cpp-stl/).

**MAIN PART**

Algorithms::

1)sorting ->

[Sorting](https://www.geeksforgeeks.org/sorting-algorithms/) is one of the most basic functions applied to data. It means arranging the data in a particular fashion, which can be increasing or decreasing. There is a builtin function in C++ STL by the name of sort().

sort(startaddress, endaddress)

|  |
| --- |
| // C++ program to sort an array  #include <algorithm>  #include <iostream>  using namespace std;  void show(int a[], int array\_size)  { for (int i = 0; i < array\_size; ++i)          cout << a[i] << " ";}  int main()  {      int a[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };        int asize = sizeof(a) / sizeof(a[0]);      cout << "The array before sorting is : \n";        // print the array      show(a, asize);  sort(a, a + asize);  cout << "\n\nThe array after sorting is :\n";      show(a, asize);   return 0;} |

2-> searching ::

[Binary search](https://www.geeksforgeeks.org/binary-search/) is a widely used searching algorithm that requires the array to be sorted before search is applied. The main idea behind this algorithm is to keep dividing the array in half (divide and conquer) until the element is found, or all the elements are exhausted.  
It works by comparing the middle item of the array with our target, if it matches, it returns true otherwise if the middle term is greater than the target, the search is performed in the left sub-array.   
If the middle term is less than the target, the search is performed in the right sub-array.

**binary\_search(startaddress, endaddress, valuetofind)**

# include <algorithm>

# include <iostream>

using namespace std;

void show(int a[], int arraysize)

{

    for (int i = 0; i < arraysize; ++i)

        cout << a[i] << ",";

}

int main()

{

    int a[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };

    int asize = sizeof(a) / sizeof(a[0]);

    cout << "\nThe array is : \n";

    show(a, asize);

    cout << "\n\nLet's say we want to search for ";

    cout << "\n2 in the array So, we first sort the array";

    sort(a, a + asize);

    cout << "\n\nThe array after sorting is : \n";

    show(a, asize);

    cout << "\n\nNow, we do the binary search";

    if (binary\_search(a, a + 10, 2))

        cout << "\nElement found in the array";

    else

        cout << "\nElement not found in the array";

    cout << "\n\nNow, say we want to search for 10";

    if (binary\_search(a, a + 10, 10))

        cout << "\nElement found in the array";

    else

        cout << "\nElement not found in the array";

    return 0;}

CONTAINERS

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

* Sequence Containers: implement data structures which can be accessed in a sequential manner.

ARRAYS::

The array is a collection of homogeneous objects and this array container is defined for constant size arrays or (static size). This container wraps around fixed-size arrays and the information of its size are not lost when declared to a pointer.

VECTORS

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

**All Vector Functions :**

* [vector::begin() and vector::end()](https://www.geeksforgeeks.org/vectorbegin-vectorend-c-stl/)
* [vector rbegin() and rend()](https://www.geeksforgeeks.org/vector-rbegin-and-rend-function-in-c-stl/)
* [vector::cbegin() and vector::cend()](https://www.geeksforgeeks.org/vector-cbegin-vector-cend-c-stl/)
* [vector::crend() and vector::crbegin()](https://www.geeksforgeeks.org/vectorcrend-vectorcrbegin-examples/)
* [vector::assign()](https://www.geeksforgeeks.org/vector-assign-in-c-stl/)
* [vector::at()](https://www.geeksforgeeks.org/vectorat-vectorswap-c-stl/)
* [vector::back()](https://www.geeksforgeeks.org/vectorfront-vectorback-c-stl/)
* [vector::capacity()](https://www.geeksforgeeks.org/vector-capacity-function-in-c-stl/)
* [vector::clear()](https://www.geeksforgeeks.org/vectorclear-vectorerase-c-stl/)
* [vector::push\_back()](https://www.geeksforgeeks.org/vectorpush_back-vectorpop_back-c-stl/)
* [vector::pop\_back()](https://www.geeksforgeeks.org/vectorpush_back-vectorpop_back-c-stl/)
* [vector::empty()](https://www.geeksforgeeks.org/vectorempty-vectorsize-c-stl/)
* [vector::erase()](https://www.geeksforgeeks.org/vectorclear-vectorerase-c-stl/)
* [vector::size()](https://www.geeksforgeeks.org/vectorempty-vectorsize-c-stl/)
* [vector::swap()](https://www.geeksforgeeks.org/vectorat-vectorswap-c-stl/)
* [vector::reserve()](https://www.geeksforgeeks.org/using-stdvectorreserve-whenever-possible/)
* [vector::resize()](https://www.geeksforgeeks.org/vector-resize-c-stl/)
* [vector::shrink\_to\_fit()](https://www.geeksforgeeks.org/vector-shrink_to_fit-function-in-c-stl/)
* [vector::operator=](https://www.geeksforgeeks.org/vectoroperator-vectoroperator-c-stl/)
* [vector::operator[]](https://www.geeksforgeeks.org/vectoroperator-vectoroperator-c-stl/)
* [vector::front()](https://www.geeksforgeeks.org/vectorfront-vectorback-c-stl/)
* [vector::data()](https://www.geeksforgeeks.org/vector-data-function-in-c-stl/)
* [vector::emplace\_back()](https://www.geeksforgeeks.org/vectoremplace_back-c-stl/)
* [vector::emplace()](https://www.geeksforgeeks.org/vector-emplace-function-in-c-stl/)
* [vector::max\_size()](https://www.geeksforgeeks.org/vector-max_size-function-in-c-stl/)
* [vector::insert()](https://www.geeksforgeeks.org/vector-insert-function-in-c-stl/)

all are defined in **<algorithm>**  header file:

auto it = max\_element(v.begin(),v.end())

or \*max\_element (min\_element)

It returns an iterator pointing to the element with the largest value in the range [first, last).

find: It returns an iterator to the first occurrence of the specified element in the given sequence in given range.

it = std::find(vec.begin(),

                 vec.end(), ser);

**if** (it != vec.end())

  {

    std::cout << "Element " << ser <<

                 " found at position : ";

    std::cout << it - vec.begin() <<

                 " (counting from zero) \n";

  }

Sort: to sort a vector

**std::search** is defined in the header file <algorithm> and used to find out the presence of a subsequence satisfying a condition (equality if no such predicate is defined) with respect to another sequence.

i1 = std::search(v1.begin(), v1.end(), v2.begin(), v2.end());

Std::if and std::if\_not:

These functions provide an efficient way to search for an element in a container using a predicate function.

**std::find\_end** is used to find the last occurrence of a sub-sequence inside a container.

**What is a tuple?**  
A tuple is an object that can hold a number of elements. The elements can be of different data types. The elements of tuples are initialized as arguments in order in which they will be accessed.

**Operations on tuple** :-

**1. get()** :- get() is used to access the tuple values and modify them, it accepts the index and tuple name as arguments to access a particular tuple element.

**2. make\_tuple()** :- make\_tuple() is used to assign tuple with values. The values passed should be in order with the values declared in tuple.

FORWARD \_ LIST :---------------------------------------------------------🡪>…..

Forward list in STL implements singly linked list. Introduced from C++11, forward list are more useful than other containers in insertion, removal and moving operations (like sort) and allows time constant insertion and removal of elements.

* [front()](https://www.geeksforgeeks.org/forward_listfront-forward_listempty-c-stl/)– This function is used to reference the first element of the forward list container.
* [begin()](https://www.geeksforgeeks.org/forward_listbegin-forward_listend-c-stl/)– begin() function is used to return an iterator pointing to the first element of the forward list container.
* [end()](https://www.geeksforgeeks.org/forward_listbegin-forward_listend-c-stl/)– end() function is used to return an iterator pointing to the last element of the list container.
* [cbegin()](https://www.geeksforgeeks.org/forward_list-cbegin-in-c-stl/)– Returns a constant iterator pointing to the first element of the forward\_list.
* [cend()](https://www.geeksforgeeks.org/forward_listcend-in-c-stl-with-example/)– Returns a constant iterator pointing to the past-the-last element of the forward\_list.
* [before\_begin()](https://www.geeksforgeeks.org/forward_listbefore_begin-in-c-stl/)– Returns a iterator which points to the position before the first element of the forward\_list.
* [cbefore\_begin()](https://www.geeksforgeeks.org/forward_listcbefore_begin-in-c-stl/)– Returns a constant random access iterator which points to the position before the first element of the forward\_list.
* [max\_size()](https://www.geeksforgeeks.org/forward_listmax_size-in-c-stl/)– Returns the maximum number of elements can be held by forward\_list.
* [resize()](https://www.geeksforgeeks.org/forward_list-resize-function-in-c-stl/)– Changes the size of forward\_list.

**.**[**splice\_after()**](https://www.geeksforgeeks.org/forward_listsplice_after-in-c-stl/) :- This function transfers elements from one forward list to other.

DEQUE ------------------------------------------------------🡪>

Double ended queues are sequence containers with the feature of expansion and contraction on both the ends.  
They are similar to vectors, but are more efficient in case of insertion and deletion of elements. Unlike vectors, contiguous storage allocation may not be guaranteed.  
Double Ended Queues are basically an implementation of the data structure double ended queue. A queue data structure allows insertion only at the end and deletion from the front. This is like a queue in real life, wherein people are removed from the front and added at the back. Double ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.

The functions for deque are same as [vector](https://www.geeksforgeeks.org/vector-in-cpp-stl/), with an addition of push and pop operations for both front and back.

0. std::deque is an indexed sequence container.

1. It allows fast insertion at both beginning and end.

2. Unlike vector elements of deque are not stored contiguous.

3. it uses individual allocated fixed size array, with additional bookkeeping, meaning index based access to deque must perform two pointer dereference, but in vector we get in one dereference.

4. The storage of a deque is automatically expanded and contracted as needed.

5. Expansion of deque is cheaper than expansion of vector.

6. A deque holding just one element has to allocate its full internal array (e.g. 8 times the object size on 64-bit libstdc++; 16 times the object size or 4096 bytes, whichever is larger, on 64-bit libc++). TIME COMPLEXITY: Random access - constant O(1) Insertion or removal of elements at the end or beginning - constant O(1) Insertion or removal of elements - linear O(n)

* [deque insert() function in C++ STL](https://www.geeksforgeeks.org/deque-insert-function-in-c-stl/): Inserts an element. And returns an iterator that points to the first of the newly inserted elements.
* [deque rbegin() function in C++ STL](https://www.geeksforgeeks.org/deque-rbegin-function-in-c-stl/): Returns a reverse iterator which points to the last element of the deque (i.e., its reverse beginning).
* [deque rend() function in C++ STL](https://www.geeksforgeeks.org/deque-rend-function-in-c-stl/): Returns a reverse iterator which points to the position before the beginning of the deque (which is considered its reverse end).
* [deque cbegin() in C++ STL](https://www.geeksforgeeks.org/deque-cbegin-in-c-stl/): Returns a constant iterator pointing to the first element of the container, that is, the iterator cannot be used to modify, only traverse the deque.
* [deque max\_size() function in C++ STL](https://www.geeksforgeeks.org/deque-max_size-function-in-c-stl/): Returns the maximum number of elements that a deque container can hold.
* [deque assign() function in C++ STL](https://www.geeksforgeeks.org/deque-assign-function-in-c-stl/): Assign values to the same or different deque container.
* [deque resize() function in C++ STL](https://www.geeksforgeeks.org/deque-resize-function-in-c-stl/): Function which changes the size of the deque.
* [deque::push\_front() in C++ STL](https://www.geeksforgeeks.org/dequepush_front-c-stl/): This function is used to push elements into a deque from the front.
* [deque::push\_back() in C++ STL](https://www.geeksforgeeks.org/dequepush_back-c-stl/): This function is used to push elements into a deque from the back.
* [deque::pop\_front() and deque::pop\_back() in C++ STL](https://www.geeksforgeeks.org/dequepop_front-dequepop_back-c-stl/): **pop\_front()** function is used to pop or remove elements from a deque from the front. **pop\_back()** function is used to pop or remove elements from a deque from the back.
* [deque::front() and deque::back() in C++ STL](https://www.geeksforgeeks.org/dequefront-dequeback-c-stl/): **front()** function is used to reference the first element of the deque container. **back()** function is used to reference the last element of the deque container.
* [deque::clear() and deque::erase() in C++ STL](https://www.geeksforgeeks.org/dequeclear-dequeerase-c-stl/): **clear()** function is used to remove all the elements of the deque container, thus making its size 0. **erase()** function is used to remove elements from a container from the specified position or range.
* [deque::empty() and deque::size() in C++ STL](https://www.geeksforgeeks.org/dequeempty-dequesize-c-stl/): **empty()** function is used to check if the deque container is empty or not. **size()** function is used to return the size of the deque container or the number of elements in the deque container.
* [deque::operator= and deque::operator[] in C++ STL](https://www.geeksforgeeks.org/dequeoperator-dequeoperator-c-stl/):  
  **operator=** operator is used to assign new contents to the container by replacing the existing contents. **operator[]** operator is used to reference the element present at position given inside the operator.
* [deque::at() and deque::swap() in C++ STL](https://www.geeksforgeeks.org/dequeat-dequeswap-c-stl/): **at()** function is used reference the element present at the position given as the parameter to the function. **swap()** function is used to swap the contents of one deque with another deque of same type and size.
* [deque::begin() and deque::end in C++ STL](https://www.geeksforgeeks.org/dequebegin-dequeend-c-stl/): **begin()**function is used to return an iterator pointing to the first element of the deque container. **end()** function is used to return an iterator pointing to the last element of the deque container.
* [deque::emplace\_front() and deque::emplace\_back() in C++ STL](https://www.geeksforgeeks.org/deque-emplace_front-deque-emplace_back-cpp-stl/): **emplace\_front()** function is used to insert a new element into the deque container. The new element is added to the beginning of the deque. **emplace\_back()** function is used to insert a new element into the deque container. The new element is added to the end of the deque.

LIST (DOUBLY LINKED LIST)--------------------------------🡪>

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about doubly linked list.

* [front()](https://www.geeksforgeeks.org/list-front-function-in-c-stl/) – Returns the value of the first element in the list.
* [back()](https://www.geeksforgeeks.org/list-back-function-in-c-stl/) – Returns the value of the last element in the list .
* [push\_front(g)](https://www.geeksforgeeks.org/list-push_front-function-in-c-stl/) – Adds a new element ‘g’ at the beginning of the list .
* [push\_back(g)](https://www.geeksforgeeks.org/list-push_back-function-in-c-stl/) – Adds a new element ‘g’ at the end of the list.
* [pop\_front()](https://www.geeksforgeeks.org/list-pop_front-function-in-c-stl/) – Removes the first element of the list, and reduces size of the list by 1.
* [pop\_back()](https://www.geeksforgeeks.org/list-pop_back-function-in-c-stl/) – Removes the last element of the list, and reduces size of the list by 1
* [list::begin() and list::end() in C++ STL](https://www.geeksforgeeks.org/listbegin-listend-c-stl/)– **begin()** function returns an iterator pointing to the first element of the list
* [end()](https://www.geeksforgeeks.org/list-end-function-in-c-stl/)– **end()** function returns an iterator pointing to the theoretical last element which follows the last element.
* [list rbegin() and rend() function in C++ STL](https://www.geeksforgeeks.org/list-rbegin-and-rend-function-in-c-stl/)– **rbegin()** returns a reverse iterator which points to the last element of the list. **rend()**returns a reverse iterator which points to the position before the beginning of the list.
* [list cbegin() and cend() function in C++ STL](https://www.geeksforgeeks.org/list-cbegin-and-cend-function-in-c-stl/)– **cbegin()** returns a constant random access iterator which points to the beginning of the list. **cend()** returns a constant random access iterator which points to the end of the list.
* [list crbegin() and crend() function in C++ STL](https://www.geeksforgeeks.org/list-crbegin-and-crend-function-in-c-stl/)– **crbegin()** returns a constant reverse iterator which points to the last element of the list i.e reversed beginning of container. **crend()** returns a constant reverse iterator which points to the theoretical element preceding the first element in the list i.e. the reverse end of the list.
* [empty()](https://www.geeksforgeeks.org/list-empty-function-in-c-stl/) – Returns whether the list is empty(1) or not(0).
* [insert()](https://www.geeksforgeeks.org/list-insert-in-c-stl/) – Inserts new elements in the list before the element at a specified position.
* [erase()](https://www.geeksforgeeks.org/list-erase-function-in-c-stl/) – Removes a single element or a range of elements from the list.
* [assign()](https://www.geeksforgeeks.org/list-assign-function-in-c-stl/) – Assigns new elements to list by replacing current elements and resizes the list.
* [remove()](https://www.geeksforgeeks.org/list-remove-function-in-c-stl/) – Removes all the elements from the list, which are equal to given element.
* [list::remove\_if() in C++ STL](https://www.geeksforgeeks.org/listremove-listremove_if-c-stl/)– Used to remove all the values from the list that correspond true to the predicate or condition given as parameter to the function.
* [reverse()](https://www.geeksforgeeks.org/list-reverse-function-in-c-stl/) – Reverses the list.
* [size()](https://www.geeksforgeeks.org/list-size-function-in-c-stl/) – Returns the number of elements in the list.
* [list resize()function in C++ STL](https://www.geeksforgeeks.org/list-resize-function-in-c-stl/)– Used to resize a list container.
* [sort()](https://www.geeksforgeeks.org/stdlistsort-c-stl/) – Sorts the list in increasing order.
* [list max\_size() function in C++ STL](https://www.geeksforgeeks.org/list-max_size-function-in-c-stl/)– Returns the maximum number of elements a list container can hold.
* [list unique() in C++ STL](https://www.geeksforgeeks.org/list-unique-in-c-stl/)– Removes all duplicate consecutive elements from the list.
* [list::emplace\_front() and list::emplace\_back() in C++ STL](https://www.geeksforgeeks.org/listemplace_front-listemplace_back-c-stl/)– **emplace\_front()** function is used to insert a new element into the list container, the new element is added to the beginning of the list. **emplace\_back()** function is used to insert a new element into the list container, the new element is added to the end of the list.
* [list::clear() in C++ STL](https://www.geeksforgeeks.org/listclear-c-stl/)– **clear()**function is used to remove all the elements of the list container, thus making it size 0.
* [list::operator= in C++ STL](https://www.geeksforgeeks.org/listoperator-c-stl/)– This operator is used to assign new contents to the container by replacing the existing contents.
* [list::swap() in C++ STL](https://www.geeksforgeeks.org/listswap-c-stl/)– This function is used to swap the contents of one list with another list of same type and size.
* [list splice() function in C++ STL](https://www.geeksforgeeks.org/list-splice-function-in-c-stl/)– Used to transfer elements from one list to another.
* [list merge() function in C++ STL](https://www.geeksforgeeks.org/list-merge-function-in-c-stl/)– Merges two sorted lists into one
* [list emplace() function in C++ STL](https://www.geeksforgeeks.org/list-emplace-function-in-c-stl/)– Extends list by inserting new element at a given position.

----------------------------CONTAINER ADAPTERS-------------------------

1 )

Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front. Queues use an encapsulated object of [deque](https://www.geeksforgeeks.org/deque-cpp-stl/)or [list](https://www.geeksforgeeks.org/list-cpp-stl/)(sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

0. std::queue class is a container adapter that gives the programmer the functionality of a queue 1. queue is FIFO (first-in, first-out) data structure.

2. std::queue provides only specific set of functions.

3. std::queue allows to push(insert) at back and pop(remove) from front.

4. std::queue gives front, back, push, pop, empty, size.

2) ----------------🡪

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end (top) and an element is removed from that end only.  Stack uses an encapsulated object of either [vector](https://www.geeksforgeeks.org/vector-in-cpp-stl/)or [deque](https://www.geeksforgeeks.org/deque-cpp-stl/)(by default) or [list](https://www.geeksforgeeks.org/list-cpp-stl/)(sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

0. std::stack class is a container adapter that gives the programmer the functionality of a stack.

1. Internally it uses std::deque STL container.

2. It is LIFO (last-in, first-out) data structure.

3. std::stack allows to push(insert) and pop(remove) only from back.

FUNCTIONS PROVIDED:

empty() – Returns whether the stack is empty – Time Complexity : O(1)

size() – Returns the size of the stack – Time Complexity : O(1)

top() – Returns a reference to the top most element of the stack – Time Complexity : O(1)

push(g) – Adds the element ‘g’ at the top of the stack – Time Complexity : O(1)

pop() – Deletes the top most element of the stack – Time Complexity : O(1)

3) PRIORITY QUEUE : ------------------------------------🡪

Priority queues are a type of container adapters, specifically designed such that the first element of the queue is the greatest of all elements in the queue and elements are in non increasing order (hence we can see that each element of the queue has a priority {fixed order}).

0. std::priority\_queue is a container adaptor that provides constant time lookup of the largest OR smallest element.

1. By default std::vector is the container used inside.

2. Cost of insertion and extraction is logarithmic.

3. std::priority\_queue is implemented using std::make\_heap, std::push\_heap, std::pop\_heap functions.

4. by default its output values in desceding order (max heap tree)

ASSOCIATIVE CONTAINERS::

1 --SET -- Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

NOTES:

1. std::set is an Associative Container that contains a sorted set of unique objects of type Key.

2. It is usually implemented using Red-Black Tree.

3. Insertion, Removal, Search have logarithmic complexity.

4. If we want to store user defined data type in set then we will have to provide compare function so that set can store them in sorted order.

5. We can pass the order of sorting while constructing set object. BOTTOM LINE: It store unique elements and they are stored in sorted order (A/D)

**Methods of set:**

* [begin()](https://www.geeksforgeeks.org/setbegin-setend-c-stl/) – Returns an iterator to the first element in the set.
* [end()](https://www.geeksforgeeks.org/setbegin-setend-c-stl/) – Returns an iterator to the theoretical element that follows last element in the set.
* [rbegin()](https://www.geeksforgeeks.org/setrbegin-and-setrend-in-c-stl/)– Returns a reverse iterator pointing to the last element in the container.
* [rend()](https://www.geeksforgeeks.org/setrbegin-and-setrend-in-c-stl/)– Returns a reverse iterator pointing to the theoretical element right before the first element in the set container.
* [crbegin()](https://www.geeksforgeeks.org/set-crbegin-and-crend-function-in-c-stl/)– Returns a constant iterator pointing to the last element in the container.
* [crend()](https://www.geeksforgeeks.org/set-crbegin-and-crend-function-in-c-stl/)– Returns a constant iterator pointing to the position just before the first element in the container.
* [cbegin()](https://www.geeksforgeeks.org/set-cbegin-and-cend-function-in-c-stl/)– Returns a constant iterator pointing to the first element in the container.
* [cend()](https://www.geeksforgeeks.org/set-cbegin-and-cend-function-in-c-stl/)– Returns a constant iterator pointing to the position past the last element in the container.
* [size()](https://www.geeksforgeeks.org/setsize-c-stl/) – Returns the number of elements in the set.
* [max\_size()](https://www.geeksforgeeks.org/set-max_size-function-in-c-stl/) – Returns the maximum number of elements that the set can hold.
* [empty()](https://www.geeksforgeeks.org/setempty-c-stl/) – Returns whether the set is empty.
* [insert(const g)](https://www.geeksforgeeks.org/set-insert-function-in-c-stl/) – Adds a new element ‘g’ to the set.
* [iterator insert (iterator position, const g)](https://www.geeksforgeeks.org/set-insert-function-in-c-stl/) – Adds a new element ‘g’ at the position pointed by iterator.
* [erase(iterator position)](https://www.geeksforgeeks.org/seterase-c-stl/) – Removes the element at the position pointed by the iterator.
* [erase(const g)](https://www.geeksforgeeks.org/seterase-c-stl/)– Removes the value ‘g’ from the set.
* [clear()](https://www.geeksforgeeks.org/setclear-c-stl/) – Removes all the elements from the set.
* [key\_comp()](https://www.geeksforgeeks.org/setkey_comp-in-c-stl/) / [value\_comp()](https://www.geeksforgeeks.org/set-value_comp-function-in-c-stl/) – Returns the object that determines how the elements in the set are ordered (‘<‘ by default).
* [find(const g)](https://www.geeksforgeeks.org/set-find-function-in-c-stl/) – Returns an iterator to the element ‘g’ in the set if found, else returns the iterator to end.
* [count(const g)](https://www.geeksforgeeks.org/set-count-function-in-c-stl/) – Returns 1 or 0 based on the element ‘g’ is present in the set or not.
* [lower\_bound(const g)](https://www.geeksforgeeks.org/set-lower_bound-function-in-c-stl/) – Returns an iterator to the first element that is equivalent to ‘g’ or definitely will not go before the element ‘g’ in the set.
* [upper\_bound(const g)](https://www.geeksforgeeks.org/set-upper_bound-function-in-c-stl/) – Returns an iterator to the first element that will go after the element ‘g’ in the set.
* [equal\_range()](https://www.geeksforgeeks.org/set-equal_range-function-in-c-stl/)– The function returns an iterator of pairs. (key\_comp). The pair refers to the range that includes all the elements in the container which have a key equivalent to k.
* [emplace()](https://www.geeksforgeeks.org/setemplace-c-stl/)– This function is used to insert a new element into the set container, only if the element to be inserted is unique and does not already exists in the set.
* [emplace\_hint()](https://www.geeksforgeeks.org/set-emplace_hint-function-in-c-stl/)– Returns an iterator pointing to the position where the insertion is done. If the element passed in the parameter already exists, then it returns an iterator pointing to the position where the existing element is.
* [swap()](https://www.geeksforgeeks.org/setswap-c-stl/)– This function is used to exchange the contents of two sets but the sets must be of same type, although sizes may differ.
* [operator=](https://www.geeksforgeeks.org/set-operator-in-c-stl/)– The ‘=’ is an operator in C++ STL which copies (or moves) a set to another set and set::operator= is the corresponding operator function.
* [get\_allocator()](https://www.geeksforgeeks.org/set-get_allocator-in-c-stl/)– Returns the copy of the allocator object associated with the set.

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

2🡪  **MULTISET**

NOTES:

1. std::multiset is an Associative Container that contains a sorted set of duplicate objects of type Key.

2. It is usually implemented using Red-Black Tree.

3. Insertion, Removal, Search have logarithmic complexity.

4. If we want to store user defined data type in multiset then we will have to provide compare function so that multiset can store them in sorted order.

5. We can pass the order of sorting while constructing set object. BOTTOM LINE: multiset is similar to set except it can have multiple elements with same value

[begin()](https://www.geeksforgeeks.org/multiset-begin-and-end-function-in-c-stl/) – Returns an iterator to the first element in the multiset   
[end()](https://www.geeksforgeeks.org/multiset-begin-and-end-function-in-c-stl/) – Returns an iterator to the theoretical element that follows last element in the multiset   
[size()](https://www.geeksforgeeks.org/multiset-size-in-c-stl-with-examples/) – Returns the number of elements in the multiset   
[max\_size()](https://www.geeksforgeeks.org/multiset-max_size-in-c-stl/) – Returns the maximum number of elements that the multiset can hold   
[empty()](https://www.geeksforgeeks.org/multiset-empty-function-in-c-stl/) – Returns whether the multiset is empty

* a.erase() – Remove all instance of element from multiset having same value
* a.erase(a.find()) – Remove only one instance of element from multiset having same value.
* [pair insert(const g)](https://www.geeksforgeeks.org/multiset-insert-function-in-c-stl/) – Adds a new element ‘g’ to the multiset.
* [iterator insert (iterator position,const g)](https://www.geeksforgeeks.org/multiset-insert-function-in-c-stl/) – Adds a new element ‘g’ at the position pointed by iterator.
* [erase(iterator position)](https://www.geeksforgeeks.org/multiset-erase-in-c-stl/)– Removes the element at the position pointed by the iterator.
* [erase(const g)](https://www.geeksforgeeks.org/multiset-erase-in-c-stl/)– Removes the value ‘g’ from the multiset.
* [clear()](https://www.geeksforgeeks.org/multiset-clear-function-in-c-stl/)– Removes all the elements from the multiset.
* [key\_comp()](https://www.geeksforgeeks.org/multiset-key_comp-function-in-c-stl/) / [value\_comp()](https://www.geeksforgeeks.org/multiset-value_comp-method-in-c-stl/)– Returns the object that determines how the elements in the multiset are ordered (‘<‘ by default).
* [find(const g)](https://www.geeksforgeeks.org/multiset-find-function-in-c-stl/)– Returns an iterator to the element ‘g’ in the multiset if found, else returns the iterator to end.
* [count(const g)](https://www.geeksforgeeks.org/multiset-count-function-in-c-stl/)– Returns the number of matches to element ‘g’ in the multiset.
* [lower\_bound(const g)](https://www.geeksforgeeks.org/multiset-lower_bound-in-cpp-stl-with-examples/)– Returns an iterator to the first element that is equivalent to ‘g’ or definitely will not go before the element ‘g’ in the multiset if found, else returns the iterator to end.
* [upper\_bound(const g)](https://www.geeksforgeeks.org/multiset-upper_bound-in-cpp-stl-with-examples/)– Returns an iterator to the first element that is equivalent to ‘g’ or definitely will go after the element ‘g’ in the multiset if found, else returns the iterator to end.
* [multiset::swap()](https://www.geeksforgeeks.org/multisetswap-c-stl/)– This function is used to exchange the contents of two multisets but the sets must be of same type, although sizes may differ.
* [multiset::operator=](https://www.geeksforgeeks.org/multisetoperator-c-stl/)– This operator is used to assign new contents to the container by replacing the existing contents.
* [multiset::emplace()](https://www.geeksforgeeks.org/multisetemplace-c-stl/)– This function is used to insert a new element into the multiset container.
* [multiset equal\_range()](https://www.geeksforgeeks.org/multiset-equal_range-function-in-c-stl/)– Returns an iterator of pairs. The pair refers to the range that includes all the elements in the container which have a key equivalent to k.
* [multiset::emplace\_hint()](https://www.geeksforgeeks.org/multiset-emplace_hint-function-in-c-stl/)– Inserts a new element in the multiset.
* [multiset::rbegin()](https://www.geeksforgeeks.org/multiset-rbegin-and-rend-function-in-c-stl/)– Returns a reverse iterator pointing to the last element in the multiset container.
* [multiset::rend()](https://www.geeksforgeeks.org/multiset-rbegin-and-rend-function-in-c-stl/)– Returns a reverse iterator pointing to the theoretical element right before the first element in the multiset container.
* [multiset::cbegin()](https://www.geeksforgeeks.org/multiset-cbegin-and-cend-function-in-c-stl/)– Returns a constant iterator pointing to the first element in the container.
* [multiset::cend()](https://www.geeksforgeeks.org/multiset-cbegin-and-cend-function-in-c-stl/)– Returns a constant iterator pointing to the position past the last element in the container.
* [multiset::crbegin()](https://www.geeksforgeeks.org/multiset-crbegin-and-crend-function-in-c-stl/)– Returns a constant reverse iterator pointing to the last element in the container.
* [multiset::crend()](https://www.geeksforgeeks.org/multiset-crbegin-and-crend-function-in-c-stl/)– Returns a constant reverse iterator pointing to the position just before the first element in the container.
* [multiset::get\_allocator()](https://www.geeksforgeeks.org/multiset-get_allocator-function-in-c-stl/)– Returns a copy of the allocator object associated with the multiset.

#include<iostream>

#include<set>

using namespace std;

class person{

public:

int age;

string name;

bool operator < (const person & rhs) const {return age<rhs.age;}

bool operator > (const person & rhs) const {return age>rhs.age;}

};

int main(){

multiset<person,less<>>m1{{12,"sd"},{2,"Sd"}};

cout << "The elements in multiset are: ";

for( auto i: m1){

cout<<i.age<<" "<<i.name<<endl;

}

return 0;}

3) MAP:: Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have same key values.

1. std::map is associative container that store elements in key value combination where key should be unique, otherwise it overrides the previous value.

2. It is implement using Self-Balance Binary Search Tree (AVL/Red Black Tree) .

3. It store key value pair in sorted order on the basis of key (assending/decending).

4. std::map is generally used in Dictionay type problems.

Some basic functions associated with Map:  
[begin()](https://www.geeksforgeeks.org/mapbegin-end-c-stl/) – Returns an iterator to the first element in the map  
[end()](https://www.geeksforgeeks.org/mapbegin-end-c-stl/) – Returns an iterator to the theoretical element that follows last element in the map  
[size()](https://www.geeksforgeeks.org/mapsize-c-stl/) – Returns the number of elements in the map  
[max\_size()](https://www.geeksforgeeks.org/map-max_size-in-c-stl/) – Returns the maximum number of elements that the map can hold  
[empty()](https://www.geeksforgeeks.org/mapempty-c-stl/) – Returns whether the map is empty  
[pair insert(keyvalue, mapvalue)](https://www.geeksforgeeks.org/map-insert-in-c-stl/) – Adds a new element to the map  
[erase(iterator position)](https://www.geeksforgeeks.org/map-erase-function-in-c-stl/) – Removes the element at the position pointed by the iterator  
[erase(const g)](https://www.geeksforgeeks.org/map-erase-function-in-c-stl/)– Removes the key value ‘g’ from the map  
[clear()](https://www.geeksforgeeks.org/mapclear-c-stl/) – Removes all the elements from the map.

* [map insert() in C++ STL](https://www.geeksforgeeks.org/map-insert-in-c-stl/)– Insert elements with a particular key in the map container. .
* [map count() function in C++ STL](https://www.geeksforgeeks.org/map-count-function-in-c-stl/)– Returns the number of matches to element with key value ‘g’ in the map.
* [map equal\_range() in C++ STL](https://www.geeksforgeeks.org/map-equal_range-in-c-stl/)– Returns an iterator of pairs. The pair refers to the bounds of a range that includes all the elements in the container which have a key equivalent to k.
* [map erase() function in C++ STL](https://www.geeksforgeeks.org/map-erase-function-in-c-stl/)– Used to erase element from the container.
* [map rend() function in C++ STL](https://www.geeksforgeeks.org/map-rend-function-in-c-stl/)– Returns a reverse iterator pointing to the theoretical element right before the first key-value pair in the map(which is considered its reverse end).
* [map rbegin() function in C++ STL](https://www.geeksforgeeks.org/map-rbegin-function-in-c-stl-2/)– Returns a reverse iterator which points to the last element of the map.
* [map find() function in C++ STL](https://www.geeksforgeeks.org/map-find-function-in-c-stl/)– Returns an iterator to the element with key value ‘g’ in the map if found, else returns the iterator to end.
* [map crbegin() and crend() function in C++ STL](https://www.geeksforgeeks.org/map-crbegin-and-crend-function-in-c-stl/)– **crbegin()** returns a constant reverse iterator referring to the last element in the map container. **crend()** returns a constant reverse iterator pointing to the theoretical element before the first element in the map.
* [map cbegin() and cend() function in C++ STL](https://www.geeksforgeeks.org/map-cbegin-and-cend-function-in-c-stl/)– **cbegin()** returns a constant iterator referring to the first element in the map container. **cend()** returns a constant iterator pointing to the theoretical element that follows last element in the multimap.
* [map emplace() in C++ STL](https://www.geeksforgeeks.org/map-emplace-in-c-stl/)– Inserts the key and its element in the map container.
* [map max\_size() in C++ STL](https://www.geeksforgeeks.org/map-max_size-in-c-stl/)– Returns the maximum number of elements a map container can hold.
* [map upper\_bound() function in C++ STL](https://www.geeksforgeeks.org/map-upper_bound-function-in-c-stl/)– Returns an iterator to the first element that is equivalent to mapped value with key value ‘g’ or definitely will go after the element with key value ‘g’ in the map
* [map operator= in C++ STL](https://www.geeksforgeeks.org/map-operator-in-c-stl/)– Assigns contents of a container to a different container, replacing its current content.
* [map lower\_bound() function in C++ STL](https://www.geeksforgeeks.org/map-lower_bound-function-in-c-stl/)– Returns an iterator to the first element that is equivalent to mapped value with key value ‘g’ or definitely will not go before the element with key value ‘g’ in the map.
* [map emplace\_hint() function in C++ STL](https://www.geeksforgeeks.org/map-emplace_hint-function-in-c-stl/)– Inserts the key and its element in the map container with a given hint.
* [map value\_comp() in C++ STL](https://www.geeksforgeeks.org/map-value_comp-in-c-stl/)– Returns the object that determines how the elements in the map are ordered (‘<' by default).
* [map key\_comp() function in C++ STL](https://www.geeksforgeeks.org/map-key_comp-function-in-c-stl/)– Returns the object that determines how the elements in the map are ordered (‘<' by default).
* [map::size() in C++ STL](https://www.geeksforgeeks.org/mapsize-c-stl/)– Returns the number of elements in the map.
* [map::empty() in C++ STL](https://www.geeksforgeeks.org/mapempty-c-stl/)– Returns whether the map is empty.
* [map::begin() and end() in C++ STL](https://www.geeksforgeeks.org/mapbegin-end-c-stl/)– **begin()** returns an iterator to the first element in the map. **end()** returns an iterator to the theoretical element that follows last element in the map
* [map::operator[] in C++ STL](https://www.geeksforgeeks.org/map-operator-cpp-stl/)– This operator is used to reference the element present at position given inside the operator.
* [map::clear() in C++ STL](https://www.geeksforgeeks.org/mapclear-c-stl/)– Removes all the elements from the map.
* [map::at() and map::swap() in C++ STL](https://www.geeksforgeeks.org/mapat-mapswap-c-stl/)– **at()** function is used to return the reference to the element associated with the key k. **swap()** function is used to exchange the contents of two maps but the maps must be of same type, although sizes may differ.

----------------------------------MULTIMAP--------------------------------🡪>>>

1. Multimap is an associative container that contains a sorted list of key-value pairs, while permitting multiple entries with the same key.

2. It store key value pair in sorted order on the basis of key (assending/decending).

3. Data structure used in multimap is not defined by standard, but red-black tree is assumed by most of the people.

4. Lookup: count, find, contains, equal\_range, lower\_bound, upper\_bound

5. We dont have at() and [] functions to get element like we had in std::map.

Multimap is similar to [map](http://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/) with an addition that multiple elements can have same keys. Also, it is NOT required that the key value and mapped value pair has to be unique in this case. One important thing to note about multimap is that multimap keeps all the keys in sorted order always. These properties of multimap makes it very much useful in competitive programming.

* [multimap::operator= in C++ STL](https://www.geeksforgeeks.org/multimapoperator-c-stl/)– It is used to assign new contents to the container by replacing the existing contents.
* [multimap::crbegin() and multimap::crend() in C++ STL](https://www.geeksforgeeks.org/multimapcrbegin-and-multimapcrend-in-c-stl/)– **crbegin()** returns a constant reverse iterator referring to the last element in the multimap container. **crend()** returns a constant reverse iterator pointing to the theoretical element before the first element in the multimap.
* [multimap::emplace\_hint() in C++ STL](https://www.geeksforgeeks.org/multimapemplace_hint-in-c-stl/)– Inserts the key and its element in the multimap container with a given hint.
* [multimap clear() function in C++ STL](https://www.geeksforgeeks.org/multimap-clear-function-in-c-stl/)– Removes all the elements from the multimap.
* [multimap empty() function in C++ STL](https://www.geeksforgeeks.org/multimap-empty-function-in-c-stl/)– Returns whether the multimap is empty.
* [multimap maxsize() in C++ STL](https://www.geeksforgeeks.org/multimap-maxsize-in-c-stl/)– Returns the maximum number of elements a multimap container can hold.
* [multimap value\_comp() function in C++ STL](https://www.geeksforgeeks.org/multimap-value_comp-function-in-c-stl/)– Returns the object that determines how the elements in the multimap are ordered (‘<‘ by default)
* [multimap rend in C++ STL](https://www.geeksforgeeks.org/multimap-rend-in-c-stl/)– Returns a reverse iterator pointing to the theoretical element preceding to the first element of the multimap container.
* [multimap::cbegin() and multimap::cend() in C++ STL](https://www.geeksforgeeks.org/multimapcbegin-and-multimapcend-in-c-stl/)– **cbegin()** returns a constant iterator referring to the first element in the multimap container. **cend()** returns a constant iterator pointing to the theoretical element that follows last element in the multimap.
* [multimap::swap() in C++ STL](https://www.geeksforgeeks.org/multimapswap-c-stl/)– Swap the contents of one multimap with another multimap of same type and size.
* [multimap rbegin in C++ STL](https://www.geeksforgeeks.org/multimap-rbegin-in-c-stl/)– Returns an iterator pointing to the last element of the container.
* [multimap size() function in C++ STL](https://www.geeksforgeeks.org/multimap-size-function-in-c-stl/)– Returns the number of elements in the multimap container.
* [multimap::emplace() in C++ STL](https://www.geeksforgeeks.org/multimapemplace-in-c-stl/)– Inserts the key and its element in the multimap container.
* [multimap::begin() and multimap::end() in C++ STL](https://www.geeksforgeeks.org/multimapbegin-and-multimapend-in-c-stl/)–**begin()** returns an iterator referring to the first element in the multimap container. **end()** returns an iterator to the theoretical element that follows last element in the multimap.
* [multimap upper\_bound() function in C++ STL](https://www.geeksforgeeks.org/multimap-upper_bound-function-in-c-stl/)– Returns an iterator to the first element that is equivalent to multimapped value with key value ‘g’ or definitely will go after the element with key value ‘g’ in the multimap.
* [multimap::count() in C++ STL](https://www.geeksforgeeks.org/multimapcount-in-c-stl/)– Returns the number of matches to element with key value ‘g’ in the multimap.
* [multimap::erase() in C++ STL](https://www.geeksforgeeks.org/multimaperase-in-c-stl/)– Removes the key value from the multimap.
* [multimap::find() in C++ STL](https://www.geeksforgeeks.org/multimapfind-in-c-stl/)– Returns an iterator to the element with key value ‘g’ in the multimap if found, else returns the iterator to end.
* [multimap equal\_range() in C++ STL](https://www.geeksforgeeks.org/multimap-equal_range-in-c-stl/)– Returns an iterator of pairs. The pair refers to the bounds of a range that includes all the elements in the container which have a key equivalent to k.
* [multimap insert() in C++ STL](https://www.geeksforgeeks.org/multimap-insert-in-c-stl/)– Used to insert elements in the multimap container.
* [multimap lower\_bound() function in C++ STL](https://www.geeksforgeeks.org/multimap-lower_bound-function-in-c-stl/)– Returns an iterator to the first element that is equivalent to multimapped value with key value ‘g’ or definitely will not go before the element with key value ‘g’ in the multimap.
* [multimap key\_comp() in C++ STL](https://www.geeksforgeeks.org/multimap-key_comp-in-c-stl/)– Returns the object that determines how the elements in the multimap are ordered (‘<‘ by default)

multimap::operator= is used to assign new contents to the container by replacing the existing contents. It also modifies the size according to the new contents.

Input : multimap1 = { ('a', 1), ('b', 2), ('c', 3)}

multimap2 = { ('d', 4), ('e', 5), ('f', 6)}

multimap1 = multimap2;

Output : multimap1

d 4

e 5

f 6

The **multimap::equal\_range()** is a built-in function in C++ STL which returns an iterator of pairs. The pair refers to the bounds of a range that includes all the elements in the container which have a key equivalent to k. If there are no matches with key K, the range returned is of length 0 with both iterators pointing to the first element that has a key considered to go after k according to the container’s internal comparison object (key\_comp).

auto it = mp.equal\_range(1);

    cout << "The multimap elements of key 1 is : \n";

    cout << "KEY\tELEMENT\n";

    // Prints all the elements of key 1

    for (auto itr = it.first; itr != it.second; ++itr) {

        cout << itr->first

             << '\t' << itr->second << '\n';

    }

Emplace In STL

1. All the containers supports insert and emplace operation to store data.

2. Emplace is used to construct object in-place and avoids unnecessary copy of objects.

3. Insert and Emplace is equal for premetive data types but when we deal with heavy objects we should use emplace if we can for efficiency.

---------------------------------------------------------------------------------------------------------🡪>

UNORDERED ASSOSIATIVE CONTAINERS:

1)

UNORDERED SET:--------

An **unordered\_set** is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized. All operations on the **unordered\_set** takes constant time **O(1)** on an average which can go up to linear time **O(n)** in worst case which depends on the internally used hash function, but practically they perform very well and generally provide a constant time lookup operation. 

The **unordered\_set** can contain key of any type – predefined or user-defined data structure but when we define key of type user define the type, we need to specify our comparison function according to which keys will be compared.

0. Unorder Set is a associative container that contains set of unique objects.

1. Search, insertion, and removal have average constant-time complexity.

2. Internally, the elements are organized into buckets.

3. It uses hashing to insert elements into buckets.

4. This allows fast access to individual elements, since once a hash is computed, it refers to the exact bucket the element is placed into.

WHY UNORDERED SET?

0. maintain a collection of uniqe items with fast insertion and removal.

2) UNORDERED MULTISET ---------------------------------------------------------------🡪

0. std::unordered\_multiset is an associative container that contains set of possibly non-unique objects.

1. Search, insertion, and removal have average constant-time complexity.

2. Internally, the elements are organized into buckets.

3. It uses hashing to insert elements into buckets.

4. This allows fast access to individual elements, because after computing the hash of the value it refers to the exact bucket the element is placed into.

WHY UNORDERED\_MULTISET

0. maintain a collection of non-unique items with fast insertion and removal

* [insert()](https://www.geeksforgeeks.org/unordered_multiset-insert-in-c-stl/)– Inserts new elements in the unordered\_multiset. Thus increases the container size.
* [begin()](https://www.geeksforgeeks.org/unordered_multiset-begin-function-in-c-stl/)– Returns an iterator pointing to the first element in the container or to the first element in one of its bucket.
* [end()](https://www.geeksforgeeks.org/unordered_multiset-end-function-in-c-stl/)– Returns an iterator pointing to the position immediately after the last element in the container or to the position immediately after the last element in one of its bucket.
* [empty()](https://www.geeksforgeeks.org/unordered_multiset-empty-function-in-cstl/)– It returns true if the unordered\_multiset container is empty. Otherwise, it returns false.
* [find()](https://www.geeksforgeeks.org/unordered_multiset-find-function-in-cstl/)– Returns an iterator which points to the position which has the element val.
* [cbegin()](https://www.geeksforgeeks.org/unordered_multiset-cbegin-function-in-c-stl/)– Returns a constant iterator pointing to the first element in the container or to the first element in one of its bucket.
* [cend()](https://www.geeksforgeeks.org/unordered_multiset-cend-function-in-c-stl/)– Returns a constant iterator pointing to the position immediately after the last element in the container or to the position immediately after the last element in one of its bucket.
* [equal\_range()](https://www.geeksforgeeks.org/unordered_multiset-equal_range-function-in-cstl/)– Returns the range in which all the elements are equal to a given value.
* [emplace()](https://www.geeksforgeeks.org/unordered_multiset-emplace-function-in-c-stl/)– Inserts a new element in the unordered\_multiset container.
* [clear()](https://www.geeksforgeeks.org/unordered_multiset-clear-function-in-c-stl/)– Clears the contents of the unordered\_multiset container.
* [count()](https://www.geeksforgeeks.org/unordered_multiset-count-function-in-c-stl/)– Returns the count of elements in the unordered\_multiset container which is equal to a given value.
* [size()](https://www.geeksforgeeks.org/unordered_multiset-size-in-c-stl/)– The size() method of unordered\_multiset is used to count the number of elements of unordered\_set it is called with.
* [max\_size](https://www.geeksforgeeks.org/unordered_multiset-max_size-in-c-stl/)– The max\_size() of unordered\_multiset takes the maximum number of elements that the unordered\_multiset container is able to hold.
* [swap()](https://www.geeksforgeeks.org/unordered_multiset-swap-function-in-c-stl/)– Swaps the contents of two unordered\_multiset containers.
* [erase()](https://www.geeksforgeeks.org/unordered_multiset-erase-function-in-c-stl/)– Used to remove either a single element or, all elements with a definite value or, a range of elements ranging from start(inclusive) to end(exclusive).
* [bucket()](https://www.geeksforgeeks.org/unordered_multiset-bucket-function-in-c-stl/)– Returns the bucket number in which a given element is. Bucket size varies from 0 to bucket\_count-1.
* [bucket\_size()](https://www.geeksforgeeks.org/unordered_multiset-bucket_size-function-in-c-stl/)– Returns the number of elements in the bucket which has the element val.
* [reserve()](https://www.geeksforgeeks.org/unordered_multiset-reserve-in-c-stl/)– The reverse() function of unordered\_multiset sets the number of buckets in the container (bucket\_count) to the most appropriate to contain at least n elements.
* [max\_bucket\_count()](https://www.geeksforgeeks.org/unordered_multiset-max_bucket_count-function-in-c-stl/)– Returns the maximum number of buckets that the unordered multiset container can have.
* [load\_factor()](https://www.geeksforgeeks.org/unordered_multiset-load_factor-function-in-c-stl/)– Returns the current load factor in the unordered\_multiset container.
* [max\_load\_factor()](https://www.geeksforgeeks.org/unordered_multiset-max_load_factor-in-c-stl/)– Returns the maximum load factor of the unordered\_multiset container.
* [bucket\_count()](https://www.geeksforgeeks.org/unordered_multiset-bucket_count-function-in-c-stl/)– Returns the total number of buckets in the unordered\_multiset container.
* [hash\_function()](https://www.geeksforgeeks.org/unordered_multiset-hash_function-function-in-c-stl/)– This hash function is a unary function which takes a single argument only and returns a unique value of type size\_t based on it.
* [rehash()](https://www.geeksforgeeks.org/unordered_multiset-rehash-function-in-c-stl/)– Sets the number of buckets in the container to N or more.
* [key\_eq()](https://www.geeksforgeeks.org/unordered_multiset-key_eq-function-in-c-stl/)– Returns a boolean value according to the comparison.
* [emplace\_hint()](https://www.geeksforgeeks.org/unordered_multiset-emplace_hint-function-in-c-stl/)– Inserts a new element in the unordered\_multiset container.
* [get\_allocator](https://www.geeksforgeeks.org/unordered_multiset-get_allocator-in-c-stl/)– This function gets the stored allocator object and returns the allocator object which is used to construct the container.
* [operator =](https://www.geeksforgeeks.org/unordered_multiset-operator-in-c-stl/)– The ‘=’ is an operator in C++ STL which copies (or moves) an unordered\_multiset to another unordered\_multiset and unordered\_multiset::operator= is the corresponding operator function.

UNORDERED MAP-……………………………………………………….

0. std::unordered\_map is an associative container that contains key-value pairs with unique keys.

1. Search, insertion, and removal have average constant-time complexity.

2. Internally, the elements are organized into buckets.

3. It uses hashing to insert elements into buckets.

4. This allows fast access to individual elements, because after computing the hash of the value it refers to the exact bucket the element is placed into.

WHY UNORDERED\_MAP

1. maintain a collection of uniqe {key:value} pairs with fast insertion and removal.

* [at()](https://www.geeksforgeeks.org/std-unordered_map-c/): This function in C++ unordered\_map returns the reference to the value with the element as key k.
* [begin()](https://www.geeksforgeeks.org/unordered_map-begin-in-c/): Returns an iterator pointing to the first element in the container in the unordered\_map container
* [end()](https://www.geeksforgeeks.org/unordered_map-end-function-in-c-stl/): Returns an iterator pointing to the position past the last element in the container in the unordered\_map container
* [bucket():](https://www.geeksforgeeks.org/unordered_map-bucket-in-c-stl/) Returns the bucket number where the element with the key k is located in the map.
* [bucket\_count:](https://www.geeksforgeeks.org/stdbucket_count-stdbucket_size-unordered_map-cpp/) bucket\_count is used to count the total no. of buckets in the unordered\_map. No parameter is required to pass into this function.
* [bucket\_size:](https://www.geeksforgeeks.org/stdbucket_count-stdbucket_size-unordered_map-cpp/) Returns the number of elements in each bucket of the unordered\_map.
* [count()](https://www.geeksforgeeks.org/unordered_map-count-in-c/): Count the number of elements present in an unordered\_map with a given key.
* [equal\_range](https://www.geeksforgeeks.org/unordered_map-equal_range-in-c/): Return the bounds of a range that includes all the elements in the container with a key that compares equal to k.

ITERATORS::

<https://www.geeksforgeeks.org/introduction-iterators-c/>

std::copy : <https://www.geeksforgeeks.org/different-methods-copy-c-stl-stdcopy-copy_n-copy_if-copy_backward/>

pair in c++:

<https://www.geeksforgeeks.org/pair-in-cpp-stl/>

TUPLE:

<https://www.geeksforgeeks.org/tuples-in-c/>

decltype : <https://www.scaler.com/topics/cpp-decltype/>