STL : C++

1. Vector :-

1. Store element at contiguous position
2. Dynamically change its size in an efficient way (by reserving some more space)
3. Uses Dynamically allocated array to store elements.
4. Vector consume more memory than array.
5. Drawback :- Worse, when you try to delete or enter value at any other position than first & last pos.

Accessors :-

v = {5, 2, 9, 8, 1}

v[i] ⇒ get or set ith element ⇒ O(1) ⇒ 2 for i = 1
v.at(i) ⇒ " with bound checking ⇒ O(1) ⇒ 2 for i = 1
v.size() ⇒ get current no. of element ⇒ O(1) ⇒ 5
v.empty() ⇒ get True if vector is empty ⇒ O(1) ⇒ False
v.begin() ⇒ get random access iterator to start ⇒ " ⇒ 5 (element)
v.end() ⇒ get random access iterator to end ⇒ " ⇒ 1 (")
v.front() ⇒ get the first element ⇒ " ⇒ 5
v.back() ⇒ get the last " ⇒ " ⇒ 1
v.capacity() ⇒ get the max no. of element ⇒ " ⇒ 8

Date

Modifiers :-

v.push_back(val)	Add val to end	O(1)
v.insert(iterator, val)	Insert val at the pos entered by iterator, val can be as arr or val	O(n)
~~v.pop_back()~~		
v.pop_back()	remove from end	O(1)
v.erase(iterator)	remove val index by iterator	O(n)
v.erase(beg, end)	remove element from beg to end	O(n)

```
int main() {
    vector <int> g1;
    vector <int> :: iterator i;
    vector <int> :: reverse_iterator ir;

    for (int i = 1 ; i <= 5 ; i++)
        g1.push_back(i);

    for (i = g1.begin() ; i != g1.end(); ++i)
        cout << *i << '\t' ;
        ↳ 1 2 3 4 5

    for (ir = g1.rbegin() ; ir != g1.rend(); ir++)
        cout << '\t' << *ir;
        ↳ 5 4 3 2 1

}
```

Stack :-

It supports LIFO

Accessors :-

			(5,2,9,8,1)
s.top()	Ret top element	O(1)	5
s.size()	Ret current no. of element	"	5
s.empty()	Ret True if stack is empty	"	false

Modifiers :-

s.push(val)	Push val at top	O(1)
s.pop()	Pop val from top (doesnot rest anything)	O(1)

```
#include <iostream>
#include <stack>
using name space std;
int main(){
    stack <int> s1;
    s1.push (10);
      "    (20);
      "    (20);
      "    (5);
      "    (1);

    stack <int> q = s1;

    while (!q.empty()){   => 1 5 20 20 10
        cout<< q.top() << ' ' ;
        q.pop();
    }
    cout << s1.size() <<endl;  => 5
  ret 0;
}
```

Queue :-

Date ________

is a type of container adaptor, which follow FIFO (insertion at last, removal at first)

{standard container classes list & deque}

Accessors :-

			(5, 2, 9, 8, 1)
q.front()	Ret the front element	O(1)	5
q.back()	" " rear "	"	1
q.size()	" the current no. of "	"	5
q.empty()	True if empty.	"	false

Modifiers :-

q.push(val) : add val to end O(1) (5, 9, 2, 8) if val=8

q.pop() : remove from front O(1) (9, 2, 8)

Deque

Date

1. Provide functionality similar to vector but with efficient insertion & deletion at beg or at end.
2. No guarantee for contiguous storage.
3. The element of deque can be scattered in diff chunks of storage.
4. Operation that require frequent remove & insertion at pos diff that beg & end. deque perform worse than list & forward -list.
5. Deque can be expanded or contracted on both end (dynamic sizes);

Accessors :-

			(5, 2, 9, 8, 1)
d[i]	set or get ith element	O(1)	9 for i = 2
d.at(i)	" with bound checking	O(1)	9 for i = 2
d.size()	" current no. of ele	O(1)	5
d.empty()	True if empty	"	false
d.begin()	iterator to start	"	5 *(d.begin())
d.end()	iterator to end	"	1 *(d.end()-1)
d.front()	get first element	"	5
d.back()	get last "	"	1

Modifiers :-

d.push_front(val)	add val to front	O(1)
d.push_back(val)	" val to back	O(1)
d.insert(iterator, val)	same as vector	O(n)
d.pop_front()	remove val from front	O(1)
d.pop_back()	" val from end	O(1)
d.erase(iterator)	" val indexed by it	O(n)
d.erase(beg, end)	same as vector	O(n)

List :—

Date

1. are sequence container allow const. time insert & erase operation anywhere with in a sequence & iterate in both direction.
2. implemented as double-link-list
3. they are like forward_list just iterate in both direction.
4. used in sorting algo becoz of const time in insertion, deletion & moving element.
5. Drawback: No random access by pos., also they consume extra memory for linking.

Accessors:

(5, 2, 5, 6, 1)

l.size()	ret current no. of element	O(1)	5
l.empty()	true is empty	"	false
l.end()	ret bidirectional iterator to start	"	5
l.begin()	" " " to end	"	1
l.front()	ret first element	O(1)	5
l.back()	ret back element	O(1)	1

Modifiers:

l.push_front(val)
l.push_back(val)
l.insert(iterator, value)
l.pop_front()
l.pop_back()
l.erase(it)
l.erase(beg, end)

Same as previous

l.remove(value) remove all occurance of val, O(n)
l.remove_if(test) remove all element that satisfy test (class or function & return 0 or 1

Date

l.reverse () Reverse the list O(n)

l.sort () sort the list O(nlogn)

l.sort(comparison) sort with comparison function, take 2 element O(nlogn) as i/p if ret 0, means swap the items

bool comparison(int a, int b) {
return (a>b) ; }
{ this give reverse sorting.

l.merge(l2) merge sorted list

bool test (const int & val) {
return (value>4) ; } { remove val > 4 }

l.unique() remove duplicates

Scanned by CamScanner

Priority Queue :-

Date

1. It is a container adapter.
2. first element is greatest of all element.
3. it is based on heap.
4. container can be vector, deque
5. uses make_heap, push_heap & pop_heap;

Accessors :

q.top()	return biggest element	O(1)
q.size()	same	,,
q.empty()	same.	,,

Modifier :-

q.push(val)	max-heap insertion	O(log n)
q.pop()	remove biggest value	O(log n)

Set :-

Date

1. It is a container that store unique element following a specific order.
2. The val in set can't be modified but they can be inserted or remove from the container.
3. element in a set are alway sorted.
4. access individual element by their key.
5. They are implemented as binary search tree.

Accessors :-

s.find(key) Ret pointer pointing to s or O(logn)
s.end() if not found

s.lower_bound(key) (3, 5, 8, 9) ret 8 for key = 8 O(logn)
ret 5 for key = 4

s.upper_bound(key) ret 8 for key = 6
ret 9 for key = 8

both return a iterator

s.equal_range(key) Return pair (lowerbound(key) O(logn)
& upper_bound(key))

pair <int, int> (5, 8) for key = 7

s.count(key) ret 1 or 0 if O(logn)
key founds or not

s.size() size of tree O(1)
s.empty() true or false "
s.begin() iter for first element. "
s.end() ret iter one past last element "

element inserted in order : 5, 8, 9, 8, 3, 5
set = {3, 5, 8, 9} multiset (3, 5, 5, 8, 8, 9)

Date

1. Modifier:
→ does not matter what it points.

s.insert (iter, key) key is inserted in sorted order & iter to that key is returned. O(log n)

s.insert(key) return a pair, an iter of key & a bool val if key is inserted in set. O(log n)

s.erase(key) If present remove O(log n)

Map :-

Date

1. are associative container that store elements formed by a combination of a key value & a mapped value.
2. key are sorted & unique.
3. Type of key & value may differ.
4. Order of insertion is maintained & slower than unordered map.
5. implemented as binary search tree.

Accessors:-

m[key] Ret val for key or add default val for key if no key is present in map (ie 0 or null) O(log n)

m.find[key] Ret a iter point to pair of (k, v) or .end() if no key found. O(log n)
p = *(m.find(key))
p.first or p.second

m.lower_bound(key) for map O(log n)
((2,1),(3,5),(5,9),(8,4))
(5,9) for iter pointing if k=5
(2,1) ret iter " k=0
(8,4) " " k=8
m.end() " " k=9

m.upper_bound(key) O(log n)
(8,4) ret iter if key=5
(2,1) " if key=0
m.end() " if key=8

Scanned by CamScanner

Date

m.equal_range(key) Ret a pair containing upper & lower bound for key.

s (8,4) for item first } k = 2
(2,1) for item second
m.end()

s. m.size() same O(1)
m.empty() " "
m.begin() " "
m.end() " "

s Modifier :-

m[key] = val O(log n)
m.insert(pair) to insert (k,v) pair "
m.erase(key) If k is present remove "
key,value pair