STL Containers

Containers library

**1. Sequence containers**  
Sequence containers implement data structures which can be accessed sequentially.

1. array : (C++11) static contiguous array
2. vector : dynamic contiguous array
3. deque : double-ended queue
4. forward\_list (C++11) : singly-linked list
5. list : doubly-linked list

**2. Associative containers**  
Associative containers implement sorted data structures that can be quickly searched (O(log n) complexity).

1. set : collection of **unique keys**, sorted by keys
2. map : collection of key-value pairs, sorted by keys, **keys are unique**
3. multiset : collection of keys, sorted by keys
4. multimap : collection of key-value pairs, sorted by keys

**3. Unordered associative containers**  
Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (O(1) amortized, O(n) worst-case complexity).

1. unordered\_set : collection of unique keys, hashed by keys
2. unordered\_map : collection of key-value pairs, hashed by keys, keys are unique
3. unordered\_multiset : collection of keys, hashed by keys
4. unordered\_multimap : collection of key-value pairs, hashed by keys

**4. Container adaptors**  
Container adaptors provide a different interface for sequential containers.

1. stack
2. queue
3. priority\_queue

Now, lets look for the time complexities of containers

1. Priority Queue

2. Map : Time Complexities mentioned below are for Ordered Map.

3. Set : Time Complexities mentioned below are for Ordered Set.

4. Stack

5. Queue

6. Vector

7. List

| **Sr. No** | **Data Structure** | **Sub Type** | **Syntax** | **Operations** | **Time Complexity** | **Space Complexity** | **Comments** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | Priority Queue | Max Heap | priority\_queue<data\_type> Q | Q.top() | O(1) | O(1) |  |
|  |  | Min Heap | priority\_queue<data\_type, vector<data\_type>, greater<data\_type>> Q | Q.push() | O(log n) | O(1) |  |
|  |  |  |  | Q.pop() | O(log n) | O(1) |  |
|  |  |  |  | Q.empty() | O(1) | O(1) |  |
| 2 | Map | Ordered Map | map <int, int> M | M.find(x) | O(log n) | O(1) | The map <int, int> M is the implementation of **self-balancing Red-Black Trees.** |
|  |  | Unordered Map | unordered\_map<int, int> M | M.insert(pair<int, int> (x, y) | O(log n) | O(1) | The unordered\_map<int, int> M is the implementation of Hash Table which makes the complexity of operations like insert, delete and search to Theta(1). |
|  |  | Ordered Multimap | multimap<int, int> M | M.erase(x) | O(log n) | O(1) | The multimap<int, int> M is the implementation of **Red-Black Trees which are self-balancing** trees making the cost of operations the same as the map. |
|  |  | Unordered Multimap | unordered\_multimap<int, int> M | M.empty( ) | O(1) | O(1) |  |
|  |  |  |  | M.clear( ) | Theta(n) | O(1) |  |
|  |  |  |  | M.size( ) | O(1) | O(1) |  |
| 3 | Set | Ordered set | set<data\_type> S | s.find( ) | O(log n) | O(1) | Set (set s) is the implementation of **Binary Search Trees**. |
|  |  | Unordered set | unordered\_set S | s.insert(x) | O(log n) | O(1) | Unordered set (unordered\_set S) is the implementation of **Hash Table**. The complexity becomes Theta(1) and O(n) when using unordered the ease of access becomes easier due to Hash Table implementation. |
|  |  | Ordered Multiset | multiset S | s.erase(x) | O(log n) | O(1) | Multiset (multiset S) is implementation of **Red-Black trees.** |
|  |  | Unordered Multiset | unordered\_multiset S | s.size() | O(1) | O(1) | Unordered\_multiset(unordered\_multiset S) is implemented the same as the unordered set but uses an extra variable that keeps track of the count. |
|  |  |  |  | s.empty( ) | O(1) | O(1) |  |
| 4 | Stack |  | stack<data\_type> A | s.top( ) | O(1) | O(1) | Stack is implemented using the **linked list** implementation of a stack. |
|  |  |  |  | s.pop( ) | O(1) | O(1) |  |
|  |  |  |  | s.empty( ) | O(1) | O(1) |  |
|  |  |  |  | s.push(x ) | O(1) | O(1) |  |
| 5 | Queue |  | queue<data\_type> Q | q.push(x) | O(1) | O(1) | Queue in STL is implemented using a **linked list**. |
|  |  |  |  | q.pop( ) | O(1) | O(1) |  |
|  |  |  |  | q.front( ) | O(1) | O(1) |  |
|  |  |  |  | q.back( ) | O(1) | O(1) |  |
|  |  |  |  | q.empty( ) | O(1) | O(1) |  |
|  |  |  |  | q.size( ) | O(1) | O(1) |  |
| 6 | Vector | 1D vector | vector A | sort(v.begin( ), v.end( )) | Theta(nlog(n)) | Theta(log n) | Vector is the implementation of **dynamic arrays** and uses new for memory allocation in heap. |
|  |  |  |  | reverse(v.begin( ), v.end( )) | O(n) | O(1) |  |
|  |  |  |  | v.push\_back(x) | O(1) | O(1) |  |
|  |  |  |  | v.pop\_back() | O(1) | O(1) |  |
|  |  | 2-dimensional vector | vector<vector> A | v.size() | O(1) | O(1) |  |
|  |  |  |  | v.clear() | O(n) | O(1) |  |
|  |  |  |  | v.erase() | **O(n^2)** | O(1) |  |
| 7 | List |  | list<data\_type> L | L.emplace\_front(val) | O(1) |  | Constructs and insert element at the beginning of the list. |
|  |  |  |  | L.emplace\_back(val) | O(1) |  | Constructs and insert element at the end of the list. |
|  |  |  |  | L.push\_front(val) | O(1) |  | Insert element at the beginning of the list. |
|  |  |  |  | L.push\_back(val) | O(1) |  | Insert element at the end of the list. |
|  |  |  |  | L.pop\_front() | O(1) |  | Delete element at the beginning of the list. |
|  |  |  |  | L.pop\_back() | O(1) |  | Delete element at the end of the list. |
|  |  |  |  | L.insert(iterator, val) | O(1) |  | Insert element at the specified position. |
|  |  |  |  | L.erase(iterator) | O(1) |  | Delete element from specified position. |
|  |  |  |  | L.clear() | **O(n)** |  | Erase all the elements from list. |