List Container

Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.

- ✓ **List containers are implemented as doubly-linked lists; a link to the element preceding it and a link to the element following it.**
- ✓ **They are very similar to forward_list.**
- ✓ **The main difference being that forward_list objects are single-linked lists, and thus they can only be iterated forwards, in exchange for being somewhat smaller and more efficient.**

List Container

front()	Returns the value of the first element in the list.
back()	Returns the value of the last element in the list.
push_front(g)	Adds a new element ‘g’ at the beginning of the list.
push_back(g)	Adds a new element ‘g’ at the end of the list.
pop_front()	Removes the first element of the list, and reduces size of the list by 1.
pop_back()	Removes the last element of the list, and reduces size of the list by 1.

Forward List Container

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 allow time constant insertion and removal of elements.

1. assign(): This function is used to assign values to the forward list, its other variant is used to assign repeated elements and using the values of another list.
2. push_front(): This function is used to insert the element at the first position on forward list. The value from this function is copied to the space before first element in the container. The size of forward list increases by 1.
3. pop_front(): This function is used to delete the first element of the list.
4. insert_after(): This function gives us a choice to insert elements at any position in forward list. The arguments in this function are copied at the desired position.
5. erase_after(): This function is used to erase elements from a particular position in the forward list. There are two variants of 'erase after' function.

Dequeue Container

Double-ended queues are sequence containers with the feature of expansion and contraction on both ends. They are similar to vectors, but are more efficient in case of insertion and deletion of elements. The functions for deque are same as vector, with an addition of push and pop operations for both front and back.

```
q.size() : 4
q.max_size() : 4611686018427387903
q.at(2) : 10
q.front() : 15
q.back() : 30
q.pop_front() : 20 10 30
q.pop_back() : 20 10
```

Stack Container

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.

The functions associated with stack are:

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)

Queue Container

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.

Priority Queue Container

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}).

We can pass another parameter while creating the priority queue to make it a min heap.
C++ provides the below syntax for the same.

Syntax:

```
priority_queue <int, vector<int>, greater<int>> g = q;
```

Set Container

Sets are a type of associative containers in which each element has to be unique because the value of the element identifies it. The values are stored in a specific order.

- ✓ The set stores the elements in **sorted** order.
- ✓ All the elements in a set have **unique values**.
- ✓ 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**.
- ✓ Sets follow the **Binary search tree** implementation.
- ✓ The values in a set are **unindexed**.

Multiset Container

Multisets are a type of associative containers similar to set, with an exception that multiple elements can have same values.