STL : C++ : part 2 :

lists

```
list<int> ls;

ls.push_back(2);        // {2}

ls.emplace_back(4);     // {2,4}
cheap← ls.push_front(5);    // {5,2,4};   (Insert in a vector is costlier)

ls.emplace_front(1);    // {1,5,2,4}

// rest begin, end, rbegin, rend, clear, insert, size, swap
   are same as vectors
```

Deque

```
deque <int>dq;
```

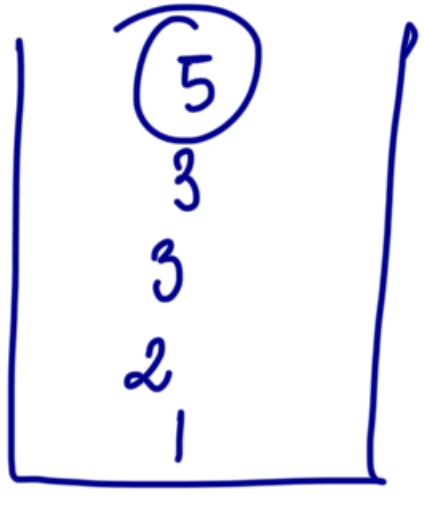
```
dq.push_back(1);      → {1}
dq.emplace_back(2);   → {1,2}
dq.push_front(4);     → {4,1,2}
dq.emplace_front(3);  → {3,4,1,2}
dq.pop_back();        → {3,4,1}
dq.pop_front();       → {4,1}
dq.back();            → 1
dq.front();           → 4
```

list: cbegin, end, rbegin, rend, clear, insert, size, swap are same as vectors

Stack: last in first out (LIFO)

```
stack<int> st;
st.push(1);
st.push(2);
st.push(3);
st.push(3);
st.emplace(5);

cout<<st.top(); → 5
```



indexing not allowed

3 functions: push, pop, top → O(1)

all things happen

st.pop(); // 5 is deleted in constant time

st.top(); → 3
cout << st.size(); → 4
cout << st.empty(); → false

stack <int> st1, st2;
st1.swap(st2);

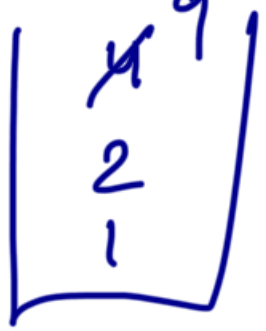
#Queue : Fifo : first in first out

queue <int> q;

q.push(1); q.push(2); q.emplace(4);

q.back() += 5;

cout << q.back(); → 9
cout << q.front(); → 1
q.pop → 9 2



// size, swap, empty same as stack

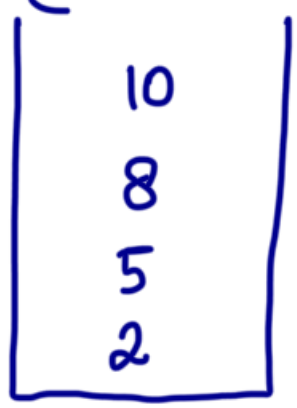
cout << q.front(); → 2 // all operations in constant time

---

priority Queue :

```
priority_queue <int> pq;
pq.push(5);
pq.push(2);
pq.push(8);
pq.push(10);   {10, 8, 5, 2}
```

(max heap)



tree data structure, not Linear.

```
cout << pq.top(); → 10

pq.pop(); →   8
              5
              2
```

push, top, pop : main
size, swap, empty : side func.

// Minimum heap → minimum priority queue

```
priority_que <int, vector<int>, greater<int>> pq;
pq.push(5);
pq.push(2);
```

pq.push(8);
pq.emplace(10);

8
10

push, pop → O(logn)
top → O(1)

cout << pq.top(); ⟶ 2

// Set : Sorted order, unique — O(log N)

```
set<int> st;
st.insert(1);      {1}
st.emplace(2);     {1,2}
st.insert(2);      {1,2}
st.insert(4);      {1,2,4}
st.insert(3);      {1,2,3,4}
```

4
3
2
1

tree, not a linear container

// {1,2,3,4,5}

(begin(), end(), rbegin(), rend(), size())

auto it = st.find(3) → returns an iterator that points to 3

auto it = st.find(6) → returns st.end()

st.erase(5) //erases 5, logarithmic time

int cnt = st.count(1); → 1
                       → 0

auto it = st.find(3);
st.erase(it); // constant time

// {1, 2, 3, 4, 5}
auto it1 = st.find(2);
auto it2 = st.find(4);
st.erase(it1, it2); [first, last)

(size, empty, swap, begin : vector)

// lower_bound() & upper_bound() work in the same way as in vector it does.

auto it = st.lower_bound(2);
auto it = st.upper_bound(3);

(do more after learning binary search)