Dequeue:-

It provides facility of both stack and queue.

In stack we can insert or delete from only one end and in queue insert from one end and delete from other end but in deque you can insert or delete from any one of two ends, as you required.

In c++ deque is implimented in other way then other languages. In other languages dequeue is implimented as doubly linked list but in C++ STL it provide random access.

dequeue is different from vector in it's implementation. They seem very similar but not.

Vector provide insertion and deletion in middle and back but dequeue provide insertion in middle, back and front too.

vector elements are continously stored in memmory but in dequeue elements are devided into chunks and in each chunks elements are continuous but these chunks are not they are at different location and connected.

but deque provide random acess for element.

now what we should use dequeue or vector, it dependens upon problem.

If we need container in which insertion and deletion is requised at both ends we will use dequeue but if it need only at end or middle we can use vector.

Also in deque insertion or delection at front or end is O(1) operation.

• ~~Initialisation and~~

Initialisation and traversal is same as vector container as it also supports random access.

• Functions supported by dequeue (main ones) -

• push_front(), • pop_front()

• push_back() • pop_back()

• insert() • erase() • size()

• clear() • empty()

• front() • back()

• begin() • end()

It is defined in <deque> header file.

We can use dequeue to design a data structure in which we can use these operation -

• insert_min() → inserts minimum element
• insert_max() → inserts maximum element
• get_max() → get current maximum "
• get_min() → " " minimum "

- extract_min() :- remove current min. ele.
- extract_max() :- " " max. ele.

For this we can use dequeue, we always insert & get min element from front and max. element from end().

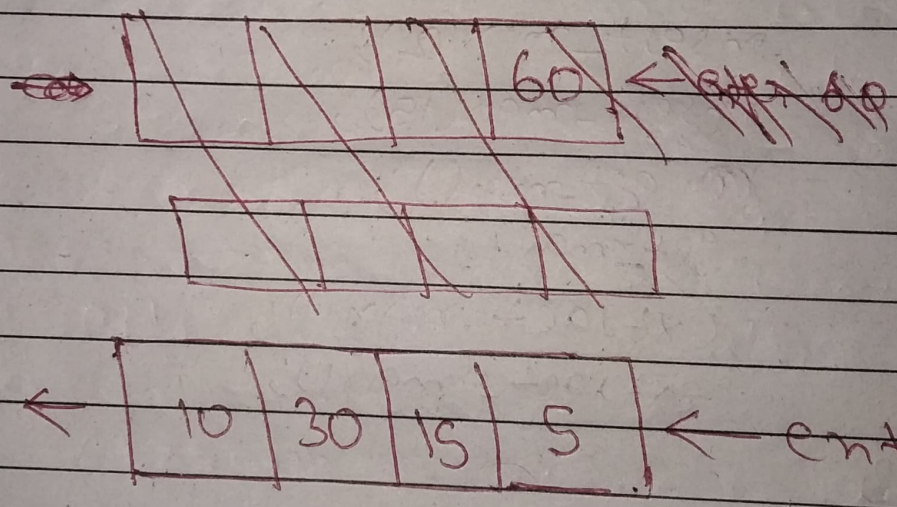
```
deque<int> dq;
int main()
{
    deque<int> dq;
    insert_min(2);
    insert_max(15);
    get_min();
    get_max();
    extract_min();
    extract_max();
}

void insert_min(int x){ dq.push_front(x);}
void insert_max(int x){dq.push_back(x);}
int get_min(){ return dq.front();}
int get_max(){ return dq.back();}

int extract_min(){return dq.pop_front();}
int extract_max(){return dq.pop_back();}
```

→ Queue :-

Queue data structure is FIFO type, First in First out.

In this data structure, insertion happens from one end and deletion or extraction happends from other end.

ex. -



- Creation of queue :-

```
queue <int> q; // creating empty
                  queue with
                  name 'q'
```

• Traversing in queue :-

Since queue don't have any iterator to traverse in a queue we have to remove elements one by one after going through them.

for example,

```
queue<int> q;
q.push(10);
q.push(30);
q.push(7);
//Traversing while(q.empty() == false())
{
    cout<<q.front()<<" "<<q.back
    <<endl;
    q.pop(); //removing ele.
}
```

• Functions provides by queue :-

• Push() :-

function pushes the element from back. Take O(1) TC.

- pop():-

Removes element (one) from front. Take O(1) T.C.

- front():-

Returns reference of front element. O(1) T.C.

- back():-

Returns reference of back element. O(1) T.C.

- empty():-

Checks where queue is empty or not. return true if empty or false if not. O(1) T.C.

- size():-

Returns no. of element in queue. O(1) T.C.

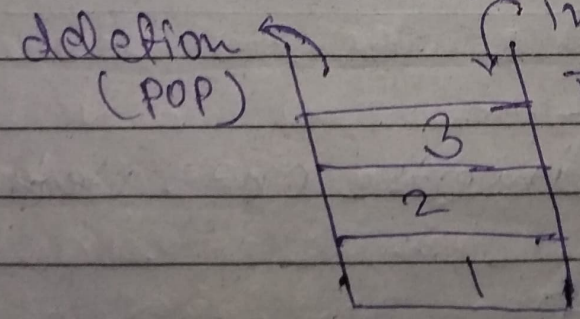
As we know queue is a container adapter, so, it must be implimented on other containers who provides these operations in O(1) T.C.

- pop_front()
- push_back()
- front()
- back()
- empty()
- size()

There are two containers, who can do this list and dequeue. Vector doesn't have pop_front() function in O(1) T.C.

→ Stack :-

Stack is another container adaptor. In C++ STL it work on LIFO (last in first out) rule. Insertion and deletion happens from same and one side of data structure, which is top side.



Declaration of stack:-

```
stack<int> s;
// declaring empty stack.
```

~~stack<int> s{~~

Traversal of stack:-

Since, stack is container adaptor, it doesn't have iterators, so, to traverse through it, we call Top element then remove/pop top element, until stack is empty.

```
for(s.empty() == false)
{
    cout<<s.top()<<" ";
    s.pop();
}
```

function associated with stack:-

- Push():-
  pushes/inserts one item at top.

- pop():-
  removes top item from stack if there.

- top():-
  returns reference of top item.

- size():-
  Tells no of elements in stack.

- empty():-
  checks if stack is empty or not.
  if empty returns True,
  if not returns false.

All these 3 functions have time complexity of O(1).

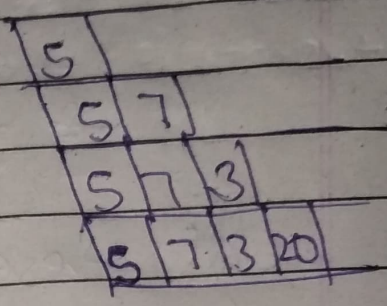
As stack is container adaptor, it is implemented upon other containers who provides these operations:-

```
back();
empty();
size();
push_back();
pop_back();
```

vector, list and dequeue provides these 3 operations so, it can implemented on basis of these.

NOTE:- As we see, when we insert some element in stack and then traverse it, we get them in reverse order of their insertion, this can be helpful in case we want to reverse or rotate some elements.

```
stack<int> s;

s.push(5);
s.push(7);
s.push(3);
s.push(20);
```




```
for(s.empty() == false)
{
    cout<< s.top()<<" ";
    s.pop();
}
```

Output:-

20 3 7 5

We don't use stack to reverse. Coz in every senario, as in vector, array, string, dequeue like container, reverse function can be used and this function is O(1) T.C. and takes no extra space but while using stack, it

requires extra space.

So, we use stack to reverse container like list and forward list, as list has only forward container and they don't has random access.

→ priority_queue :-

Priority_queue is another container adapter (concept of) implimented using heap data-structure, but uses vector as underlying container.

When we create priority_queue in C++ STL it is max heap by default. It means maximum element will always be at top.

When we remove top, we get max element of container also ~~container~~ elements changes position such that next max element comes on top.

Declairing empty priority_queue :-

```
Priority_queue <int> Pq;
```

priority_queue is defined in <queue> header file.

Functions used with priority_queue

- push():-

It pushes element into p.queue and makes sure max element is always at top by suffling the element if needed.

- pop():-

removes top max element from priority_queue and sets next max element as top.

- top():-

Returns reference of top element of priority queue.

- size():-

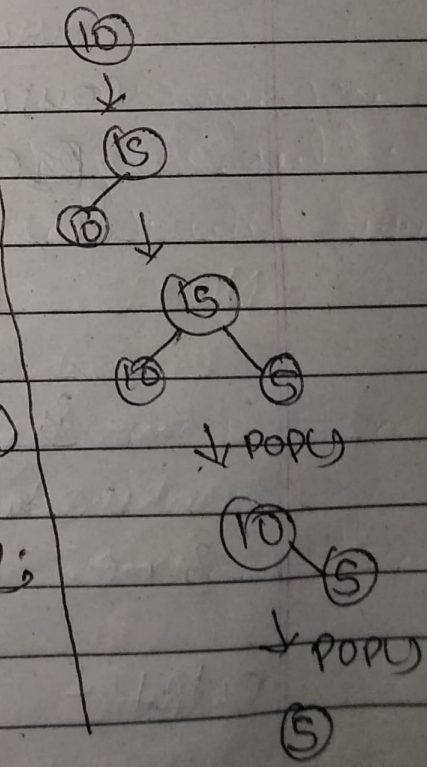
Gives size of p.queue in terms of no. of elements.

• empty() :-

Returns true if p-queue is empty, else false.

ex.-

```
#include<iostream>
#include<queue>

using namespace std;

int main(){
    priority_queue<int> pq;
    pq.push(10);
    pq.push(15);
    pq.push(5);
    cout<<pq.size()<<" ";
    cout<<pq.top()<<" ";
    while(pq.empty()==false)
    {
        cout<<pq.top()<<" ";
        pq.pop();
    }
    return 0;
}
```



output :-

3 15 15 10 5

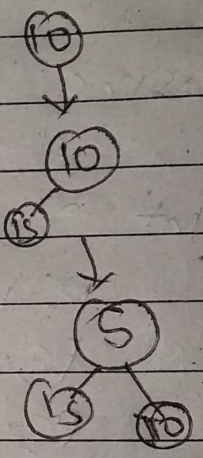
Creating minheap using priority queue

- Using different syntax:-

same.

```
int main(){
    priority_queue<int,vector<int>, greater<int>> pq
```

// It is C++ syntax to create min heap using
// priority queue.



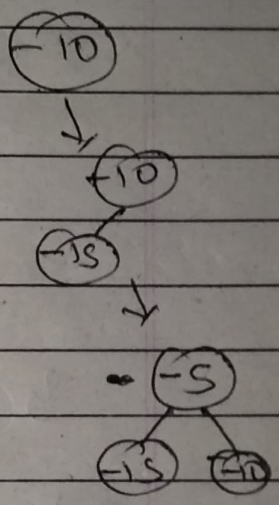

```
    pq.push(10);
    pq.push(15);
    pq.push(5);
    cout<<pq.size()<<" ";
    cout<<pq.top()<<" ";
    pq.pop();
    for(pq.empty()==false){
        cout<<pq.top()<<" ";
        pq.pop();
    }
    return 0;
}
```

Output:-

3 5 5 10 15

- Using max heap syntax but instead of passing 'x' pass '-x' and while getting top element use (-pq.top()) :-

```
int main(){
    priority_queue<int> pq;
    pq.push(-10);
    pq.push(-15);
    pq.push(-5);
    cout<<pq.size()<<" ";
    cout<<(-pq.top())<<" ";
    for(pq.empty()==false)
    {
        cout<<(-pq.top())<<" ";
        pq.pop();
    }
    return 0;
}
```



Output:-

3 5 5 10 15

NOTE:- Second method is simple but will not work in all cases.

Creating priority queue using existing vector / array :-

```
int main(){

    int arr[] = {10, 5, 15};
    priority_queue<int> pq(arr, arr+3);

    while(pq.empty() == false){
        cout<<pq.top()<<" ";
        pq.pop();
    }

    return 0;
}
```

Output :- 15 10 5

NOTE :- This method of initialising a priority queue is better than pushing one by one, it take O(n) times while pushing one by one takes much more time.

Func	T.C
top()	O(1)
empty()	O(1)
size()	O(1)
pop()	O(log n)
push()	O(log n)

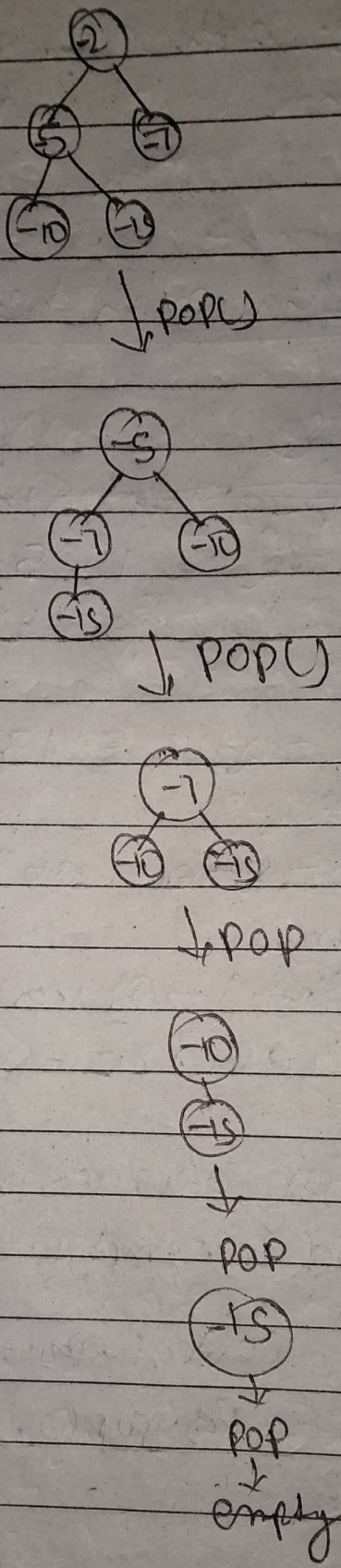
creating pq with vector or array O(n)

Implimenting min heap of array using pq :-

```
int main(){
    priority_queue<int> pq;
    arr[] = {10, 15, 5};
    priority_queue<int> pq(arr)

    arr[] = {10, 5, 15, 2, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    for(int i=0; i<n; i++)
        arr[i] = -arr[i];

    priority_queue<int> pq(arr, arr+n);
    while(pq.empty() == false){
        cout<<(-pq.top())<<" ";
        pq.pop();
    }
return 0;
}
```

Output:-

2 5 7 10 15


-2
-5
-1
-10
-3
pop()
-5
-7
-10
-15
pop()
-7
-10
-15
pop
-10
-15
pop
-15
pop
empty

Also if you are creating a priority queue of pairs, then pairs will be arranged on basis of first by default, to change the order we have to write our comparision function or use greater function. Same in case of structure.

When we write our own camparision function we have to do operator overloading.

All programmes of this type is later on this copy.