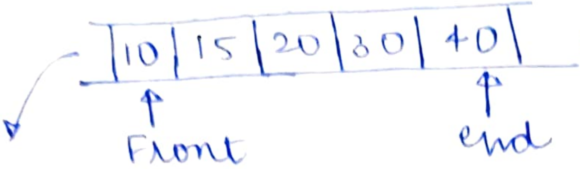
QUEUE

FIFO – first in first out

(Abstract DT)



← enqueue()

Push something to the queue.

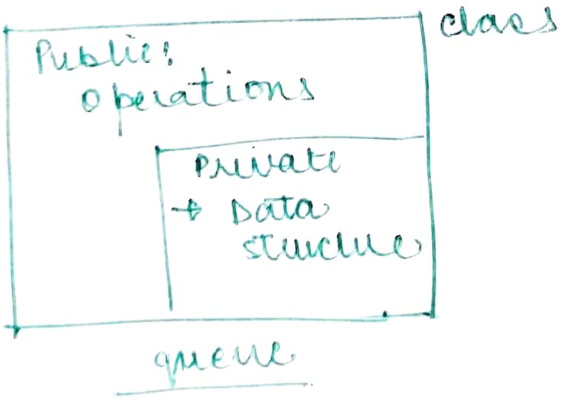
dequeue()

Pop from the queue.

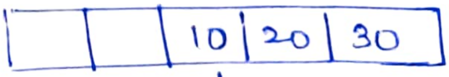
Operations

- enqueue(x) ← pushing x queue
- dequeuel ← Removing element from back of queue
- getFront() ← get the element Front → the item which is going to be remove next
- getRear() → get the element Rear → the iteme which is inserted last is called rear.
- size → returns the size of queue
- is Empty () → returns true if queue is empty)

Implementation



- We have to maintain two variables namely front and rear.
- Suppose after enqueue & dequeue operations, this is the status of queue



one way → shift all the elements by k spaces (k = number of free spaces at front).

other way → We should try to implement queue in circular manner.

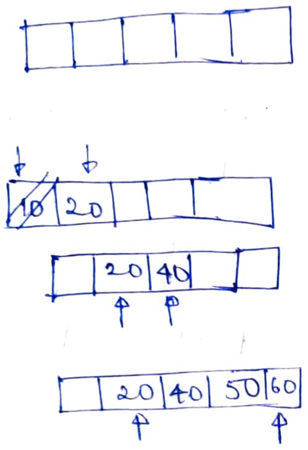
① In the 1st option - the complexity will by O(k)

② implementing 2nd option - updating indices in circular manner.

~~ex~~ cap = 5

enqueue (10)
enqueue (20)
dequeue ();
enqueue (40);
dequeue ();
enqueue (50)
———(60)
———(70)

(Error!) as the nextIndex has reached to cap-1



Ideally we should mode nextIndex to 0 ~~i.e~~ (circular)

Enqueue()

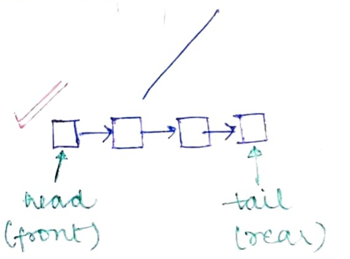
```
void enqueue (int el){
    if (~~isEmpty()~~ size == cap){
        cout << "queue is full";
        return;
    }
    queue[nextIndex] = el;
    nextIndex = (nextIndex + 1)% capacity;
    if (firstIndex == -1)
        firstIndex = 0;

    size++;
}
```

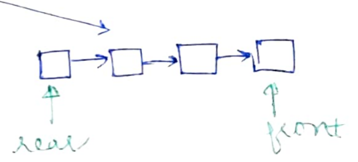
equeue()

```
int dequeue(){
    if (isEmpty()){
        cout<<" stack is empty! <<endl;
    }
    * int res = queue[firstIndex];
    firstIndex = (firstIndex+1)% capacity;
    size--;
    if( size == 0){
        firstIndex = -1;
        nextIndex = 0;
    }
    return ~~as~~ res;
}.
```

Using Linked List

we have two choices





- Insertion at rear O(1)
- deletion at front (O(1))

(Better choice)

* Insertion of rear O(1)
* deletion at front O(N)

Queue in STL

- queue<datatype> queuename;

  int; char; float..

Operations

① q.front() → Returns the element which is going to be poped next

② q.back() → Returns the element which is inserted at last

③ q.pop() → front will be deleted (dequeue)

④ q.push() → adds an item at end (enqueue)

⑤ q.size() → Returns size of queue

⑥ empty() → Returns if

internally queue is a container adaptor which uses dequeue. stack also uses deque if not specified

Implement stack using queue

we will use an auxilary queue for the purpose.

we want to maintain lifo pattern so any element must be inserted in the front of queue.

q1 - | 10 | 20 | 30 | | | |

enqueue (45)

q2 → | 45 | | | |

q1 → | 10 | 20 | 30 | |

q2 → | 45, 10, 20, 30 |

More variables

① Implement stack using ~~array~~ queue by making pop operation costly.

② Implement stack using 1 queue (using Recursion call stack).

③ Implement queue using stack.

Reversing a Queue

I/P → q = {10, 5, 15, 20}. O/P → {20, 15, 5, 10}

front rear front rear.

① put items of queue in the stack and then pop (it will reverse the order)

```
void reverse(queue<int> q){
    stack <int> s;
    while(q.empty()==false){
        s.push(q.front());
        q.pop();
    }.
    while(s.empty() == false){
        q.push(s.top());
        s.pop();
    }
}.
```

② Recursive

suppose we have reversed n-1th queue

| 10 | 30 | 20 | |

rever

and dequeue 10 & enqueue it

30 20 10 ← Reverse.

```
void reverse (queue <int> q){
      if (q.empty == true)
        return;

      int x = q.top()
      q.pop();
      reverse (q);
      q. push (x)
}.
```

Recursively Reverses q.queue.

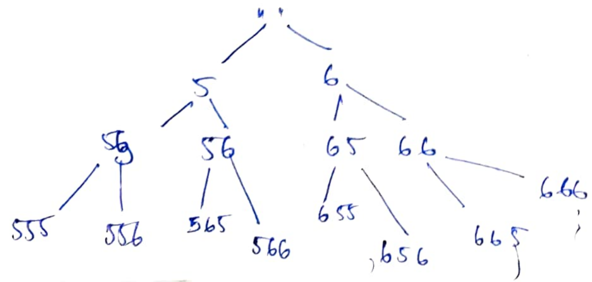
Generate Numbers with given Digit

digits → {5,6}

numbers → 5, 6, 55, 56, 65, 66, 555, 556 ---

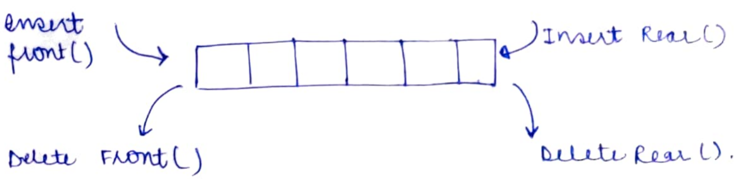
we can use recursive method + queue

```
void printFirstN (int n){
      queue <int> q;
      q. push ('5');
      q. push ('6');

      for (int i=0; i<count ; i++){
            string curr = q.top();
            cout << curr;
            q.pop();
            q. push (curr + '5');
            q. push (curr + '6');
      }
}.
```



Deque~~ue~~

- insertion and ~~deltion~~ deletion at both ends.



Operations

① getFront() → Returns the front element (inserted first)

② get Rear() → Returns the rear element (inserted last)

③ isFull() → True if queue is full.

④ is Empty() → True if queue is empty()

⑤ size() → Returns size of queue.

Empty deque

① insertFront (10);

② insertFront (20);

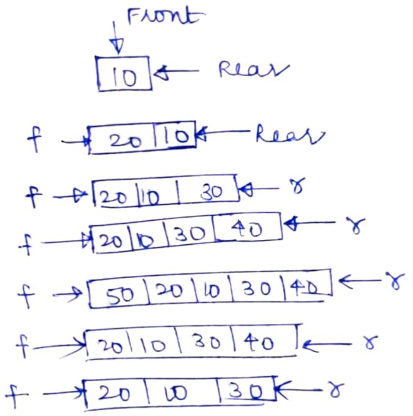
③ insertRear (30);

④ ——— (40);

⑤ insert Front (50);

⑥ deleteFront ();

⑦ — Rear ();



Implementation

- linked list (using doubly liste)
- Array (circular array)

O(1) complexity

Applications

1. Maintaining history of actions
2. A steal process scheduling algorithm
3. Implementing a priority queue with only two priorities
   - priority 1 items can be finished in only order but must be completed before priority 2.
4. Maximum/minimum of all subarrays of size 'k' in an array.

Deque in C++ STL

- allows random access

```
for ( auto x ; dq) {
    cout << x ;
}
```

} prints queue

- operations

(a) dq. push_front (5); ← Insertion at front

(b) dq. push_back (50); ← insertion at rear end

(c) dq. front()

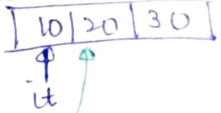
(d) dq. back ()

(e) dq. begin () ← iterators pointing to first element

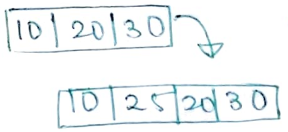
(f) dq. end () ← points to iterator beyond the last element

→ auto it = dq. begin();

it++;



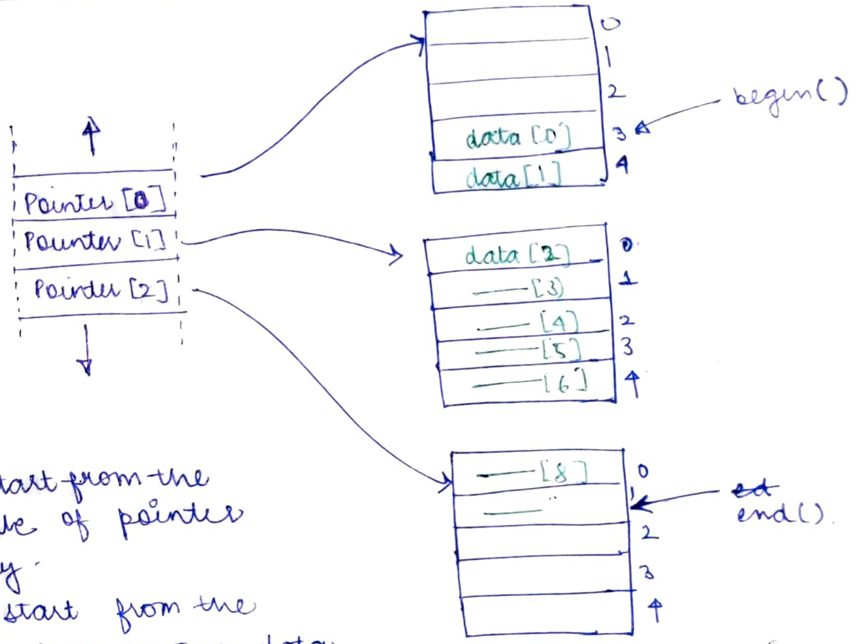
→ dq. insert (it, 25) ← inserts 25 before the element pointed by it



→ dq. pop_front() ← Removes element from front

→ dq. pop_back() ← Removes element from last

How does dequeu work



- we start from the middle of pointer array.
- we start from the ~~ms~~ middle of the data chunk and on insertFirst we insert data from the middle in the upward direction

For instance

data[4] is inserted first

data [3] inserted on insertFirst (x).

push_back (O(1))

push_front O(1)

~~push~~

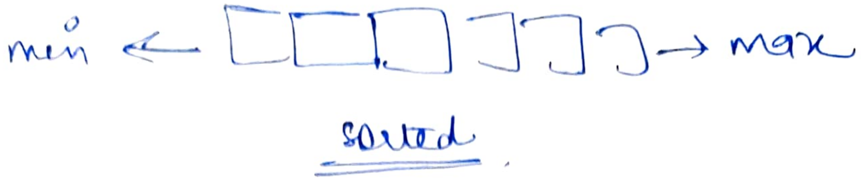
pop_front O(1)

pop_back O(1)

Data structure with min/max operations:

~~S~~ If ~~ge~~ insertMin(x) ← Insert at front

insertMax(x) ← Insert at end

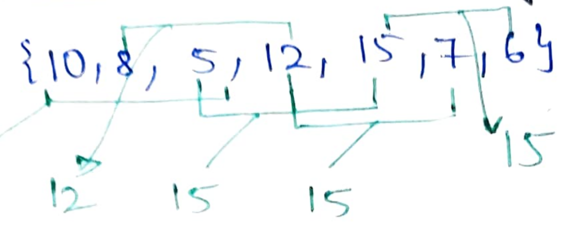


(Deque)

O(1)

Maximum of all subarray of size k

I/P — {10, 8, 5, 12, 15, 7, 6}. k=3



no of output = n-k+1

O/P → {10,12,15,15,15}.

Naive approach O(n^2)

find the max of all the windows by using two loops.

Efficient Approach O(n)

{10, 8, 5, 12, 15, 7, 6}.

- we will make a deque of size k, whenever we see an element ~~of~~ ~~is~~ greater than the front of deque~~ss~~, we remove the front element and ~~into~~ insert the element

ugghh...

Let me show you by an example okay :)

10	8	5

when we see 12 (Remove 10 & all the elements on right)

12		

The idea is whenever we see a larger element, smaller element is of no use to us.

i = 0 dq – | 10 | | |

i = 1 dq – | 10 | 8 | |

i = 2 dq – | 10 | 8 | 5 |

i = 3 dq – | 12 | | |

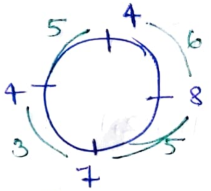
i = 4 dq – | 15 | | |

i = 5 dq | 15 | 7 | |

i = 6 dq – | 15 | 7 | 6 |

Circular Tour ☆☆

I/P → {4, 8, 7, 4} ← petrol
{6, 5, 3, 5} ← dist



we maintain a deque and add the petrol stops till the curr petrol is non-negative.
As soon as the curr petrol becomes negative we remove one petrol spot from the front of queue.

petrol - {50, 10, 60, 100}
{30, 20, 100, 10}

deque {} curr_petrol = 0

| 0 | curr_p = (50 – 30) + 0
= 20

| 0 | 1 | curr_p = 20 + (10 – 20) = 10

| 0 | 1 | 2 | curr_p = 10 + (60 – 100) = –30.

| 1 | 2 | curr_p = –30 – (50 – 30).
= –50

| 2 | curr_p = –50 – (10 – 20)
= –40

{ } curr_p = –40 – (~~10~~ 60 – 100)
= 0

| 3 | curr_p = 0 + 100 – 0
= 100

| 3 | 0 | curr_p = 100 + (50 – 30)
= 120

| 3 | 0 | 1 | curr_p = 120 + (10 – 20)
= 110

| 3 | 0 | 1 | 2 | ✓ curr_p = 110 + (60 – 100)
= 110 – 40
= 70

Ans – 3 q.front().

If we are traversing for $p_0 \cdots p_i$ and at p_i the curr petrol becomes negative, the claim is that none of the points from p_0 to p_i can be a valid solution.

```
int firstPetrolPump (int petrol[], int dist[], int n){
    int start = 0, curr_p = 0;
    int prev_p = 0;
    for (int i=0; i < n; i++) {
        curr_p += (petrol[i] - dist[i]);
        if (curr_p < 0){
            start = i+1;
            prev_p += curr_p;
            curr_p=0;
        }.
    }
    return ((curr_p+ prev_p) >= 0) ? (start+1):-1;
}
```