Queues

Intro To Queues
Queues Using Arrays
Queues Using Stack
Queues Using LL
Nth perfect Number
Dequeues
Sliding Window Maximum of size k

Intro To Queues

Queues are a type of container adaptors that operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front. Queues use an encapsulated object of **deque or list** (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

Method	Definition
queue::empty()	Returns whether the queue is empty. It return true if the queue is empty otherwise returns false.
queue::size()	Returns the size of the queue.
queue::swap()	Exchange the contents of two queues but the queues must be of the same data type, although sizes may differ.
queue::emplace()	Insert a new element into the queue container, the new element is added to the end of the queue.
queue::front()	Returns a reference to the first element of the queue.
queue::back()	Returns a reference to the last element of the queue.
queue::push(g)	Adds the element 'g' at the end of the queue.
queue::pop()	Deletes the first element of the queue.

To use a queue, you have to include the `<queue>` header file

Queues Using Arrays

rear (insertion) — front (deletion)

FIFO — First In First Out.

eg:- Movie Counter
Printer
job scheduler.

Operations On Queue :-

i) Enqueue (x) ; Inserts the data at rear end.
ii) dequeue ( ) ; Deleting element from front end.
iii) front ( ) ; access data at front
iv) isEmpty ( ) ; is your queue empty / not.
v) size ( ) ; return number of element in queue.

Implementation of Queue Using Array

Capacity = N
front = -1 // last index where element deleted.
rear = -1 // last index where element inserted.

f r

```
Void enqueue (int x) {
    if (r == N-1) //queue full.
        return;
    rear ++;
    ar[rear] = x;
}
```

```
Void dequeue ( ) {
    if (isEmpty ( )) { //queue empty
        return;
    }
    front ++;
}
```

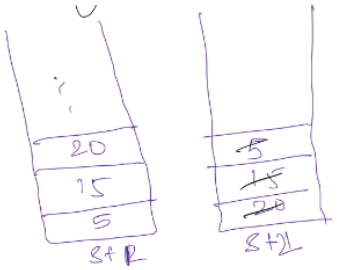
```
int front ( ) {
    if (isEmpty) {return}
    return ar[front + 1];
}
```

```
bool isEmpty ( ) {
    return (f == r);
}
```

To optimise space use
rear = (rear + 1) % N;

Queues Using Stack

O(1) { enqueue(x), dequeue(), front(), size(), isEmpty() }



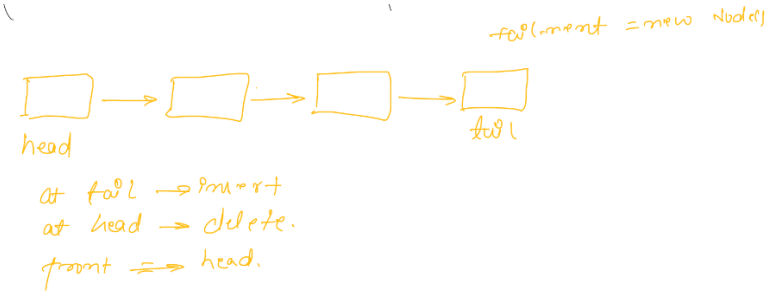

```
Void enqueue (int a)
{ st.push(a);
} // push new element in the stack.
```

```
int front() {
    if(!st2.isEmpty()) //return st2.top();
        return;
    while(!st1.isEmpty()){
        st2.push(st1.top())
        st1.pop();
    }
    return st2.top();
}
```

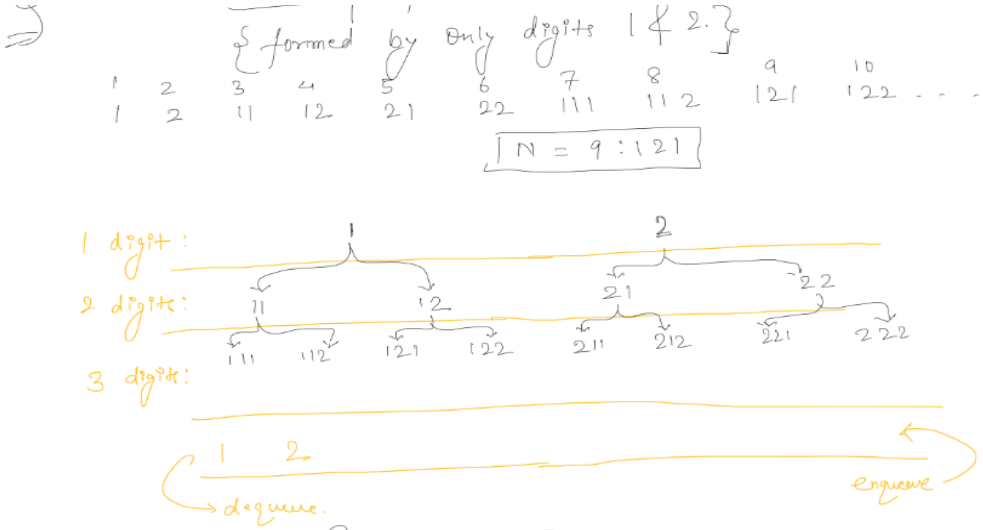
```
Void dequeue() {
    if(!st2.isEmpty()){
        st2.pop();
        return;
    }
    while(!st1.isEmpty()) {
        st2.push(st1.top())
        st1.pop();
    }
    st2.pop();
}
```

T.C: O(1)

Queues Using LL

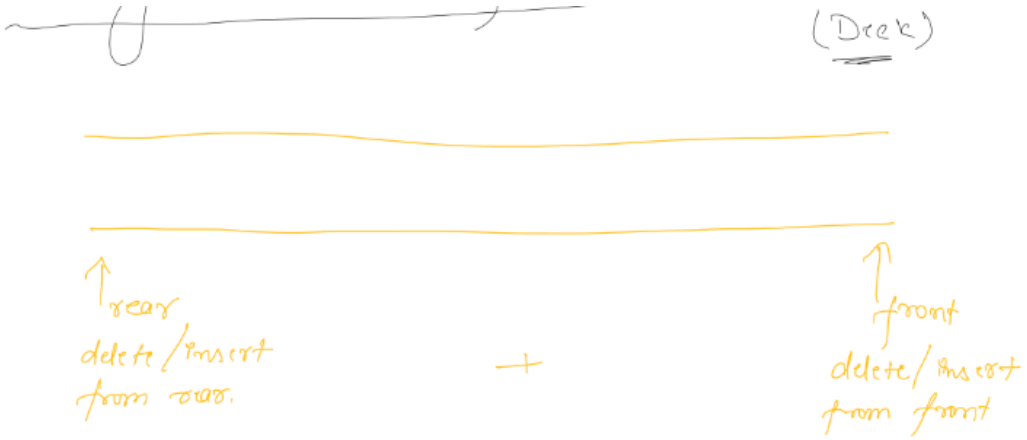


Nth perfect Number

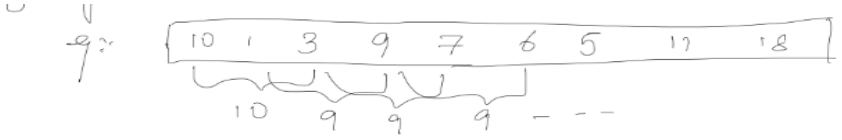



```
Queue<Int> Q:
Q.enqueue(1); Q.enqueue(2)
Int p = 3
while (p <= N) {
    int x = q.front()
    q.dequeue();
    int a = x*10 + 1
    int b = x*10 + 2
    if (p == N) {
        return a;
    }
    q.enqueue(a)
    p++;
    if (p == N) {
        return b;
    }
    q.enqueue(b);
    p++;
}
```

Dequeues


(Deck)
rear
delete / insert
from rear.
+
front
delete / insert
from front

Sliding Window Maximum of size k

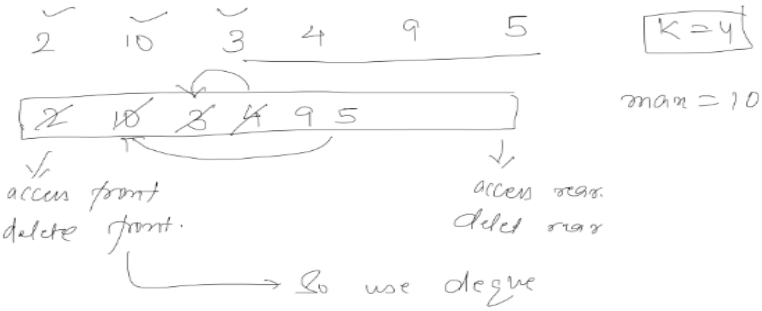


N = 9
K = 3

B.F : $O(N^2)$

Sliding Window cannot be applied
(max in 3 elements we cannot get by shifting window

Optimized Approach




```
s = 0
e = k - 1
// for 1st window.
for (int i = 0; i < k; i++) {
    while (!dq.empty() && arr[dq.back()] <= arr[e])
    {
        dq.pop_back();
    }
    dq.push_back(arr[i])
}
while (e < N) {
    if (dq.front() == s-1) { dq.pop_front(); }
    while (!dq.empty() && arr[dq.back()] <= arr[e])
    {
        dq.pop_back();
    }
    dq.push_back(arr[e])
    Print(dq.front());
    s++;
    e++;
}
```