Queue 1

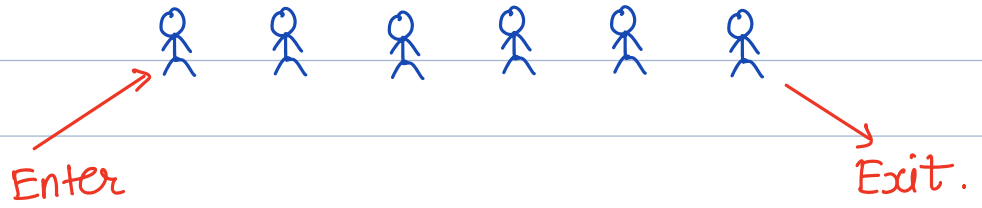
Content

- Introduction
- Implementation
- Implement Queue via stack
- Perfect Numbers
- Sliding window Maximum.

Real life examples

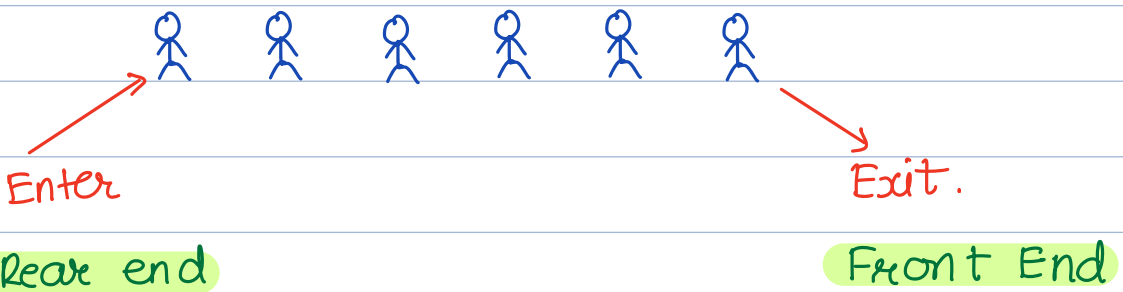
1> Pipe

2> Standing Queue

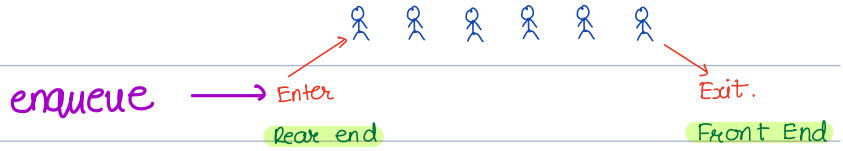


3> Music playlist.

Queue ⟶ First In, First Out



Operations on Queue



- enqueue(x) ⟶ Add x at the rear end of the queue
- dequeue() ⟶ Remove from the front end of the queue
- isEmpty() ⟶ Check if the queue is empty.

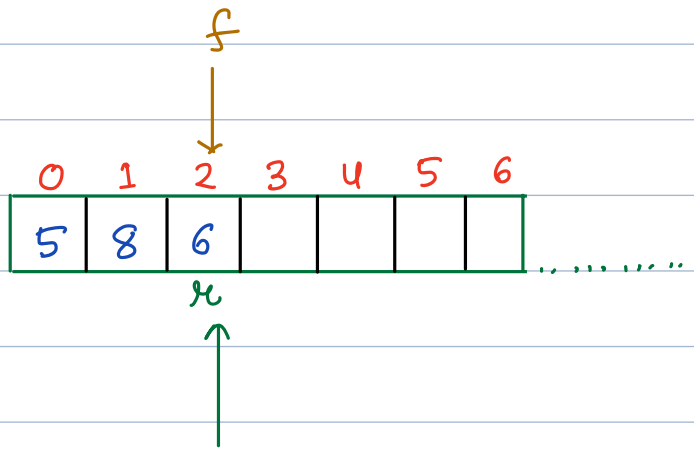
Optional functions

- front() ⟶ Get val at the front of queue
- rear() ⟶ Get val at the rear of queue

TC ∀ above operations = O(1)

Implement Queue using Dynamic Arrays

enqueue (5)
enqueue (8)
enqueue (6)
dequeue () ⟶ 5
dequeue () ⟶ 8
isEmpty () ⟶ False



queue ⟶ f – front
r – rear.

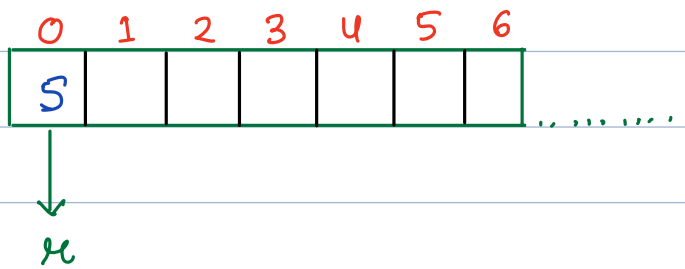
A ⟶ Dynamic array

r = −1

f = 0

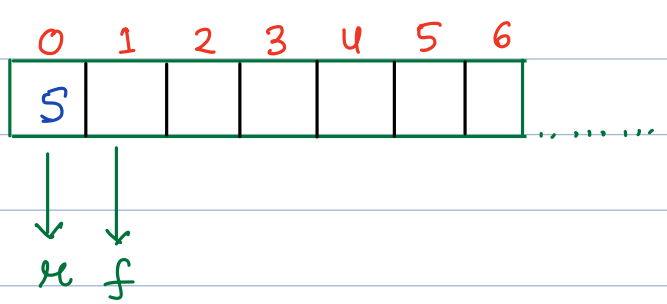
```
enqueue (x) {
    r++
    A .add (x)
}
```

enqueue (5)




```
dequeue (x) {
    if (isEmpty()) ret -1
    val = A[f]
    f++
    return val
}
```

dequeue ( ) ⟶ 5

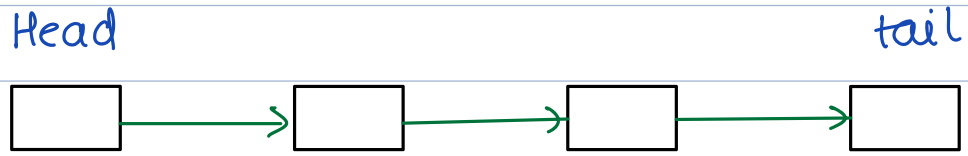


TC : O(1)

```
isEmpty () { return f > r }
```

Implement Queue using Linked List

enqueue (5)
enqueue (8)
enqueue (6)
dequeue ( ) ⟶ 5
dequeue ( ) ⟶ 8
isEmpty ( ) ⟶ False



insert O(1)
delete O(1)

insert O(1)
delete O(n)

⟶ To implement queue using linked list
we insert at the tail node
we delete from the head node

Implement Queue using Two stacks *

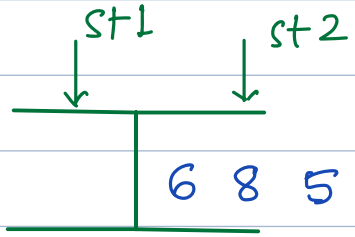
NOTE: Assume all operations done are valid.

enqueue (5)
enqueue (8)
enqueue (6)
dequeue ()
dequeue ()
isEmpty ()

Allowed Operations
→ push
→ pop
→ isEmpty ()

Queue is like a pipe

Stack is like a glass



→ For every enqueue operation
→ Just push the value on st1

→ For every dequeue operation
→ If st2 is empty
→ add all values from st1
else
→ pop from st2

→ isEmpty st1.isEmpty() && st2.isEmpty()

stack 1 = []
stack 2 = []

pop
push
isEmpty

```
void enqueue(x) {
    stack1.push(x)
}
```

→ O(1)

```
int dequeue() {
    // if stack2 is empty add all values
    // from stack 1 to stack 2
    if (stack2.isEmpty()) {
        while(!stack1.isEmpty()) {
            val = stack1.pop()
            stack2.push(val)
        }
    }

    return stack2.pop()
}
```

→ O(N)

```
bool isEmpty() {
    return stack1.isEmpty() &&
           stack2.isEmpty()
}
```

→ O(1)

Avg TC per operation → O(1)

Break 8:34

N^{th} Perfect Number

Find N^{th} perfect number ie number formed by only digits 1 & 2

N	1	2	3	4	5	6	7	8	9	10 .....
	1	2	11	12	21	22	111	112	121	122 .....

Brutforce

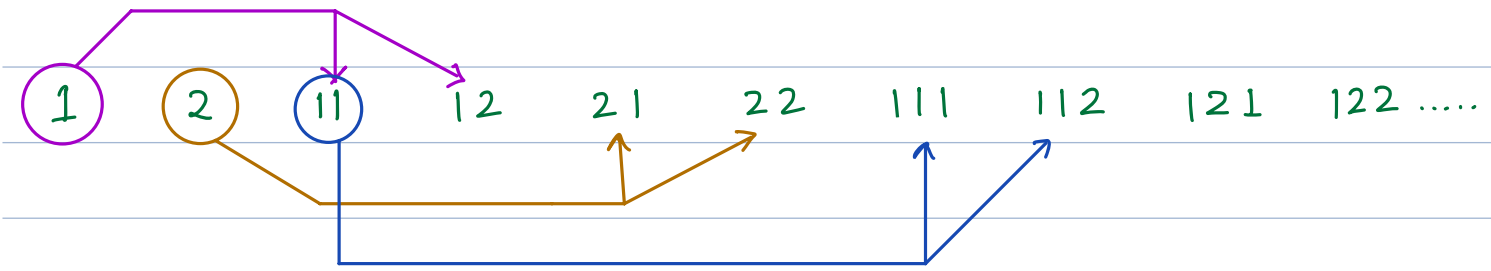
Iterate over all natural no.

if number only contains 1 or 2

count ++

if count == N

return number.



(1) (2) 11 12 21 22 111 112 121 122

↑

Idea → use queue to generate no. in sorted order

if you are at a val → v

then enqueue v + "1", v + "2"

```
int perfectNumber(N) {

    queue // init
    queue.enqueue("1")
    queue.enqueue("2")

    perfect = []  // dynamic array

    while (perfect.size() < N) {
        val = queue.dequeue() //
        perfect.add(val)

        queue.enqueue(val + "1")
        queue.enqueue(val + "2")
    }

    return perfect[N-1]
}
```

TC: O(N)

SC: O(N)

---

stacks + queue ⟶ deque

Doubly ended queue

DLL

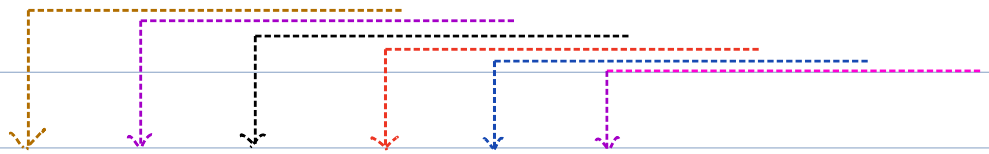
Read about deque equivalent in your language.

Sliding window Maximum *

Given an integer array A,
∀ window of size K find the max element.

K = 4

	0	1	2	3	4	5	6	7	8
A =	1	8	5	6	7	4	2	0	3



Output	8	8	7	7	7	4

Bruteforce

Check for all the window of size K
and print out max for each window.

TC : O(N*K)

Algo steps

→ Initialise doubly ended queue { deque }
→ Store the indexes in deque such that
the values of indexes is monotonically decreas...
{ similar to next smaller on left }
→ Check if the first index in deque is out of
queue. If so, then remove
→ ans is always front of deque.

k = 4

	0	1	2	3	4	5	6	7	8
A =	1	8	5	6	7	4	2	0	3

			ans
0	1	$[0^1]$	no
1	8	$[1^8]$ ⟶ 8 pops 1	no
2	5	$[1^8\ 2^5]$	no
3	6	$[1^8\ 3^6]$	8
4	7	$[1^8\ 4^7]$	8
5	4	$[4^7\ 5^4]$ ⟶ remove 8	7
6	2	$[4^7\ 5^4\ 6^2]$	7
7	0	$[4^7\ 5^4\ 6^2\ 7^0]$	7
8	3	$[5^4, 8^3]$	4

TC : O(N) O(K)