Lecture : Queues

Agenda

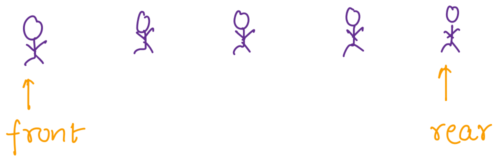
- Intro
- Implementation
- Queue using stacks
- Perfect numbers
- Sliding window max

Queue

fifo : first in first out.

Ex: Ticketing queue.

Operation



1.) add(x) : x adds at end.

2) poll() : remove el from front end.

3.) front(): return front el

4.) rear(): return rear el.

Implementation

1. Arrays

data: 8 14 9 20 poll() ↑ (8) 30 front() (14) ↑ (14) rear() (30) 60 ↑ (9)

13 39 41 ↑ (20) ↑ (30) ↑ (60) ↑ (13) 49 ↑ (39) ↑ (41) ↑ (49) 59

arr[7] =

39	41	49	59			
~~8~~	~~14~~	~~9~~	~~20~~	30	60	13
0	1	2	3	4	5	6

front: ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~0~~ ~~1~~ ~~2~~ 3

rear: ~~-1~~ ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~0~~ ~~1~~ ~~2~~ 3

size: ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

Queue for visualisation

~~8~~ ~~14~~ ~~9~~ ~~20~~ ~~30~~ ~~60~~ ~~13~~ ~~39~~ ~~41~~ ~~49~~ 59

2.) Linkedlist

8 14 9 20 ↑ 30 front() ↑ rear() 60 ↑ 13

(14) (30) . . . .

head = ~~null~~ ~~8~~ ~~14~~ 9

tail = ~~null~~ ~~8~~ ~~14~~ ~~9~~ ~~20~~ 30

size = ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~3~~ 4

LL : 8 ⟶ 14 ⟶ 9 ⟶ 20 ⟶ 30 . . . . . .

head head

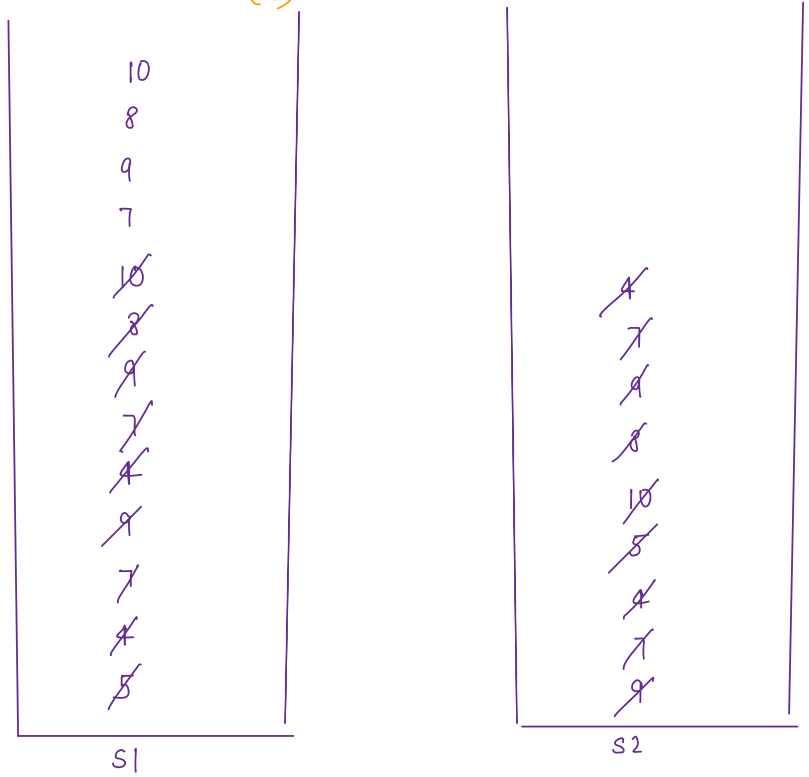
Queue: | ~~8~~ 14 9 20 30 - - - - |

```
void add( int x) {
    xn = new Node (x);
    size +=1;
    if ( h = = null) {
        h = xn;
        t = xn;
    } else {
        t.next = xn;
        t = xn;
    }
}
```

```
int poll( ) {
    if( h==null) {
        return -1;
    }
    size --;
    temp = h;
    h = h.next;

    return temp.data;
}
```

Qu Implement queue using stacks

Sol 5 4 7 9 ↑ (5) 8 10 ↑ (4) ↑ 14 ↑ ↑ 21.



Queue: 5 4 7 9

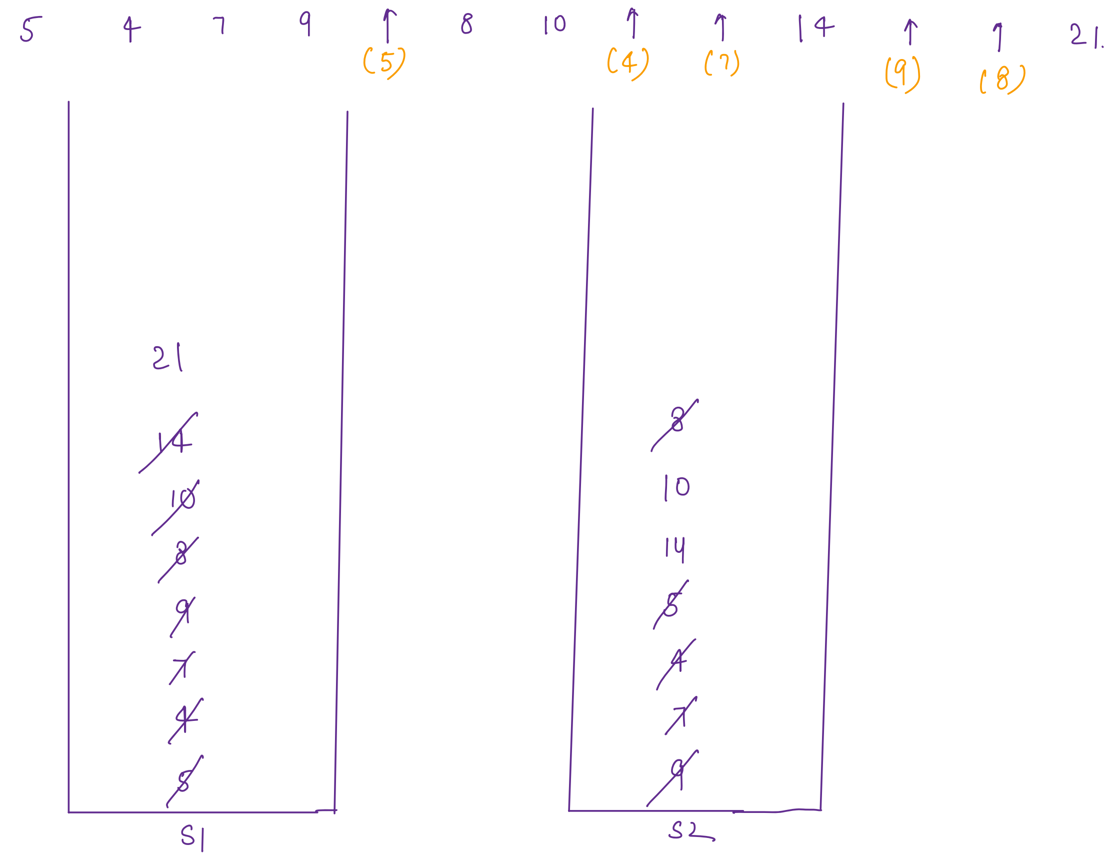
add(x) : push(x) in S1 — O(1)

poll(): Transfer all el from S1 to S2

Removing top el from S2 → S2.pop() — O(n)

Transfer all el from S2 to S1.

Approach 2



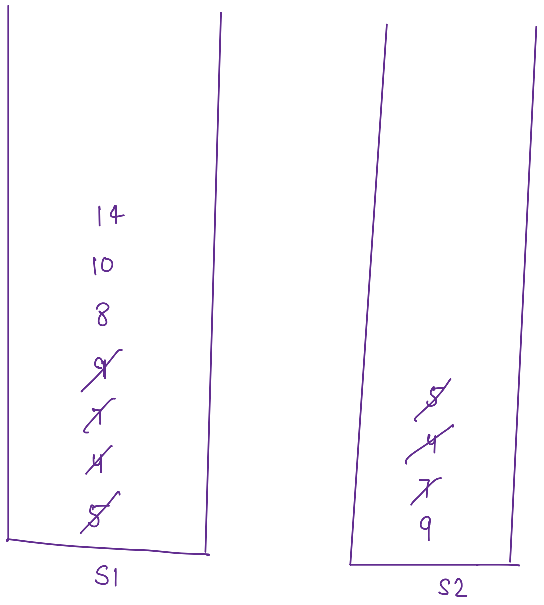
```
add(x) :   push(x) in s1 :   O(1)

poll() :        if ( s2.size() == 0) {
                      transfer s1 to s2'
                }
                s2.pop();
```

Best TC: O(1)

Worst TC: O(n) ??

Proof:

5 4 7 9 ↑ (5) 8 10 ↑ (4) ↑ (7) 14 ↑ (9) ↑ (8) 21.



1st poll(): Transfer 4 el from S1 to S2 — 8 operations

```
while( !s1.isEmpty()) {
    el = s1.pop();
    s2.push(el);
}
```

final ans: s2.pop() ⟶ 9 operations

2nd poll(): s2.pop() ⟶ 1 operation

3rd poll(): s2.pop() ⟶ 1 operation

4th poll(): s2.pop() — 1 opr

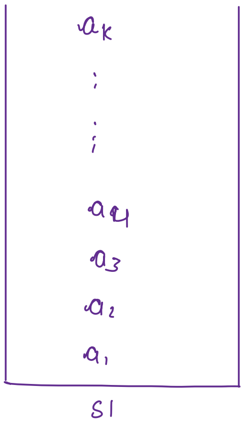
4 polls: 12 opr.

1 poll: $\frac{12}{4}$ = 3 opr. ≃ O(1)

↑
amortized

One expensive poll operation makes future operation cheaper.

Generalisation



1st poll():

Transfer all el from S1 to S2: 2K op^{r}.

S2.pop(): 1 op^{r}

Total: $(2K+1)$ op^{r}

2nd poll(): 1 op^{r}

3rd poll(): 1 op^{r}

4th poll(): 1 op^{r}

:

Kth poll(): 1 op^{r}.

$(K-1)$ op^{r}.

K polls(): $2K + \cancel{1} + K - \cancel{1}$

K polls(): 3K

1 poll(): $\frac{3\cancel{K}}{\cancel{K}} = 3$ op^{r}. $\simeq$ O(1)

Break: 8: 25 AM

Qu Generate kth no using digits only 1 and 2.

1, 2, 11, 12, 21, 22, 111, 112, 121, 122 - - - - -

K = 9, ans = 121

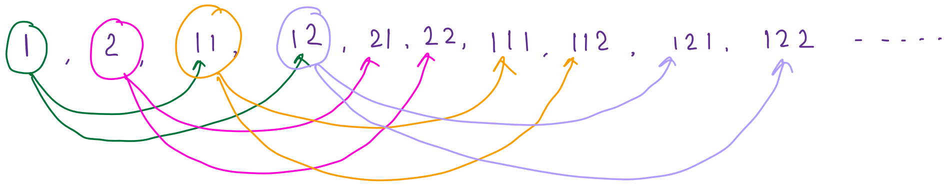
Brute force: ∀ natural numbers —

check if a num contains only 1 and 2

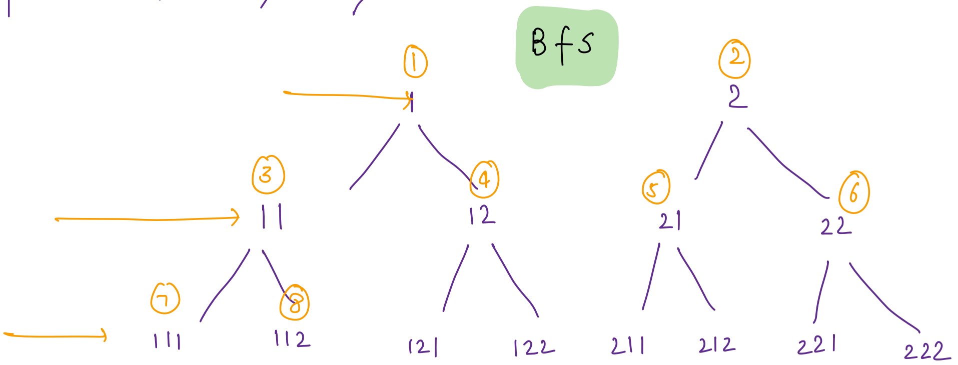
if yes — cnt++

if cnt == k, return num;

Approach2:



| ~~1~~ ~~2~~ ~~11~~ ~~12~~ 21 22 111 112 121 122 ...... |



```
String getkthNumber( int k) {
        Queue<string> q;
        q.add("1");
        q.add("2");
        int  cnt=0;
       while( !q.isempty()) {
              String  el = q.poll();
              cnt++;

              if( cnt == k) {
                   return el;
              }
              q.add( el + "1");
              q.add( el + "2");
       }
     return -1;
}
```

can you optimise the space complexity ??

TC: O(K)

SC: ??

Dry run:

K = 8

cnt = ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~7~~ 8

Ans.

| ~~1~~ ~~2~~ ~~11~~ ~~12~~ ~~21~~ ~~22~~ ~~111~~ $\boxed{112}$ 121 122

211 212 221 222 1111 1112

extra numbers

Dequeue [Doubly ended queue]

DLL

front
Queue
rear
stack.

Operations

addLast()

addfirst()

removeLast() → Implemented using DLL

removefirst()

front()

rear()

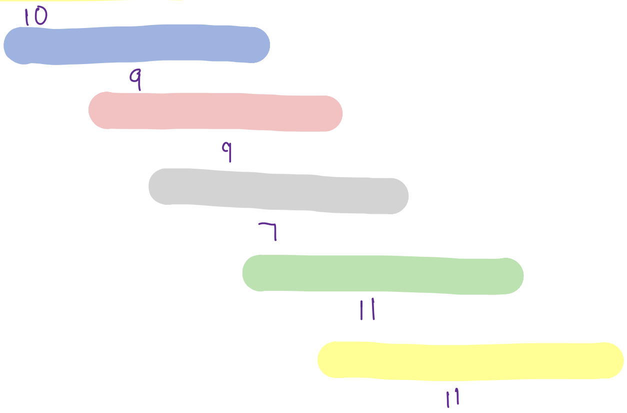
Qu

Sliding window maximum

Given arr[n] and K.

Find and print max in every window of size = K

arr[9] = [10 1 9 3 7 6 5 11 8] K = 4



Brute force: ∀ subarrays of window of len == K — O(n)

find max — O(n)

TC: $O(n^2)$

SC: $O(1)$

Approach 2 Dequeue

Hint: Only possible candidate ans will be inserted

arr[] = [3 15 6 12 4 2 10 9 13] , K = 4.

0	1	2	3	4	5	6	7	8
3	15	6	12	4	2	10	9	13

15 15 12 12 10

Deque: ~~3~~ ~~15~~ ~~6~~ ~~12~~ ~~4~~ ~~2~~ 10 9 ......

add(x): rear End()

ans(): front()

Challenge: How to decide which el to remove from deque which is not part of a window?

arr[] = [3 15 6 12 4 2 10 9 13] , K = 4.

0	1	2	3	4	5	6	7	8
3	15	6	12	4	2	10	9	13

print: { arr[1] (15) , arr[1] (15) , arr[3] (12) , arr[3] (12) , arr[6] 10 , arr[8] 13

dequeue: ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~7~~ 8

```
if( dq.front == i-k) {
        dq.removefirst();
}
```

```
void slidingWindowMax(arr[], k) {

    Dequeue<Integer> dq = new LinkedList<>();

    // Handle first window alone

    for(i=0; i<k; i++) {
        while( ! dq.isEmpty()    &&
                    arr[i] >= arr[dq.rear()] {
                dq.removeLast();
        }
        dq.addLast(i);
    }
    print( arr [dq.front()] );

    for(i=k; i<n; i++) {
        while( ! dq.isEmpty()    &&
                    arr[i] >= arr[dq.rear()] {
                dq.removeLast();
        }
        dq.addLast(i);

        // Remove el out of window

        if( dq.front == i-k) {
                dq.removefirst();
        }

        print( arr [dq.front()] );
    }
}
```

TC: O(n)

SC: O(k)

Thankyou ☺

Doubts

1 str

~~2~~ str

21
22

~~11~~ → str + "1"

12 → str" + 2

12 1
2

111 — str + "1"

112 str + "1