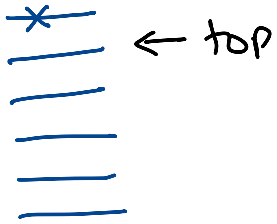
Data Structure & Algorithms

Sunbeam Infotech

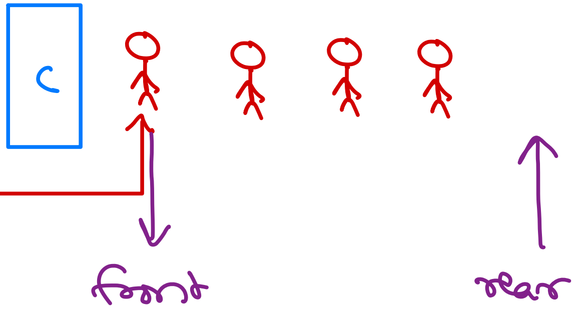
Stack and Queue

- Stack & Queue are utility data structures. → temp storage during process.
- Can be implemented using array or linked lists.
- Usually time complexity of stack & queue operations is O(1).
- Stack is Last-In-First-Out structure.
- Stack operations → ADT
  - ✓ push()
  - ✓ pop()
  - ✓ peek()
  - ✓ isEmpty()
  - isFull()*



ops done from same end

- Simple queue is First-In-First-Out structure.
- Queue operations – ADT
  - ✓ push() / enqueue()
  - ✓ pop() / deque()
  - ✓ peek()
  - ✓ isEmpty()
  - isFull()*



- Queue types
  - Linear queue
  - Circular queue
  - Deque
  - Priority queue

Linear Queue

push → rear
pop ← front

init:

```
arr = new int[];
f = -1;
r = -1;
```

push:

```
r++;
arr[r] = val;
```

pop:

```
f++;
```

peek:

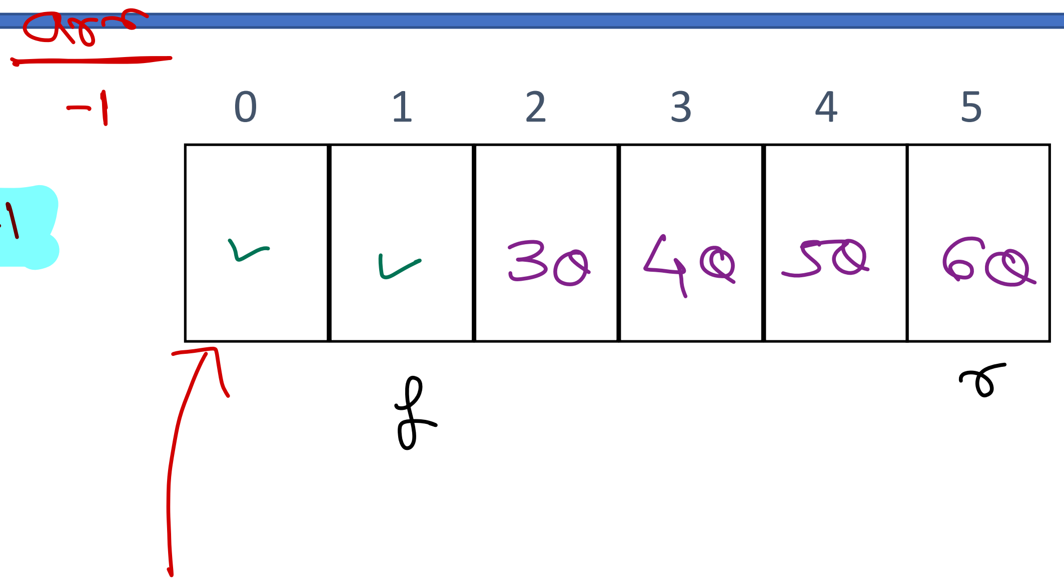
```
return arr[f+1];
```

isFull:

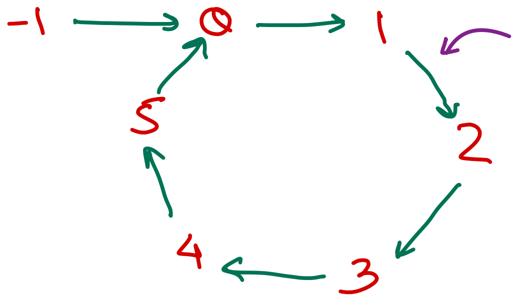
```
r == MAX-1
```

isEmpty:

```
f == r
```



improper memory utilization



```
if (r == MAX-1)
    r = 0;
else
    r = r+1;
```

✓

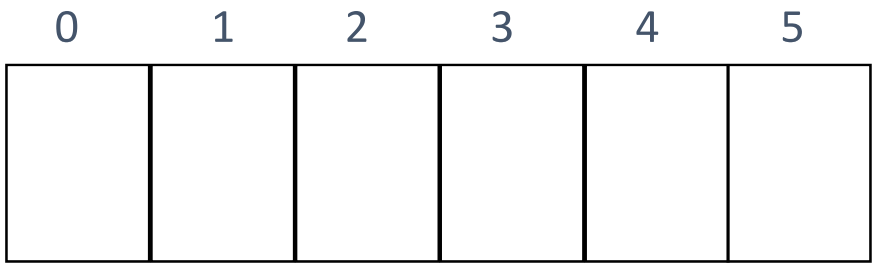
r = (r+1) % MAX;

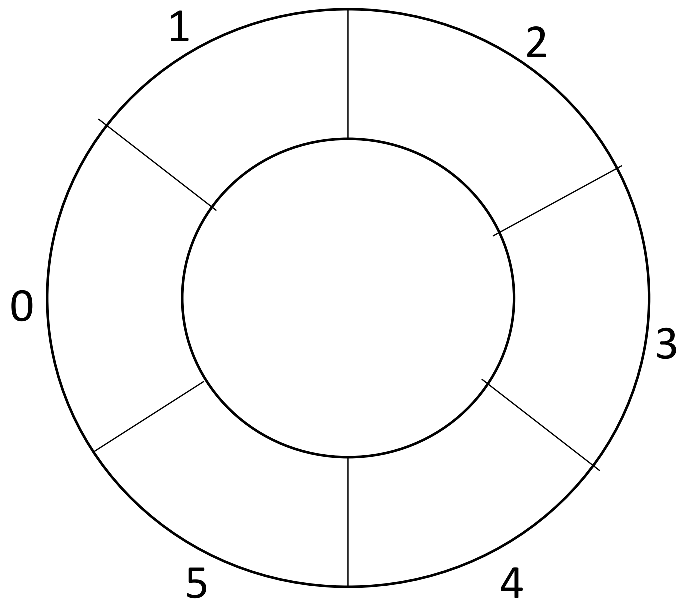
r	
-1	0
0	1
1	2
2	3
3	4
4	5
5	0

✓

Circular Queue

- In linear queue (using array) when *rear* reaches last index, further elements cannot be added, even If space is available due to deletion of elements from *front*. Thus space utilization is poor.
- Circular queue allows adding elements at the start of array if *rear* reaches last index and space is free at the start of the array.
- Thus *rear* and *front* can be incremented in circular fashion i.e. 0, 1, 2, 3, …, n-1. So they are said to be circular queue.
- However queue full and empty conditions become tricky.





Circular Queue – full & push

-1	0	1	2	3	4	5
	10	20	30	40	50	60
f						

OR

-1	0	1	2	3	4	5
	70	80	90	40	50	60
			f r			

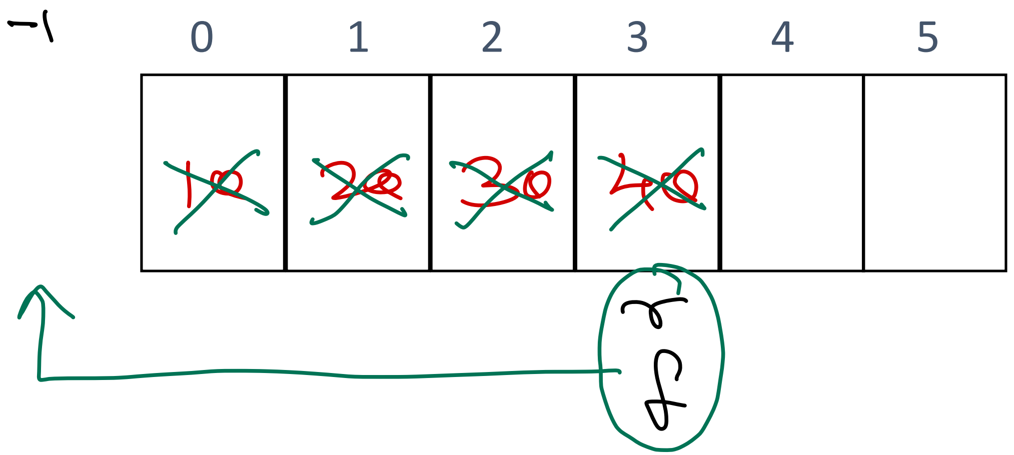
$(f == -1 \;\&\&\; r == \overset{r}{MAX}-1) \;||\; (f == r \;\&\&\; f != -1)$

push:

```
r = (r+1) % MAX;
arr[r] = val;
```

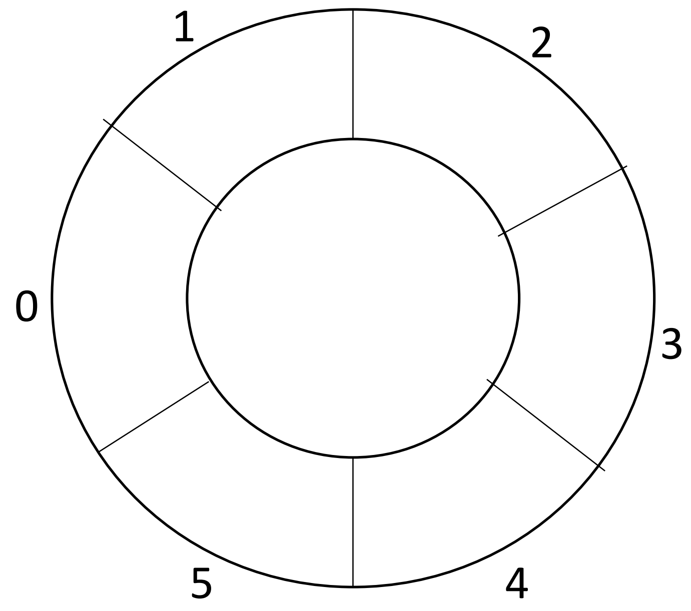
Circular Queue - empty & pop

(r == f && f == -1)



```
pop :
    f = (f + 1) % MAX ;

    if( f == r)
    {   f = -1
           r = -1
    }
```

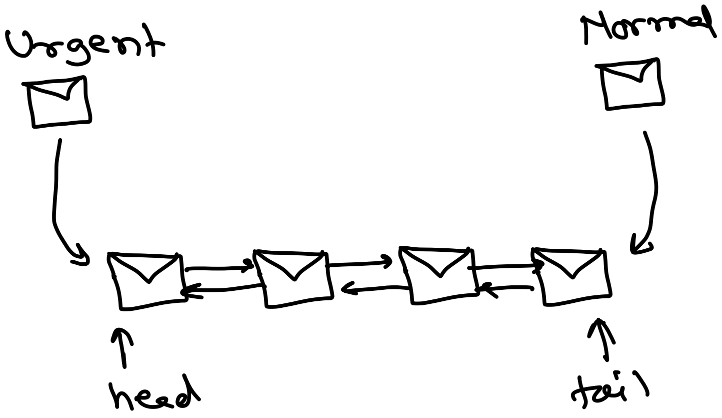


DeQueue

- In double ended queue, values can be added or deleted from front end or rear end.

applications

message queue in RTOS.

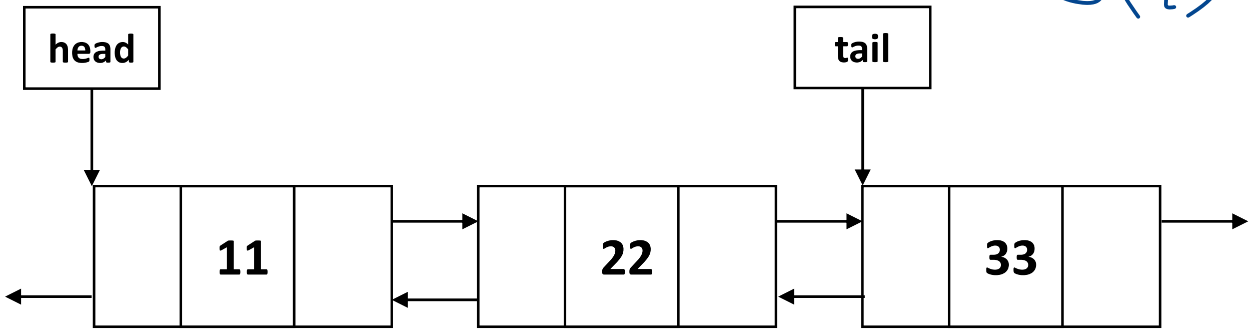


O(1)
push_front

pop_front
O(1)

O(1)
push_back

pop_back
O(1)



Priority queue → Not a FIFO queue

- In priority queue, element with highest priority is removed first.

↳ internally elements are stored in sorted manner.

Can be implemented

① using array → insertion logic.

② using linked list → insertion logic.

push = $O(n)$

pop = $O(1)$

③ using heap (array representation of bin tree).

✓ efficient

push = $O(\log n)$

pop = $O(\log n)$

5 9 6 2

↓

head

↓

$2 \to 5 \to 6 \to 9$

×

Stack / Queue in Java collections

- class java.util.Stack<E>
  - ✓ E push(E);
  - ✓ E pop();
  - ✓ E peek();
  - ✓ boolean isEmpty();

- interface java.util.Queue<E>
  - ✓ boolean offer(E e); - push
  - ✓ E poll(); = pop
  - ✓ E peek();
  - ✓ boolean isEmpty();

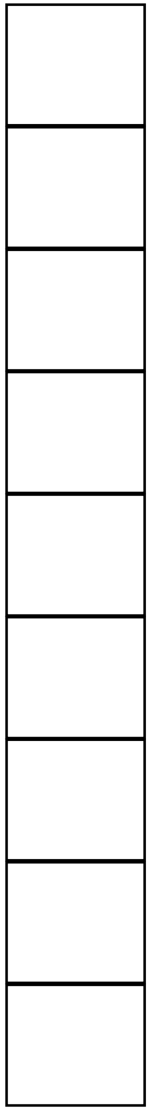
ArrayDeque<> ✓

LinkedList<> ✓

PriorityQueue<>

Infix to Postfix

- 5 + 9 – 4 * (8 – 6 / 2) + 1 $ (7 – 3)



① traverse infix from left to right.

② if operand found, apped to postfix.

③ if operator found, push to stack ⊛.

⊛ if priority of topmost operator on stack >= priority of current operator, pop it from stack & append to postfix.

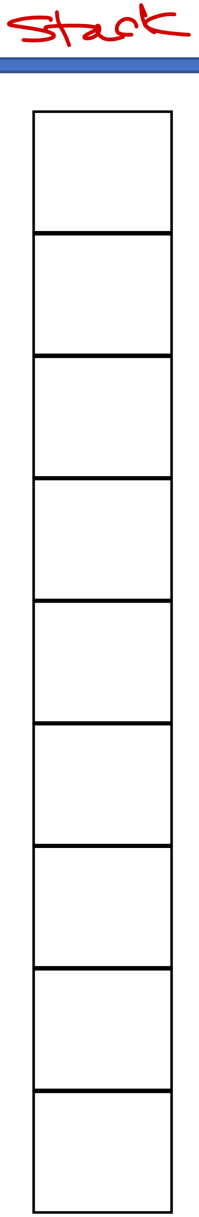
④ if opening (found, push it on stack.

⑤ if closing) found, pop operators from stack & append to postfix until opening is found. also pop & discard opening (from stack.

⑥ if all syms from infix are completed, pop one by one & append to postfix,

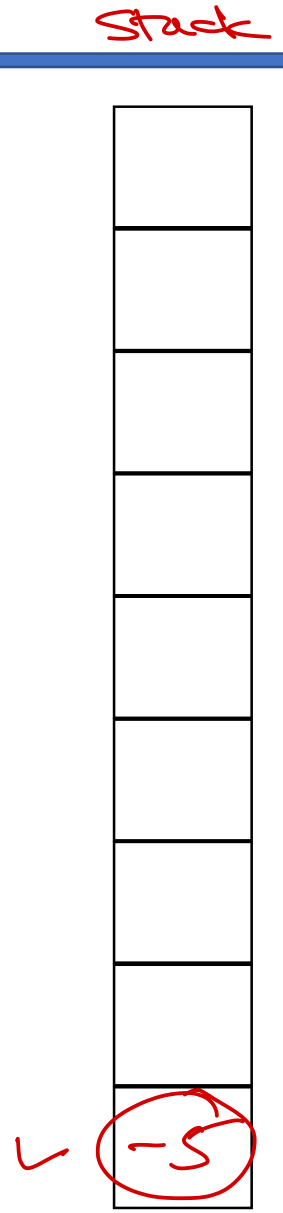
Infix to Prefix

- 5 + 9 – 4 * (8 – 6 / 2) + 1 $ (7 – 3)



Prefix Evaluation

- + - + 5 9 * 4 – 8 / 6 2 $ 1 – 7 3



Parenthesis Balancing

- 5 + ([9 – 4] * (8 – { 6 / 2 }))) ✗

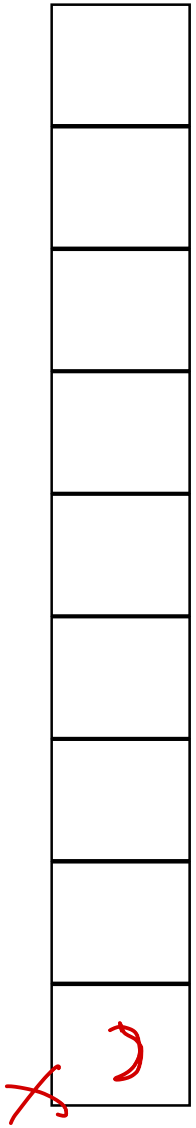
↑

	0	1	2
open	(	[	{
close	)	]	}

0 0

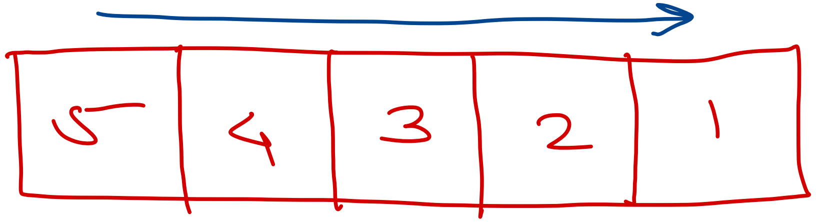
| |

( )



Stack / Queue – Competitive Programming

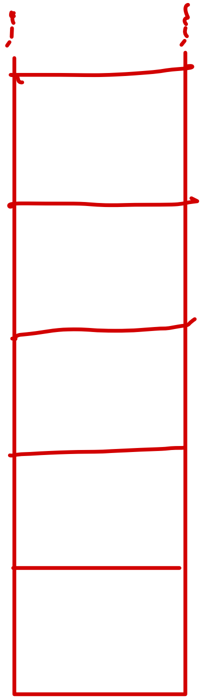
- Reverse array, string or linked list.



```
for(i=0; i<n; i++)        | O(n)
    s.push(arr[i]);       |
```

$O(2n) = O(n)$

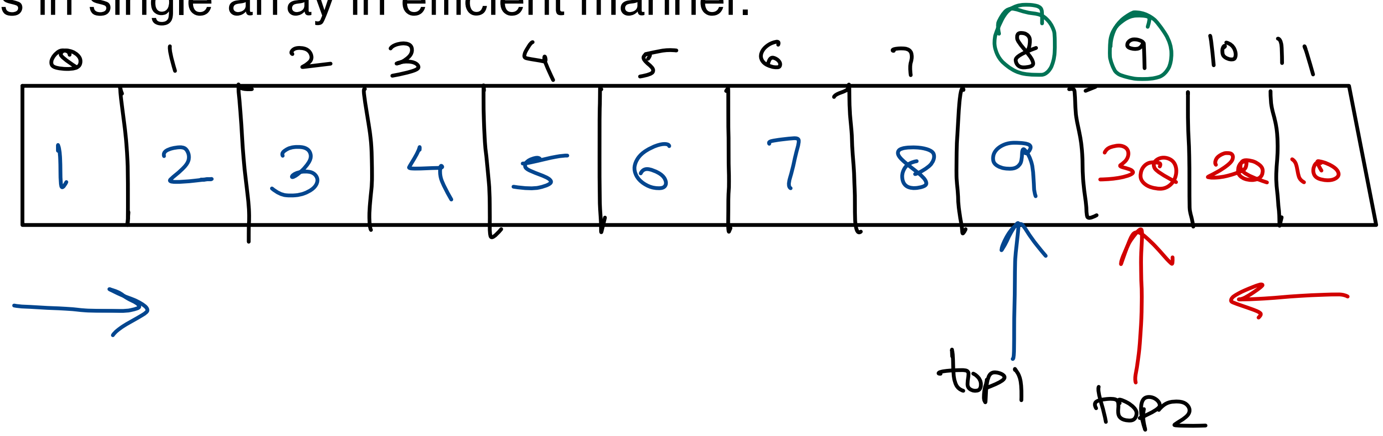
```
for(i=0; i<n; i++)        | O(n)
    arr[i] = s.pop();     |
```



Stack / Queue – Competitive Programming

- Create two stacks in single array in efficient manner.

```
init1:
  top1 = -1
isEmpty1:
  top1 == -1
push1():
  top1++;
  arr(top1) = val;
pop1():
  top1--;
isFull():
  top1 + 1 == top2
```



0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	30	20	10

```
✓ top1 + 1 == top2
✓ top1 == top2 - 1
✓ top2 - top1 == 1
```

```
init2:
  top2 = arr.length
isEmpty2:
  top2 == arr.length;
push2():
  top2--;
  arr(top2) = val;
pop2():
  top2++
isFull:
  top1 + 1 == top2
```

Stack / Queue – Competitive Programming

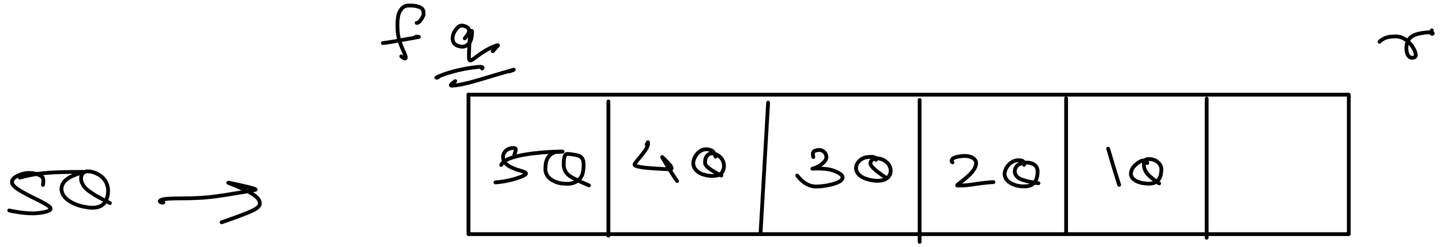
- Create stack using queue.

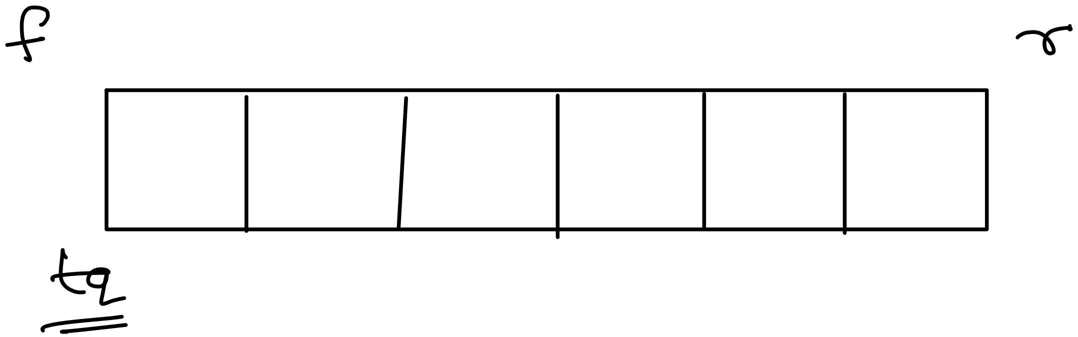
LIFO FIFO

push → O(n)

tq

```
while(!q.isEmpty()){
    tq.push(q.pop());
}
q.push(val);
while(!tq.isEmpty()){
    q.push(tq.pop());
}
```





pop → O(1)

q.pop()

Stack / Queue – Competitive Programming

- Create queue using stack.

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>