Bubble Sort

//1. compare all pairs of consecutive elements
//2. if left element is greater than right element
//2.1 swap both elements
//3. repeat above two step till array is not sorted

No. of element = n
No. of passes = n-1

pass 1 → n-1
pass 2 → n-2
pass 3 → n-3
⋮
pass n-2 → 2
pass n-1 → 1

Total comps = $1+2+3+\cdots\cdots+(n-1)$ (n terms)

$$= \frac{n(n+1)}{2}$$

N = 6

i	i < N	j < N-i	j < N-i
	i < 6	j < 5	
0		j = 0-4	j < 5
1		j = 0-4	j < 4
2		j = 0-4	j < 3
3		j = 0-4	j < 2
4		j = 0-4	j < 1
5		j = 0-4	

i	j < N-i
0	j = 0-4
1	j = 0-3
2	j = 0-2
3	j = 0-1
4	j = 0-0

$$\text{Time} \propto \frac{1}{2}(n^2+n)$$

$T(n) = O(n^2)$ – Avg, Worst

$T(n) = O(n)$ – Best

Insertion Sort

//1. pick one element (start from 2nd index) of the array
//2. compare picked element with all its left neighbours one by one
//3. if left neighbour is greater than picked element
//4. move left neighbour one position ahead
//5. insert picked element at its appropriate position
//6. repeat above steps till array is not sorted

```
public static void insertionSort(int arr[], int N) {
        for(int i = 1 ; i < N ; i++ ) {
            int temp = arr[i];
            int j = i - 1;
            while(j >= 0 && arr[j] > temp) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = temp;
        }
}
```

22 44 55 66

44 55

arr = ~~55~~ ~~44~~ 22 ~~66~~ 11 33

(6)

i	i<N	temp	j	j>=0 && arr[j]>temp
1	T	44	0,-1	T, F
2	T	22	1,0,-1	T, T, F
3	T	66	2	F

No. of elements = N
No. of passes = N-1
Pass1 → 1
Pass2 → 2
⋮
PassN-1 → N-1

Total comps = 1+2+3+...(n-1) (n-1 = n)

$= \frac{n(n+1)}{2}$

Time $\propto \frac{1}{2}(n^2+n)$

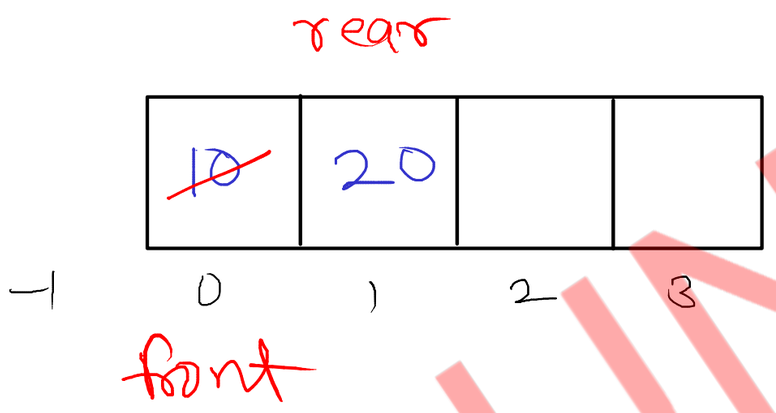
Avg, Worst → $T(n) = O(n^2)$

Best $T(n) = O(n)$

Linear Queue

- it is a linear data structure which stores similar type of data
- it has two end for data insertion and deletion
    1. rear - data is inserted
    2. front - data is deleted
- queue works on the principle of "First In First Out" / "FIFO"

size=4



- All operations of queue data structure are performed in O(1) time

Operations

1. Add/Insert/Enqueue/Push:

a. reposition rear (inc)

b. add value/data at rear index

2. Delete/Remove/Dequeue/Pop:

a. reposition front (inc)

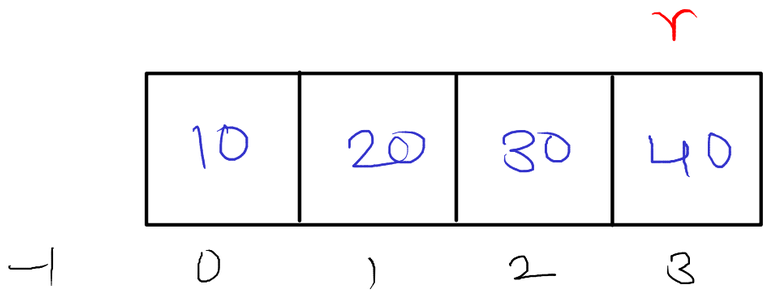
3. Peek:

a. read data/value from front end

Linear Queue

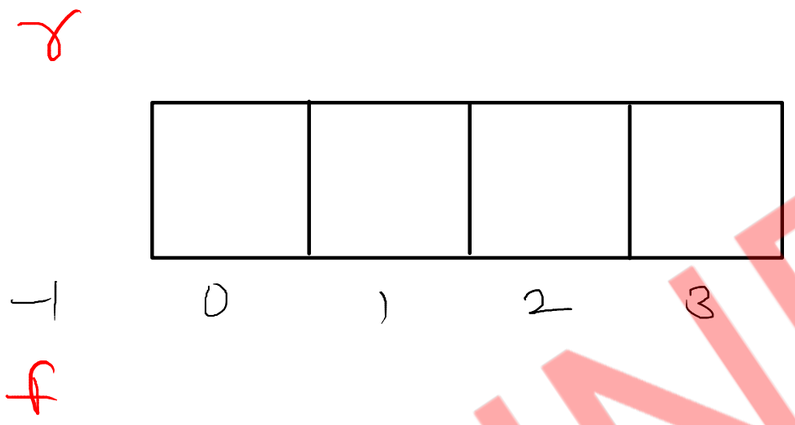
conditions

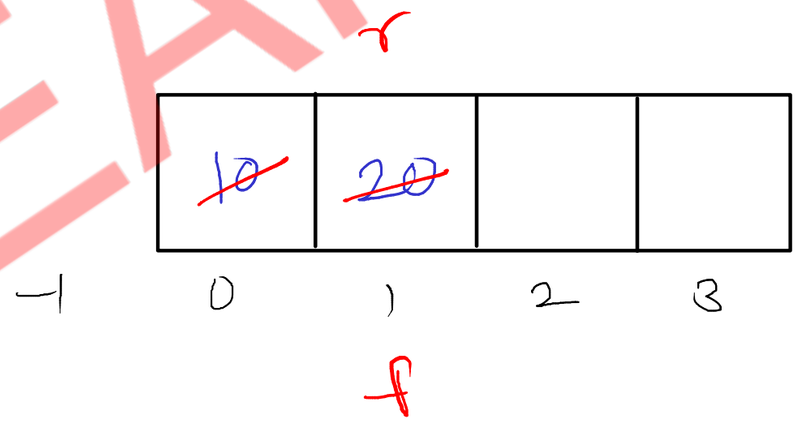
Full:



rear == size - 1

Empty:

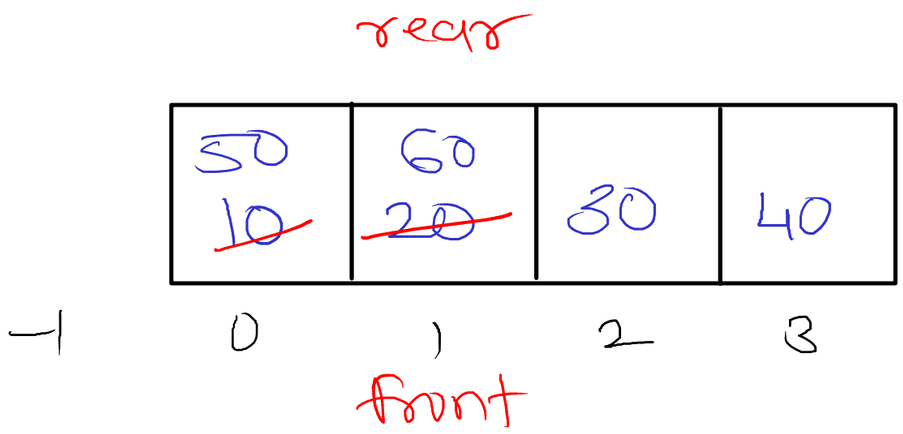




front == rear

- in linear queue, once rear is reached to last index and initial few locations are vaccant then also queue is full. Means we are not utilising those empty locationsa again
- this will lead to poor memory utilization
- solution for this is "circular queue"

Circular Queue



1. Add/Insert/Enqueue/Push:

a. reposition rear (inc)

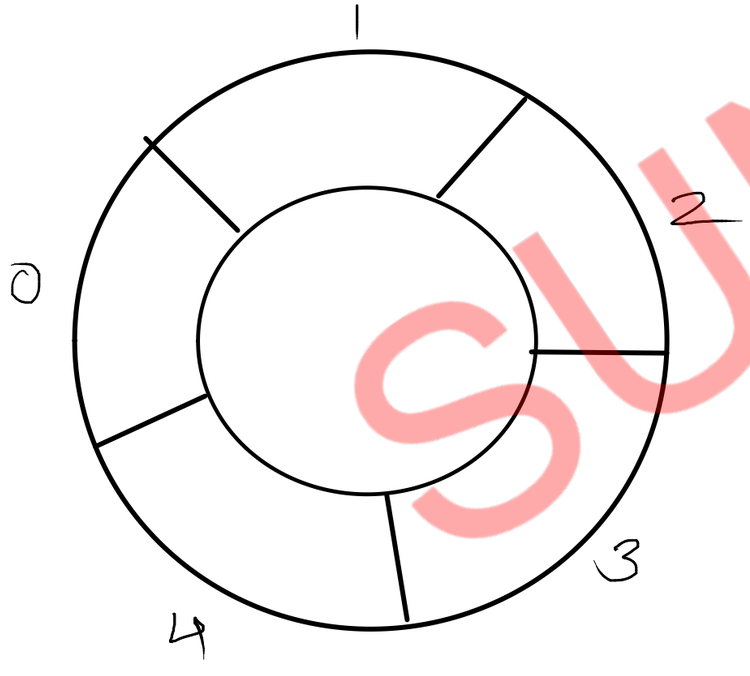
b. add value/data at rear index

2. Delete/Remove/Dequeue/Pop:

a. reposition front (inc)

3. Peek:

a. read data/value from front end



; size = 4

front = (front + 1) % size

rear = (rear + 1) % size

front = rear = -1

= (-1+1) % 4 = 0

= (0+1) % 4 = 1

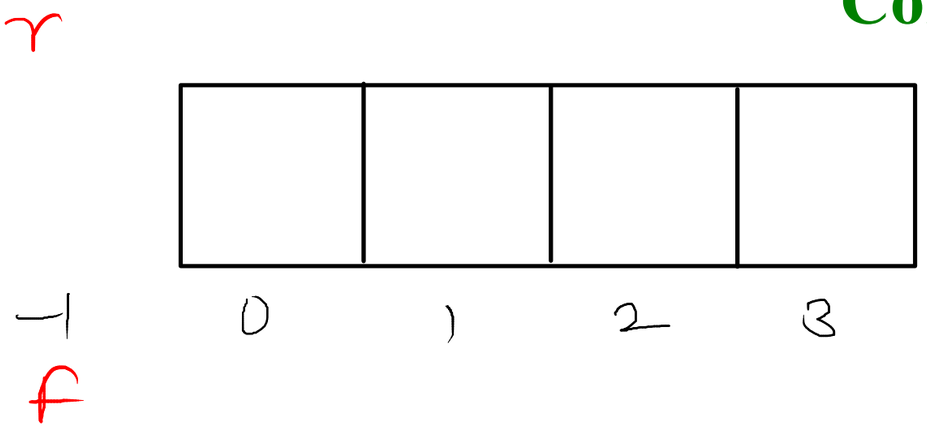
= (1+1) % 4 = 2

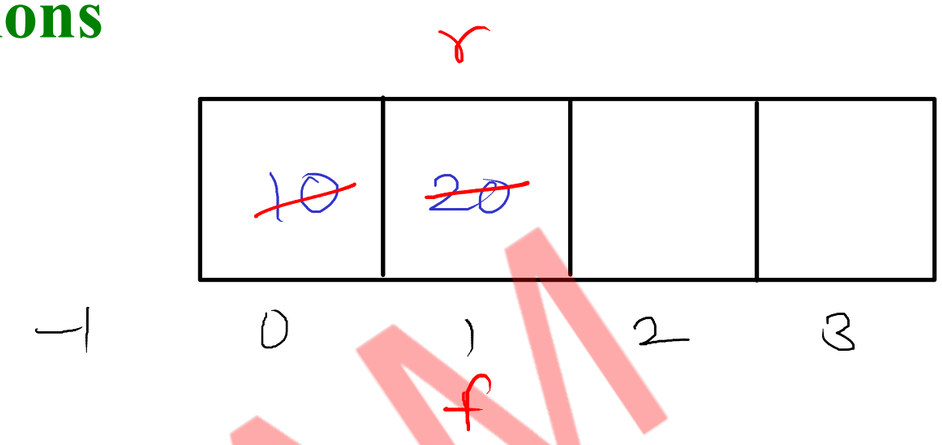
= (2+1) % 4 = 3

= (3+1) % 4 = 0

Circular Queue

Conditions



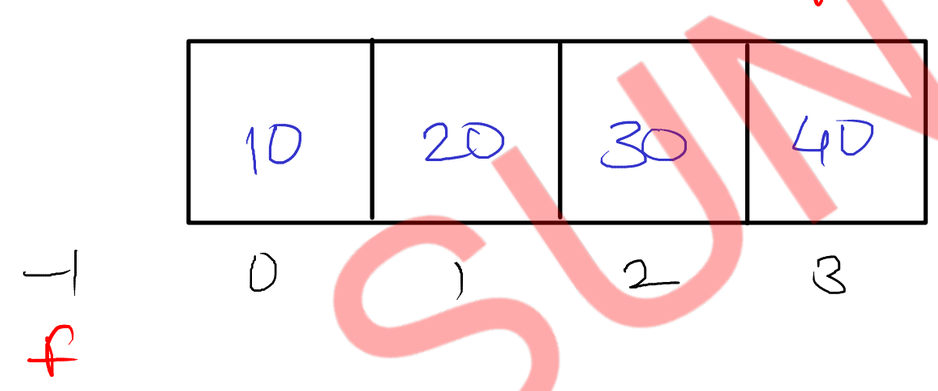


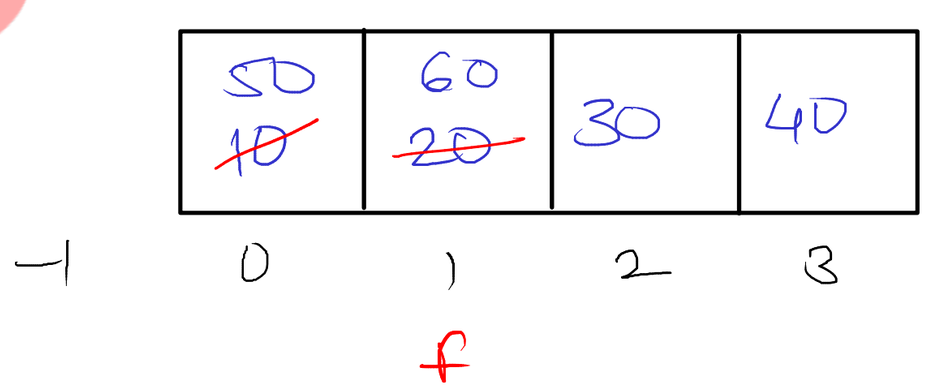
Empty:

```
front == rear && rear == -1
```

Pop:

```
front = (front + 1) % SIZE;
if(front == rear)
    front = rear = -1;
```





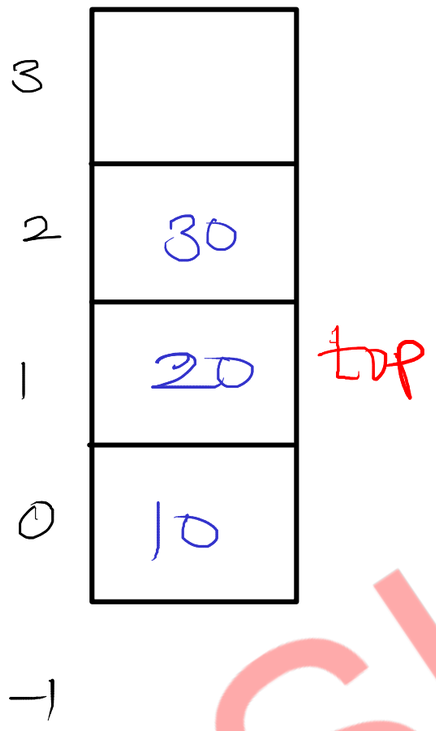
Full:

```
front == -1 && rear == SIZE-1
```

```
front == rear && rear != -1
```

Stack

- it is a linear data structure which stores similar type of data
- it has only one end (top) to insert and delete data
- stack works on principle of "Last In First Out" / "LIFO"
- top always points to last inserted data



- all operations of stack, can be performed in O(1) time complexity

Operations:

1. Add/Insert/Push:

1. reposition top (inc)
2. add value/data at top index

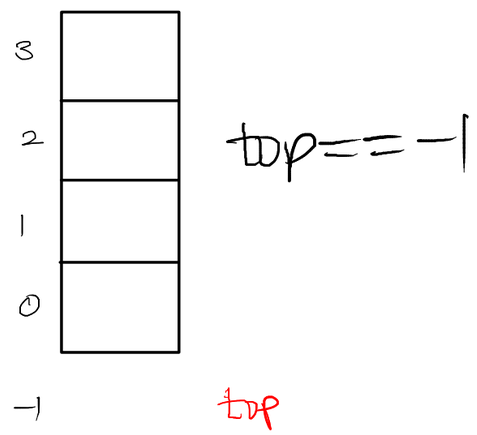
2. Delete/Remove/Pop:

1. reposition top (dec)

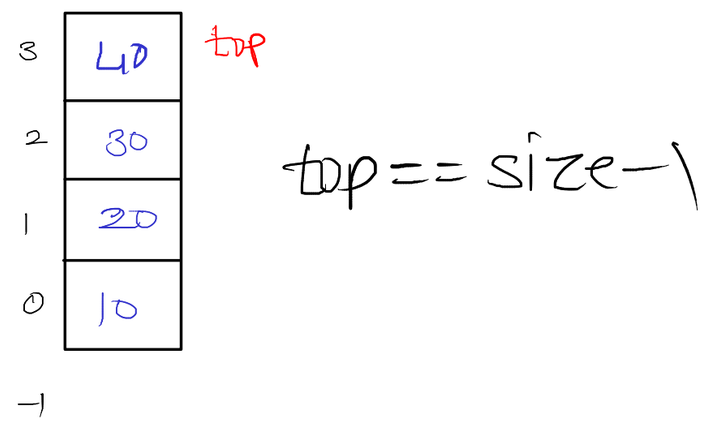
3. Peek:

1. read / return data of top end(index)

Empty



Full



Stack and Queue Time Complexity Analysis
(Array Implementation)

	Stack	Linear Queue	Circular Queue
Push	O(1)	O(1)	O(1)
Pop	O(1)	O(1)	O(1)
Peek	O(1)	O(1)	O(1)

Stack Application

Expression Evaluation and Conversion

1. Postfix Evaluation
2. Prefix Evaluation
3. Infix to Postfix Conversion
4. Infix to Prefix Conversion

Expression:

- set/combination of operands and operators

operands - values/variables

operators - mathematical symbols (+, -, /, *, %)

e.g. a + b, 4 * 2 - 3

Types:

1. Infix	a + b	human
2. Prefix	+ a b	computer
3. Postfix	a b +	computer

Operators:

()

power

* / %

+ -

Postfix Evaluation

Postfix : 4 5 6 * 3 / + 9 + 7 -

left ⟶ right

Result = 16

23 − 7
= 16

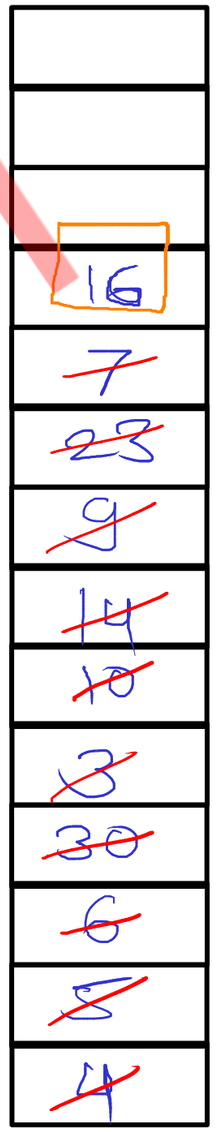
14 + 9
= 23

4 + 10
= 14

30 / 3
= 10

5 * 6
= 30

Stack



Prefix Evaluation

Prefix : - + + 4 / * 5 6 3 9 7

left ← right

25 − 7
= 16

Result = 16

14 + 9
= 23

4 + 10
= 14

30 / 3
= 10

5 * 6
= 30