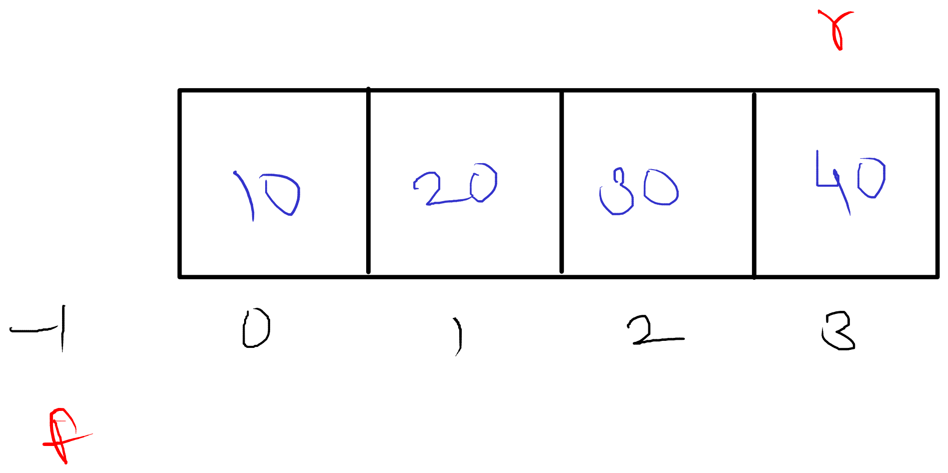
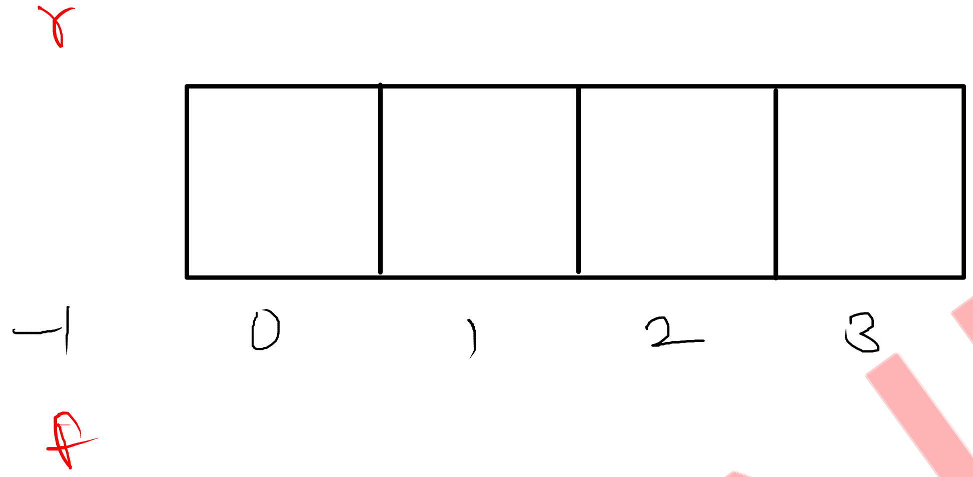
Linear Queue

Full →



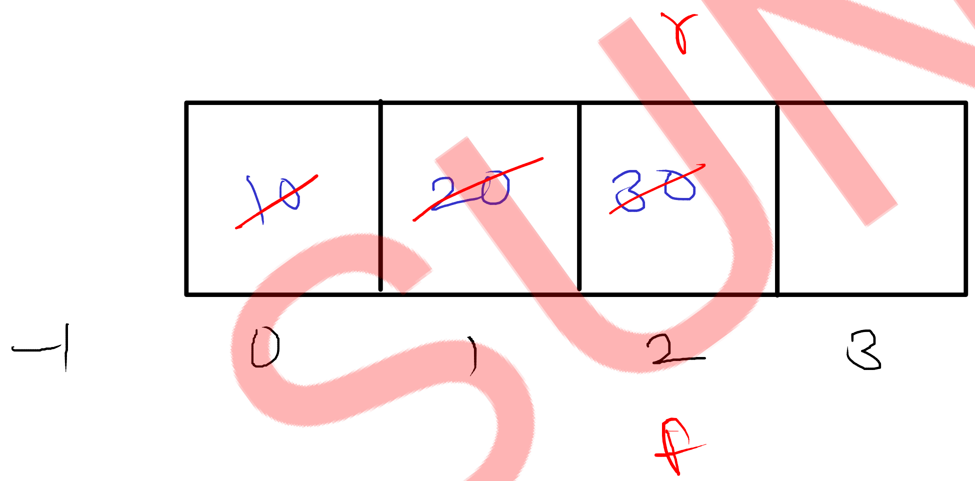
r= size-1

Emply →



r==f

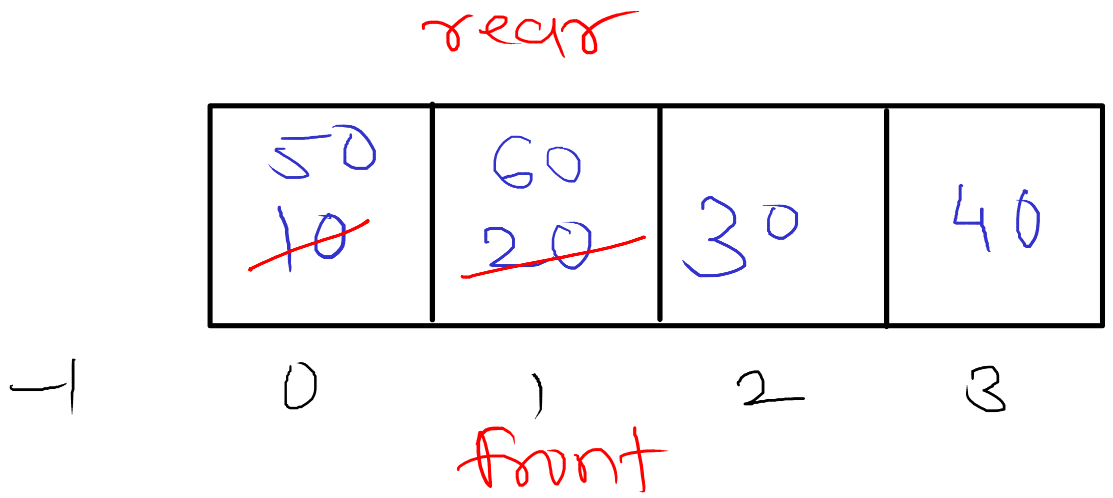
Emply →



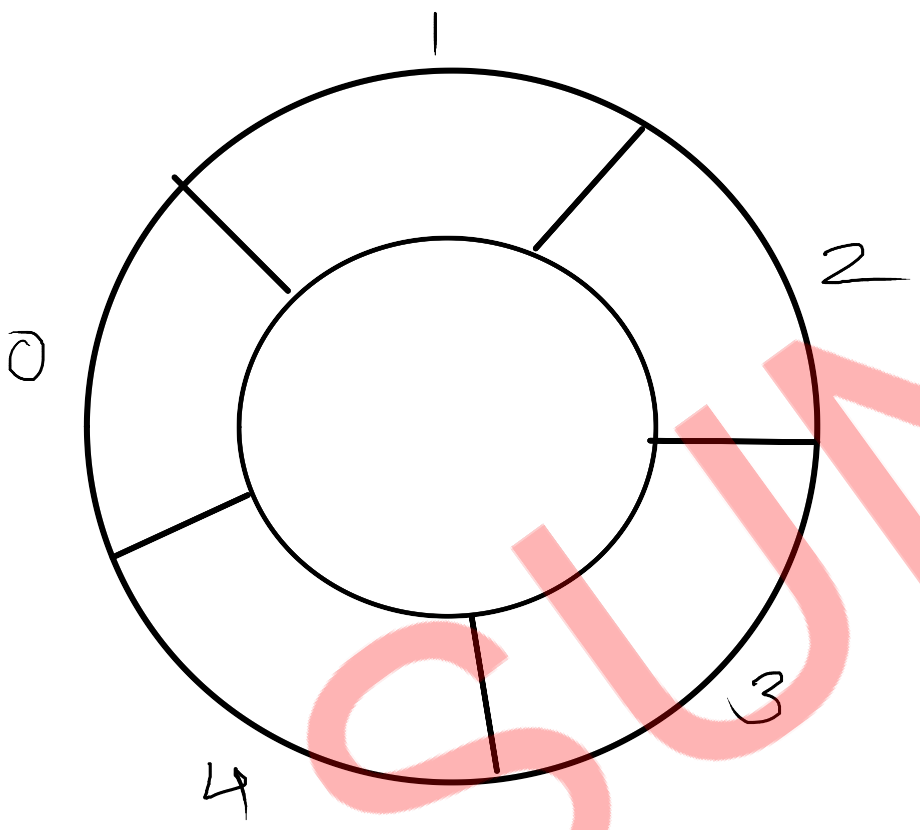
r==f

- once rear is reached to last index of array & few initial locations are empty, still we are not able to utilise them, this will lead to poor memory utilization

Circular Queue



f=r=-1



front = (front+1) % size

rear = (rear+1) % size

Operation:

1> Push

i) reposition rear

ii) add value at rear index

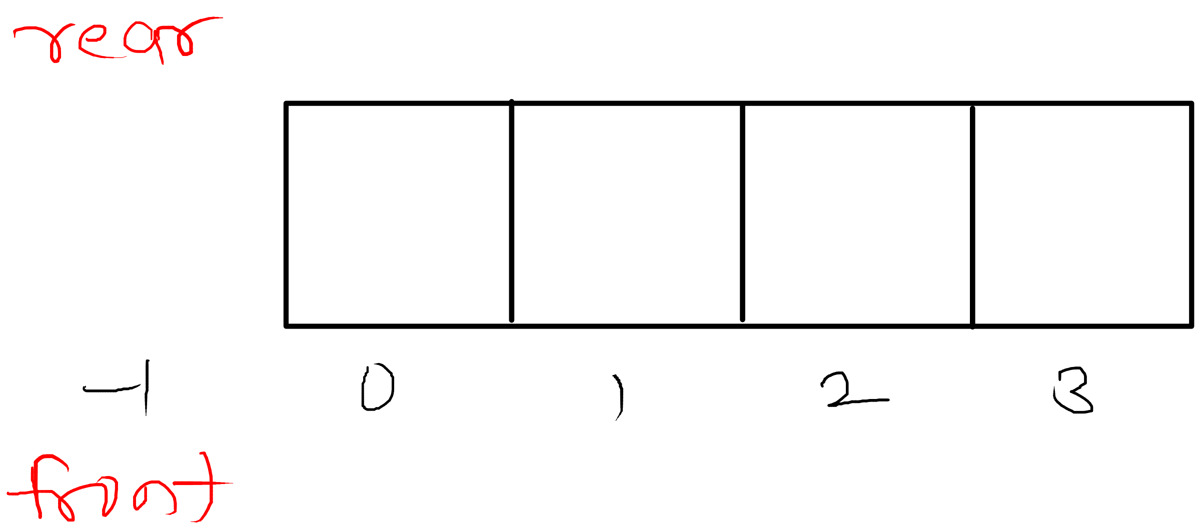
2> Pop

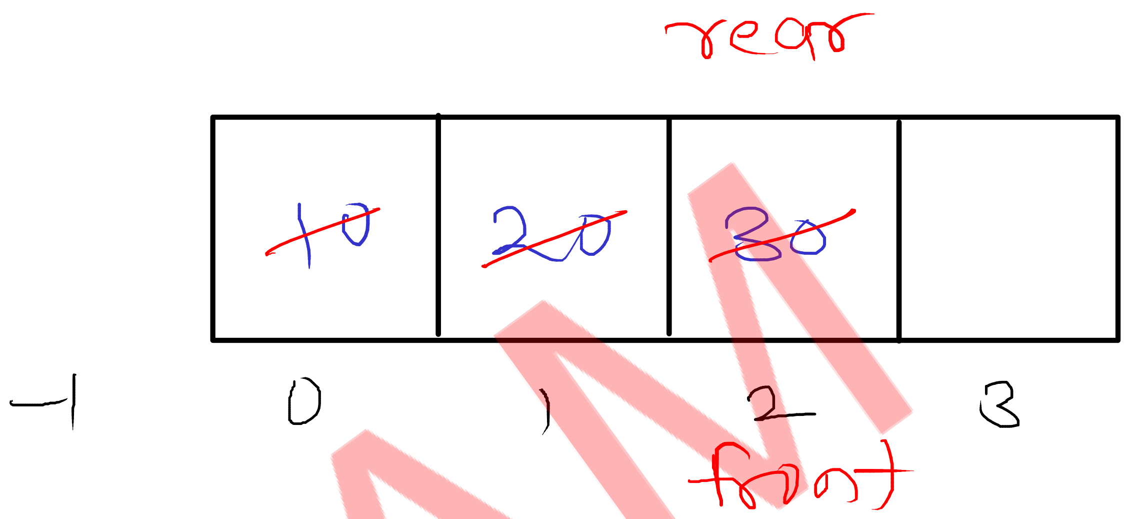
i) reposition front

3> peek

i) read from front read

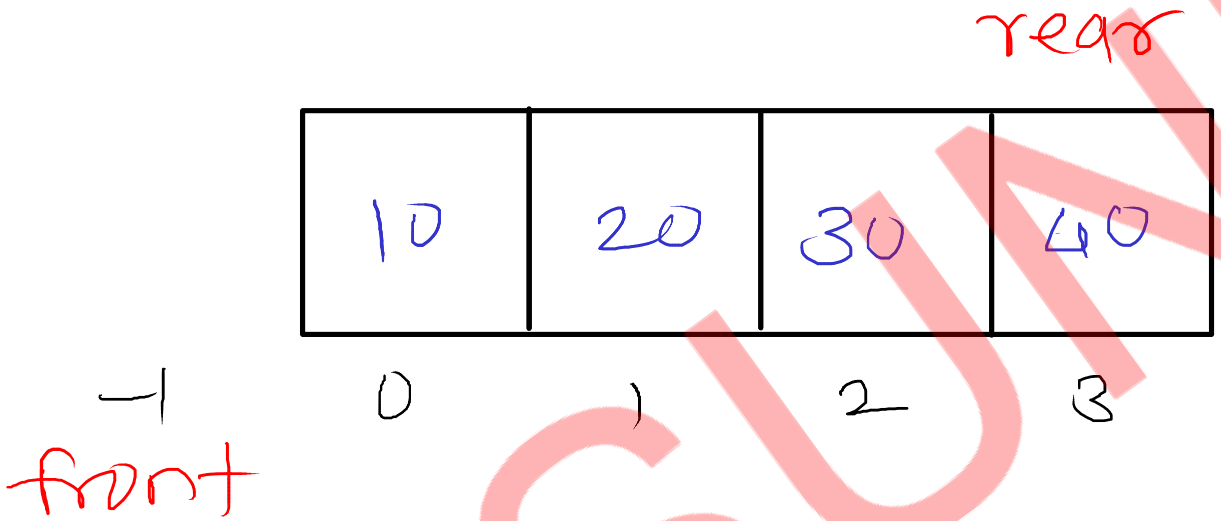
-All operations can be performed in $\underline{O(1)}$ time complexity

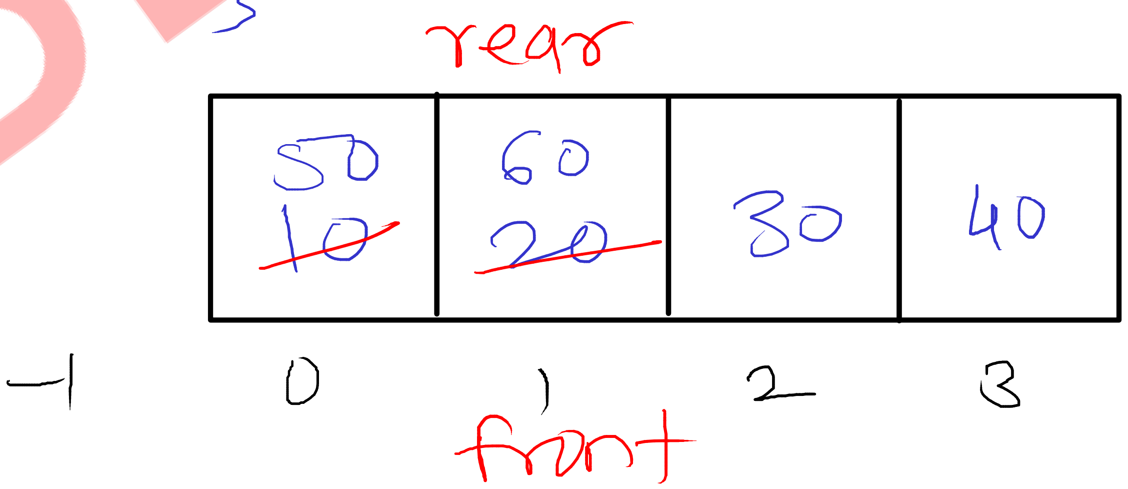




Empty: rear==front && rear==-1

```
pop() {
  front = (front+1)% size;
  if(front==rear)
     front=rear=-1
}
```





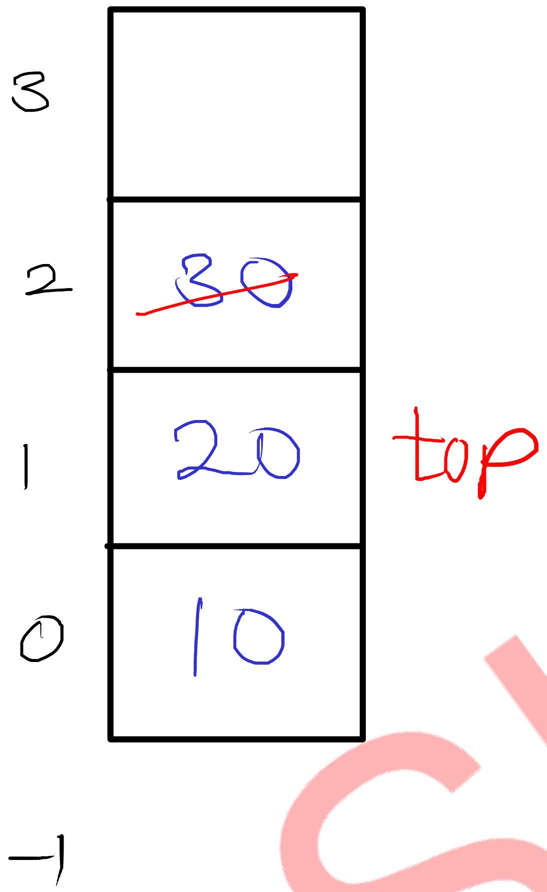
front==-1 && rear==size-1

rear==front && rear!=-1

Full: (front==-1 && rear=size-1) || (rear==front && rear!=-1)

Stack

- linear data structure which stores similar data
- insertion and removal of data is allowed only from one end (top)
- works on principle of "Last In First Out" (LIFO)
- top always points to the last inserted data



- All operations are performed in O(1) time complexity

Operations

1. Push

i. reposition top (inc)

ii. add value at top index

2. Pop

i. reposition top (dec)

3. Peek

i. read data of top index

Conditions

1. isEmpty

top == -1

2. isFull

top == size - 1

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

5 23 – 7
= 16

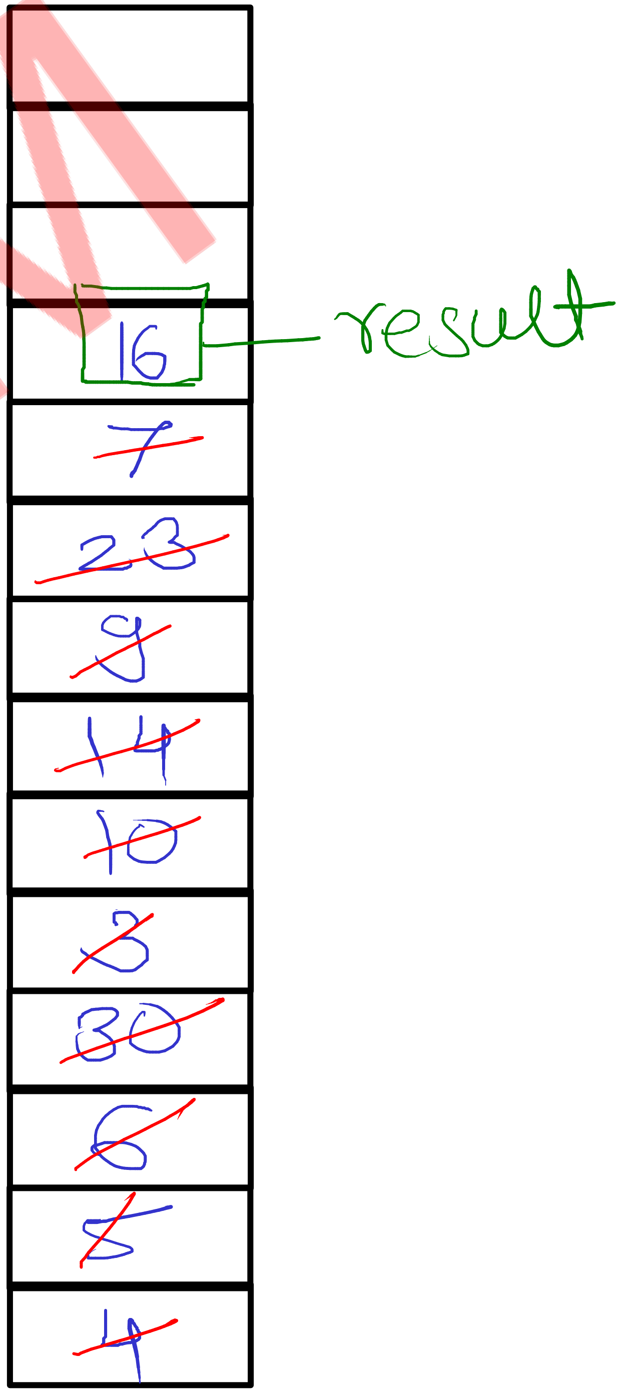
4 14 + 9
= 23

3 4 + 10
= 14

2 30 / 3
= 10

1 5 * 6
= 30

Stack



Prefix Evaluation

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

left ← right

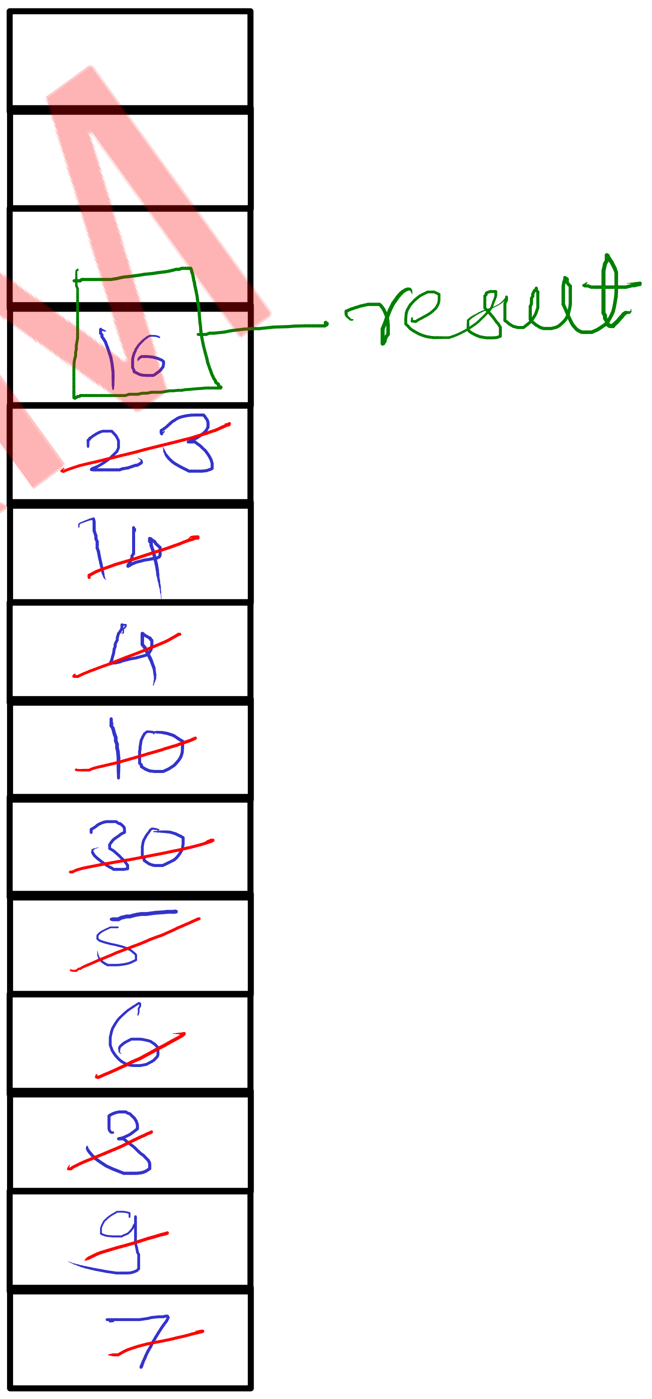
⑤ 23 − 7
= 16

④ 14 + 9
= 23

③ 4 + 10
= 14

② 30 / 3
= 10

① 5 * 6
= 30

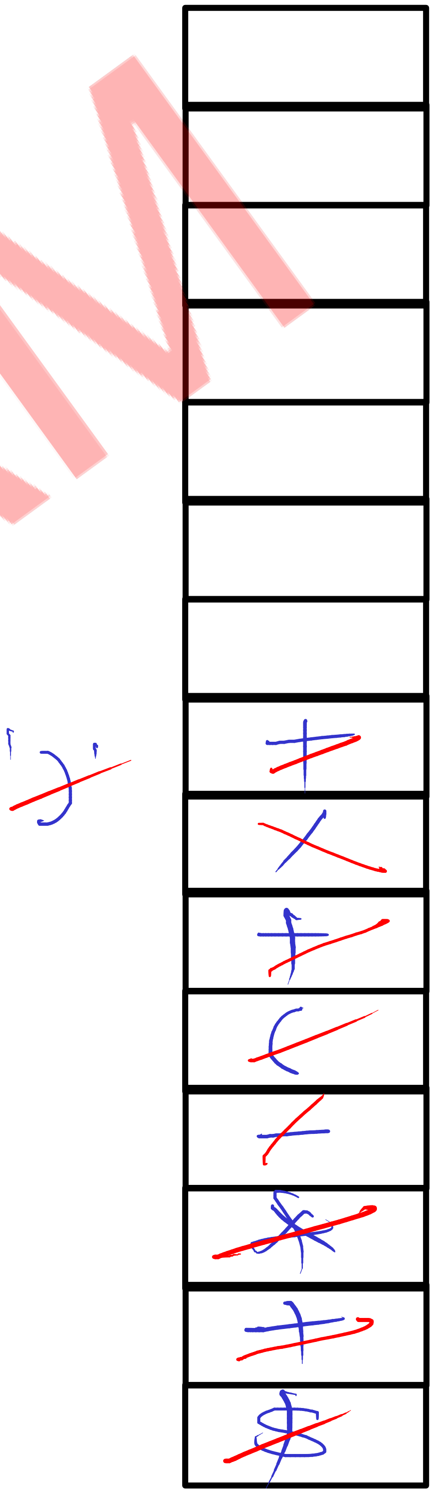


Infix to Postfix conversion

Infix : 1 $ 9 + 3 * 4 - (6 + 8 / 2) + 7

left → right

Postfix : 1 9 $ 3 4 * + 6 8 2 / + - 7 +



( )
$
* / %
+ -

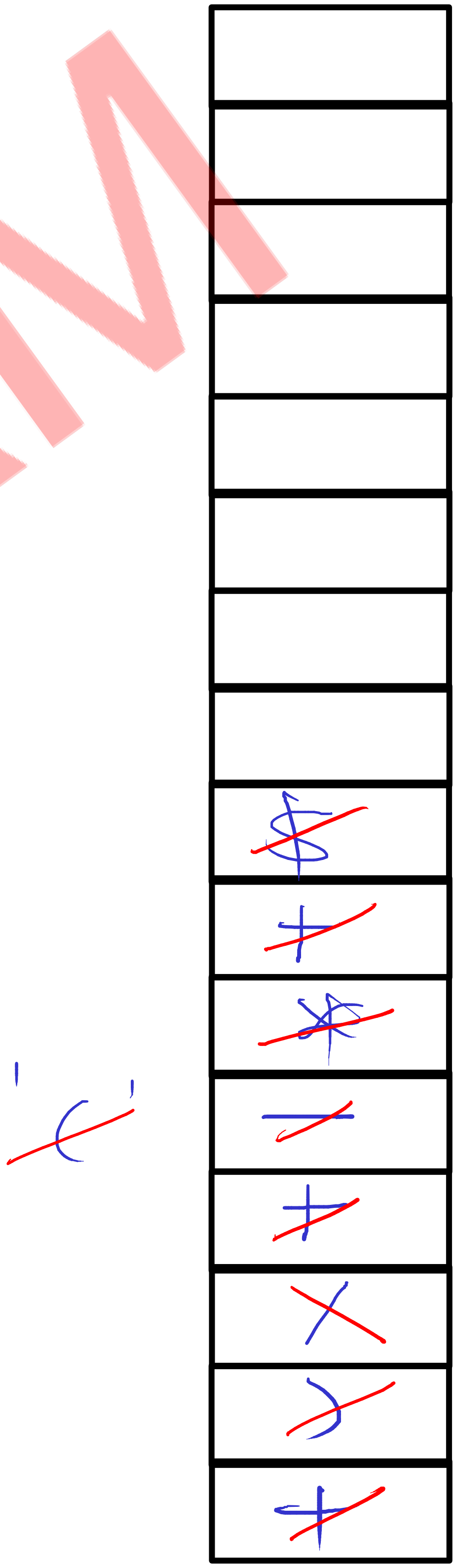
Infix to Prefix conversion

Infix : 1 $ 9 + 3 * 4 - (6 + 8 / 2) + 7

left ← right

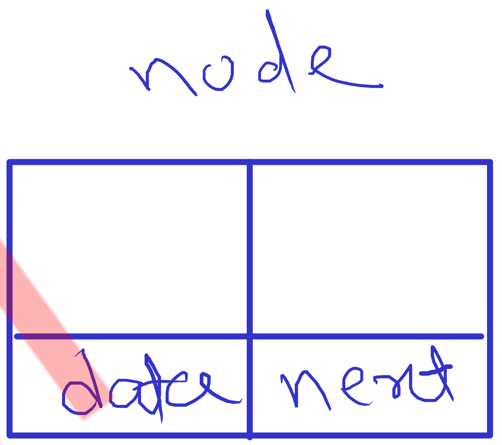
728/6+43*91$+-+

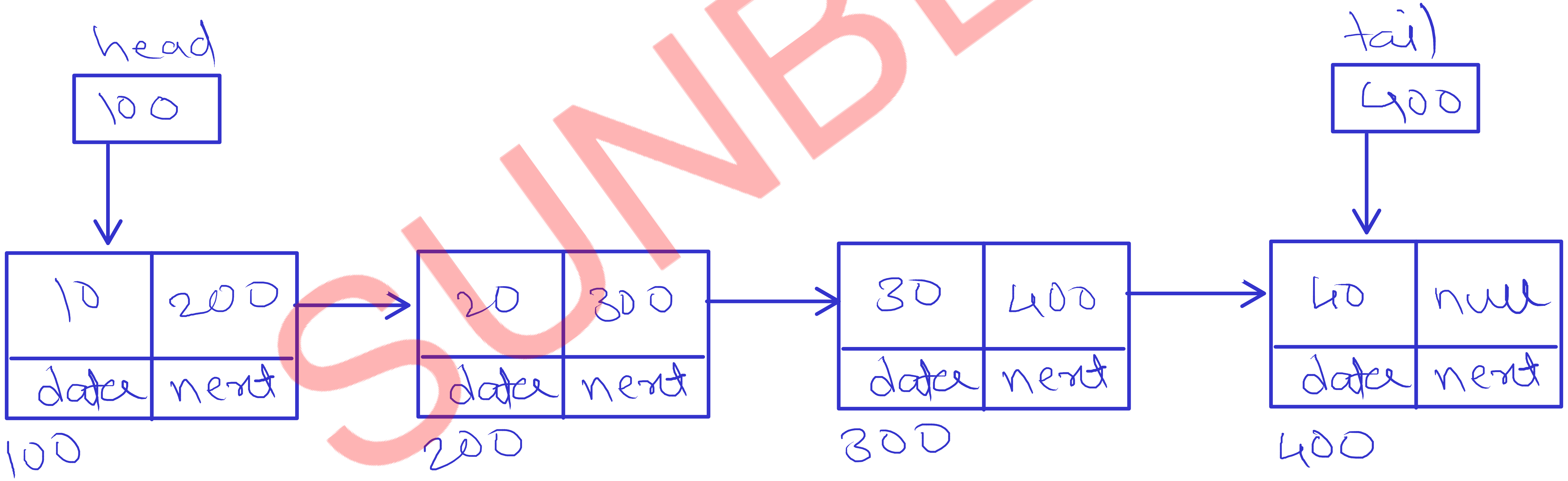
prefix : +-+$19*34+6/827



Linked List

- linear data structure
- link of next data is kept with previous/current data
- every element is known as "node" and it consist of two part
    - data - actual data of the node
    - link - address/referance of next data/node
- address of first node is kept into head pointer/referance
- address of last node is kept into tail pointer/referance (optional)





Linked list Operations

1. Add first
2. Add last
3. Add pos (in between)

4. Delete first
5. Delete last
6. Delete pos (in between)

7. Traverse (Display)

8. reverse
9. search
10. sort

Linked List Types:

1. Singly Linear linked List
2. Singly Circular linked list
3. Doubly Linear linked list
4. Doubly Circular linked list