Insertion Sort

//1. pick one element(start from seccond position) from array
//2. compare picked element with all its left neighbours one by one
//3. if left neighbour is greater than picked element then move left
    // neighbour one position ahead
//4. insert picked element at its appropriate position
//5. repeat above steps untill array is sorted

pass 1 → 1
pass2 → 2
pass 3 → 3
⋮
pass n → n-1

Total comparisions = 1+2+3+ ------ + n-1

$\not n$

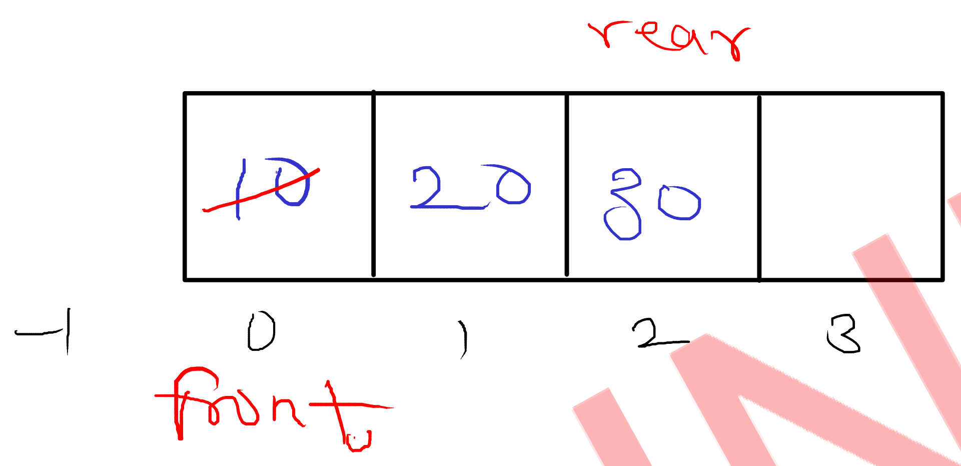
$$= \frac{n(n-1)}{2}$$

$$\approx n^2$$

$\boxed{T(n) = O(n^2)}$ $\boxed{\text{Best case} = O(n)}$

Linear Queue

- it is a linear data structure (data is arranged sequentially)
- data is inserted from one end of the queue (rear)
- data is removed from another end of the queue (front)
- queue works on principle of "First In First Out" / "FIFO"



Operations:

1. Insert/Add/Enqueue/Push:

a. reposition rear (inc)
b. add value/data at rear index

2. Remove/Delete/Dequeue/Pop:

a. reposition front (inc)

3. Peek/Collect:

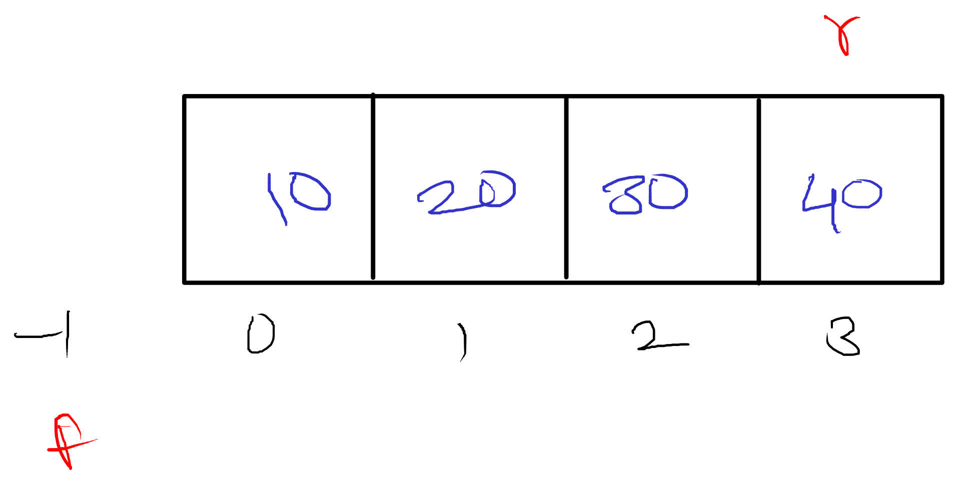
a. read/return data/value of front end

- All operations of queue are performed in constant time / O(1)

Linear Queue

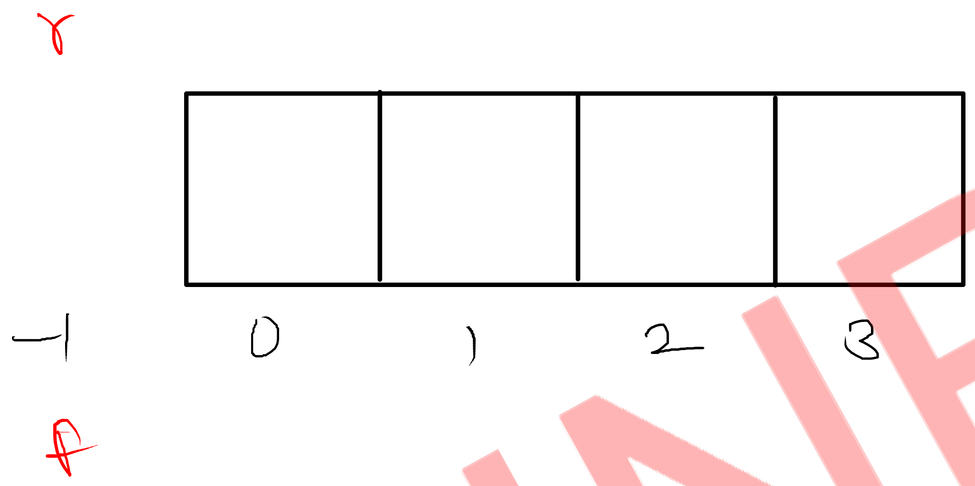
conditions

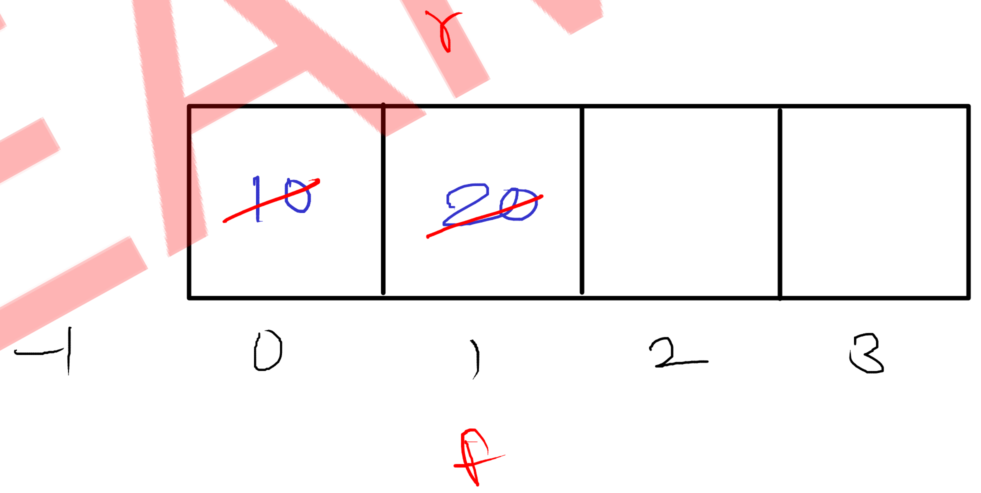
Full:



rear == size - 1

Empty:



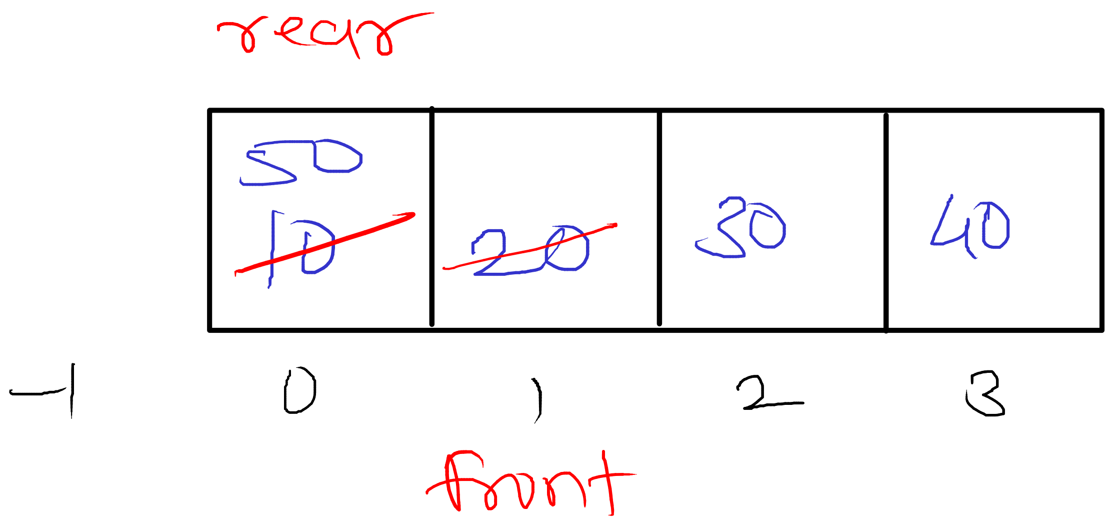


front == rear

- if rear is at last index and initial few locations are empty in linear queue, still we will not be able to use those empty locations. This will lead to poor memory utilization

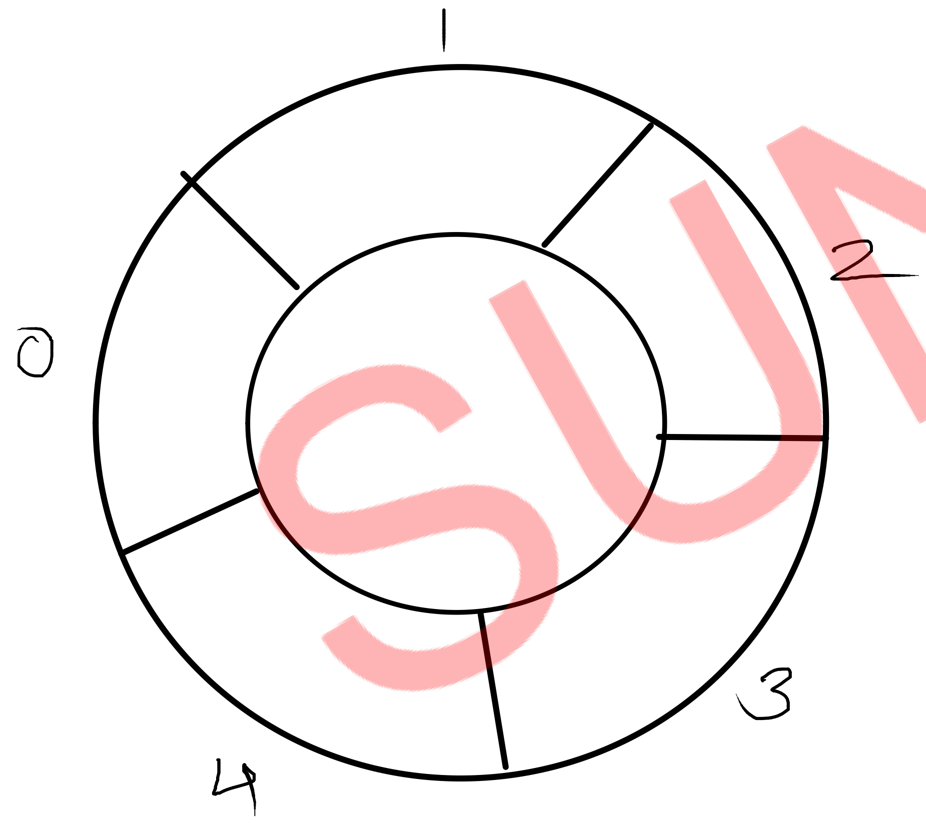
- solution for above problem is circular queue size=4

Circular Queue



rear = (rear + 1) % size
front = (front + 1) % size

rear = front = -1
= (-1+1) % 4 = 0
= (0+1) % 4 = 1
= (1+1) % 4 = 2
= (2+1) % 4 = 3
= (3+1) % 4 = 0



Operations:

1. Insert/Add/Enqueue/Push:

a. reposition rear (inc)
b. add value/data at rear index

2. Remove/Delete/Dequeue/Pop:

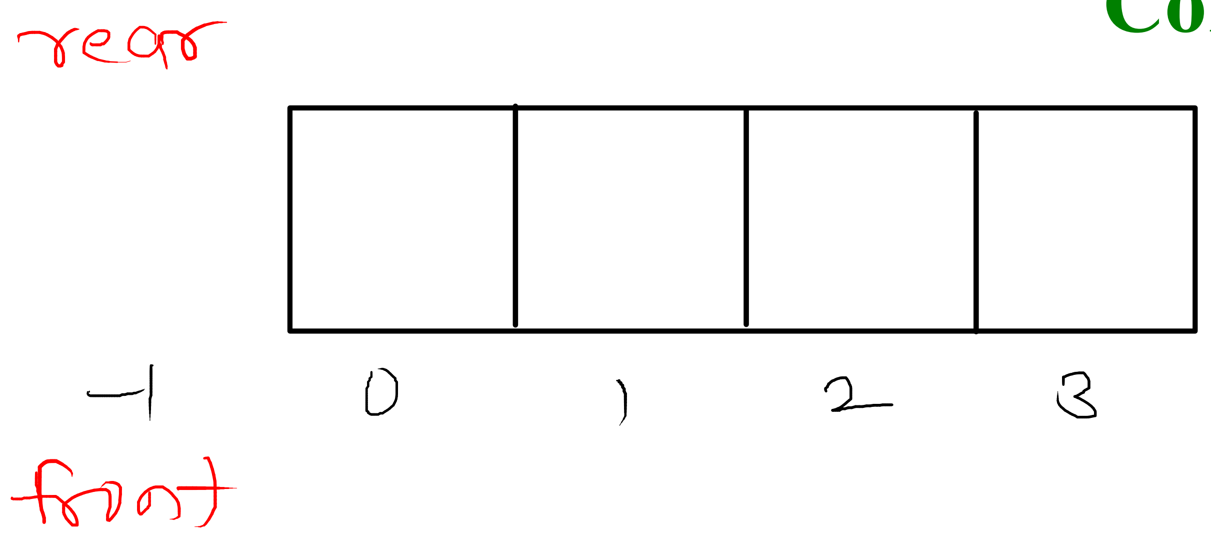
a. reposition front (inc)

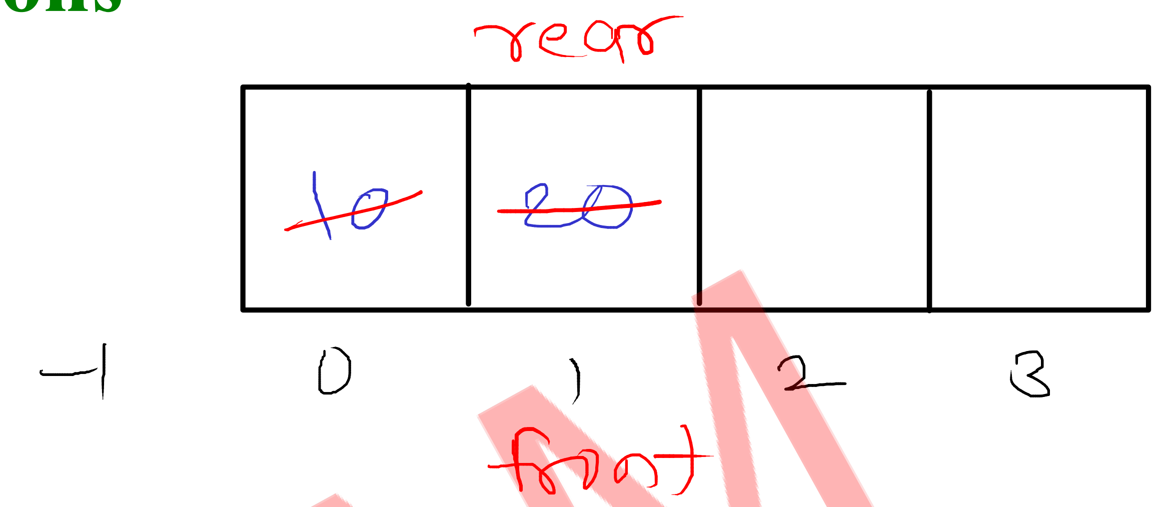
3. Peek/Collect:

a. read/return data/value of front end

Circular Queue

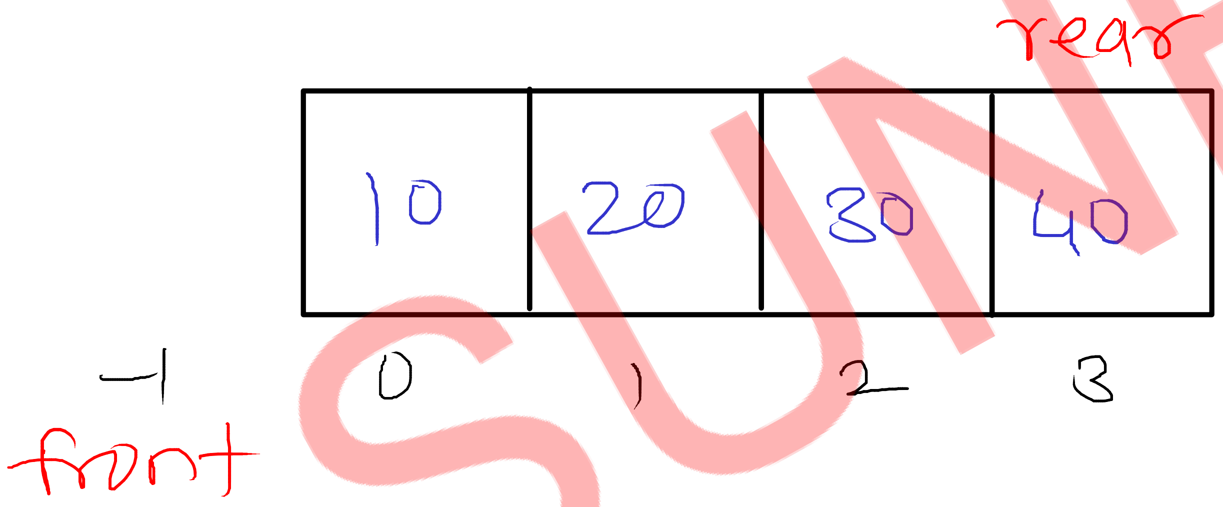
Conditions

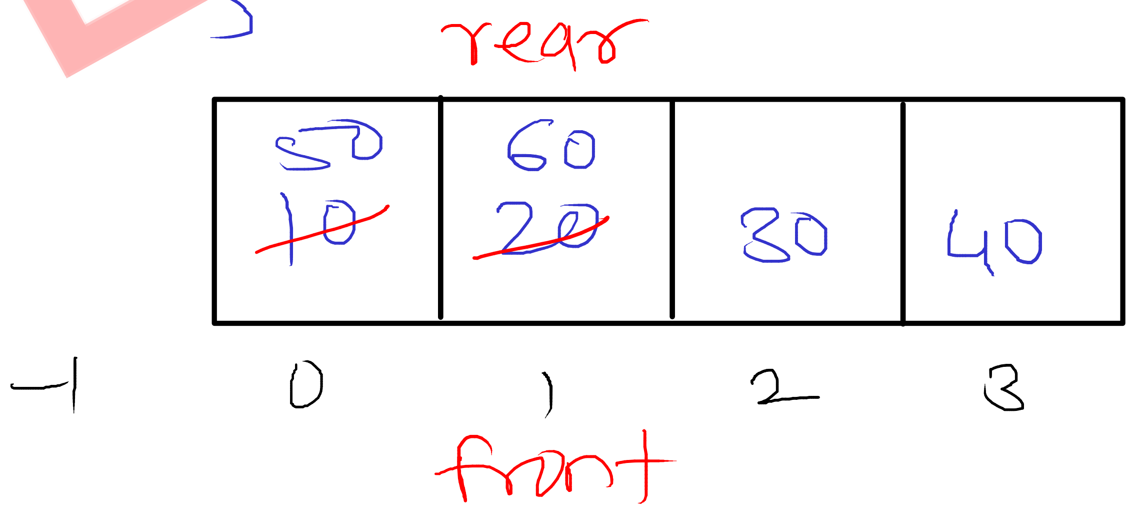




Empty : front == rear && rear == -1

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



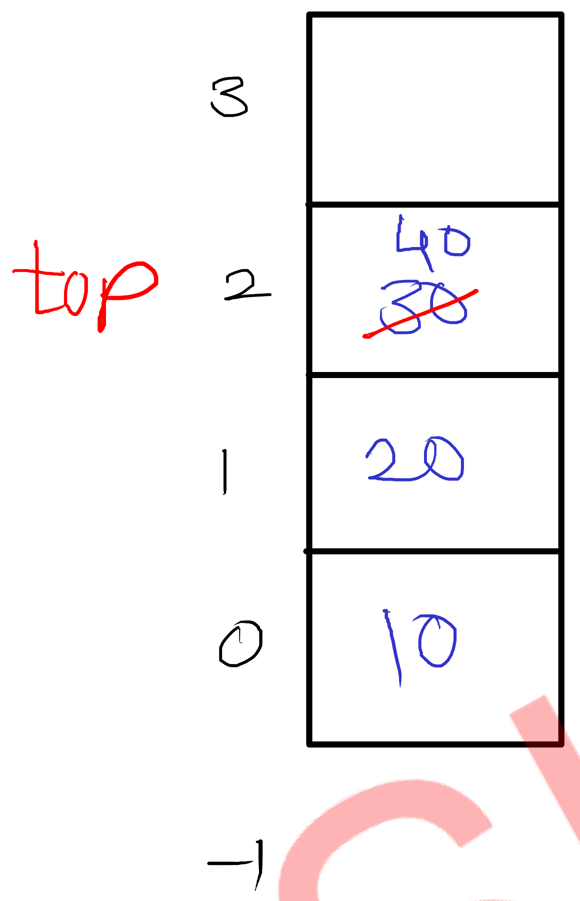


front == -1 && rear == size-1   front == rear && rear != -1

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

Stack

- it is a linear data structure(data is stored sequentially)
- insert and remove of data is allowed from single end (top)
- stack follows principle of "Last In First Out" / "LIFO"
- top will always points to last inserted data



Operations:

1. Insert/Add/Enqueue/Push:

//a. reposition top (inc)
//b. add value/data at top index

2. Remove/Delete/Dequeue/Pop:

//a. reposition top (dec)

3. Peek/Collect:

//a. read/return data of top index

- all operations of stack can be performed in constant time

O(1)

Conditions:

1. Full

top == size -1

2. Empty

top == -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:

$\uparrow$

()
power
* / %
+ -

Postfix Evaluation

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

$\text{left} \xrightarrow{\hspace{3em}} \text{right}$

(5) $23 - 7$
$= 16$

(4) $14 + 9$
$= 23$

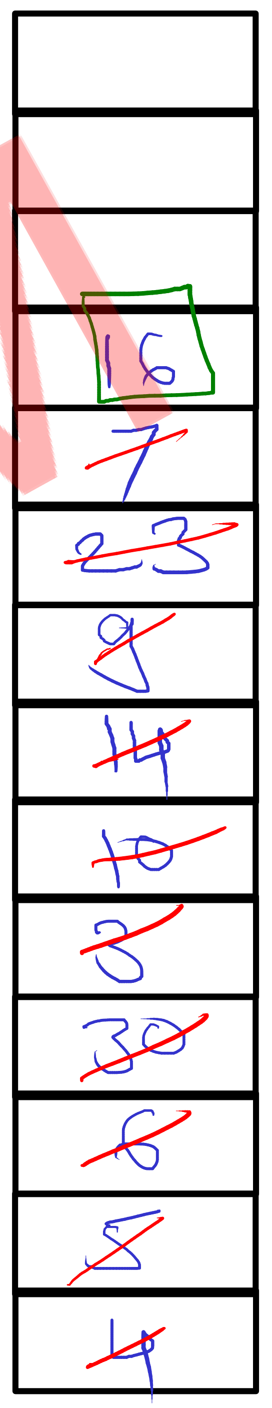
(3) $4 + 10$
$= 14$

(2) $30 / 3$
$= 10$

(1) $5 * 6$
$= 30$

Result = 16

Stack

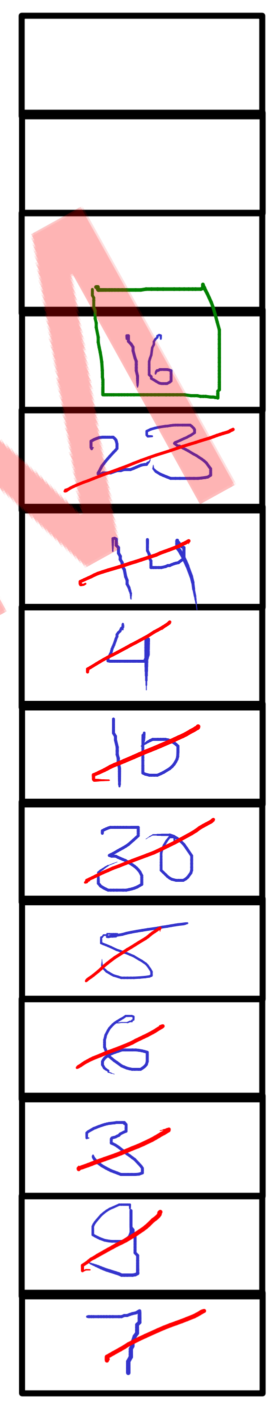


Prefix Evaluation

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

$\text{left} \longleftarrow \text{right}$

Result = 16



23 − 7
= 16

14 + 9
= 23

4 + 10
= 14

30 / 3
= 10

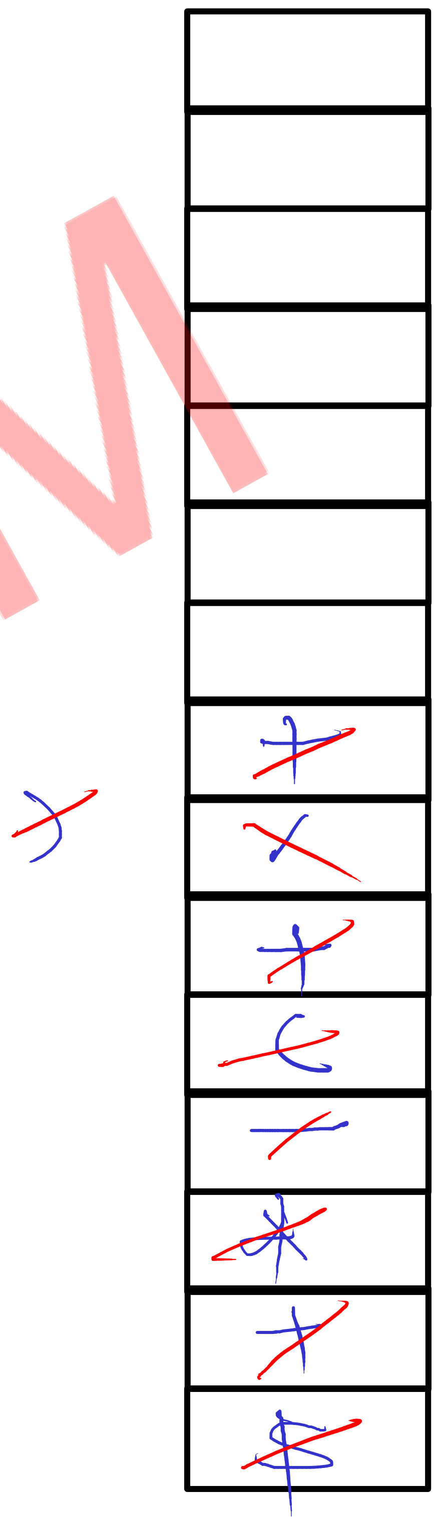
5 * 6
= 30

Infix to Postfix conversion

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

left $\longrightarrow$ right

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

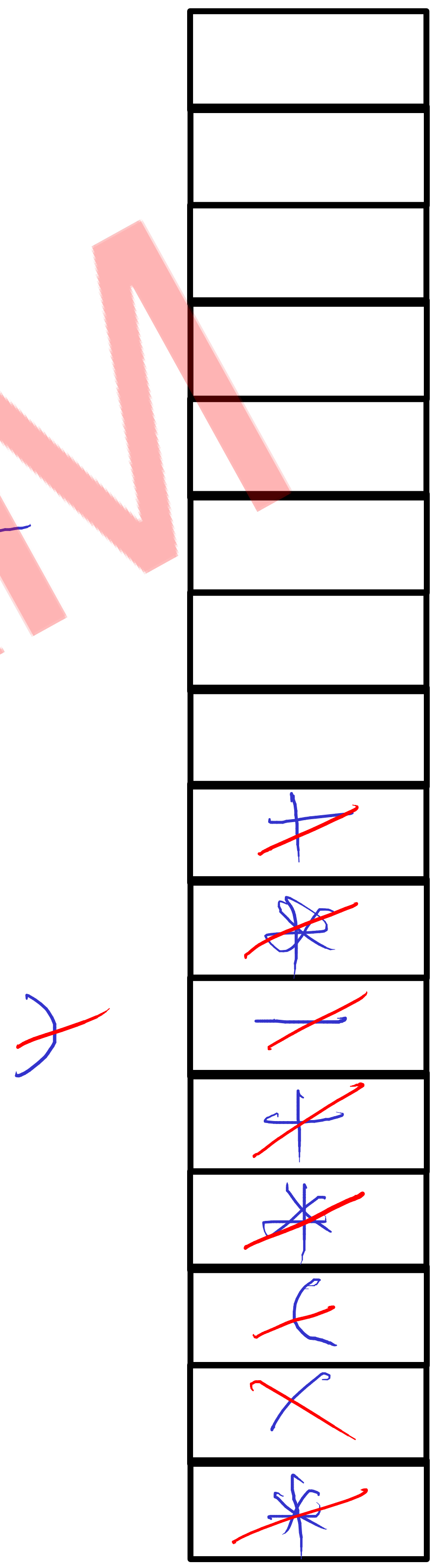


Infix to Postfix conversion

Infix : a * b / (c * d + e) - f * h + i

left $\longrightarrow$ right

Postfix : ab*cd*e+/fh*-i+



Infix to Prefix conversion

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

left ⟵ right

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

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