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
$\boxed{16}$
~~7~~
~~23~~
~~9~~
~~14~~
~~10~~
~~3~~
~~30~~
~~6~~
~~5~~
~~4~~

Prefix Evaluation

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

left ← **right**

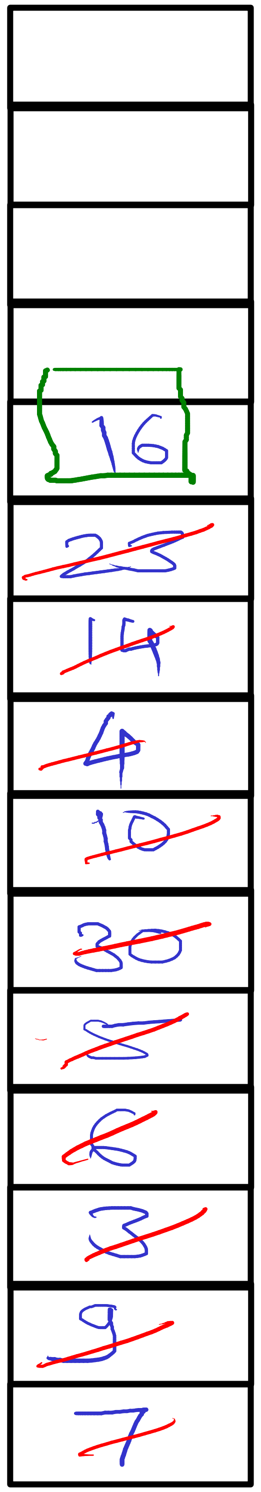
23 − 7
= 16

14 + 9
= 23

4 + 10
= 14

30 / 3
= 10

5 * 6
= 30



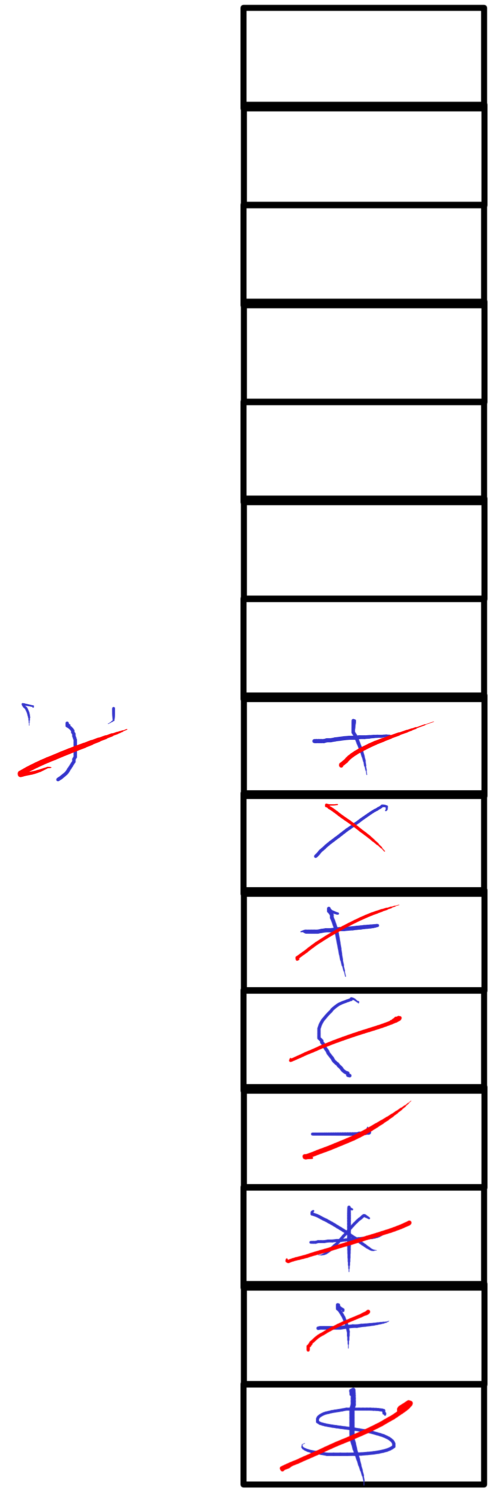
Result = 16

Infix to Postfix conversion

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

left ⟶ right

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



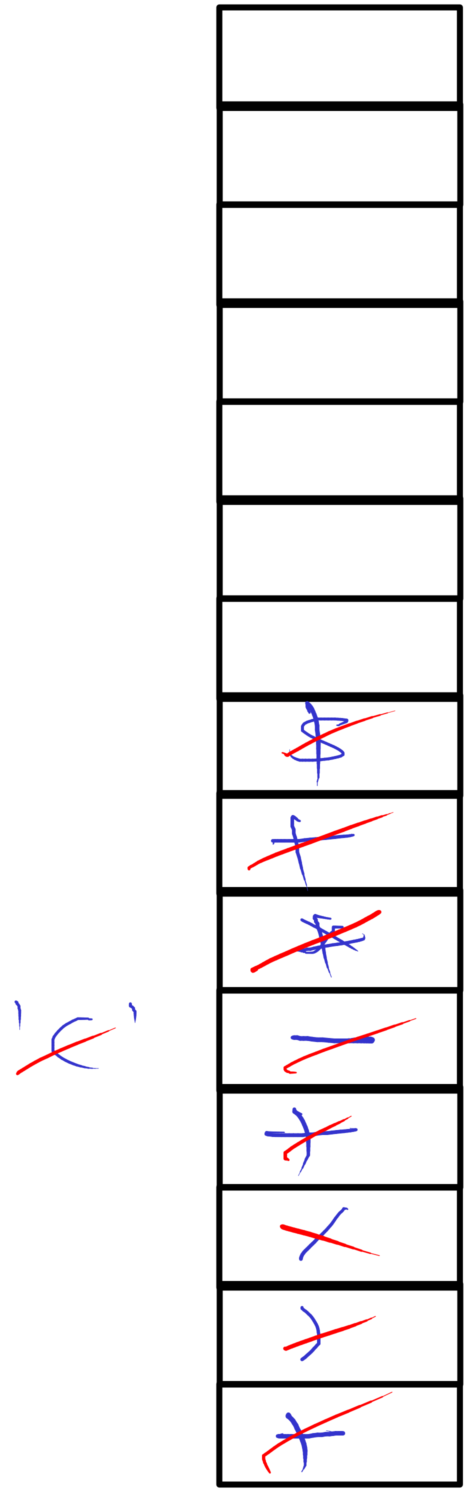
Infix to Prefix conversion

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

left ⟵ right

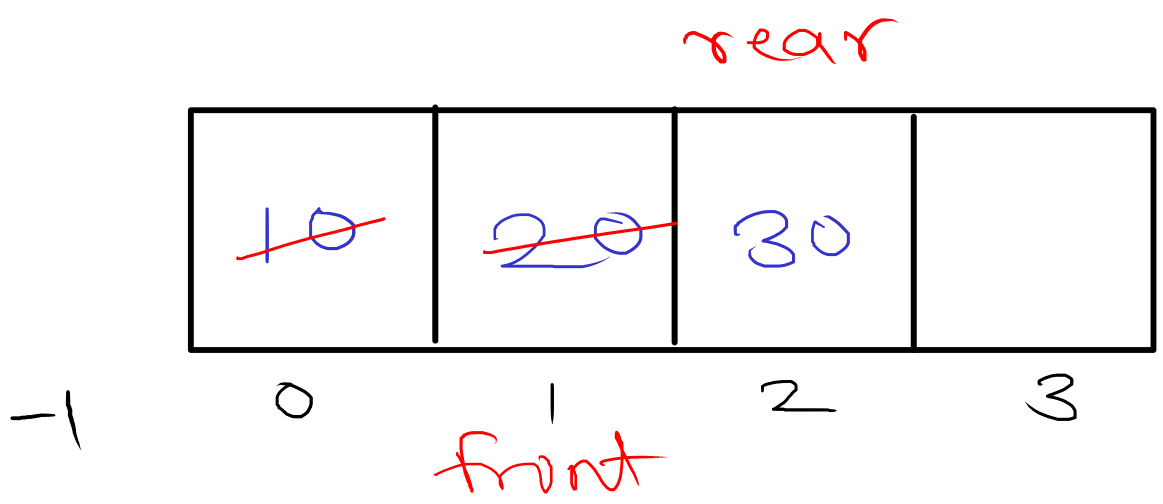
728/6+43*91$+-+

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



Linear Queue

- linear data structure of similar data elements
- insert is allowed from one end and it is called as 'rear'
- remove is allowed from another end and it is called as 'front'
- works on principle of "First In First Out" (FIFO)



Operations:

1. Insert/Add/Push/Enqueue:

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

2. Remove/Delete/Pop/Dequeue:

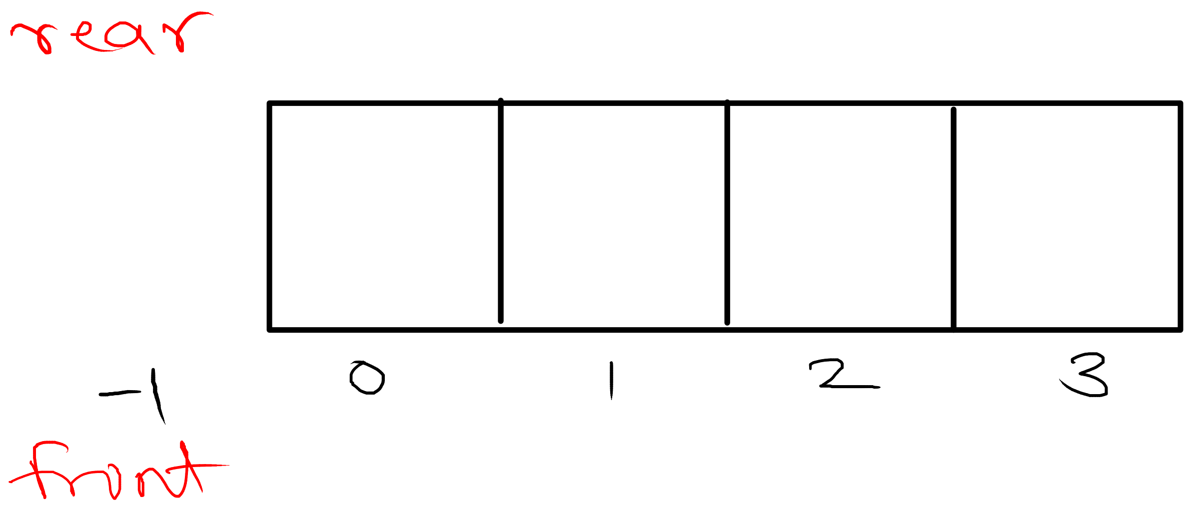
a. reposition front (inc)

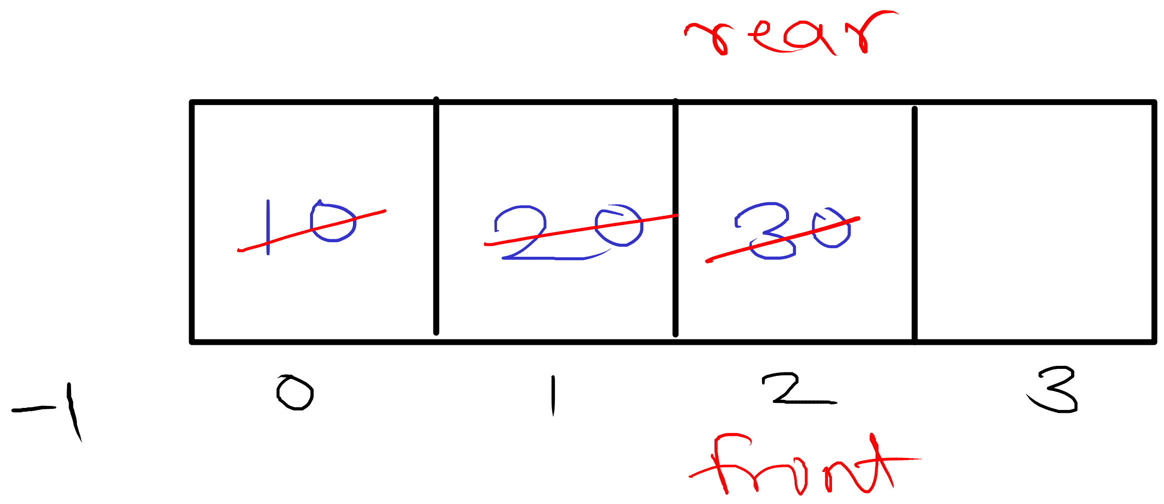
3. Peek

a. read value from front+1 index

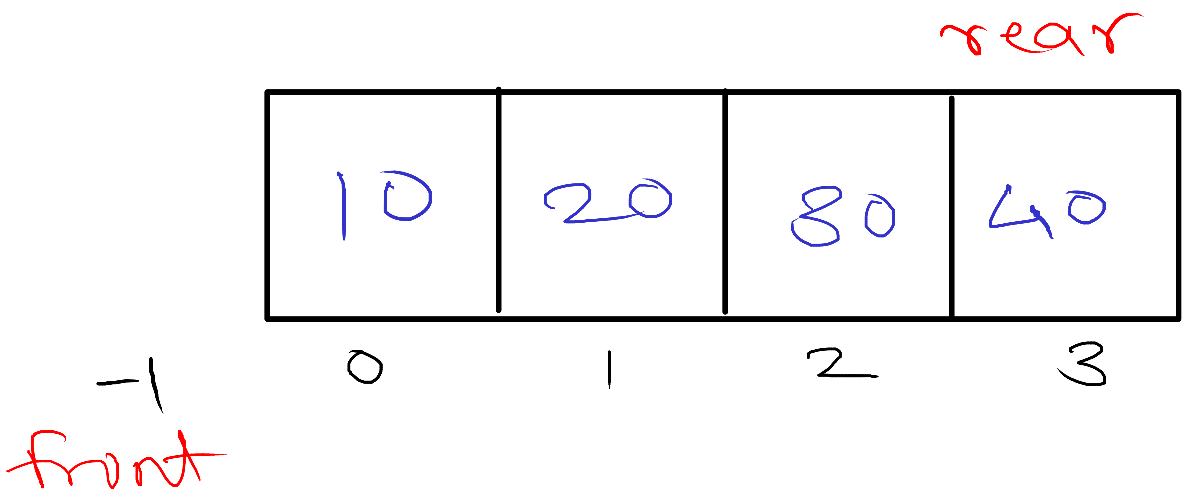
All operations of queue are performed in O(1) time complexity.

Linear Queue - Empty and Full Conditions

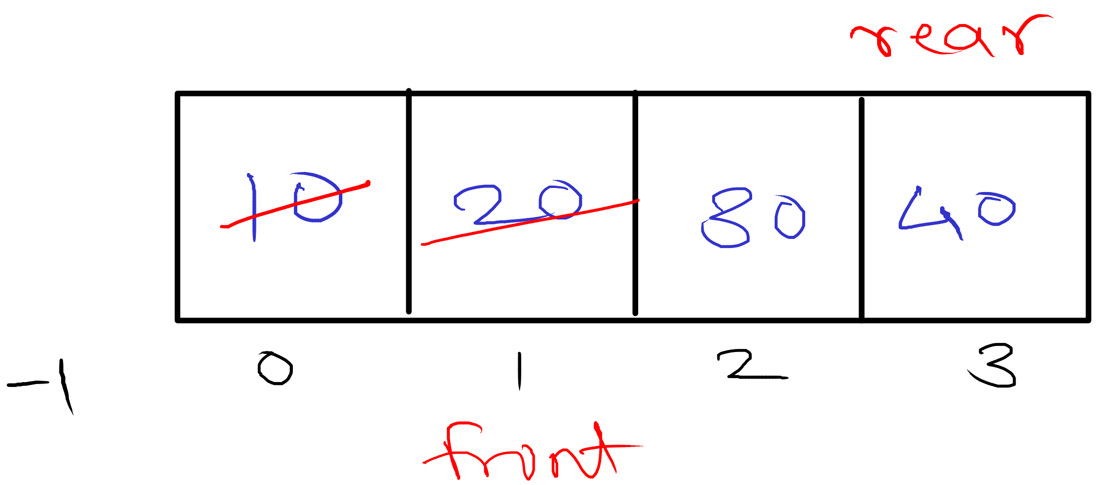




Empty : front == rear

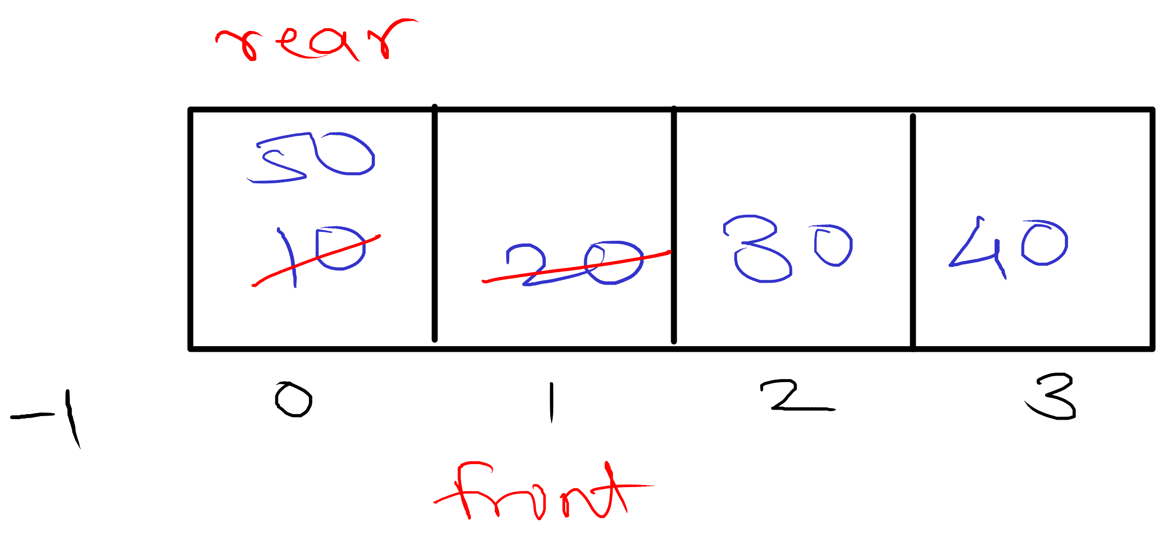


Full : rear == SIZE-1



When rear reaches to last index & few initial locations are empty, we can not utilize them to insert new data.
This will lead to poor memory utilization

Circular Queue



front
rear $= -1, 0, 1, 2, 3, 0, 1, 2 \ldots$

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

front
rear $= -1$

$= (-1+1) \% 4 = 0$

$= (0+1) \% 4 = 1$

$= (1+1) \% 4 = 2$

$= (2+1) \% 4 = 3$

$= (3+1) \% 4 = 0$

Operations:

1. Insert/Add/Push/Enqueue:

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

2. Remove/Delete/Pop/Dequeue:

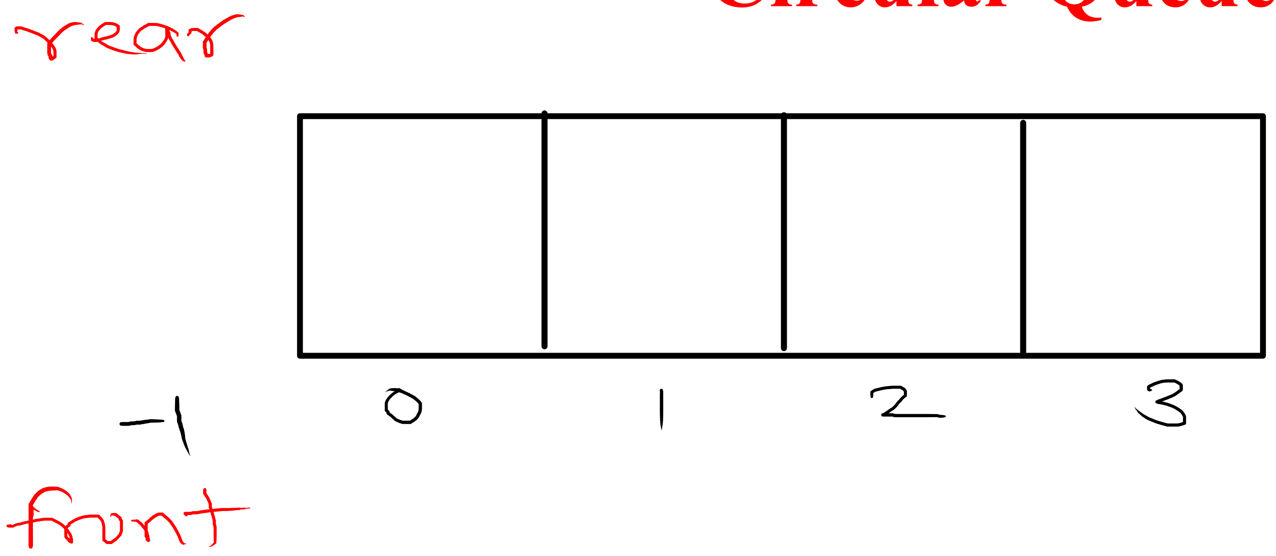
a. reposition front (inc)

3. Peek

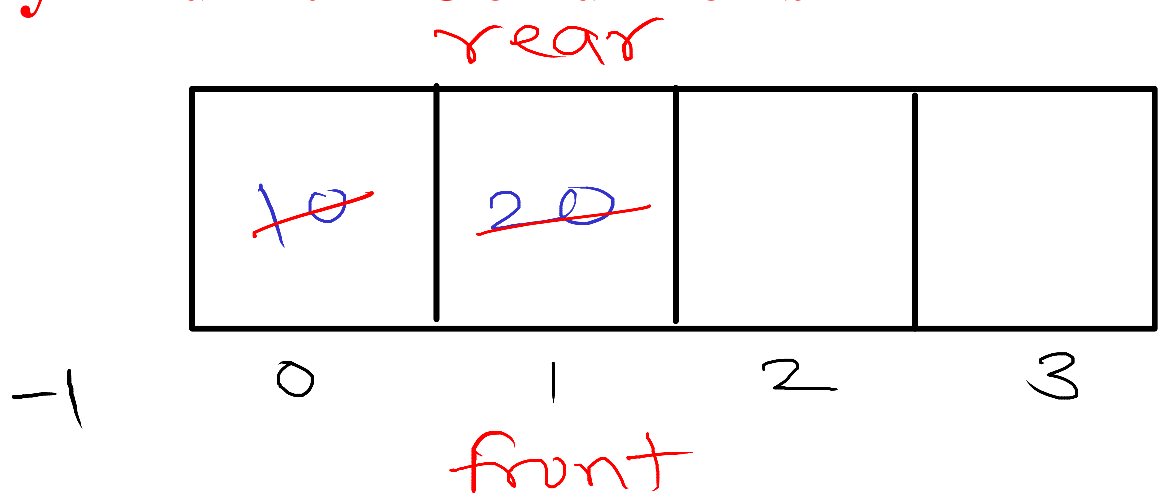
a. read value from front+1 index

All operations of queue are performed in O(1) time complexity.

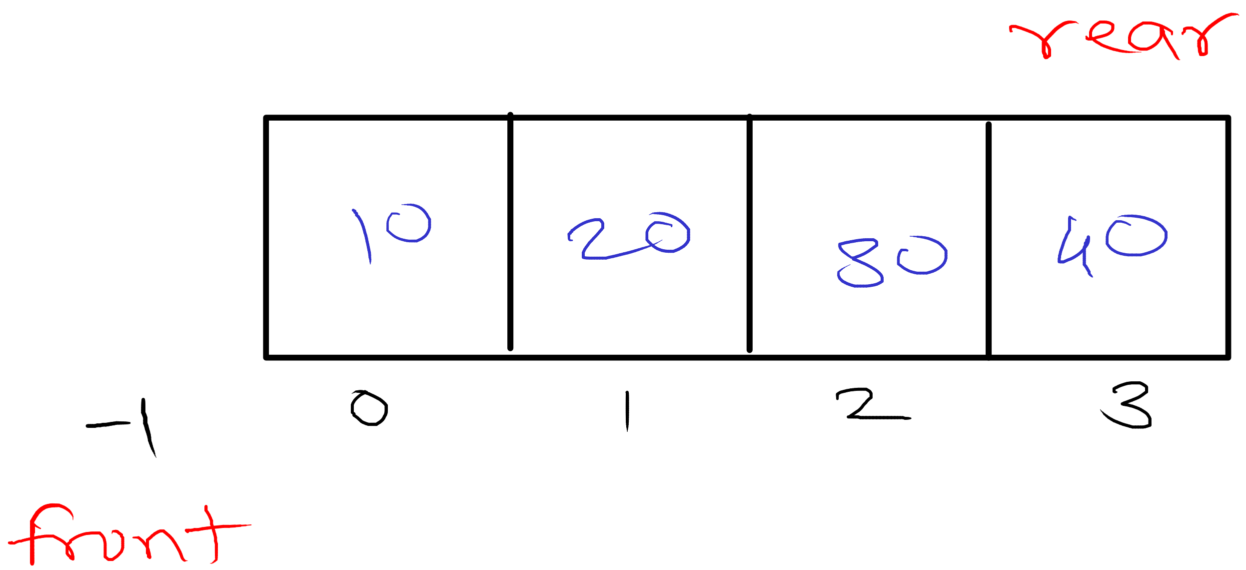
Circular Queue - Empty and Full Conditions



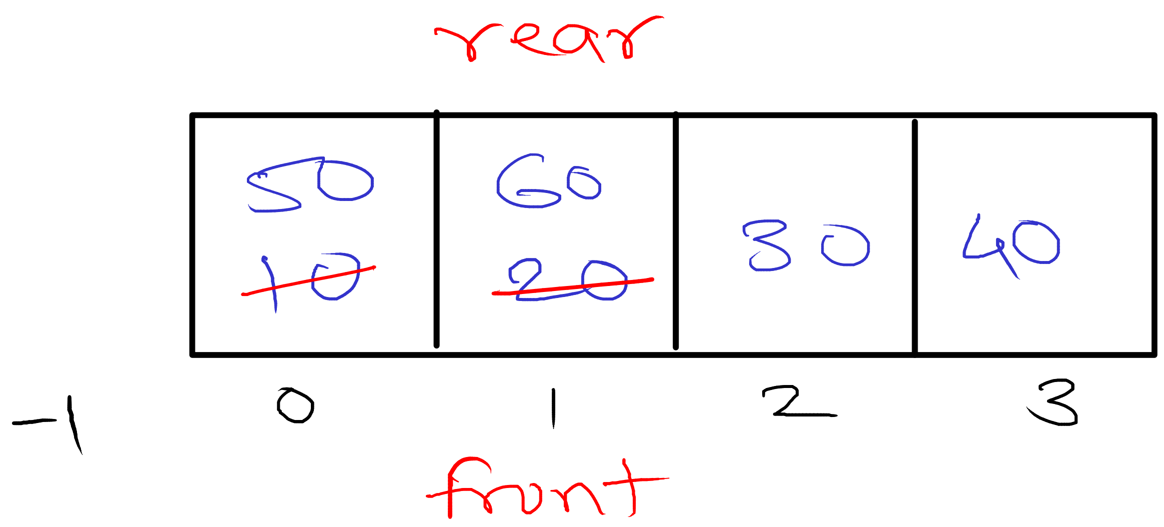
Empty : front == rear && rear == -1




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



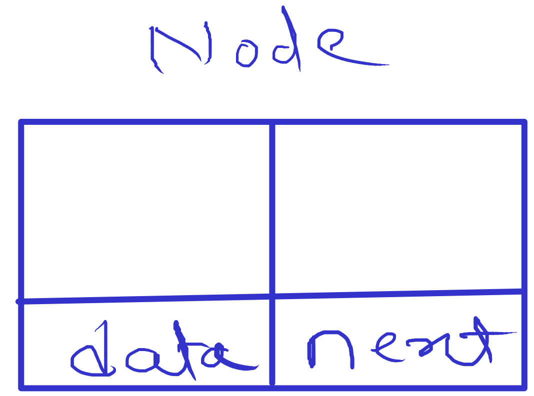
front == -1 && rear == SIZE-1



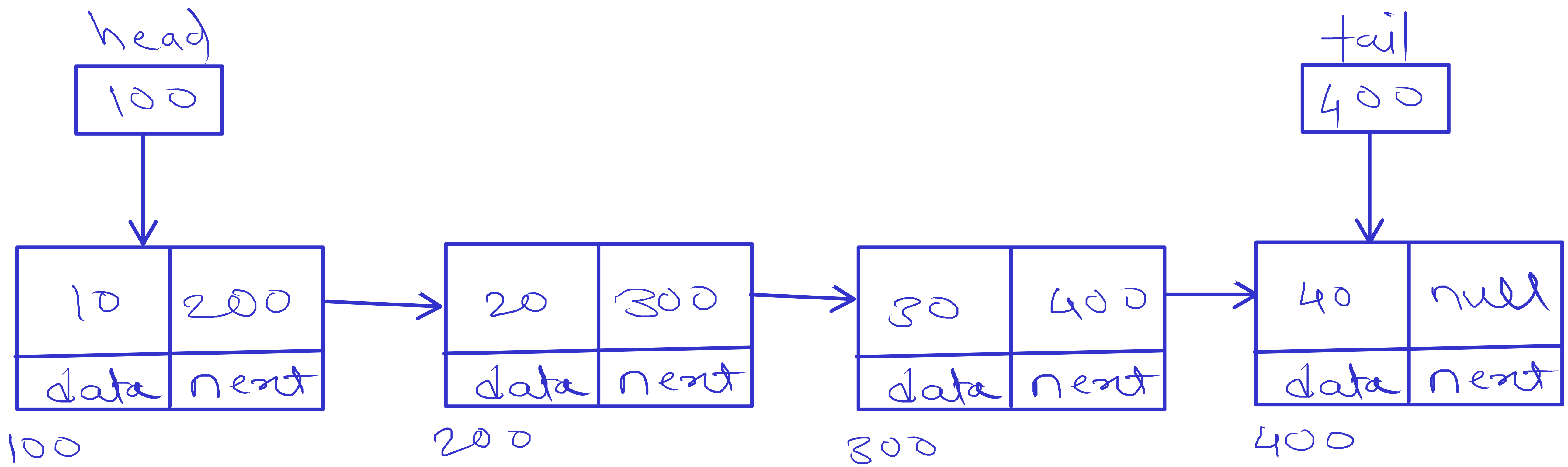
front == rear && rear != -1

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

Linked List



- linear data structure which stores simillar data
- address/link of next data is kept with current data
- linked elements are known as "Node"
- Node consist of two parts:
    - data
    - link/next
- 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 at first
2. Add at last
3. Add at in between (position)

4. Delete from first
5. Delete from last
6. Delete from in between (position)

7. Traverse (Display)

8. search
9. sort
10. reverse

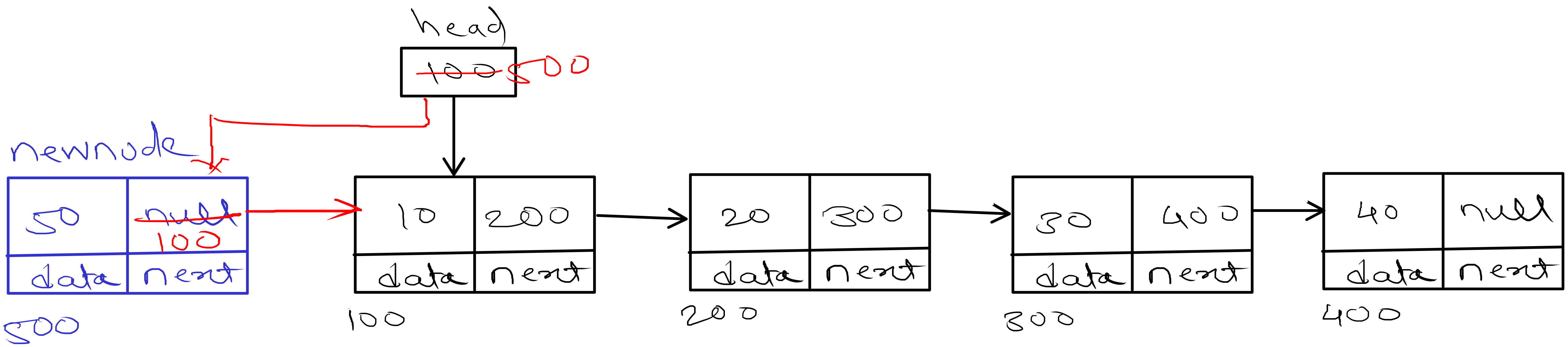
Linked List Types:

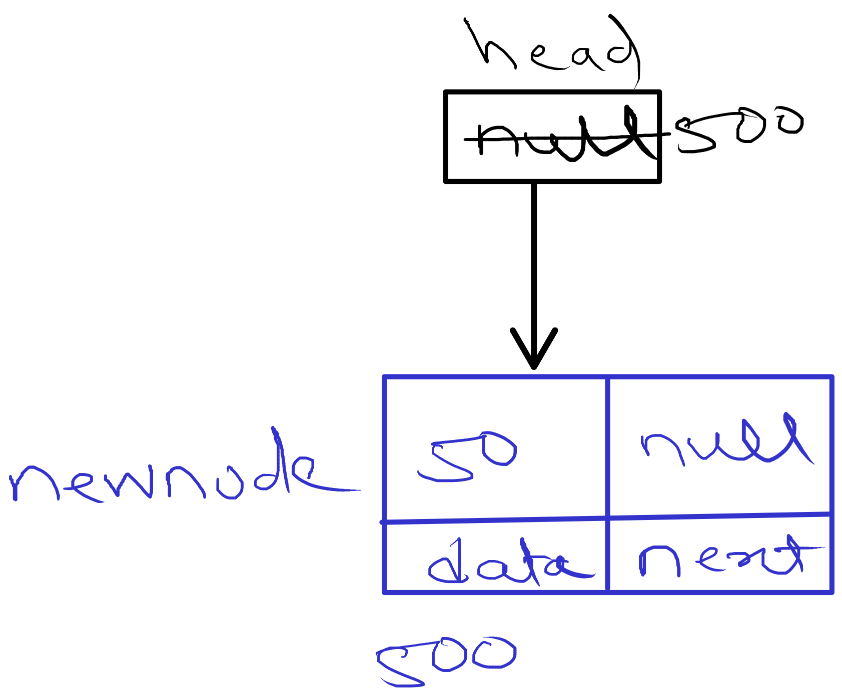
1. Singly linear linked list
2. Singly circular linked list
3. Doubly linear linked list
4. Doubly circular linked list

```
class List{
        static class Node{
                private int data;
                private Node next;
                public Node(){}
        }

        private Node head;
        private Node tail;
        public List(){}
        public isEmpty(){}
        public add(){}
        public delete(){}
        public display(){}
}
```

SLLL - Add First

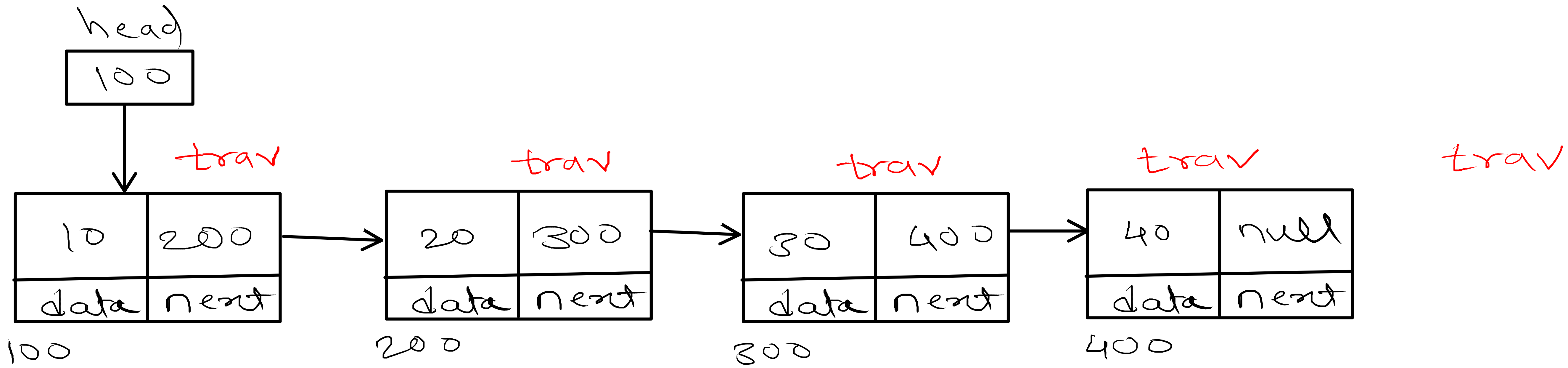




//1. create newnode with given value
//2. check if list is empty
//a. add newnode into head
//3. if list is node empty
//a. add first node into next of newnode
//b. add newnode into head

Time Complexity = O(1)

SLLL - Display



//1. create trav referance and start at head
//2. print(visit) current node
//3. go on next node
//4. repeat step 2 and 3 till last node

Time Complexity = O(n)