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)

Parenthesis Balancing

5 + ([9 - 4] * (8 - { 6 / 2 }))

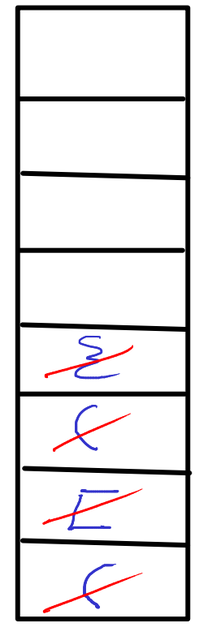
✓) == (

✓] == [

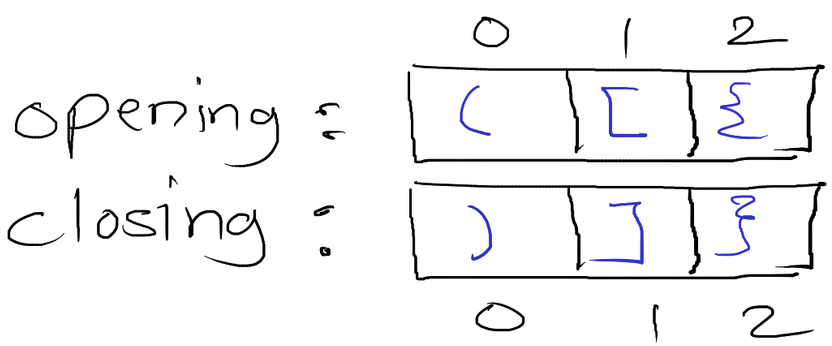
✓ } == {

✓] == [

Stack



5 + ([9 - 4] * (8 - { 6 / 2 }]))



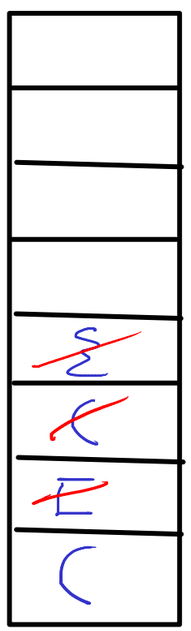
✗] ≠ (

✓ } == {

✓] == [

[== 1

Stack



Stack / Queue - Competitive Programming

Reverse array, string or linked list using stack/queue.

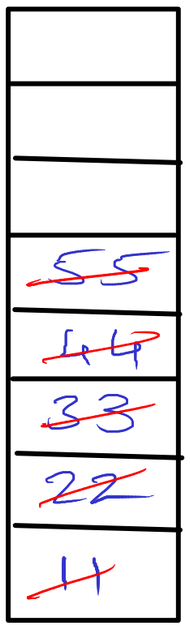
```
int [] arr = {11, 22, 33, 44, 55};
```

before arr =

11	22	33	44	55
0	1	2	3	4

after arr =

55	44	33	22	11
0	1	2	3	4




```
Stack<Integer> S = new Stack<>();
for(int i=0; i< arr.length; i++)
        S.push(arr[i]);
for(int i=0 ; i<arr.length; i++)
        arr[i] = S.pop();
```

Iterations

n

+ = 2n

n

Time Complexity:

$T \propto 2n$

$T \propto n$

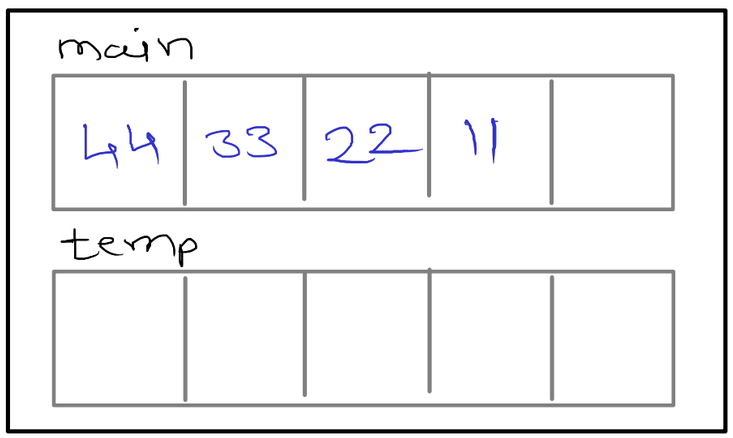
$O(n)$

Stack / Queue - Competitive Programming

Create stack using queue.

Hint:
1) Time complexity need not be O(1)
2) you can use more than one queue

Stack



Push order : 11, 22, 33, 44

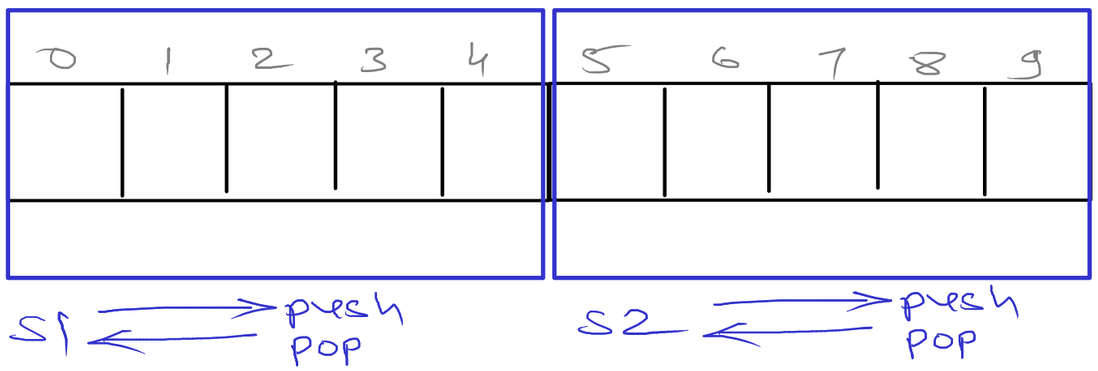
```
Push:
    while(!main.isEmpty())
        temp.push(main.pop())
    main.push(val);
    while(!temp.isEmpty())
        main.push(temp.pop());
Pop:
    main.pop();
Peek:
    main.peek()
```

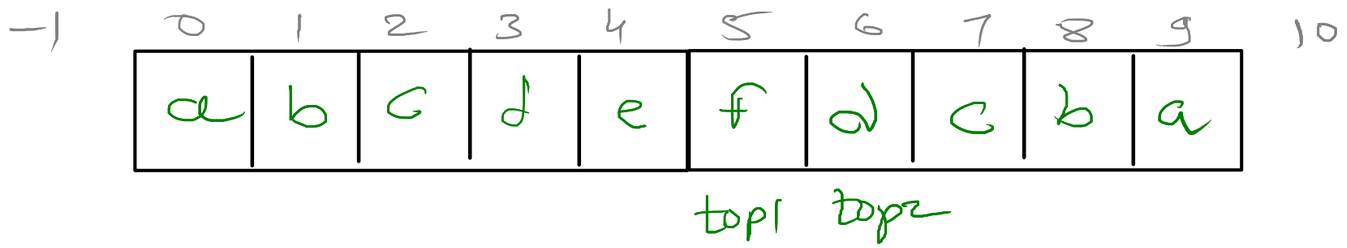
Create queue using stack.

↳ home work

Stack / Queue - Competitive Programming

How to implement two stacks in single array efficiently?





Stack1

init : top1 = -1;

Push : top1++;
arr[top1] = val;

pop : top1--;

Empty: top1 == -1;

Full : top1+1 == top2

Stack

init : top2 = arr.length

Push : top2--;
arr[top2] = val;

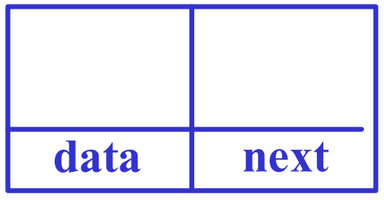
pop : top2++

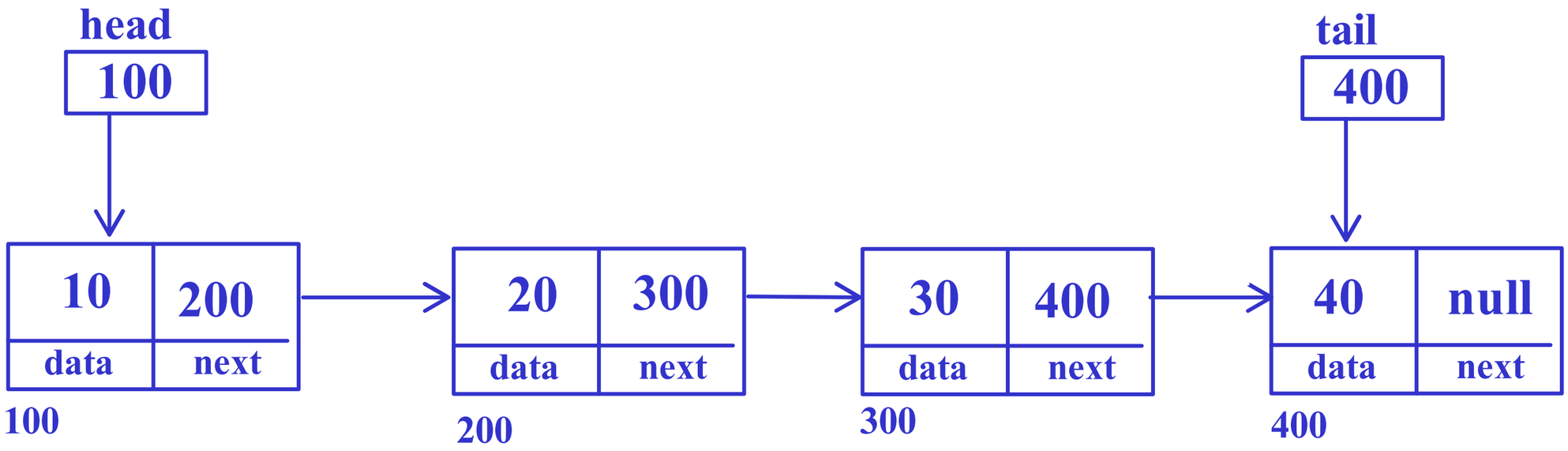
Empty: top2 == arr.length

Full : top1+1 == top2

Linked List

- linear data structure in which data is stored sequentially
- address of next data is kept with current data
- Every element of linked list is called as node
- Node consist of two parts
    - data - actual data
    - link/next - address (referance) of next node
- Address of first node is kept into one of the pointer (head)
- Address of last node is kept into one of the pointer (tail) - optional





Linked List

Operations:

1. Add node First
2. Add node Last
3. Add node at Position

4. Delete node First
5. Delete node Last
6. Delete node from position

7. Display (Traversal)
8. Search data
9. Sort data
10. Reverse display
11. Reverse list

Types:

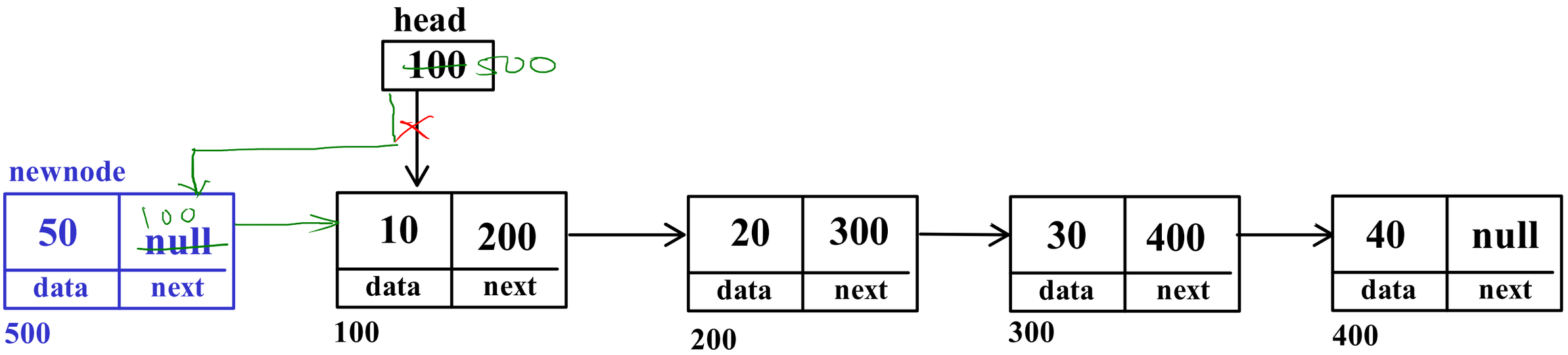
1. Singly Linear Linked List
2. Single Circular Linked List
3. Doubly Linear Linked List
4. Doubly Circular Linked List

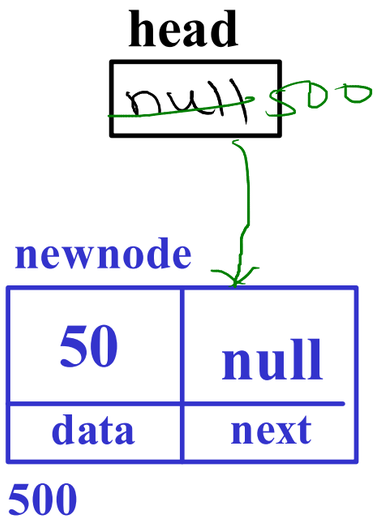
type* — int, char, double, class, enum

```
class list {
    static class node {        // self referential class
        type data;
        node next;
        public node()
        { }
    }
    node head;
    node tail;
    public list()
    { }
    public void add()
    { }
    public void delete()
    { }
}
```

there is no dependancy of outer class to create object of node class

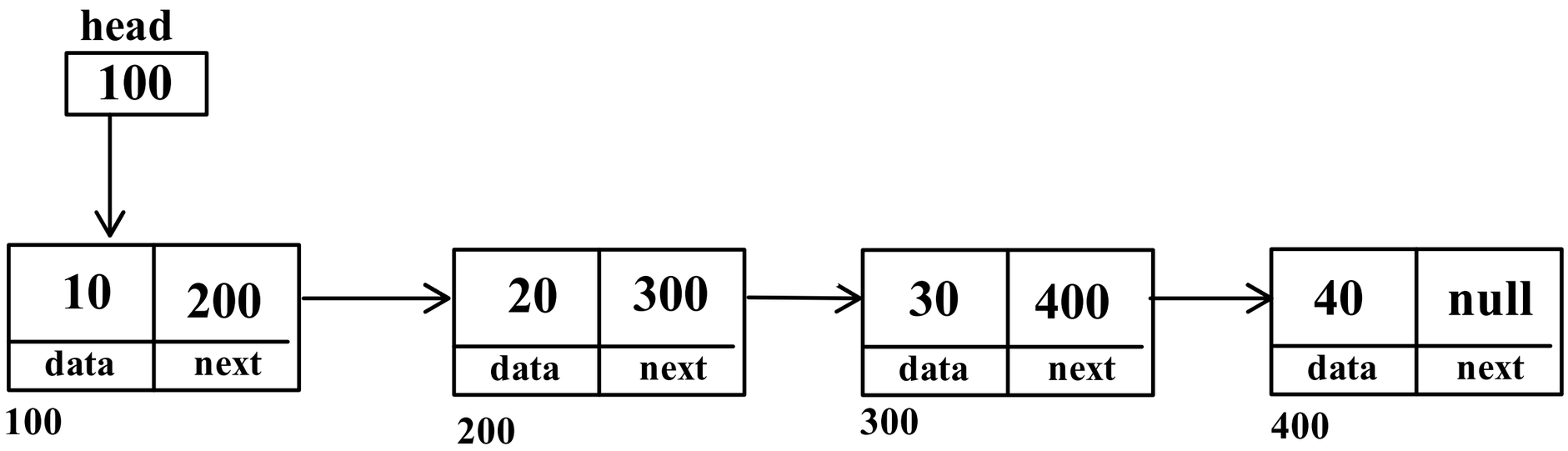
Singly Linear Linked List - Add First






```
//1. create node with given value
//2. if list is empty
      //a. add newnode into head itself
//3. if list is not empty
      //a. add first node into next of newnode
      //b. move head on newnode
```

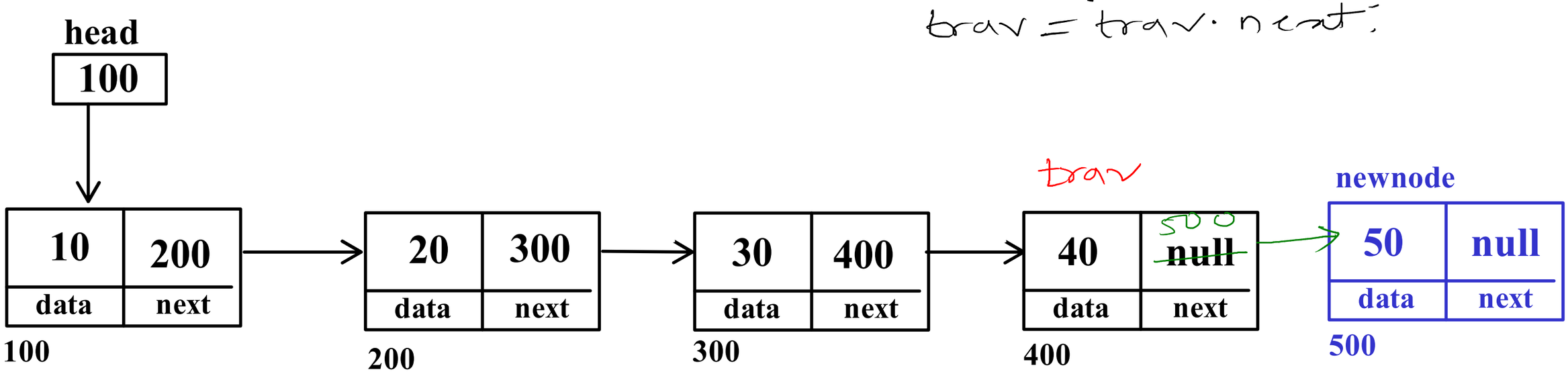
Singly Linear Linked List - Display (Traverse)

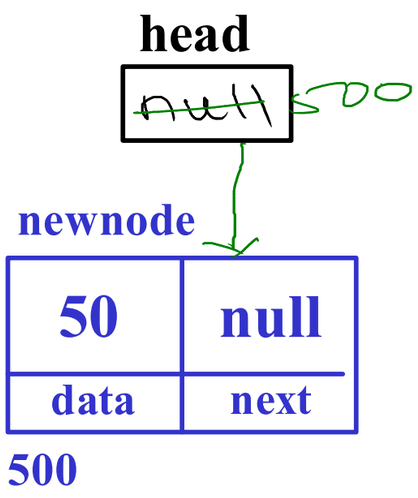


//1. create trav referance and start pointing it on head
//2. print data of current node (trav)
//3. go on next node
//4. repeat step 2 and 3 till last node

Singly Linear Linked List - Add Last

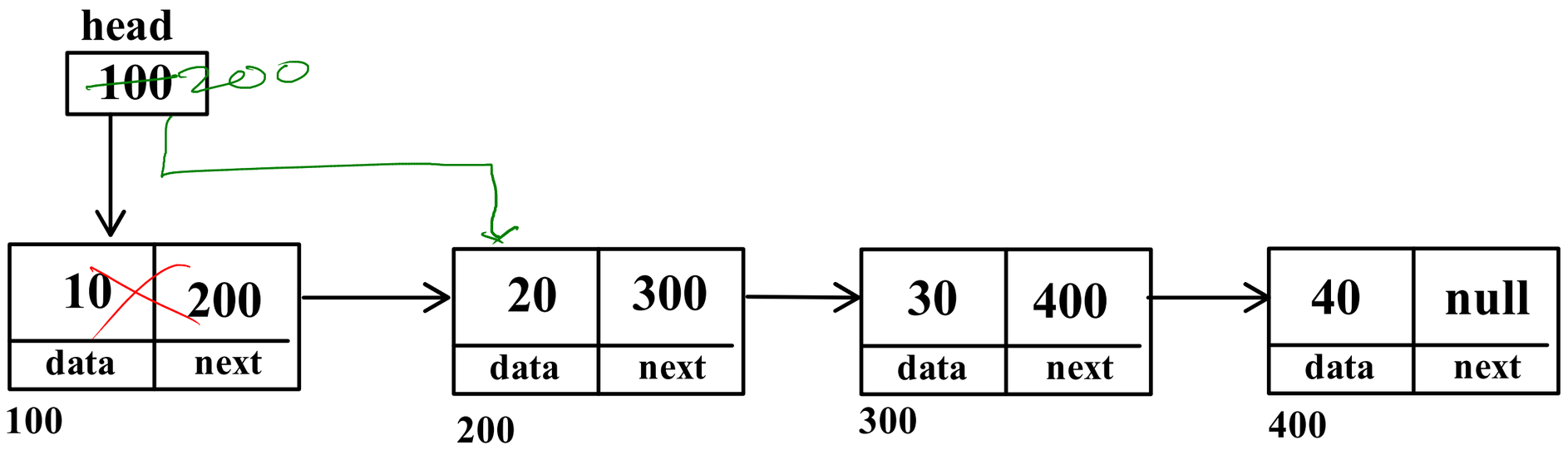
while(trav.next != null)
trav = trav.next;

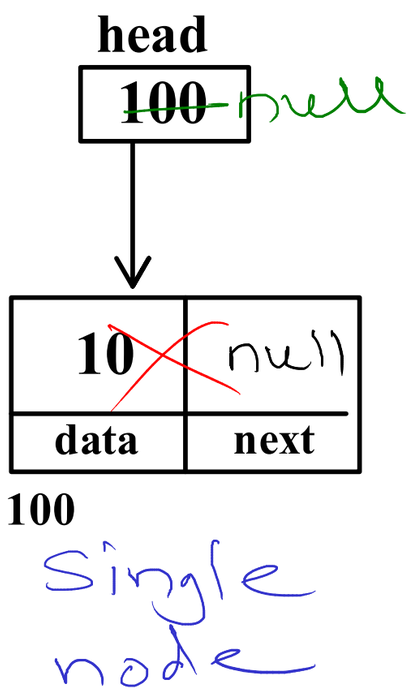




//1. create node with given value
//2. if list is empty
//a. add newnode into head
//3. if list is not empty
//a. traverse till last node (trav = last node)
//b. add newnode into next of last node

Singly Linear Linked List - Delete First

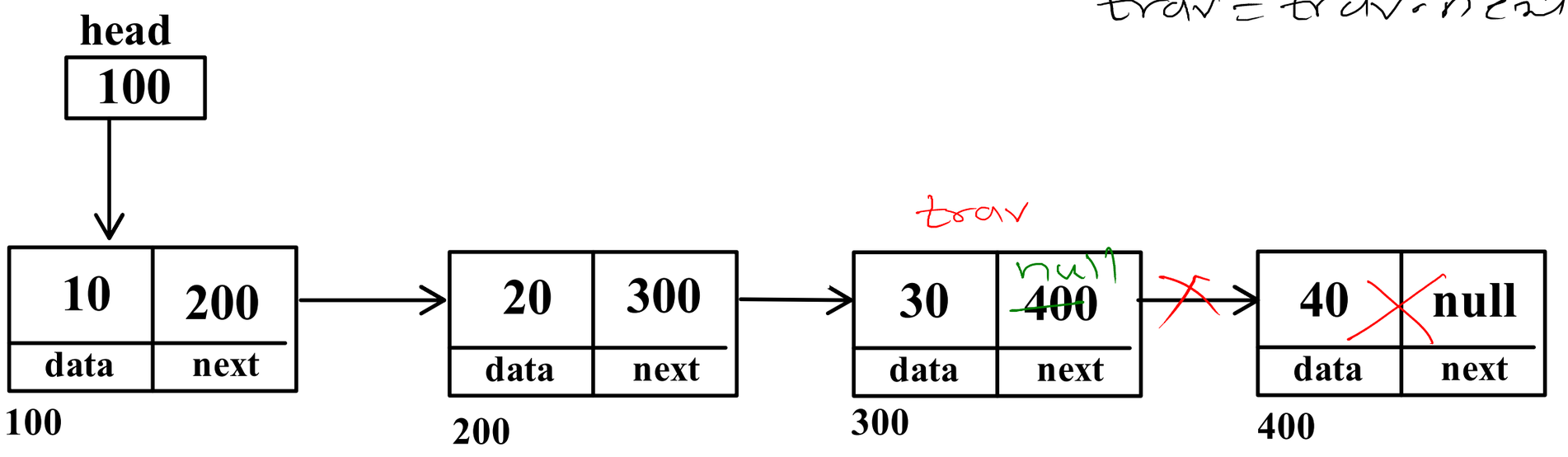


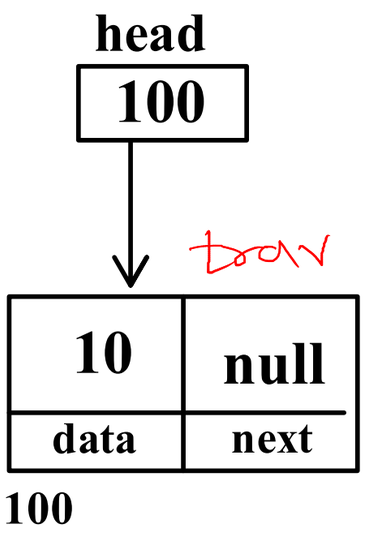



```
//1. if list is empty
        // do nothing
//2. if list is not empty
        //a. move head on second node
        head = head.next
```

Singly Linear Linked List - Delete Last

while(trav.next.next != null)
trav = trav.next





single node

```
//1. if list is empty
      //do nothing
//2. if list has single node
      //a. make head equal to null
//3. if list has multiple nodes
      //a. traverse till second last node
      //b. make next of second last node equal to null
```