Sunbeam Institute of Information Technology
Pune and Karad

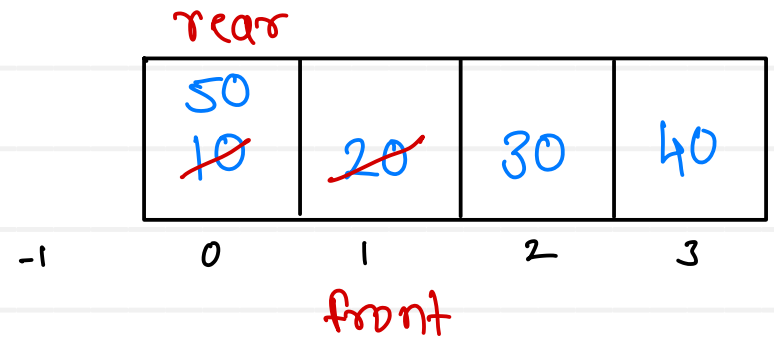
Module – Data Structures and Algorithms

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Circular queue

size = 4



front = (front + 1) % size

rear = (rear + 1) % size

front = rear = -1

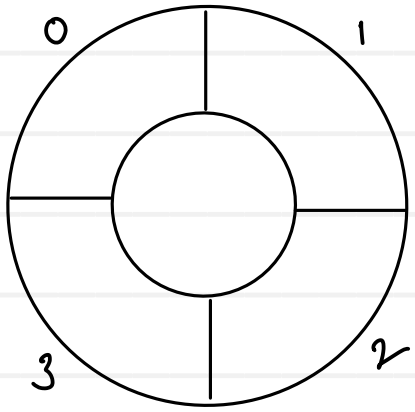
= (-1+1) % 4 = 0

= (0+1) % 4 = 1

= (1+1) % 4 = 2

= (2+1) % 4 = 3

= (3+1) % 4 = 0



Operations:

1) Push / Enqueue:
   - reposition rear (inc)
   - add value at rear index

2) Pop / Dequeue:
   - reposition front (dec)

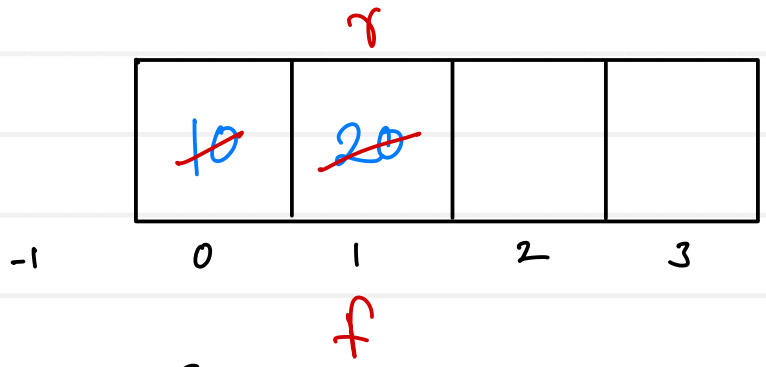
3) Peek:
   - read value of front+1 index

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

Circular queue - Conditions

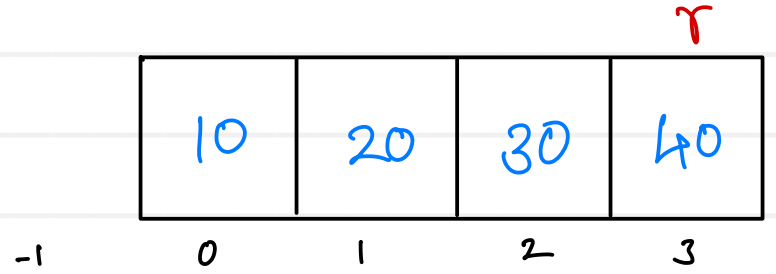
Empty

front == rear && rear == -1

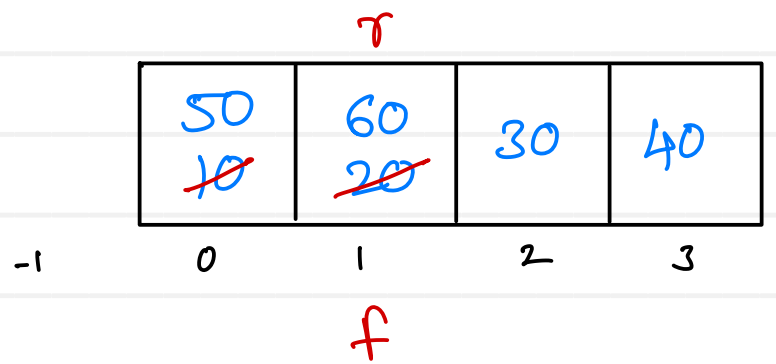



```
pop() {
    front = (front+1) % size;
    if(front == rear)
        front = rear = -1
}
```

Full



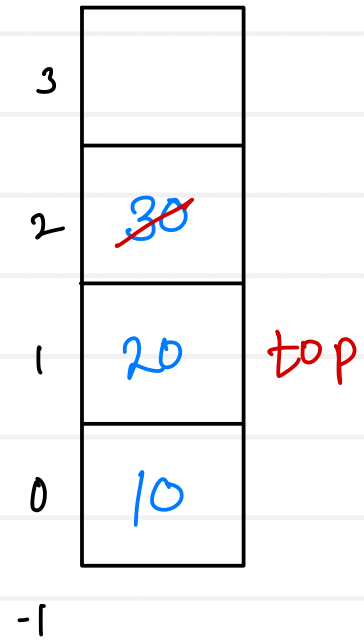
front == -1 && rear == size-1



front == rear && rear != -1

Stack

- stack is linear data structure which stores similar type of data.
- stack has only one end (top)
- insert & delete both are allowed only from top end



- stack works on principle of "LIFO" / "Last In First Out".
- top always points to last inserted data.

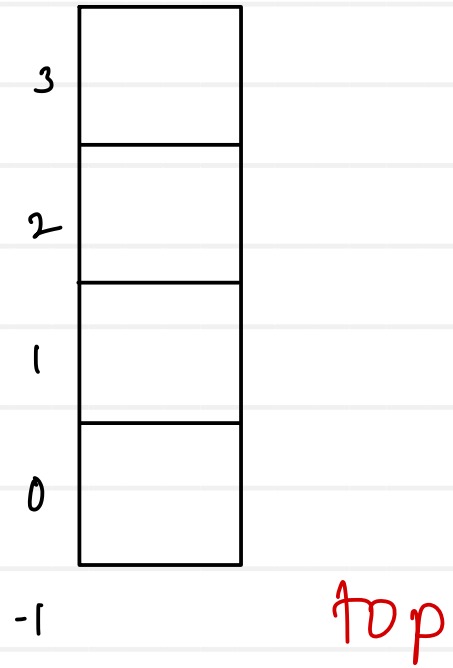
Operations :

1) Add / push :
   1) reposition top (inc)
   2) add value at top index
2) Delete / pop :
   1) reposition top (dec)
3) Peek :
   1) read / return data of top index

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

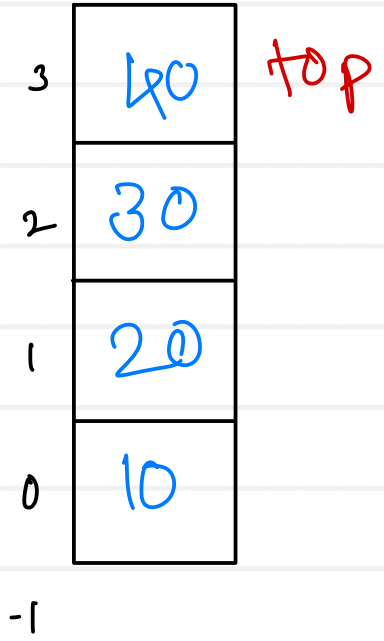
Stack - Conditions

Empty



top == -1

Full



top == size-1

Applications – Stack and Queue

Stack

- Parenthesis balancing
- Expression conversion and evaluation
- Function calls
- Used in advanced data structures for traversing

- **Expression conversion and evaluation:**
  - Infix to postfix
  - Infix to prefix
  - Postfix evaluation
  - Prefix evaluation

Expressions :

1) Infix : a+b (human)

2) Postfix : a b + } (machine)

3) Prefix : + a b }

Queue

- Jobs submitted to printer
- In Network setups – file access of file server machine is given to First come First serve basis
- Calls are placed on a queue when all operators are busy
- Used in advanced data structures to give efficiency.
- Process waiting queues in OS

Postfix Evaluation

- Process each element of postfix expression from left to right
- If element is operand
  - Push it on a stack
- If element is operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Perform current element (Operator) operation between Op1 and Op2
  - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. 4 5 6 * 3 / + 9 + 7 -

Postfix evaluation

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

l ⟶ r

Result : 16

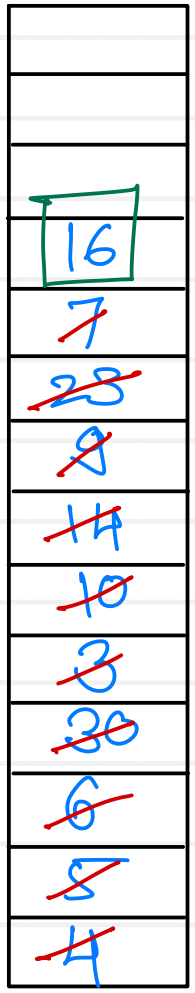
⑤ 23 - 7 = 16

④ 14 + 9 = 23

③ 4 + 10 = 14

② 30 / 3 = 10

① 5 * 6 = 30



Prefix Evaluation

- Process each element of prefix expression from right to left
- If element is operand
  - Push it on a stack
- If element is operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op1 – first popped element
    - Op2 – second popped element
  - Perform current element (Operator) operation between Op1 and Op2
  - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. - + + 4 / * 5 6 3 9 7

Prefix evaluation

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

l ← r

Result : 16

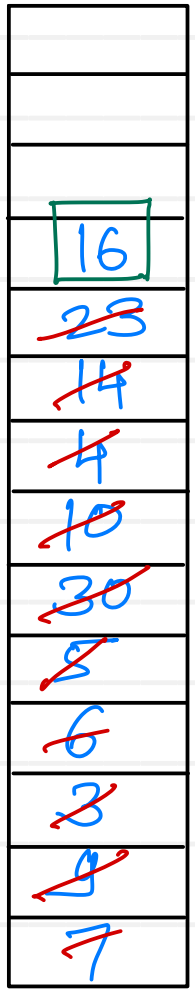
⑤ 23 − 7 = 16

④ 14 + 9 = 23

③ 4 + 10 = 14

② 30 / 3 = 10

① 5 * 6 = 30



Infix to Postfix Conversion

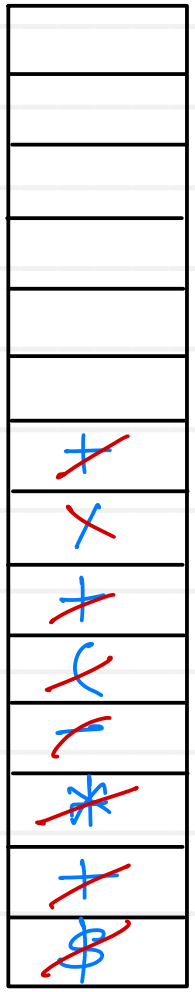
- Process each element of infix expression from left to right
- If element is Operand
  - Append it to the postfix expression
- If element is Operator
  - If priority of topmost element (Operator) of stack is greater or equal to current element (Operator), pop topmost element from stack and append it to postfix expression
  - Repeat above step if required
  - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the postfix expression
- e.g. a * b / c * d + e – f * h + i

Infix to Postfix conversion

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

l ⟶ r

Postfix expression : 19$34*+682/+-7+



Priority :
( )
$
* / %
+ -

Infix to Prefix Conversion

- Process each element of infix expression from right to left
- If element is Operand
  - Append it to the prefix expression
- If element is Operator
  - If priority of topmost element of stack is greater than current element (Operator), pop topmost element from stack and append it to prefix expression
  - Repeat above step if required
  - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the prefix expression
- Reverse prefix expression
- e.g. a * b / c * d + e – f * h + i

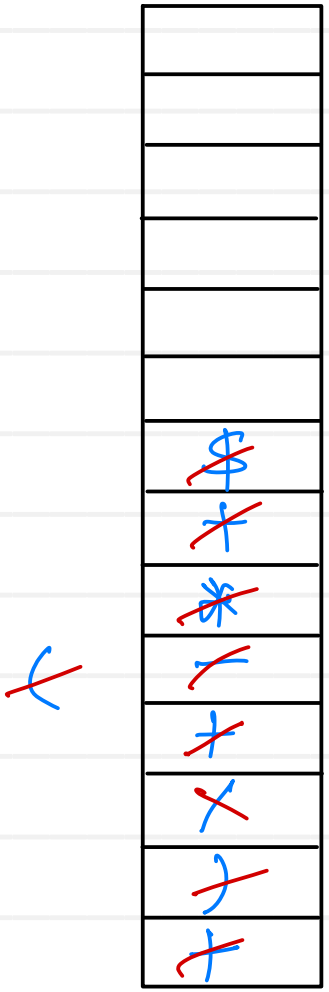
Infix to Prefix conversion

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

l ← r

Expression : 728/6+43*91$+-+

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



Prefix to Postfix

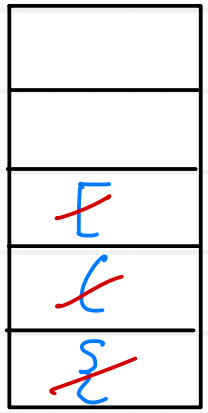
- Process each element of prefix expression from right to left
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op1 – first popped element
    - Op2 – second popped element
  - Form a string by concatenating Op1, Op2 and Opr (element)
  - String = “Op1+Op2+Opr”, push back on to the stack
- Repeat above two steps until end of prefix expression.
- Last remaining on the stack is postfix expression
- e.g. * + a b – c d

Postfix to Infix

- Process each element of postfix expression from left to right
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Form a string by concatenating Op1, Opr (element) and Op2
  - String = “Op1+Opr+Op2”, push back on to the stack
- Repeat above two steps until end of postfix expression.
- Last remaining on the stack is infix expression
- E.g. a b c - + d e – f g – h + / *

Parenthesis balancing using stack

{ ([ ]) }

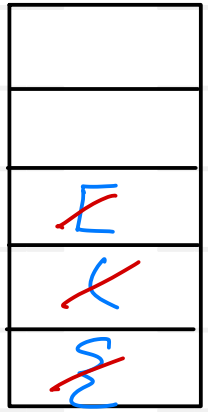


] == [

) == (

} == {

{ ([ ]) })



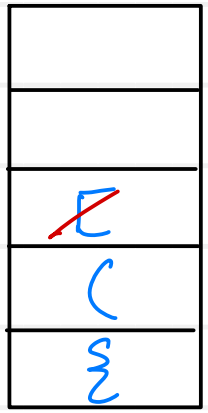
] == [

) == (

} == {

)

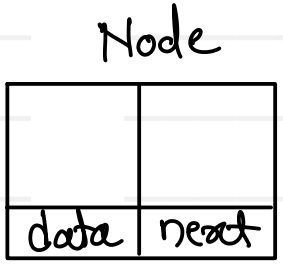
{ ([ )]



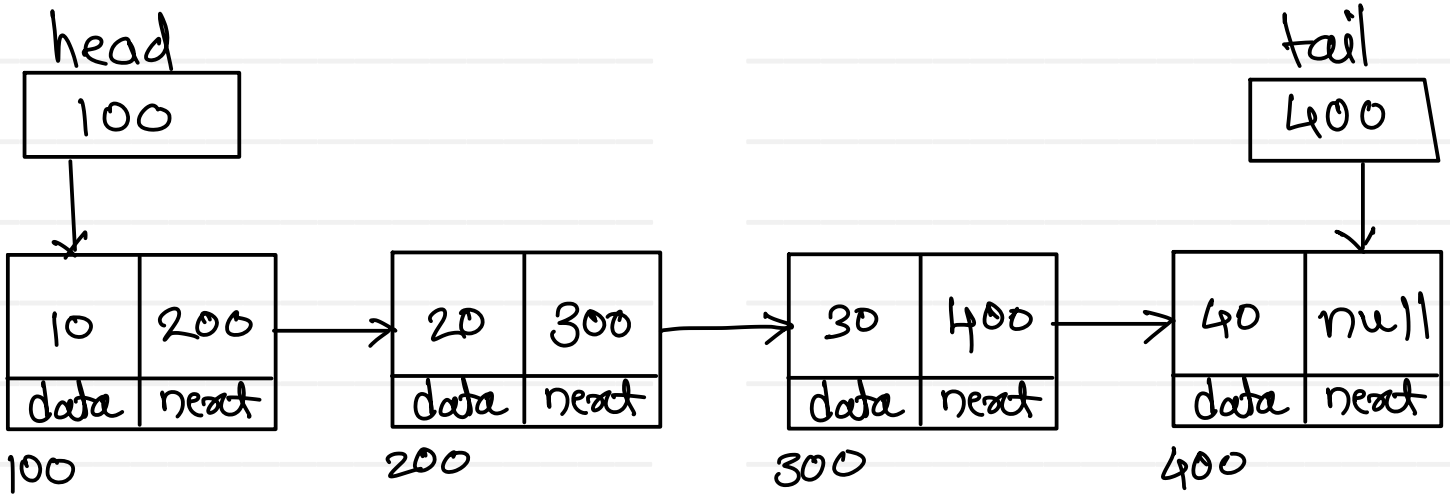
) != [

Linked List

- linear data structure
- Address / link of next data is kept with current data
- every element is known as "Node"
- every node has two parts:
  1) data : actual value
  2) link/next : add^{r}/referance of next data



- address / reference of first node is kept into head reference
- address / reference of last node is kept into tail reference (optional)



Linked List

Operations

1. Add first
2. Add last
3. Add position (insert)

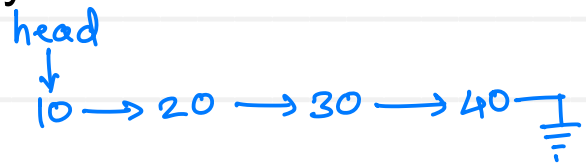
1. Delete first
2. Delete last
3. Delete position

1. Display (traverse) (forward/ backward)

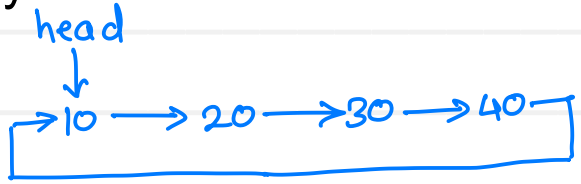
1. Search
2. Sort
3. Reverse

Types

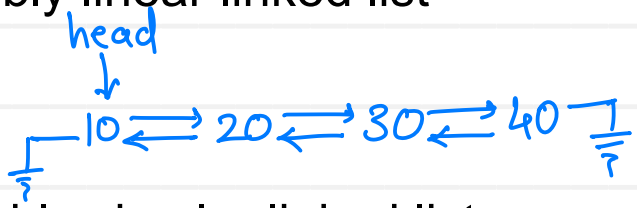
1. Singly linear linked list



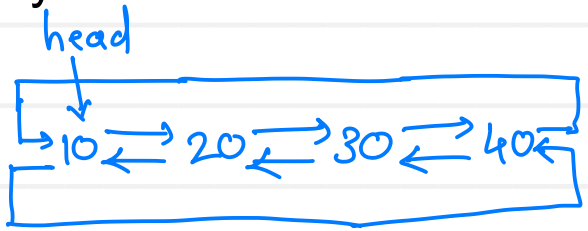
2. Singly circular linked list



3. Doubly linear linked list



4. Doubly circular linked list



Linked List

Node has two parts

data : int, char, float, double, enum, class, string

next : reference

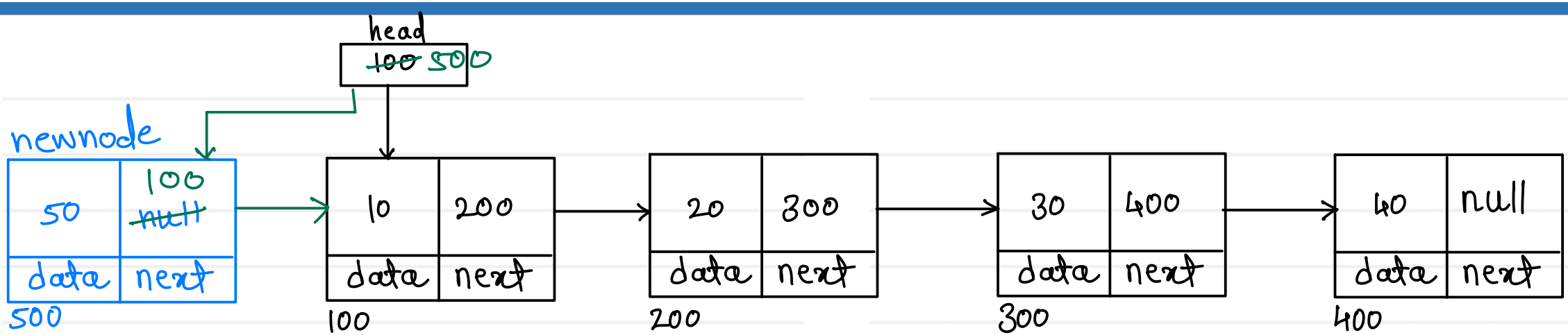
```
class Node {
    int data;
    Node next;
}
```

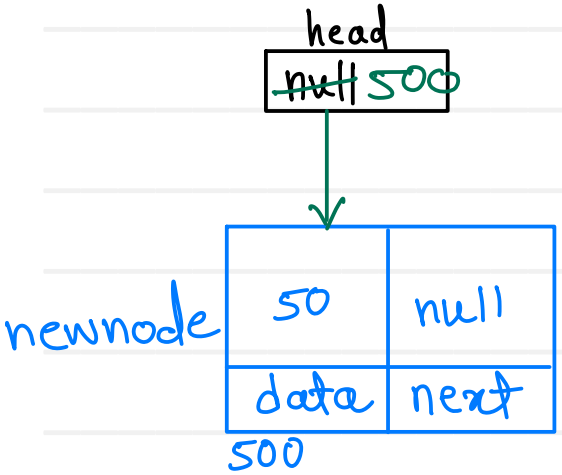
Why static? → ① no dependency of List class to create object of Node class

② private non static fields of List class should not be accessed into Node class

```
class List {
    static class Node {
        int data;
        Node next;
    }
    Node head, tail;
    int size;
    List() { ... }
    add() { ... }
    delete() { ... }
    display() { ... }
    deleteAll() { ... }
}
```

Singly linear Linked List - Add first





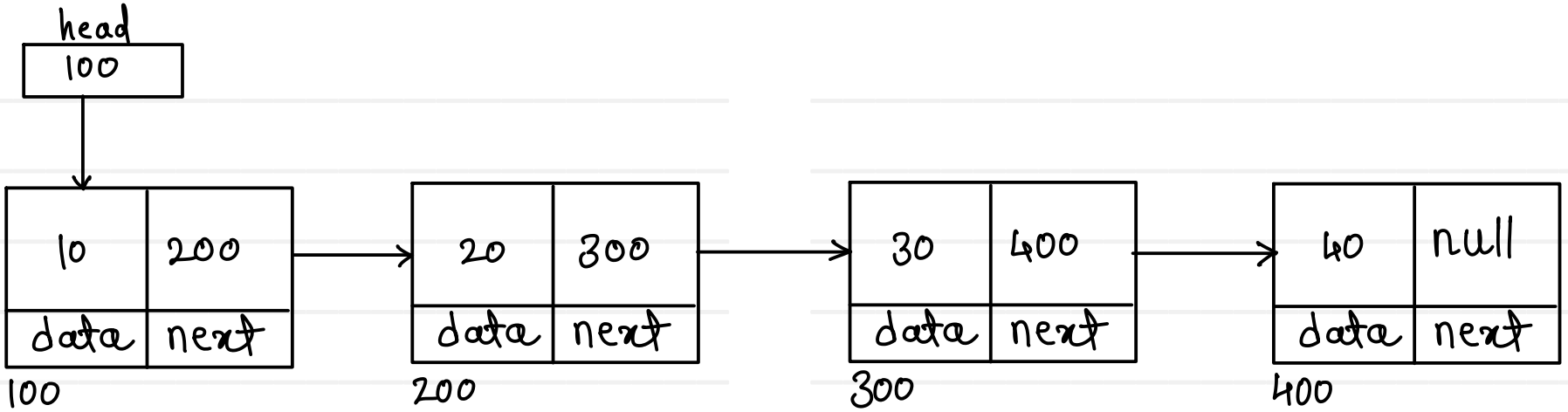
1. Create Node with given data/value.
2. add first node into next of newnode
3. Move head on newnode

newnode.next = head

head = newnode

$T(n) = O(1)$

Singly linear Linked List - Display

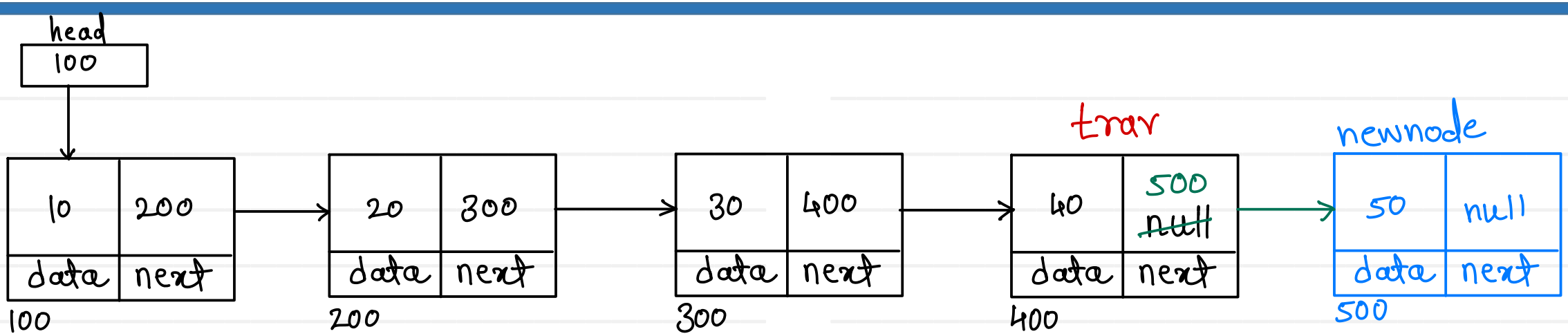


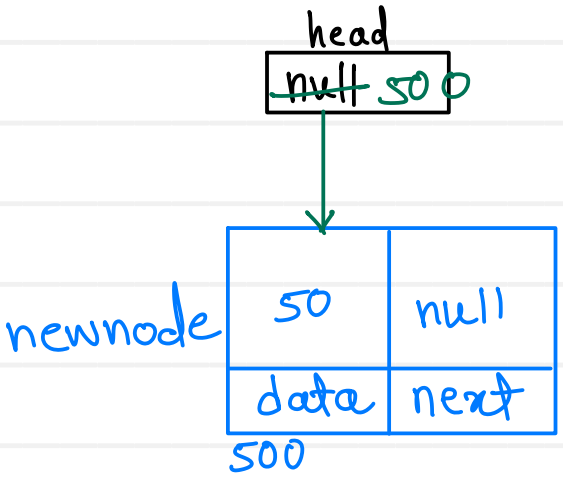
trav	trav.data	trav.next
100	10	200
200	20	300
300	30	400
400	40	null
null		

1. Create trav & start at head.
2. print/visit current node (trav.data)
3. go on next node (trav.next)
4. repeat above two steps for each node

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

Singly linear Linked List - Add last



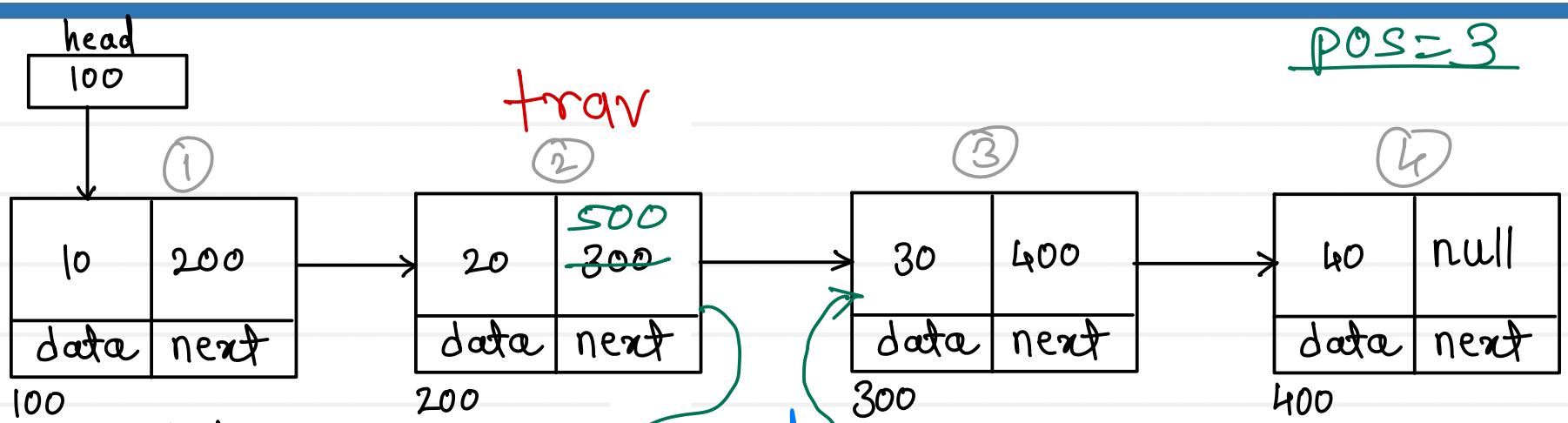



```
while(trav.next != null)
    trav = trav.next;
```

1. Create Node with value
2. if list is empty
   a. add newnode into head
3. if list is not empty
   a. traverse till last node
   b. add newnode into next of lastnode

$T(n) = O(n)$

Singly linear Linked List - Add position



1. Create node with data
2. if list is empty
   head = newnode;
3. if list is not empty
   a. traverse till pos-1 node
   b. add pos node into next of newnode
   c. add newnode into next of pos-1 node

T(n) = O(n)

```
Node trav = head;
for(int i=1 ; i<pos-1 ; i++)
    trav = trav.next;
```

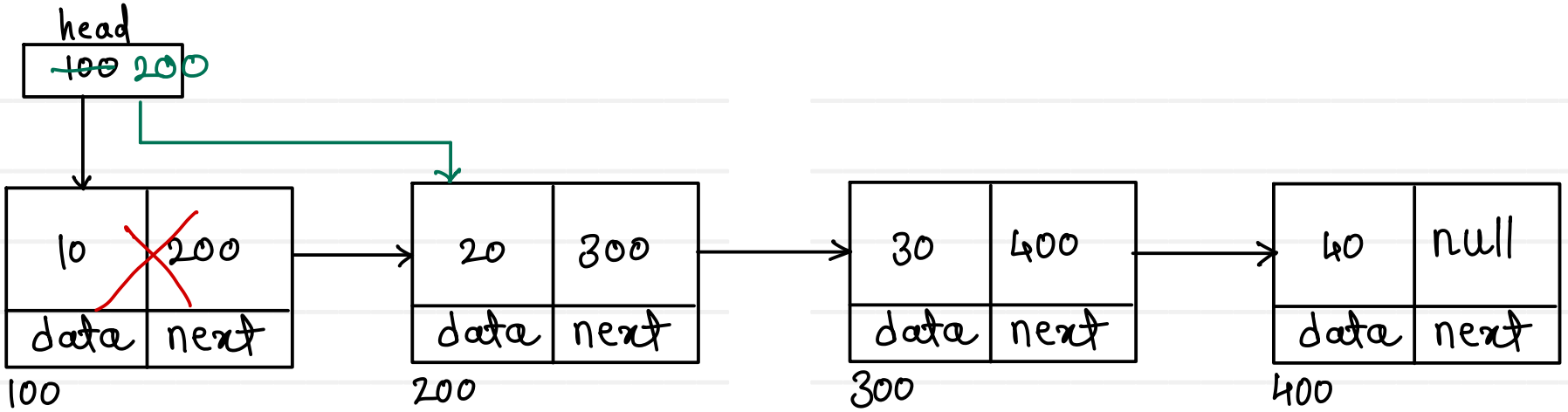
pos=3

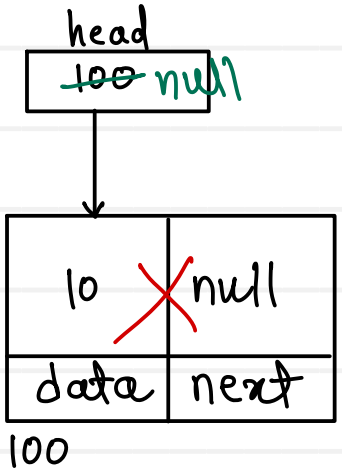
trav	i	i<2
100	1	T
200	2	F

pos=5

trav	i	i<4
100	1	T
200	2	T
300	3	T
400	4	F

Singly linear Linked List - Delete first

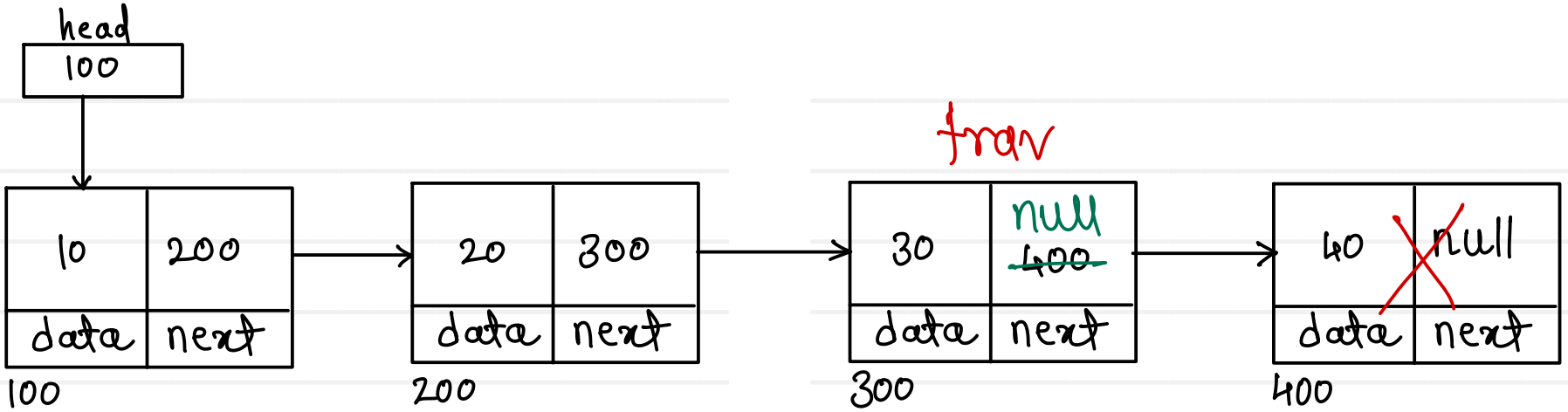




1. if list is empty
   return ;
2. if list is not empty
   move head on second node

$T(n) = O(1)$

Singly linear Linked List - Delete last

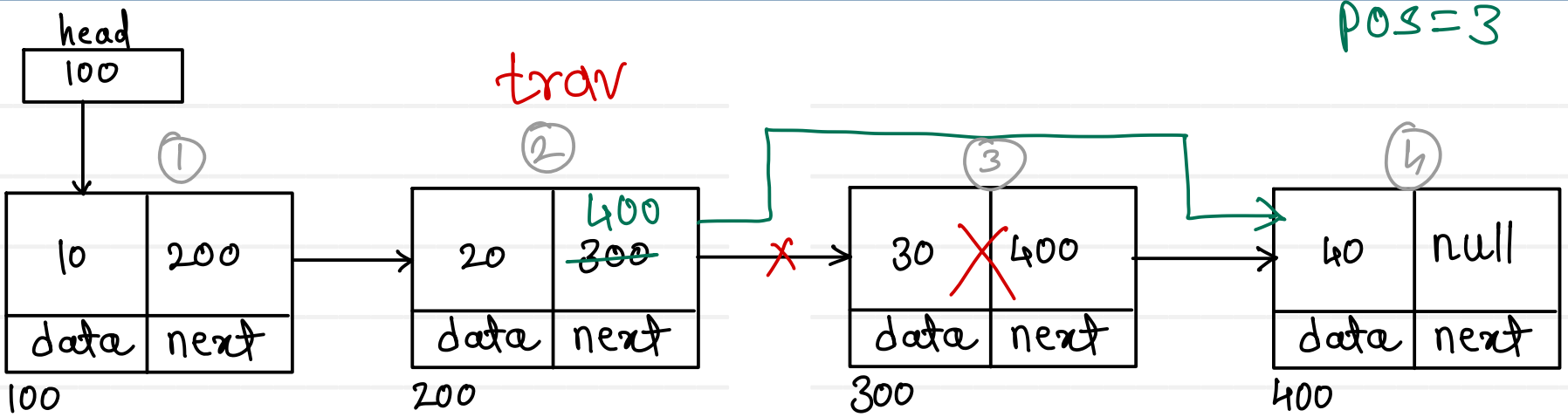



```
while (trav.next.next != null)
    trav = trav.next;
```

1. if list is empty
   return;
2. if list has single node
   head = null;
3. if list has multiple nodes
   a. traverse till second last node
   b. add null into next of second last node

$T(n) = O(n)$

Singly linear Linked List - Delete position



1. if list is empty
   return;
2. if list has single node
   head = null;
3. if list has multiple nodes
   a. traverse till pos-1 node
   b. add pos+1 node into next of pos-1 node

$T(n) = O(n)$

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com