Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Space complexity

- Finding approximate space requirement of the algorithm to execute inside memory

Total space = Input space + Auxiliary space

Input space: space required to store data

Auxiliary space: extra space required to process input

Linear search:

```
int linearSearch(int arr[], int key)
{
    for(int i=0; i<arr.length; i++)
        if(key == arr[i])
            return i;
    return -1;
}
```

input variable : arr

processing variable : i, key, size

- Auxilliary space is constant here

$S(n) = O(1)$

Algorithm analysis

Iterative

- loops are used

```
int fact(int num) {
    int f=1;
    for(int i=1; i<=num; i++)
        f *= i;
    return f;
}
```

Time ∝ no. of iterations of the loop

Time ∝ n

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

$$S(n) = O(1)$$

Recursive

- recursion is used

```
int rfact(int num) {
    if(num==1)
        return 1;
    return num * rfact(num-1);
}
```

Time ∝ no. of recursive call

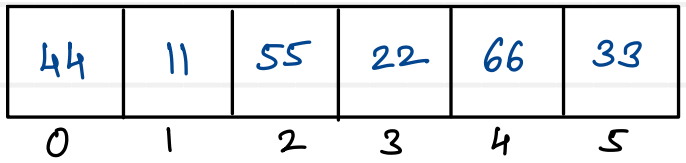
Time ∝ n

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

$$S(n) = O(n)$$

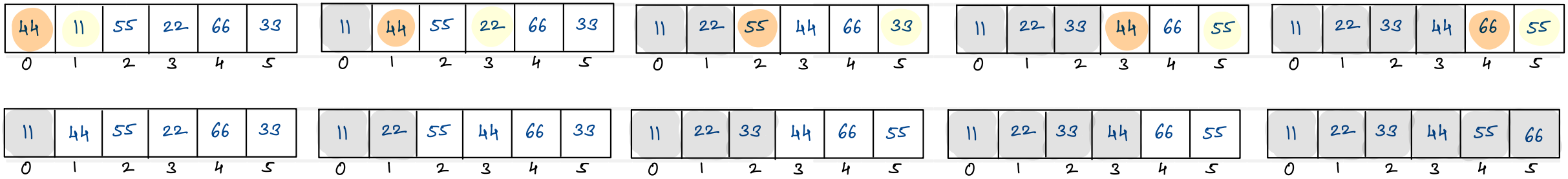
Selection sort

1. Select one position of the array
2. Find smallest element out of remaining elements
3. Swap selected position element and smallest element
4. Repeat above steps until array is sorted



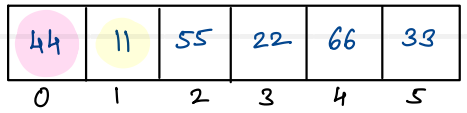
i minIndex j
3 3 4 5 6

Pass 1 Pass 2 Pass 3 Pass 4 Pass 5



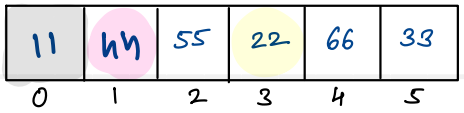
to select the position : i : 0 → N-2 (i < N-1)
to find smallest element : j : i+1 → N-1 (j < N)

Selection sort



i	minIndex	j
0	0	1
	1	2
		3
		4
		5
		6

```
minIndex = i
for(j=i+1; j<N; j++)
    if(arr[j] < arr[minIndex])
        minIndex = j;
```



i	minIndex	j
1	1	2
	3	3
		4
		5
		6

mathematical polynomial : n^2+n

Degree : highest power of variable

- while writing time complexities, consider only degree term, because it is highest growing term in that polynomial

n	n^2
1	1
10	100
100	10000
1000	1000000

$n \rightarrow$ no. of elements

$n-1 \rightarrow$ no. of passes

pass	comps
1	$n-1$ (n)
2	$n-2$
⋮	⋮
$n-2$	2
$n-1$	1

Total comps $= 1+2+3+\cdots\cdots+n$

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

Time $\propto \frac{1}{2}(n^2+n)$

$T(n) = O(n^2)$ Best Avg Worst

$S(n) = O(1)$

Bubble sort

1. Compare all pairs of consecutive elements of the array one by one
2. If left element is greater than right element , then swap both
3. Repeat above steps until array is sorted

No. of elements = n
No. of passes = n-1

pass	comps
1	n-1
2	n-2
3	:
4	2
5	1

Best case :

11 22 33 44 55 66
11 22 33 44 55 66
11 22 33 44 55 66
11 22 33 44 55 66
11 22 33 44 55 66

No. of comps = n-1

Time $\propto$ n-1

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

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

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

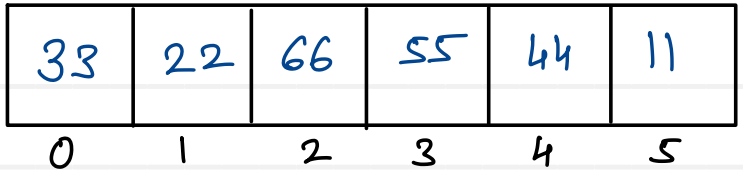
Time $\propto \frac{1}{2}(n^2+n)$

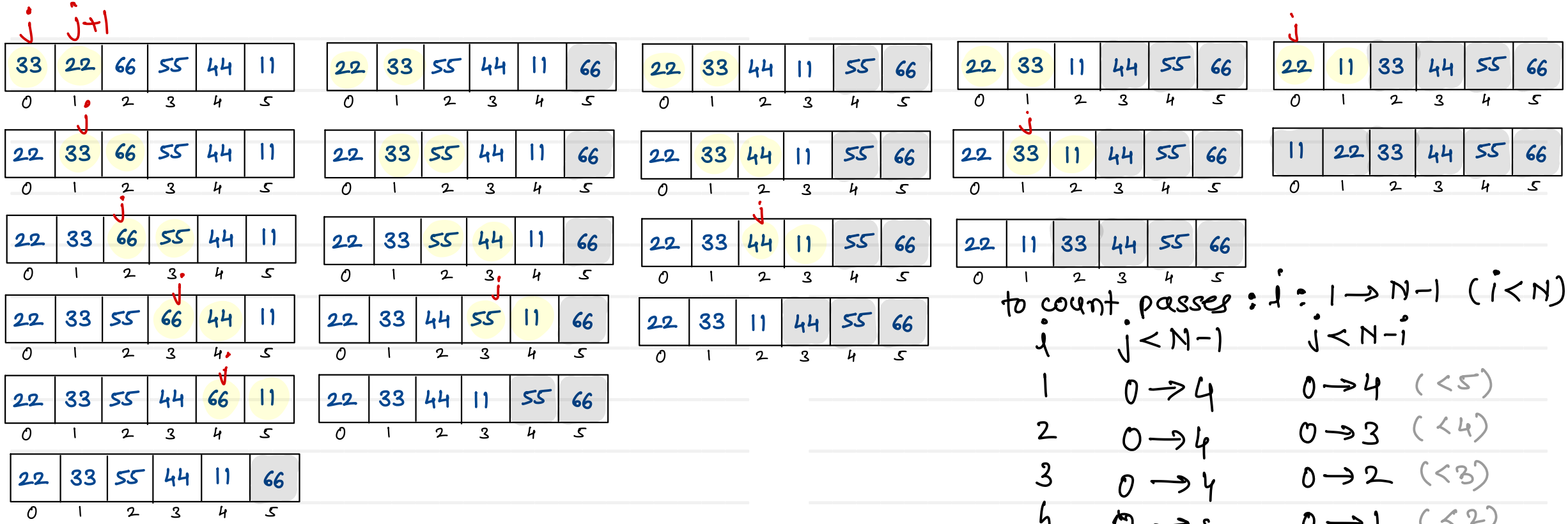
$T(n) = O(n^2)$ Avg Worst

auxiliary space is constant

$S(n) = O(1)$

Bubble sort



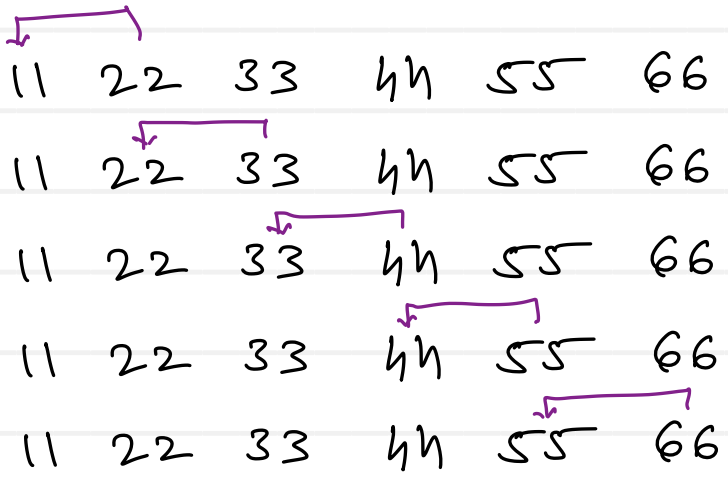


to count passes : i : 1 → N-1 (i<N)

i	j<N-1	j<N-i
1	0→4	0→4 (<5)
2	0→4	0→3 (<4)
3	0→4	0→2 (<3)
4	0→4	0→1 (<2)
5	0→4	0→0 (<1)

Insertion sort

1. Pick one element of the array (start from 2nd index)
2. Compare picked element with all its left neighbours one by one
3. If left neighbour is greater, move it one position ahead
4. Insert picked element at its appropriate position
5. Repeat above steps until array is sorted



No. of comps = n - 1

Time ∝ n - 1

$T(n) = O(n)$

No. of elements = n

No. of passes = n - 1

passes	comps
1	1
2	2
3	3
⋮	
n-1	n-1

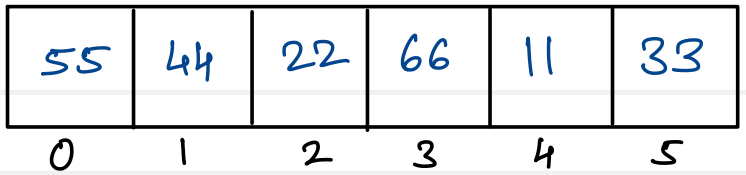
Total comps = $1+2+3+\ldots+n$

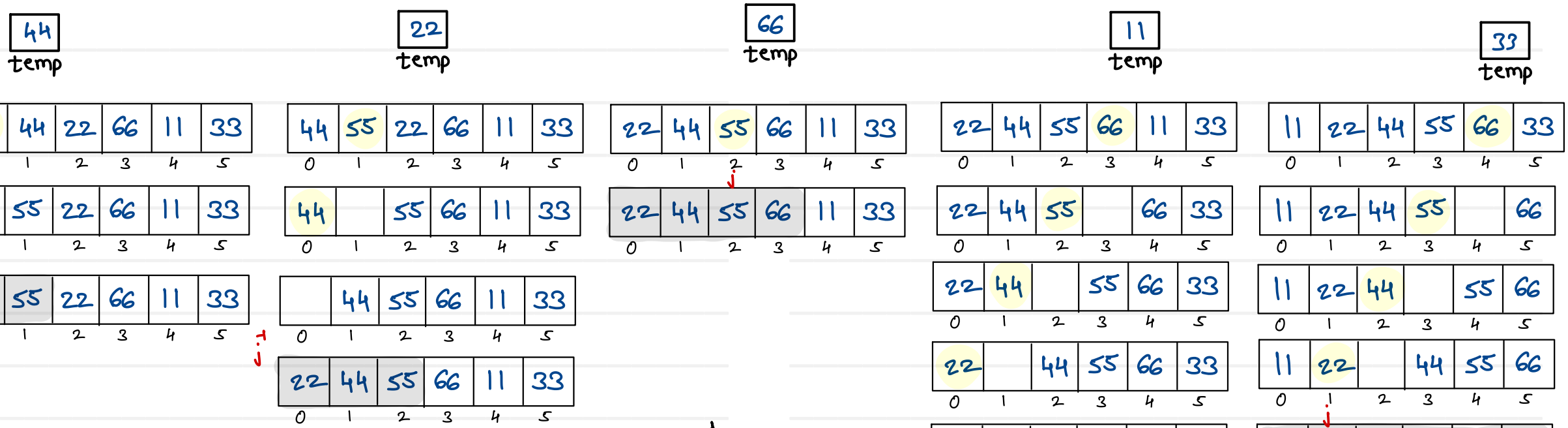
$= \frac{n(n+1)}{2}$

Time $\propto \frac{1}{2}(n^2+n)$

$T(n) = O(n^2)$ worst Avg

Insertion sort



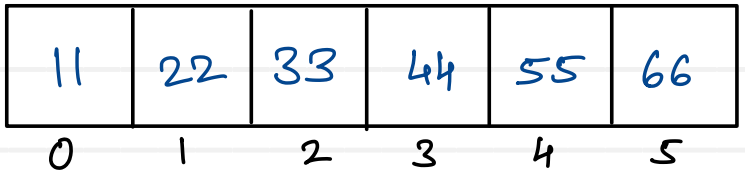


to pick the element : i : 1 → N-1 (i < N)

to compare left neighbors : j : i-1 → 0 (j >= 0)

Insertion sort

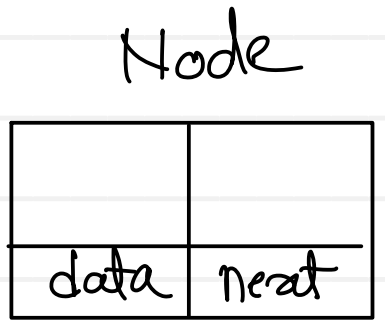
```
for(i=1; i<N; i++) {
    temp = arr[i];
    for(j=i-1; j>=0; j--) {
        if(arr[j] > temp)
            arr[j+1] = arr[j];
        else
            break;
    }
    arr[j+1] = temp;
}
```

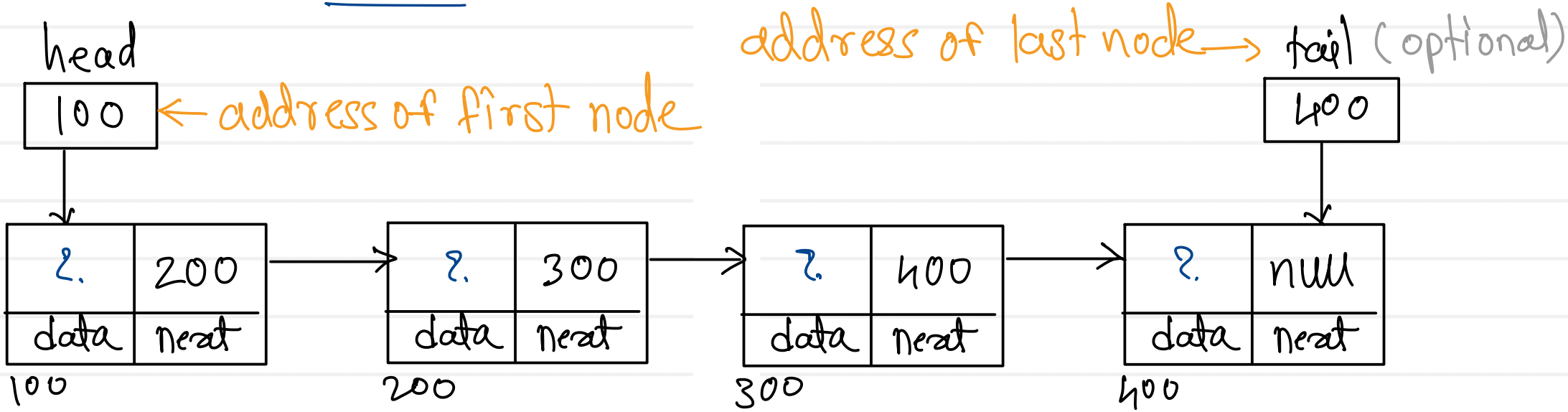


i	i<6	temp	j
1	T	44	0,-1
2	T	22	1,0,-1
3	T	66	2
4	T	11	3,2,1,0,-1
5	T	33	4,3,2,1
6	F		

Linked List

- it is a linear data structure where link / address of next data is kept with its previous data.
- every element of linked list has two parts
  1. data – actual data
  2. link/next – address of next data
- it is referred Node





Linked List

Operations

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

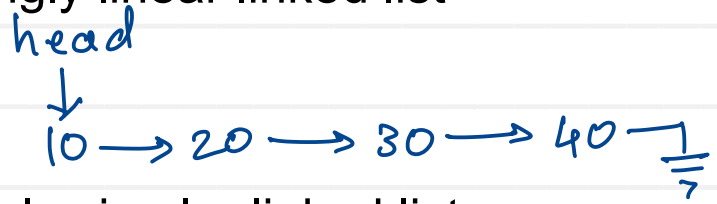
1. Delete first
2. Delete last
3. Delete position

1. Display (traverse) (forward/ backward)

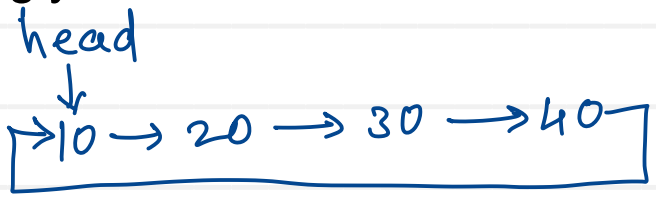
1. Search
2. Sort
3. Reverse

Types

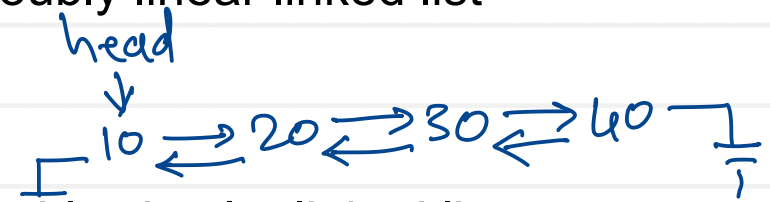
1. Singly linear linked list



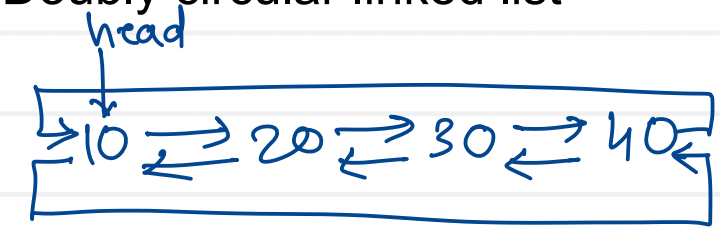
2. Singly circular linked list



3. Doubly linear linked list



4. Doubly circular linked list



Linked List

Node :
data – int, char, short, double, class
next – reference of next Node

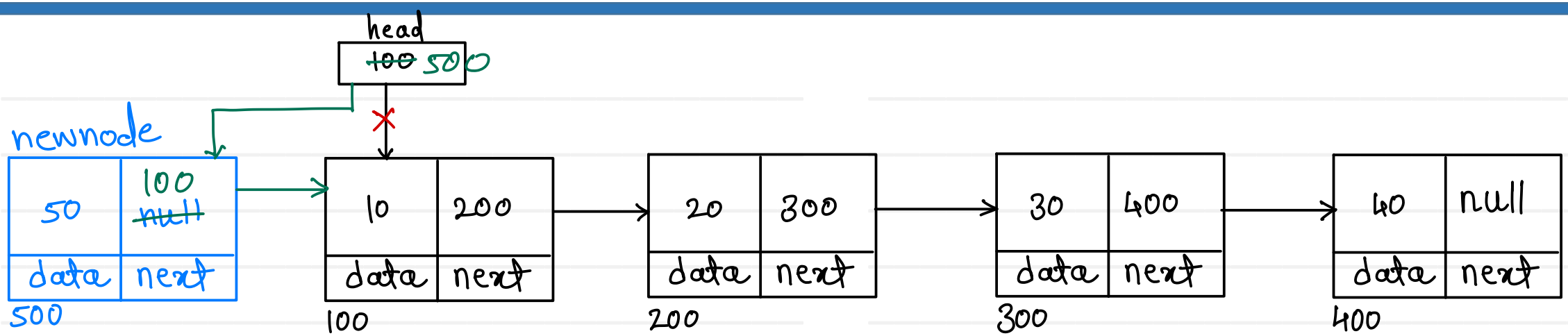
```
class Node {        <- self referential class
    int data;
    Node next;
}
```

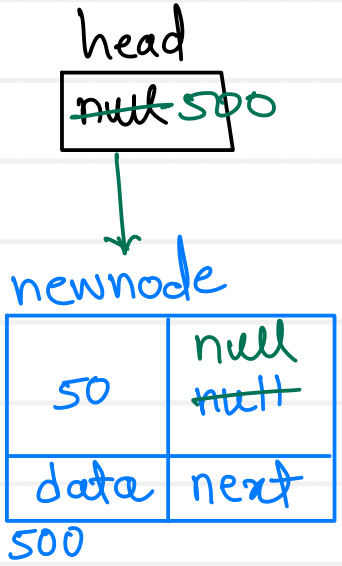
Why inner class ?
- to access private fields of Node class into List class directly

Why static ?
- to restrict access of private fields of List class into Node class directly.

```
class List {
    static class Node {
        int data;
        Node next;
    }
    Node head;
    Node tail;
    int count;
    public List() {...}
    public addNode() {...}
    public deleteNode() {...}
    public traverse() {...}
    public deleteAll() {...}
}
```

Singly linear Linked List - Add first



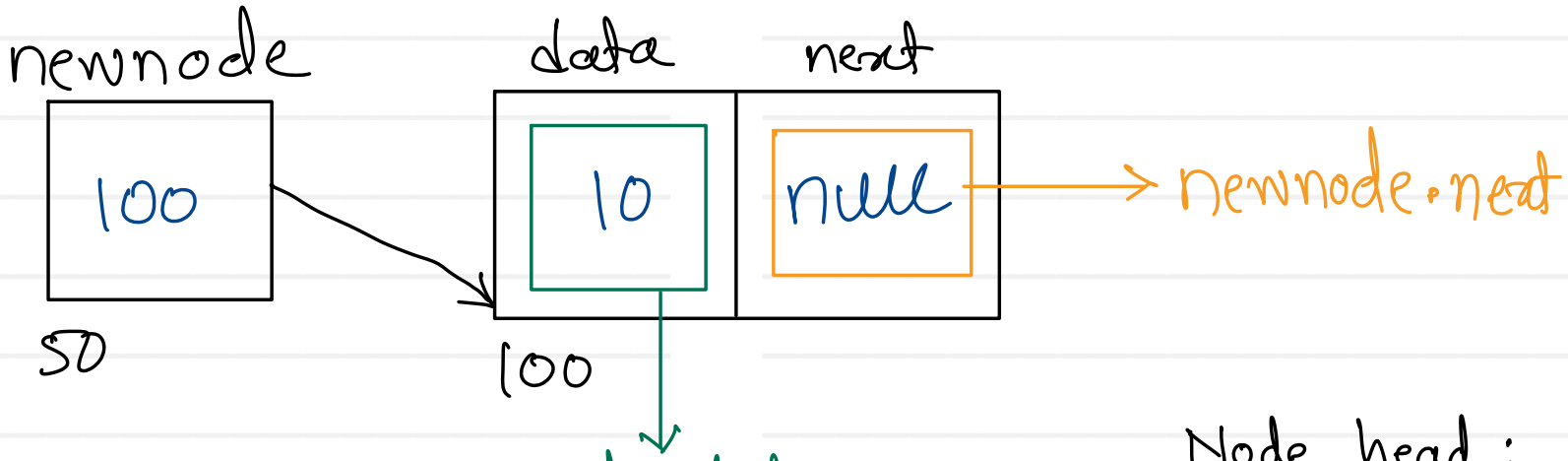


1. create a newnode
2. add first node into next of newnode
3. move head on newnode

newnode.next = head
head = newnode

$T(N) = O(1)$

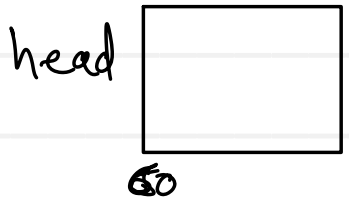
Node newnode = new Node(10);



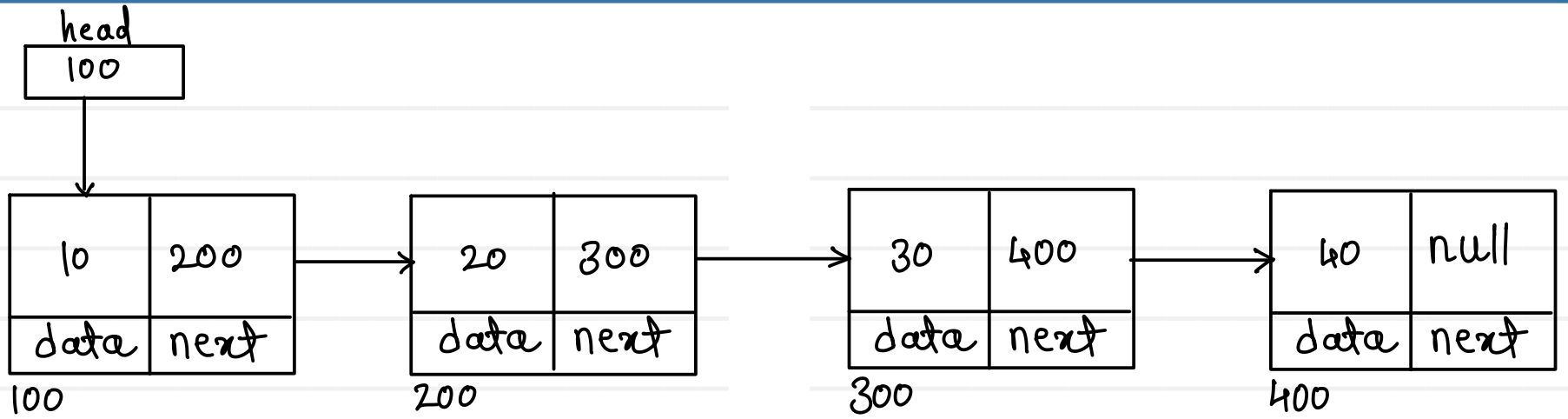
newnode.data = 8. ← writing

8. = newnode.data ← reading

Node head;



Singly linear Linked List - Display



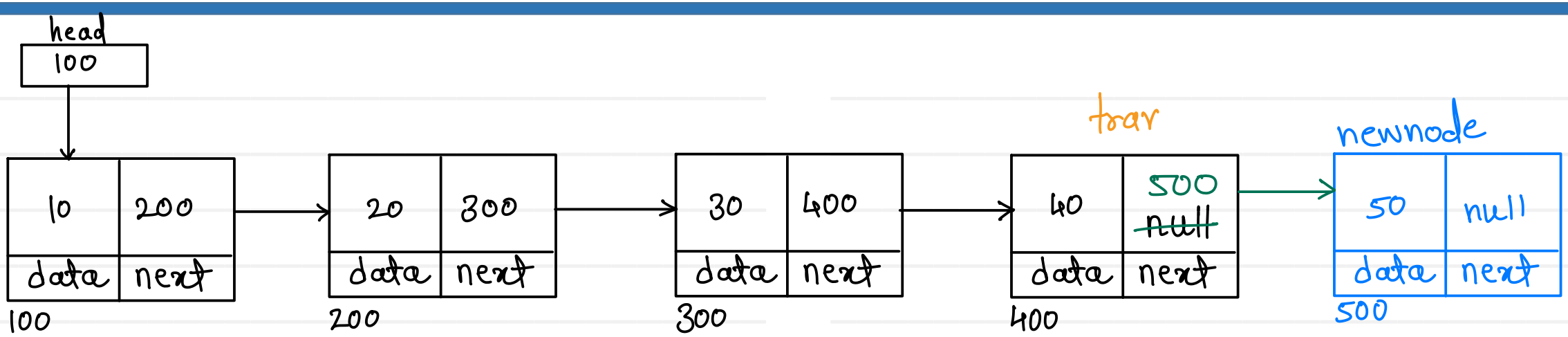

```
trav = head;
while (trav != null) {
    sysout (trav.data);
    trav = trav.next;
}
```

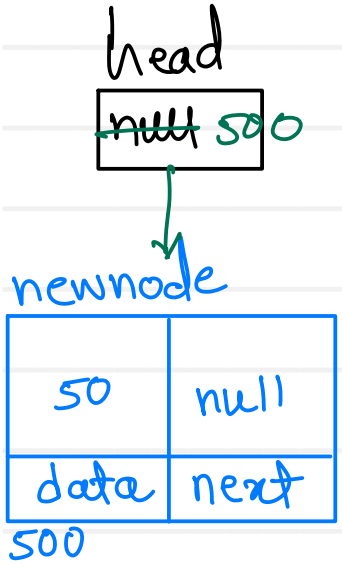
trav	trav.data
100	10
200	20
300	30
400	40
null	

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

$T(n) = O(n)$

Singly linear Linked List - Add last



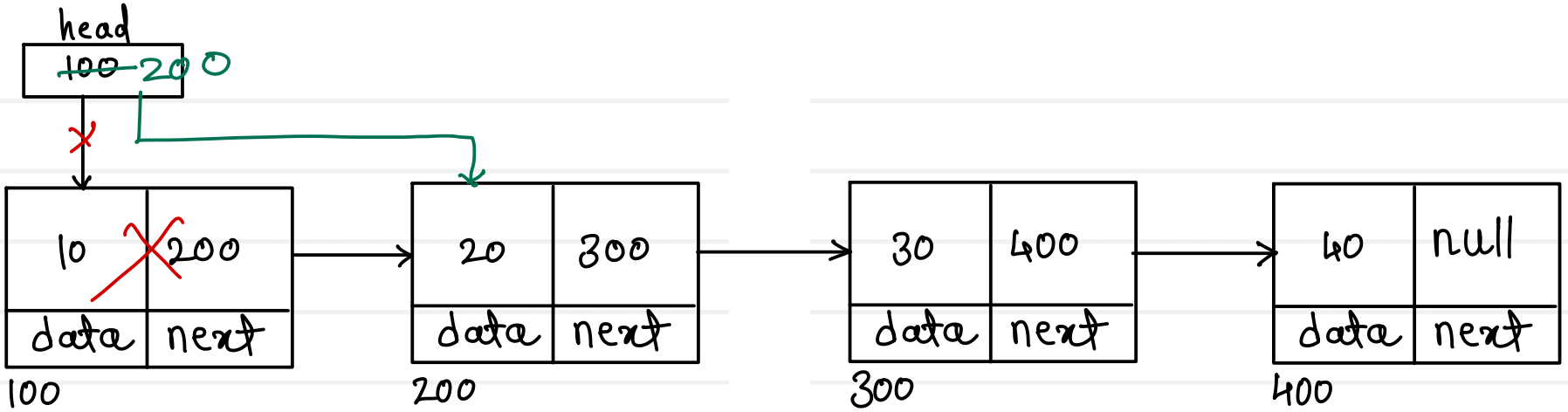


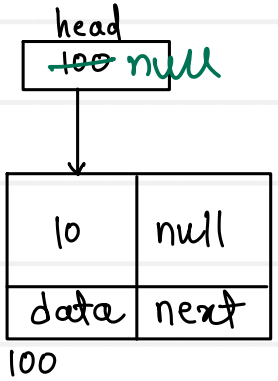
1. create a newnode
2. if list is empty
   add newnode into head
3. if list is not empty
   a. traverse till last node
   b. add newnode into next of last node

```
trav = head;
while (trav.next != null)
    trav = trav->next;
```

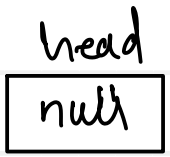
$T(n) = O(n)$

Singly linear Linked List - Delete first





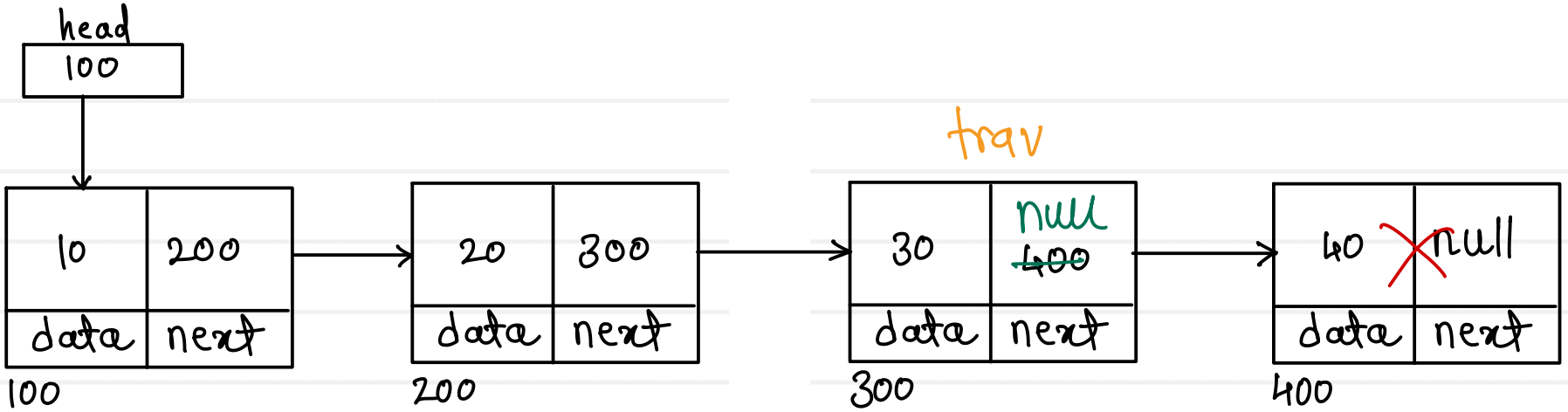
1. if list is empty
   return;
2. if list is not empty
   move head on second node

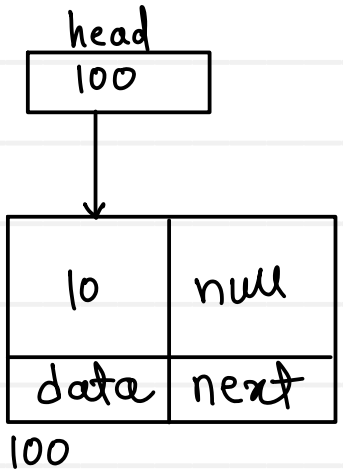



```
head = head.next;
```

$T(n) = O(1)$

Singly linear Linked List - Delete last



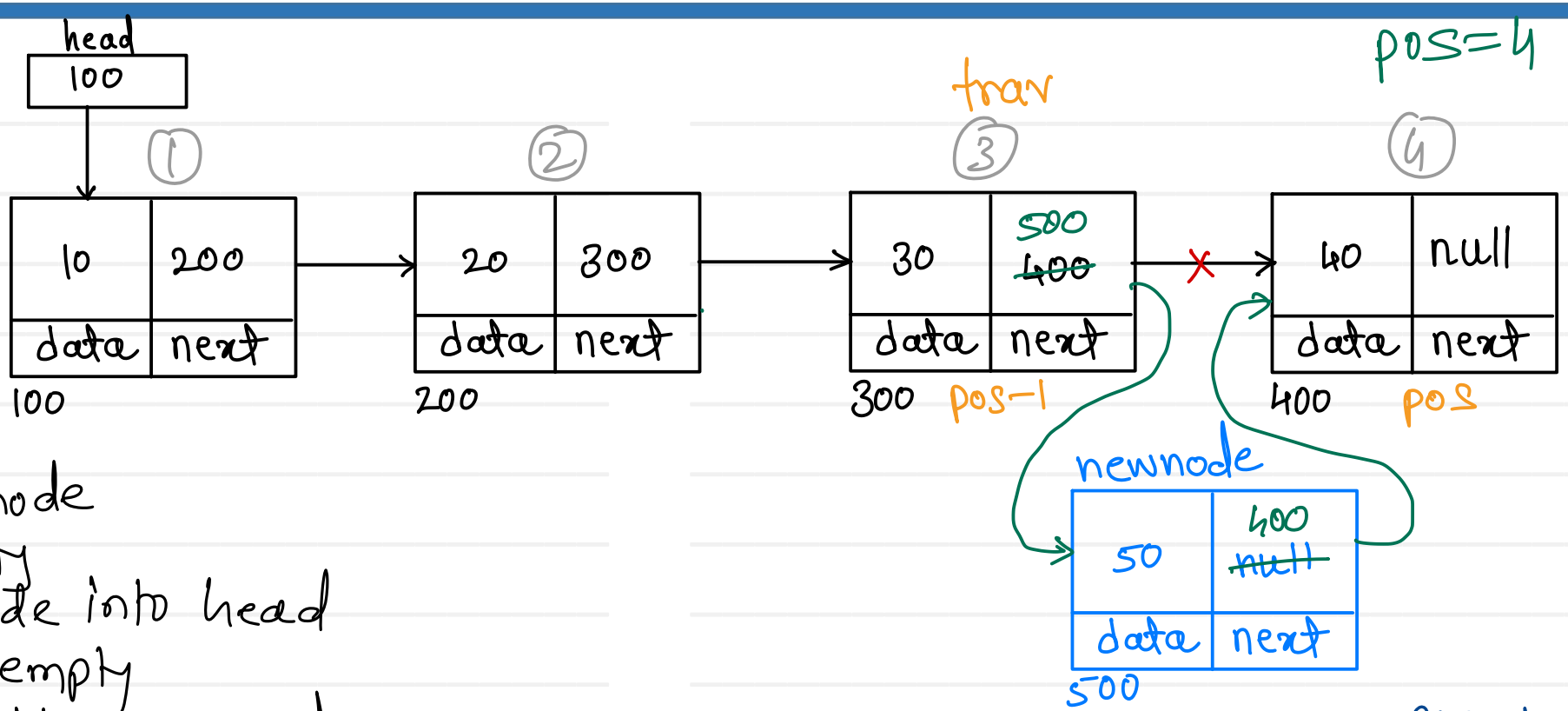



```
Node trav = head;
while(trav.next.next != null)
    trav = trav.next;
```

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

$T(n) = O(n)$

Singly linear Linked List - Add position



1. create a newnode
if list is empty
    add newnode into head
if list is not empty
    2. traverse till pos-1 node
    3. add pos node into next of newnode
    4. add newnode into next of pos-1 node

$T(n) = O(n)$

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

pos=4

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

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com