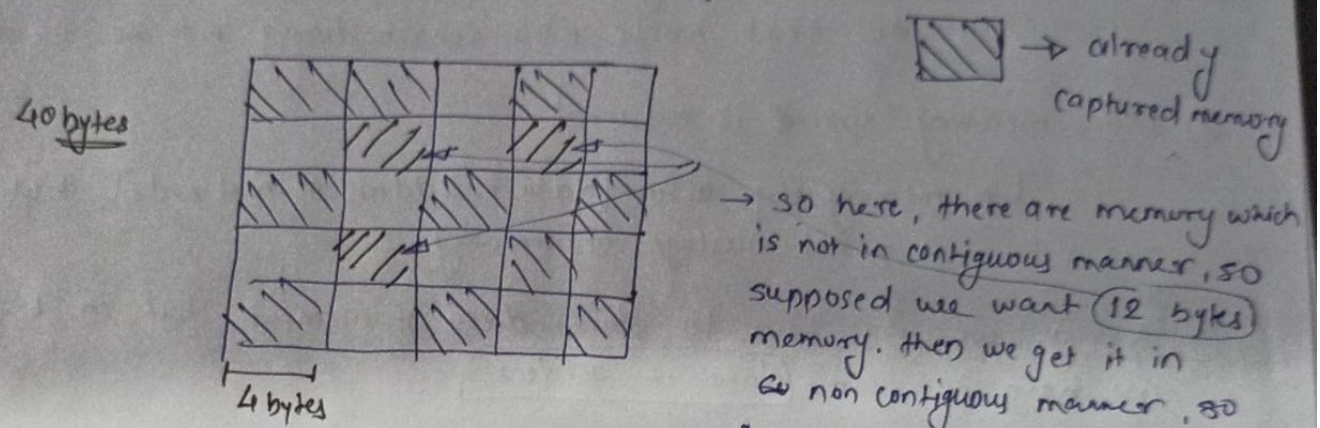
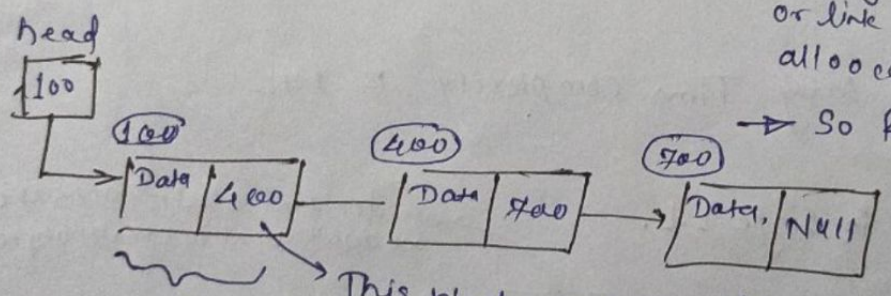
Linked List

Arrays are used only in contiguous memory allocation so for non ~~contiguous~~ contiguous we used array



→ so here, there are memory which is not in contiguous manner, so supposed we want (12 bytes) memory. then we get it in ~~so~~ non contiguous manner, so after got memory how we connect or link this non contiguous memory allocation?

→ So for that we used node.



→ In Array, we have to declared size of array, but in LinkedList we don't have to declared size.

→ Hence (In) L.L. (LinkedList), ~~we have to declared last node with~~ The node which have null value in their section, they declared as Last node

→ The address of the 1st node should be connected to head.

→ Its like train & the head is the engine block of train & it always connected to 1st block of train. & 1st connected to 2nd & 2nd to 3rd & so on.

Advantages of L.L.

→ ① No need to initialize the number of elements to be stored.

② Non contiguous Memory Allocation

Dis-advantages. L.L.

① Access Time complexity is more

↳ for accessing node we have to traverse till that node, so search timing will be increased

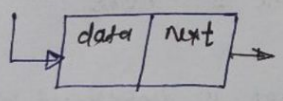
② More memory spaced is required.

↳ memory req. for [data + address of next node] but it is not wastage of memory

③ In Array, we just have to store data of element but in L.L. we have to store $\boxed{\text{data + address.}}$

* But In compare to Array Time complexity is less.

* Structure of node in Single L.L.



we declared static because we dun'4 want to create object

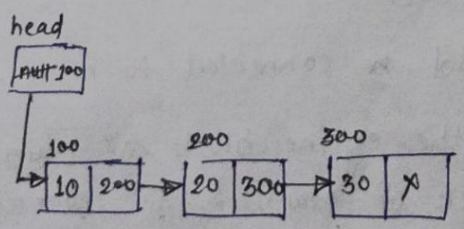
```
static class Node
{
    int data ;
    Node next ;

    Node (int data)
    {
        this. data = data ;
        this.next = null ;
    }

    Node head = null;
```

next contain the address of the next node so we write type as 'Node'

pointer's type is always 'Node'

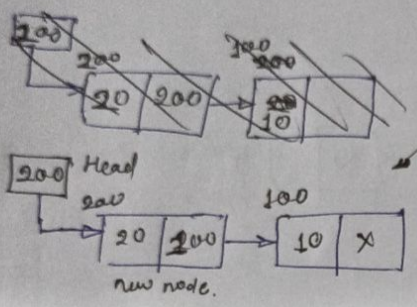


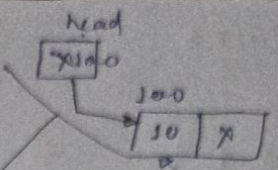
① L.L. is not exist → if head == null

② LL. is exist → if head contain address of node

```
if (head == null)
{
  head = node ;
}
```

If we want to ~~con~~link new node in between head & 1st node like this.





→ If we want to link head to this link, then this code should we write.

If we write ↓

node = ~~100~~ we are talking about address of node i.e. 100, so

node = 100

node.data = we are talking about → 10

node.next = we are talking about → = null

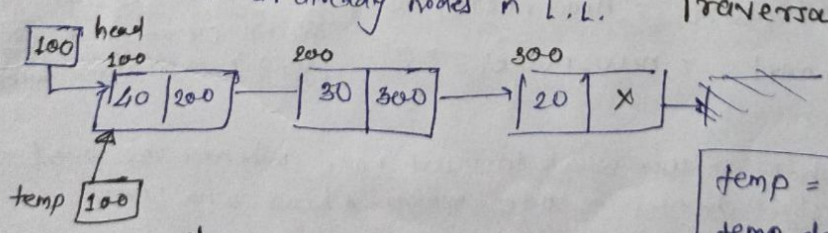
```
else
{
//head = new_node;
new_node.next = head;
head = new_node;
}
```

→ 10 value block will be garbage because of this line if we write it 1st.

→ Initially head have address of 10 i.e. 100 so new_node get address of 10 in next part of new_node. because of this line.

and now head is initialize with 200.

* If we have already nodes in L.L. Traversal ↓



temp = 100
temp.data = 40
temp.next = 200

```
else
{
  while (temp != null)
  {
    sop (temp.data);
    temp = temp.next ;
  }
}
```

40	30	20
temp = 200	temp = 300	temp = 400

Day 4

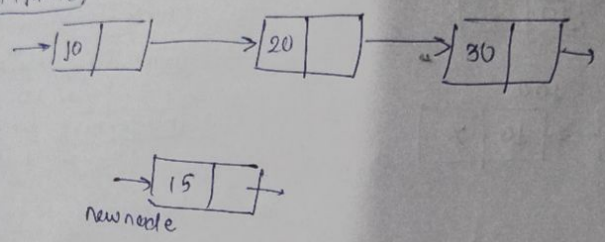
20/4/22

Linked list

① Insertion
→ ⓐ At first
ⓑ At middle
ⓒ At end.

At Middle

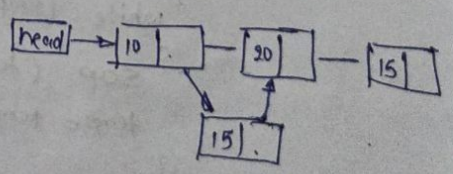
ⓑ




```
void insertAfter (Node prev., int new_data)
{
  if (head = = null)
  {
    sop ("Insertion is not possible");
    return ;
  }
  Node new_node = new Node (15);
  new_node.next = head.next; → ① → ① रचा आहे ② line नाही
  head.next = new_node; → ②  लिहायची cor लिहिली तर [20] [30] हे node island बनेल.
}
```

But what if user want to insert node wherever he want then there is slight changes in code replace 'head' with 'prev'.

```
void InsertAfter (Node prev. ,int new_data)
{ if (prev = = null)
  {
    sop
    return ;
  }
  Node new_node = new Node (new_data);
  new_node.next = prev.next;
  prev.next = new_node;
}
```



At End → If there is no node at start.

```
void append (int new_data)
{
  Node new_node = new Node (new_data);
  if (head == null)
  {
    head = new_node (new_data);
    return ;
  }
  new_node.next = null ;
  Node n = head ;
  while (n.next != null)
  {
    n = n.next ;
  }
  n.next = new_node;
  return ;
```

— x —

l1.append (33); → हे 1st print होईल

l1.insert (10); → But insert हा before 33 करेल

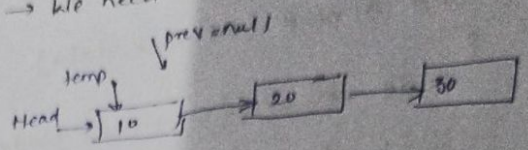
l1.insert (30); → // 30, 10, 33

l1.insert (40); → // 40 30 10 33

l1.insertAfter (l1.head, 50); // 40 50 30 10 33

→ We need additional pointer to delete

Delete
(a) At First
(b) Middle
(c) End.

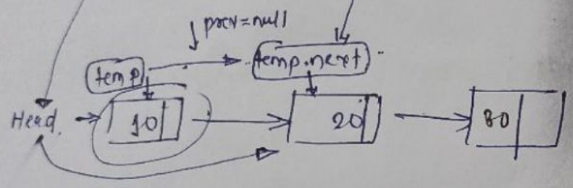


void delete (int key) जो भी delete करना है वो इस key मे डालना

```
{
  Node temp = head , prev = null
  if (temp.data == key && temp != null)
  {
    head = temp.next
    return ;
  }
```

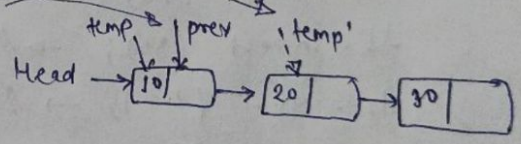
If supposed head is null so temp would be null also cuz temp = head & if head is null then delete क्या करे (काम करमट).

1st case




```
  while (temp != null && temp.data != key)
  {
    prev = temp ;
    temp = temp.next ;
  }
  if (temp == null)
    return ;
```

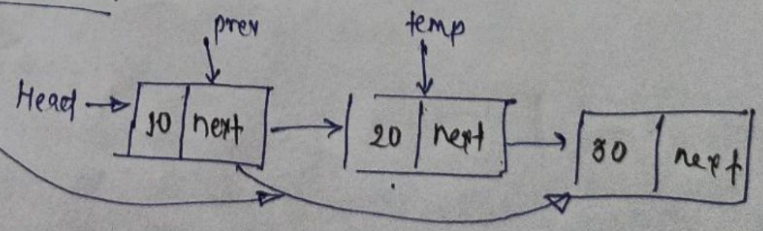
2nd case



case 3

```
prev.next = temp.next
```

→ connection establish



* How to count the node

```
int count ()
{
  Node n = head ;
  int count = 0 ;
  while (n != null)
  {
    count ++ ;
    n = n.next ;
  }
  return count ;
}
```

→ [= head], because we want started count from 1st node.

* How to count the node with [Recursion]

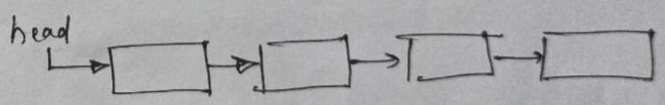
→ when we getting null the count will stop

~~if (n == null)~~

```
int revcount (Node n)
if (n == null)
    return 0 ;
return 1 + revcount (n.next) ;
```

→ (n+1) नहीं केल cox n type is Node

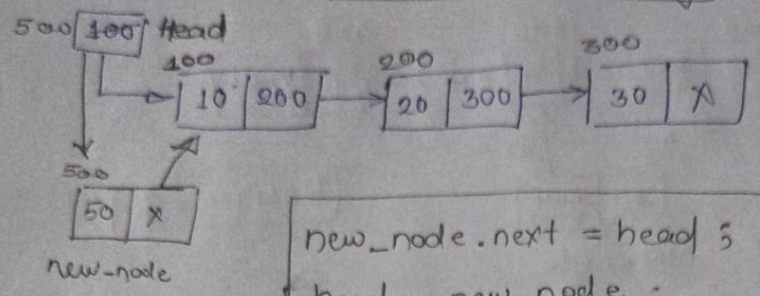
Search the node



1+ revcount (1 + revcount (1+ revcount (1+ revcount (0))))

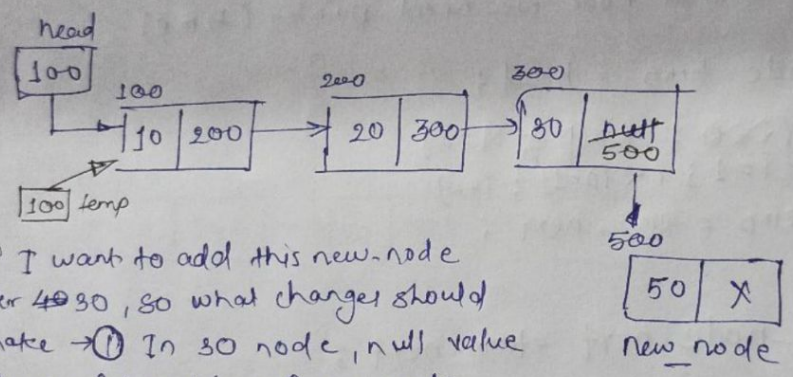
Insertion

Insertion at Beginning




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

Insertion at End



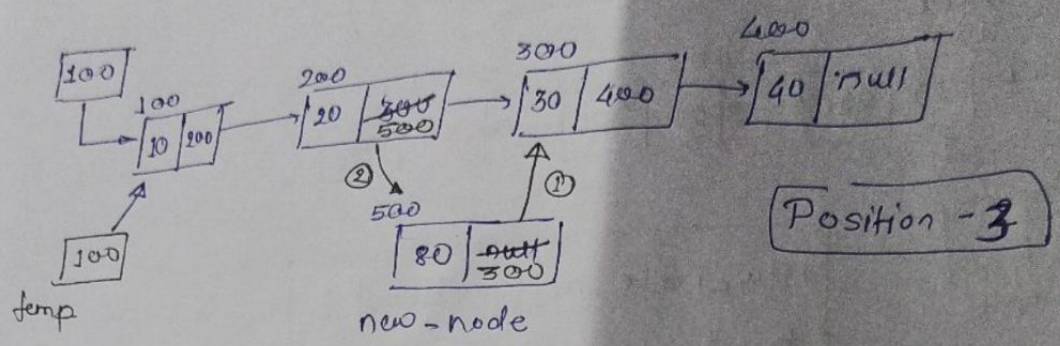
→ If I want to add this new-node after ~~40~~ 30, so what changes should I make → ① In 30 node, null value changes into address of new-node

② We need new pointer to traverse upto last node

```
Node temp = head ;
while (temp.next != null)
  {  temp = temp.next ;
  }
temp.next = new-node
```

→ because of this while loop when temp pointer reaches last node & it equals to null, it control comes out from while loop and go to the next line and link the new_node to it.

Insertion At Specific Location



What changes should I make

① 20's block should have address of new-node i.e. (500).

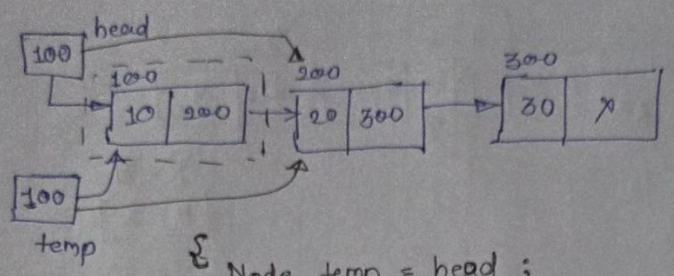
② And new-node contains the address of 80's node i.e. (300).

③ To reach upto that node we need pointer (temp)

```
Node temp = head;
for (i=1; i<(p-1); i++)
{
    temp = temp.next;
}
new_node.next = temp.next;
temp.next = new_node;
```

* One node can link with only one connection, so other connection will be deleted automatically

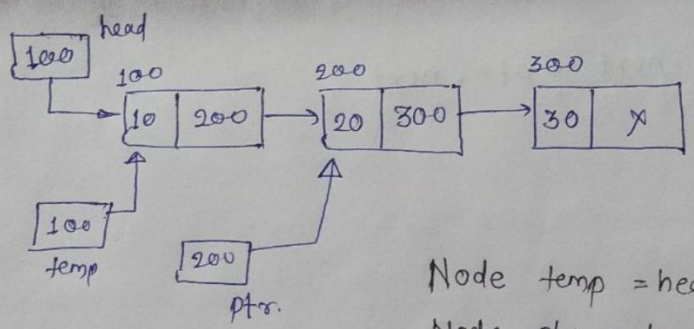
Deletion (From begining)




```
{ Node temp = head ;
  temp = temp. next ;
  head = temp ;
}
```

— x —

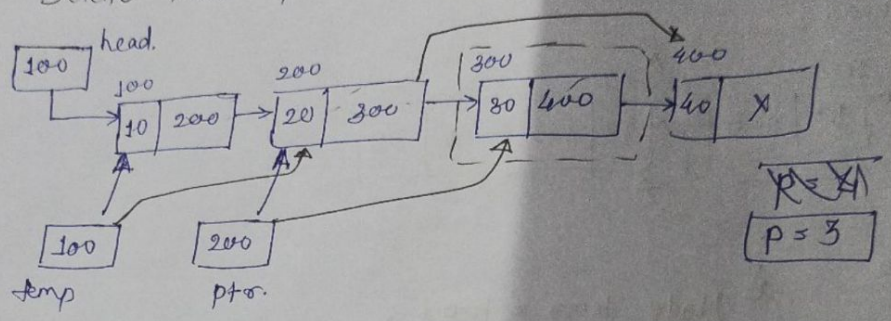
Deletion ~~at~~ From End




```
Node temp = head ;
Node ptr = temp.next ;
while (ptr.next != null)
{
   temp = ptr. ;
   ptr = ptr.next ;
}
temp.next = null ;
```

→ If we take only one pointer, then after traverse to the end we cannot move backward & Hence, we take two pointer.

Delete from specific position

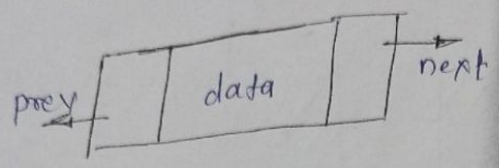



```
Node temp = head ;
Node ptr = temp.next ;
for ( i = 1; i < (p-1); i++)
{
   temp = ptr ;
   Ptr = ptr.next ;
}
temp.next = ptr.next. ;
```

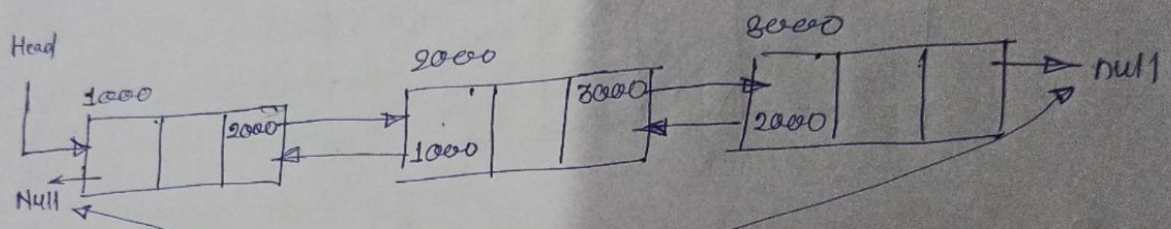
23/04/22.

Day 5

Doubly L.L.




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



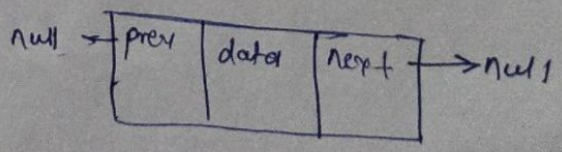
→ If the previous is null then this is the 1st node of the D.L.L.

→ If the next part is null then this is last node of the D.L.L

→ Now, no need to declard / create two pointer while deleting operation At specific position & last position

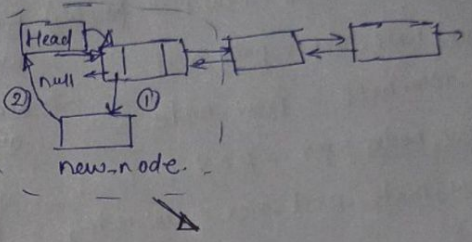
```
Class Node
{ int data;
  Node next;
  Node prev;
  Node (int d)
  {
    data = d;
    next = null;
    prev = null;
  }
Node head;
```

→ When we create node, it is singly L.L. till we are not attach the another node to it ~~& prev~~.

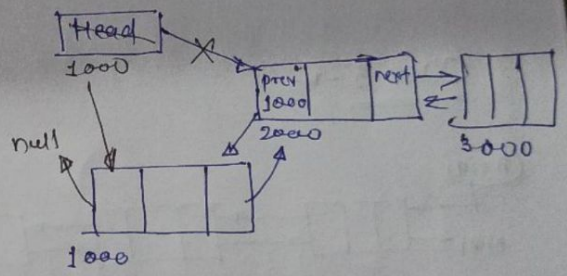


Case 1

```
void insert (int new_data)
{
  Node new_node = new Node (new_data)
  new_node = head ;
  new_node.prev = null ; // this line is mandatory as this is the 1st node of D.L.L. because prev. should be null if its 1st node.
  if (head != null)
  {
    head.prev = new_node ; // ①
    head = new_node // ②
  }
  Node n = head;
  Node p = null ;
```




```
void display (Node n)
{ sop ("Forward Display : ");
  while ( n != null)
  {
    sop (n.data)
    n = n.next ;   → p = n ;
  }
```

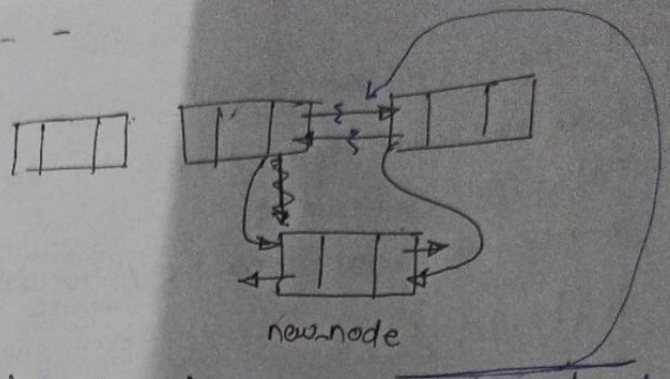


For backward display (Reverse)

```
{ Sop ( "Backward Display : " );
  while (p != null)
  {
    S op (p.data) ;
    p = p.prev ;
```

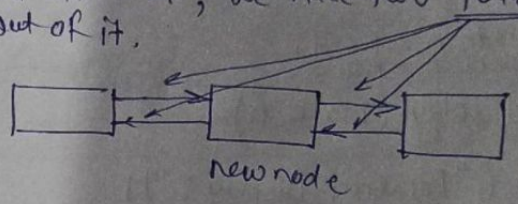
Case 2

```
void insertAfter(Node prev,
                 int new_data)
{
  if (prev = null)
   {return;
   }
Node new_node = new Node (new_data)
new_node.next = prev.next
prev.next = new_node
new_node.prev = prev;
new_node.next.prev = new_node;
```

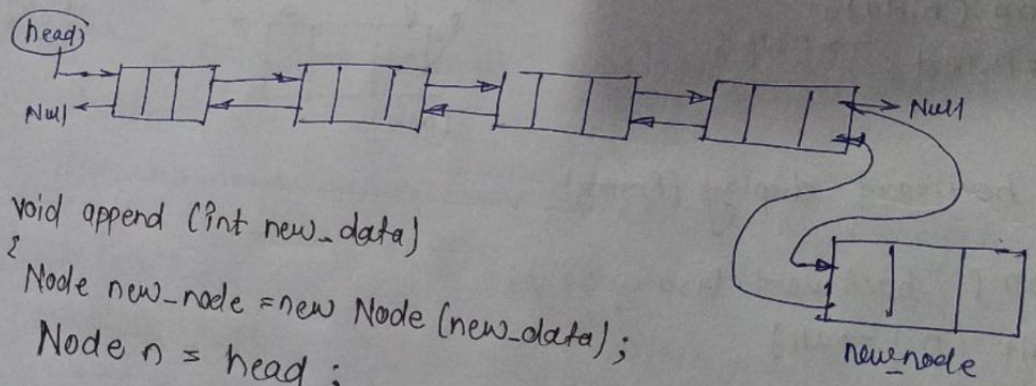


Break करनाना only two links break we have done.

→ And after that, we have (to connect) two four (4) link out of it.



Case 3 →

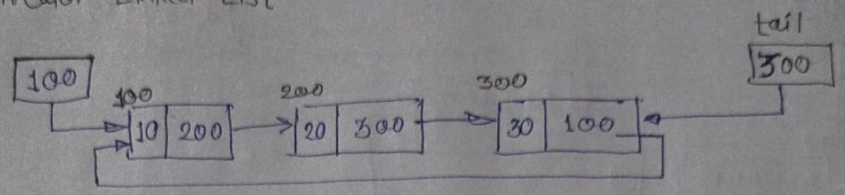



```
void append (int new_data)
{
 Node new_node = new Node (new_data);
  Node n = head;
  new_node.next = null;
  if (head == null)
  {
   ~~new_data.prev = null;~~
   new_node.prev = null;
   head = new_node;
   return;
 }
while (n.next != null)
{
   n = n.next;
}
n.next = new_node;
new_node.prev = n;
```

My Work →

Circular Linked List

→



→

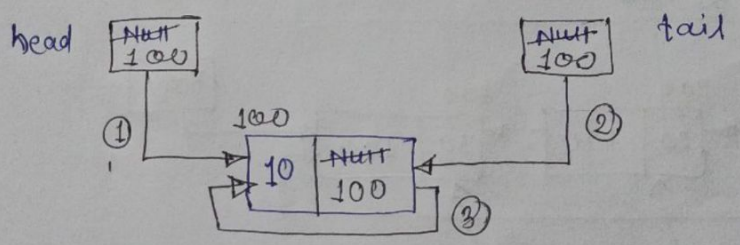
→ Normally हम last node को access करने के लिये one pointer लेते थे। और उसमे 1st node का address डालते थे and last node तक traverse करते थे।

→ SLL में last node को ~~traverse~~ access करने का time O(n) था।

→ CLL मे last node को access करने का time O(1) है।

→ last node have the address of 1st node

Add node first, when there is no NULL node.

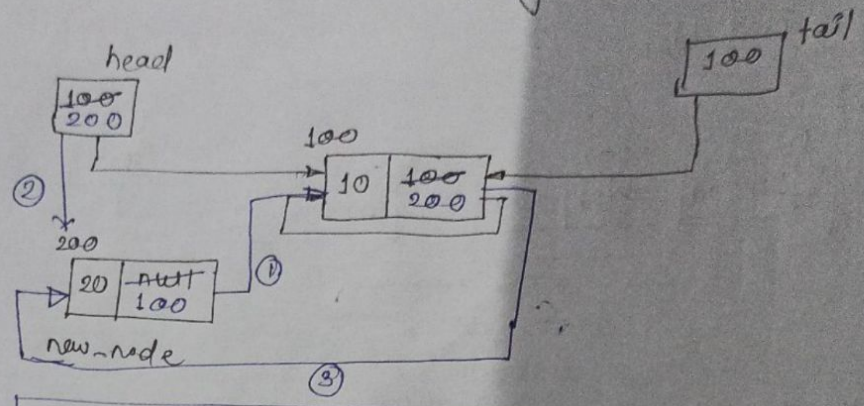


① Head = new_node;

② tail = new_node;

③ new_node.next = head;

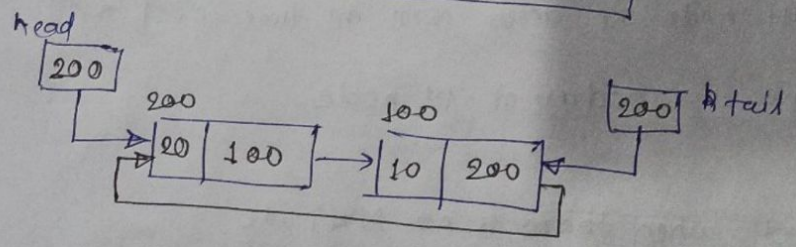
* Insertion of node At Begining



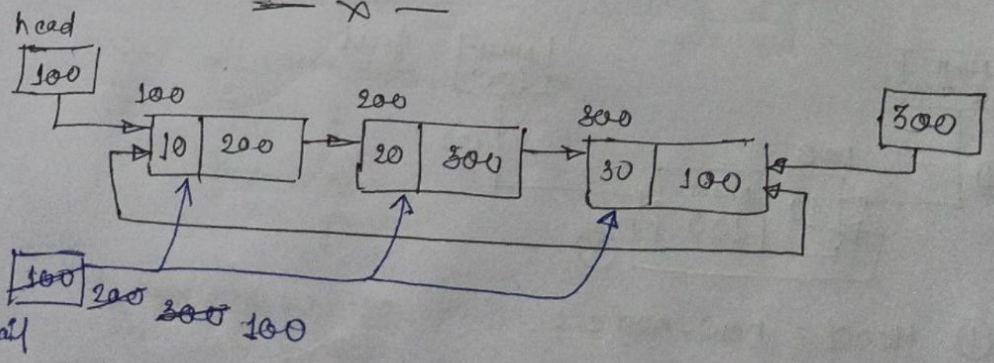
① new_node .next = head ;

② head = new_node ;

③ tail.next = head ;

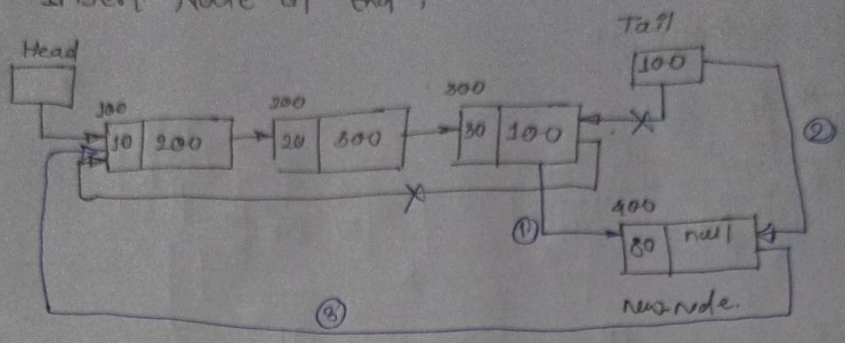


— x —



O/P → 10, 20, 30

Insert Node at End ;



① tail.next = new_node ;

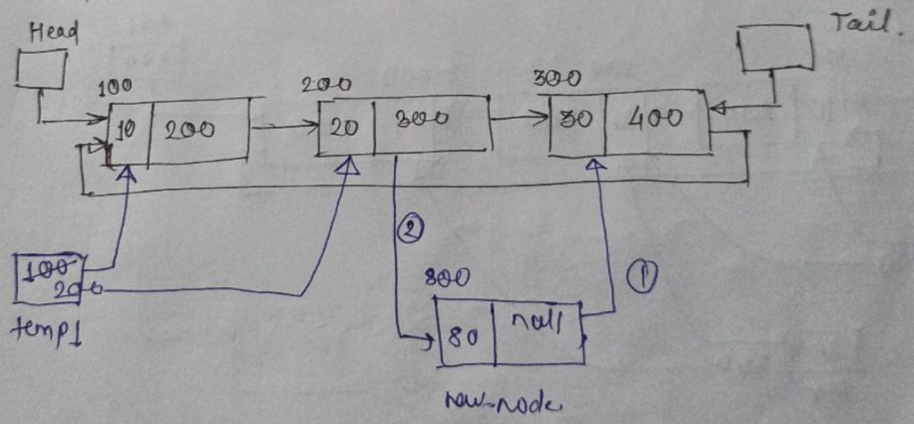
② tail = new_node ;

③ new_node.next = head ;

— x —

Insert Node at Specific position →

* Suppose we want add note at position · 2 .

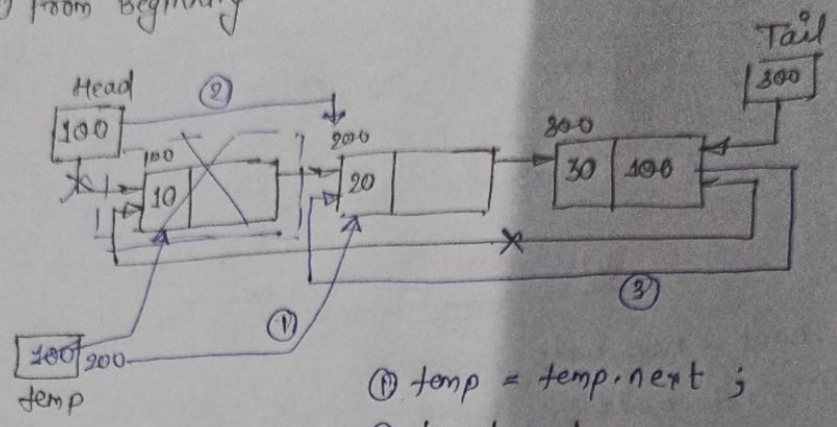


① new_node.next = temp1 . next ;

② temp1.next = new_node ;

Circular L.L. Deletion

① From Beginning →

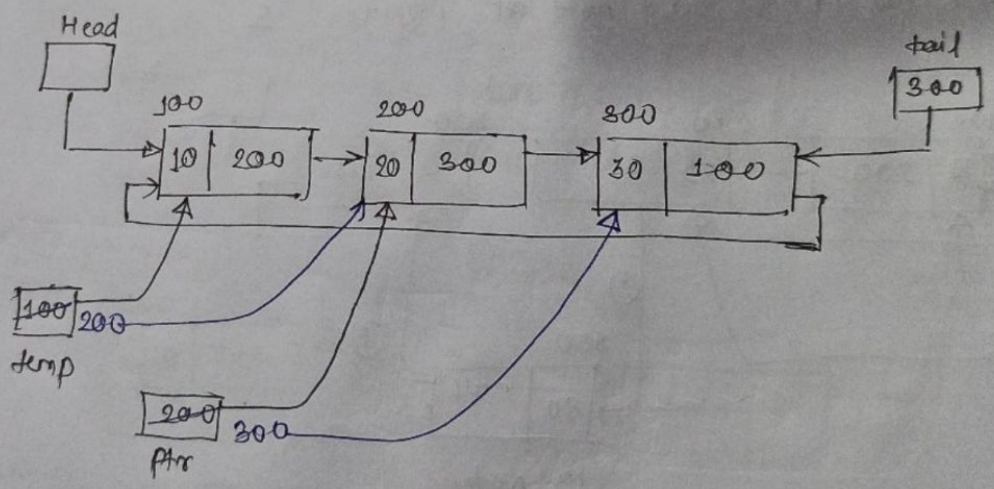


① temp = temp.next ;

② head = temp ;

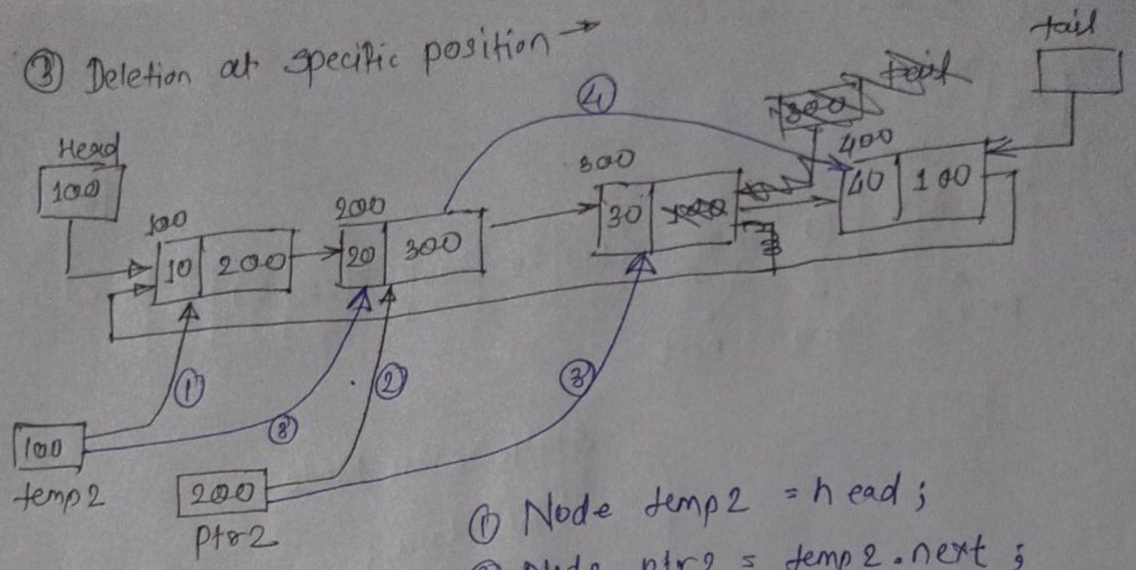
③ tail.next = head ;

② from end ;




```
① Node temp = head ;
② Node ptr = temp.next;
  while (ptr.next != head)
  {
      temp = ptr;
      Ptr = ptr.next;
  }
  temp.next = head;
  tail = temp ;
```

③ Deletion at specific position →



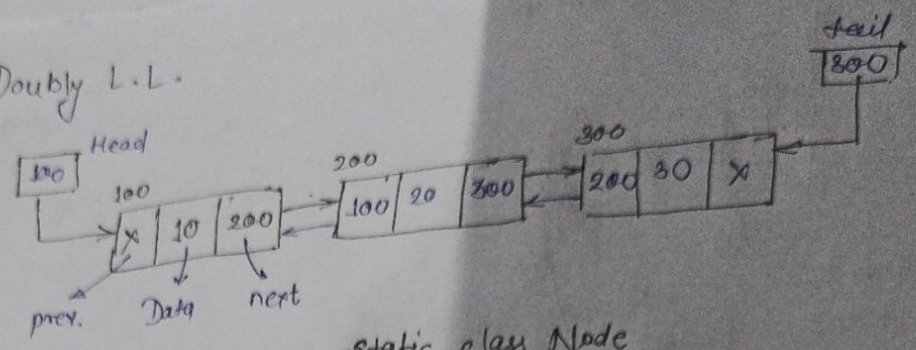
① Node temp2 = head;

② Node ptr2 = temp2.next;

③ for (int i=1; i<(pl-1); i++)
{ temp2 = ptr2;
ptr2 = ptr2.next
}

④ temp2.next = ptr2.next;

Doubly L.L.



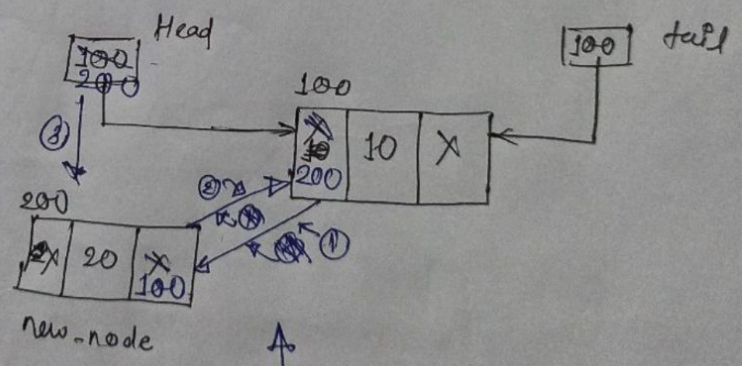
creation of DLL node.

```
static class Node
{
    int data;
    Node next;
    Node prev;

    Node (int data)
    {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}
Node head = null;
Node tail = null;
```

* Insertion

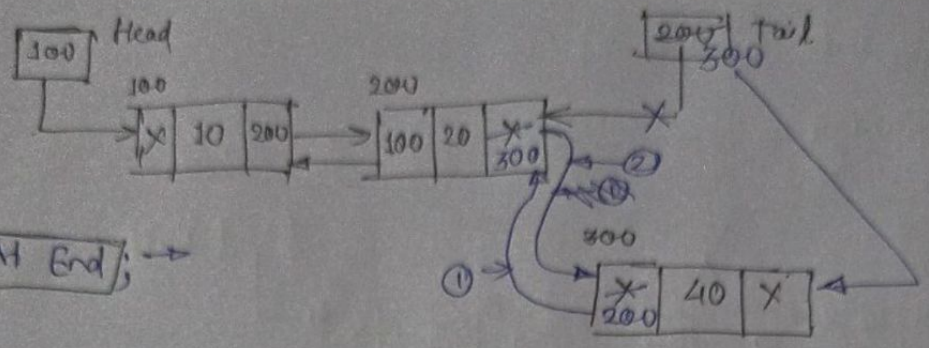
① At Beginning →



① head.prev = new_node;
② new_node.next = head;
③ head = new_node;

OR

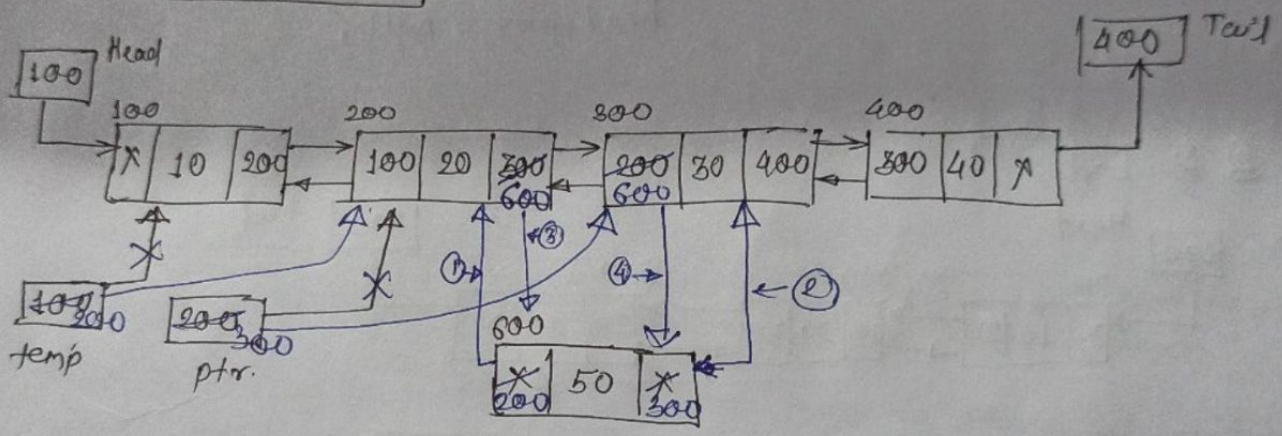
① new_node.next = head;
② head.prev = new_node;
③ head = new_node



② At End; →

```
tail.next = new_node ;
new_node.prev = tail ;
tail = new_node ;
```

③ At specific position →

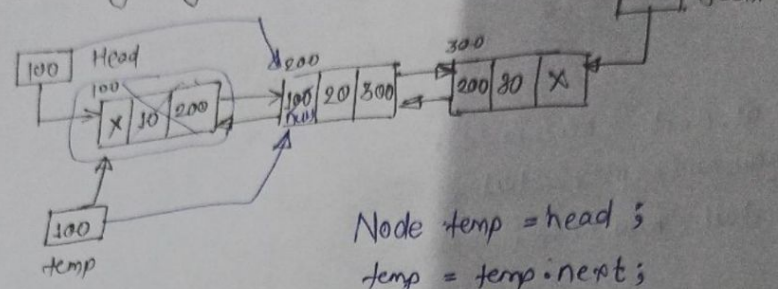



```
p = 3
for (i=1 ; i<(p-1) ; i++)
{
    temp = ptr ;
    ptr = ptr.next ;
}
```

① new_node.prev = temp ;
② new_node.next = ptr ;
③ temp.next = new_node ;
④ ptr.prev = new_node ;

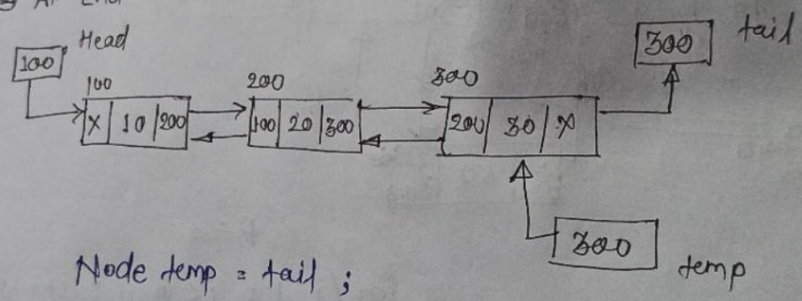
Doubly L.L [Deletion]

① At beginning



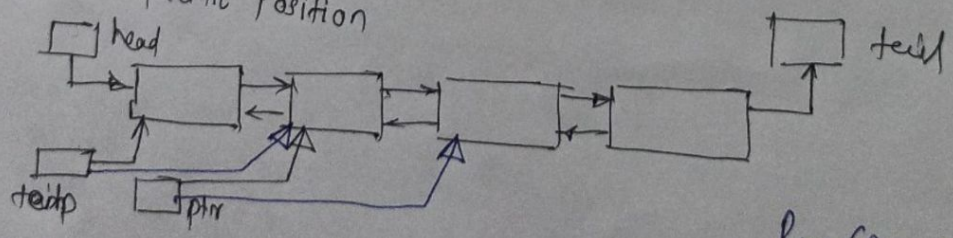

```
Node temp = head ;
temp = temp.next;
head = temp;
head.prev = null ;
```

② At End




```
Node temp = tail ;
~~temp = temp.next~~
temp = temp.prev
temp.next = null;
tail = temp. ;
```

③ At specific Position



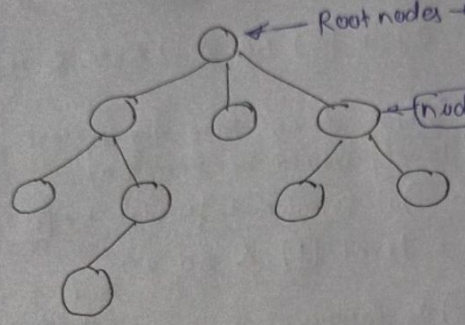

```
temp.next = ptr.next ;
ptr.next.prev = temp ;
```

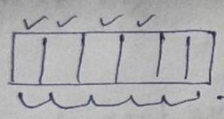
(P = 3)

```
for (i=1; i<(p-1); i++)
{ temp = ptr ;
  ptr = ptr.next
}
```

Tree

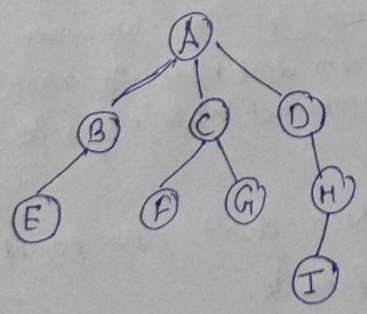
- Binary Tree
- non-Binary Tree



* 

→ In (array) (linked List) (Stack) (queue) ~~st~~ storing the data in (sequential manner) one-after-another

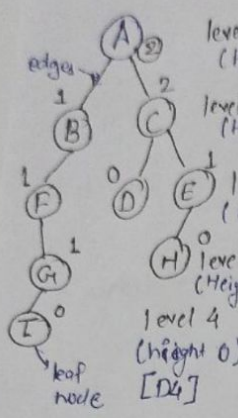
* But in case of (non-linear DS), the data is stored at different level, they have hierachy, for that.



B, C, D → child of A
E → child of B
F, G → child of C
H → child of D
I → child of H

Binary Tree → one node have max only two child node

→ Here the max. number of children is (2) so this is Binary Tree.



→ Degree of Tree → max no. of children of node is [2], so Degree of Tree is [2]

Degree of C → [2] no. of node C is (2)

* The node which do not have any parent, then that node [level] would be zero (0).

So the [level] of [root node] would be [zero] (0).

B & C have level [1] & so on

* We can also find out [level of node] as follows

[level of F] → node F to root node ~~had~~ have two edges so level of [F] would be [2].

* H1, H2, H0 → Height of node & it measured from leaf node and goes in upward direction upto root node and [Height of leaf node = 0]

[Height of Tree] →

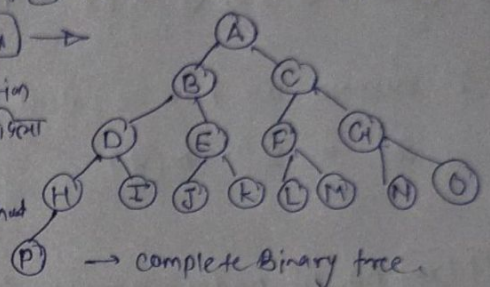
→ so we can see that [H4] is the highest at root node, so Height of tree [4] or we can calcul. also by traverse from leaf node to root node and in that path the number of ~~node~~ edges is equal to Height of Tree.

* level → level we start from the node which don't have parent node
* Height → Height would be start from leaf node.
* Depth → ~~It~~ It would be start from ~~leaf~~ root node

Types of Binary Trees

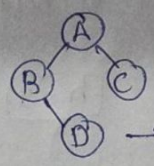
① Binary Tree → The nodes which have 0, 1, 2 degree means 0, 1, 2 child have it.

② Complete B.T. → In this binary Tree, all the nodes have feed with max. degree of nodes except the last node, means all the nodes have 2 child but last node don't have any. For ex →



→ complete Binary tree.

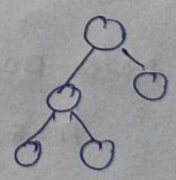
and there is another condition
जर H नंतर आपण left child दिला तर it is complete B.T. but जर right child दिला असता without left child तर, It would not be complete B.T.



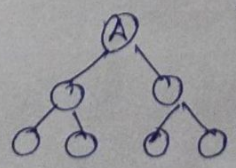
→ इथे B मध्ये पहिले right side ला node दिला, left नाही तर, not complete B.T.

③ Full B.T. / 2-Tree / Proper BT / Strictly B.T.

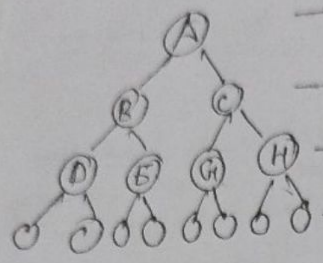
↳ If the nodes have zero child or 2 child., but not 1 child.



④ Perfect Binary Tree → Completely filled not only 1 & 0.



* Max no. of nodes possible at level $i \rightarrow \boxed{2^i}$



→ L0 nodes(n) = 1 → 2^0 → 2^i & i = level no.

→ L1 nodes(n) = 2 → 2^1

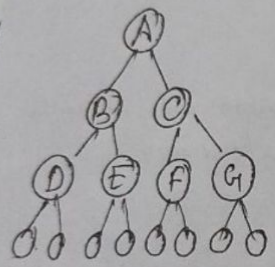
→ L2 nodes(n) = 4 → 2^2

→ L3 nodes(n) = 8 → 2^3

If suppose we have to calcul. ~~at~~ no. of nodes at level 3

$2^i \rightarrow 2^3 \rightarrow 8$

* $\boxed{\text{Max}}$ no. of nodes possible in a BT of height h → max n → $\boxed{2^{h+1}-1}$



H3 $n = 2^0$

H2 $n = 2^1$

H1 $n = 2^2$

H0 $n = 2^3$

$\boxed{h=3} \rightarrow 2^0 + 2^1 + 2^2 + 2^3$

$= \boxed{15}$

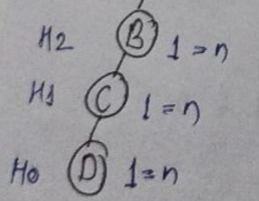
$= 2^{h+1} - 1$

$= 2^{3+1} - 1$

$= 16 - 1$

$= 15$

* $\boxed{\text{Min}}$ no. of nodes possible in a BT of height h → min n → $\boxed{h+1}$



H3 A 1 = n

H2 B 1 = n

H1 C 1 = n

H0 D 1 = n

$h = 3 \rightarrow n = 4$

$\boxed{\text{min } n = (h+1)}$

* Max height of a BT having n nodes

previously, we see that

min $n = h+1$.

$\boxed{h = n-1}$

$\boxed{\text{max height } h = n-1}$

* Min height of a BT having 'n' nodes,

previously, we see that.

max. $n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

$\log_2(n+1) = \log_2 2^{h+1}$

$\log_2(n+1) = h+1$

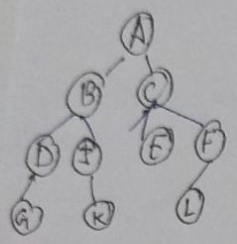
min height $h = [\log_2(n+1) - 1]$

BT implementation → It implements using array & L.L.

- Array
- L.L.

Implementation Using Array

0	1	2	3	4	5	6	7	8	9	
A	B	C	D	I					Null	k



Left child → $2i+1$

Right child → $2i+2$

i = index of parent node.

cox I doesn't have left node so it is consider Null

* A is the parents of ~~A & I~~ B & C so for

Left child → $2i+1$
$= 2(0)+1 = 1$

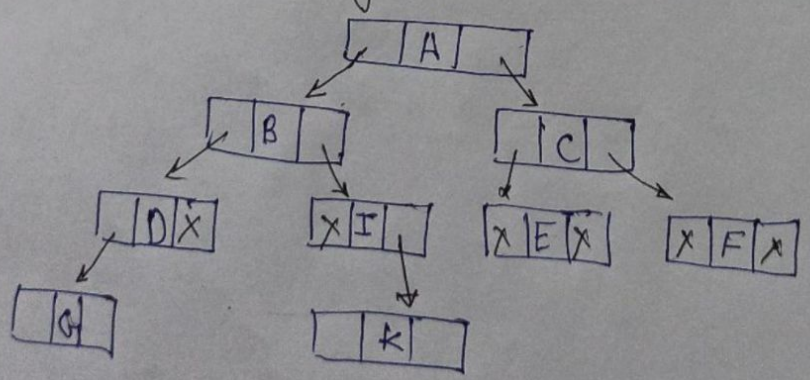
Right child → $2i+2$
$= 2(0)+2 = 2$

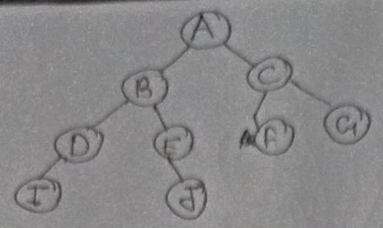
* B is the parents of D, I

Left child → $2i+1$
$= 2(1)+2$
$= 3$

Right child → $2i+2$
$= 2(1)+2$
$= 4$

Implementation Using L.L.





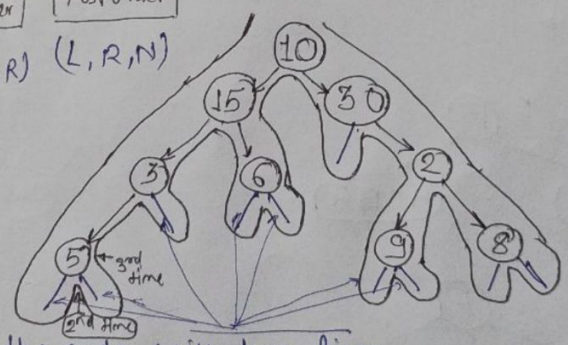
Recursive function call चालत असते. Recursive चा वर्क पाहता Function

→ सगळ्यात पहिले (A) चा left child विचारेल i.e. (B) and (B) is new node so पुन्हा new node i.e. (B) चा left child विचारेल i.e. (D) & again (D) is new node so पुन्हा new node चा left child i.e. (I)

→ आता (I) चा left नाही, so आता right विचारेल & (I) चा right पण नाही so तो (D) मध्ये जाणार & तेथे तो right child विचारणार पण नसल्यामुळे (B) कडे जाणार (म्हणजे (D) ला ज्यूस call आला तेथे) & B चा right विचारणार i.e. (E). & आता (E) is new node so त्याचा left विचारणार पण तो नाही आहे so right विचारणार i.e. (J) & (J) is new node so left & right विचारणार नसल्यामुळे return parent कडे जाणार & असे करता करता — so on

Tree Traversal

Inorder	Preorder	Postorder
(L, N, R)	(N, L, R)	(L, R, N)

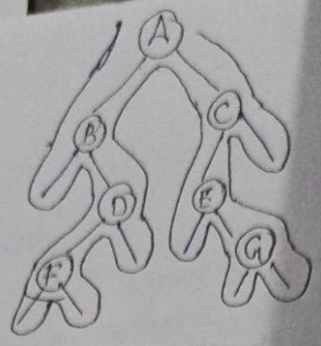


* 1st make all the nodes with two lines.
* Then traverse from left & if the nodes comes for 1 time then put it in Preorder, if the nodes comes for 2nd time, put it in Inorder and for 3rd time put it in the post order.

Preorder → 10, 15, 3, 5, 6, 30, 2, 9, 8

Inorder → 5, 3, 15, 6, 10, 30, 9, 2, 8

Postorder → 5, 3, 6, 15, 9, 8, 2, 30, 10



root node at first in preorder

Preorder → A, B, D, F, C, E, G

In middle

Inorder → B, F, D, A, E, G, C → right subtree

left subtree

Postorder → F, D, B, G, E, C, A ← root node at last in Postorder

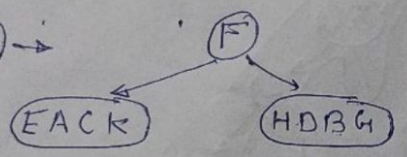
Ex → If two order given, How to find out 3rd order

(NLR) Preorder → F, A E K C D H G B

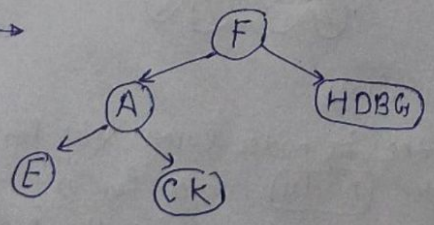
(LNR) Inorder → E A C K F H D B G

(LRN) Postorder →

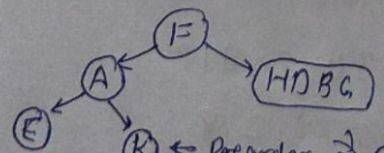
Step-1 →



Step-2 →



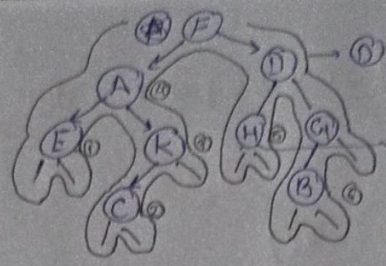
Step 3 →



← Preorder में K पहिले आता है C से

← C वो K के left है Inorder में.

Step 4 →



(D) comes 1st in preorder so we take it 1st.

In Inorder (H) is before (D) thats why we put it as left subtree

~~F G D B H I E C A~~

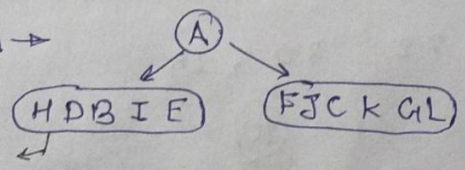
Post order → E, C, K, A, H, B, G, D, F

Problem 2 →

(LNR) Inorder → H D B I E (A) F J C K G L (A) → Root

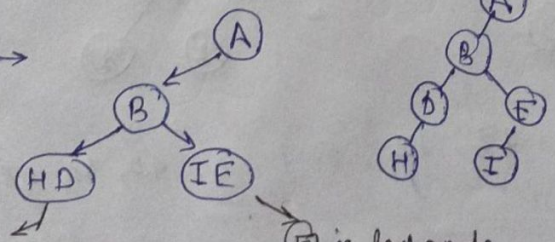
(LRN) Post order → H D I E B J F K L G C (A) → Root

Step 1 →



In postorder B is in the last among all this (HDBIE) nodes, so (B) should be root node.

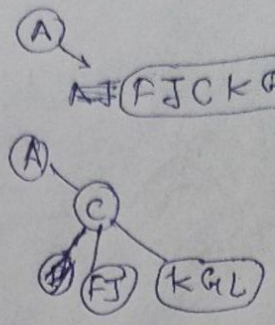
Step 2 →



(D) last node in postorder among (HD) so (D) is the root node & (H) is at left side in (Inorder) so it is left node of (D)

(E) is last node among (IE) & so (E) is root node & (I) is in left of (E) in (Inorder) so it left node of (E)

step 9 →



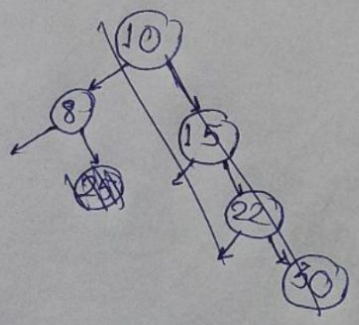
→ C is last node of Postorder

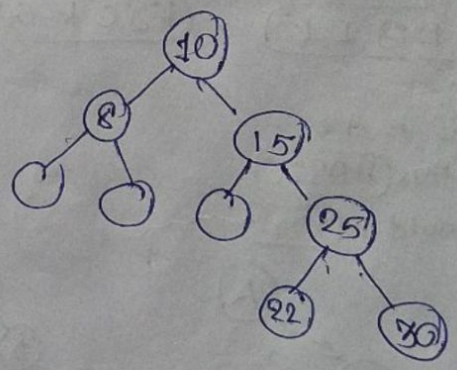
— x —

Binary Search Tree →

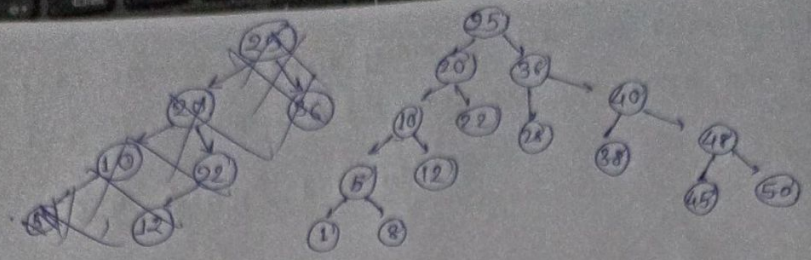
* It is BST in which every node satisfy the following conditions
  1. All values in the left subtree of a node are less than the values of the node.
  2. All the values in the right subtree of a node are greater than the values of the node.

10 8 15 25 22 30 , 15



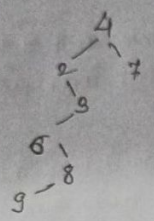


25, 20, 36, 10, 22, 12, 5, 1, 8, 40, 38, 48, 28, 45, 50



Normal Binary Tree Vs Binary search tree (BST)

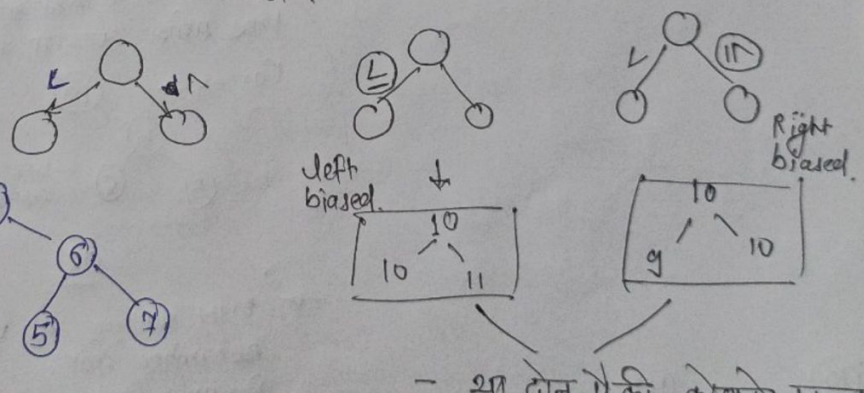
→ [4 8 2, 3 6 5 7 1]



→ इथे child 0, 1, 2 या पैकी कोणता फरक पाहिजे.

→ and in BT the data is in not proper order so search करतांना problem येईल. i.e. O(n) time लागेल.

BST → In this, there is 'search' in between there name, so it means that जर आपण systematic way नी data put केला तर search करतांना सोपी जाईल.

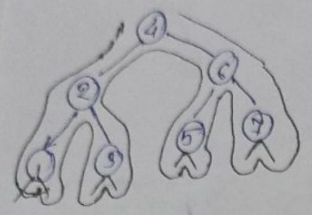


→ या diagram मध्ये दिसत आहे की, ② हा ④ पेक्षा लहान आहे so left ला येतो then ③ हा ④ पेक्षा लहान but ② पेक्षा मोठा so ② च्या right ला. Now ⑥ हा ④ पेक्षा मोठा so right ला, ⑤ हा ④ पेक्षा मोठा so right ला but ⑥ पेक्षा लहान so ⑥ च्या left ला ⑦ हा ④ पेक्षा मोठा so right ला & ⑥ पेक्षा पण मोठा so ⑥ च्या right ला, & ① हा ④ पेक्षा लहान & ② पेक्षा पण so left ला.

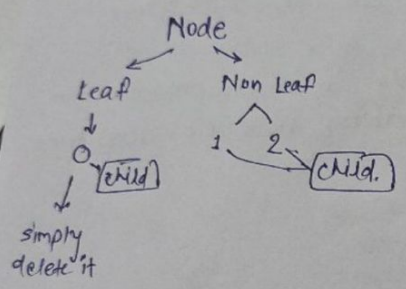
- या दोन पैकी कोणते फरक follow करायचे.
- हे used करायचे जेव्हा unique key दिली नसेल.

* Binary Search Tree करतो (Inorder), sorted असतो ascending order मध्ये.
* Inorder आपण काढतो जेव्हा आपण 2nd time element वर reach करत असेल.

Inorder → 1, 2, 3, 4, 5, 6, 7.



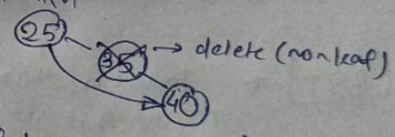
Deletion in BST.



→ Leaf node ला जर delete करत असेल तर काही rearrange नाही केल तरी चालेल.

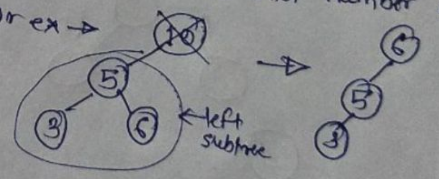
→ जर non-leaf node ला delete करत असेल ज्याच्या खाली 1 node आहे तर त्या खालच्या node ला grand parent सोबत attach करणे

For ex →



→ जर non-leaf have 2 child. असेल तर जो node delete करत आहे त्याच्या left sub tree मधील सगळ्यात मोठा number घेणे.

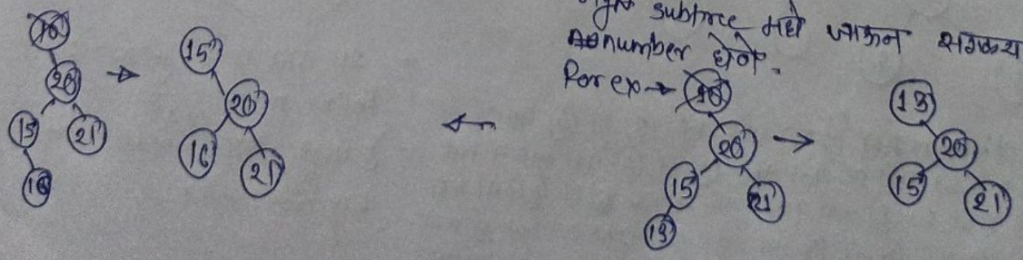
For ex →



OR

→ right subtree मधे जाऊन सगळ्यात small number घेणे.

For ex →



Pre-order, Postorder, Inorder of BST

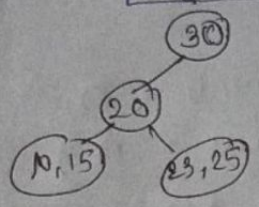
→ (11) The Preorder traversal sequence of Binary Search Tree is

Preorder → 30, 20, 10, 15, 25, 23, 39, 35, 42 (RLN)

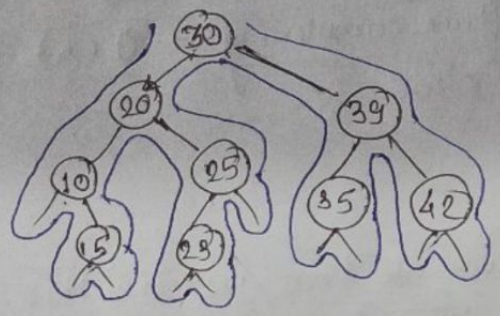
which one of the following is post order traversal of same tree?

→ Inorder BST चा नेहमी ascending (increasing) order मध्ये असते.

10 15 20 23 25 30 35 39 42 (LNR)



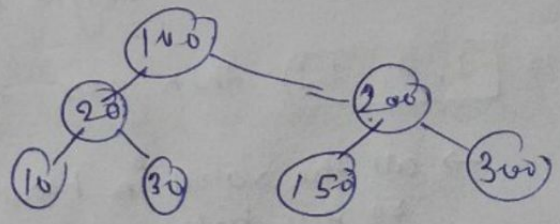
→ आपण जर preorder check केल तर 30 नंतर (20) आहे so next root is (20) & जर आपण inorder check केल तर (10,15) & (23,25) हे (20) node चा left, & right आहे.



→ Now, in preorder, (20) नंतर (10) आहे म्हणजे (10) हा root बनेल. & (10) नंतर (15) दिले आहे so (15) root बनेल But in Inorder (15) हा (20) चा left मध्ये & (10) चा right ला आहे. so तो (10) चा right ला असेल.

→ same for (23, 25) block. i.e. (15) नंतर (25) येते in preorder & in inorder (23) is in right of (20) & left of (25).

postorder → 15, 10, 23, 25 20 35 42 39 30



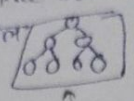
Pre – 100, 20, 10, 30, 200, 150, 300

In – 10, 20, 30, 100, 150, 200, 300

Post – 10, 30, 20, 150, 300, 200, 100

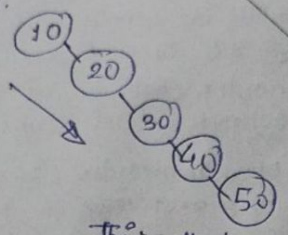
AVL Tree

Balanced BST

→ जर माझी tree balanced असेल म्हणजे त्याचा node चे proper 2 child. असेल  तर height of tree would be $(\log_2 n)$ पर्यंत जाईल. & that is the advantage of BST

time comp = $O(\log n)$

→ But there is problem in BST i.e. if we have (10 20 30 40 50) elements. so it acts like Linked List instead of Tree. like this.

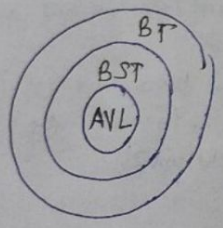


This called Right skewed

no. of elements = n

so then height = n

and search करण्यासाठी time complexity = $O(n)$ होईल.

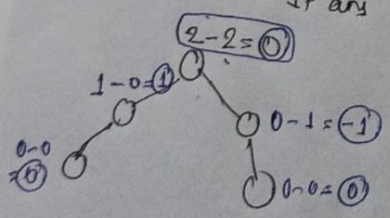


→ जर balanced शाखेले ~~ल~~ tree just like this तर height $\log(n)$ शाखेल & त्यामुळे time complexity पण $O(\log n)$ होईल.

→ AVL चा Time complexity

Balanced Factor = (Height of left sub tree − height of Right subtree)

→ If ans is $(-1, 0, +1)$ then your tree would be balanced



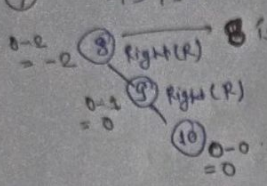
→ all the values in between $(-1, 0, +1)$ so it is balanced Tree i.e. AVL Tree

AVL Tree

55, 25, 65, 9, 8, 15

Suppose →

① 8, 9, 10



8 is imbalanced → so we have to balance the 8 node so we have to rotate the 8 node to anti-clock wise.

RR

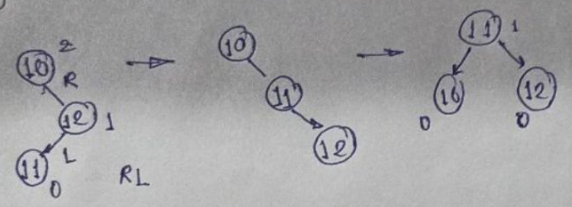
② (left) L 10 2-0 = 2 → clockwise rotate → so we have to balance the 10 node, so we have to rotate it clockwise

L 9 1-0 = 1

8 0-0 = 0

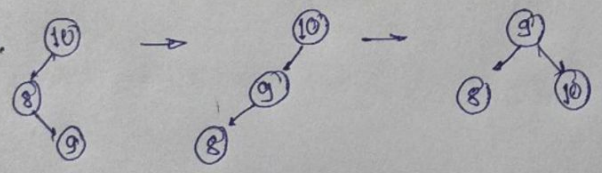
LL

3.



RL

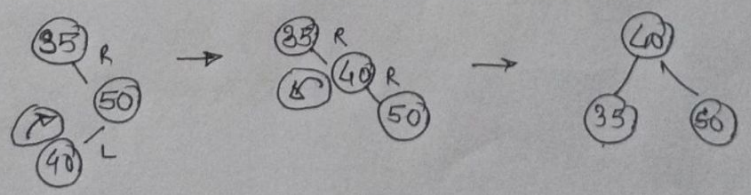
4.

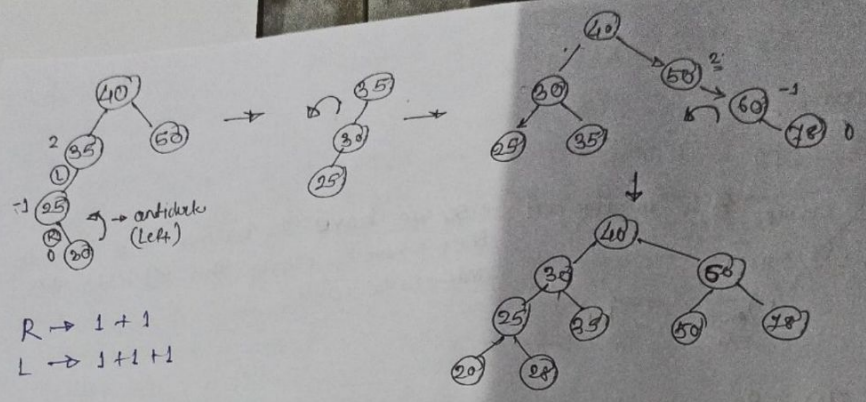


① How many rotations are required during construction of an AVL Tree if the following elements are added in order given

35, 50, 40, 25, 30, 60, 78, 20, 28.

R = 1
L = 1





R → 1 + 1

L → 1 + 1 + 1

27/4/22

Stack

Operations

→ Push = Insert an element

Deletion (pop) = Remove an element

isEmpty () = stack is empty or not

isFull () = stack is full or not

peek () = return current position of TOP

count () = count an total number of elements in stack

change () = change the position of an element

display () = print the elements of stack

5 — 73
56 — 58
47
3
2
1

Array Implementation of stack

Push Operation

```
int s[] = new int [ ]
int top ;

boolean isEmpty ()
{
    return (top < 0);
}
boolean push (int x)
{ if (top >= n-1)
  { SOP ("OverFlow");
    return false ;
  }
else
{
    s [++top] = x
    (or)
    top = top + 1
    SOP (x + " Push Operation");
    return true ;
}
```

Pop Operation

```
int pop ()
{
  if (top < 0)
  {
    SOP ("UnderFlow");
    return 0 ;
  }
  else
  {
    int x = s [top--] ;
    return x ;
  }
```

A + B * C / D - E ^ F * G.

A + B * C / D - EF^ * G

A + BC* / D - EF^G *

A + BC*D / - EF^G*

ABC*D/+ - EF^G*

ABC*D/+EF^G*-

---

7 4 -3 * 1 5 + / *

Char	Stack	O/p
7	7	
4	7 4	
-3	7 4 (-3)	
*	7 (-12)	
1	7 (-12) 1	
5	7 (-12) 1 5	
+	7 (-12) 6	
/	7 (-2)	
*	$\boxed{-14}$	

Stack Using Array

① Underflow condition →

~~void display~~

```
void underflow ()
{
    if (top == -1)
        sop ("Underflow");
}
```

② Overflow condition →

```
void overflow ()
{
    if (top == (n-1))
        sop ("Overflow");
}
```

③ Push →

```
void push ()
{
    overflow ();
    sop ("Enter data");
    int i = sc.nextInt ();
    top = top + 1;
    a[top] = i;
}
```

top नेहमी index number ला represent करते.

④ Pop →

```
void pop ()
{
    underflow ();
    top = top - 1;
}
```

Time Complexity

Best case

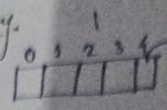
Push () → O(1) → push & pop are posible only at top so आपल्याला

pop () → O(1) → top करूनच हे करावे लागेल so त्याची O(1) असेल

display () → O(n) → display करतांना आपल्याला संपूर्ण elements ला traverse करावे लागेल so O(n).

Time complexity.

Worst case

Push() → O(n) → 

→ जर इथे push करायचा असेल तर traverse to 4th index. म्हणून O(n)

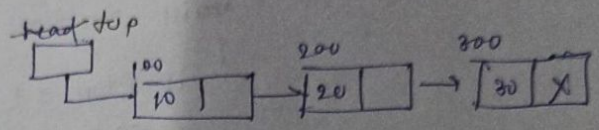
Pop() → O(n) →

→ जर 10 delete करायचा असेल तर पहिले 40, 30, 20 delete करावे लागेल म्हणून O(n)

display() → O(n)

* Stack Using L.L.

① Underflow →




```
if (head == null)
{
    sop("Underflow");
}
```

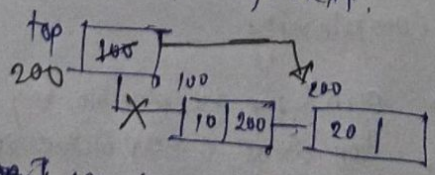
* Stack Using L.L. used करत असेल तर overflow condition येत नाही. : Using L.L. करत असेल तर size defined करायची गरज नाही.

② Overflow

③ Push() → add element आपण at beginning करायचा असेल तर L.L. best आहे. म्हणजे at end, at specific position नाही करता येत.

④ ~~del~~ pop() → deletion from beginning

* top = top.next.



If ~~top~~ I want to delete 10' node so first I ~~wa~~ have to disconnect it from top pointer & I ~~dat~~ disconned with by giving top pointer's link to the 20 node by making statement top = top.next.

Application of Stack

a+b → Infix → Infix expr. ला जास्त time लागतो coz त्यात brackets असतात so तो दुसरा prefix or postfix मध्ये convert करतो पहिले मग evaluate करतो

+ab → Prefix (Polish)

ab+ → Postfix (Reverse Polish)

Infix to Prefix

1) a + (b+c * (d + l)) + f / g)

1st step → put opening & closing bracket to the expression

(a + (b+c * (d + l)) + f / g)

Symbol	Stack	Expression
(	(	-
a	(	a
+	(+	a
(	(+(	a
b	(+(	ab
+	(+(+	ab
c	(+(+	abc
*	(+(+*	abc
(	(+(+*(	abc
d	(+(+*(	abcd
+	(+(+*(+	abcd
l	(+(+*(+	abcdl
)	(+(+*(+)	abcdl+
)	(+(+*) ① ②	abcdl+*+
+	(+ +	abcdl+*++
f	(+	abcdl+*++f
/	(+/	abcdl+*++f
g	(+/	abcdl+*++fg
)	(+/) ① ②	abcdl+*++fg/+

priority same आहे so first symbol pop होईल

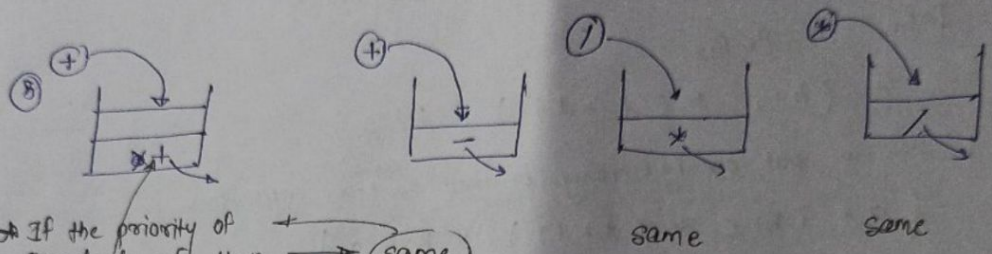
या part delete होईल stack वरून

'+' ची priority less आहे '/' च्या comparison so pop नाही.

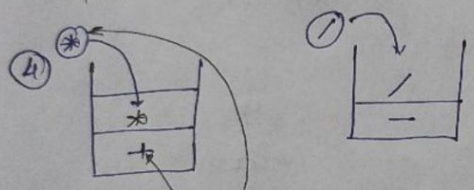
Rules

① ( ) → add bracket to the expr.

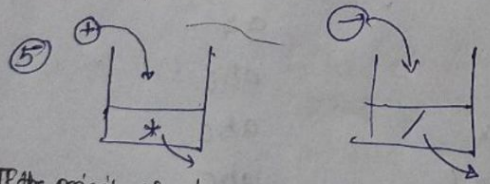
② /operators/ → push to stack || /operands/ → directly add to expr. or print



* If the priority of operator is same then the 1st operator should pop out → same



* If the priority of 2nd operator is higher than 1st operator then no need to pop out.



If the priority of 2nd operator is less than 1st operator then the 1st operator pop out.

⑥ (_) → pop out होईल यामधील symbol चा opening & closing bracket असेलतर.

Queue

→ First In First Out (FIFO)

→ Operations → Enqueue (To Insert)
→ Dequeue (to delete)

	10	20	30	40	50	60
	0	1	2	3	4	5
F=-1	F=0					
R=-1	~~R=0~~	~~R=1~~	~~R=2~~	~~R=3~~	~~R=4~~	R=5

→ Enqueue $\boxed{R = R+1}$
Dequeue $\boxed{F = F+1}$

Queue → Deletion possible only in Front
→ Insertion possible only in Rear

→ जसे stack मधे insertion possible आहे only by $\boxed{\text{Top}}$ तसे Queue मधे Insertion possible only by $\boxed{\text{Rear}}$ →

→ so आता $\boxed{\text{Rear}}$ हा $\boxed{R=5}$ आहे so जर मला 40 at index 3 वर add करायचा असेल तर not possible coz R is at index 5.

→ same for delete operation that it possible only by front end. now $\boxed{F=0}$, when $\boxed{F=1}$ then value on index 0 will be delete.

→ When 1st value we put then ~~F=~~ Front & Rear both have at zero th index (~~Remember~~ Remember)

Queue Using Array

1) Underflow Condition →

F = -1 F = 0
R = -1 R = 0

```
if (F == -1 && R == -1)
{
    sop ("Underflow");
}
```

2) Overflow Condition →

N = 10 // ~~N = Q[]~~

```
if (R == (N-1))
{
    sop ("Overflow")
}
```

3) Enqueue →

```
void Enqueue ()
{
    if
    {
        Overflow ();
    }
}
else
{
    if (F == -1 && R == -1)
    {
        F = 0;
        R = 0;
    }
else
{
    R = R+1;
}
```

→ To insert only Rear should increase

4) Dequeue →

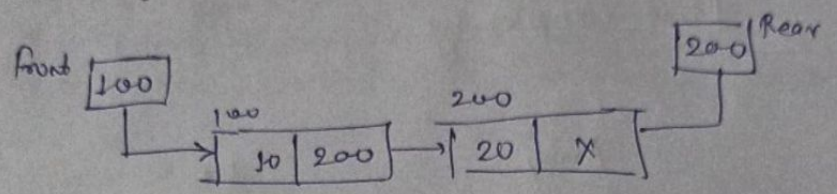
```
void dequeue ()
{
    if
    {
        ~~overflow~~
        underflow ();
    }
else
{
    F = F+1;
}
```

5. Display()

```
for (i=F ; i<R ; i++)
{
    sop (Q[i]) ;
```

— x —

Queue using L.L.



① Underflow → If there is underflow condition then we don't have delete any node coz there is no node

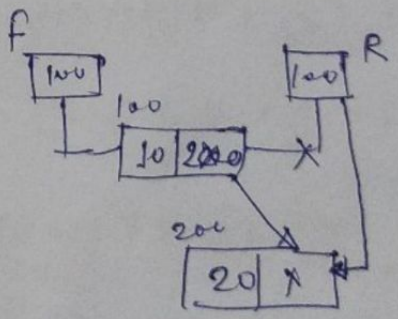
② Overflow → No overflow condition in L.L

```
if (f == null && r == null)
{
    sop( Underflow )
}
```

```
if (new_node == null)
{
    sop (Overflow)
```

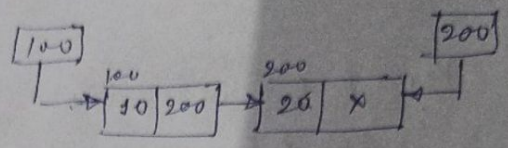
→ means there is no space in memory to new node also

③ Enqueue →




```
if (f == null)
{
    f = new_node ;
    R = new_node ;
}
else
{
    R.next = new_node ;
    R = new_node ;
}
```

Dequeue →




```
else
{
  f = f.next ;
}
```

Time complexity

	Array	LL
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Display	O(n)	O(n)

Circular Queue.

→ In linear queue, if there 10 blocks & only 10th box have value but all the 9th block value get deleted by front end & if you want to add / Insert some value again so it is not possible even there is ~~sp~~ 9 space available. Because Insertion ~~add~~ operation happend only in one direction

→ That's why we used $\boxed{\text{Circular queue}}$, when we used linear queue using array, but when we used linear queue using L.L. then we don't need to used $\boxed{\text{circular queue}}$ because L.L. don't have limited size, it increasy as per the need.

In Linear Queue

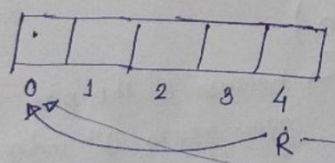
↳ Enqueue = R += R + 1

Dequeue = F = F + 1

In Circular Queue

↳ Enqueue → R = (R + 1) % maxSize

Dequeue → F = (F + 1) % maxsize.



→ R = (0 + 1) % 5 = 1 → Index

R = (1 + 1) % 5 = 2 → Index

R = (4 + 1) % 5 = 0 → Index

↳ It again go back to the 0th index and add element

F = (2 + 1) % 5 = 3

Linear

Overflow → if (R == (maxsize -1))

Underflow → if (F == -1 && R == -1)

Circular

if (F == (R+1)) % maxsize

if (F == -1 && R == -1 || F == R)

Enqueue →

```
if {
    overflow();
}
else {
    R = (R+1) % max size
```

circular queue

Dequeue →

```
if {
    underflow();
}
else {
    F = (F+1) % maxsize
```

0	1	2	3	4	5
10	20	30	40	50	60

R=3 F=4

```
for (i = F; i != R; i = (i + 1) % n)
{
    Sop (q[i]);
```

because of this part it again go to 0th index from 5th index.

Sorting

① Internal sorting

Given array : 2 5 8 9 4 7 2

Sorted array : 2 2 4 5 7 8 9 — Stable sorting

Given array → 2 5 8 9 4 7 2

Sorted array → 2 2 4 5 7 8 9 — unstable sorting

Stable sorting

→ after sorting the contents, if elements does not change the sequence of similar content

Unstable sorting

→ ——— " ——— does change — "

——— " ———

Efficiency of Algorithm:

→ Running time complexity of time & space

→ best-case, avg case, & worst case.

Bubble Sort

→ optimised Bubble sort असेल तर $\boxed{O(n)}$ best case येईल, but normal bubble sort असेल तर best case पण $\boxed{O(n^2)}$ येईल.

Bubble Sort →

0	1	2	3	4			
32	51	27	85	65	23	13	57

1) a[0] > a[1]

→ If this condition is [true] then no need to swap the index but if it is false cond compare the next numbers but if it is [false] then swap it. i.e goto ② step

2) a[0] ↔ a[1]

swap ←

Step 1.

32 51 27 85 65 23 13 57

32 27 51 85 65 23 13 57

32 27 51 85 65 23 13 57

32 27 51 65 85 23 13 57

32 27 51 65 23 85 13 57

32 27 51 65 23 13 85 57

32 27 51 65 23 13 57 85. → largest element comes at last in the 1st step.

⑦ step

Step 2. ⑥
Step 3 - ⑤
Step 4. - ④
Step 5 - ③
Step 6 → ②
Step 7 : ① → There are 8 elements so we should have ⑦ step for the bubble sort

→ एक-एक step कमी होत जात आहे म्हणजे step-1 मध्ये ⑦ steps मध्ये 85 value मिळते step 2 मध्ये ⑥ steps मध्ये 65 मिळत म्हणून next step मध्ये last element compare करायची गरज नाही. म्हणून [Ptr ≤ N-K] करतो

[(n-1)]

⑦ →

array name

Bubble (Data , N) → no. of elements

① Repeat steps 2 & 3 for k = 1 to N-1

② Set Ptr = 1

③ Repeat while Ptr ≤ N - K

ⓐ If Data[Ptr] > Data[Ptr+1] then Interchange Data[Ptr] and Data[Ptr+1]

ⓑ set Ptr = Ptr + 1

```
for (i=0; i<n ; i++)
{
 a[i] = sc.nextInt();
}
for (i=0; i<(n-1) ; i++)
{
 for (k=0; k<(n-i-1); k++)
 {
  if (a[k] > a[k+1])
  {
   temp = a[k];
   a[k] = a[k+1];
   a[k+1] = temp;
  }
}}
```

→ This Suppose, we have 5 elements & start from 0 so we have 4 passes in this For loop for this (4 passes)

→ Internal comparision साठी.

suppose we have [10, 7, 6, 5, 4] this five elements, so because of 2nd for loop [n = 5] and [i = 0 to 3] iterate करेल.

→ 10 7 6 5 4.

index: 0th 1st

[k=0] साठी [10 & 7] ला compare करेल.

[K=1] साठी 1st and 2nd element index ला compare.

[K=2] साठी 2nd & 3rd index ला compare.

[K=3] साठी 3rd & 4th. ला compare.

इथे असा last element ला compare करणार नाही cuz तो already compare करून टाकला आहे

i = 0	k < (n-i-1)
k = 0	5 - 0 - 1 = 4
k = 1	k < 4
k = 2	
k = 3 → k = 3	
i = 1.	k < (n-i-1)
k = 0	5 - 1 - 1
k = 1	k < 3
k = 2	
i = 2.	k < (n-i-1)
k = 0	5 - 2 - 1
k = 1	k < 2
i = 3	k < (n-i-1)
i = 0	5 - 3 - 1
	k < 1

Time Complexity

Worst case.

$T(n) \rightarrow (n-1) + (n-2) + (n-3) + \dots 1.$

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

$T(n) = O(n^2)$ → worst case. → this happend when elements are in descending order & we have to sort in ascending order.

$T(n) \rightarrow O(n) \rightarrow$ Best case → If elements are already sorted.

Space complexity

$O(1) \rightarrow$ we have to used only temp variable असून required not

Selection Sort

77 33 44 11 80 22 66 55

1st pass.

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
at location 3 we have min value								
k=1 j=3	77	33	44	11	88	22	66	55
k=2 j=5	11	33	44	77	88	22	66	55
k=3 j=5	11	22	44	77	88	33	66	55
k=4 j=5	11	22	33	77	88	~~33~~ 44	66	55
k=5 j=7	11	22	33	44	88	77	66	55
k=6 j=6	11	22	33	44	55	77	66	88
	11	22	33	44	55	66	77	88

→ The smallest element comes at 1st position & then sorted in increasing order.

Code

```
for (i=0 ; i<n ; i++)
{
 a[i] = sc.nextInt();
}
for (i=0 ; i<n-1 ; i++)
{   loc = i;
 for (j = i+1 ; j<n ; j++)
 {  if (a[j] < a[loc])
    {
      loc = j;
    }
 }
 temp = a[i]
 a[i] = a[loc]
 a[loc] = temp;
}
for (i=0; i<n ; i++)
{ sop (a[i]);
}
```

→ A[7] साठी आपण check केल नाही म्हणून (n-1) घेतले.

→ this 'j' loop used for getting unsorted array into sorted array, जेव्हा आपण 1st pass मध्ये A[0] i.e. (77) साठी बाकीचे elements check करत होतो तेव्हा ते [A[1]] पासून check करत आहे i.e. [i+1] पासून check करणार म्हणून [j = i+1] करतो & (33) check करायचा असेल तर next element पासून check करतो म्हणून [j = i+1] करतो.

→ आपल्याला location of minimum element find out करायचं आहे. & initially we consider as [0th] index वर जो element आहे तो minimum element आहे.

→ location of [j = i+1] & location of [loc = i] means [loc = 77] & it check with other elements by this 'if' condition.

Time complexity.

→ pass 1 मधे (n-1) comparision आले pass 2 मधे (n-2) comparision आले. so we get this → $(n-1)+(n-2)+(n-3) \ldots + 1$

$= \frac{n(n-1)}{2} = \boxed{O(n^2)}$ → Best case & worst case both have T.C. $\boxed{O(n^2)}$

Best case → जेव्हा data already sorted आहे

Worst case → जेव्हा data हा descending order मधे आहे ← व्यवस्थित ascending order मधे लावायचा आहे.

Insertion Sort

0	1	2	3	4	5	6
25	15	30	9	99	20	26

→ In this sort the we start with a[1] means 2nd value पासून start करायचा आहे. & compare it with previous value like this

step 1 → temp = 15

a[1] < a[0] → If this statement is false तेव्हा त्याच्या पुढे जायचा नाही.

15 25 30 9 99 20 26

step 2 → temp = 30

a[2] < a[1] → It gets false so पुढे जाणार नाही.

step 3 → temp = 9

temp < a[2] → It true तर मग पुढे जाणार

temp < a[1]

temp < a[0]

15 9 9 30

~~15~~ ~~25~~ ~~30~~ ~~99~~ 99 20 26

9 15 25

9 15 25 30 99 20 26

step 4 → temp = 99

temp < a[3]

step 5 → temp = 20

temp < a[4]

< a[3]

< a[2]

< a[1]

9 15 20 25 30 99 26

9 15 20 25 ~~30~~ 26 → 26 99 ← 26 99 → 26

temp = 26

temp < a[5]
temp < a[4]
temp < a[3]

Insertion_Sort (A[max], item)

① set k = 1 → index → because with started from index 1.

② for k = 1 to (n-1)

③ set temp = a[k]

④ set j = k - 1 → previous value compare करायची आहे so म्हणून $\boxed{k-1}$

⑤ while temp < a[j] and (j >= 0)

⑥ set a[j+1] = a[j]
j = j - 1 :

⑦ A[j+1] = temp

⑧ Exit.

0 नंतर -1 मध्ये j ची value नाही गेली पाहिजे म्हणून ही condition put केली coz आपण previous index compare करत ~~आहोत~~ करत मागे मागे येतो

इथे $\boxed{j = -1}$ आले तर तो while loop run बाहेर येईल & ⑦ number ची line i.e. A[j+1] = temp म्हणजे A[-1+1] = temp
A[0] = temp → म्हणजे A[0] वर value put करणार

Quick Sort

divide conquer method

7	2	1	6	8	5	3	4

pivot

pivot ek sth left side me smaller value & right side me greater value.

p i

(P) Pindex → index

Pivot → element / value

start → index

end → index

Iteration - 1

Step-1 → P=0 i=0 Pivot = 4

```
if (A[i] < Pivot)
{
}
```

if it is false then go to else

```
else
{
  i++;
}
```

Step-2 → P=0 i=1 Pivot = 4

```
if (A[i] < Pivot)
{ swap (A[p], A[i]);
  p++, i++;
}
```

→ 2 7 1 6 8 5 3 4

p i

Step 3 → A[i] < Pivot

{ same as **step 2** }

→ 2 1 7 6 8 5 3 4

Step 4 → A[i] < pivot false → else { i++; }

Step 5 → 8 < 4 → false → i++

Step 6 → 5 < 4 → false → i++

Step 7 → 3 < 4 → | 2 | 1 | 3 | 6 | 8 | 5 | 7 | 4 |

swap (A[p], pivot)

| 2 | 1 | 3 | 4 | 8 | 5 | 7 | 6 |

we get 4 as final index

Iteration 2

0	1	2
2	1	3

pivot

step 1 → A[i] < pivot

swap (2, 2)

step 2 → 1 < 3

swap (1, 1)

swap (A[p], pivot)

2	1	3

→ Index of 2 is fixed for 3

Iteration 3

2	1

pivot, end

i ची value increase तेवढाच नाही, coz (end - 1) पर्यंतच (1) जाऊ शकते

step 1 → 2 < 1.

swap (A[p], Pivot)

1	2

Index 0 for 1 is fixed

end index 1 is seen remaining so it is also fixed for 2

Iteration 4

8	5	7	6

pivot

step 1 → 8 ≮ 6 → i++

step 2 → 5 < 6

swap (8, 5)

p++, i++

5	8	7	6

end

step 3 → 7 ≮ 6 i++ → एण 1 ने आता i++ होणार नाही coz i ची value (end - 1) पर्यंतच राहते

swap (A[p], pivot)

swap (8, 6)

→ Iteration complete आल की ही step used करायची

5	6	7	8

Iteration 5

7	8

7 < 8

p++ i++

swap (A[p], pivot)

→ swap होणार नाही coz [p] & pivot सारखा elements ला point करत आहे.

Quick sort

```
Quicksort (A , start , end)
{
   if (start < end)
   {
     Pindex ← Partition (A , start , end)
     Quicksort (A , start , Pindex - 1)
     Quicksort (A , Pindex + 1 , end)
   }
}
Partition (A , start , end)
{
   Pivot ← A [end]
   Pindex ← start
   for i ← start to (end - 1)
   { if (A [i] <= Pivot)
     {swap (A[i] , A [Pindex])
      Pindex ← Pindex + 1
     }
   }
   swap (A [Pindex] , A [end])
   return Pindex.
}
```

array

Best case → O (nlogn)

Avg case → O (nlogn)

worst case → ~~O(n^2)~~ O(n^2)