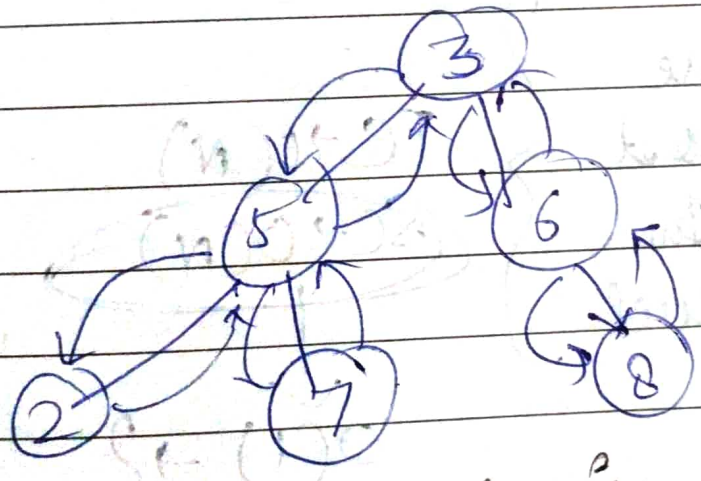
Inorder predecessor -



Inorder: LNR

2 5 7 3 6 8

I.P. of 3 :-
↳ left
↳ go to right
till ! NULL

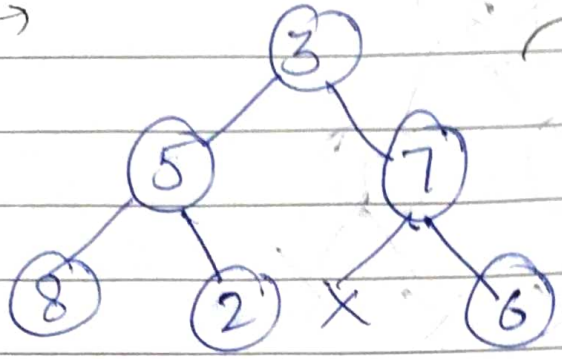
brute force → store inorder in an array & print previous

optimised → using recursion

RANKA
DATE / /
PAGE

Inorder Successor -

BF →

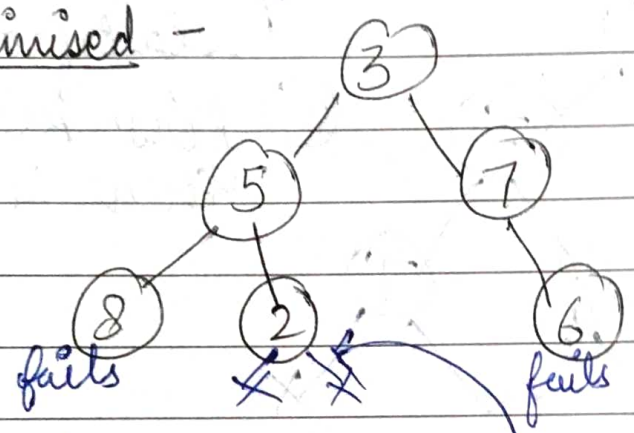


→ 8 5 2 3 7 6

IS(2)

↳ 3

Optimised -



① curr → right
② jab tak curr → left != NULL move in curr→left

→ but fails because agar 2 ka successor nikaalna h then there's no right ?

✱ fails in case of leaf nodes.

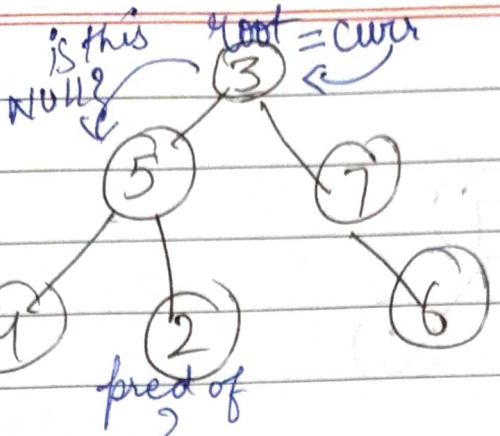
Tree Traversals → pre, post, In, level.

TC → O(N)
SC → O(N)

O(1) → ?

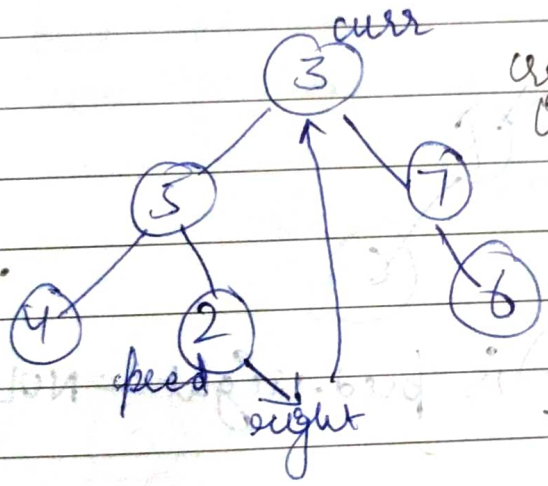
{Google}
Morris Traversal → O(1) → space ✓

Morris Traversal —



Inorder → L N R

1) if NULL
2) print root
3) move to right



creating link

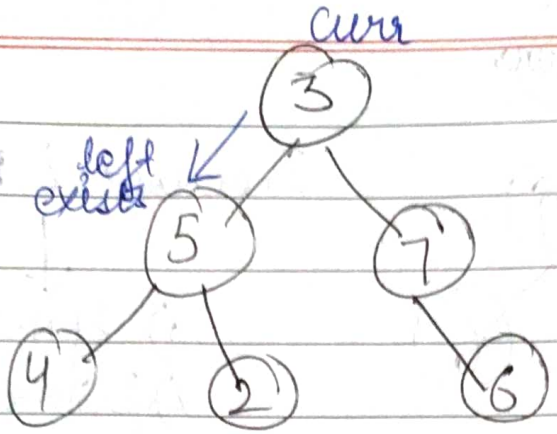
if left is not NULL
1) find pred of root (3)
2) create link i.e.
pred → right = curr.
curr = curr → left

~~but if there is no right~~
if right part is not NULL —
remove link.

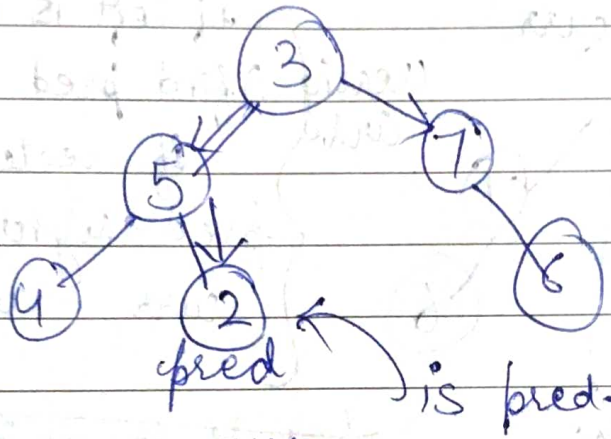
Algo —

① if (curr → left == NULL)
{ }

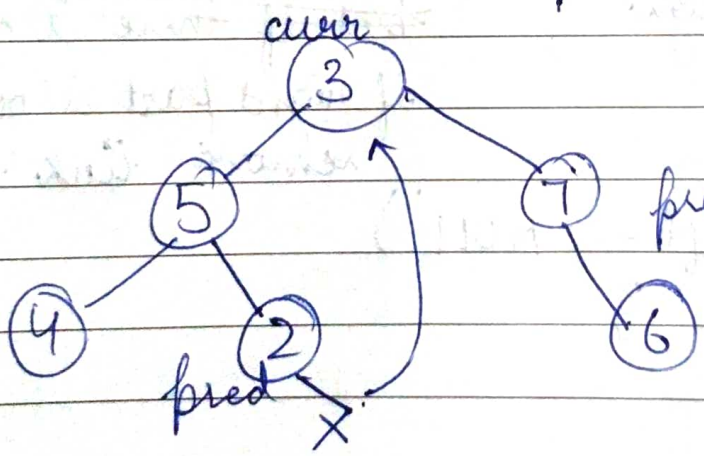
② else
// pred
// link create
or
// link remove.



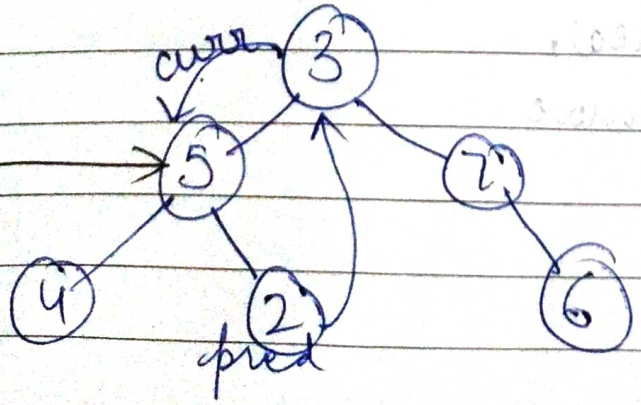
find pred of (3) —



is pred->right == NULL ?

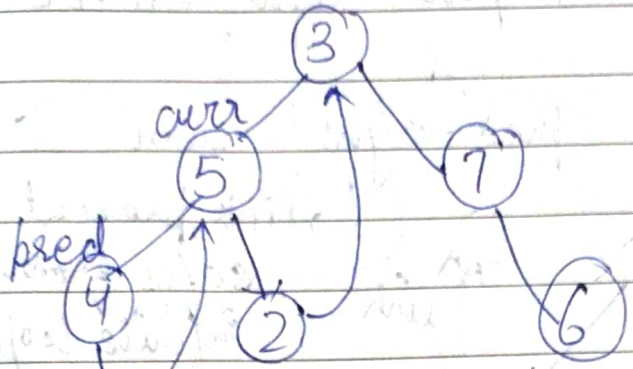


pred->right = curr



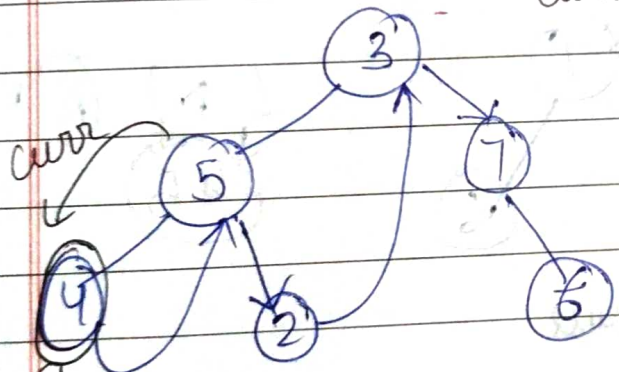
curr = curr->left

find pred of 5.

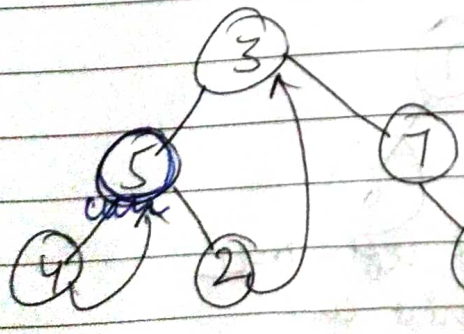


pred→right == NULL
then pred→right = curr

and
curr = curr→left



check if curr→left == NULL
then { print curr→data;
curr = curr→right;
}



4 5 2 3
↓
print

curr = curr→right se
vapas 5 par agye

pred cond's -

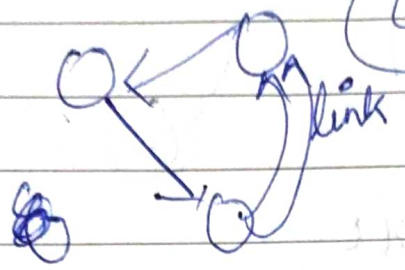
↳ pred → right != NULL

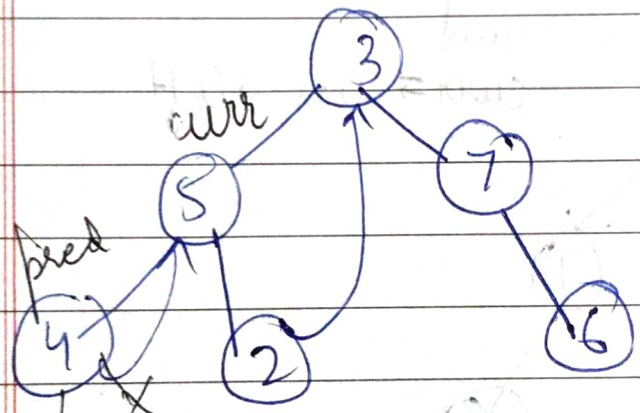
&&

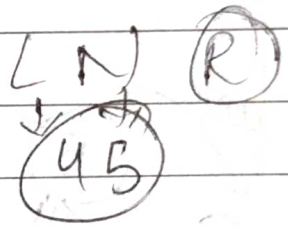
pred → right != curr

↳ link present

safes from infinite loop







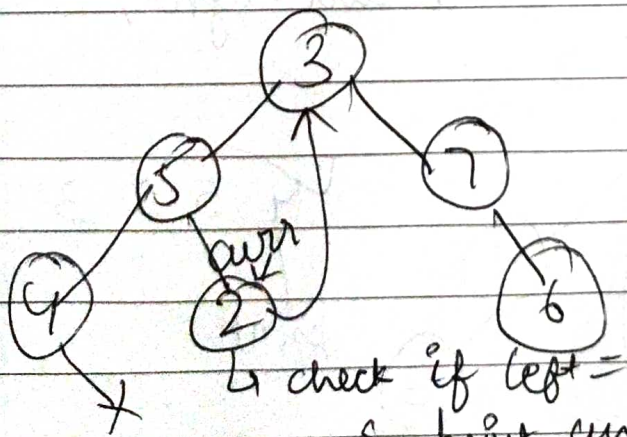
↳ right is not null

then

pred → right = NULL

print curr

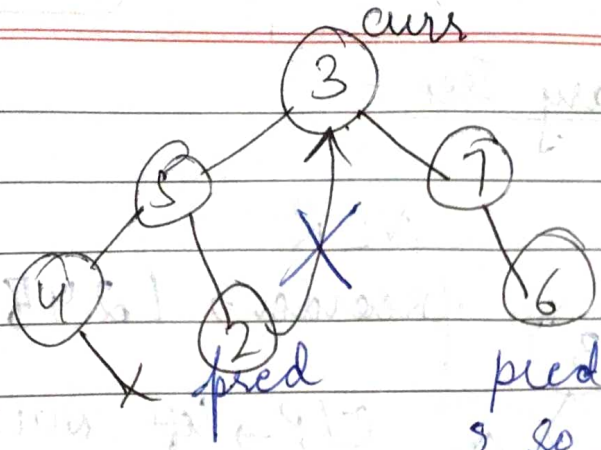
curr = curr → right



↳ check if left = NULL

& print curr

& move to right

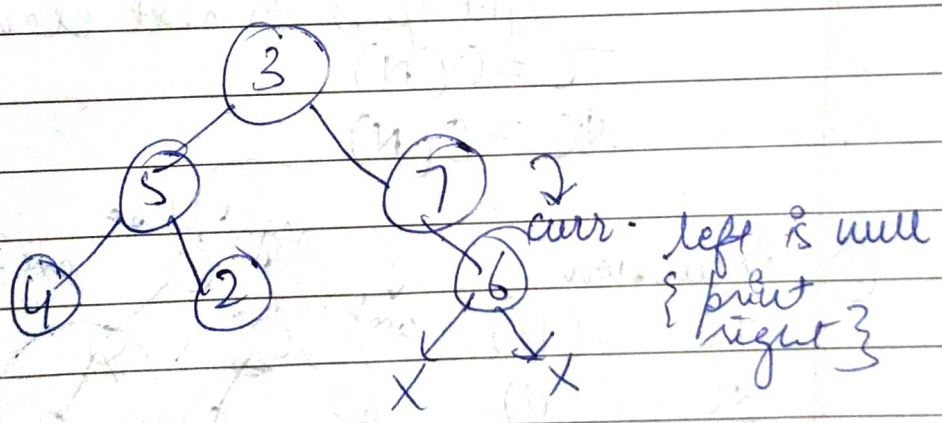


curr->left != NULL
↳ pred = (2)

pred->right != NULL
{ so remove thread.
print curr
curr = curr -> right }

printed ↳ 4 5 2 3

7 -> left = NULL
{ print it
move to right }.

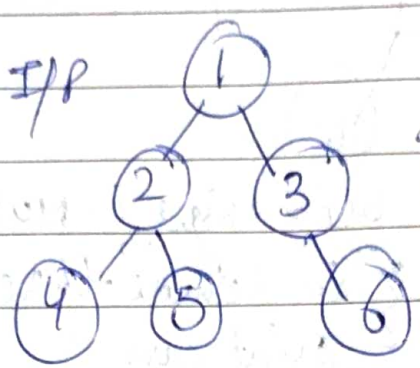


printed 4 5 2 3 7 6

* Stops when curr points to NULL.

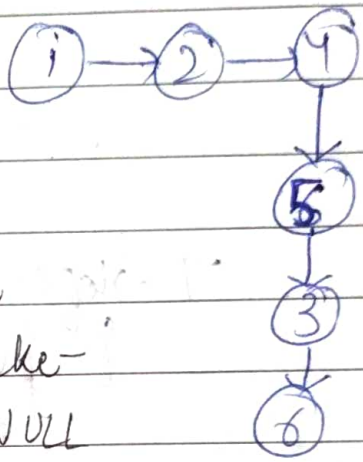
Why we create link?
Ie N because in iteration we can't go back like recursion
∴ we create threads / link to go back.

Flatten a Binary Tree -



NLR
preorder → 1 2 4 5 3 6

O/P → left = NULL
next ↔ right

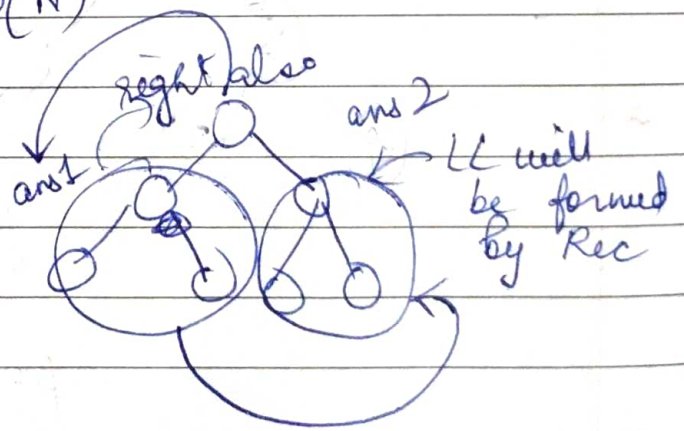


Approaches -

(1) preorder → store in vector
↳ traverse karke -
left → NULL
right → point to next element

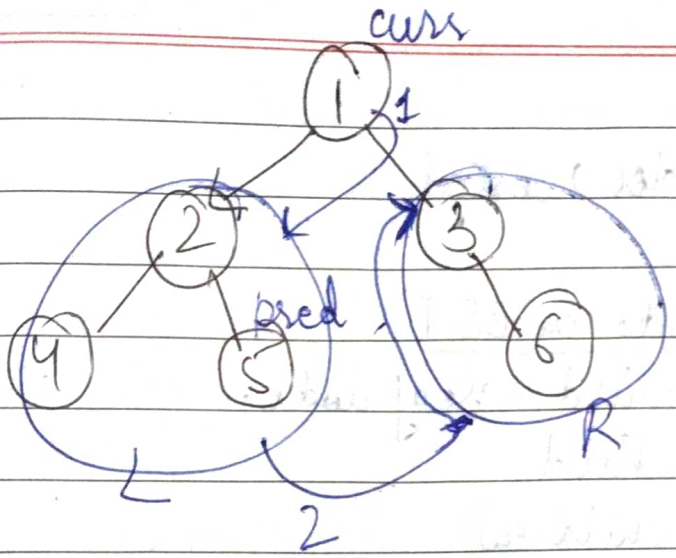
TC = O(N)
SC = O(N)

(2) Recursion
TC = O(N)
SC = O(N)



(3) Stack

(4) TC → O(N) SC → O(1). → Morris
Other traversals → SC → O(n). Traversal



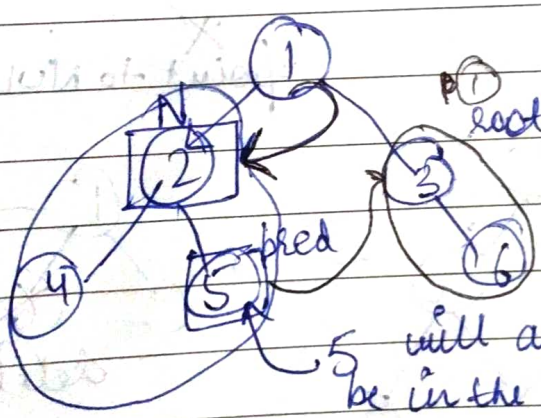
pred → right = curr → right

curr → right = curr → left

curr → left = NULL

first include root.

if left exists → LL will include it.



root → root → left always

5 will always be in the end of LL

and pred → right will connect to curr → right ie. 3

BST -

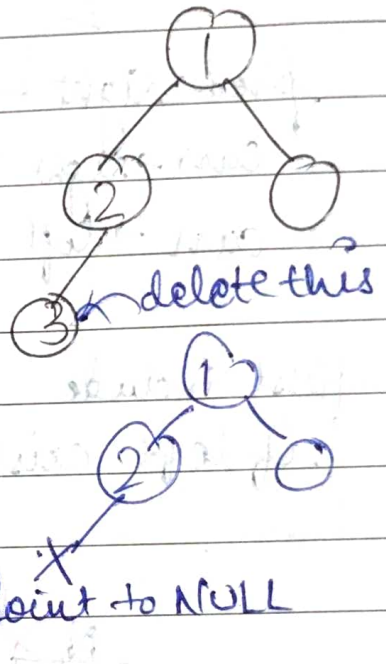
BST Inorder → sorted

Delete Node in BST

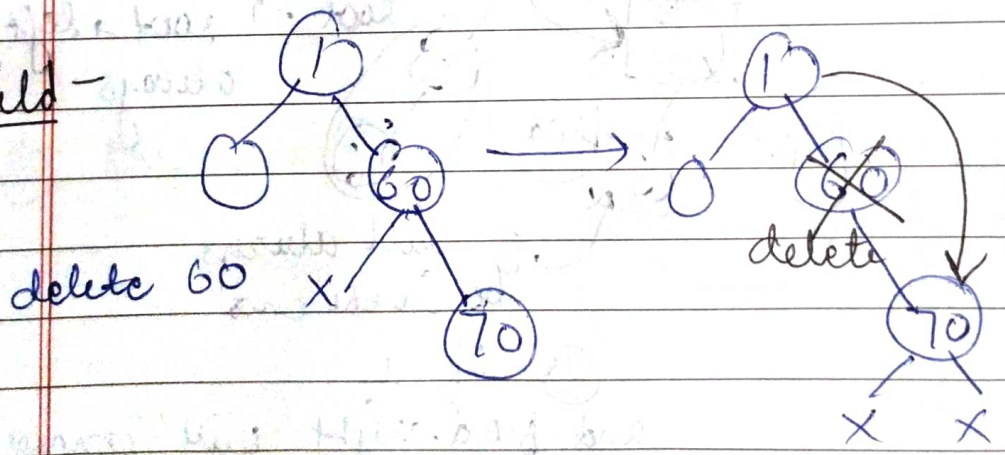
↳ 0 child → leaf node.
↳ 1 child
↳ 2 children

0 child - if (leafNode)

delete root;
return NULL.

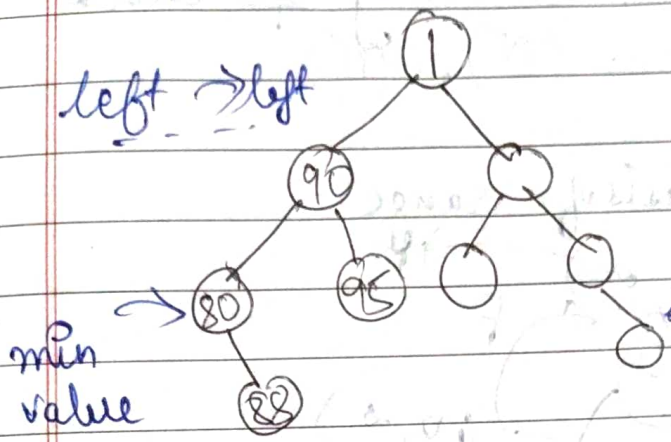


1 child -



if (1 child)
{
Node* temp = root → left/right
delete root;
return temp;
}

2 child – for this, we need concept of Min/Max in BST.

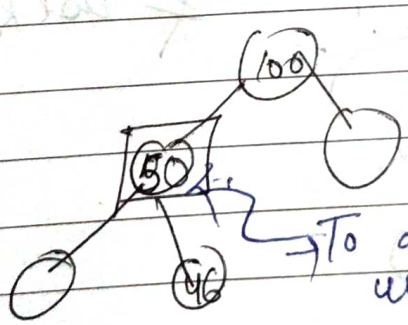



```
Node *temp = root;
while(temp->right != NULL)
{temp = temp->right;}
return temp;
}
```

Inorder pred/Successor logic → was not applicable to leaf nodes.
but node with 2 children can apply it.

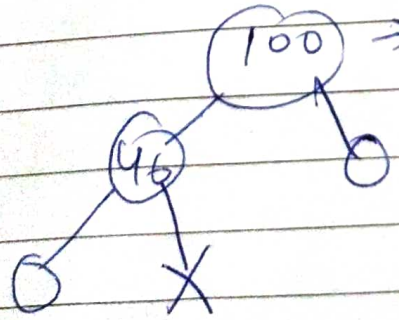
90 → IP → 88

90 → IS → 98



→ To delete this, replace this with IP.

→ delete that IP node



→ RC to delete

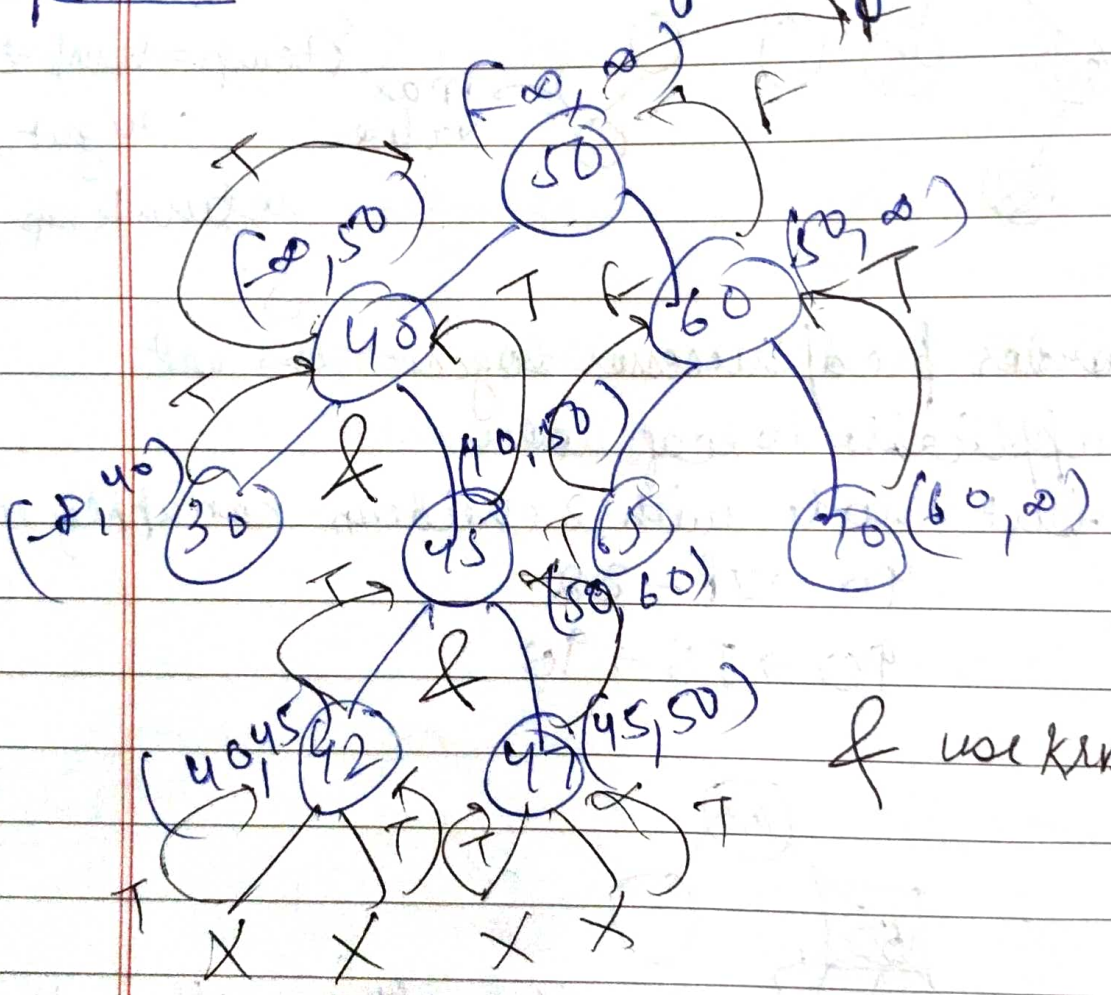
OR

→ find IS and same as IP.

check if tree is BST/not ?

BF → storing Inorder in array & check if its sorted. O(N), O(N)

Optimised – On the basis of range



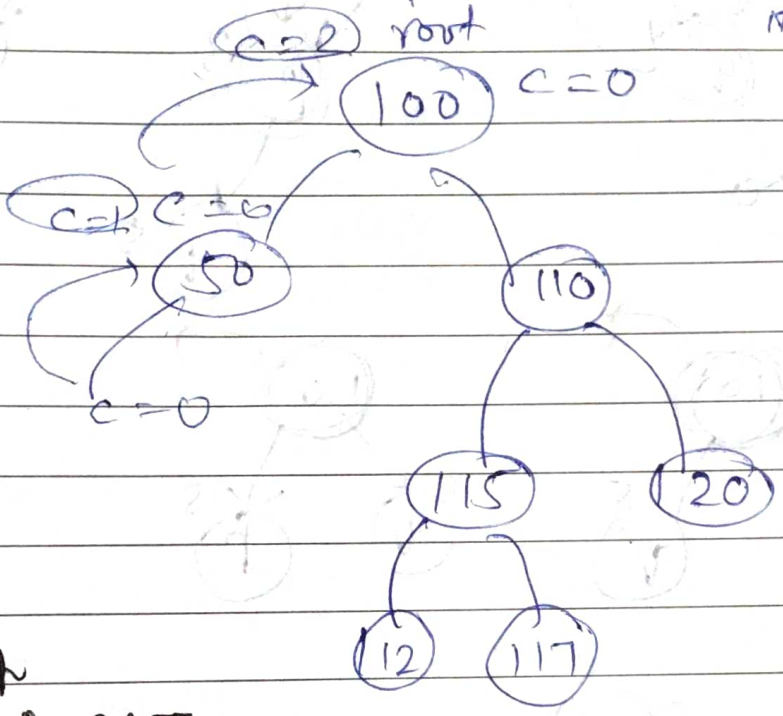
& vice versa

O(N), O(H)
TC SC

RANKA
DATE / /
PAGE

K^{th} Smallest

① Store inorder & print k^{th} value.
② Take a counter & decrement it until it matches k.



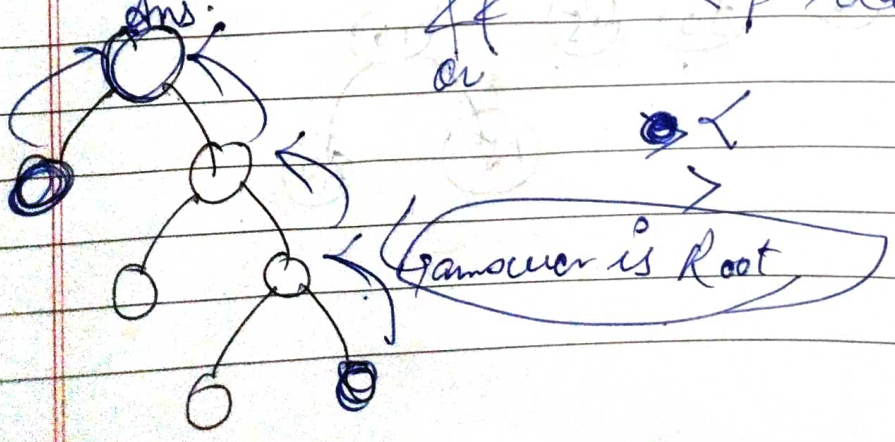
VVImp

LCA in BST.

Case I → both nodes on left i.e < root node.
(root → data > p → data && root → data > q → data)
{ left }.

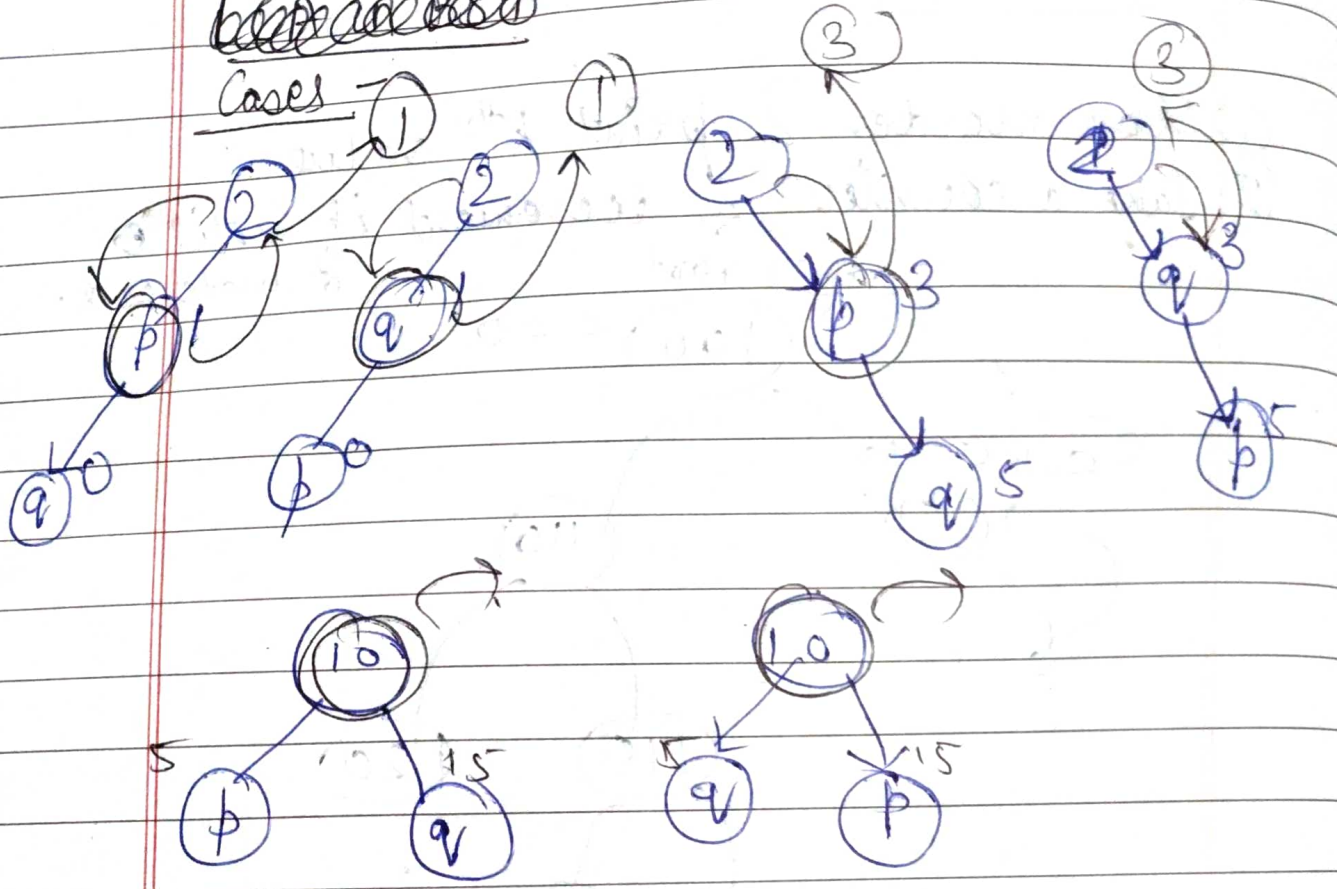
Case II – else if root data < ~~
right

Case III → Root → data > q → data
&& " " < p → data
or



~~Lowest Common Ancestor~~

Cases →



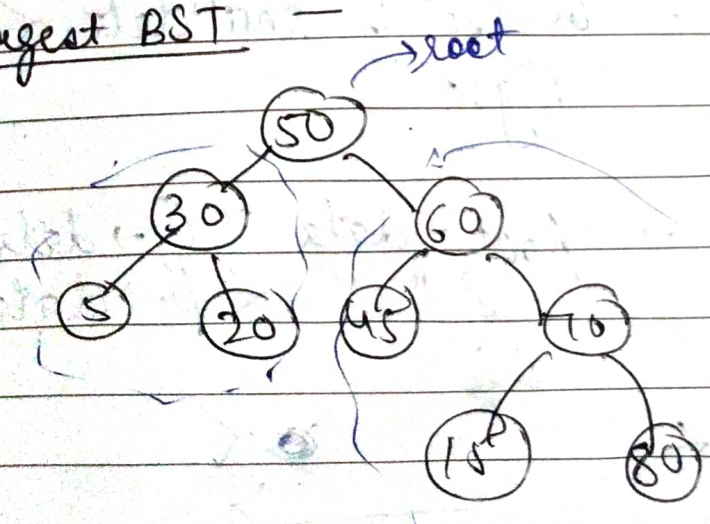
TC → O(N)

SC → O(N) → recursion

↳ O(1) → yes → using Iteration (loop)

UV Sub -

Largest BST —



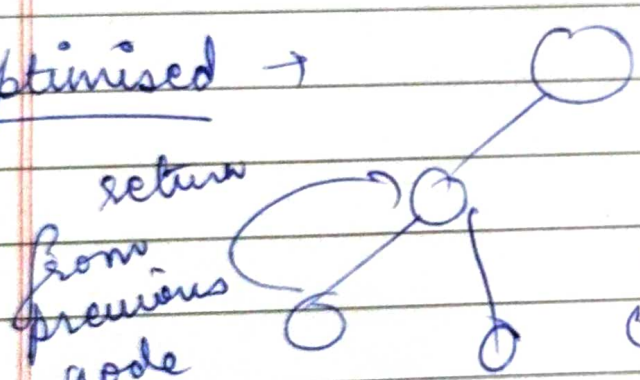
BF → check on every node whether it is largest BST or not?

every node → check BST

Yes (ans. update with size)

No → ignore

optimised → use pair.



checkBST —
- min
- max
- size
- IsBST