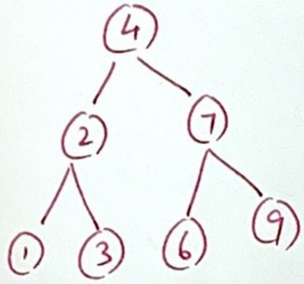
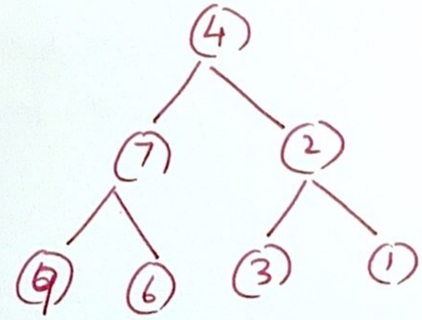
INVERT A BINARY TREE:-

Invert a Binary Tree is swapping The LEFT & RIGHT subtrees for every node in The Tree.



=>



This problem can be solved.

→ Iteratively

→ Recursively (TOP - DOWN)

→ Recursively (Bottom-up).

When SOLVING tree problems recursively, The point of view must be 1 single Node & what happens at That 1 Single Node at any given point.

Top Down Vs Bottom Up

This is more on where The operation happens.

TOP-DOWN:-

For any Given Node,

=> its subTrees are 1st swapped (child Nodes Swap)

Before They are added to call stack.

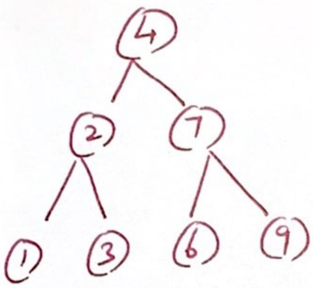
=> Then The Sub-trees are traversed to further process Them.

BOTTOM-UP:-

for any Given Node,

After They are added to The call stack.

=> Sub-Trees are Traversed & swapped 1st



Recursive

TOP - DOWN

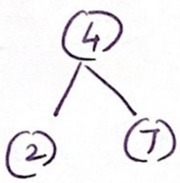
⇓

Child Nodes are Swapped Before pushing to Call Stack

By pushing into Call stack.

⇒ recursively calling Same function over The childNodes, so That The childNode's sub-Trees are also processed.

Take a single Node.



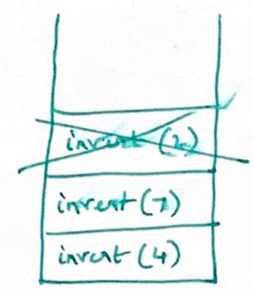
⇒ Swap Child Nodes

⇒ Recursively Call The invert function so That The Child Nodes (2), (7)'s SUBTREES are also inverted.

Note That child Nodes are swapped before The recursive call to invert Child Nodes SubTrees ⇒ Swap happens before The next recursive call is pushed into Call stack.

~~Call (1)~~

Call (1):- invert (4)



temp = 4.left = 2

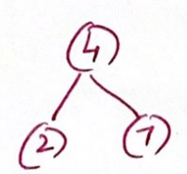
4.left = 2 = 4.right = 7

⇒ 4.left = 7

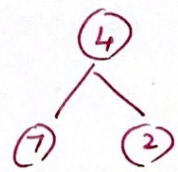
4.right = temp = 2.

∴ 4.left = 7

4.right = 2



⇒



Call invert (4.left) ⇒ invert (7)

invert (4.right) ⇒ invert (2)

Call (2) | invert (7)

temp = node.left = 7.left = 6

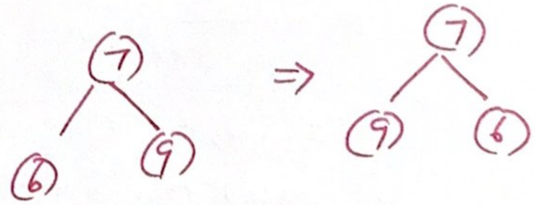
node.left = node.right =

= 7.right = 9

$\boxed{\text{node.left} = 9}$

node.right = temp = 6.

$\boxed{\text{node.right} = 6}$



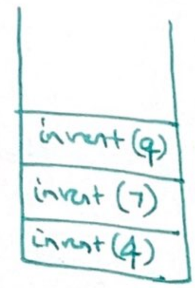
Call,

invert (node.left)

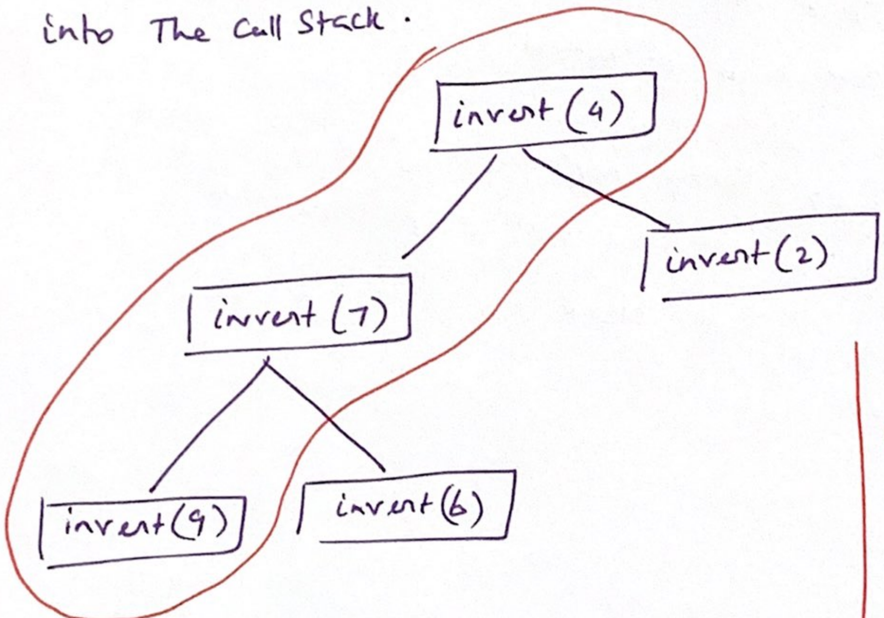
⇒ invert (9)

invert (node.right)

⇒ invert (6)



Note That The call invert (6) has not yet been pushed into The call stack.

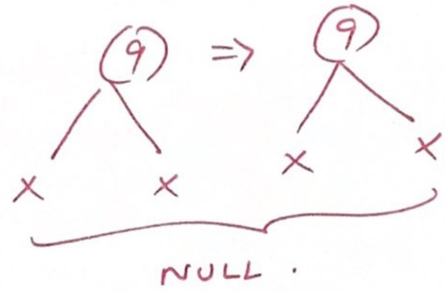


→ we are onto this path.

once (9) Completes, it will be popped out of call stack & Then invert (6) Be popped in.

In The Same way, untill invert (7) is popped out of call stack, invert (2) wont be pushed into call stack.

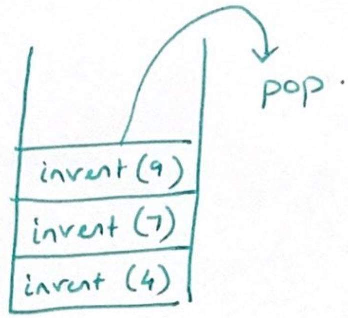
Call (3) | invert(9)



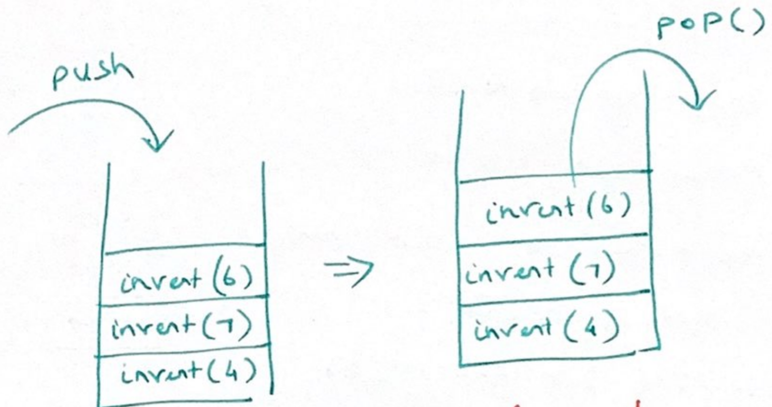
(9).left → NULL.

(9).right → NULL.

Makes No diff when swapped. This is actually The Base Case for recursion, where a NULL comes in & we just return.



Call (4) | invert(6)

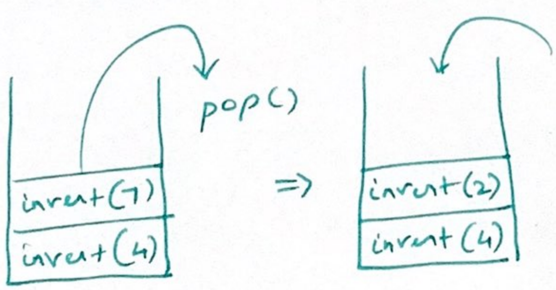


6.left → NULL

6.right → NULL.

Again Leaf Node => The Base Case of recursion is hit & we return.

Now The entry Comes back to invert (7). Since all its subtrees are inverted.



(7) also gets popped out of Call Stack & Control Comes to invert (4). which has its LEFT Completed So its Right Subtree invert(2) is now pushed into Stack.

Call (5) **invert (2)**

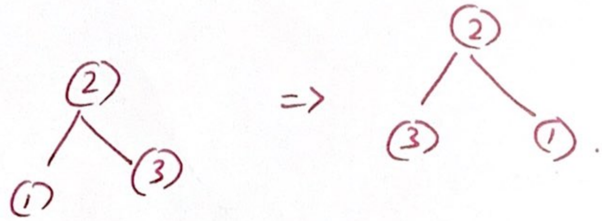
temp = node.left = 1

node.left = node.right = 3

node.right = temp = 1

∴ Node.left = 3

Node.right = 1



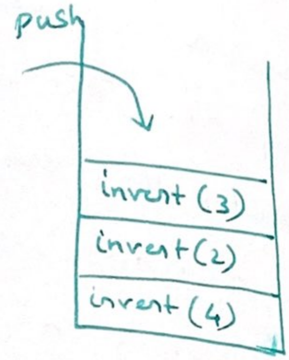
Call

invert (node.left)

=> invert(3)

invert (node.right)

=> invert(1).



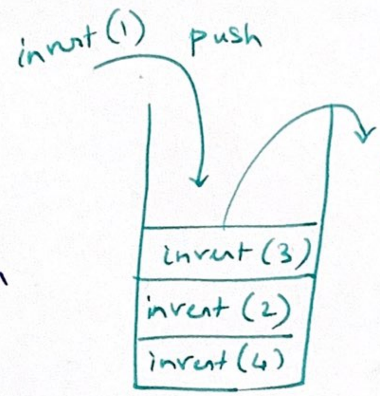
Call (6) invert (3)

node.left → NULL

node.right → NULL.

Base case of recursion is hit & returns.

So, The Call invert (3) is popped off The stack.



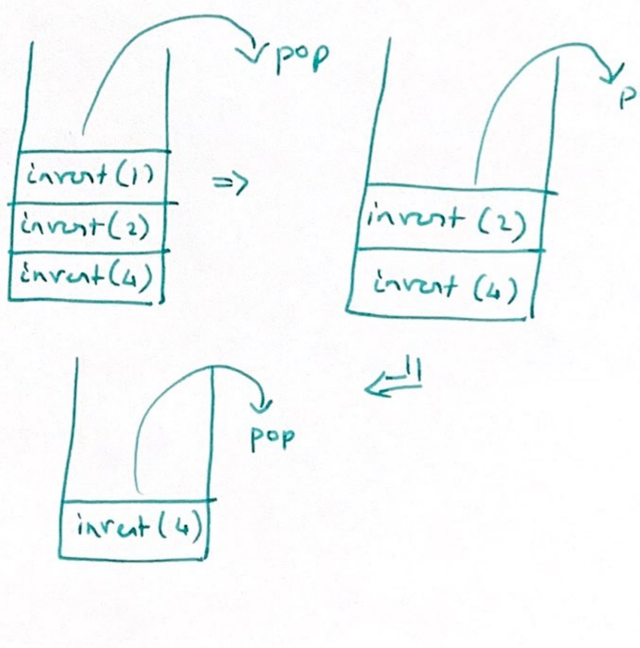
The control goes back to invert (2) & it begins to process The right by pushing invert(1) into Call Stack.

Call (7) invert (1)

node.left → NULL

node.right → NULL.

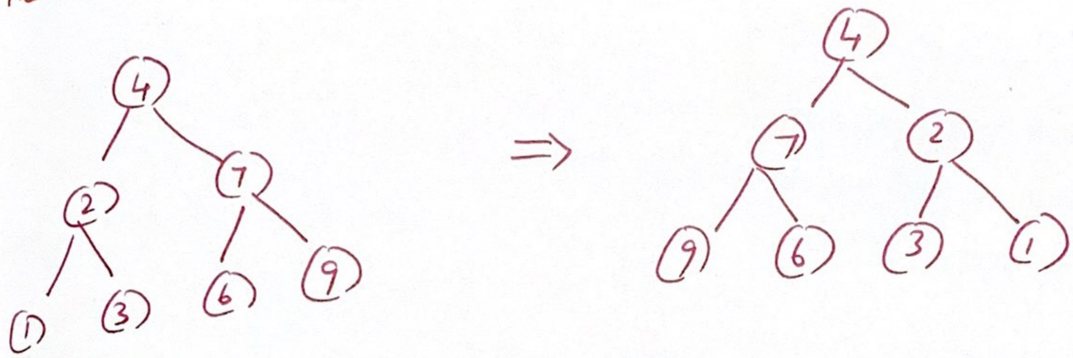
Base case hits & we return. So The Call invert (1) is popped off The stack.



The Control Goes Back to invert (2) which has Both The LEFT & RIGHT calls Complete & it returns to its parent, so, invert (2) is popped off The stack.

The Control goes Back to invert (4), which has BoTh LEFT & RIGHT Completed & so its popped off The stack.

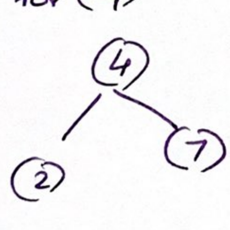
The Stack is empty => end of Recursion.



Recursive (Bottom - up)

In This, for any Node, The recursive Calls to invert its subTrees need to be Completed before The Node's child Node are swapped. Only if The recursive Call Completes, will The child Nodes be swapped.

For (eg)

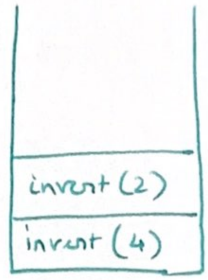


until The recursive Call to
→ invert(2) ⟶ inverts (2)'s SubTrees
→ invert(7) ⟶ inverts (7)'s subtrees.
completes,
Node (2) & Node (7) which are ChildNodes to (4) will Not be swapped.

Pass ①:- invert (4)

left = node.left ? TRUE => invert(node.left)

=> invert (2)



right = node.right ? TRUE

=> invert (node.right)

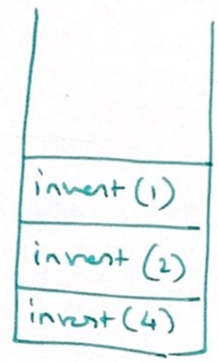
=> invert (7) {{pending}}

Not yet pushed into call stack as invert (2) must complete.

Pass ②:- invert (2)

left = node.left ? TRUE => invert (node.left)

=> invert (1)

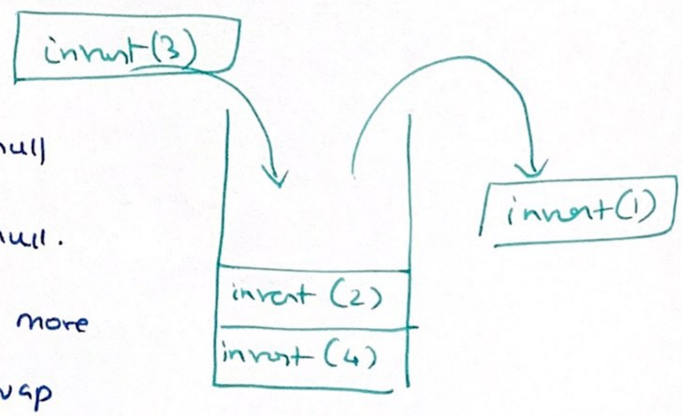


right = node.right ? TRUE => invert (node.right)

=> invert (3) {{pending}}

Not yet pushed into call stack as invert (1) must complete.

Pass ③: invert (1)

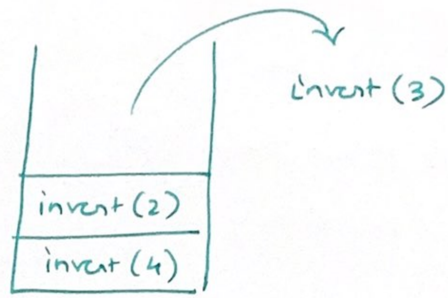


left = node.left ? False => null

right = node.right ? False => null.

Both Nodes are NULL, no more recursive calls, nothing to swap return.

Since invert(1) introduced no more recursive calls & is complete, is popped off call stack, & control comes back to pass (2) where the call to invert (3) is pending & is pushed into call stack.



Pass (4) : invert (3)

left = node . left ? false => null.

right = node . right ? false => null.

Both ~~Nodes have no~~ children are NULL

=> No more recursive calls

=> Nothing to swap Return.

Pass (2) :- invert (2)

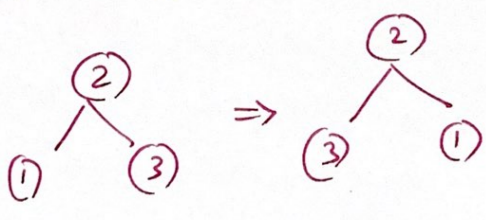
left => invert (1) complete. (Node (1)'s child Nodes are inverted)

right => invert (3) complete. (Node (3)st child Nodes are inverted)

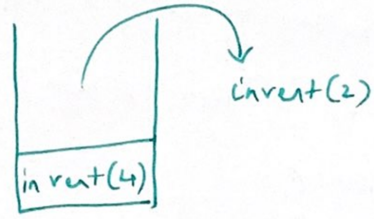
=> Time to swap the child Nodes of current Node (2)

∴ Node . left = right = 3

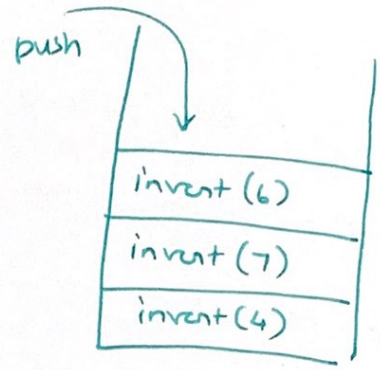
Node . right = left = 1



Thus Node (2) has been inverted & its sub-trees are inverted & can be returned. The function invert (2) is complete & pops off call stack



When Control Comes to pass (1) for invert (4).
The left SubTree invert (2) is Complete. But
The Right SubTree invert (7) is yet to be processed.
So, its pushed into Call stack.



PASS 5 invert (7)

left = node.left ? TRUE

=> invert (node.left)

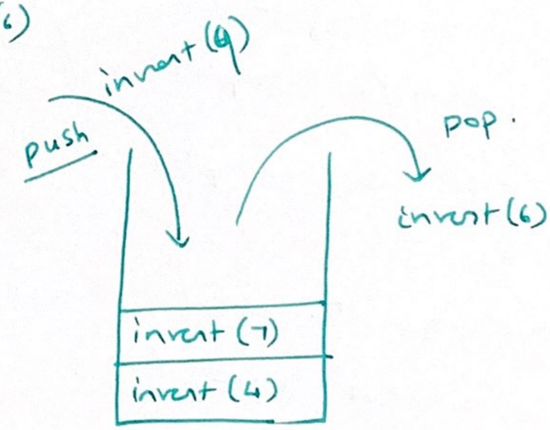
=> invert (6)

right = node.right ? TRUE

=> invert (node.right)

=> invert (9) → {{pending}}

Not yet pushed into Call stack as invert (6) must Complete.



PASS (6) invert (6)

left = node. left ? False => NULL

right = node. right ? False => NULL

Both child Nodes are NULL

=> no more recursive Calls

=> Nothing to swap, so return.

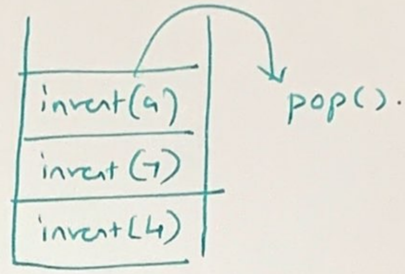
invert (6) is Thus popped off Call stack.

once popped, Control goes Back to Pass (5) invert (7) Call & since The LEFT is done, The right invert (9) is pushed into STACK

pass (7) invert (9)

left = node. left ? false => NULL
right = node. right ? false => NULL
Both child are NULL

=> No more recursive calls
=> Nothing to swap
∴ invert (9) is popped off
The stack.



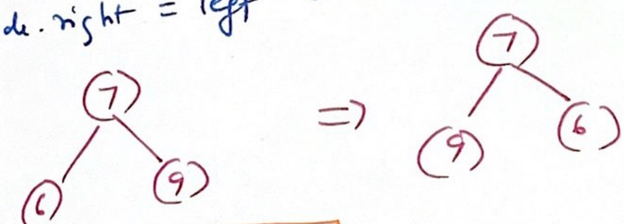
Now The Control goes to
invert (7) of pass ⑤

Pass ⑤ invert (7)

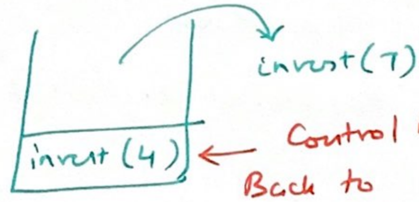
left => invert (6) Complete (Node ⑥'s child Nodes are inverted)
right => invert (9) Complete (Node ⑨'s child Nodes are inverted)

=> Time to swap The
Child Nodes of Current Node (7)

Node. left = right = 9
Node. right = left = 6



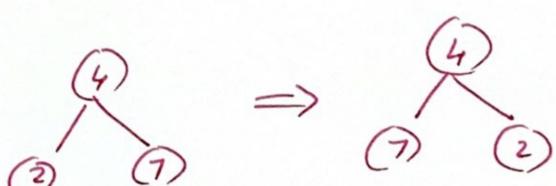
Thus Node (7) has been
inverted. & its SubTrees
are inverted
The invert (7) Completes
& is popped off stack.



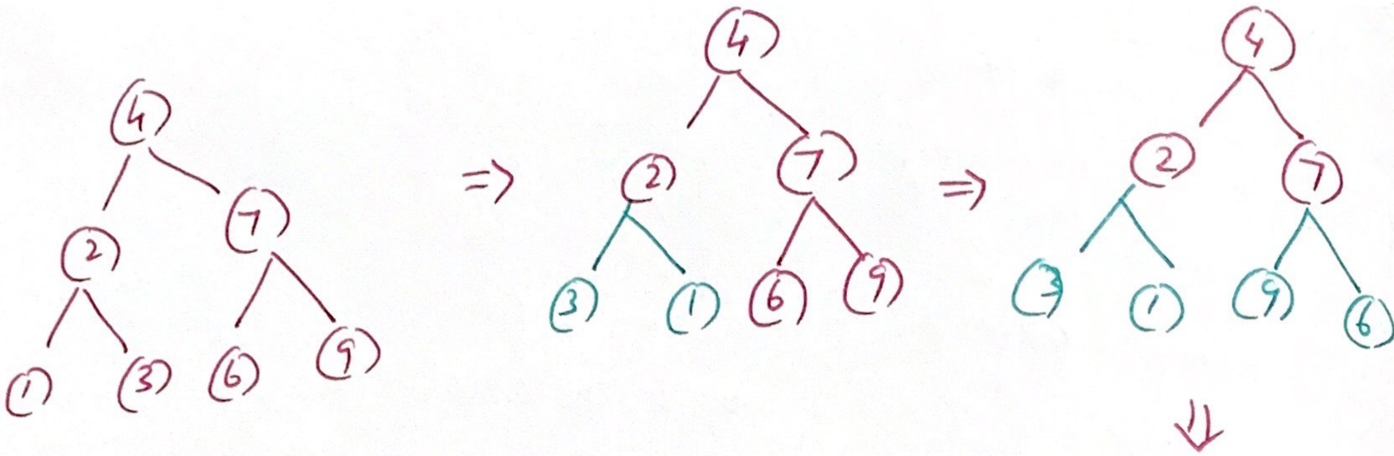
Control is
Back to
invert (4) of pass 1

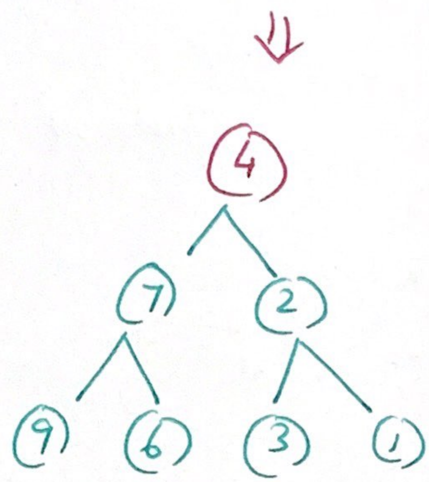
Pass (1) invert (4)

left => invert(2) Complete.
right => invert(7) Complete
=> ChildNode of ④ Can be swapped
node. left = right = 7
node. right = left = 2

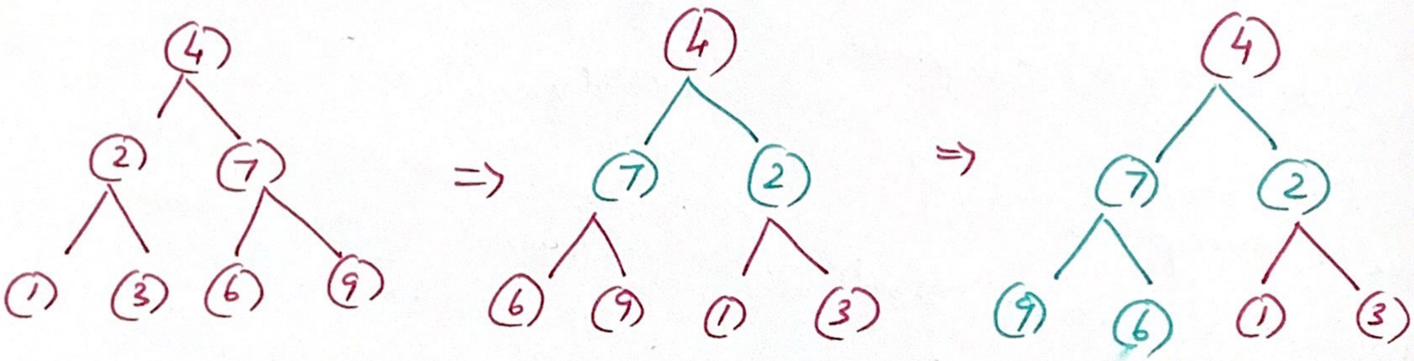


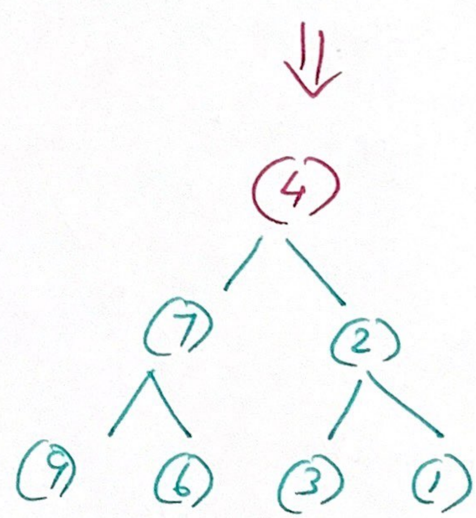
∴ Node ④ has been inverted &
invert (4) is popped off stack.
And Recursion Completes.





Bottom-up recursion explained

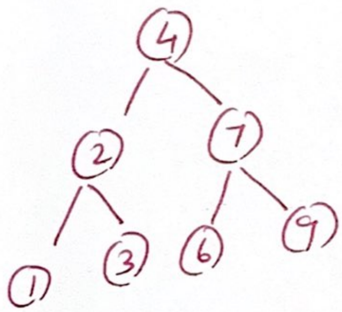




Top-down Recursion explained

Invert Binary Tree – Iterative.

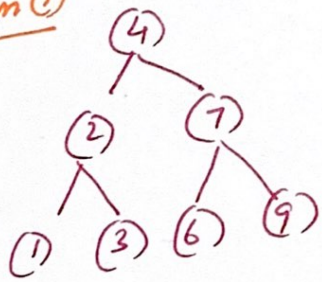
for iterative Invert of Binary Tree, LEVEL ORDER TRAVERSAL Can be used.



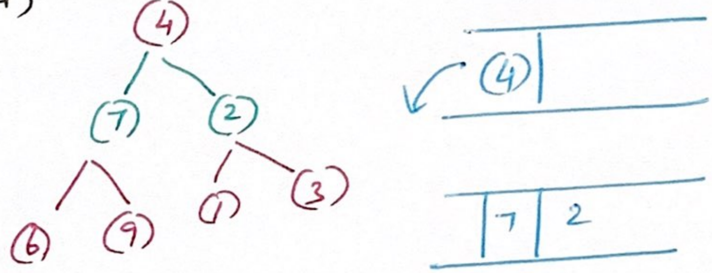
(1) For every node, Swap its LEFT & RIGHT.

(2) Only child Nodes are swapped, & not the children of those child nodes.

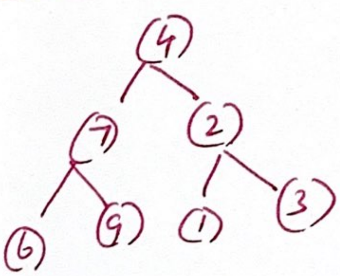
Pass (1) (current node dequed = 4)



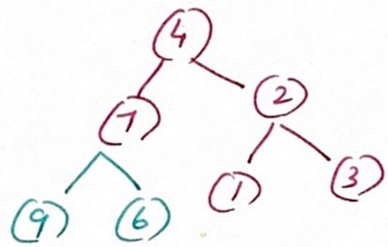
⇒

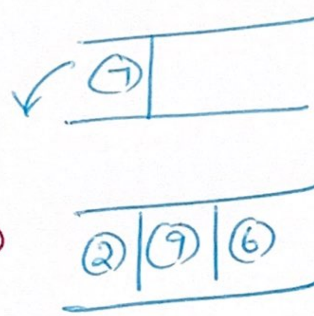


Pass (2) :- (Current Node dequed = 7)

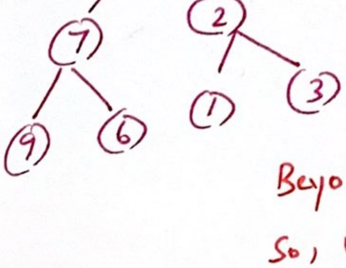


⇒

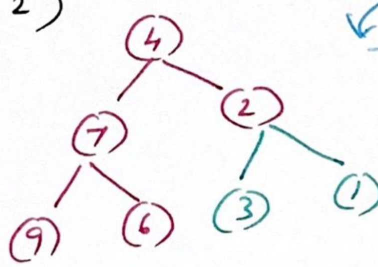


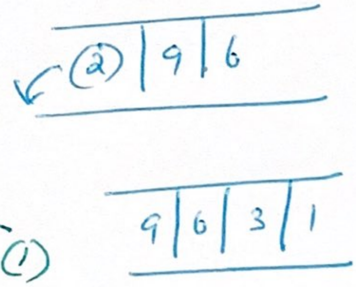


Pass (3) :- (Current Node dequed = 2)



⇒





Beyond This, The next Level Nodes have No Child Nodes.

So, with This The iterative inversion is done.