



University of
Pittsburgh

Algorithms and Data Structures 2

CS 1501



Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
 - Homework 3: this Friday @ 11:59 pm
 - Lab 2: next Monday @ 11:59 pm
 - Assignment 1: Monday Oct 10th @ 11:59 pm
- Lecture recordings are available on Canvas under Panopto Video
- Please use the “Request Regrade” feature on GradeScope if you have any issues with your grades
- TAs student support hours available on the syllabus page

Previous lecture

- Binary Search Tree
 - How to delete
- Runtime of BST operations
 - add, search, delete
- Red-Black BST (Balanced BST)
 - definition and basic operations

Muddiest Points

- **Q: Was just curious of the real-life applications of red-black tree**
- **A: Example applications:**
 - TreeMap and TreeSet in Java
 - CPU Scheduling inside the Linux Kernel
- **Q: what does the triangle mean at the bottom of tree drawings?**
- **A: It means a subtree**

Muddiest Points

- **Q: when deleting interior nodes how does the parent node know where its new child is**
- **A:**
 - The removal method (`removeFromRoot`) returns the new root
 - The return value of the removal method is assigned to the right or left child of the parent
- **Q: how does the new node know where it's child is?**
- **A:** The new root has the same two children that the old root has; we are just replacing the data of the root node.

Muddiest Points

- Q: Can you explain how all the different functions for the removal of a node connect together into remove(T)
- A:
 - remove(T) creates an empty wrapper object and calls removeEntry, passing to it the entry to delete and the empty wrapper
 - removeEntry searches for *entry* by either moving left or right (depending on the comparisons between entry and the data portion of the current node)
 - Once the node that contains entry is found, the data inside the node is put inside the wrapper object
 - Then, removeFromRoot is called on the found node
 - removeFromRoot removes the node (the three cases) and may need to call findLargest and removeLargest on the left subtree of the node (if it has two children)
 - removeFromRoot returns the new root of the tree to removeEntry
 - removeEntry then sets the right or left child of current node to the return value of removeFromRoot and returns the node after this modification
 - Eventually removeEntry returns back to remove the (possibly) new root of the tree after removing entry
 - Finally, remove sets the root of the tree to the return value of removeEntry and returns the entry inside the wrapper object

Muddiest Points

- Q: Why do you need a wrapper class for removeEntry?
- A: Because we want removeEntry to return the root after removing entry.
- But we want it also to return a reference to the deleted data object, which is achieved using the wrapper object.

Muddiest Points

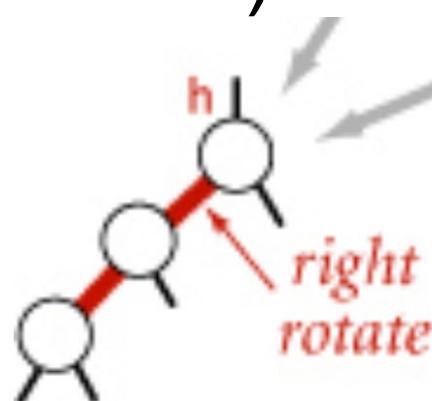
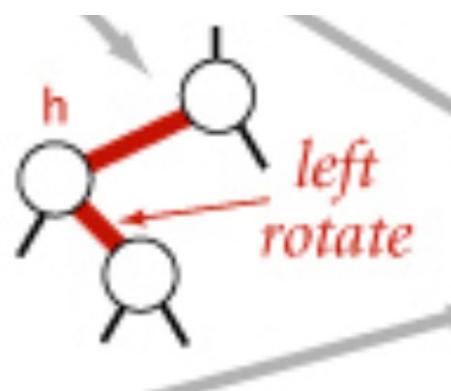
- **Q: Would the theoretical worst case BST operation be when the tree is only left nodes/right nodes?**
- **A:** Yes. For example, when searching for the deepest node in such a degenerate tree, the depth of the node will be $n-1$ and the run time is $\Theta(n)$.

Muddiest Points

- Q: A red node can be anywhere in the right subtree just so long as it's leaning to the left side of that subtree, correct?
- A: Correct

Muddiest Points

- Q: how to fix violations in B-R BST in terms of recursive call
- A: Violations are checked and corrected as we climb back up the tree.
- The code for violation corrections is after the recursive calls.
- Violations are detected and corrected in the following order (h is the current node)



Muddiest Points

- **Q: what is the difference between a red node and black node?**
- **A:** A red node has a red edge to its parent, whereas a black node has a black edge to its parent. The root is always a black node.
- **Q: Where is this going? What utility do we get out of labelling the tree this way?**
- **A:** By labeling the tree nodes this way with the other constraints, the height of the tree is at most $2 \log n$

Muddiest Points

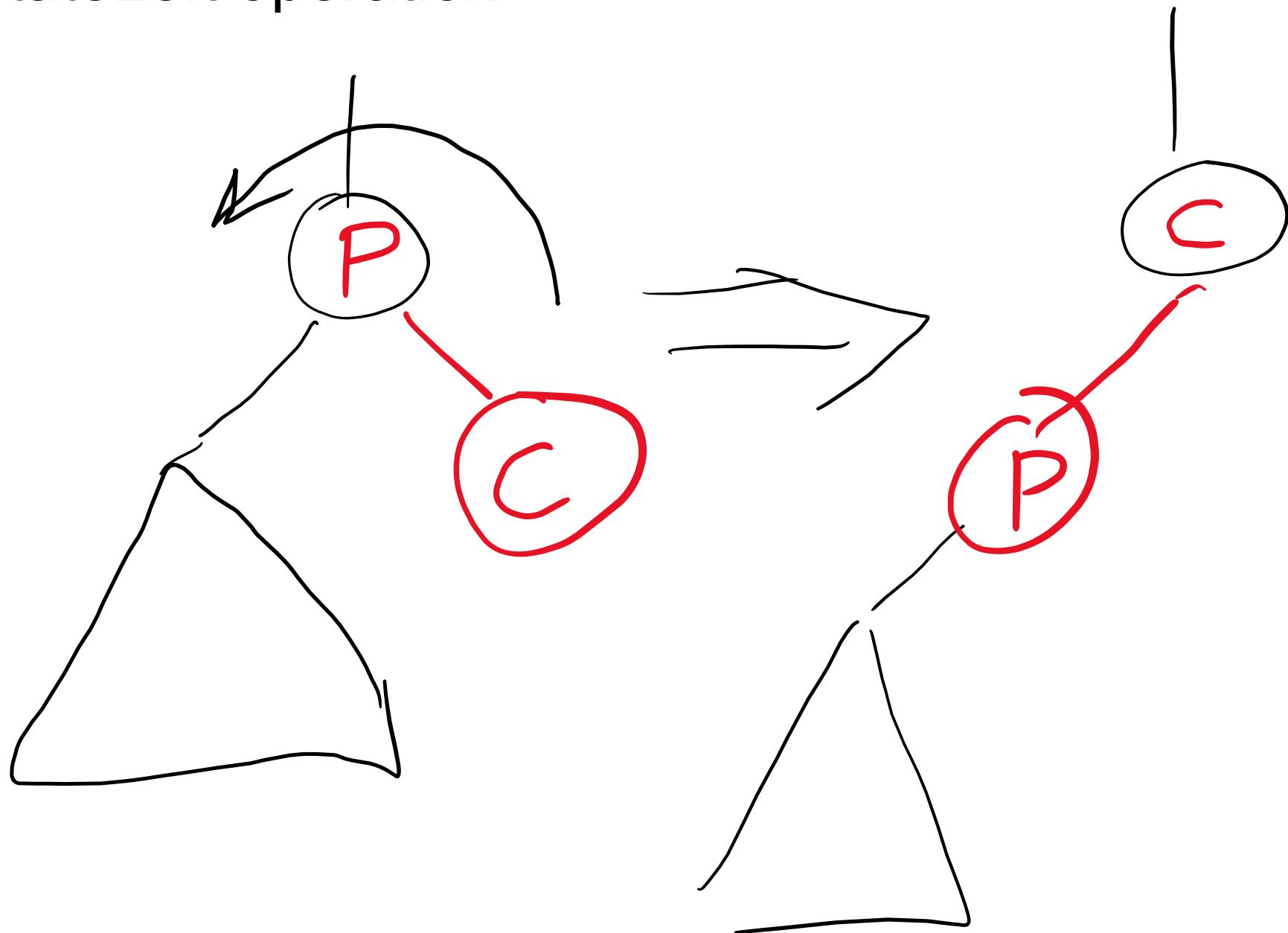
- **Q: when do we add red/black nodes**
- **Q: How do we construct an RB BST if each node we add has to be red?**
- A: We always add red nodes then potentially change color of new node
- **Q: I found the transient right red edges confusing**
- **Q: I didn't quite understand how the form other than a red node being added or a rotation/color flip.**
- **Q: How does a Red black BST with red edges eventually turn into an all black BST**
- A: Let's see an example

Adding to a RB-BST

- Ok, so we add a red leaf node!
- What can go wrong then?
 - The new node is a right child
 - Fix it by left rotation

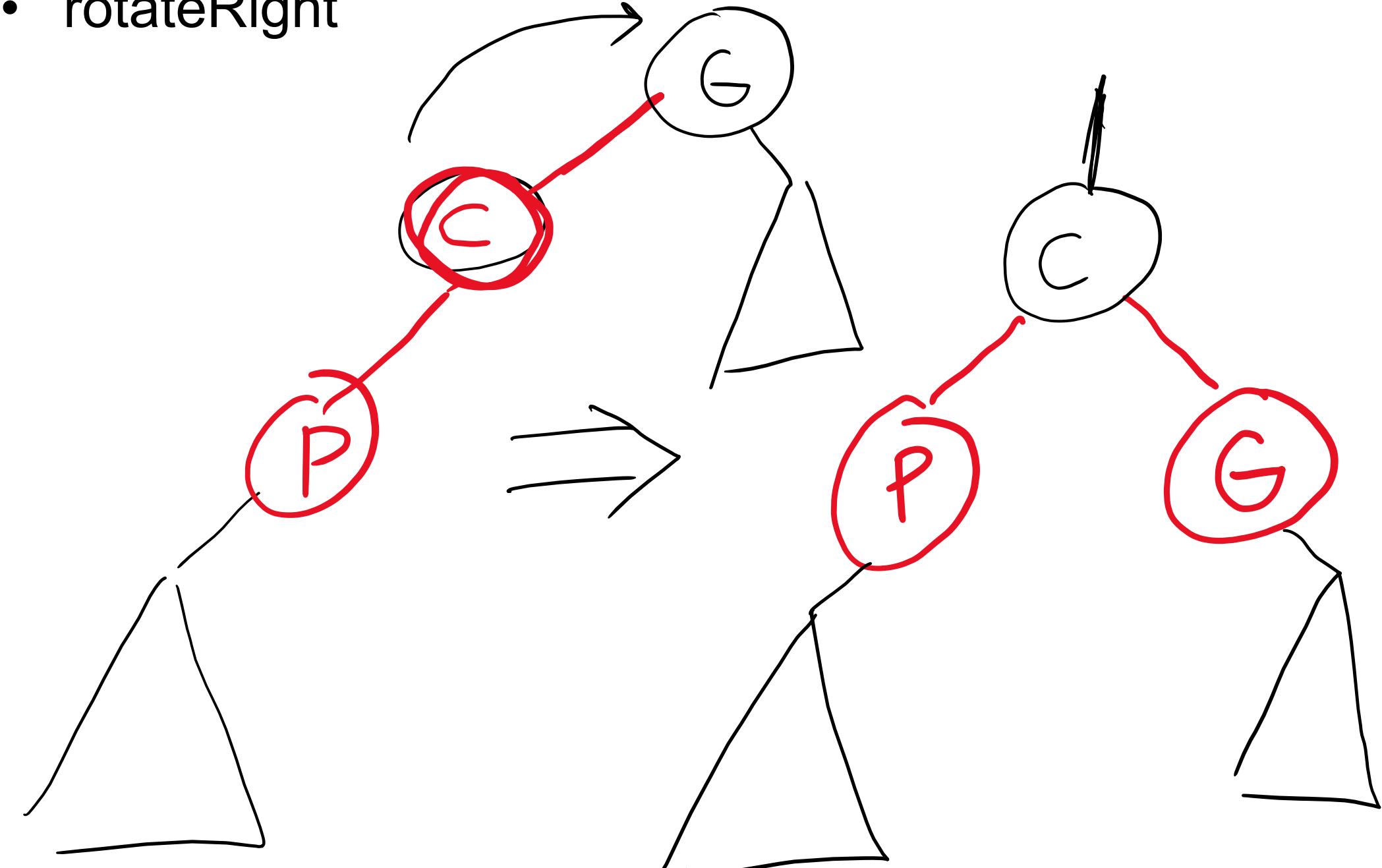
What if the new red node is a right child?

- `rotateLeft` operation



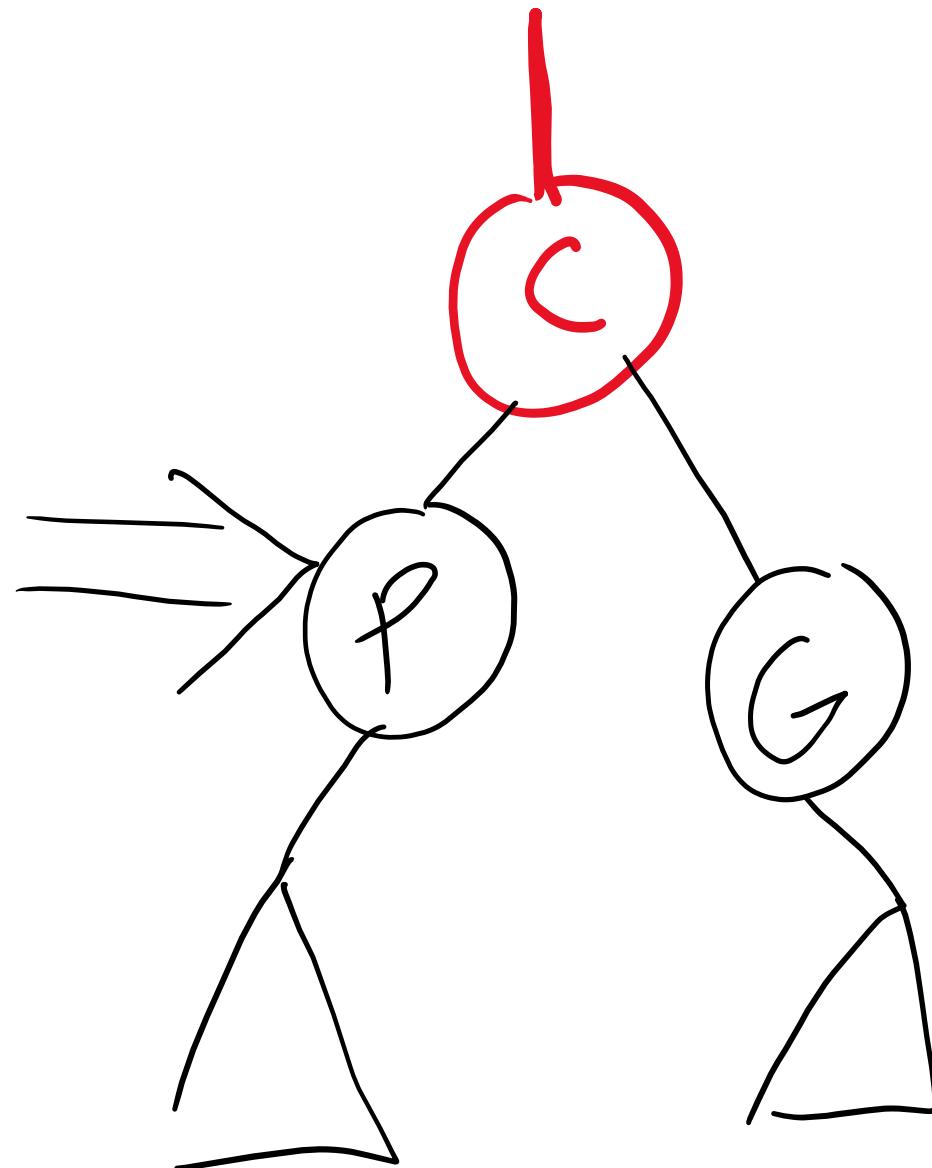
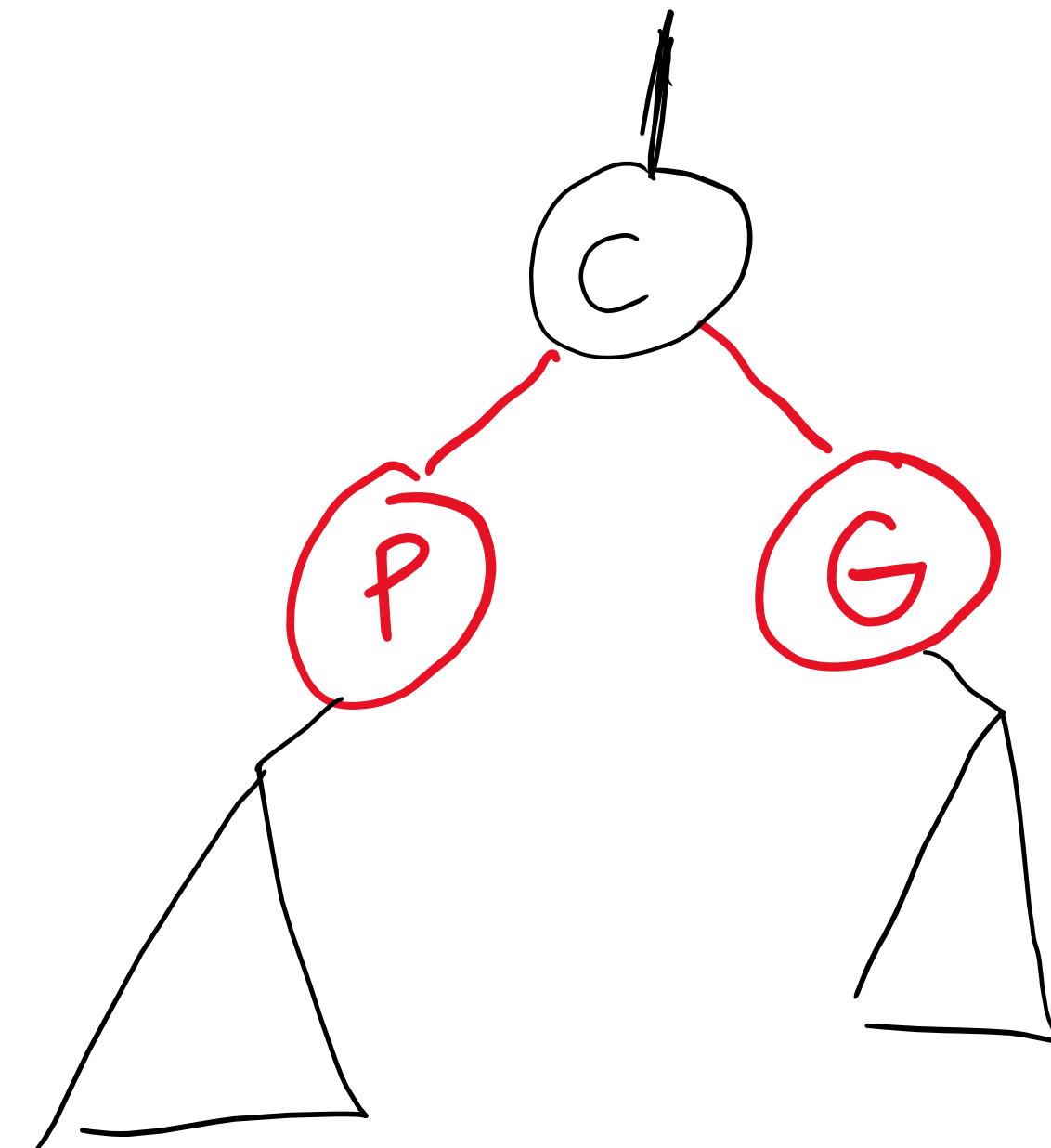
What if the new node becomes red?

- rotateRight



What if both children of a node are red?

- flipColors()



Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7

1

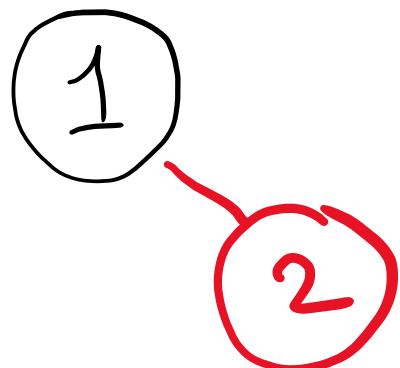
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7

1

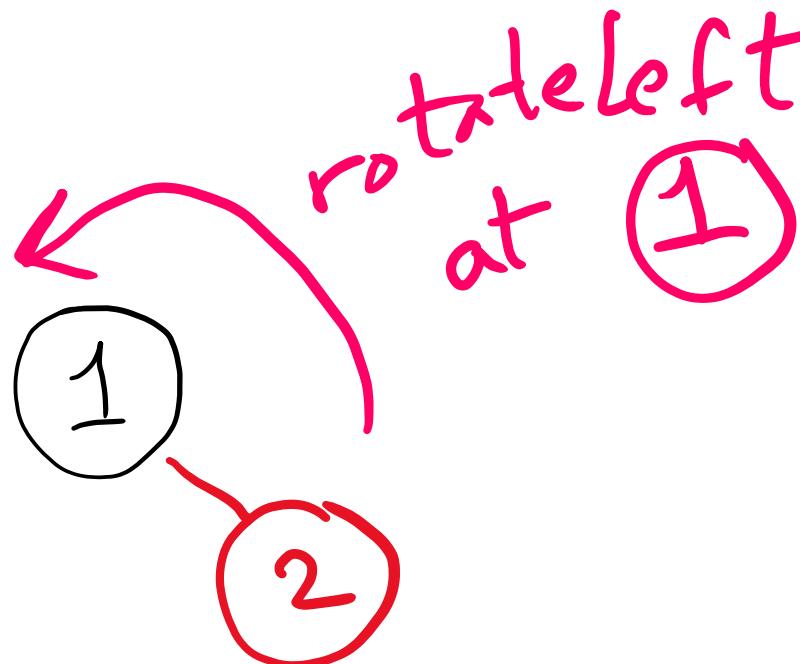
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



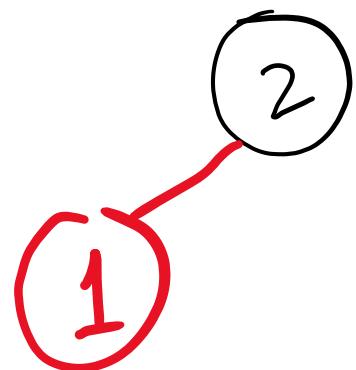
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



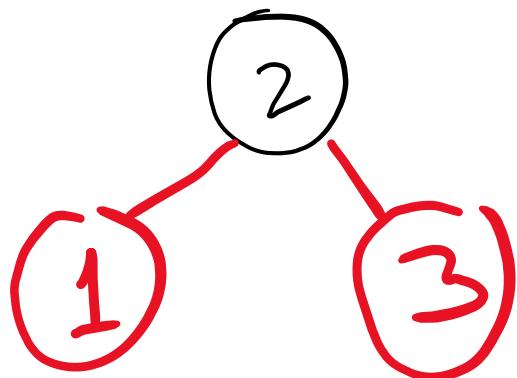
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



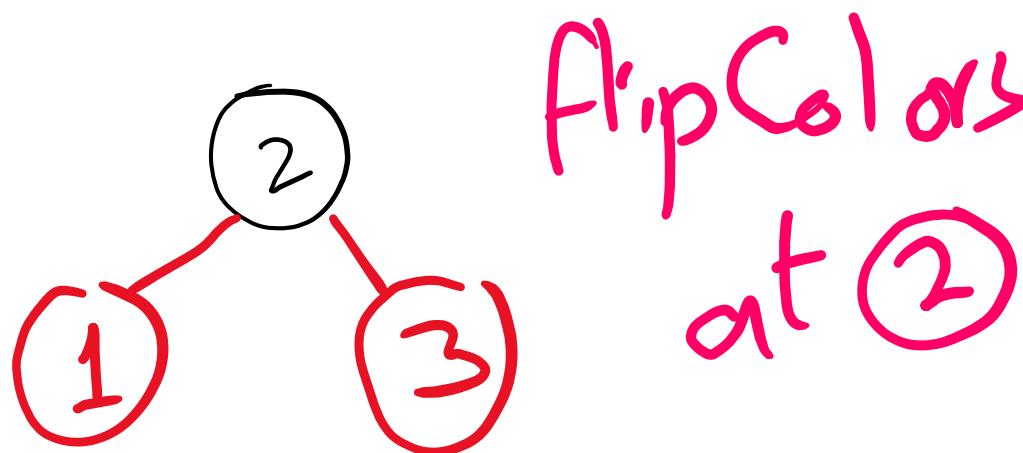
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



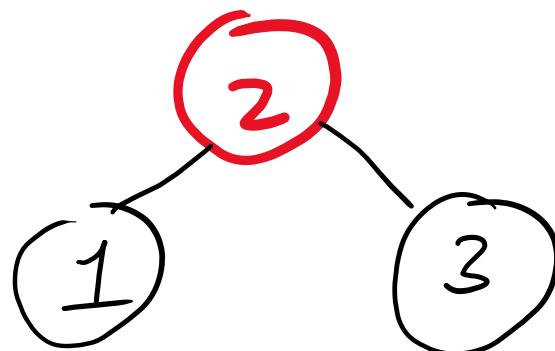
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



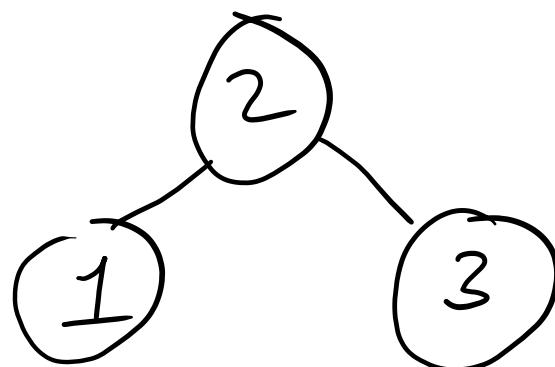
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



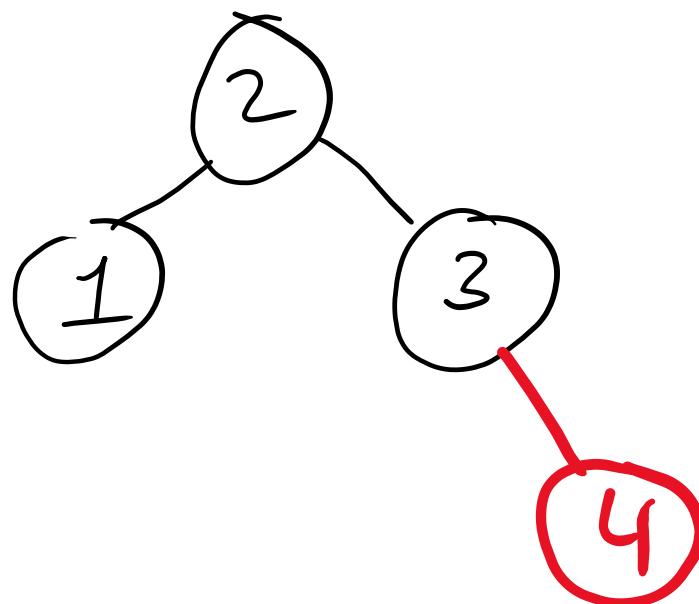
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



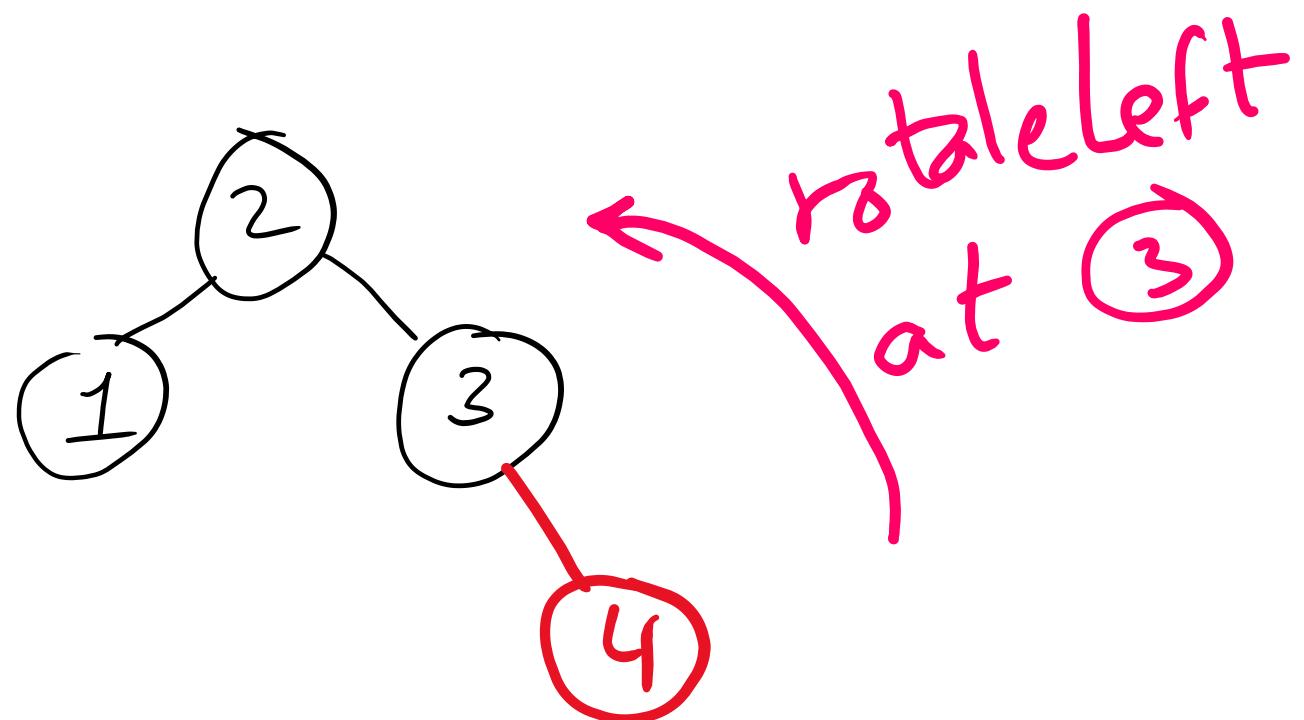
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



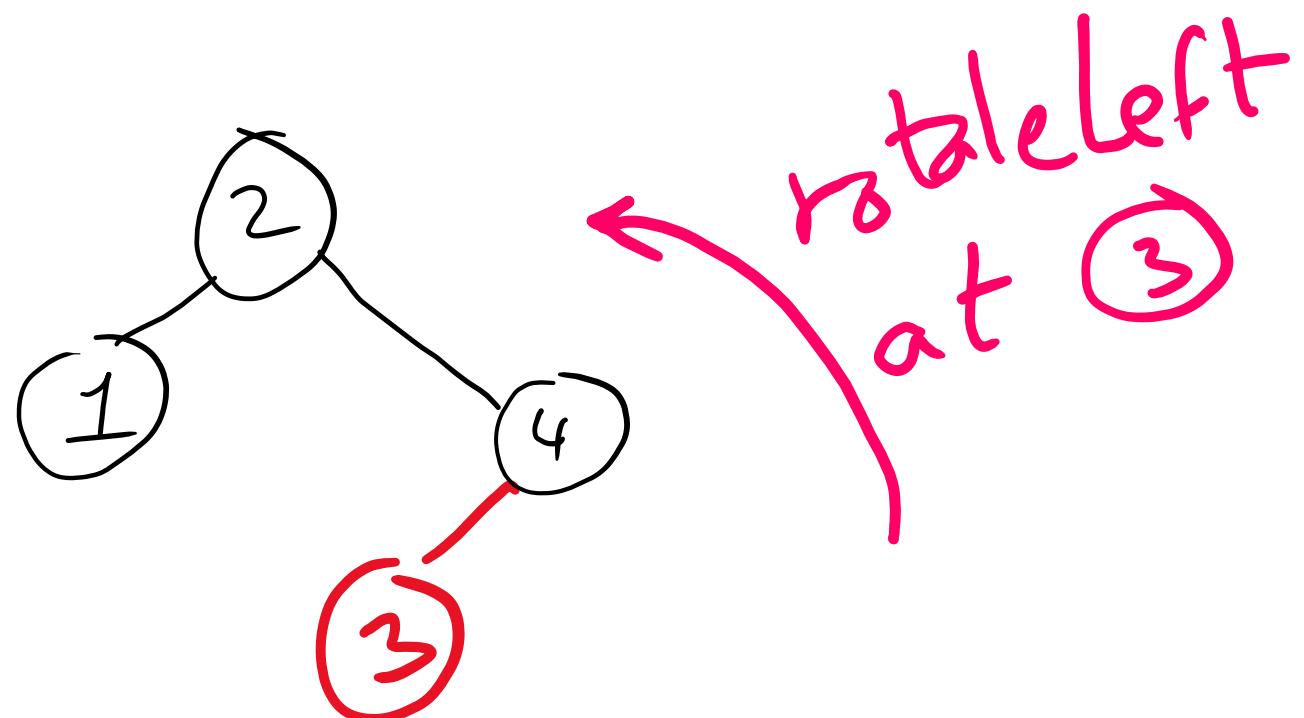
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



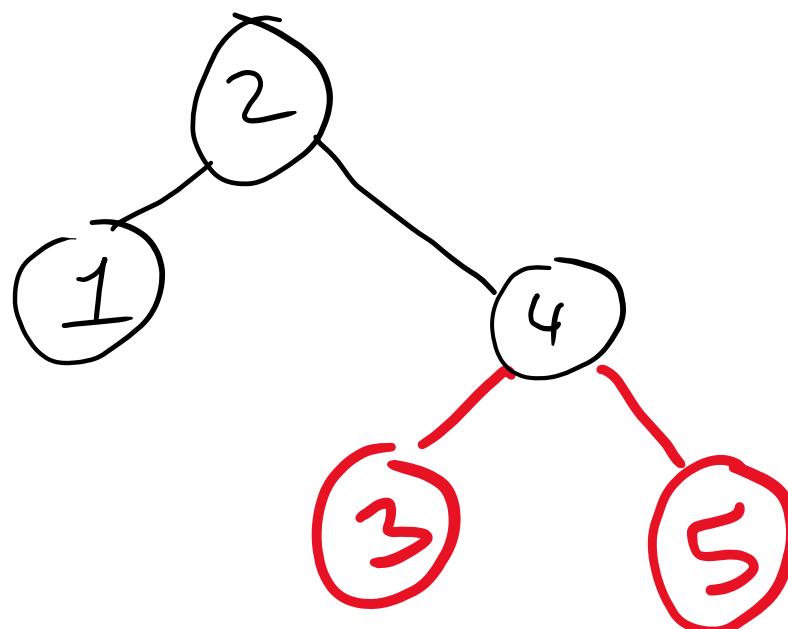
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



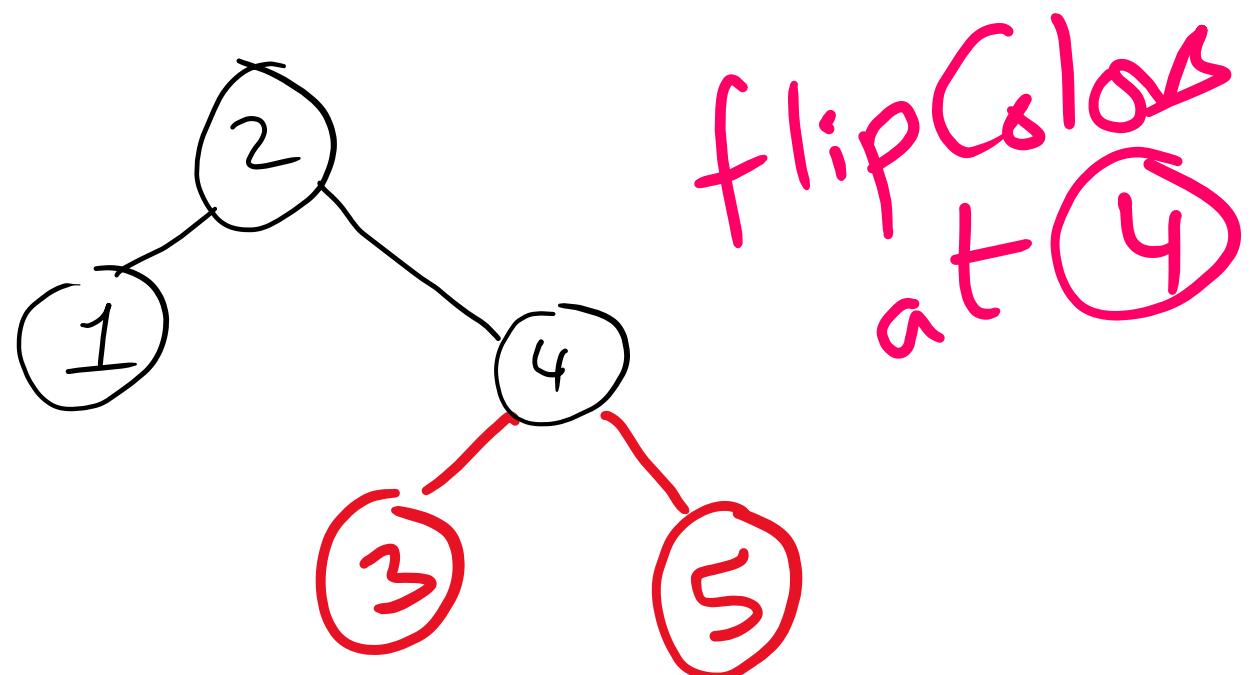
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



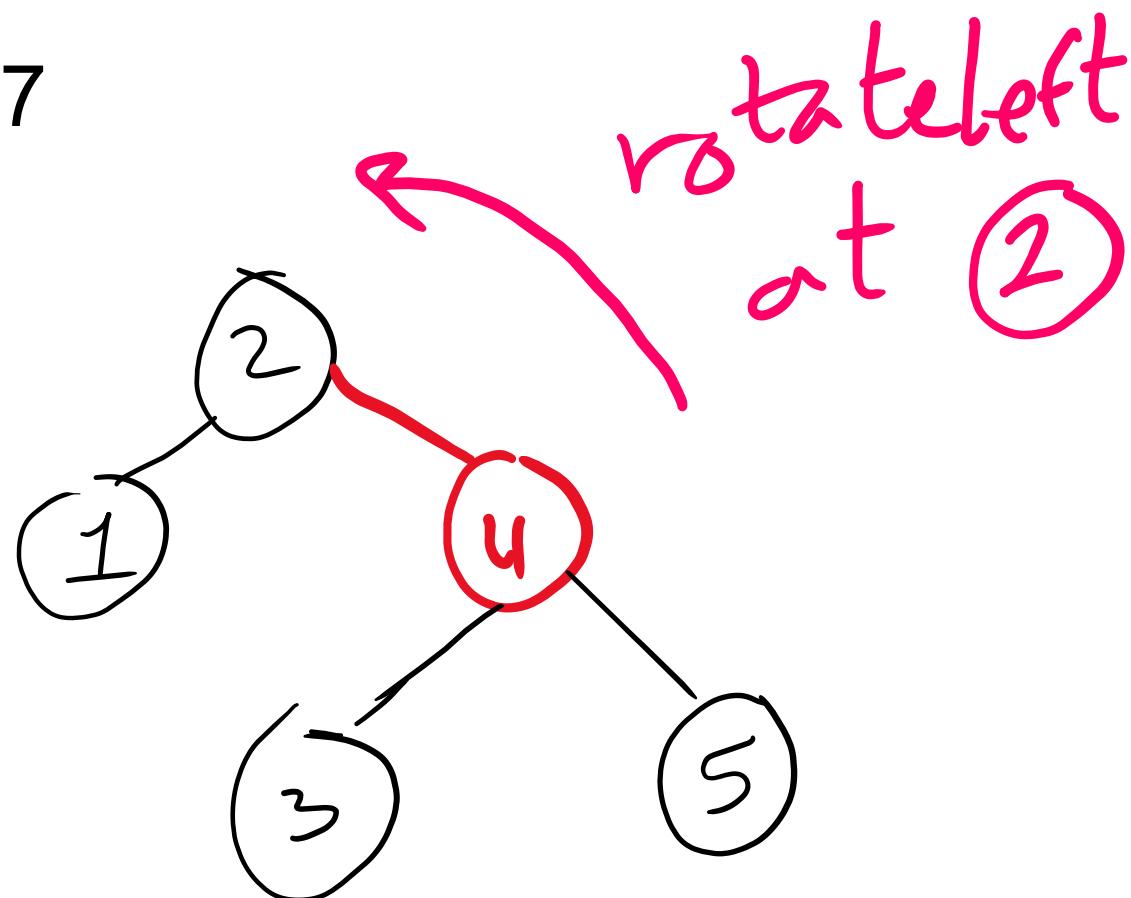
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



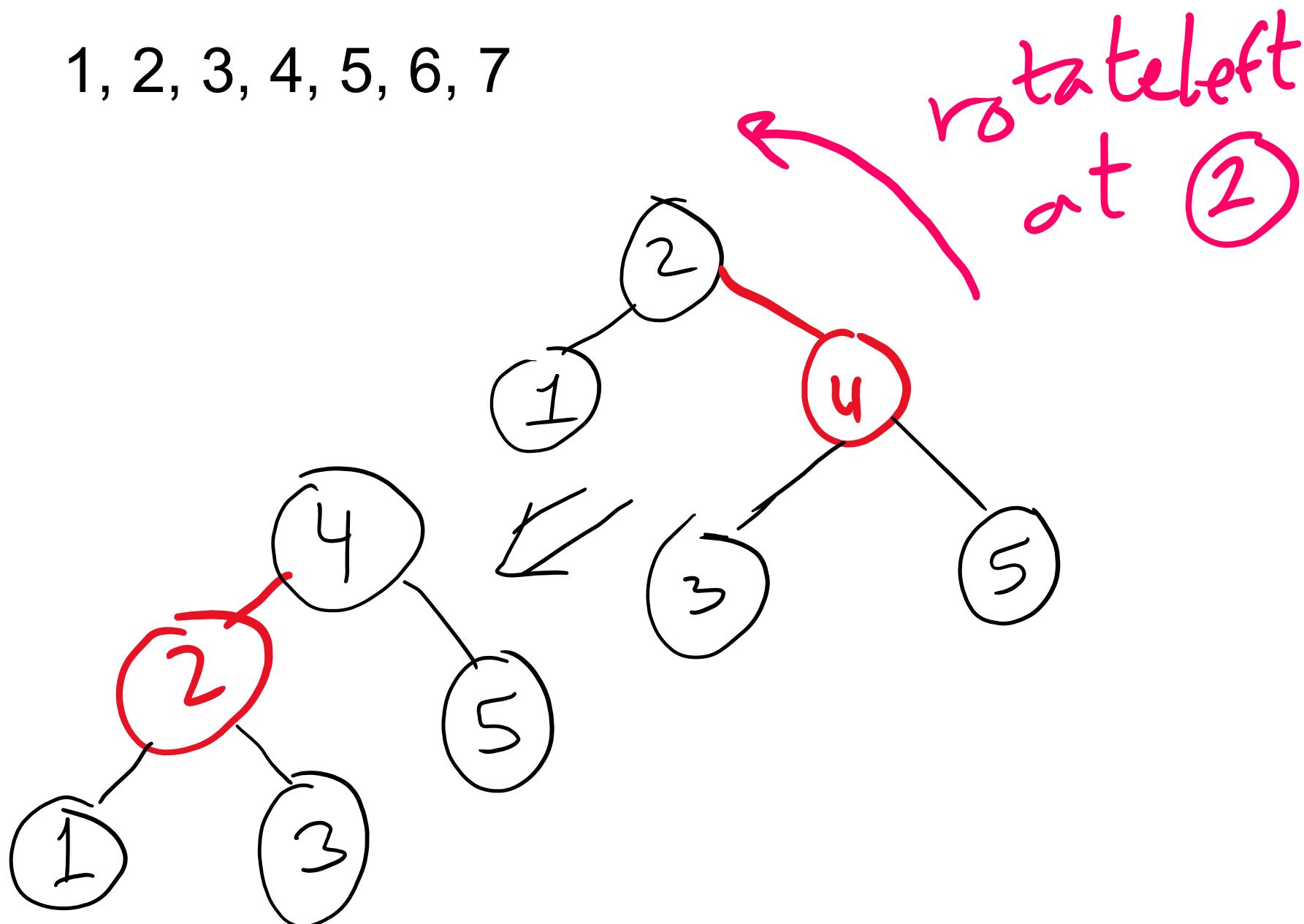
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



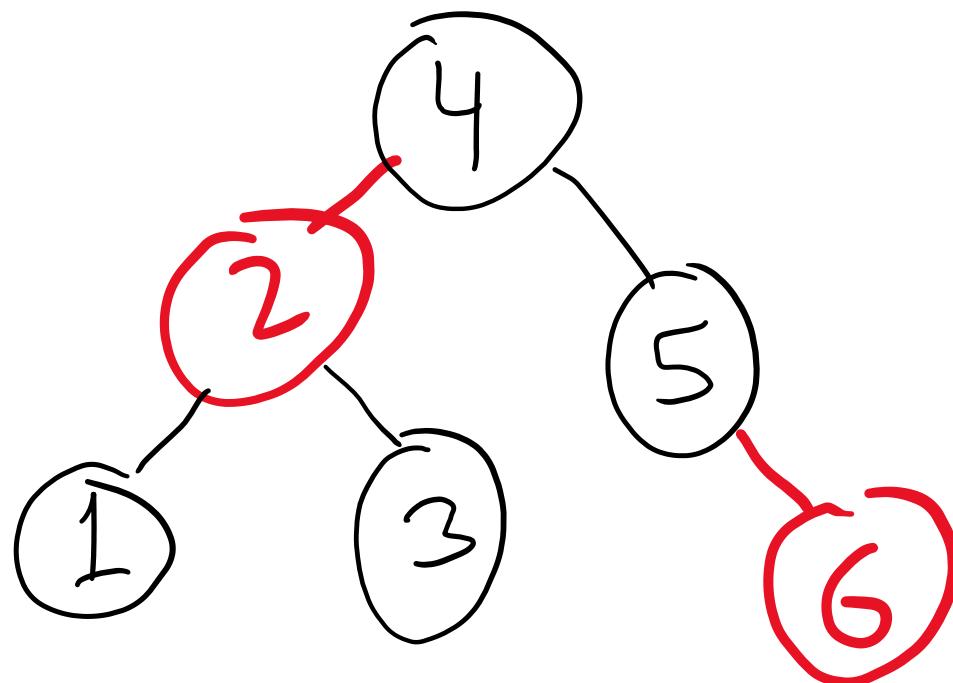
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



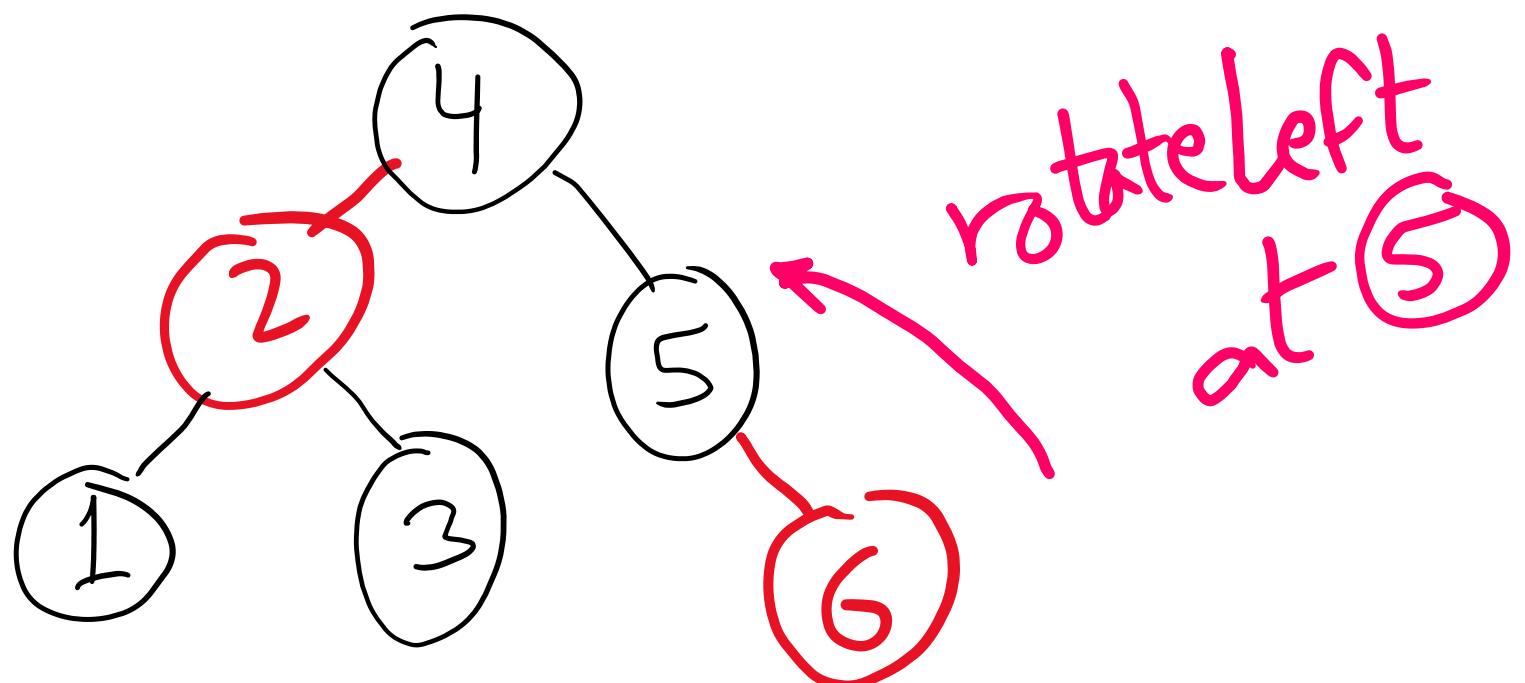
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



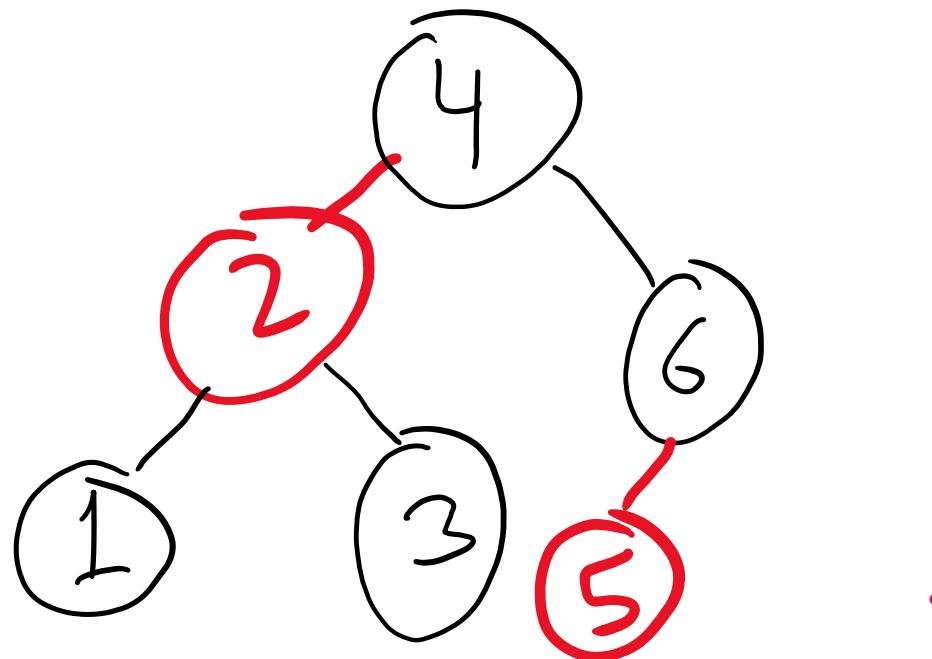
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



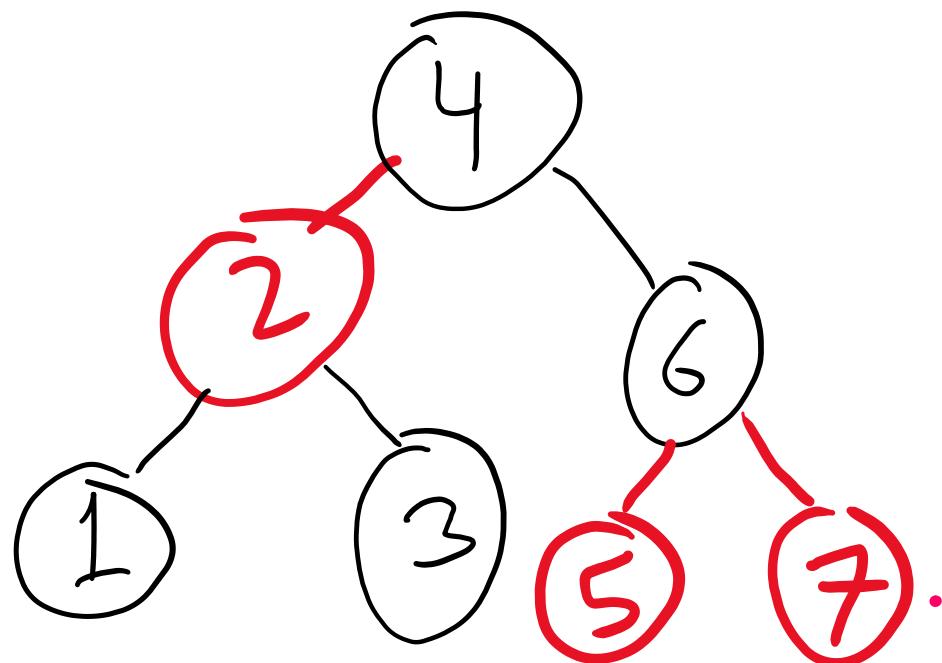
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



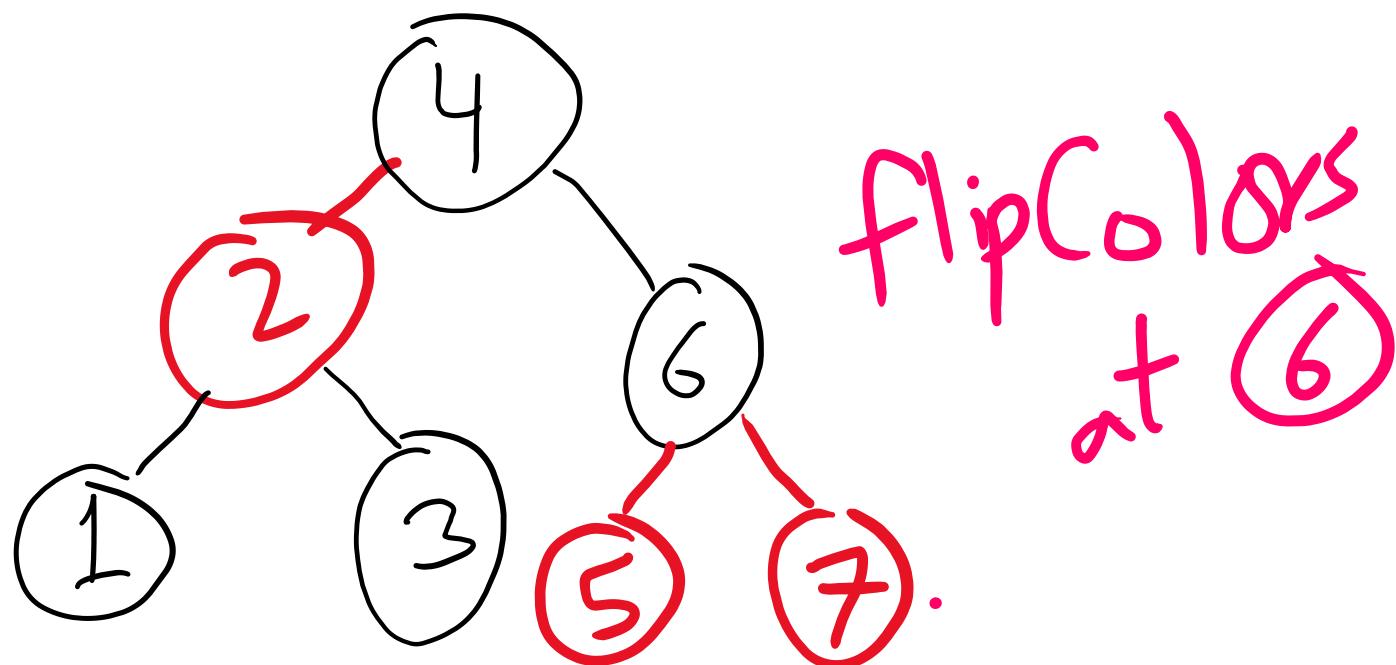
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



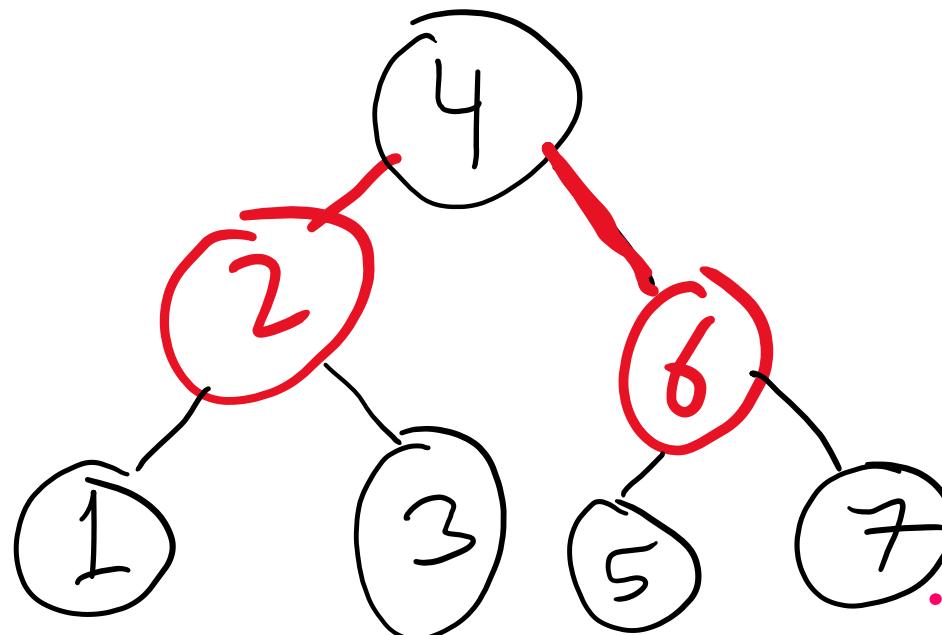
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



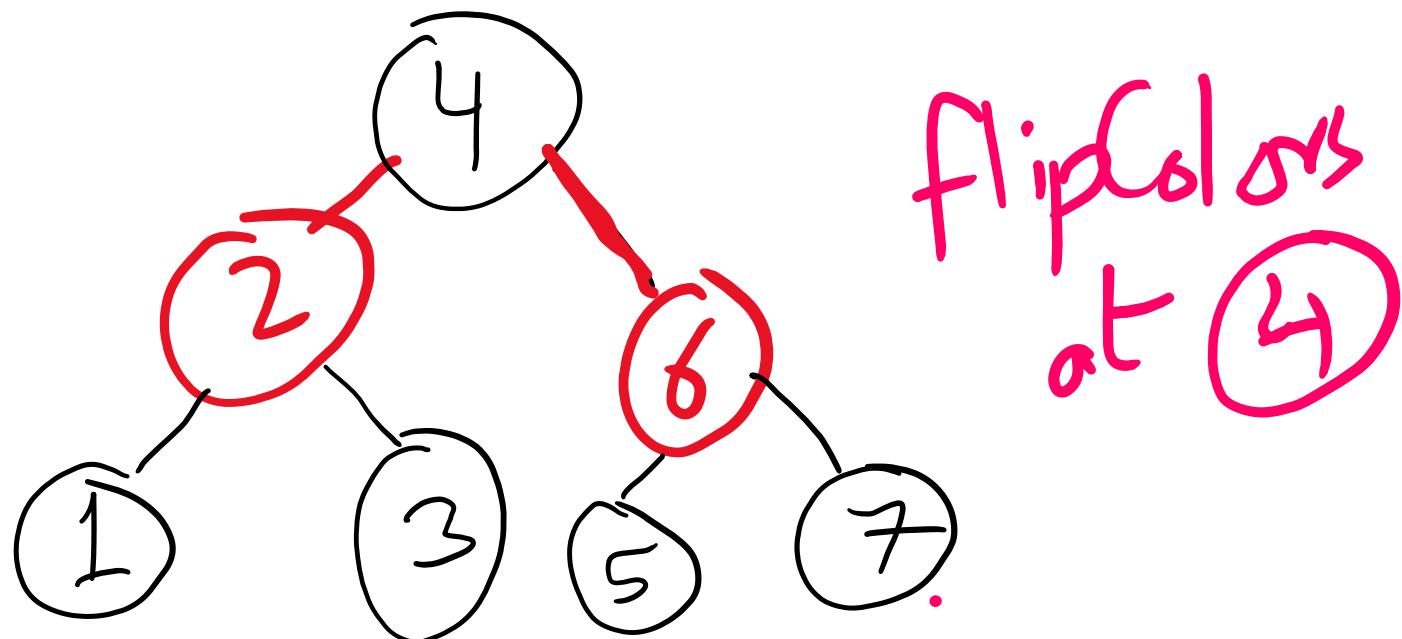
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



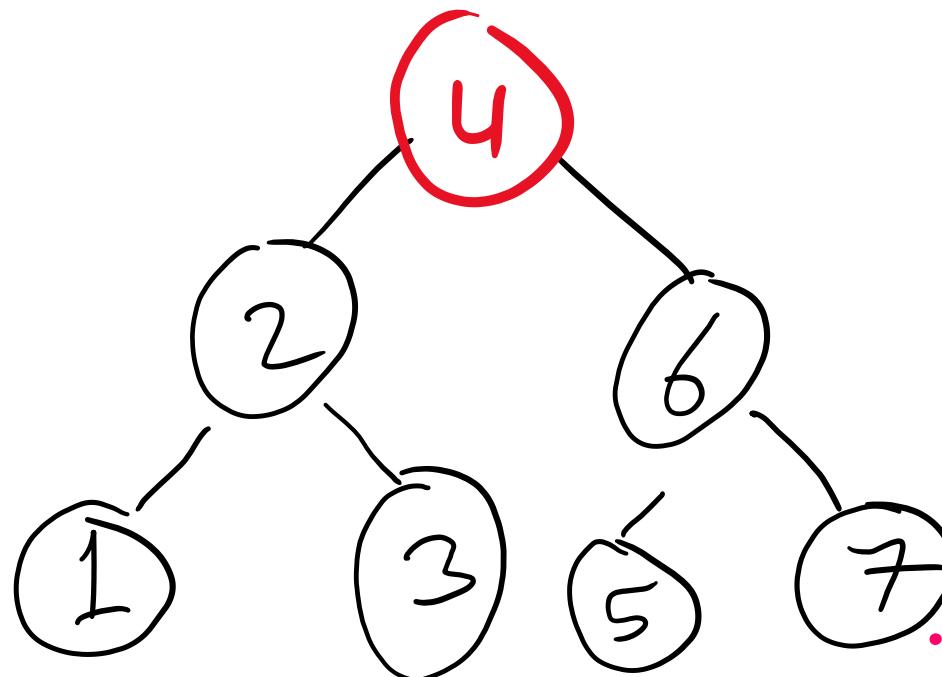
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



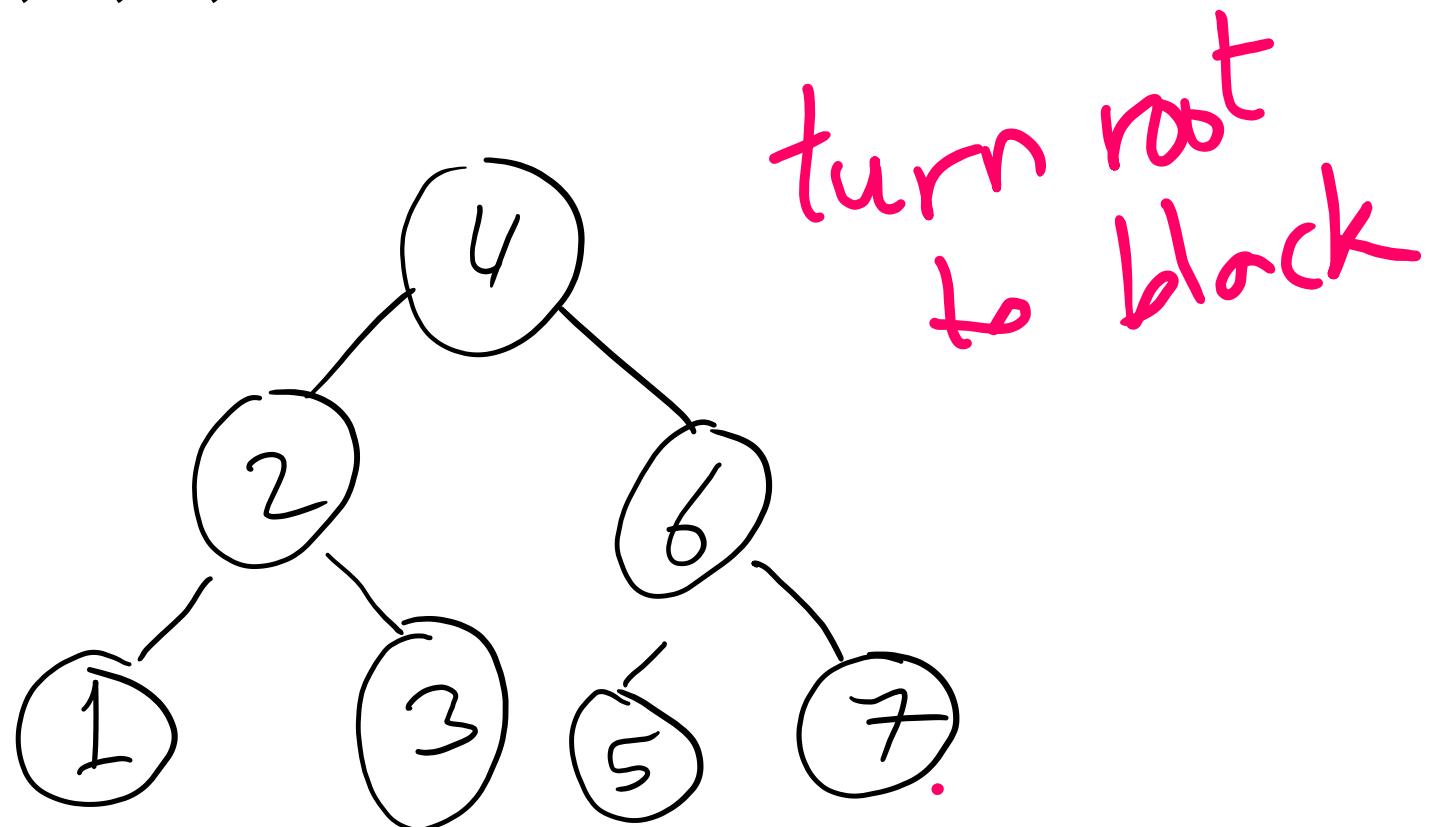
Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



Adding to a RB-BST: Example

- Let's add the numbers:
- 1, 2, 3, 4, 5, 6, 7



Muddiest Points

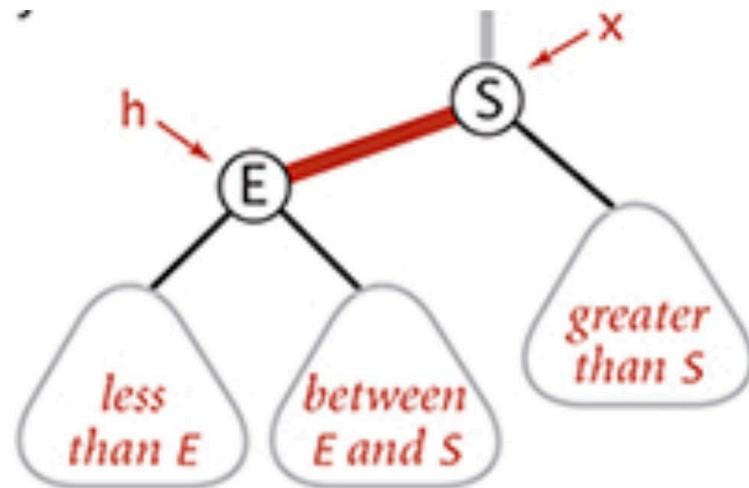
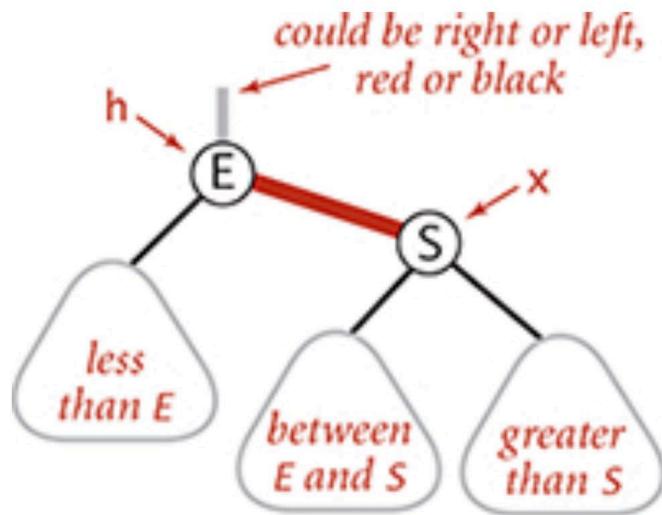
- **Q: If correcting one violation for the red-black BST, leads to another violation, how is that corrected?**
- A: One violation may lead to another violation of a different type, which may lead to another violation of a different type.
- However, this process stops after at most three corrections passing the violation to the next level up.
- Violations keep propagating up the tree until we have a violation at root, which is root node being red instead of black
 - This can be easily corrected by coloring root black again

Muddiest Points

- Q: Confused on the benefits of adding red nodes and what that would look like in an application
- A: Adding a black node would violate the perfect balance constraint (all paths from root to null-edges have the same number of black edges)
 - Hard to fix!

Muddiest Points

- Q: The code for the right and left rotations were confusing
- A: The rotateLeft at node h code emerges from contrasting the before and after pictures



```
x = h.right;  
h.right = x.left;  
x.right = h;  
x.color = h.color;  
h.color = RED;
```

Muddiest Points

- Q: What purpose does key serve in Red-Black BST?
- A: The data objects inside the nodes are compared based on the key field.

Muddiest Points

- Q: How do you tell if a node is red/black based on the color of the link?
- A: A node has the same color as the link to its parent.

Muddiest Points

- **Q: can you explain the implementations again or point me to a resource that explains them if that would take too much time?**
- A: The code is available in the code handouts. The lecture recordings are available on Canvas.
- **Q: It is hard to visualize a red black bst without pictures and diagrams for me**
- A: I agree! When you study, use diagrams of example trees.
- **Q: PLEASE GO THROUGH THE HOMEWORKS**
- A: Please use Piazza to ask about any homework questions that you are confused about

Muddiest Points

Anonymous

a day ago



Comments 0

1



This Lecture

- Red-Black BST (self-balancing BST)
 - definition and basic operations
 - delete
 - runtime of operations
- Turning recursive tree traversals to iterative

Deleting a node

- Make sure that we are not deleting a black node
 - as we go down the tree, make sure that the next node down is red
 - using a different set of operations
 - as we go back up the tree, correct any violations
 - same as we did while adding
 - if deleting a node with 2 children
 - replace with minimum of right subtree
 - delete minimum of right subtree
 - similar trick to delete in regular BST

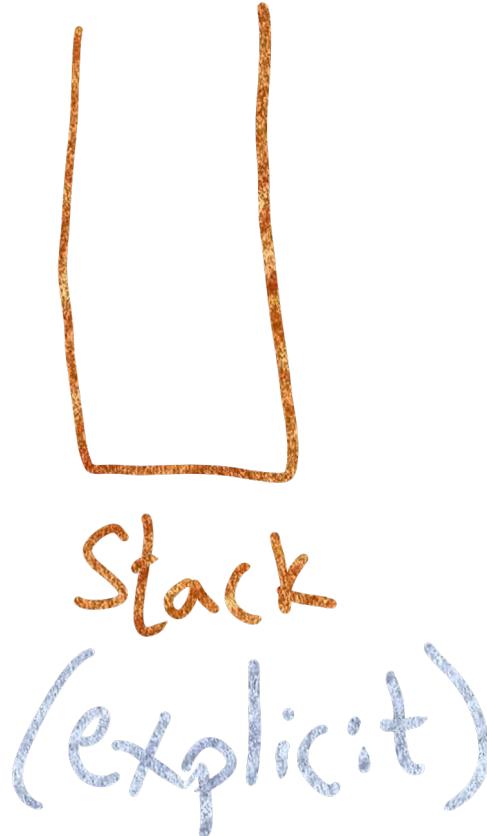
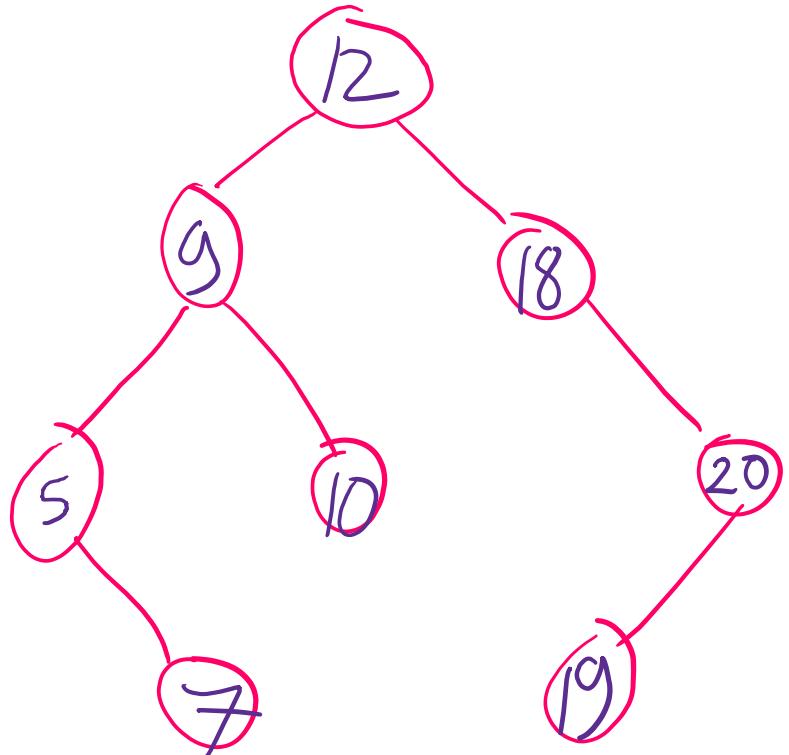
Other BST operations

- Find successor and predecessor of an item
 - Lab 3
- Find all items within a specific range
 - Please check the keys methods in RedBlackBST.java inside the TreeADT folder in the code handouts
- Same code as regular BST!
- **worst-case runtime = Theta(log n)**

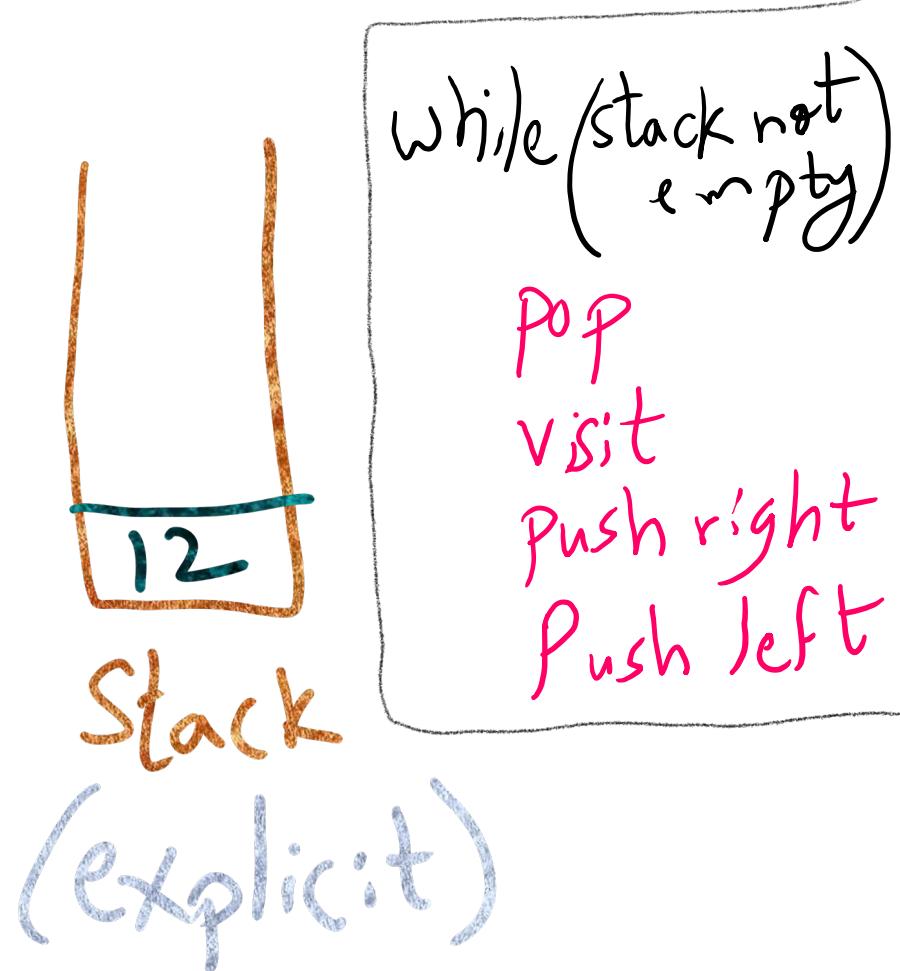
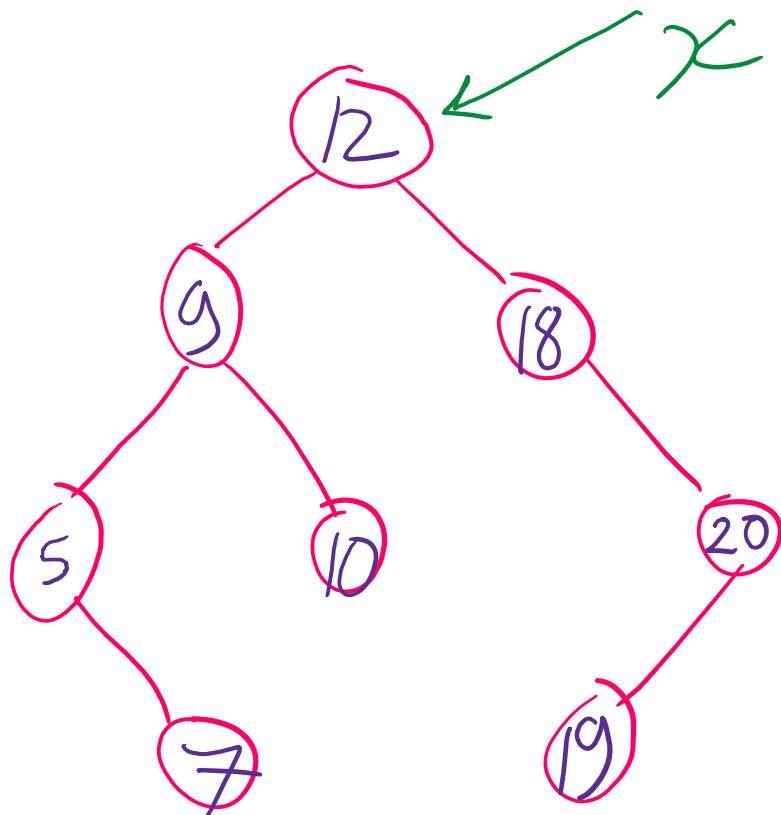
Why use iteration instead of recursion?

- Iteration is faster than recursion
 - No function call overhead (stack allocation)
- Especially useful for frequently used operations
 - Search is a good example of these operations

Iterative Preorder traversal

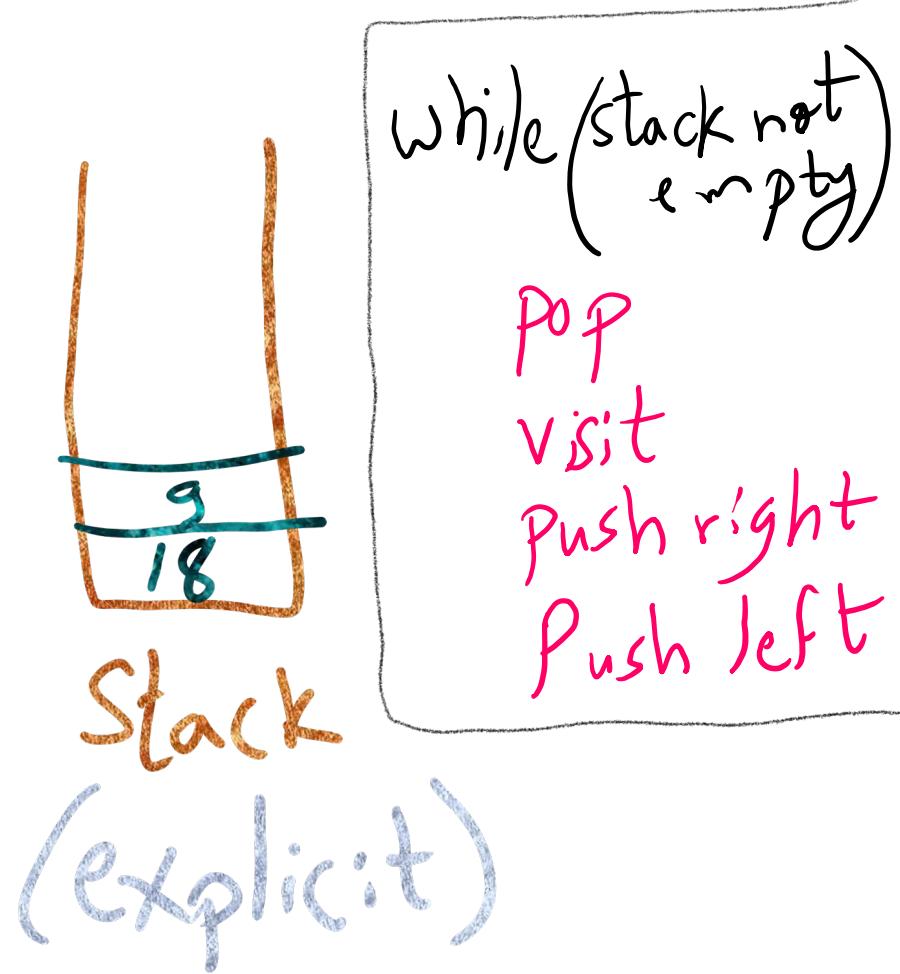
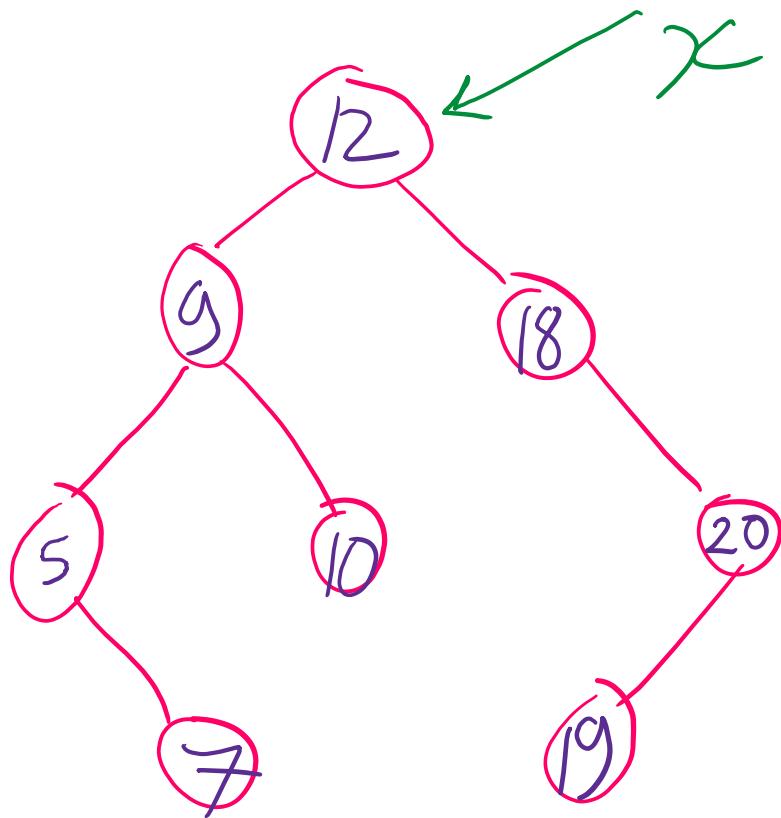


Iterative Preorder traversal



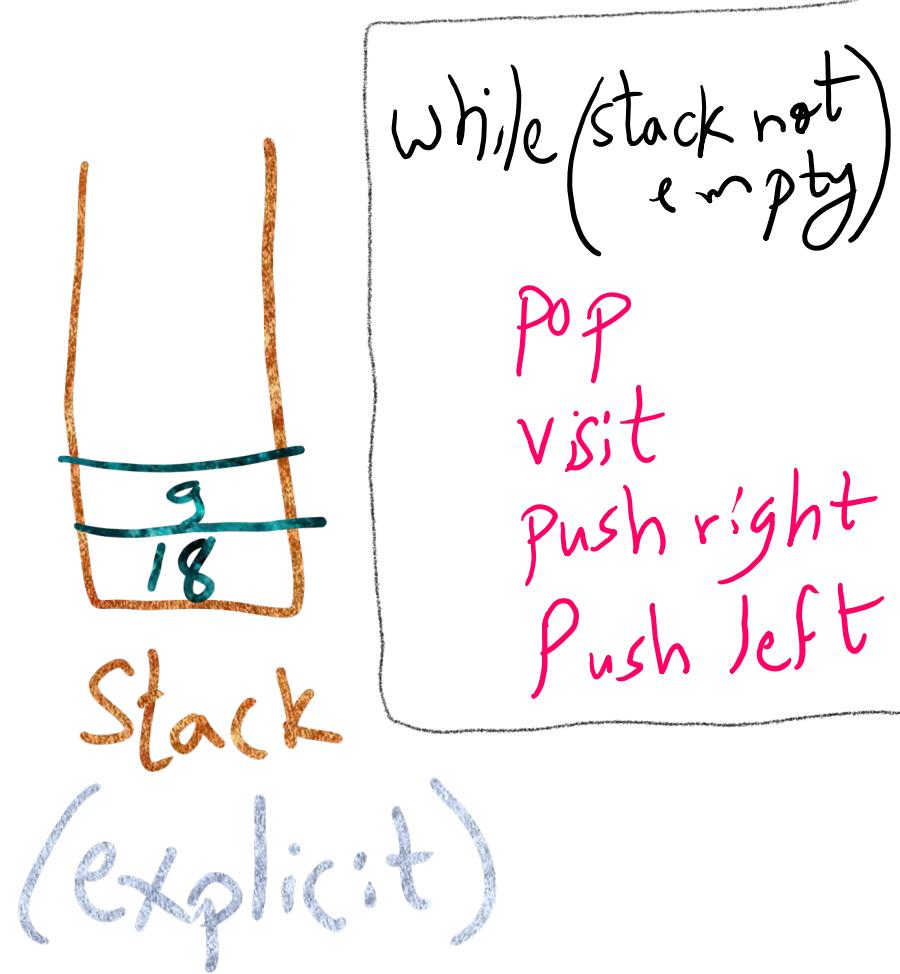
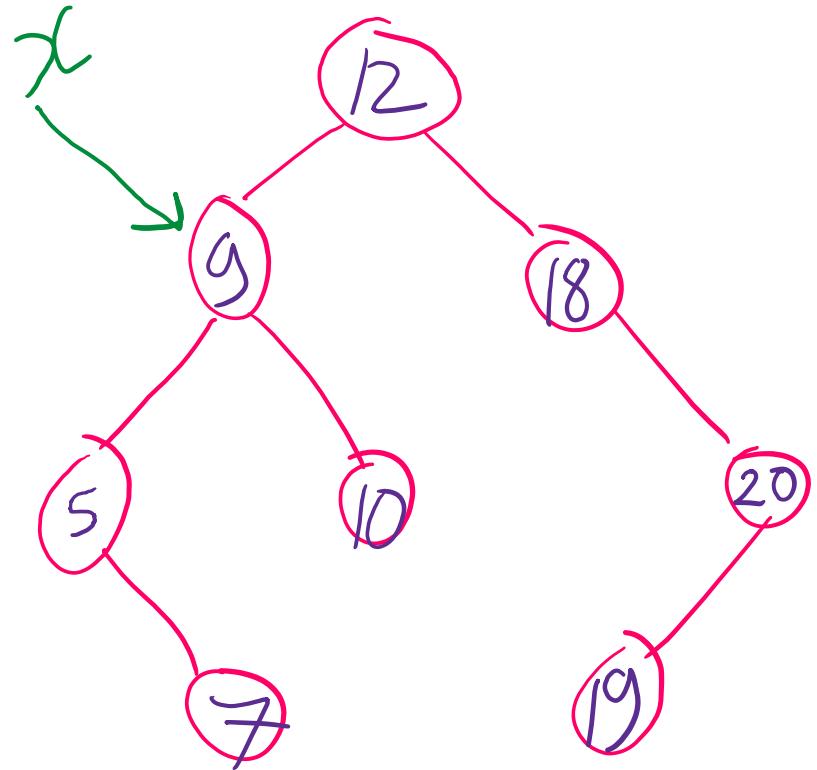
Visit order : 12

Iterative Preorder traversal



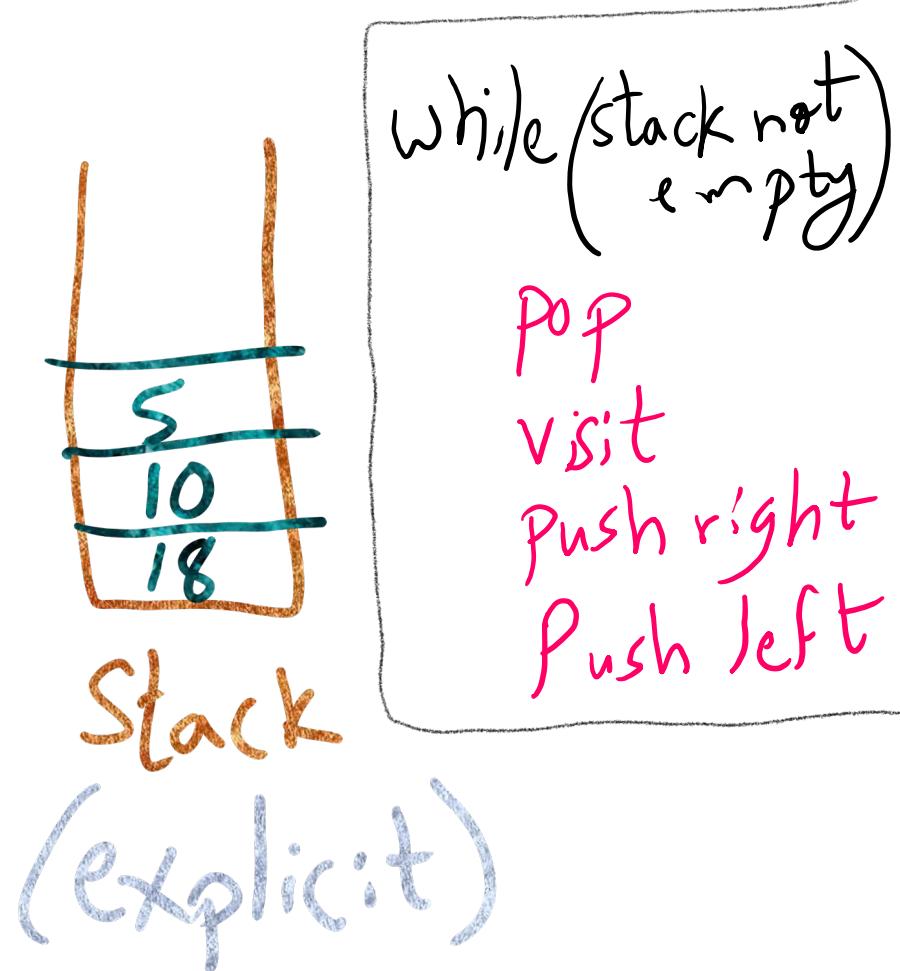
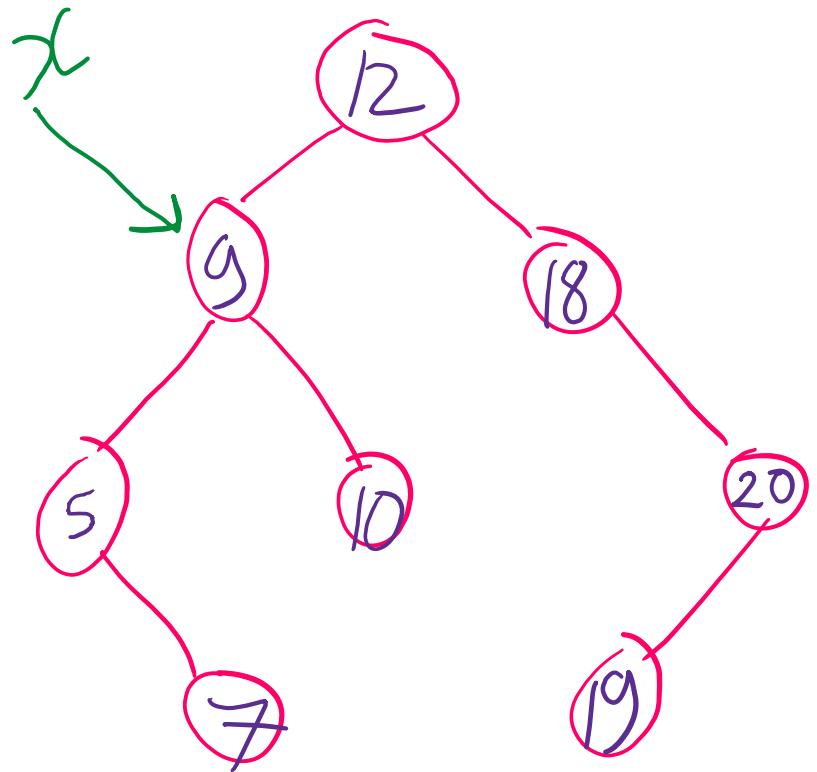
Visit order : 12

Iterative Preorder traversal



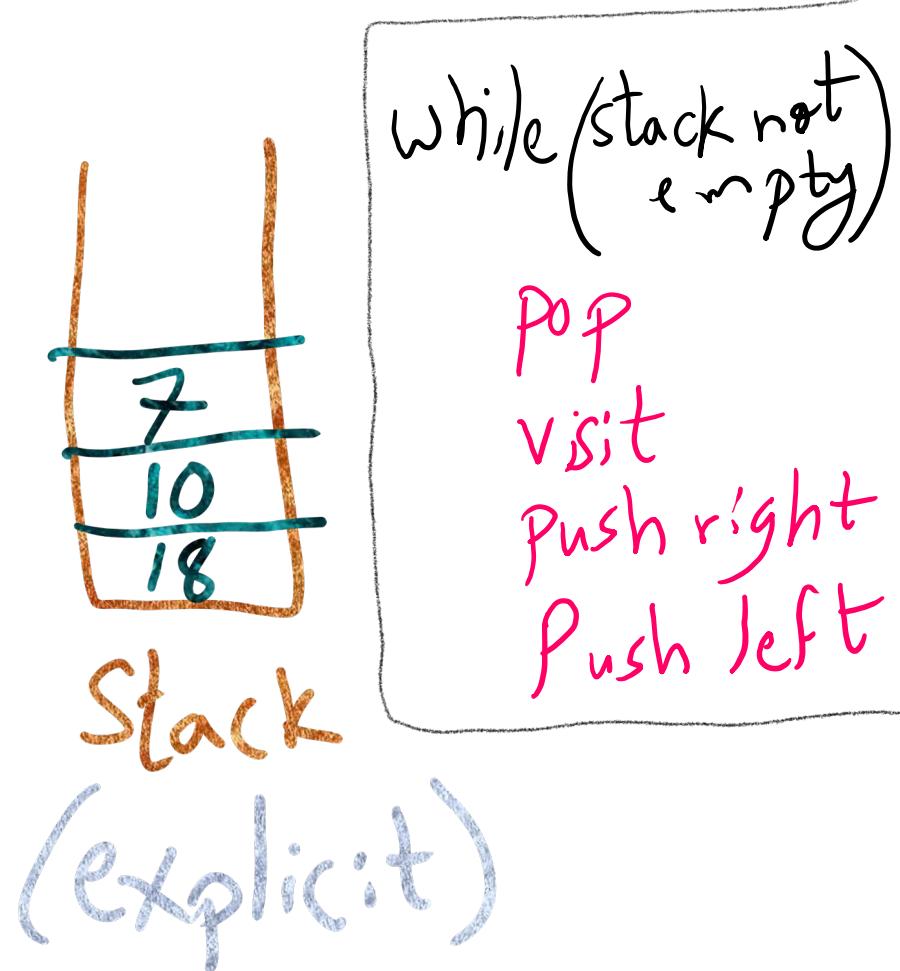
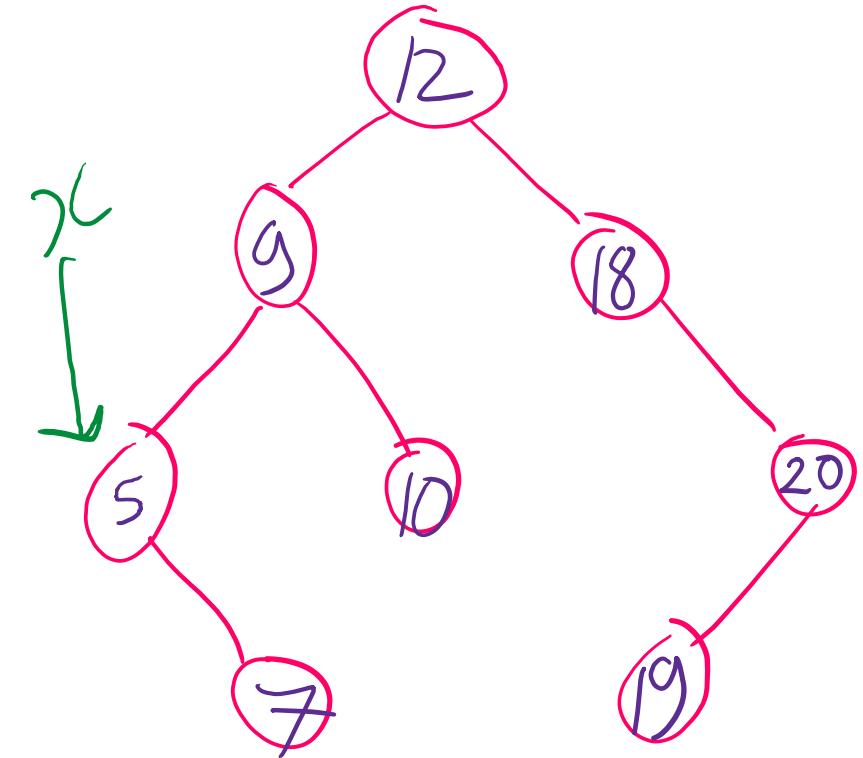
Visit order : 12, 9

Iterative Preorder traversal



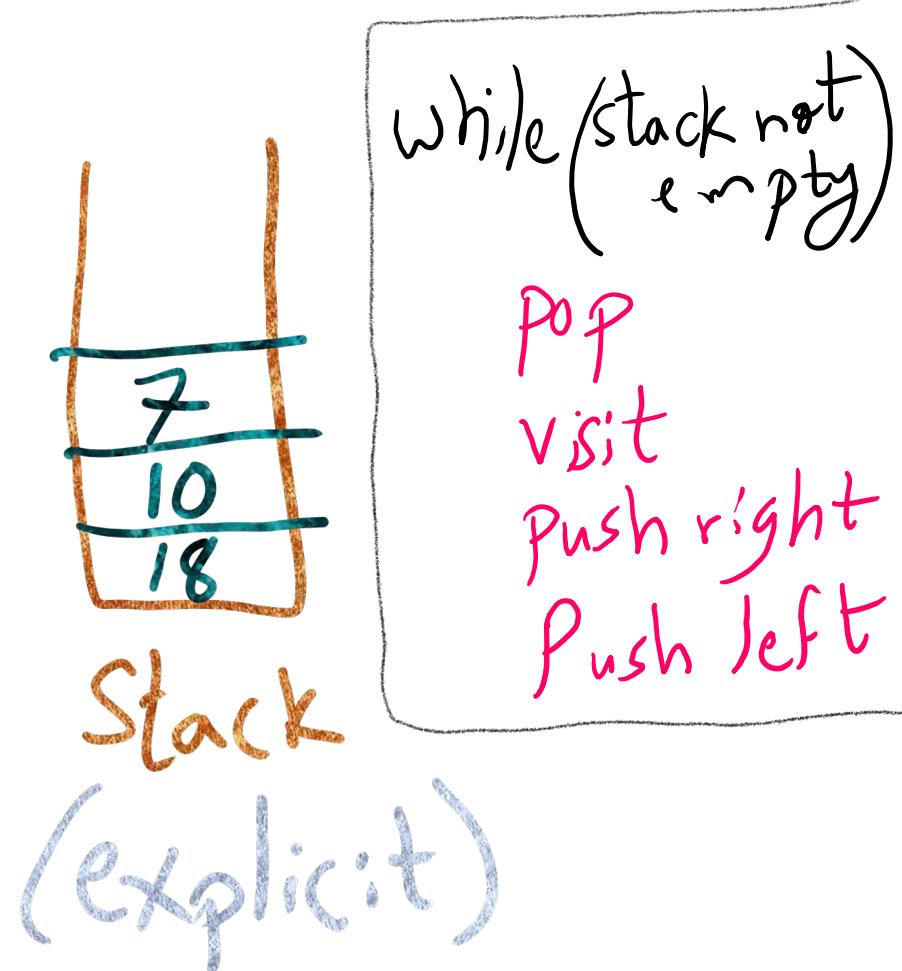
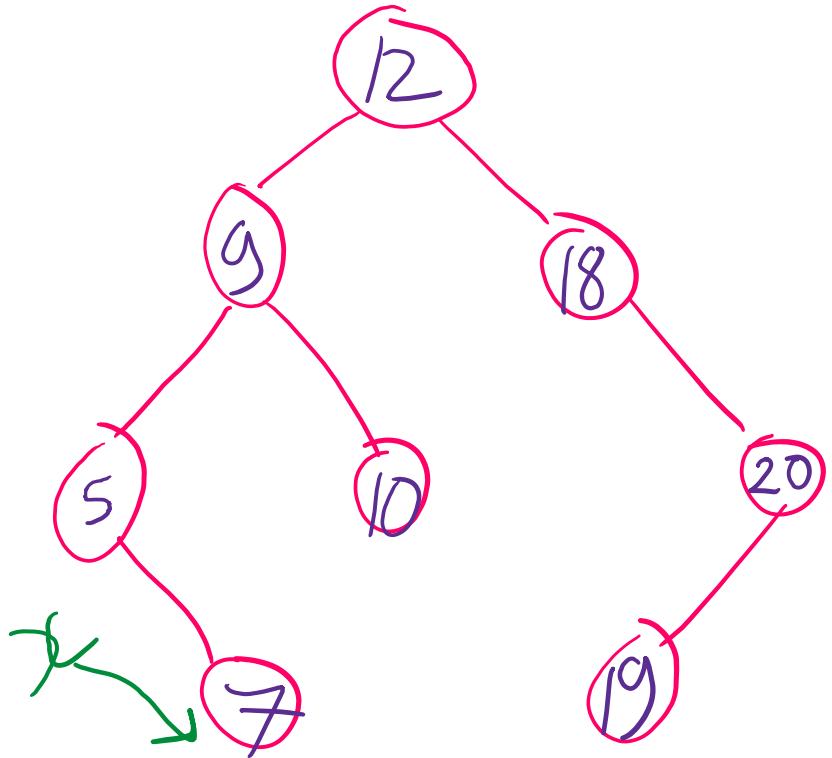
Visit order : 12, 9

Iterative Preorder traversal



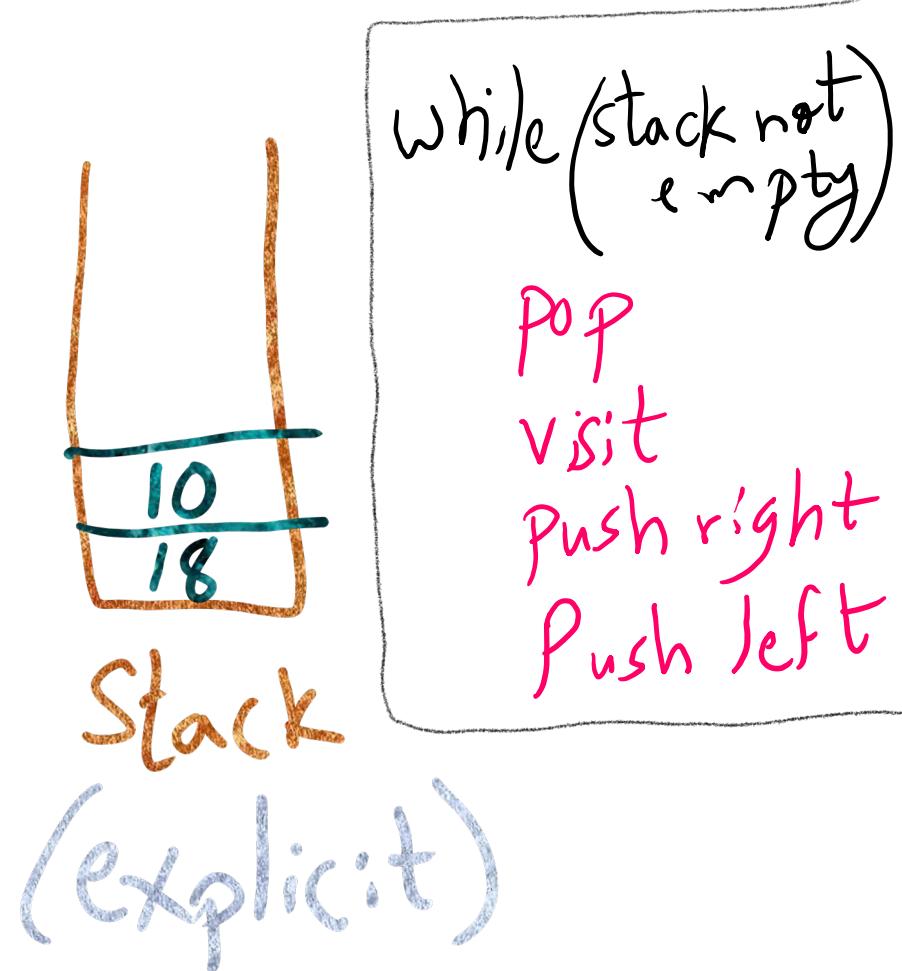
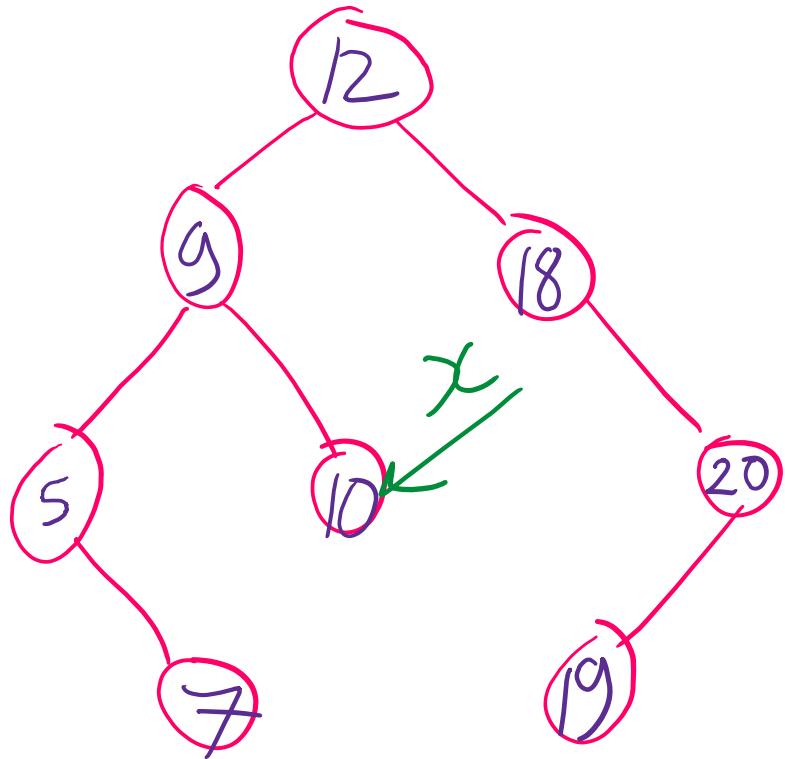
Visit order : 12, 9, 5

Iterative Preorder traversal



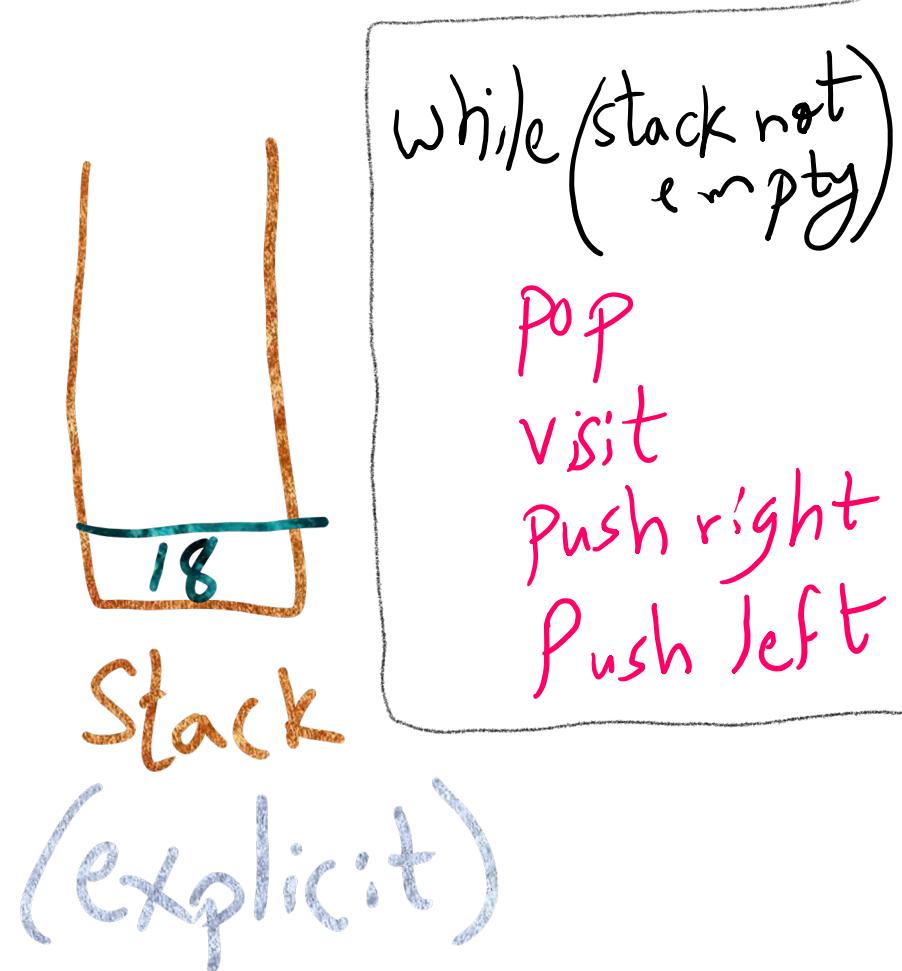
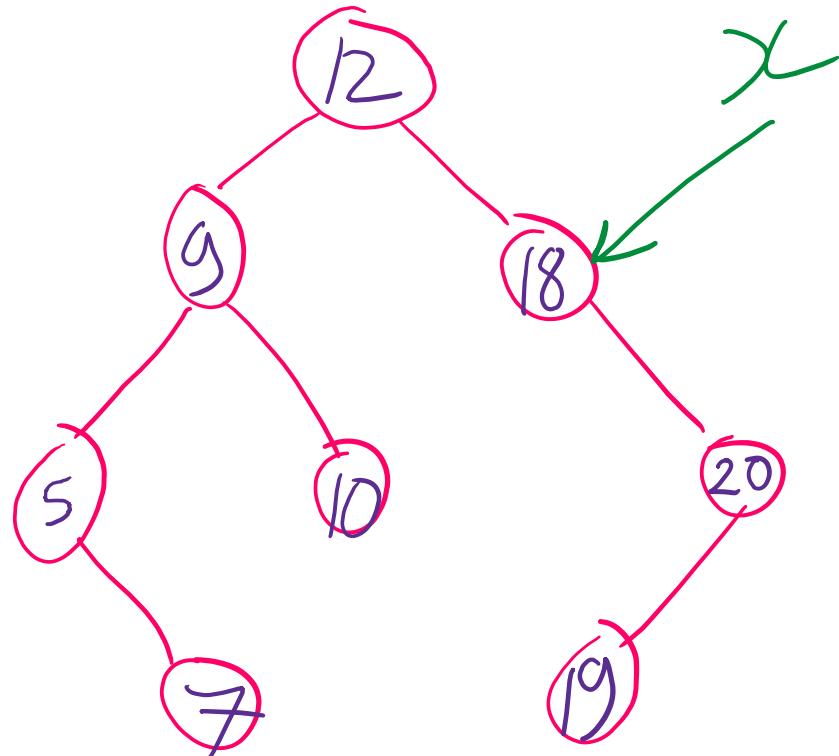
Visit order : 12, 9, 5, 7

Iterative Preorder traversal



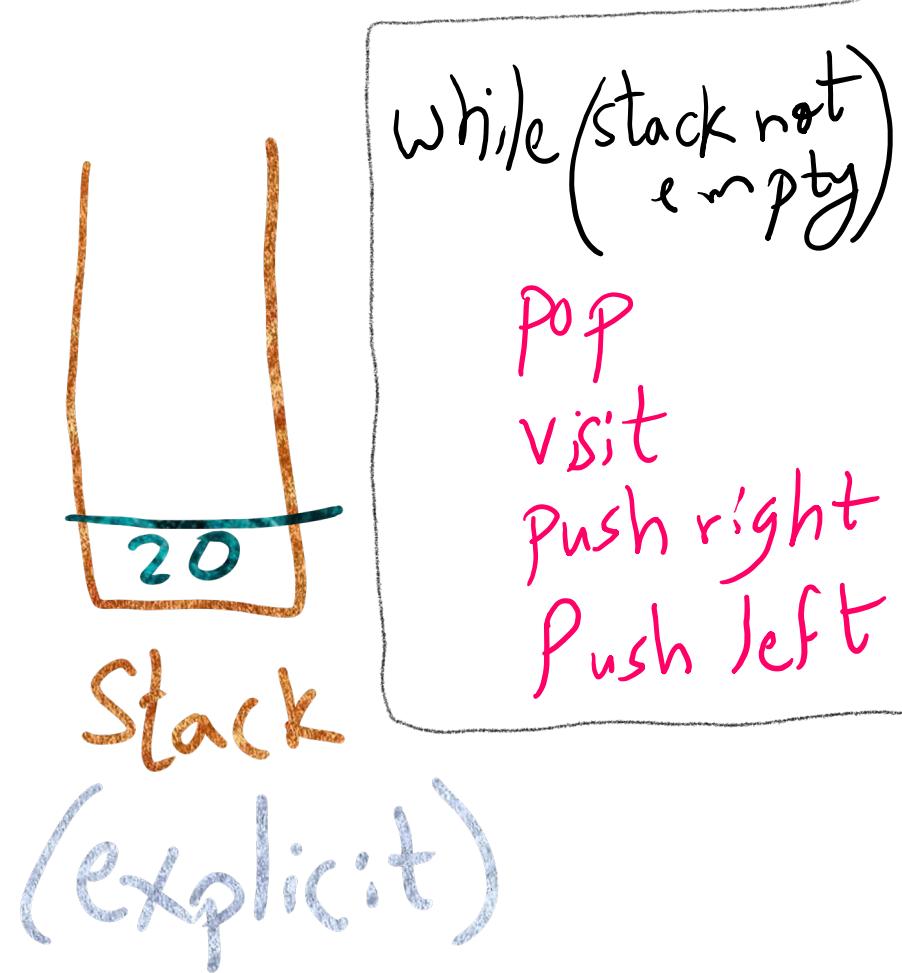
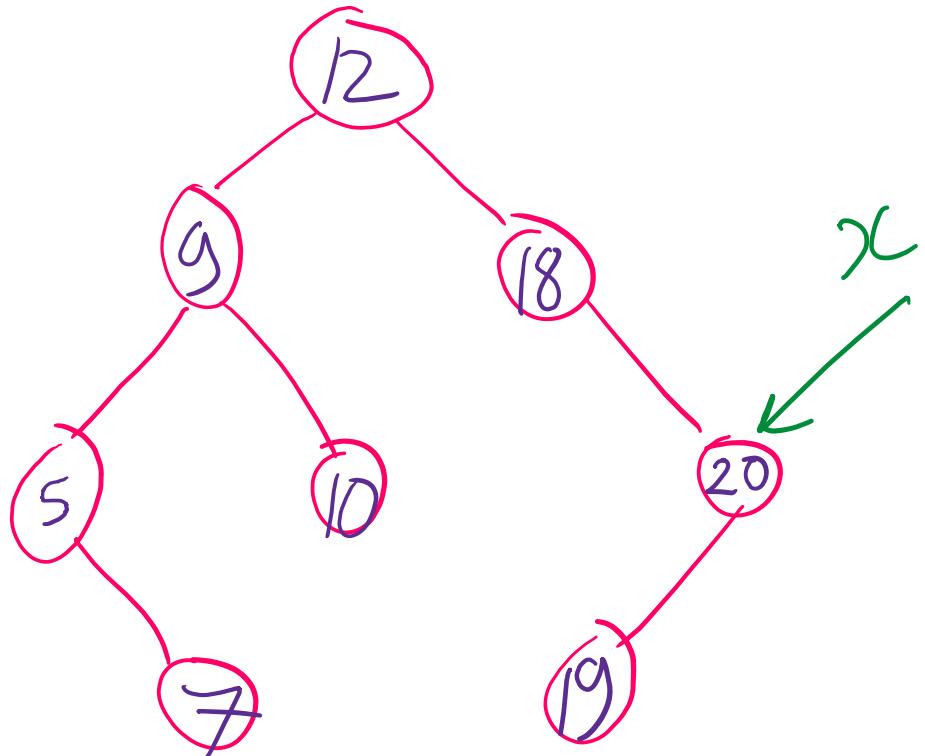
Visit order : 12, 9, 5, 7, 10

Iterative Preorder traversal



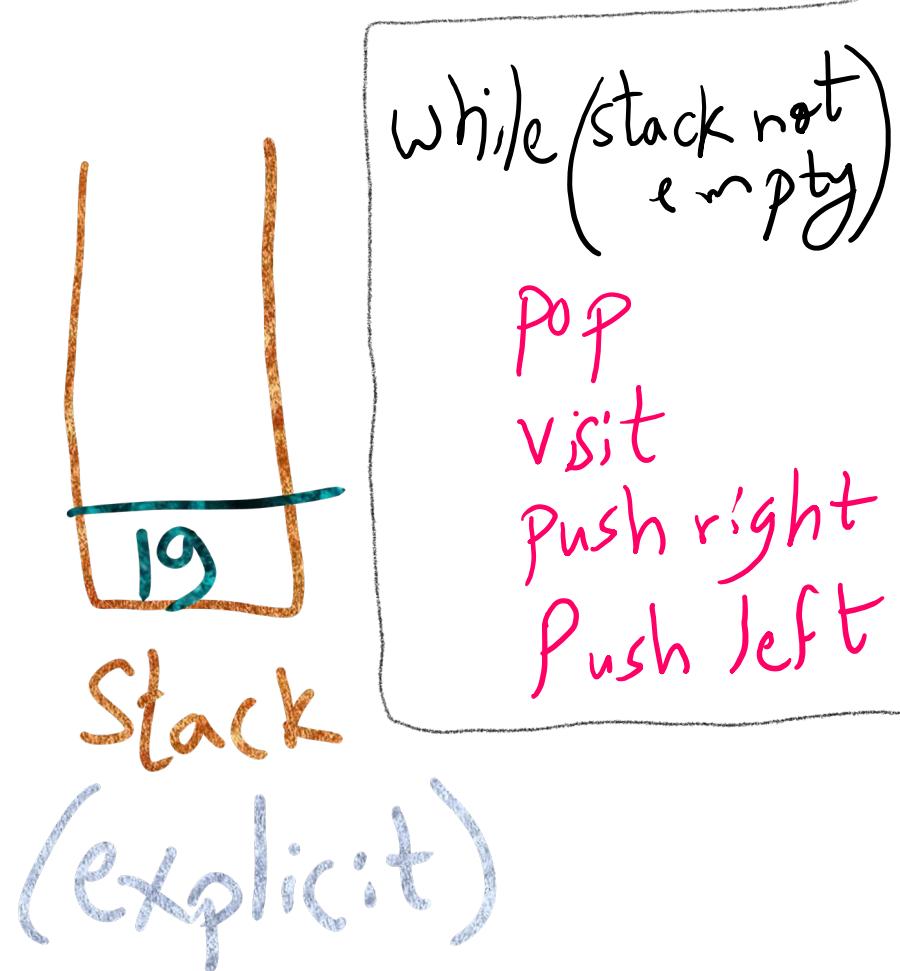
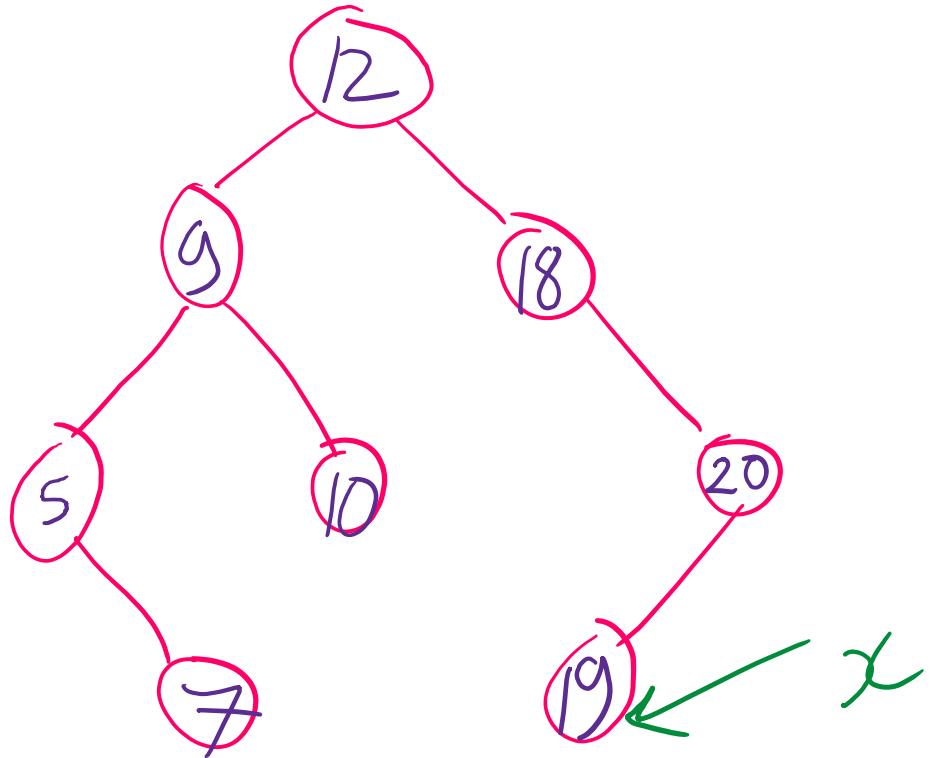
Visit order : 12, 9, 5, 7, 10, 18

Iterative Preorder traversal



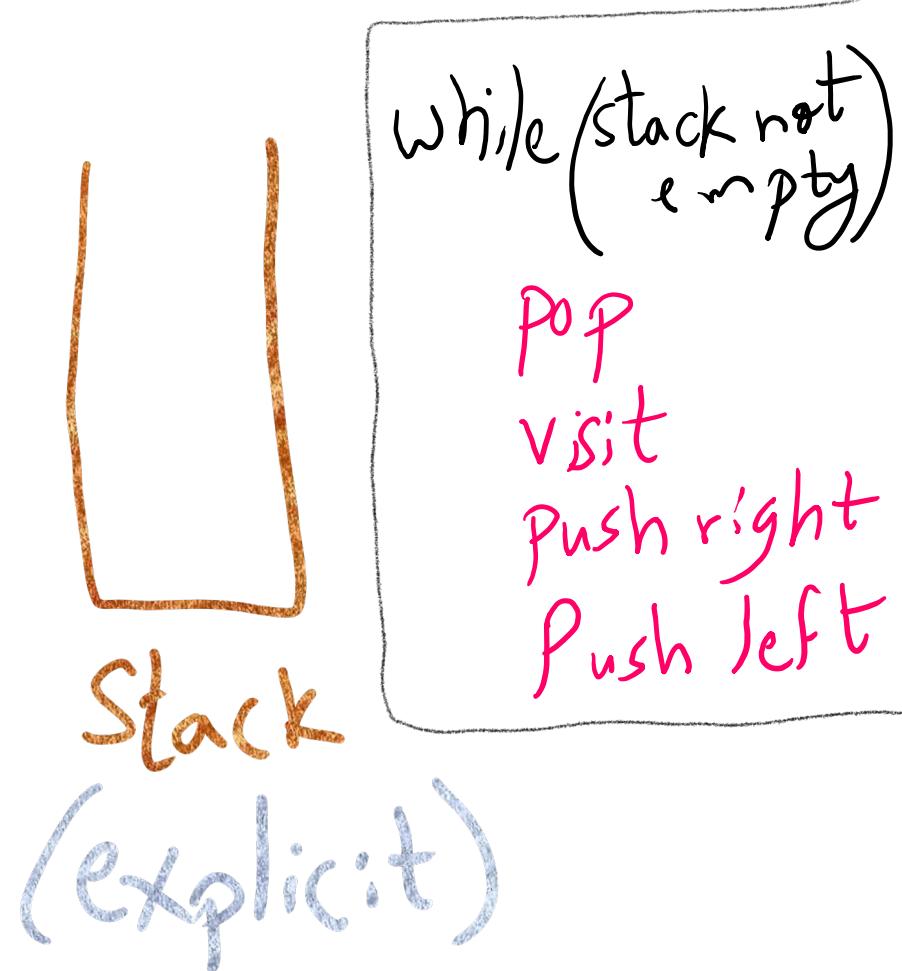
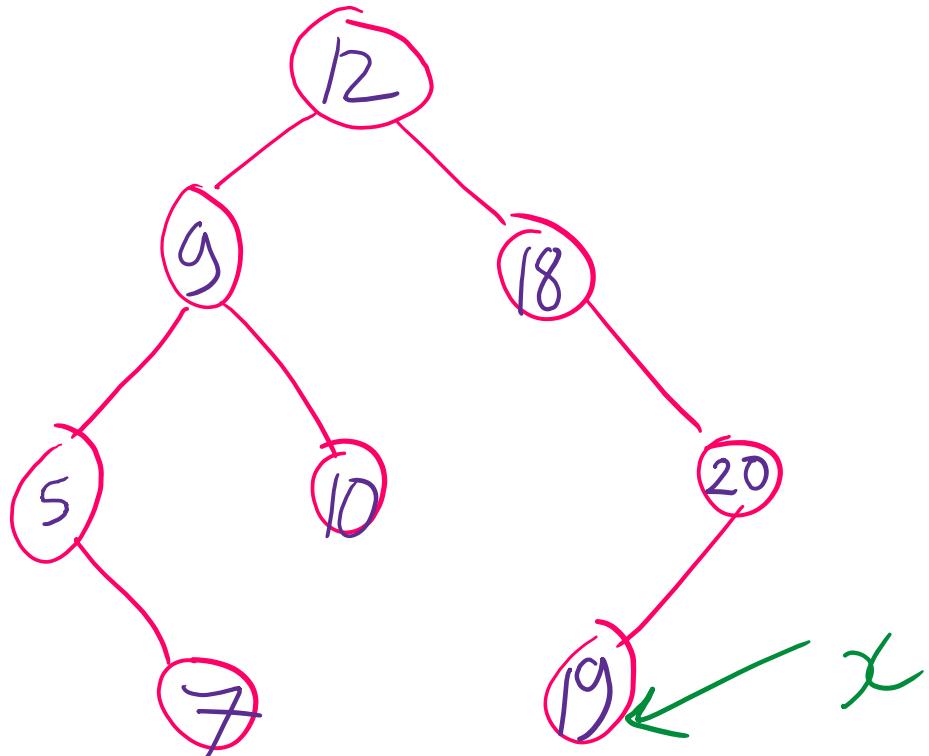
Visit order : 12, 9, 5, 7, 10, 18, 20

Iterative Preorder traversal



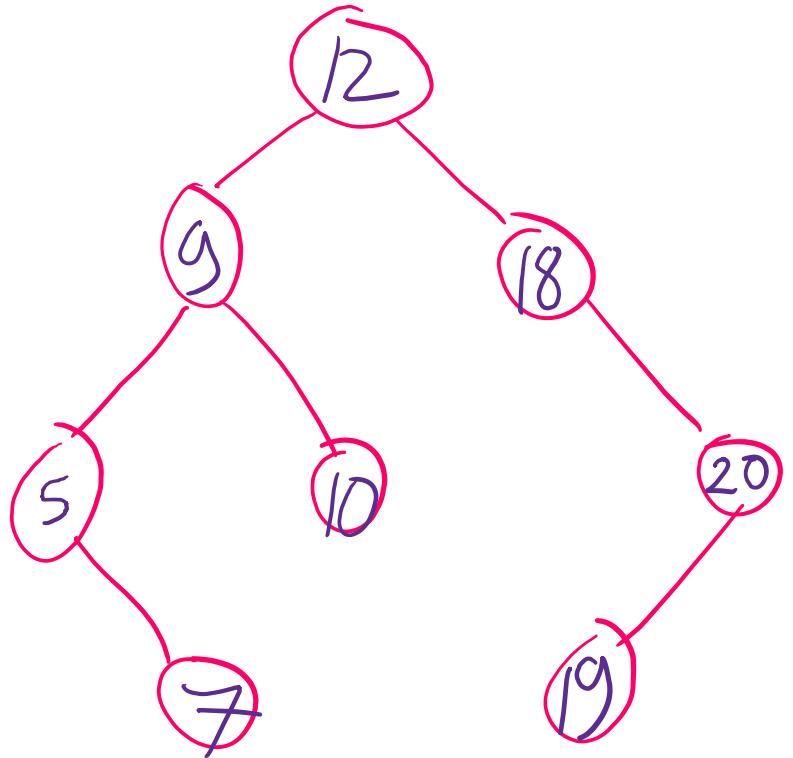
Visit order : 12, 9, 5, 7, 10, 18, 20, 19

Iterative Preorder traversal



Visit order : 12, 9, 5, 7, 10, 18, 20, 19

Iterative Inorder traversal



Visit order :

repeat until stack empty & $x == \text{null}$

$x = \text{leftmost node}$

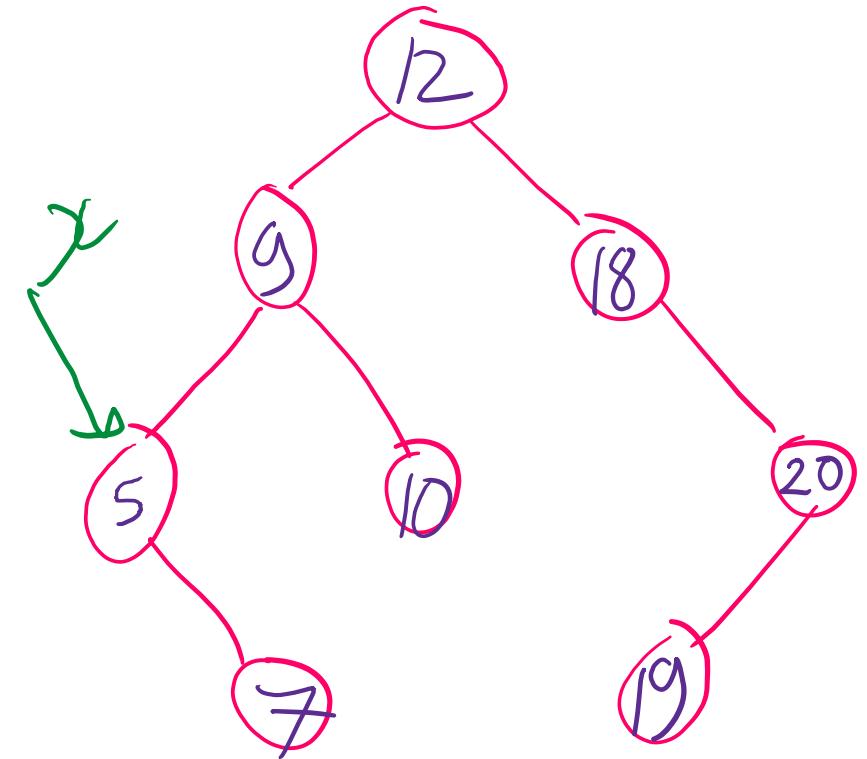
push to stack while moving down

pop & visit

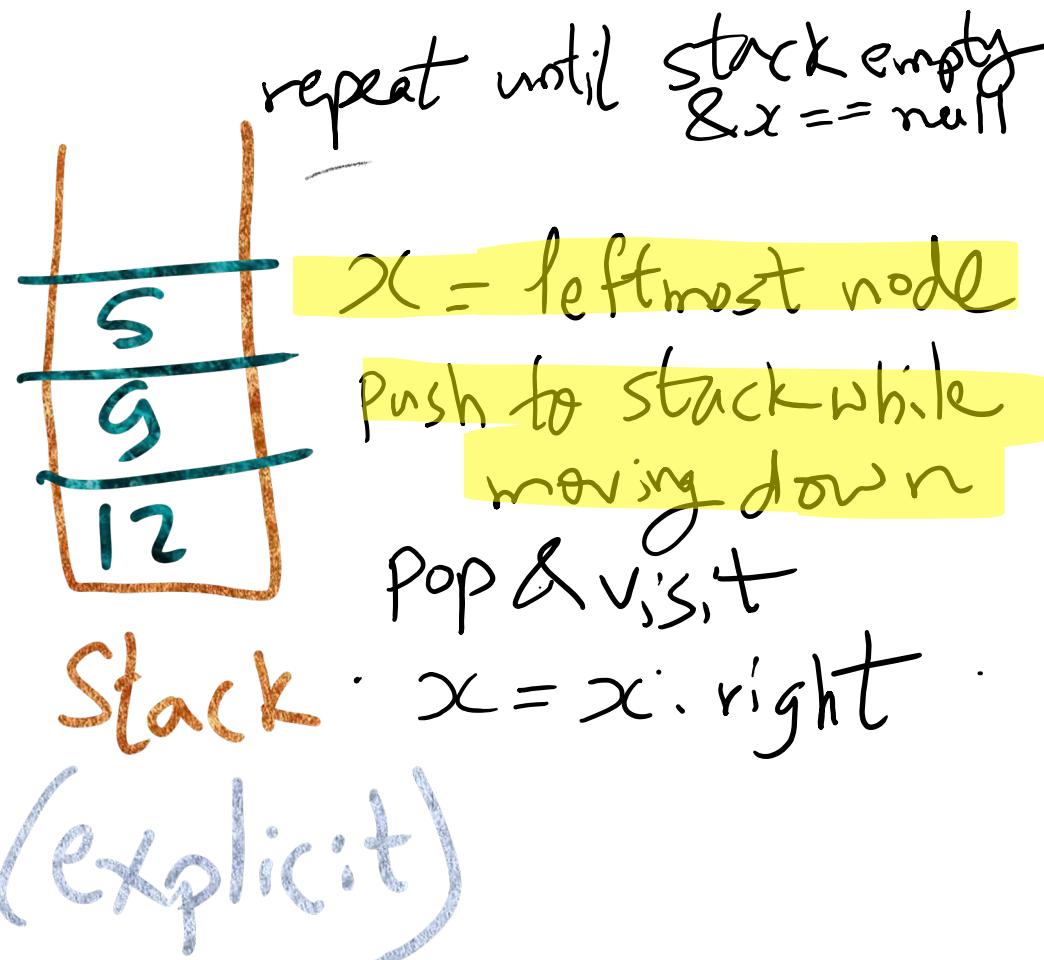
$x = x.\text{right}$

Stack (explicit)

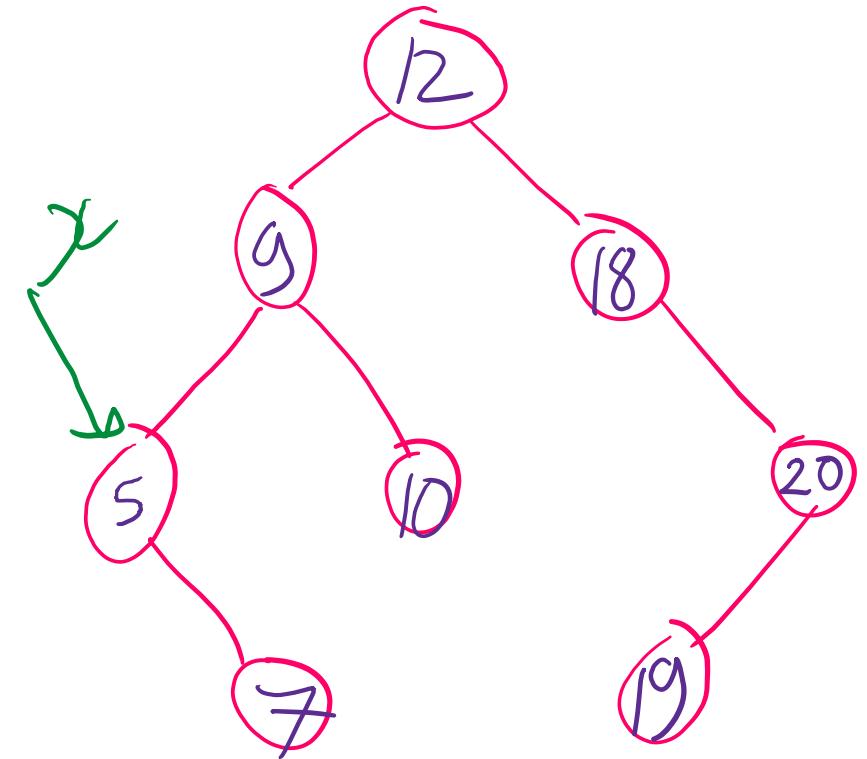
Iterative Inorder traversal



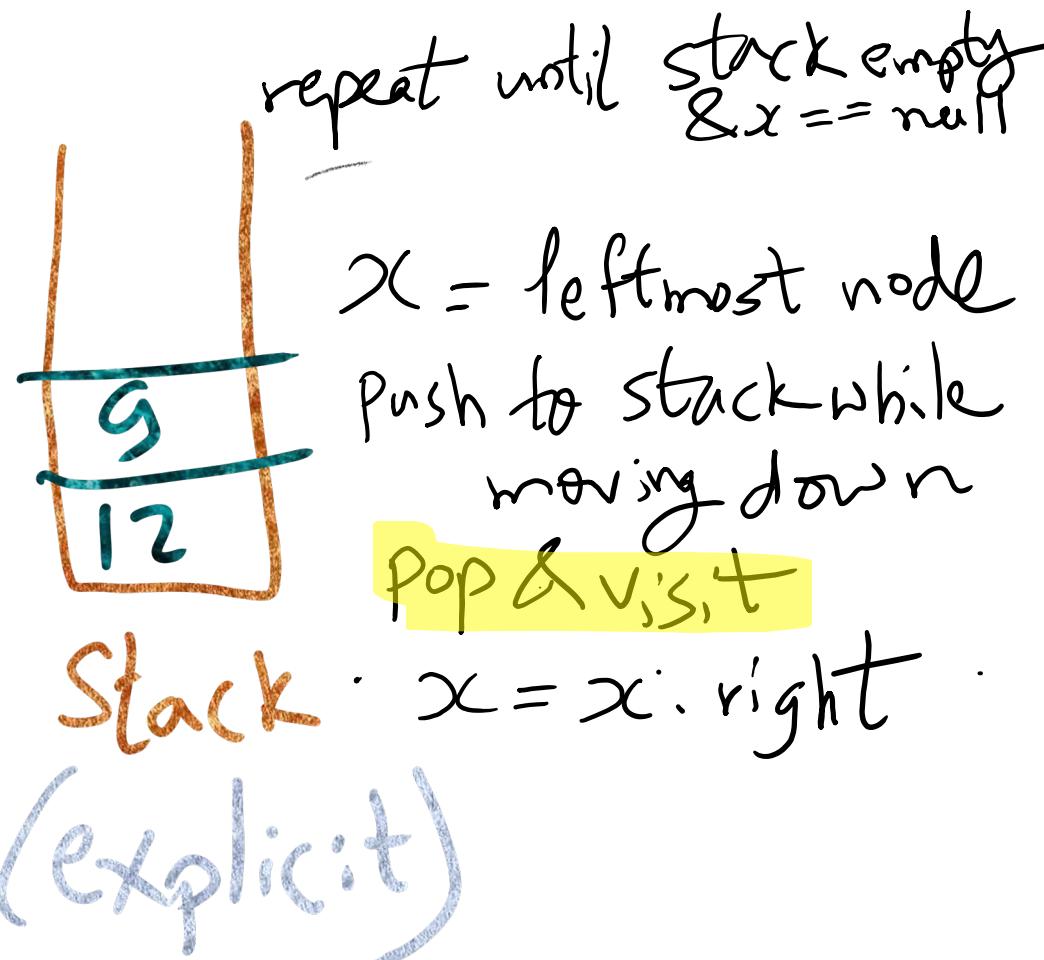
Visit order :



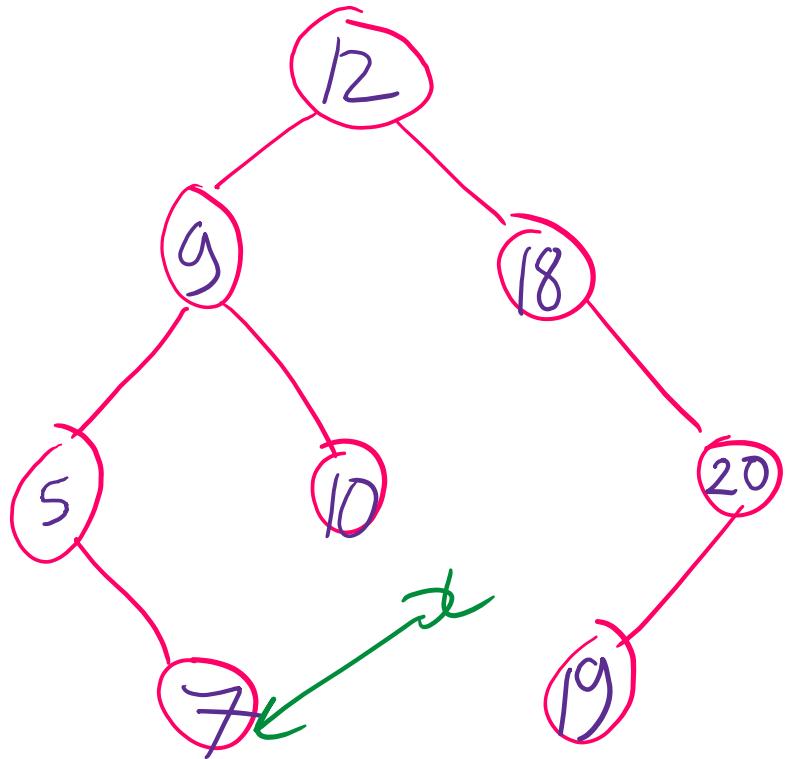
Iterative Inorder traversal



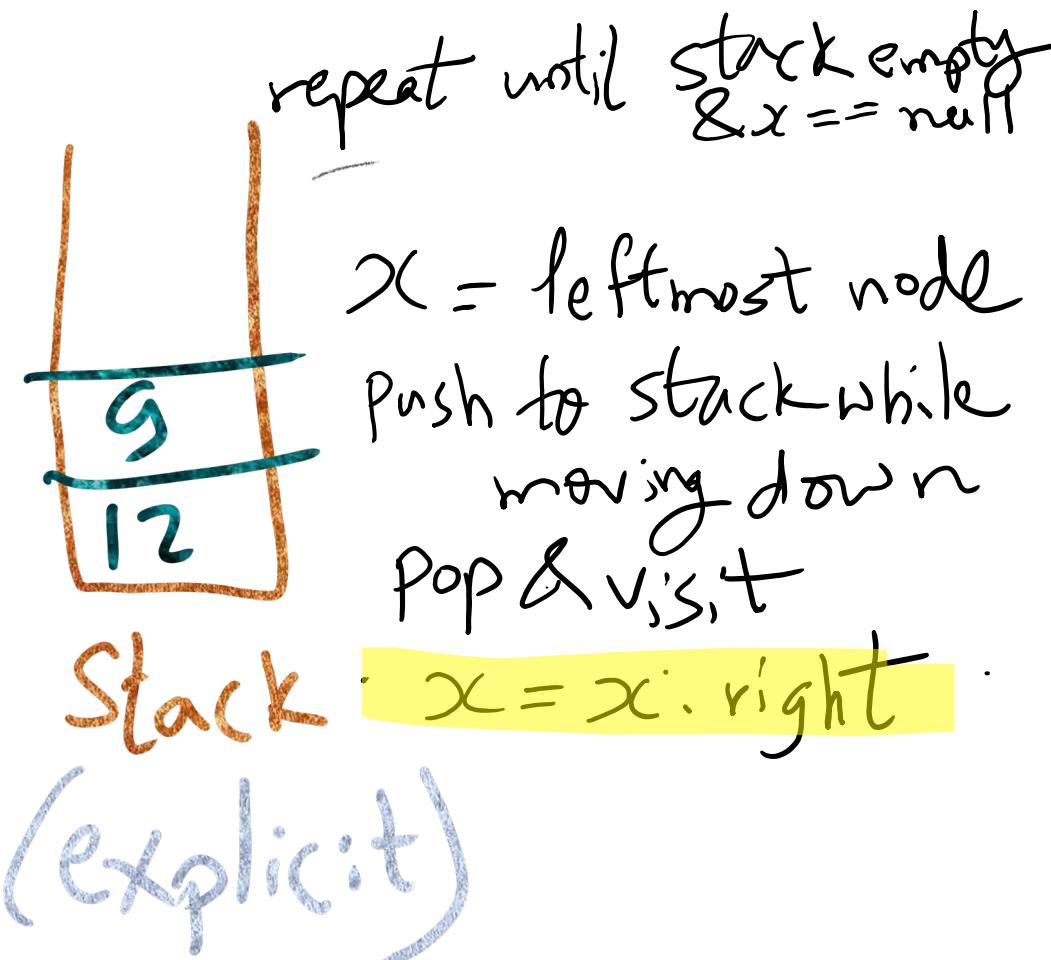
Visit order : 5



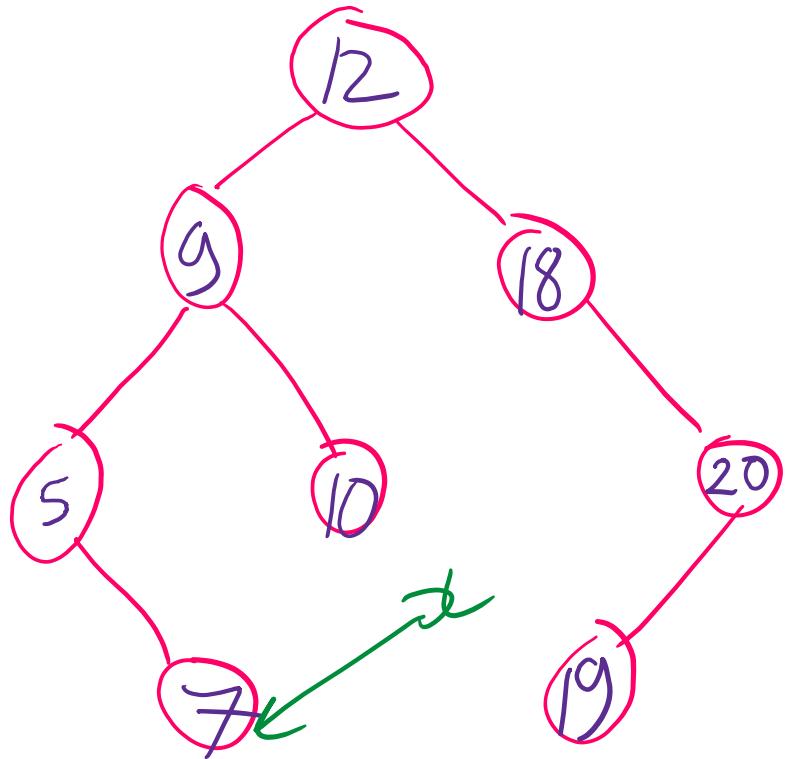
Iterative Inorder traversal



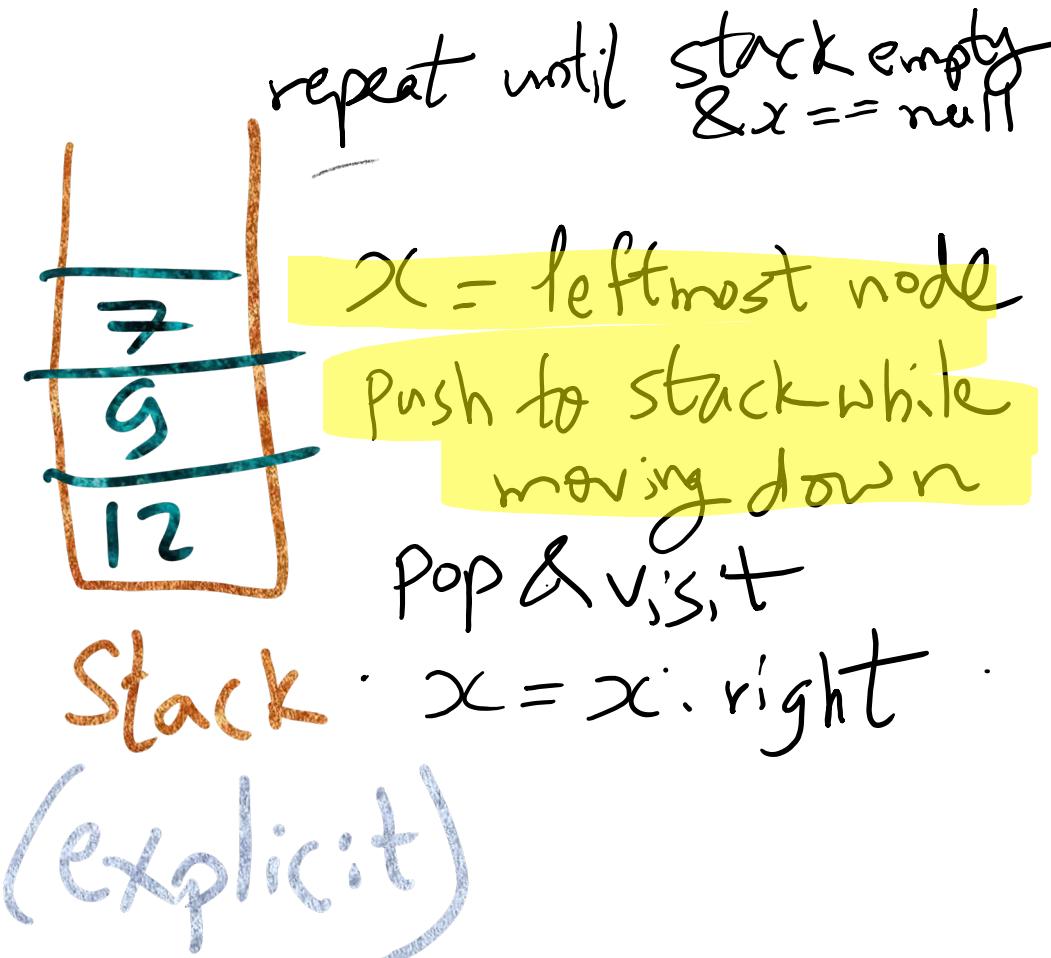
Visit order : 5



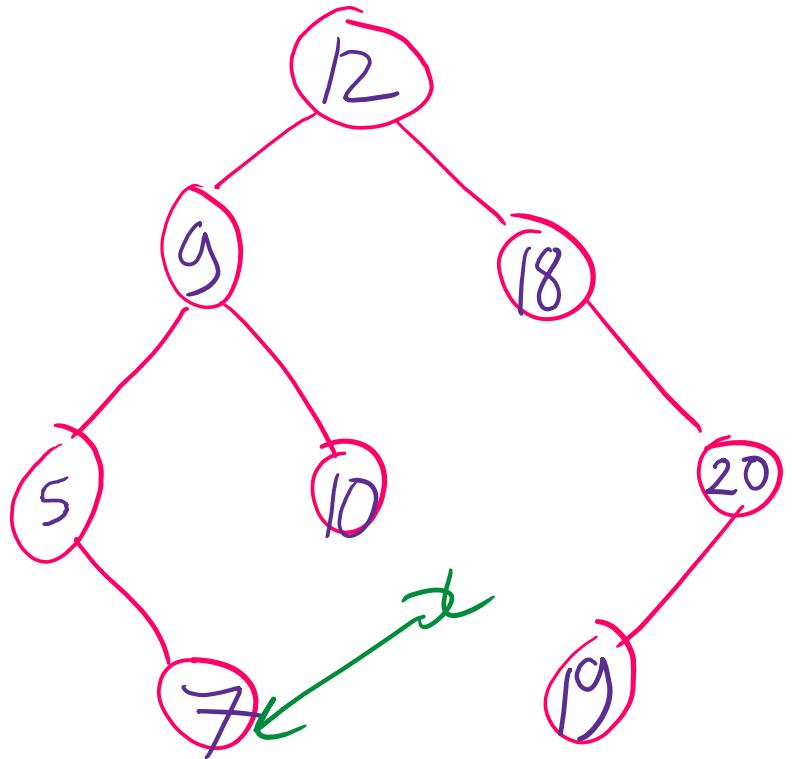
Iterative Inorder traversal



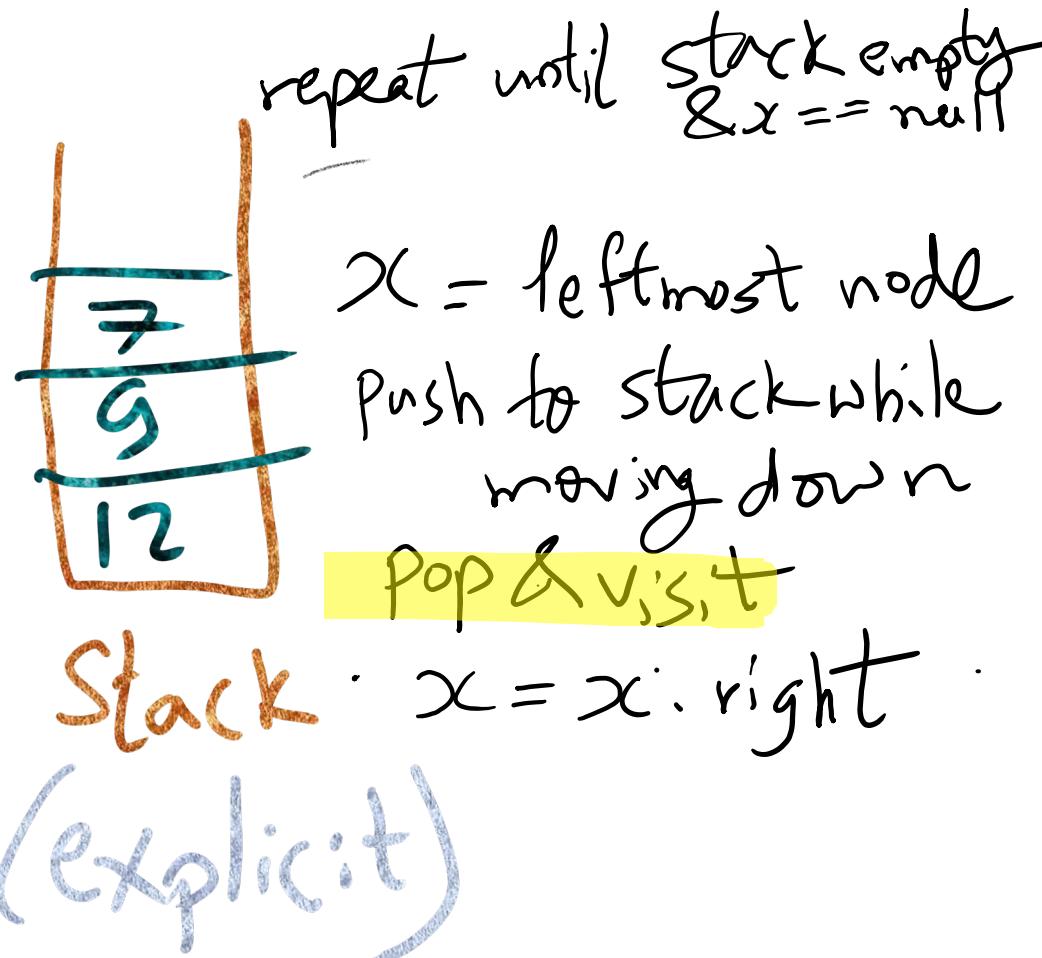
Visit order : 5



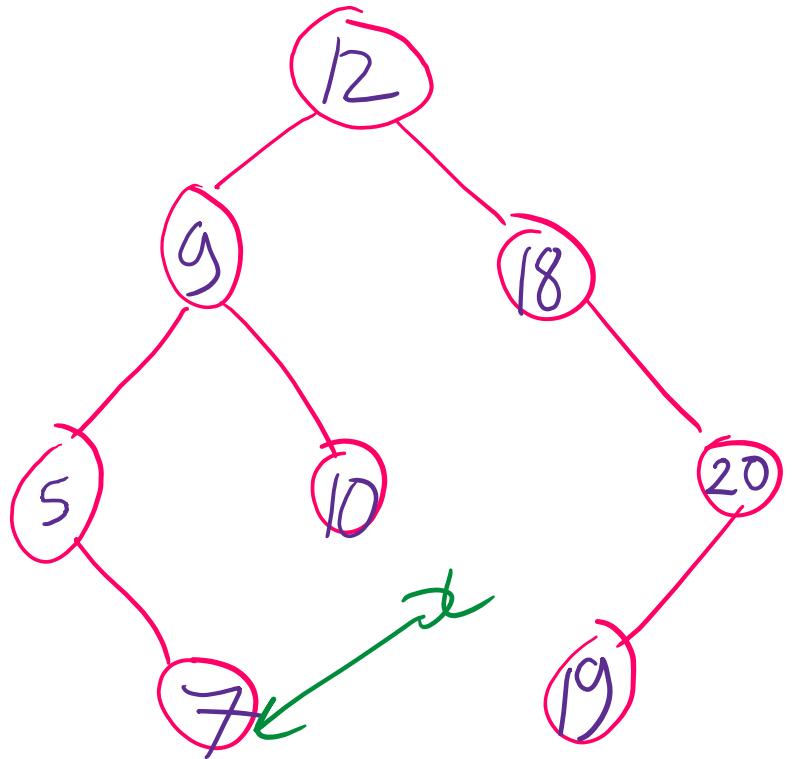
Iterative Inorder traversal



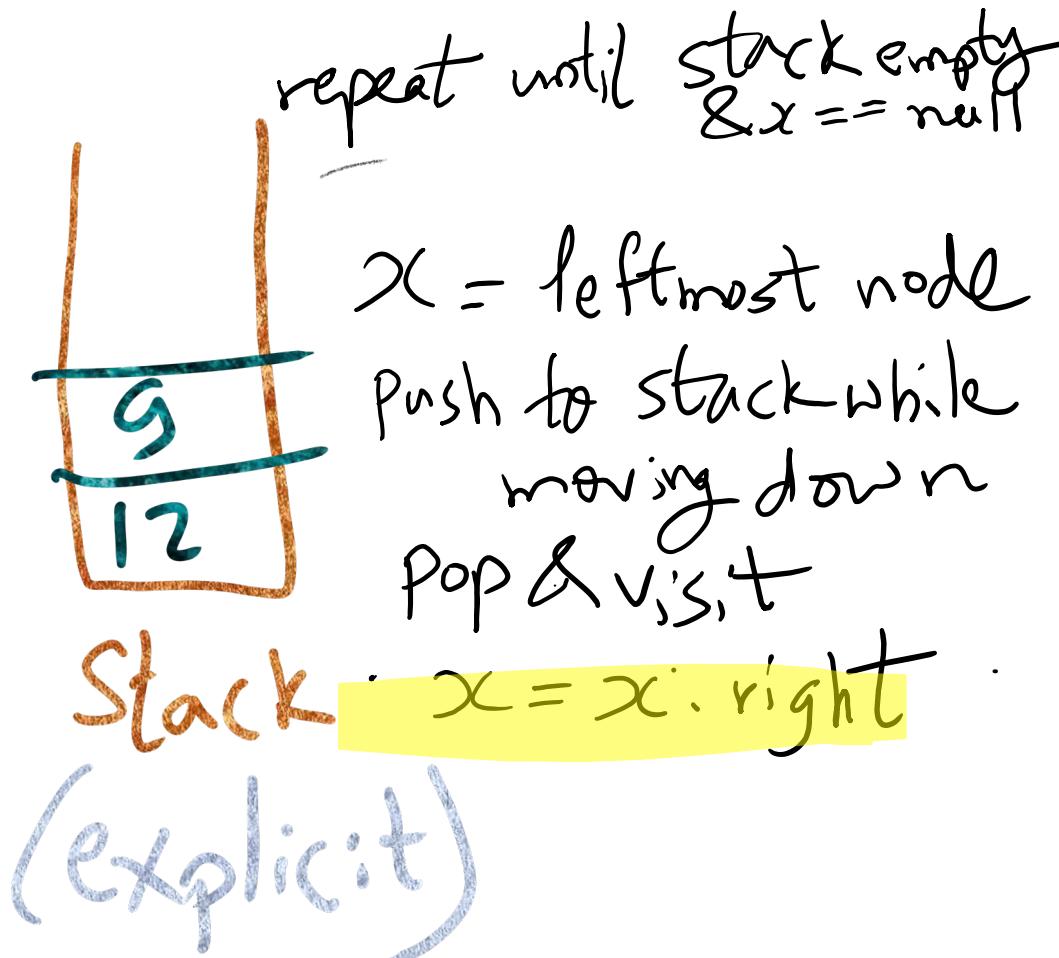
Visit order : 5, 7



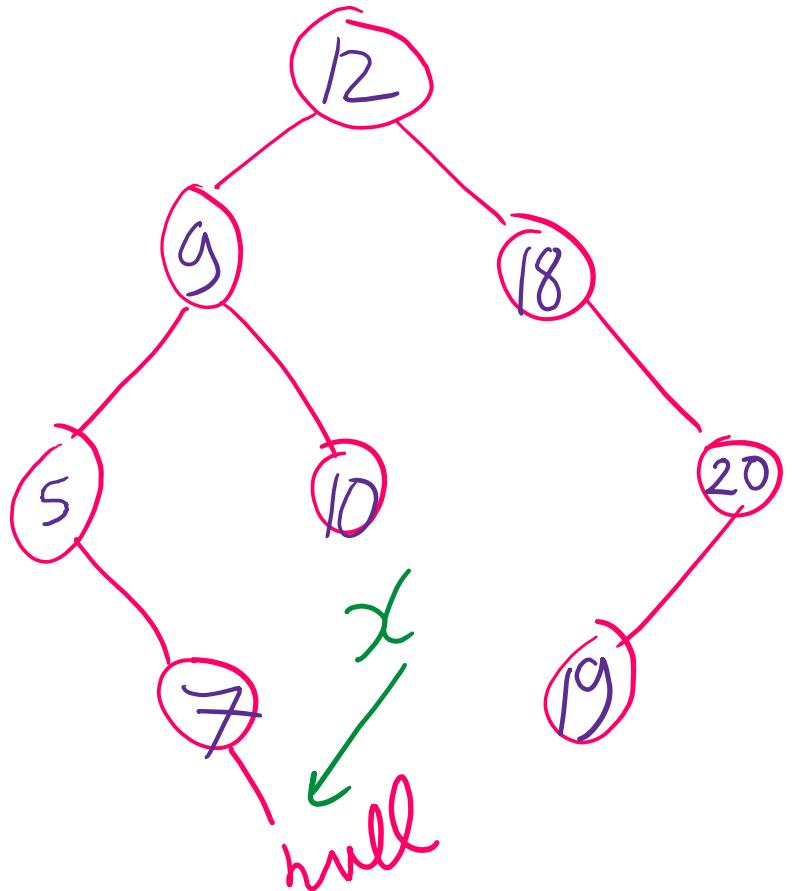
Iterative Inorder traversal



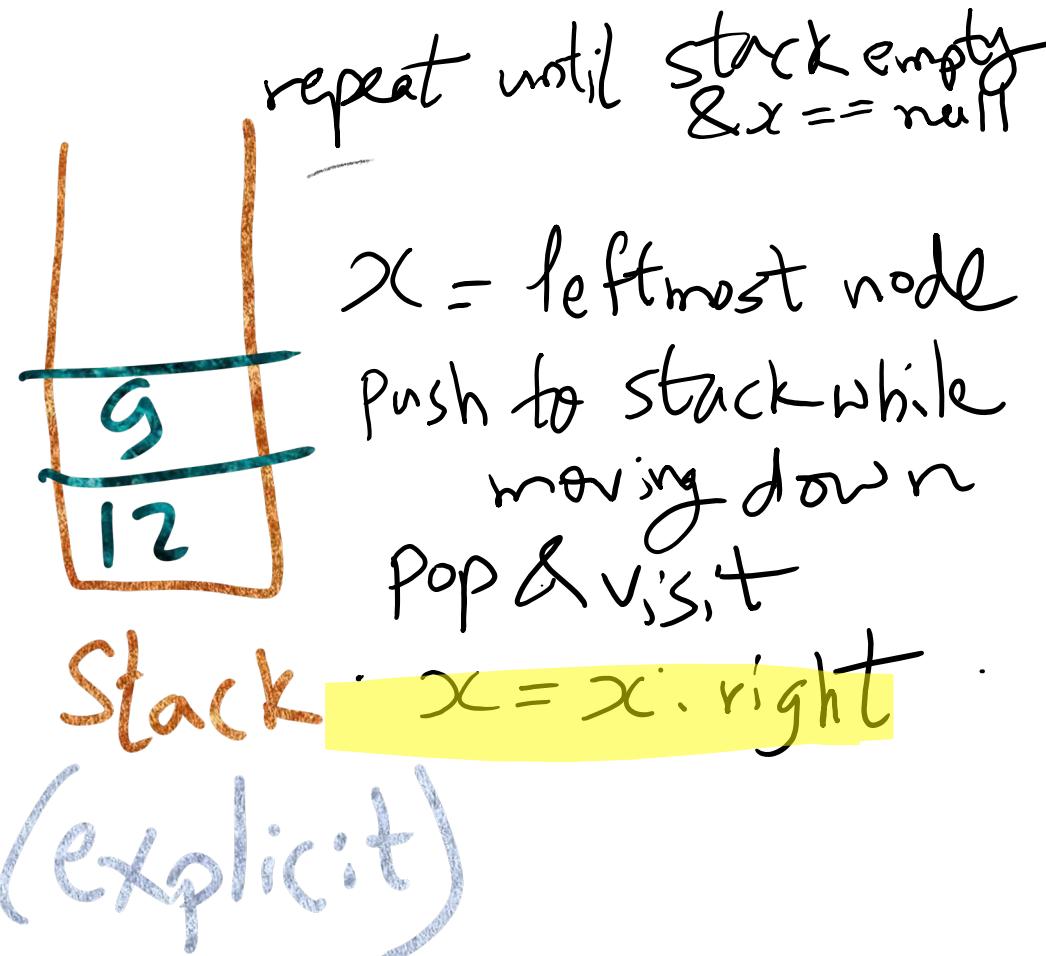
Visit order : 5, 7



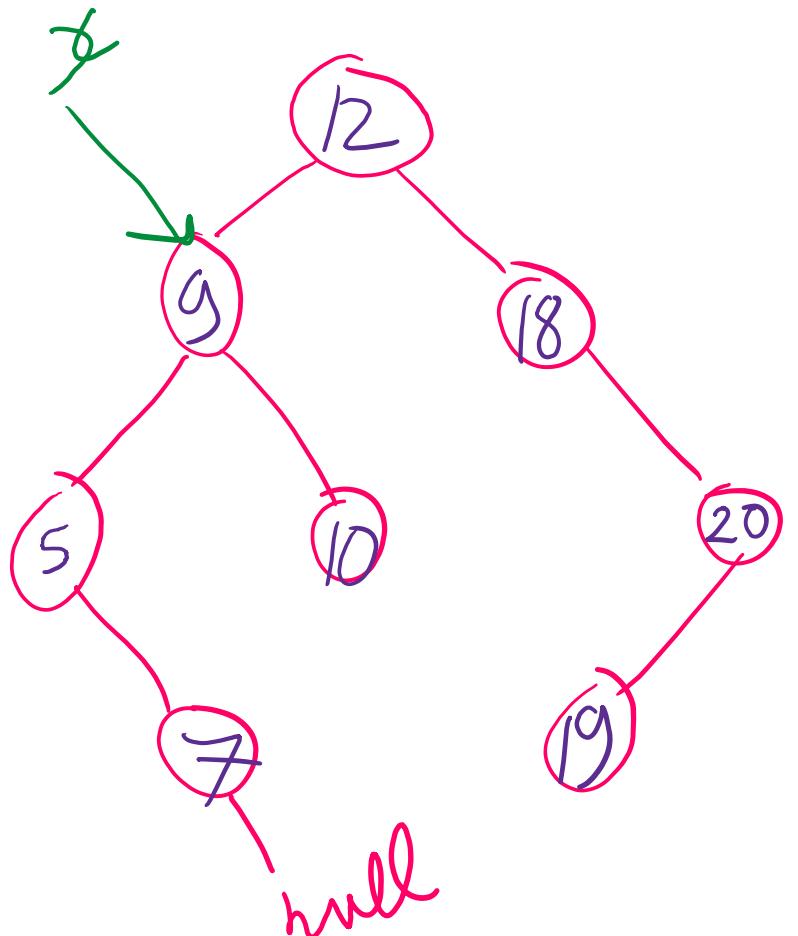
Iterative Inorder traversal



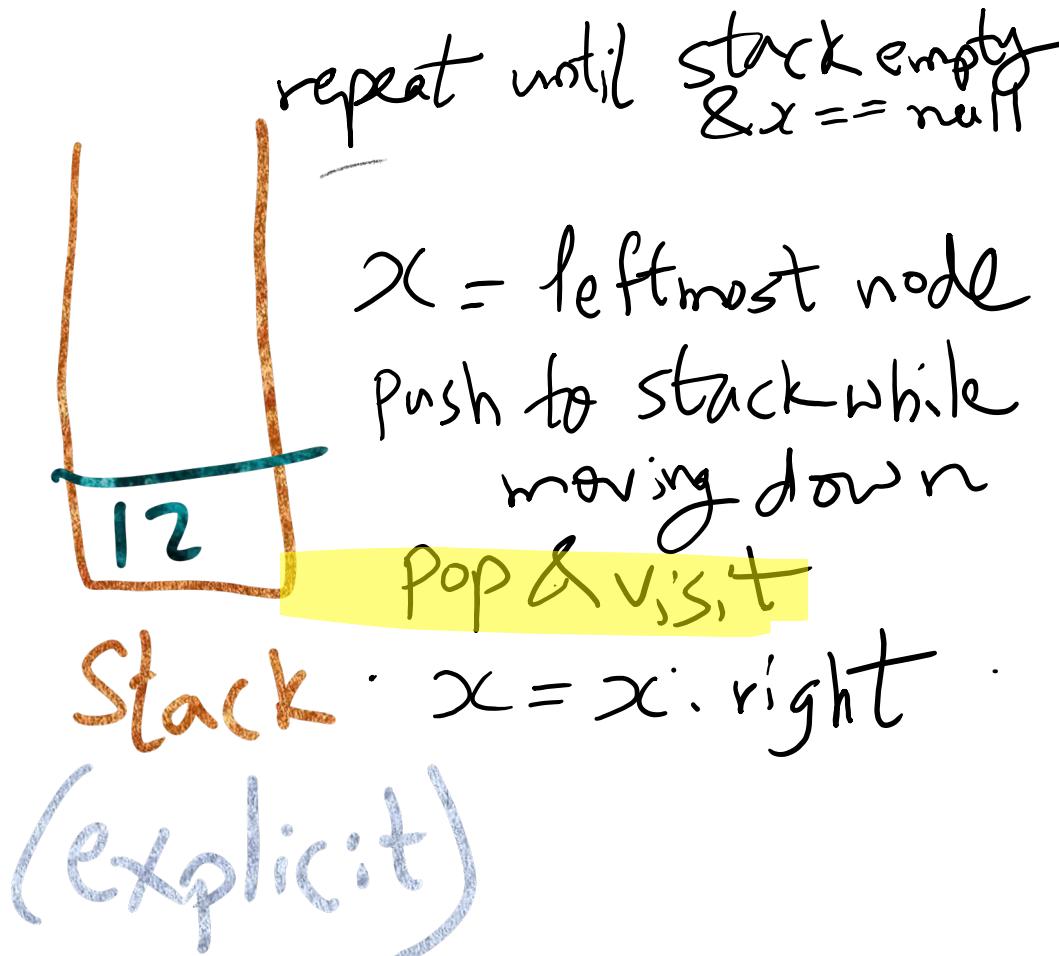
Visit order : 5, 7



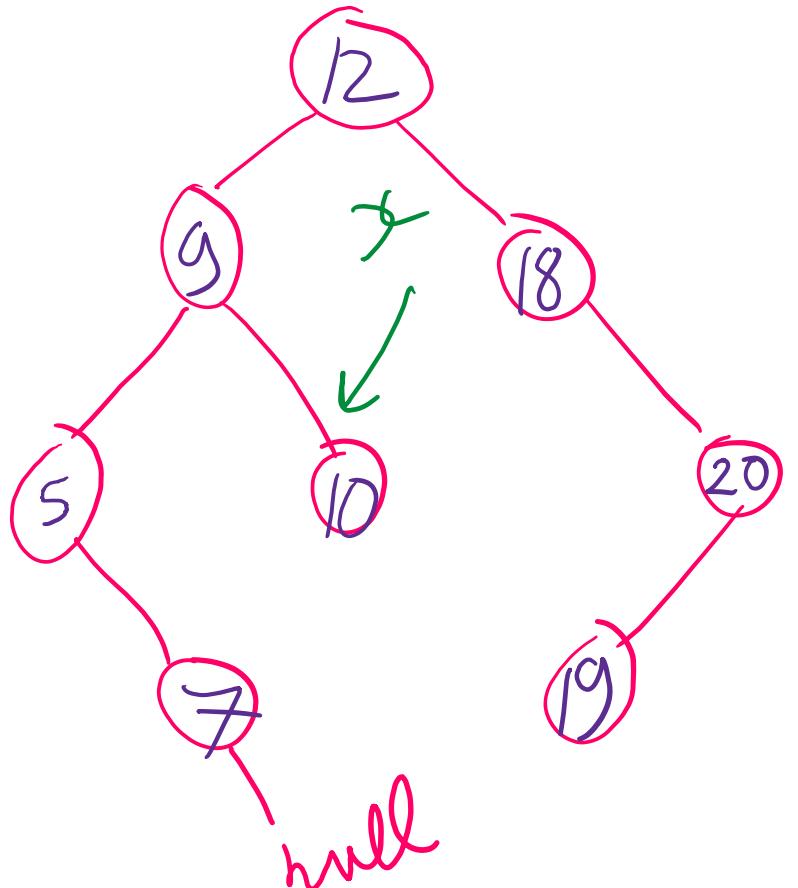
Iterative Inorder traversal



Visit order : 5, 7, 9



Iterative Inorder traversal



Visit order : 5, 7, 9

repeat until stack empty & $x == \text{null}$

$x = \text{leftmost node}$

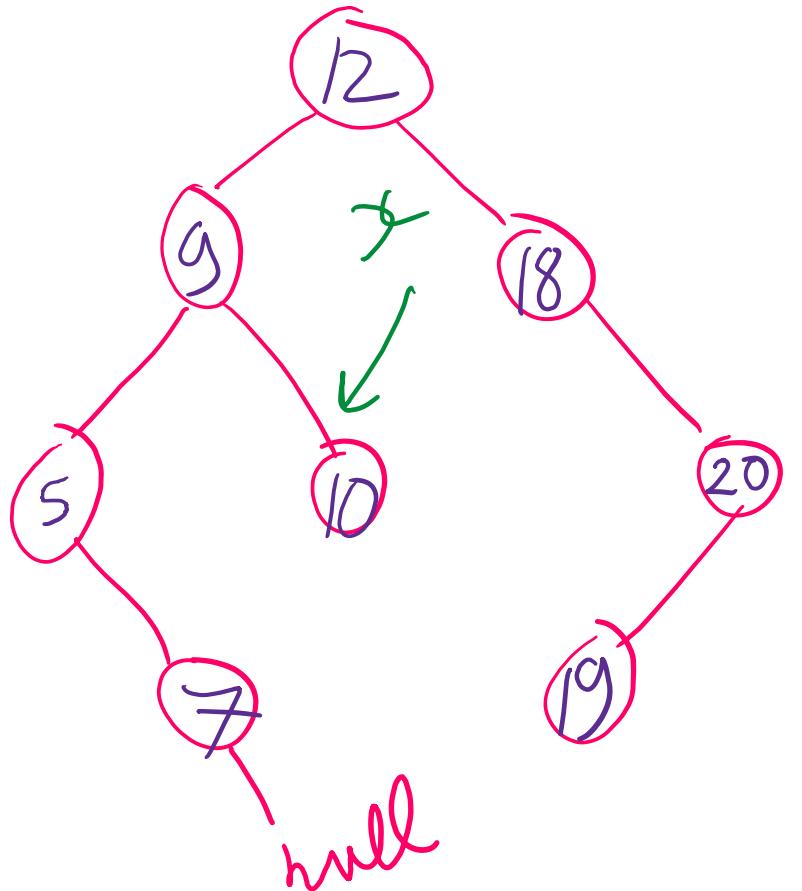
push to stack while moving down

pop & visit

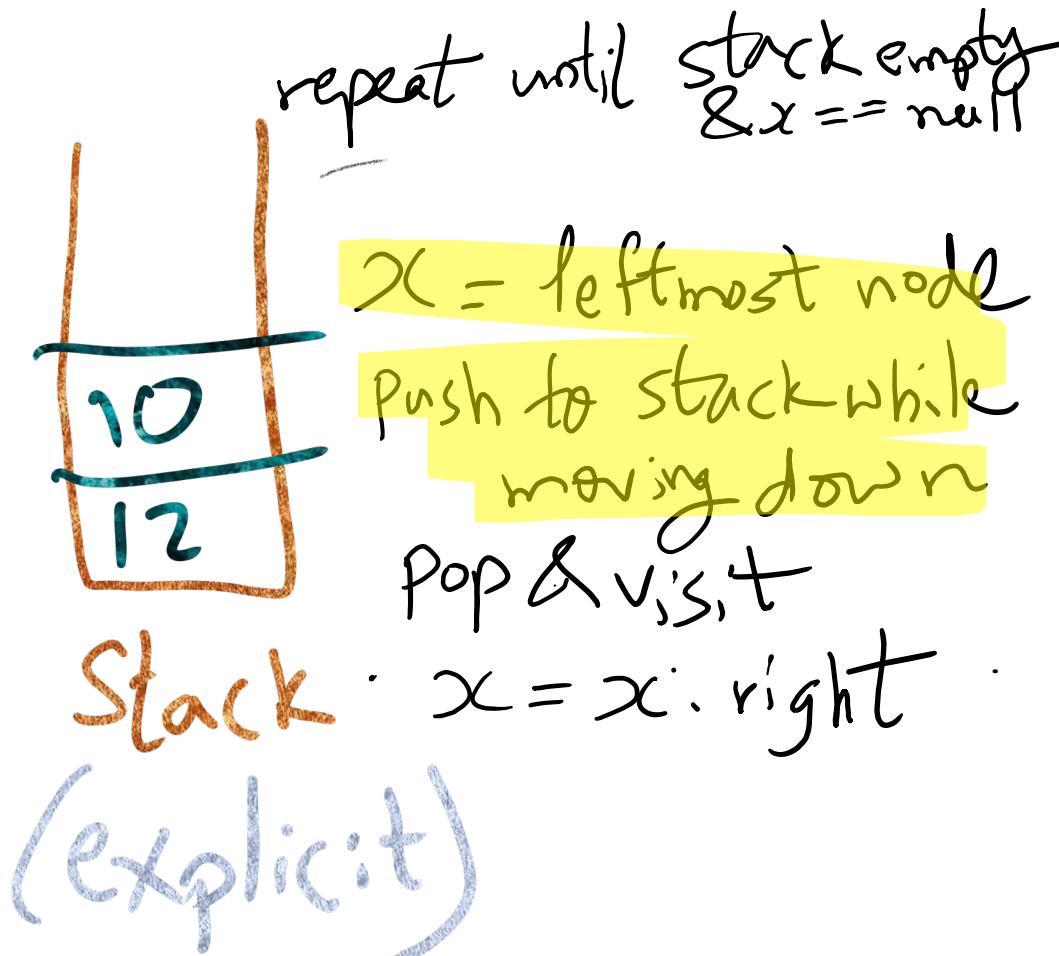
Stack : $x = x.\text{right}$

(explicit)

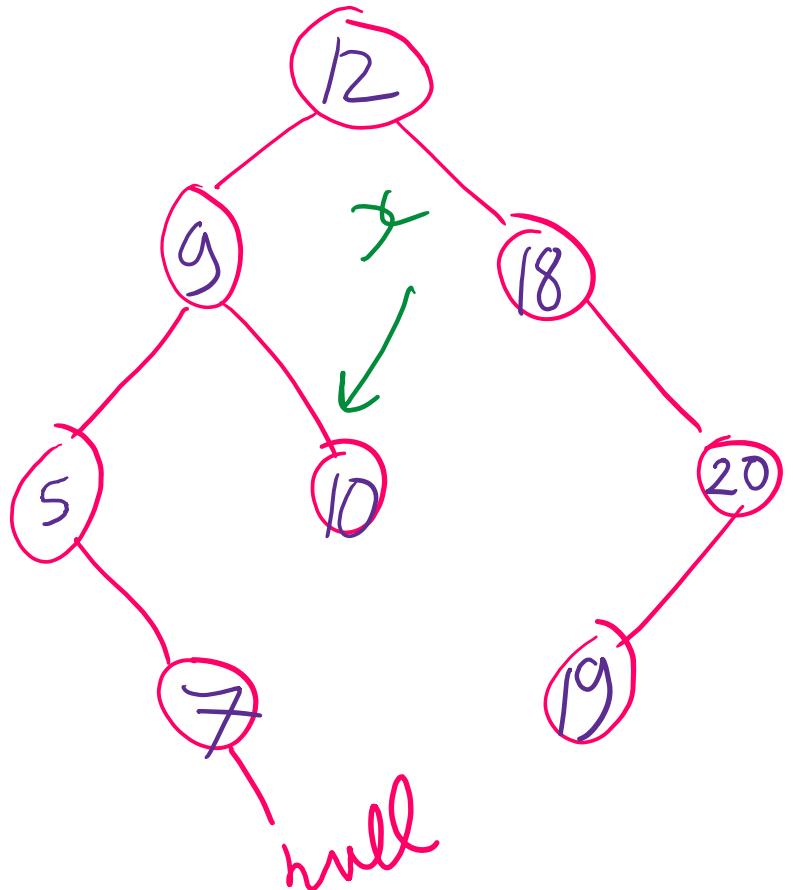
Iterative Inorder traversal



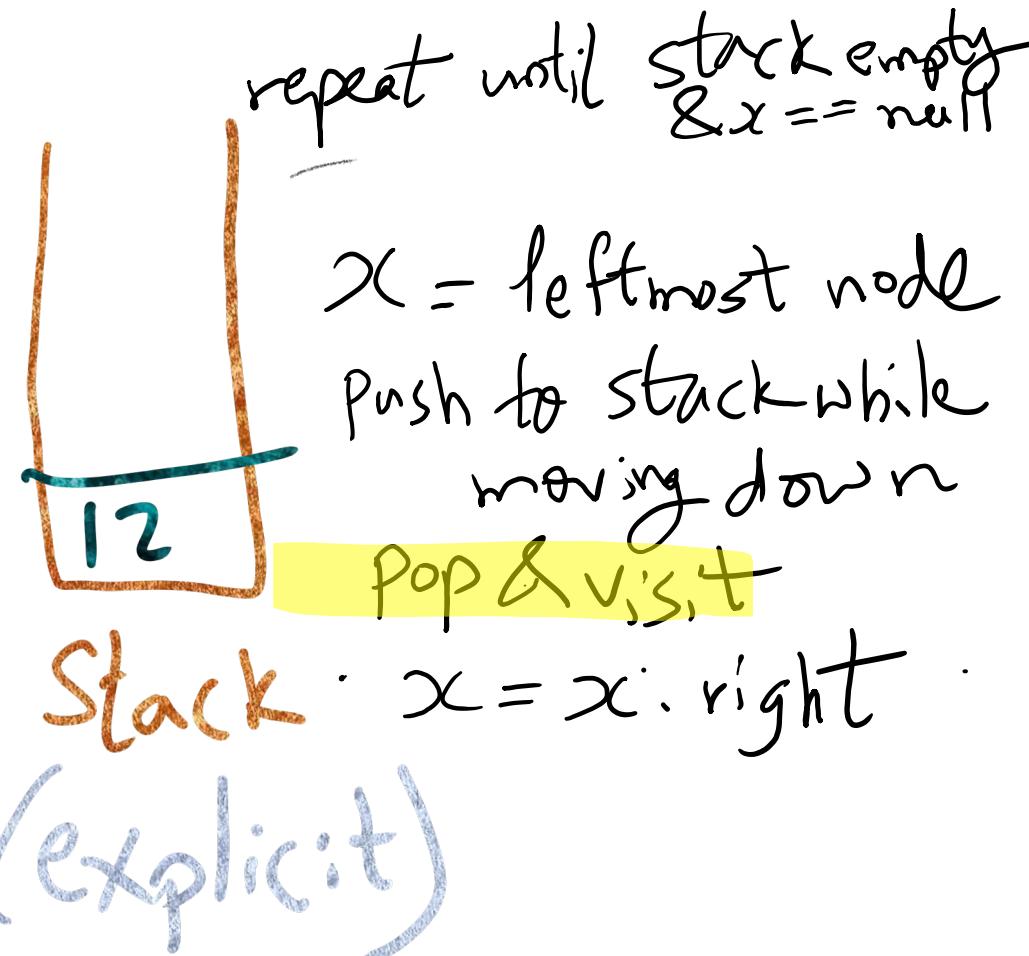
Visit order : 5, 7, 9



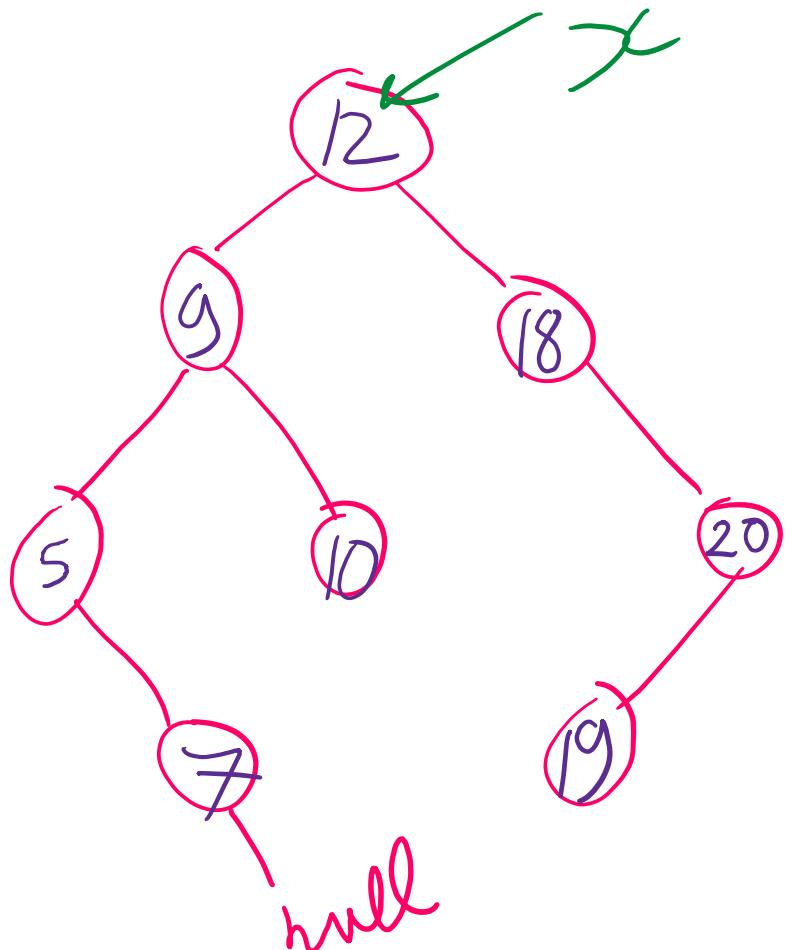
Iterative Inorder traversal



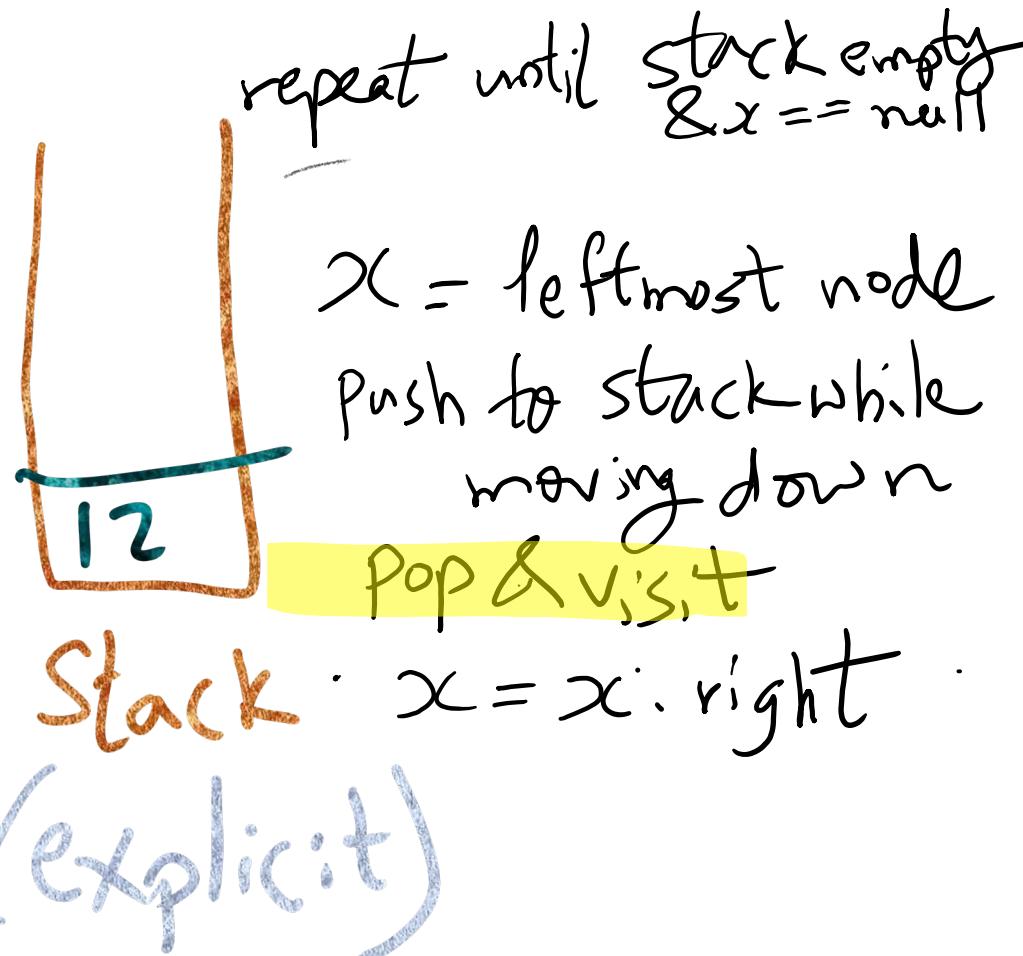
Visit order : 5, 7, 9, 10



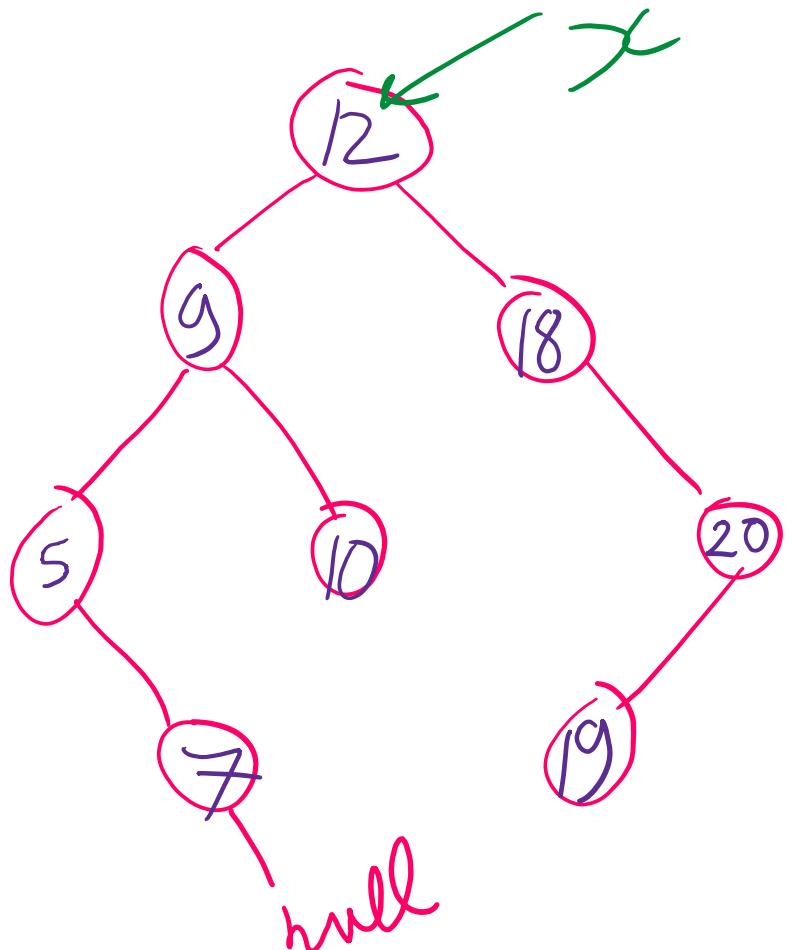
Iterative Inorder traversal



Visit order : 5, 7, 9, 10



Iterative Inorder traversal



Visit order : 5, 7, 9, 10, 12

repeat until stack empty & $x == \text{null}$

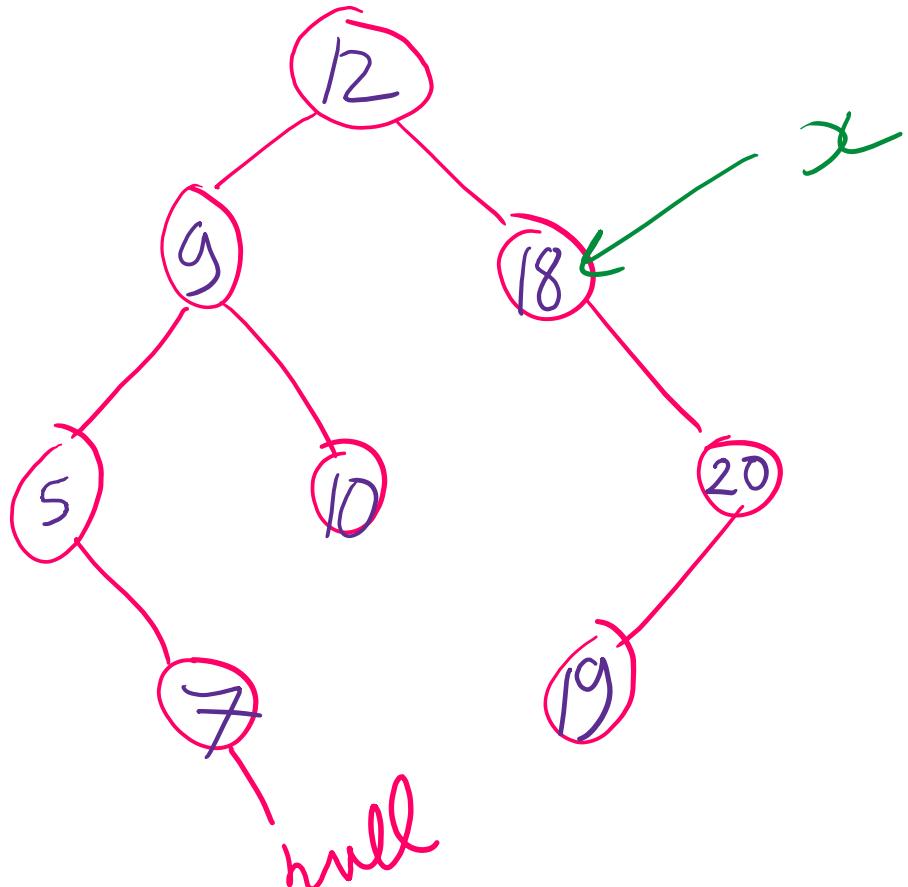
$x = \text{leftmost node}$
push to stack while moving down

pop & visit

Stack : $x = x.\text{right}$

(explicit)

Iterative Inorder traversal



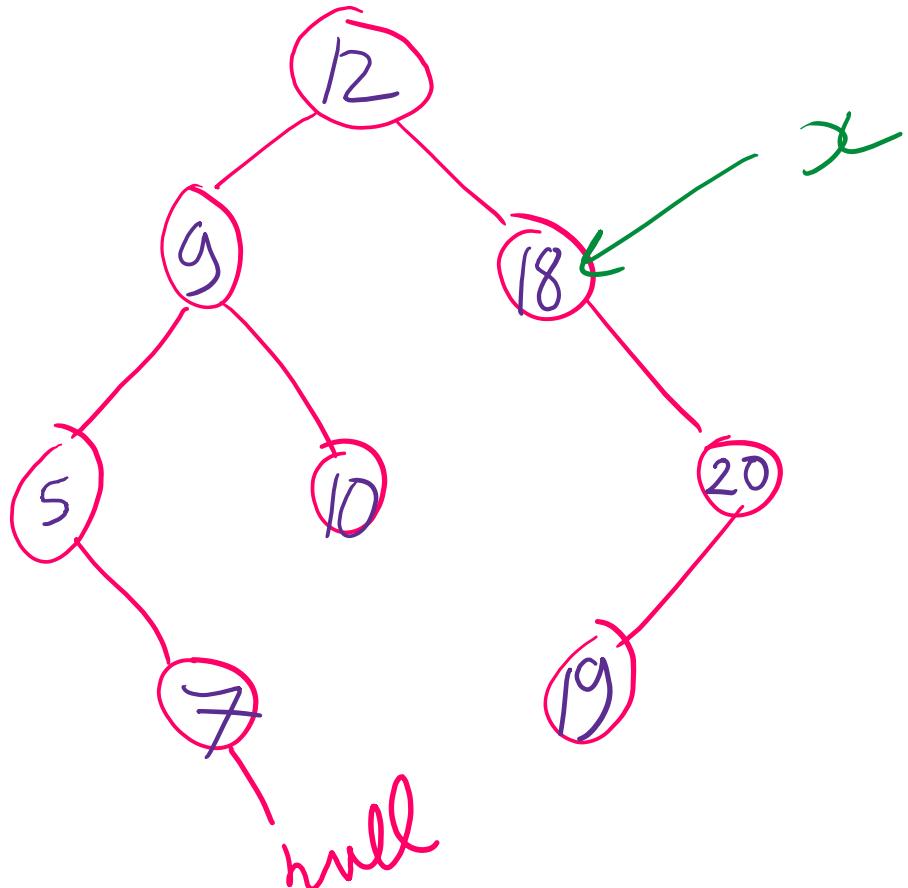
repeat until stack empty
& $x == \text{null}$

$x = \text{leftmost node}$
push to stack while moving down
pop & visit

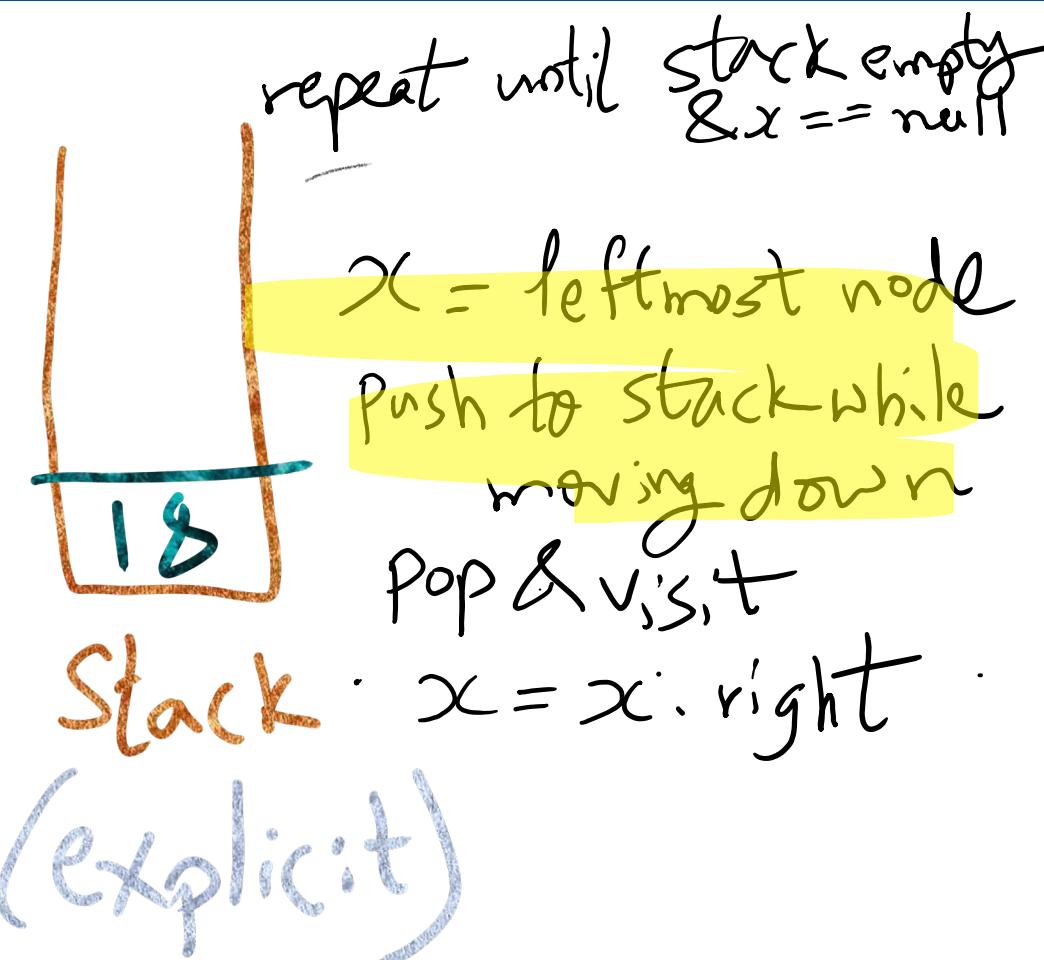
Stack : $x = x.\text{right}$
(explicit)

Visit order : 5, 7, 9, 10, 12

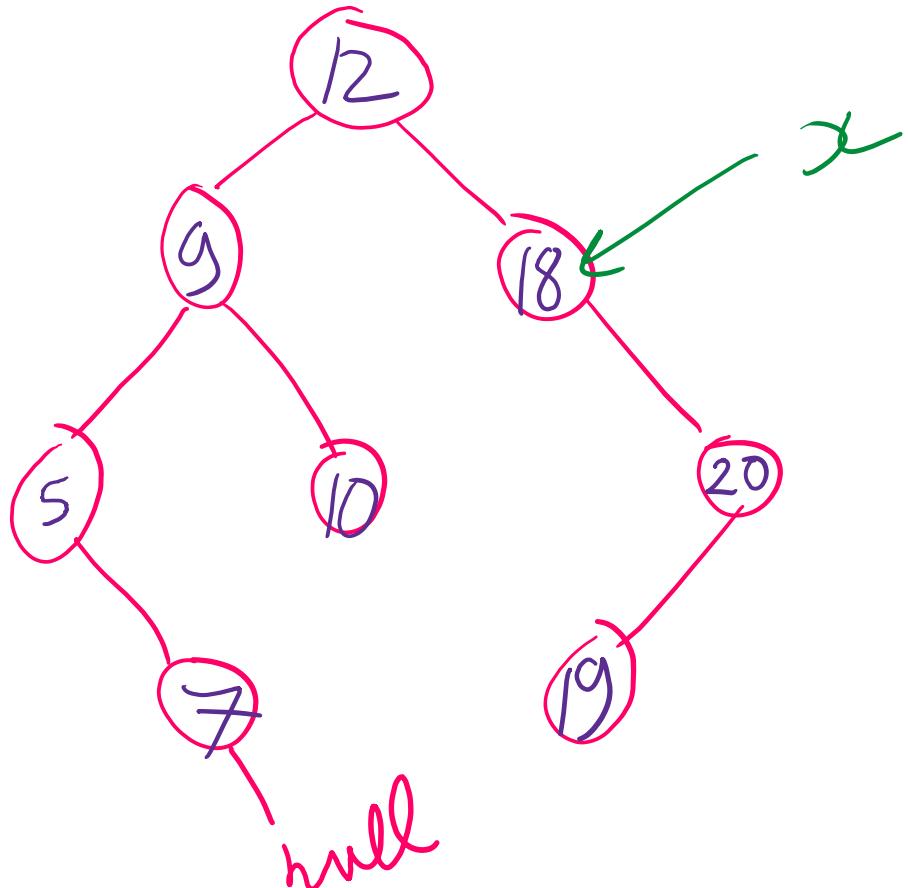
Iterative Inorder traversal



Visit order : 5, 7, 9, 10, 12



Iterative Inorder traversal



Visit order : 5, 7, 9, 10, 12, 18

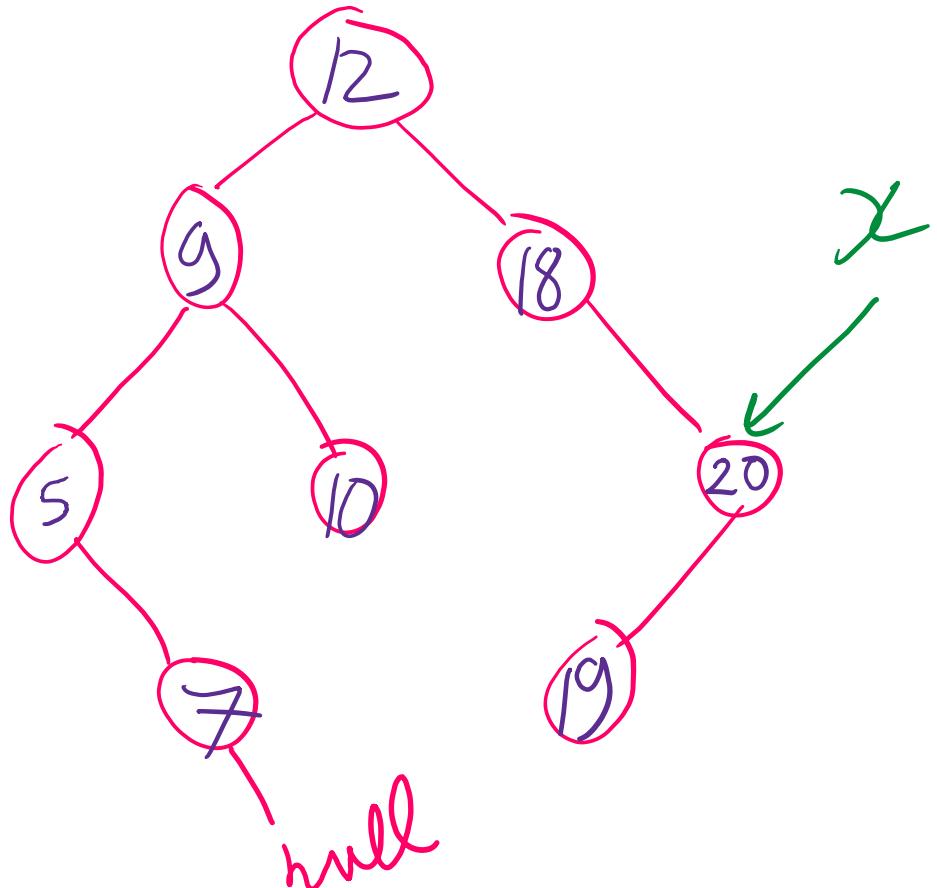
repeat until stack empty
 $\& x == \text{null}$

$x = \text{leftmost node}$
push to stack while moving down

Pop & visit

Stack · $x = x.\text{right}$ ·
(explicit)

Iterative Inorder traversal



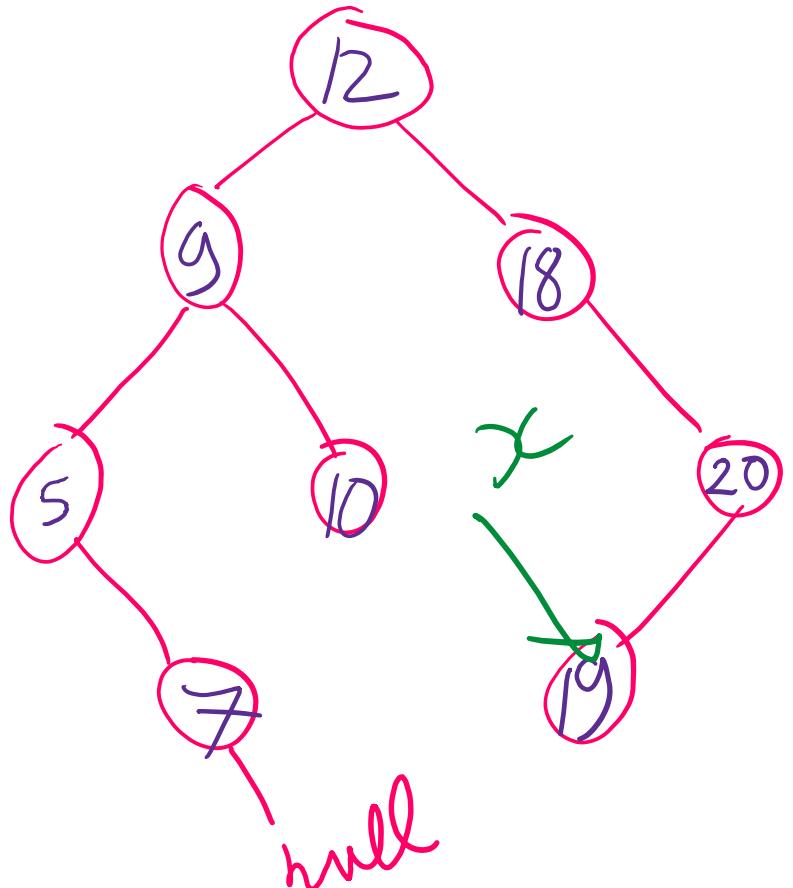
Visit order : 5, 7, 9, 10, 12, 18

repeat until stack empty
& $x == \text{null}$

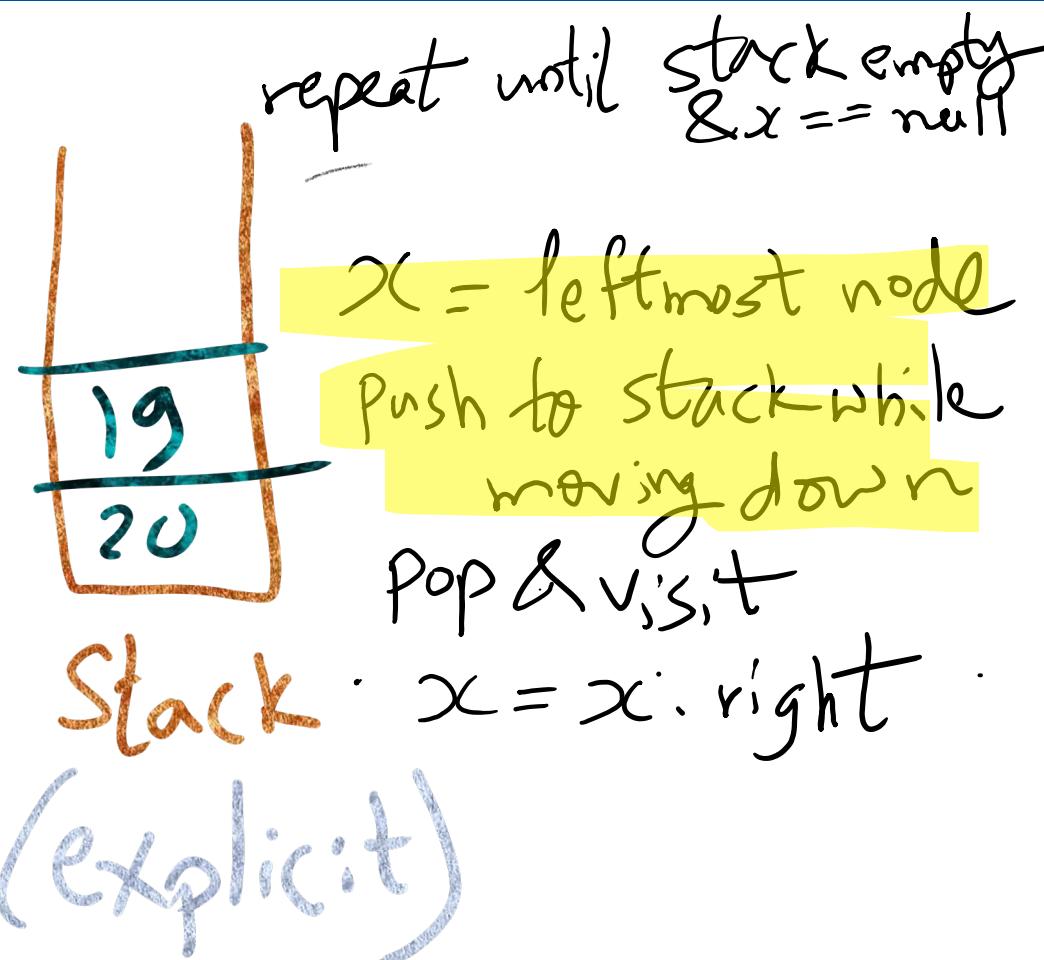
$x = \text{leftmost node}$
push to stack while moving down
pop & visit

Stack : $x = x.\text{right}$
(explicit)

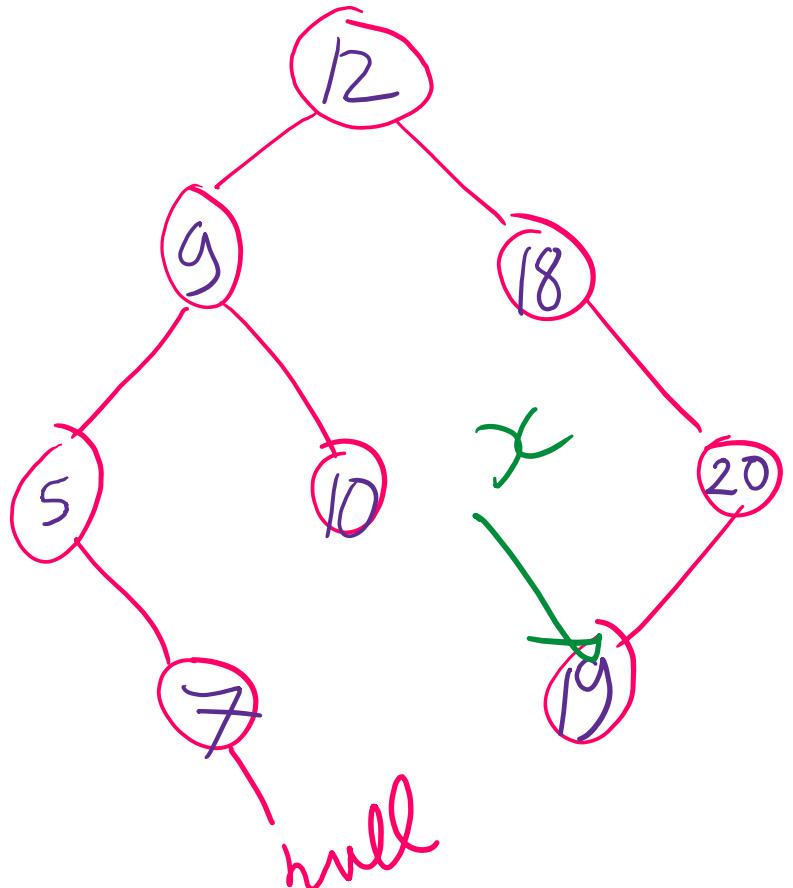
Iterative Inorder traversal



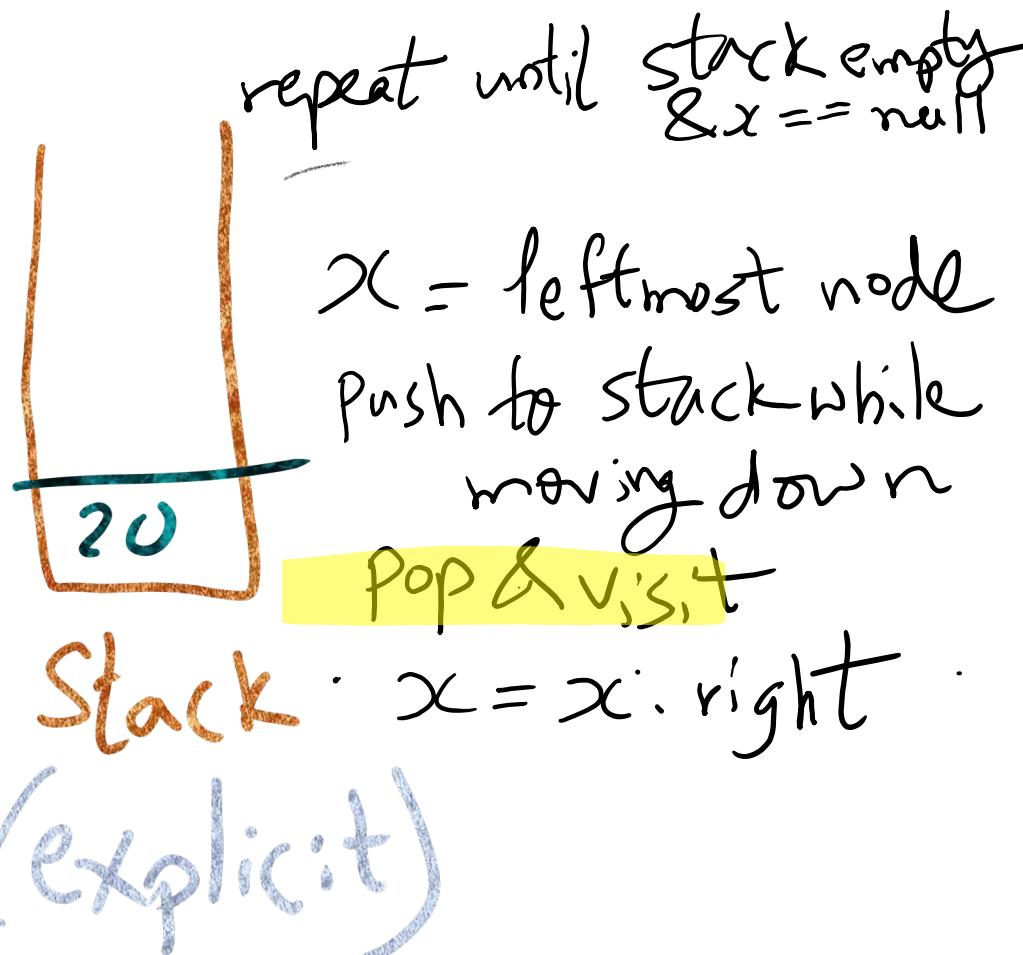
Visit order : 5, 7, 9, 10, 12, 18



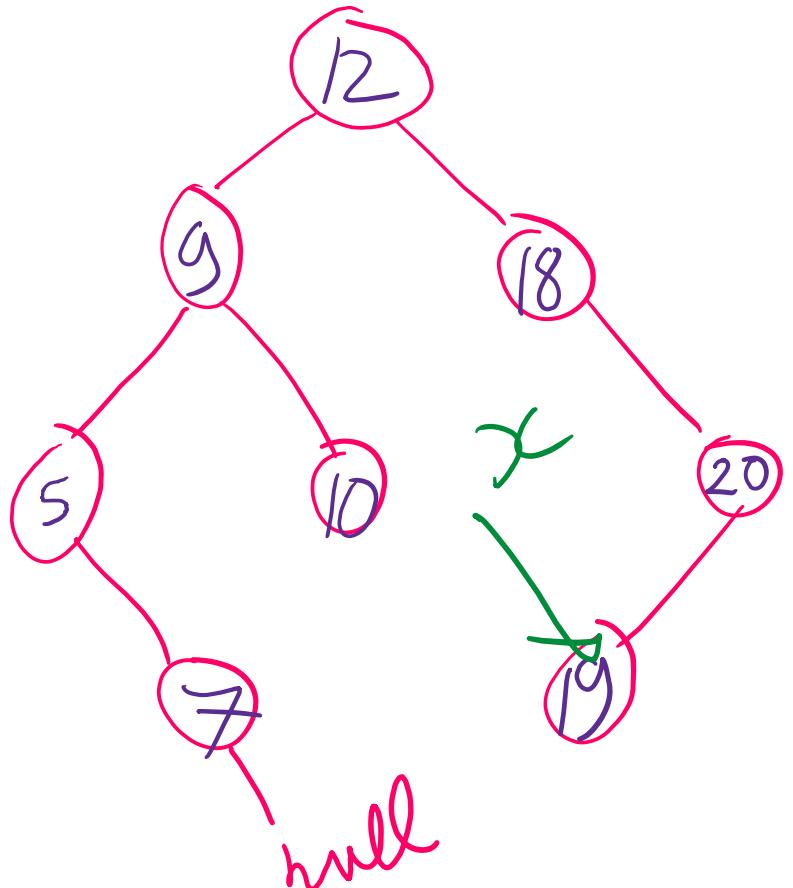
Iterative Inorder traversal



Visit order : 5, 7, 9, 10, 12, 18, 19



Iterative Inorder traversal



Visit order : 5, 7, 9, 10, 12, 18, 19

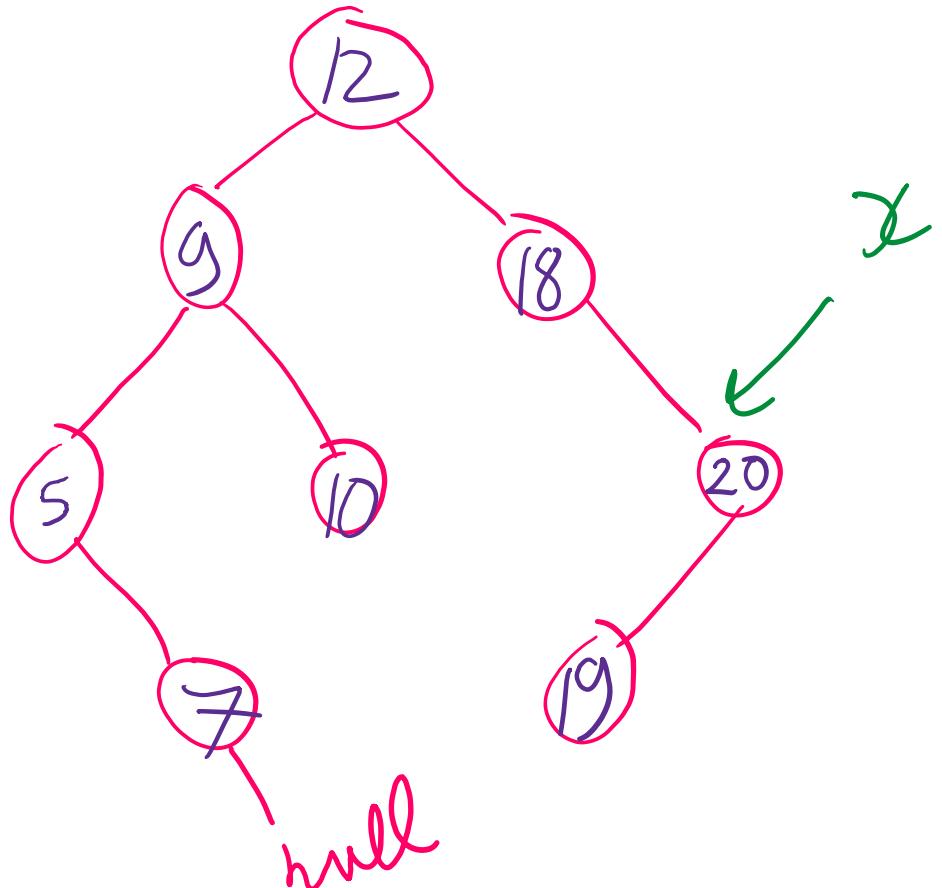
repeat until stack empty
 $\& x == \text{null}$



$x = \text{leftmost node}$
push to stack while moving down
pop & visit
 $x = x.\text{right}$

Stack (explicit)

Iterative Inorder traversal



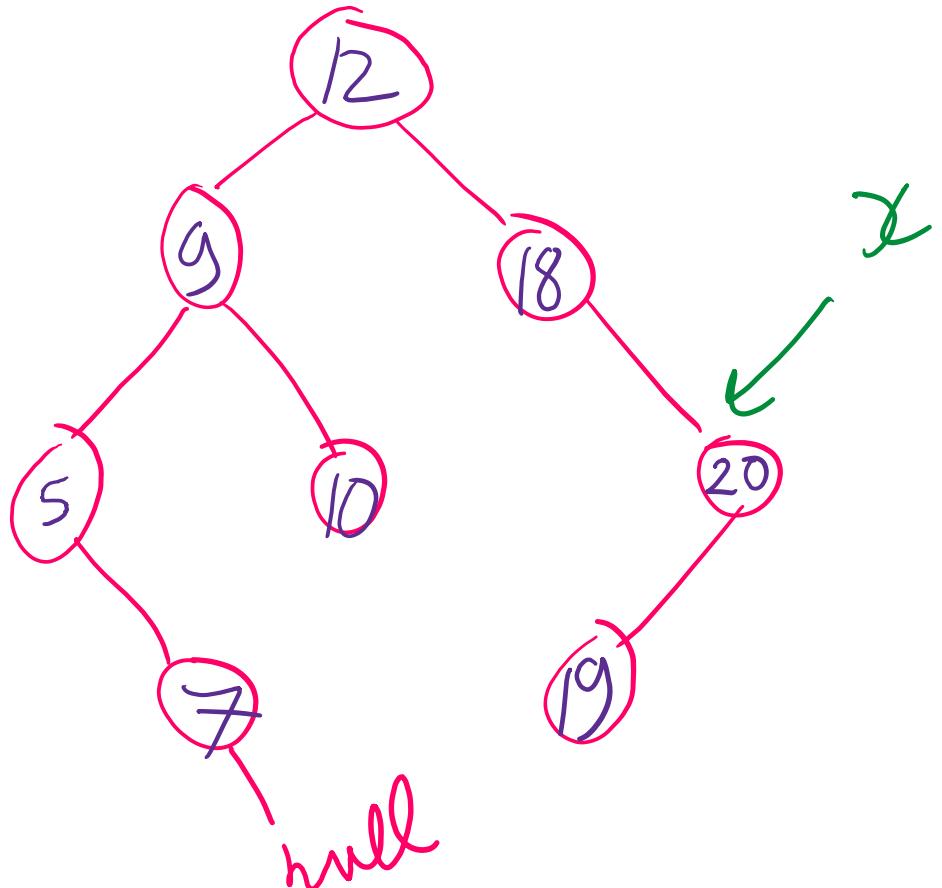
Visit order : 5, 7, 9, 10, 12, 18, 19, 20

repeat until stack empty
& $x == \text{null}$

$x = \text{leftmost node}$
push to stack while moving down
pop & visit

Stack (explicit) : $x = x.\text{right}$

Iterative Inorder traversal



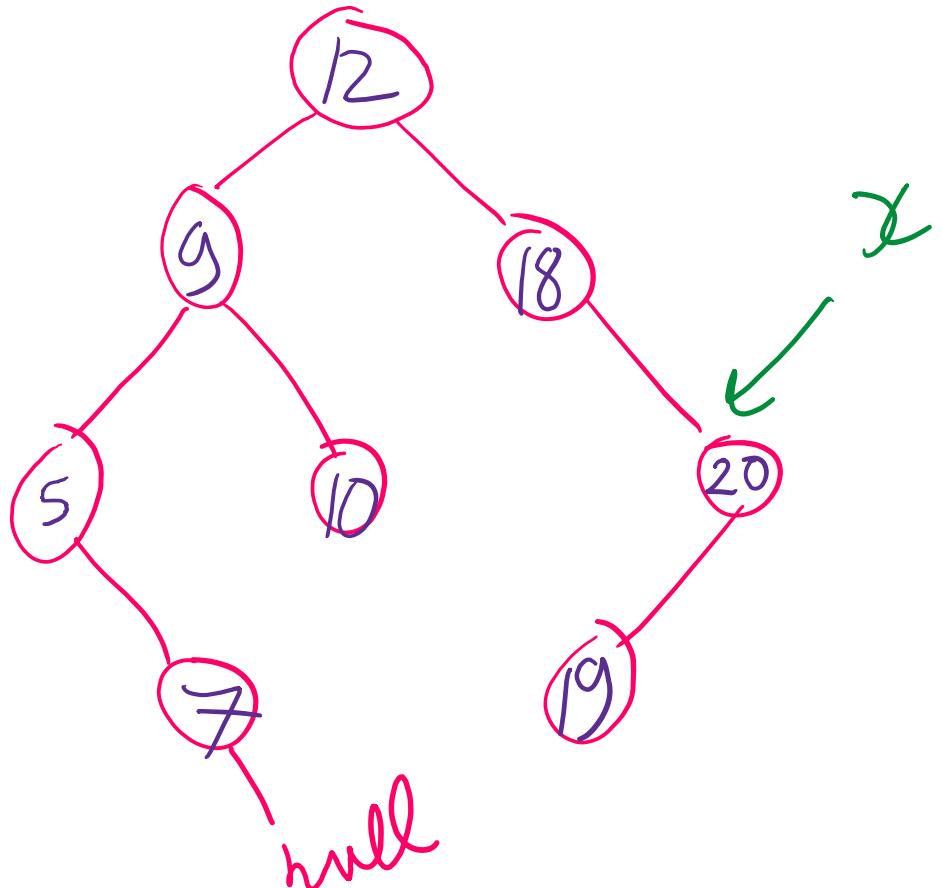
Visit order : 5, 7, 9, 10, 12, 18, 19, 20

repeat until stack empty
 $\& x == \text{null}$

$x = \text{leftmost node}$
push to stack while moving down
pop & visit

Stack : $x = x.\text{right}$
(explicit)

Iterative Inorder traversal



Visit order : 5, 7, 9, 10, 12, 18, 19, 20

repeat until stack empty & $x == \text{null}$

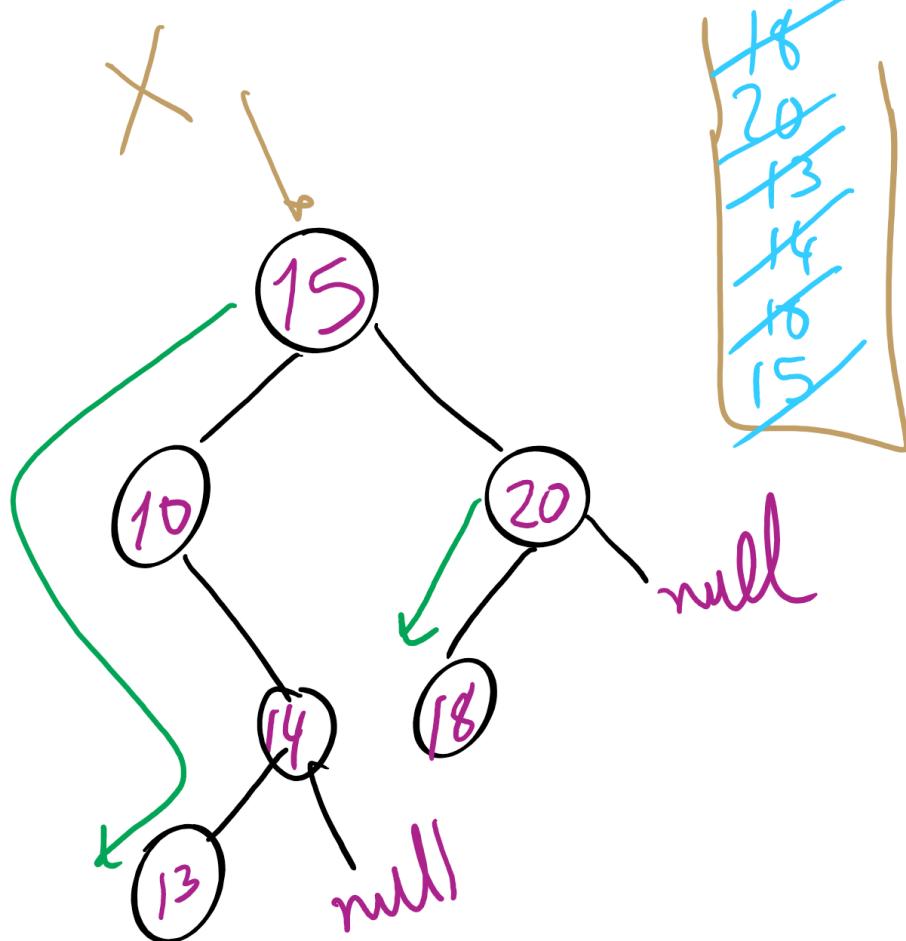
$x = \text{leftmost node}$
push to stack while moving down
pop & visit
 $x = x.\text{right}$

Stack (explicit)

Traversals of a Binary Tree

- Preorder traversal
 - Visit root before we visit root's subtrees
- Inorder traversal
 - Visit root of a binary tree between visiting nodes in root's subtrees.
- Postorder traversal
 - Visit root of a binary tree after visiting nodes in root's subtrees
 - left then right then root

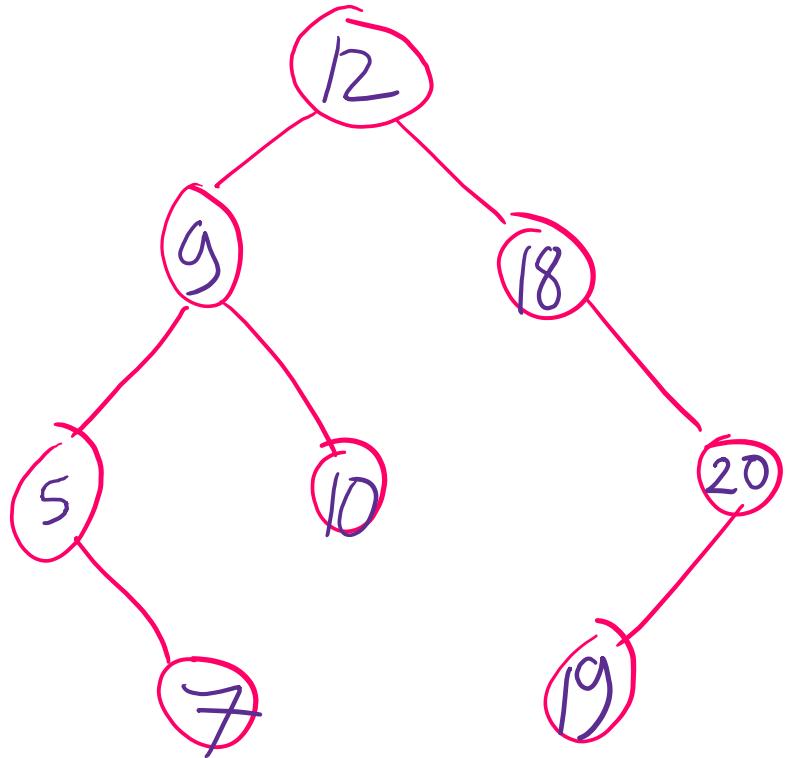
Iterative Post-order Traversal



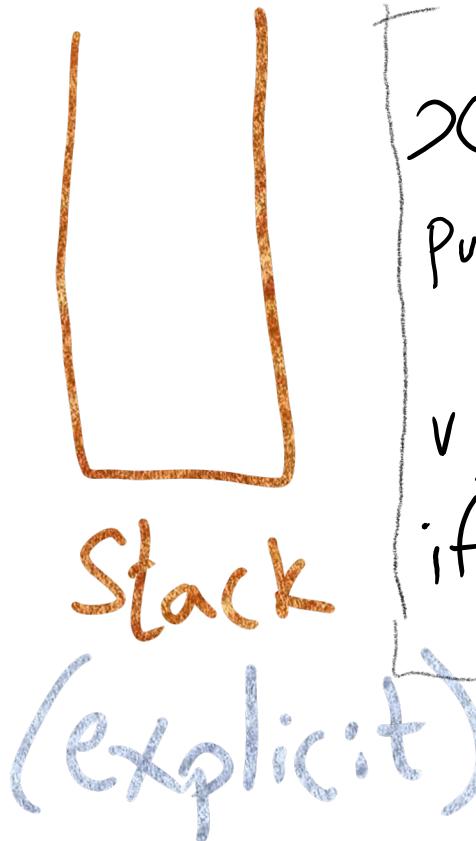
13, 14, 10, 18, 20 15

stack = empty stack
x = root
while(x != null || stack is not empty){
 - find leftmost leaf
 - push nodes as we go down the tree
 - x = pop() & visit x
 - if x is a left child
 x = right sibling
 else
 x = null
y

Iterative Postorder traversal



Visit order :

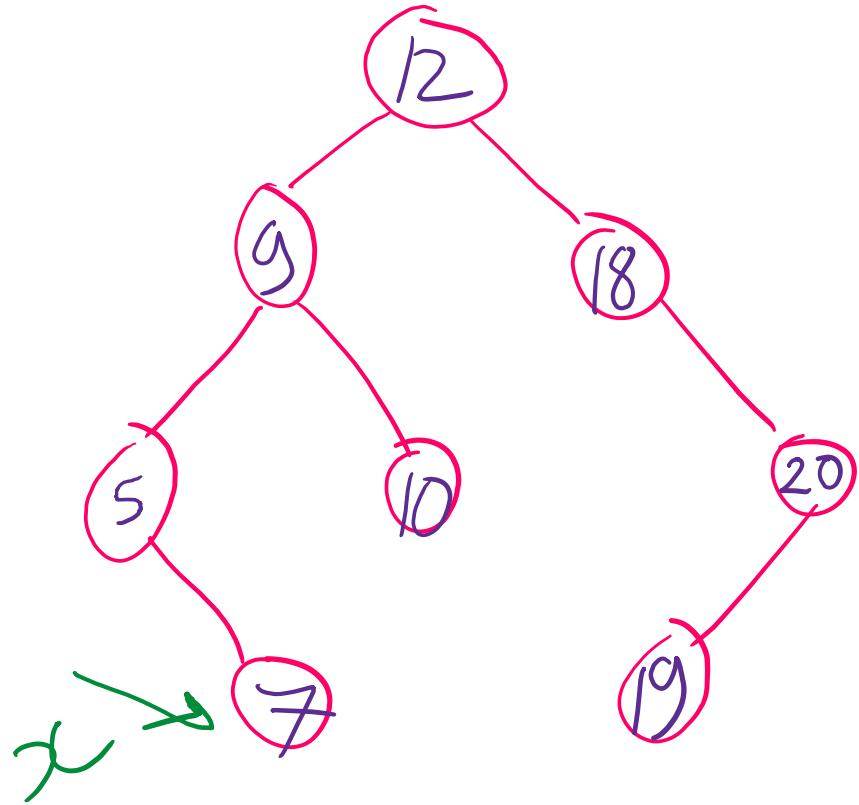


repeat until stack empty
& $x = \text{null}$

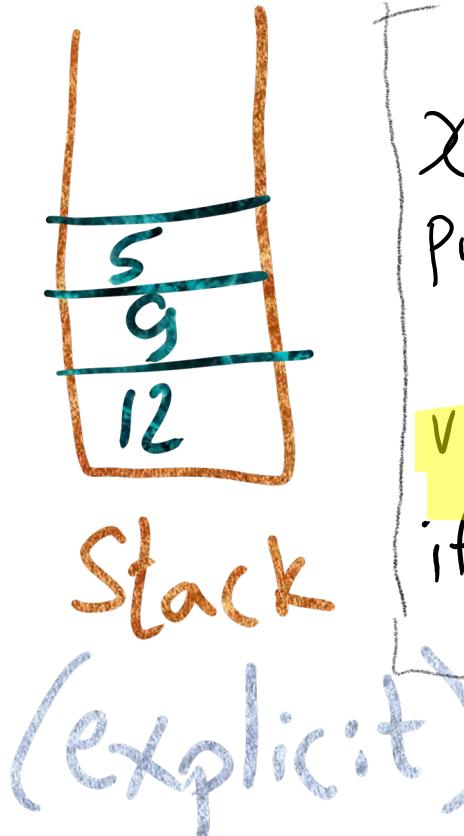
$x = \text{leftmost leaf}$
push to stack while moving down
visit x
if x is right child
pop & visit
skip leftmost leaf

else $x = \text{parent, right}$

Iterative Postorder traversal



Visit order : 7



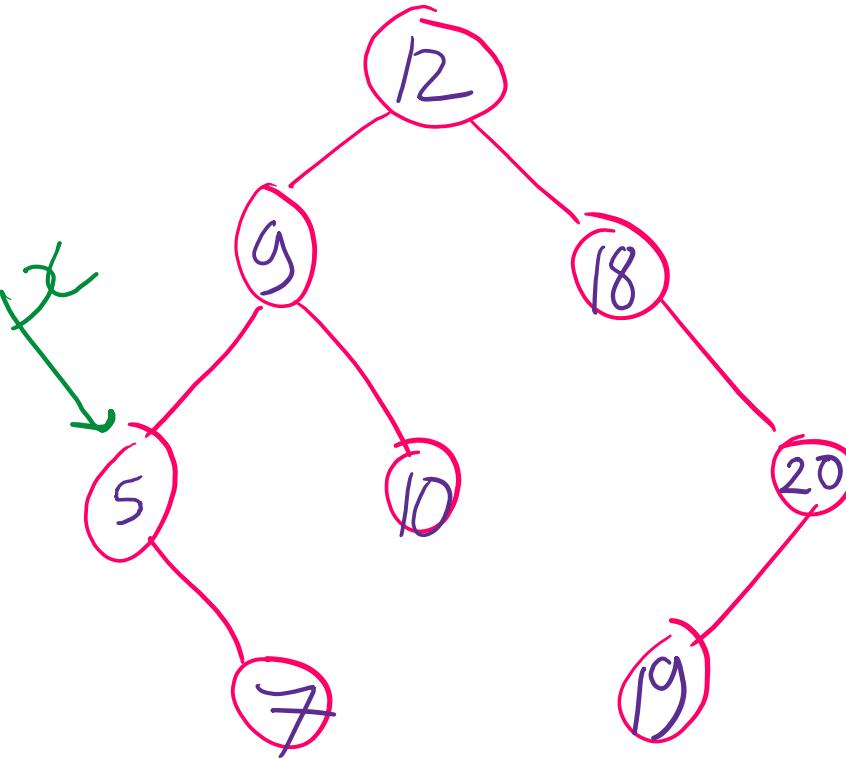
repeat until stack empty & $x = \text{null}$

$x = \text{leftmost leaf}$
push to stack while moving down

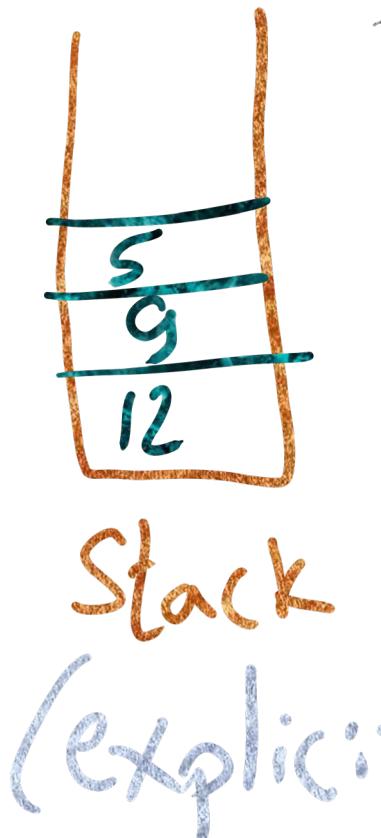
visit x
if x is right child
pop & visit
skip leftmost leaf

else $x = \text{parent. right}$

Iterative Postorder traversal



Visit order : 7, 5

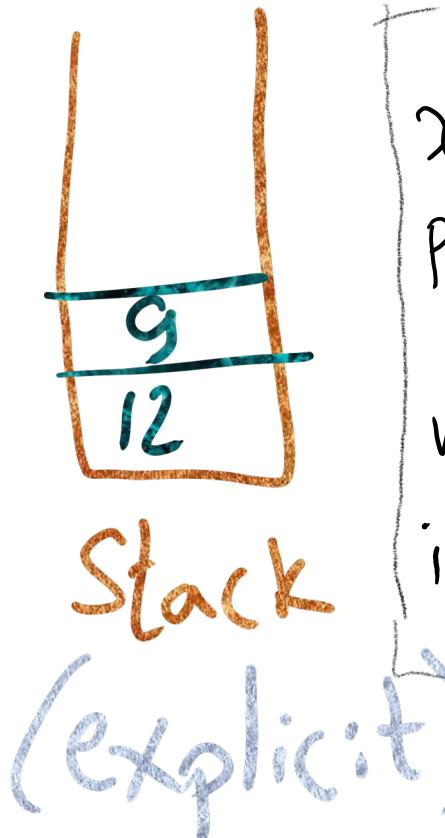
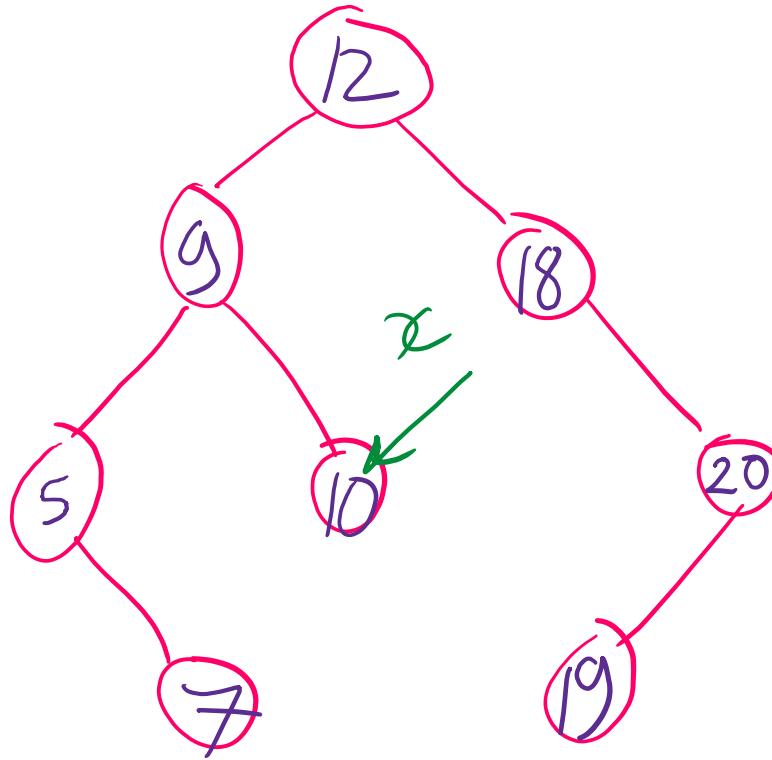


repeat until stack empty & $x = \text{null}$

$x = \text{leftmost leaf}$
push to stack while moving down
visit x
if x is right child
pop & visit
skip leftmost leaf

else $x = \text{parent. right}$

Iterative Postorder traversal



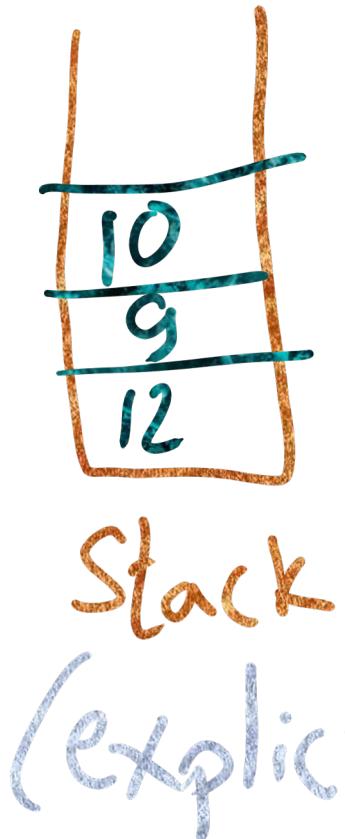
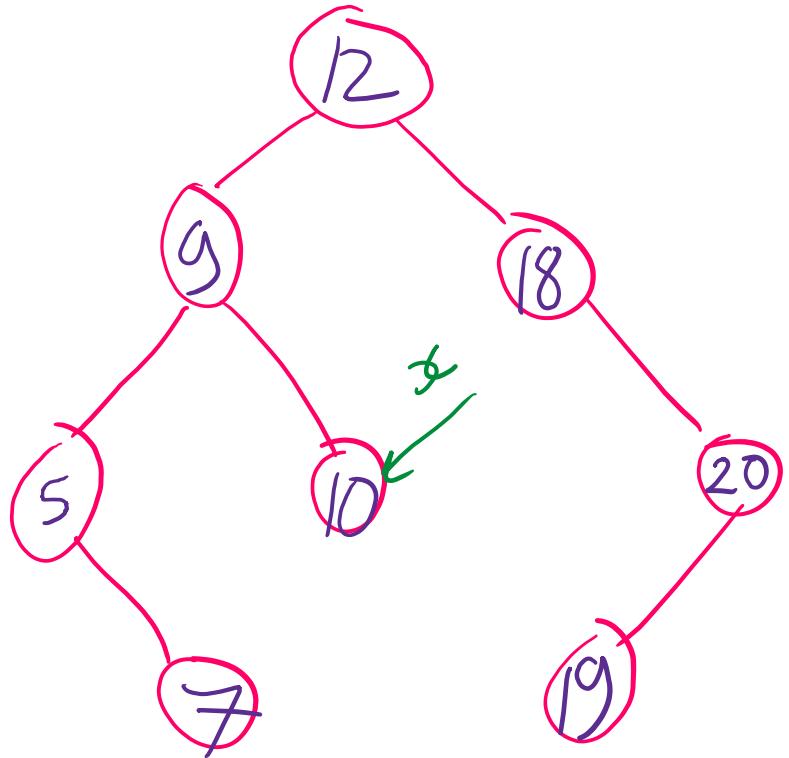
repeat until stack empty & $x = \text{null}$

$x = \text{leftmost leaf}$
push to stack while moving down
visit x
if x is right child
pop & visit
skip leftmost leaf

Visit order : 7, 5

else $x = \text{parent. right}$

Iterative Postorder traversal



repeat until stack empty & $x = \text{null}$

$x = \text{leftmost leaf}$

Push to stack while moving down

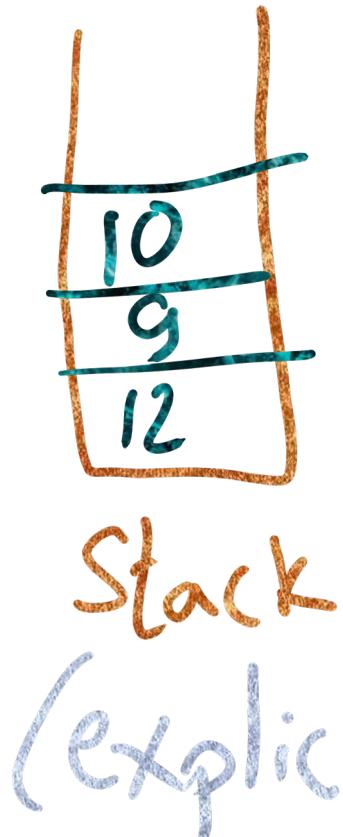
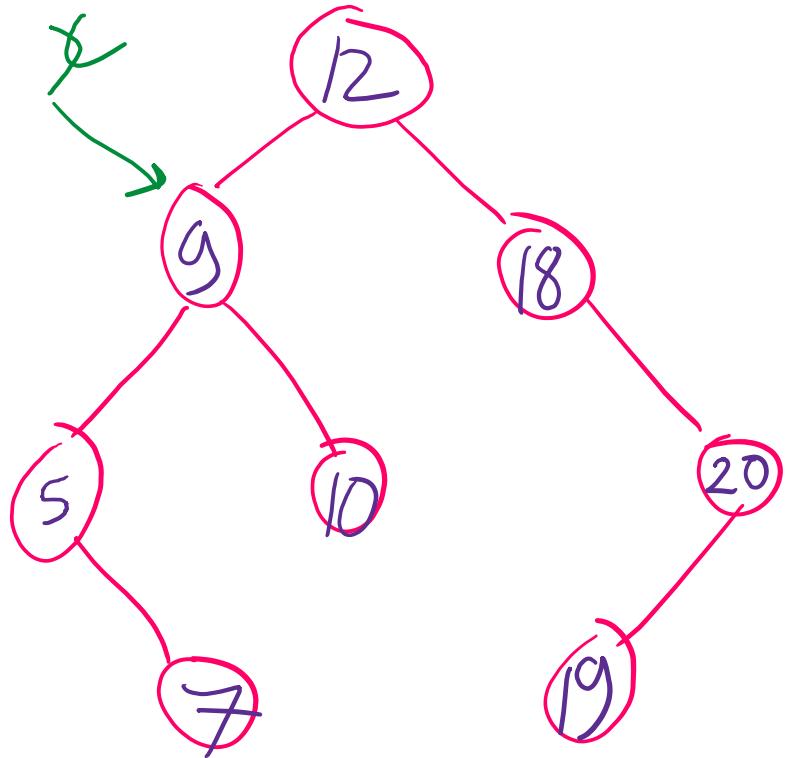
visit x

if x is right child
pop & visit
skip leftmost leaf

Visit order: 7, 5, 10

else $x = \text{parent. right}$

Iterative Postorder traversal



repeat until stack empty & $x = \text{null}$

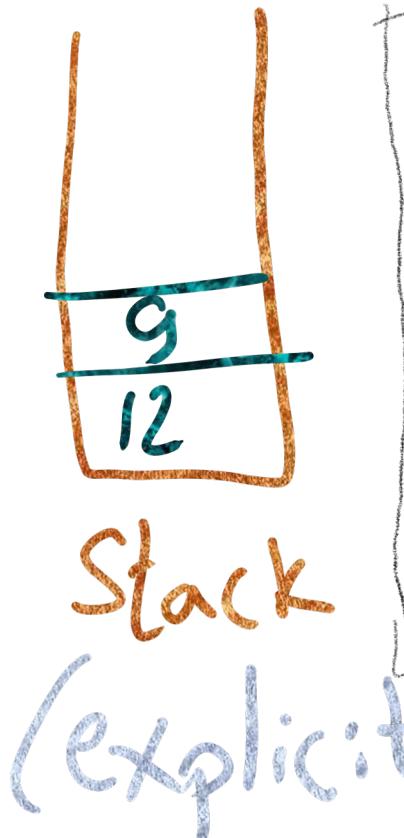
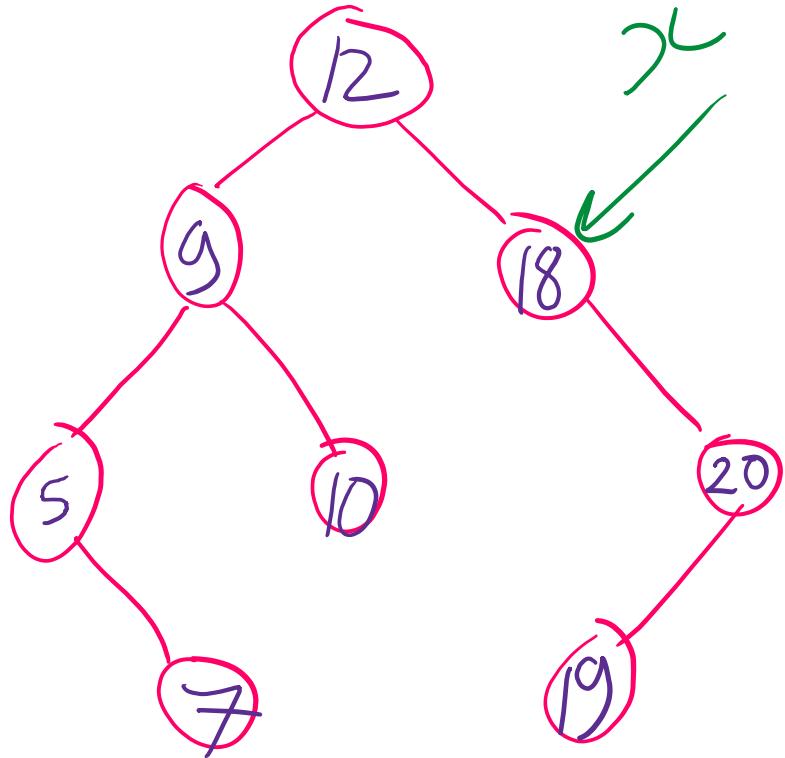
$x = \text{leftmost leaf}$
push to stack while moving down
visit x

if x is rightchild
pop & visit
skip leftmost leaf

Visit order: 7, 5, 10, 9

else $x = \text{parent. right}$

Iterative Postorder traversal



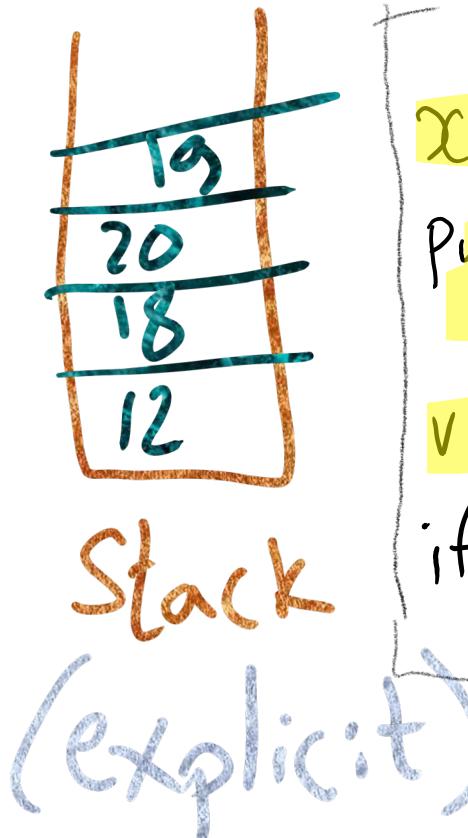
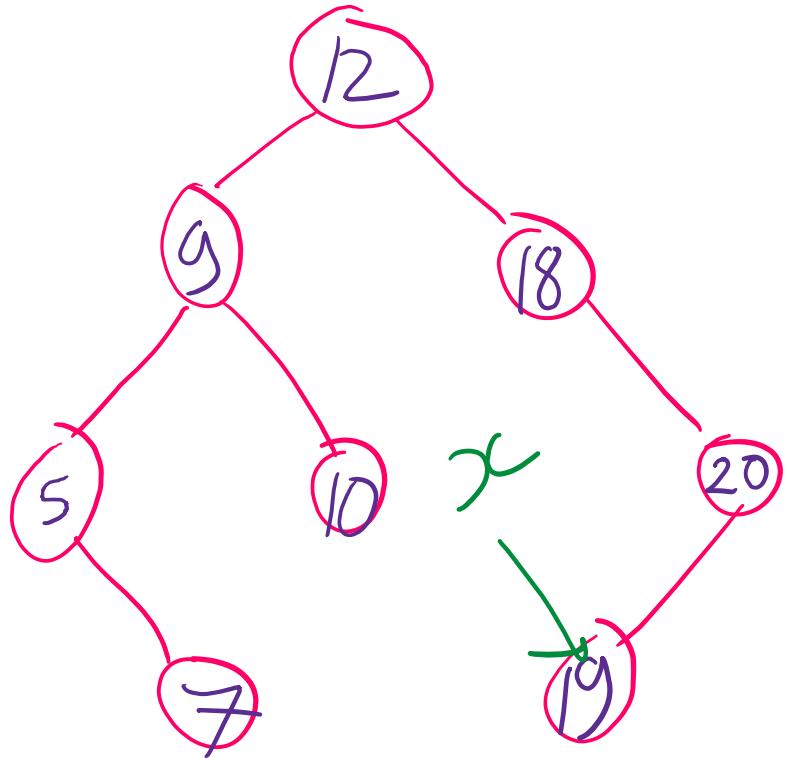
repeat until stack empty & $x = \text{null}$

$x = \text{leftmost leaf}$
push to stack while moving down
visit x
if x is right child
pop & visit
skip leftmost leaf

Visit order: 7, 5, 10, 9

else $x = \text{parent. right}$

Iterative Postorder traversal



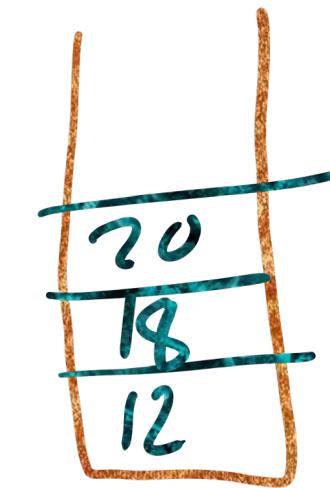
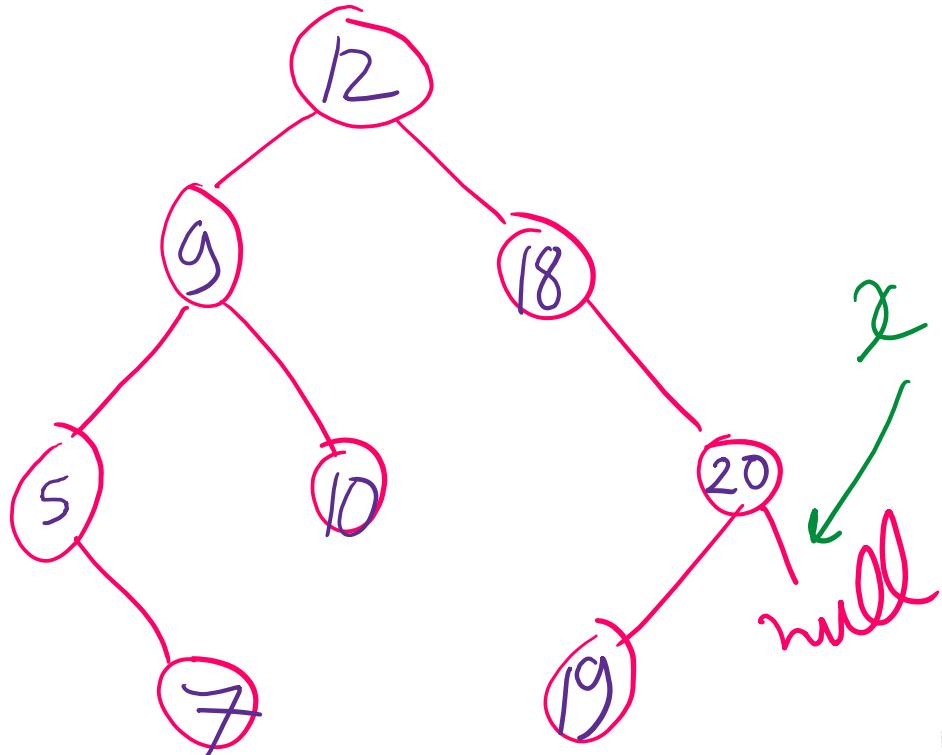
repeat until stack empty
& $x == \text{null}$

$x = \text{leftmost leaf}$
push to stack while moving down
visit x
if x is right child
pop & visit
skip leftmost leaf

else $x = \text{parent. right}$

Visit order: 7, 5, 10, 9, 19

Iterative Postorder traversal



Stack
(explicit)

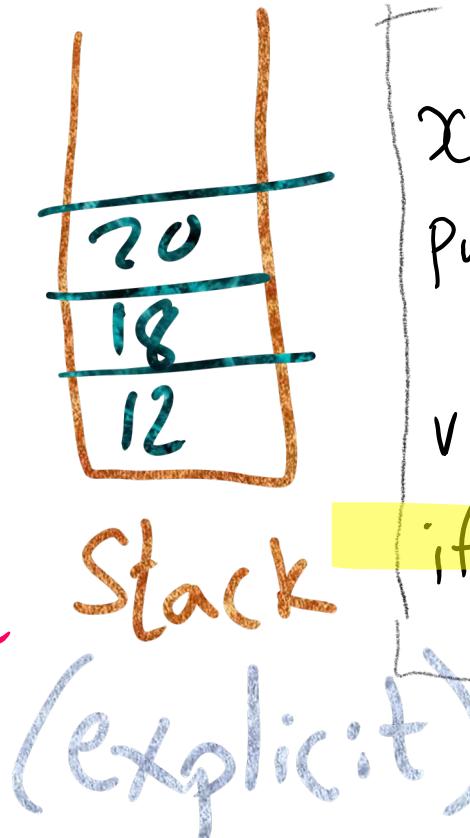
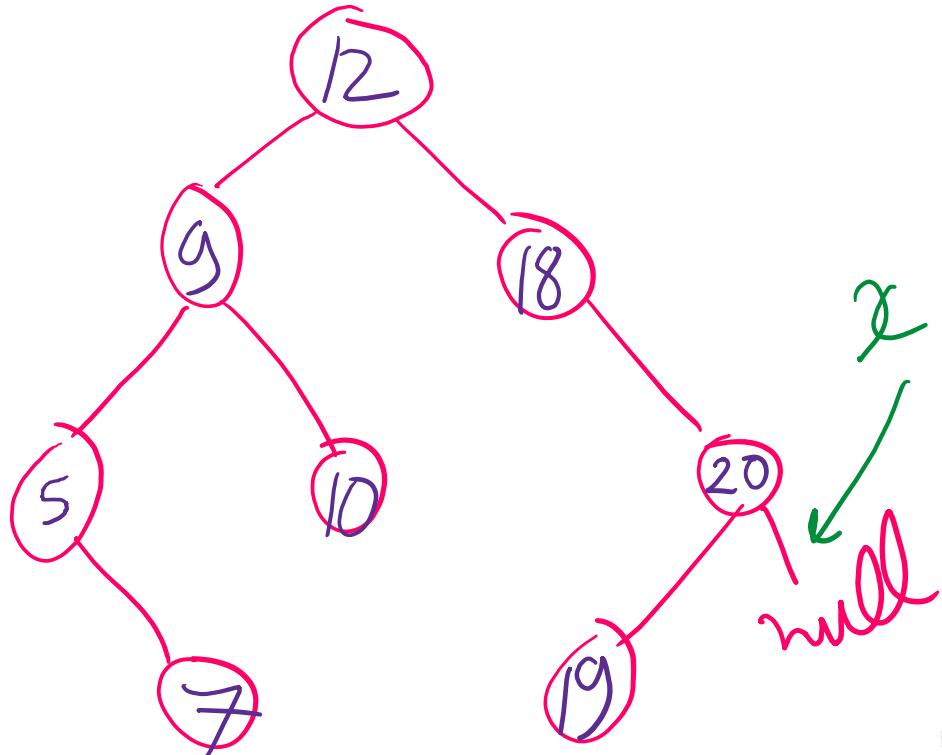
repeat until stack empty
& $x = \text{null}$

$x = \text{leftmost leaf}$
push to stack while moving down
visit x
if x is right child
pop & visit
skip leftmost leaf

Visit order : 7, 5, 10, 9, 19

else $x = \text{parent. right}$

Iterative Postorder traversal



repeat until stack empty & $x = \text{null}$

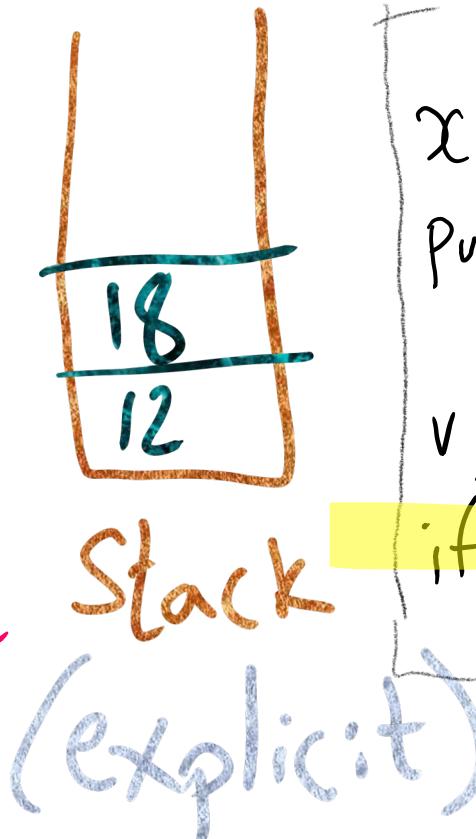
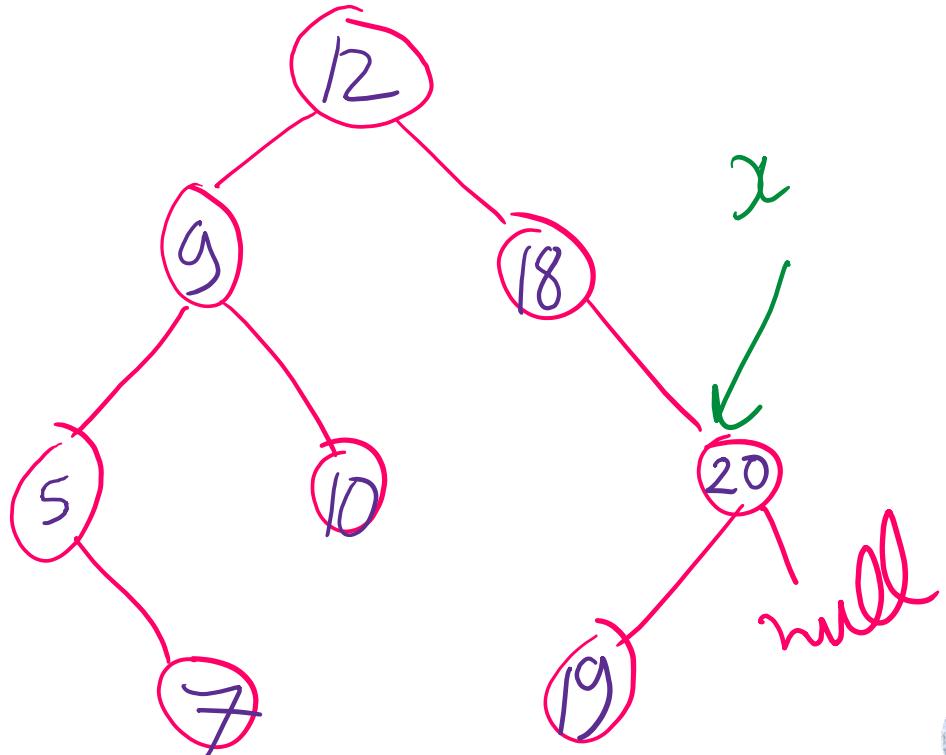
$x = \text{leftmost leaf}$
push to stack while moving down
visit x

if x is right child
pop & visit
skip leftmost leaf

Visit order: 7, 5, 10, 9, 19

else $x = \text{parent. right}$

Iterative Postorder traversal



repeat until stack empty & $x = \text{null}$

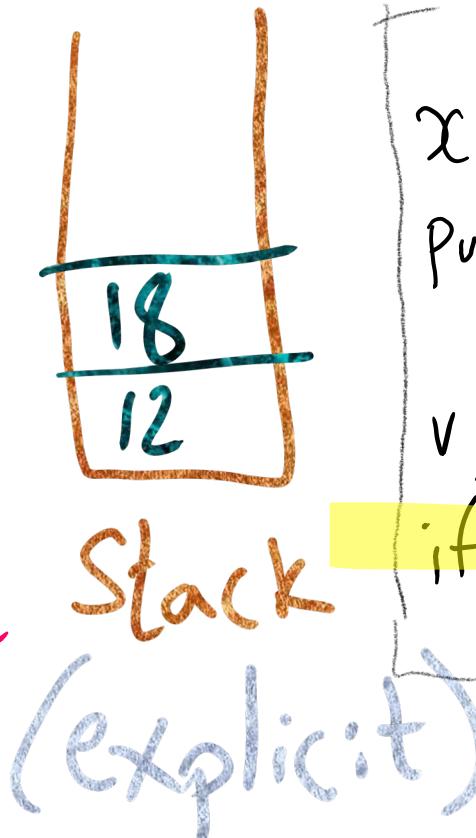
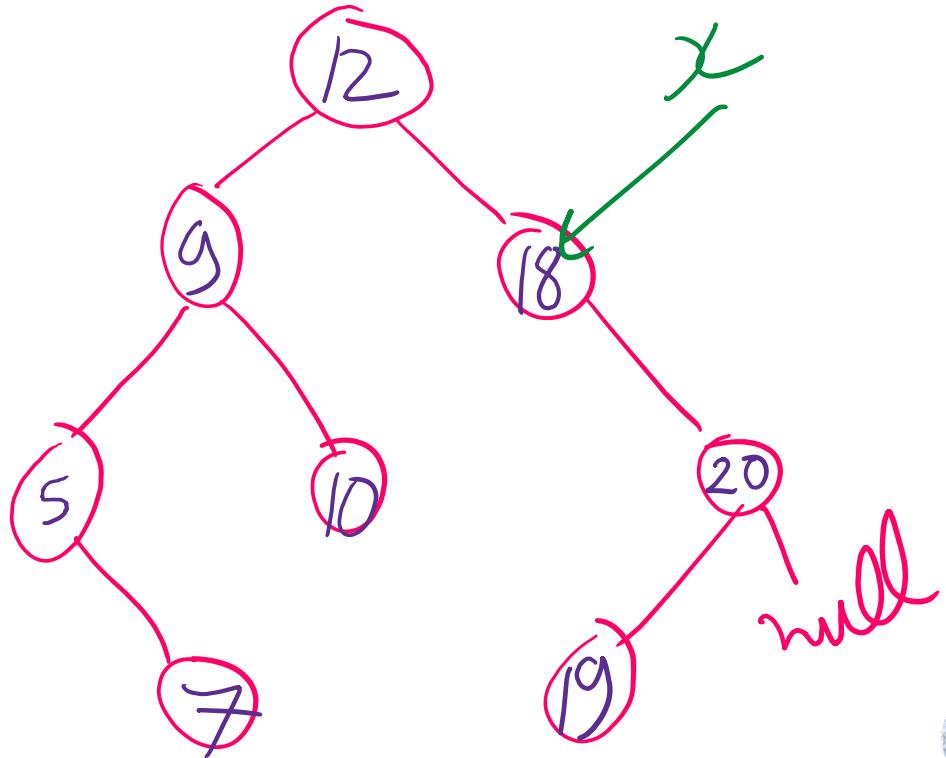
$x = \text{leftmost leaf}$
push to stack while moving down
visit x

if x is right child
pop & visit
skip leftmost leaf

Visit order: 7, 5, 10, 9, 19, 20

else $x = \text{parent. right}$

Iterative Postorder traversal



repeat until stack empty & $x = \text{null}$

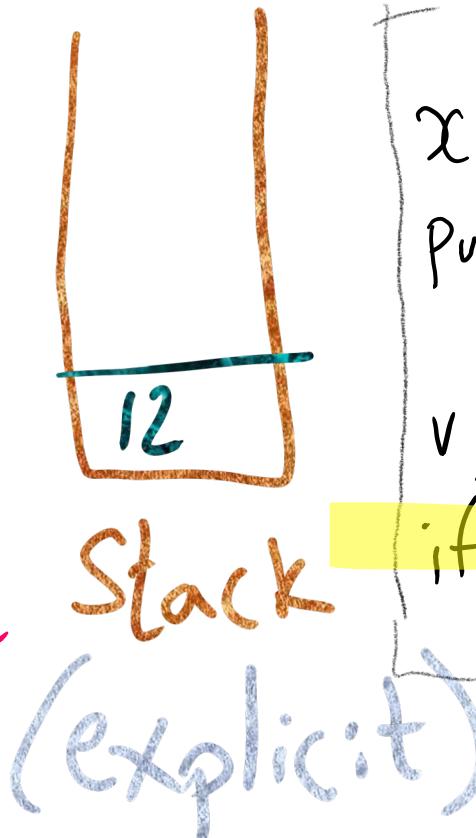
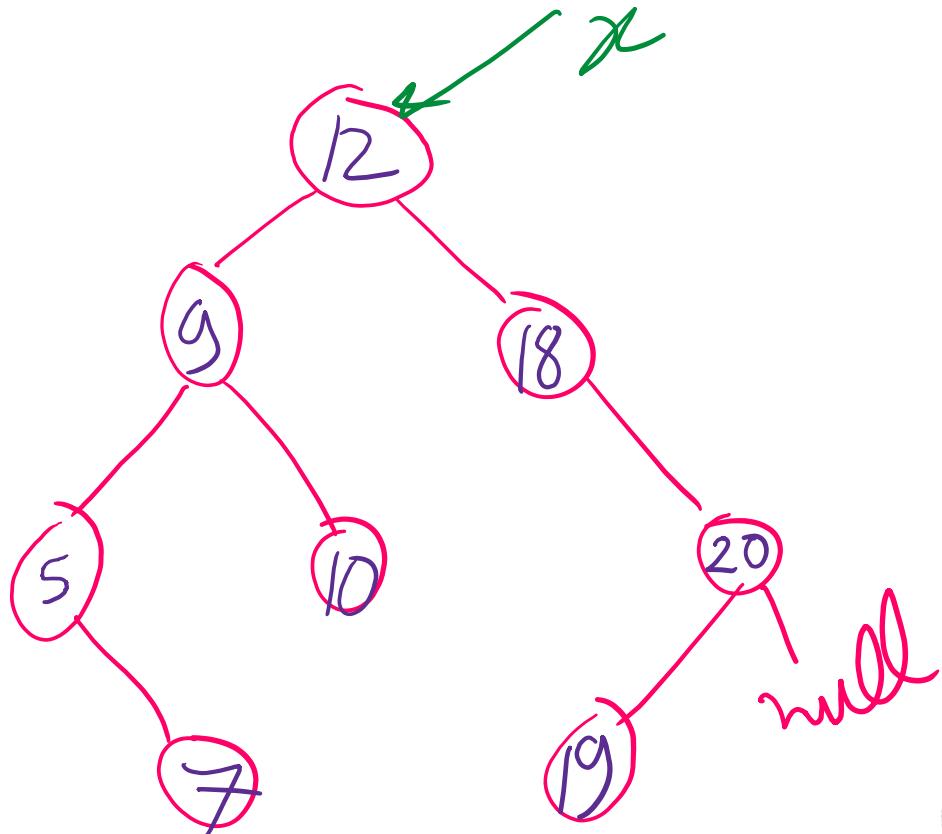
$x = \text{leftmost leaf}$
push to stack while moving down
visit x

if x is right child
pop & visit
skip leftmost leaf

Visit order: 7, 5, 10, 9, 19, 20, 18

else $x = \text{parent. right}$

Iterative Postorder traversal



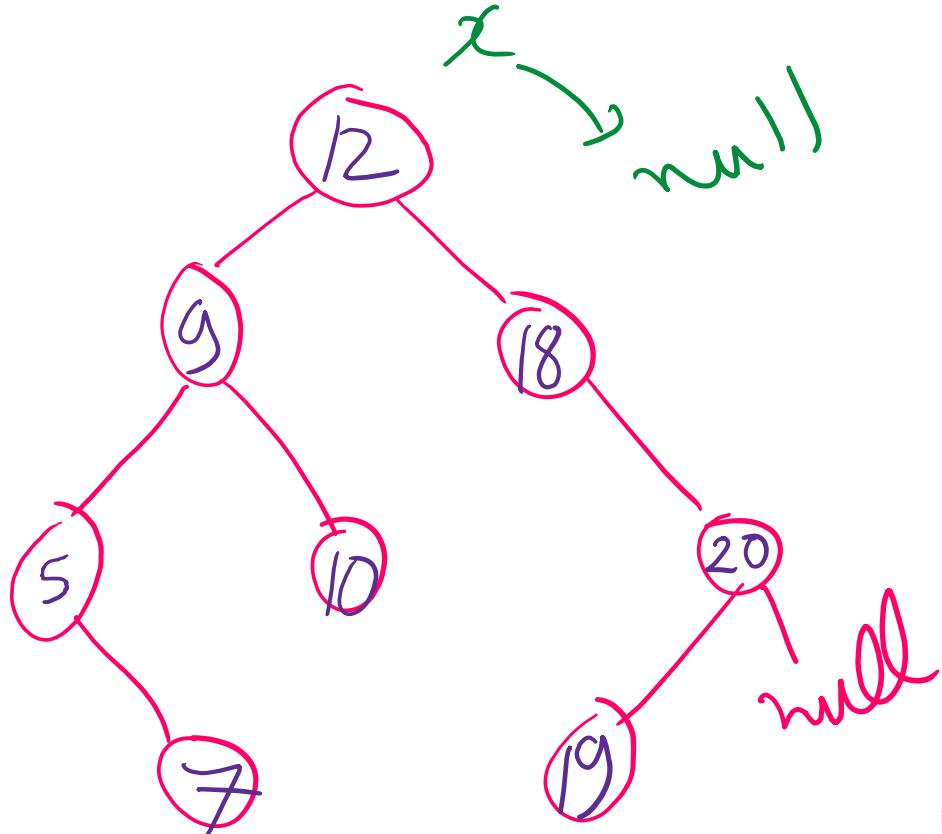
repeat until stack empty & $x = \text{null}$

$x = \text{leftmost leaf}$
push to stack while moving down
visit x
if x is right child
pop & visit
skip leftmost leaf

Visit order: 7, 5, 10, 9, 19, 20, 18, 12 right

else $x = \text{parent.}$

Iterative Postorder traversal



Stack
(explicit)

repeat until stack empty
& $x = \text{null}$

$x = \text{leftmost leaf}$
push to stack while
moving down
visit x

if x is right child
pop & visit
skip leftmost
leaf

Visit order : 7, 5, 10, 9, 19, 20, 18, 12
else $x = \text{parent. right}$

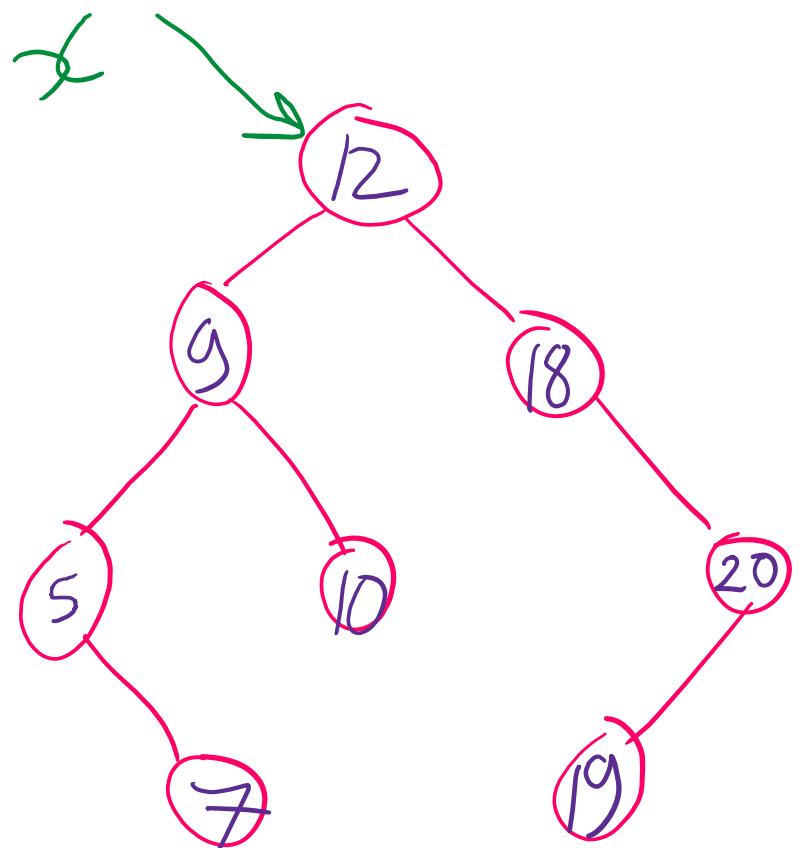
Traversals of a Binary Tree

- Preorder traversal
 - Visit root before we visit root's subtrees
- Inorder traversal
 - Visit root of a binary tree between visiting nodes in root's subtrees.
- Postorder traversal
 - Visit root of a binary tree after visiting nodes in root's subtrees
- Level-order traversal
 - Begin at root and visit nodes one level at a time
 - We will see the implementation when we learn Breadth-First Search of Graphs

Traversals of Binary Search Trees

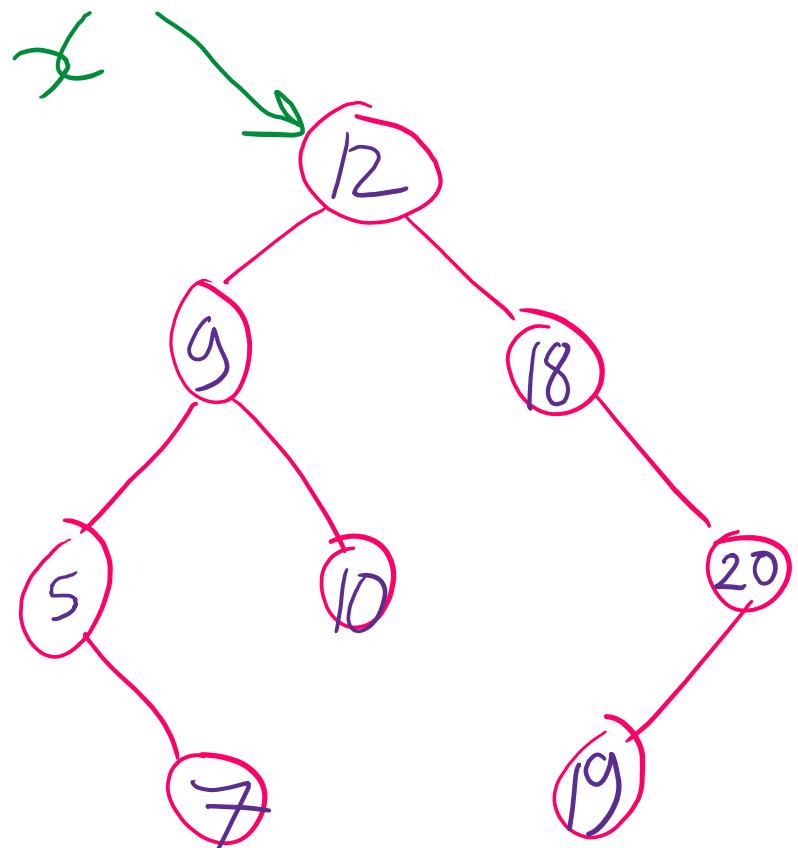
- Traversal for enumerating all items
- Traversal for finding an item

BST search using iteration



$x = \text{root}$
while($x \neq \text{null}$)
if equal break;
if $<$ $x = x.\text{left}$
if $>$ $x = x.\text{right}$
}

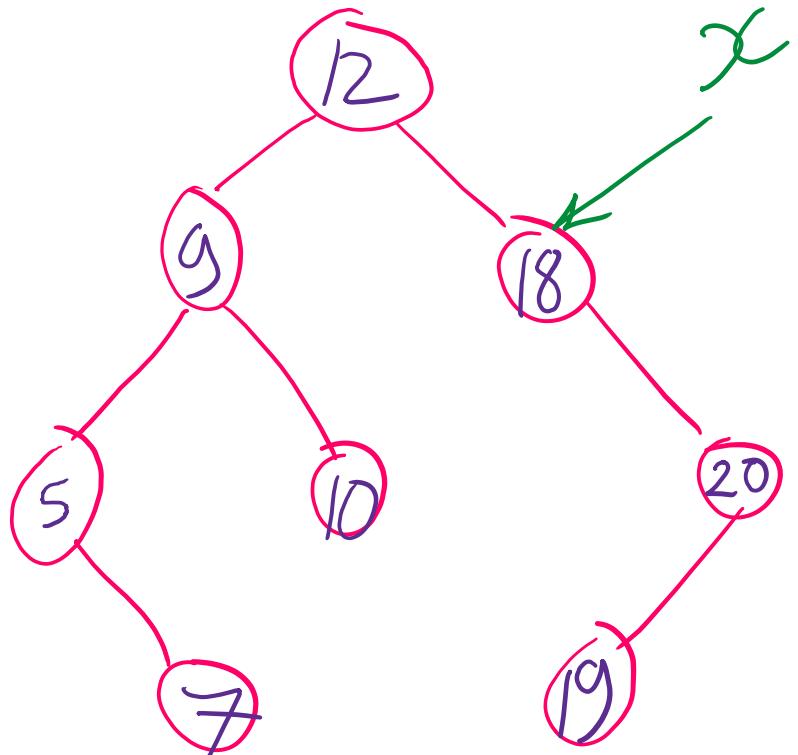
BST search using iteration



```
x = root  
while(x != null){  
    if equal break;  
    if < x=x.left  
    if > x=x.right  
}
```

Search for 19

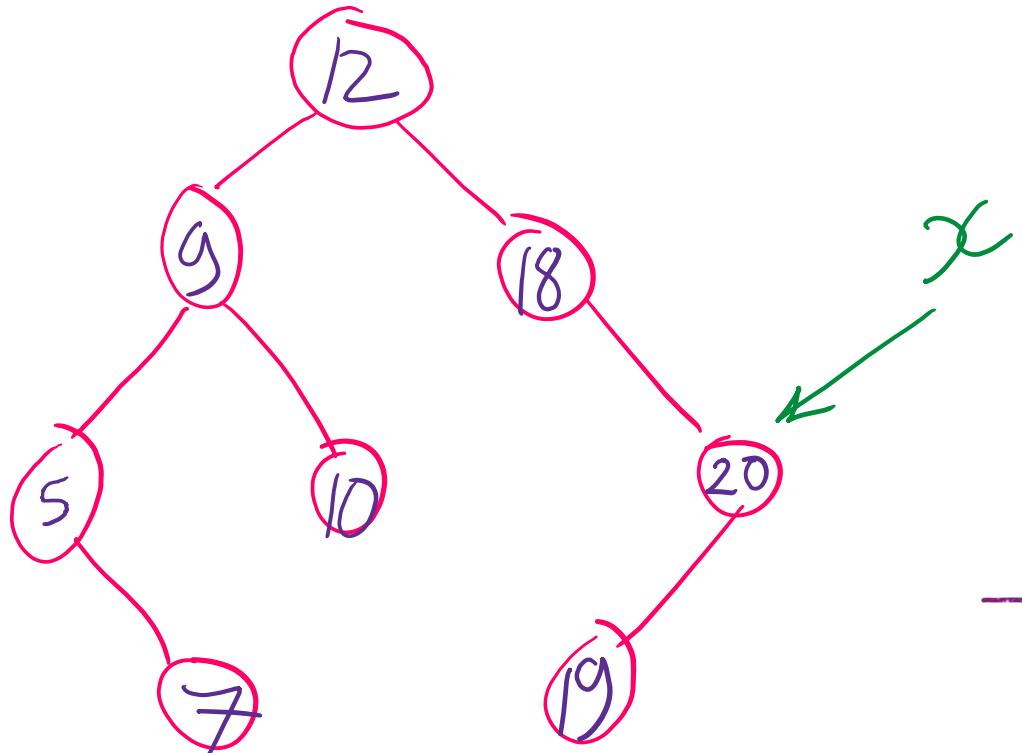
BST search using iteration



```
 $x = \text{root}$ 
while( $x \neq \text{null}$ )
    if equal break;
    if  $<$   $x = x.\text{left}$ 
    if  $>$   $x = x.\text{right}$ 
}
```

Search for 19

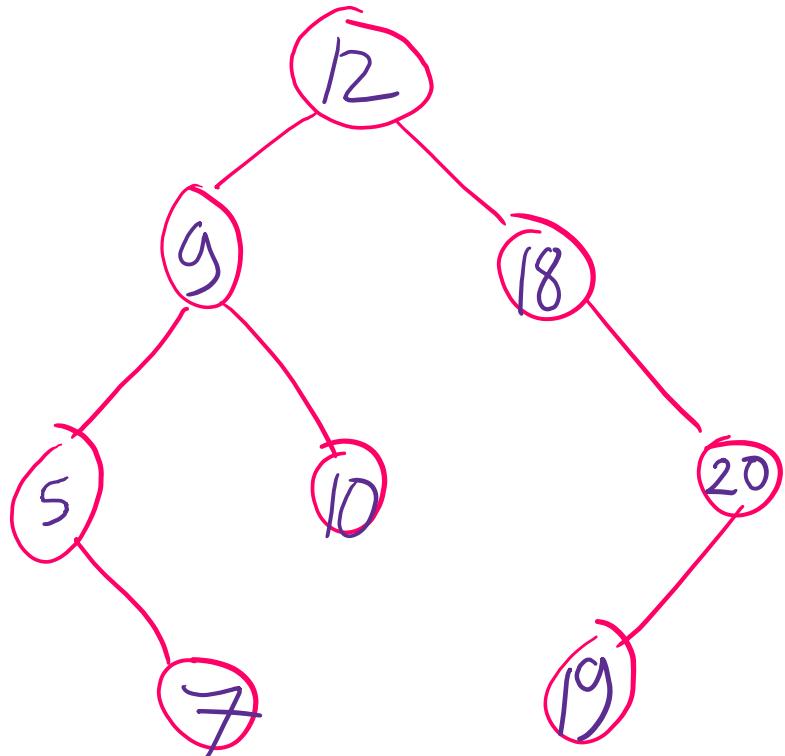
BST search using iteration



```
x = root  
while(x != null){  
    if equal break;  
    if < x = x.left  
    if > x = x.right  
}
```

Search for 19

BST search using iteration

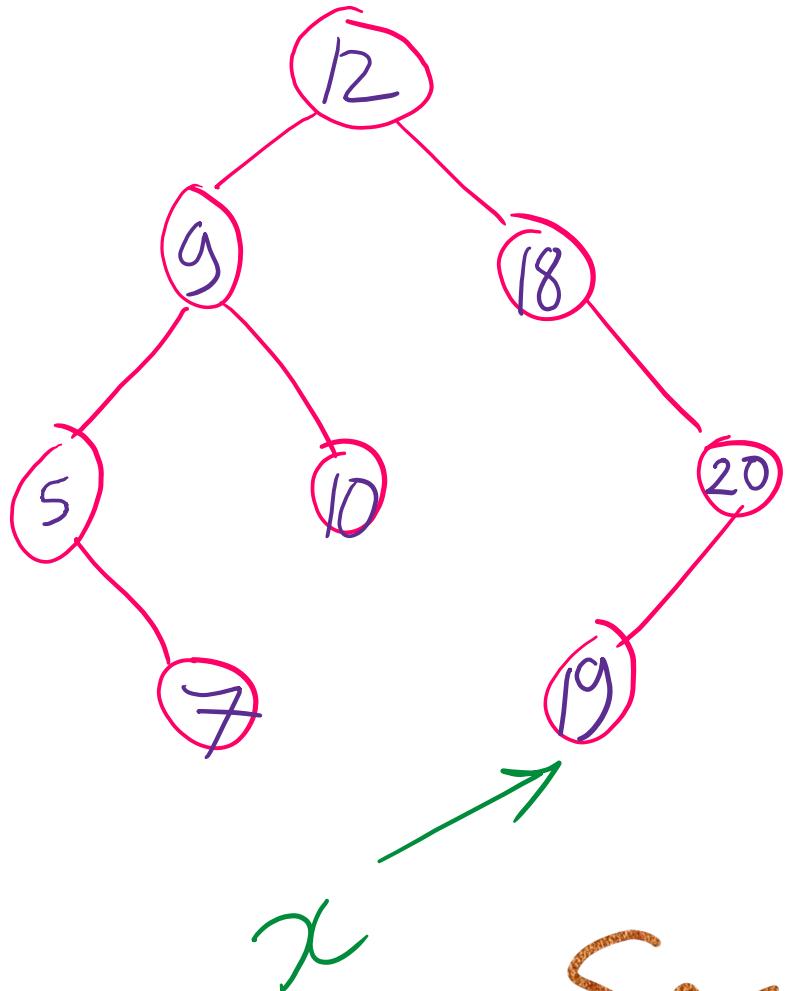


x

Search for 19

```
 $x = \text{root}$ 
while( $x \neq \text{null}$ ){
    if equal break;
    if  $<$   $x = x.\text{left}$ 
    if  $>$   $x = x.\text{right}$ 
}
```

BST search using iteration



Search for 19

```
 $x = \text{root}$ 
while( $x \neq \text{null}$ ){
    if equal break;
    if  $<$   $x = x.\text{left}$ 
    if  $>$   $x = x.\text{right}$ 
}
```

Symbol Table Implementations

| | Unsorted Array | Sorted Array | Unsorted L.L. | Sorted L.L. | BST | RB BST |
|--------|----------------|------------------|---------------|--------------------------|-------------|------------------|
| add | $\Theta(1)$ | $\Theta(n)$ | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(\log n)$ |
| Search | $\Theta(n)$ | $\Theta(\log n)$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(\log n)$ |
| | | Binary Search | | Binary Search | | |