Level Order Traversal

→ Start with root

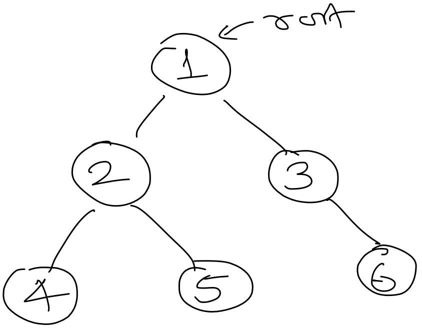
→ Print 1

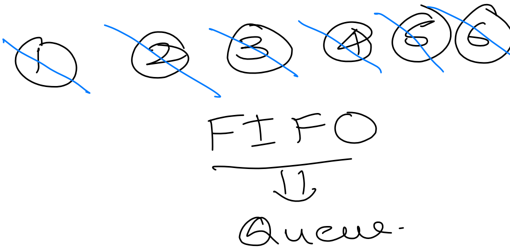
→ Goto node 2

→ Print 2

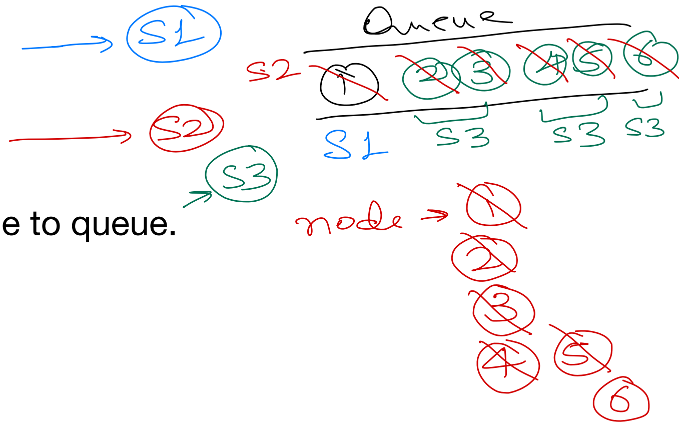
→ Goto node 3

→ Print 3





```
LevelOrderTraversal(root)
- if root is empty then
  - Stop.
- Add root node to queue.
- while queue is not empty do
  - Get a node from queue.
  - Process the node.
  - Add non empty child of node to queue.
```

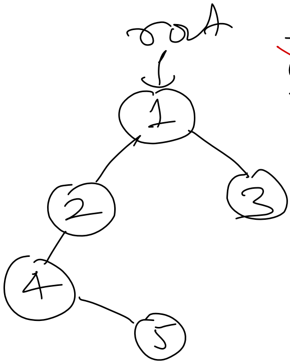


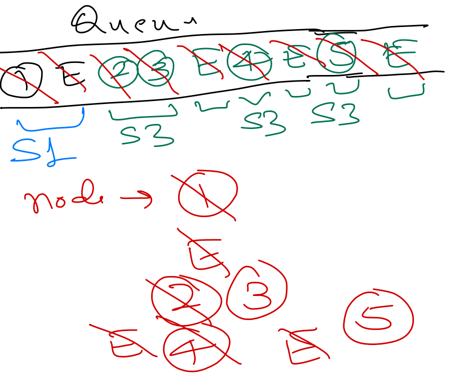
1 2 3 4 5 6

Output of Level Order should be, one line per level.

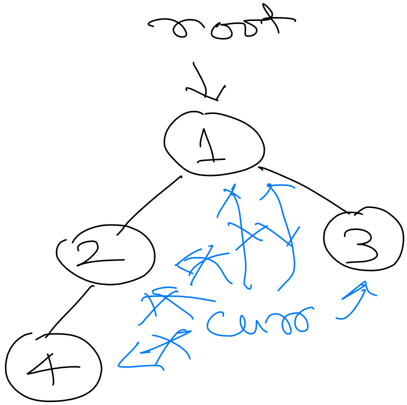
1
2 3
4 5 6

1.
→ 2 3
→ 4
→ 5



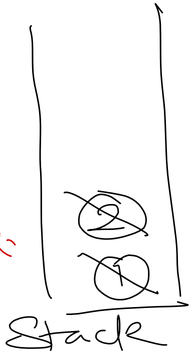


InOrder Traversal - Iterative



```
Inorder(root) - Iterative
- if root is empty then
  - Stop.
- Set current to root node.

- do
// Find the leftmost node of current.
- while current's left child is not empty do
  - Push current node on stack.
  - Move current to its left child.
- Process current node.

// Process parent of left sub tree that do not
have right child.
- while current node do not have right child do
  - Pop node from stack, into current.
  - Process current node.
```

Handle stack emty condi



```
- if current node had right child the
  - Set current to current's right child.

- while stack it not empty.
```

Terminating condition?

Binary Search Tree - BST

→ Search

→ Add

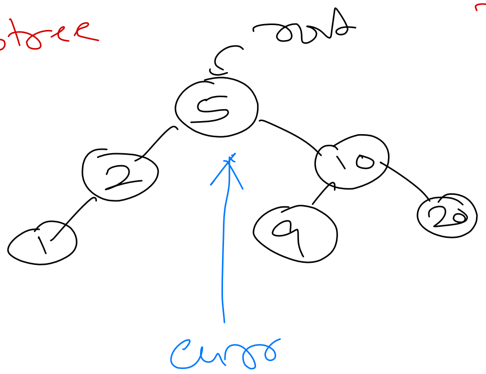
→ Delete

Each node satisfies following property

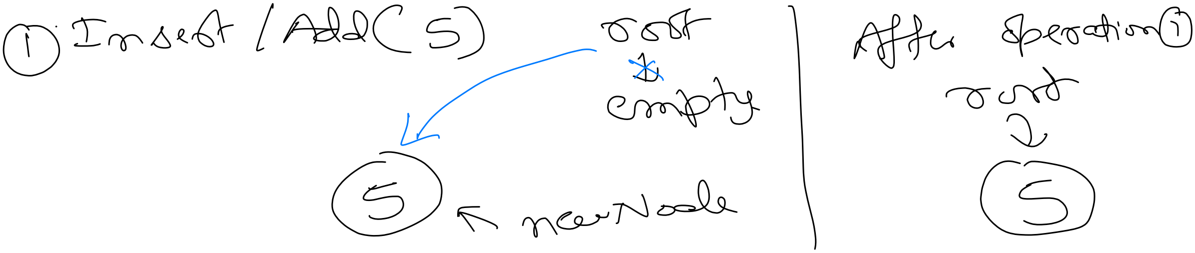
Data of nodes in left subtree < Node Data < Data of Nodes in right subtree.

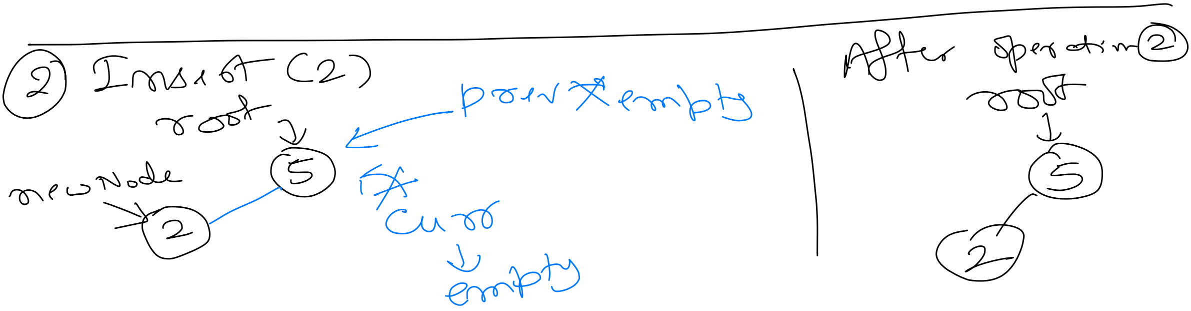
→ Search(3)

→ Search (4)



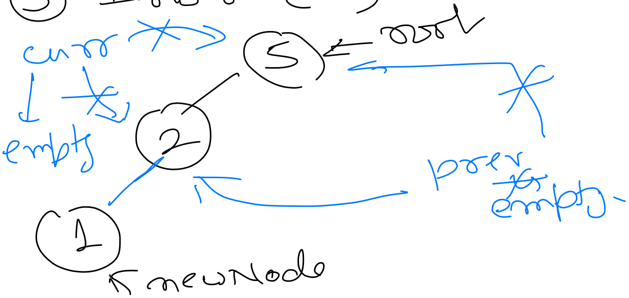
```
SearchInBST(root, element)
- Set current to root node.
- while current is not empty do
  - if current node's data is element then
    - End the loop.
  - if element < current node's data then
    - Move current to current's left child
  - Move current to current's right child.

- if current node is empty then
  - Element not found.
  Else
  - Element is present.
- Stop.
```



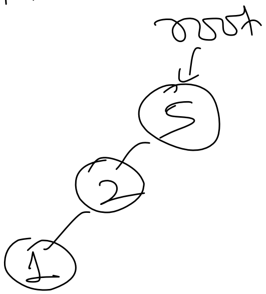


```
Insert(element) - In BST
- Make space to store the element, say newNode.
- Store element in newNode and set child nodes to empty.
- If tree is empty then
  - Make newNode as root node.
  - Stop.
- Set current to root node.
- Set previous to empty.
- while current is not empty do
  - Set previous to current.
  - if element < current node's data then
    - Move current to current node's left child.
  - Move current to current node's right child.
// Make new node a child of previous node.
- if element < previous node data then
  - Make newNode as left child of previous node.
- Make newNode as right child of previous node.
- Stop.
```

③ Insert (1)



After operation ③

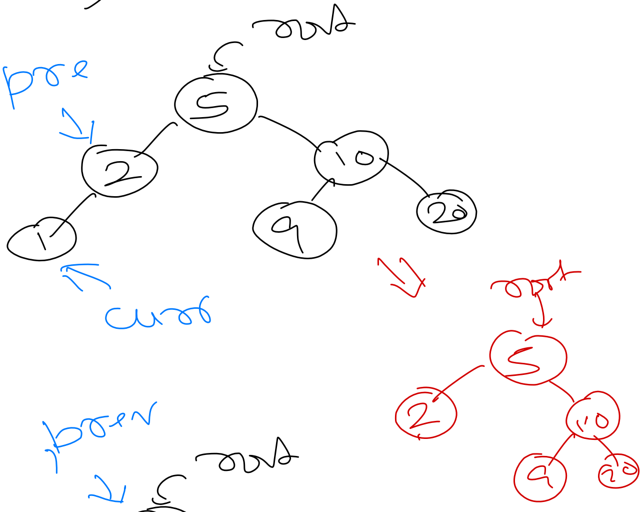


→ Delete (element)

→ Delete (1)

⇓

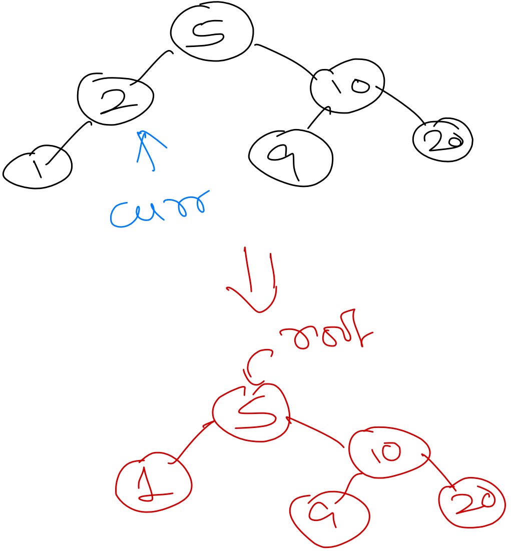
Delete leaf node.



→ Delete (2)

⇓

Delete a node with only one child.



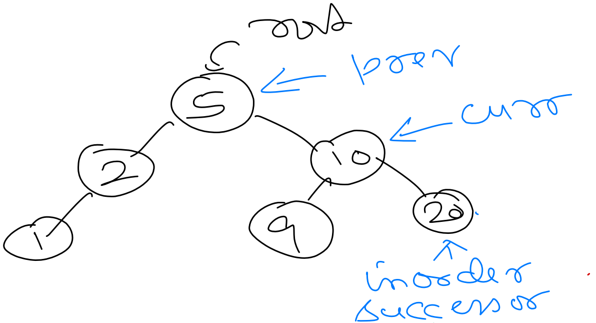
→ Delete (10)

⇓

Delete a node with two child.

⇓

We delete the inorder successor of the node to be deleted.



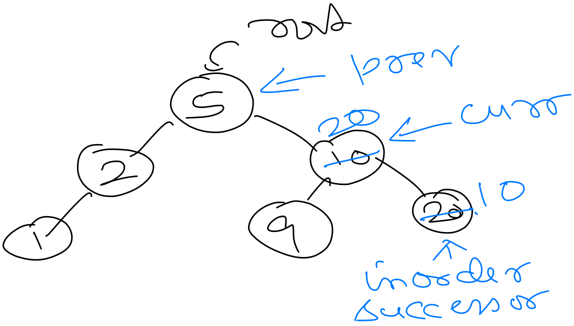
Inorder traversal

1 2 5

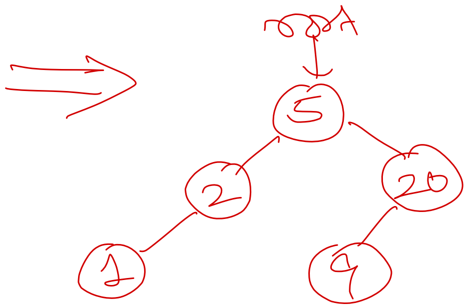
9 10 20

Node that is processed after this node, in inorder traversal.

Left most node in the right subtree.

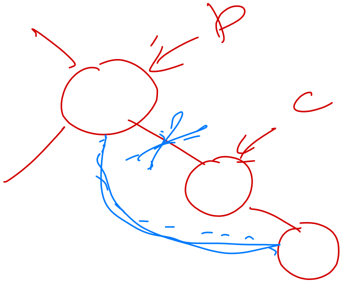


After deleting 10



→ Swap curr & inorder successor element & delete inorder successor.

```
Delete(element) - In BST
- Set current to root node.
- Set previous to empty.

// Find the node, current, that needs to be deleted.
- while current is not empty do
  - if current node's data is element then
    - End the loop.
  - Set previous to current node.
  - if element < current node's data then
    - Move current to current node's left child.
  - Move current to current node's right child.

- if current is empty then
  - Stop. // Element not present in BST.
```



```
// Delete current if its a leaf node.
- if current node is a leaf node then
  - if previous left child is current then
    - Set previous left child to empty.
 - Set previous right child to empty.
 - Stop.
```

Block1

```
// Delete current having only one child.
- if current left child is empty then
  // Make current's right child, a child of previous node.
  - if previous left child is current then
    - Set current node's right child as left child of previous.
   Else
    - Set current node's right child as right child of previous.
  - Stop.
- if current right child is empty then
  // Make current's left child, a child of previous node.
  - if previous left child is current then
    - Set current node's left child as left child of previous.
   Else
    - Set current node's left child as right child of previous.
```

Block2

```
  - Stop.

// Current had two children's.
// Find inorder successor of current.
- Set previous to current.
- Set inorderSuccessor to current node's right child.
- while inorderSuccessor have a left child do
  - Set previous to inorderSuccessor.
  - Move inorderSuccessor to its left child.

- Swap data of current and inorderSuccessor node.

// Delete inorderSuccessor node, that will either be leaf or only one child.
- Set current to inorderSuccessor.
// Perform Block 1
// Perform Block 2

- Stop.
```

→ what if we are deleting root node?

⇒ previous will be empty.