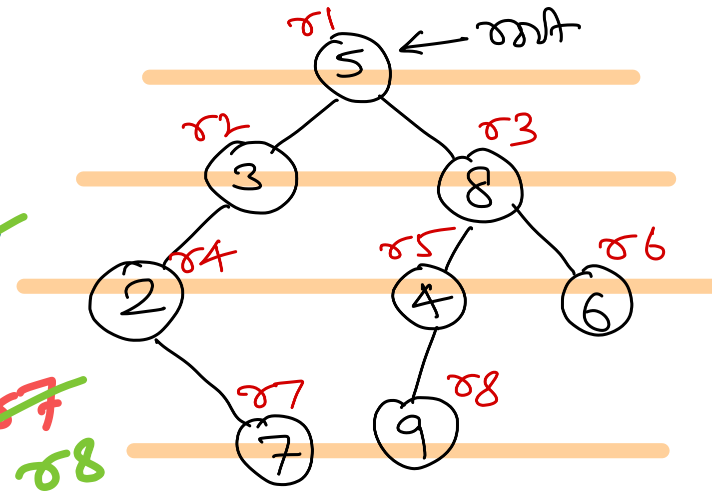
Level Order Traversal

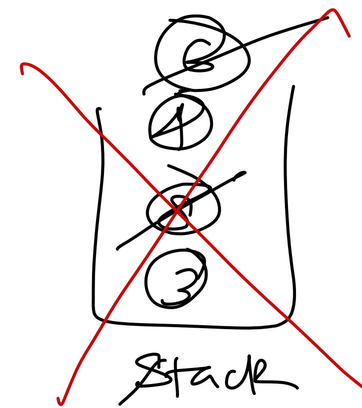


queue

~~r1~~ ~~r2~~ ~~r3~~ ~~r4~~ ~~r5~~ ~~r6~~ ~~r7~~ ~~r8~~

current → ~~r1~~ ~~r2~~ ~~r3~~ ~~r4~~ ~~r5~~ ~~r6~~ ~~r7~~ r8

O/P: 5 3 8 2 4 6 7 9



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

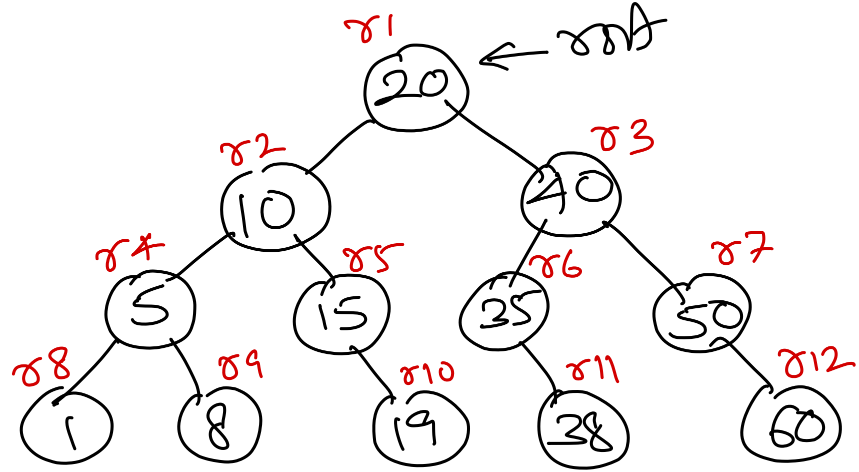
BST Property

value of left subtree < value of parent

Value of parent < value of right subtree

Binary Search Tree

Binary Search Tree → A binary tree in which each node satisfies Binary Search Tree (BST) property.



search(root, element)

root → r1

element → 5

⇓

search(r2, 5)

⇓

search(r4, 5) ✓

search (r1, 9)

$\Downarrow$

search (r2, 9)

$\Downarrow$

search (r7, 9)

$\Downarrow$

search (r4, 9)

$\Downarrow$

search (null, 9)

search (root, element)

→ if (root == empty) then

→ return false

→ if (root's data == element) then

→ return true;

→ if (element < root's data) then

→ return search (root's left child, element);

→ return search (root's right child, element);

||

⇓ Removing tail recursion

<u>search (root, element)</u>

→ while (root != empty)

→ if (root's data == element) then

→ return true;

→ if (element < root's data) then

→ root = root's left child.

else

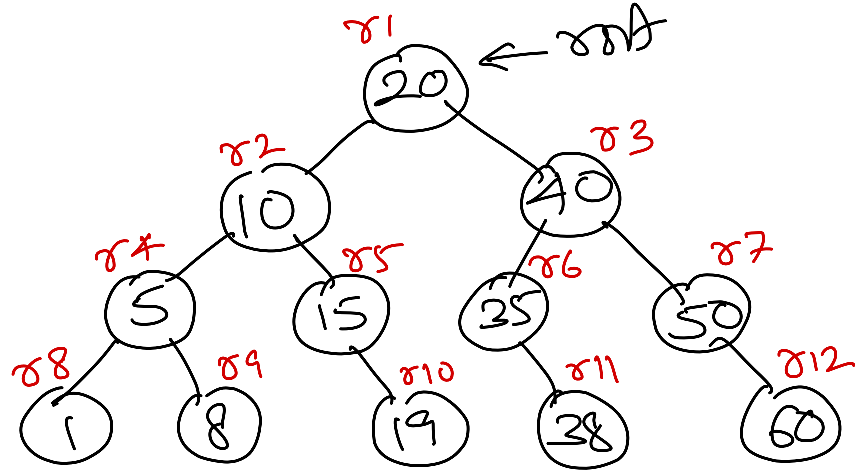
→ root = root's right child

→ return false;

Search (element)

- Set current to root.
- while (current is not empty) do
  - if (current node's data = element) then
    - Element found.
    - Stop.
  - if (element < current node's data) then
    - Move current to current's left child.

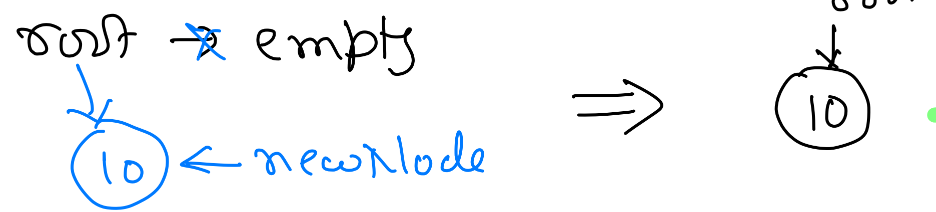
  Else
    - Move current to current's right child.
- Element NOT found.
- Stop



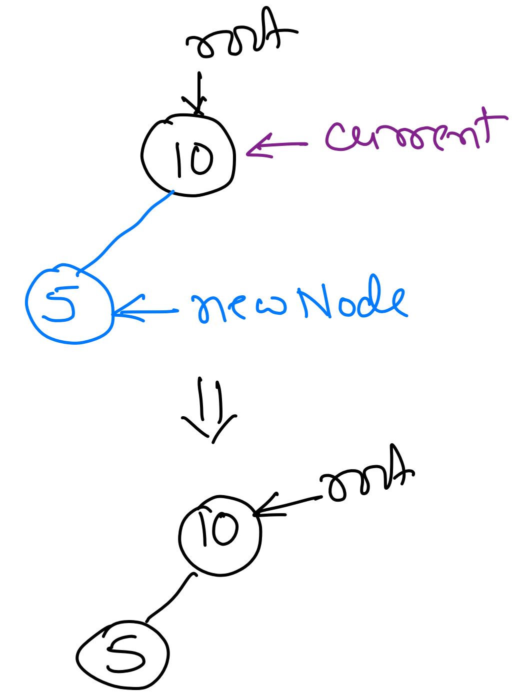
```java
public void treeOperation() {
    Stack<Node> stack = new Stack<>();
    Node current = root;

    while (current != null || !stack.isEmpty()) {
        while (current != null) {
            if (current.right != null) {
                stack.push(current.right);
            }
            stack.push(current);
            current = current.left;
        }

        current = stack.pop();

        if (!stack.isEmpty() && current.right == stack.peek()) {
            stack.pop();
            stack.push(current);
            current = current.right;
        } else {
            System.out.print(current.data + " ");
            current = null;
        }
    }
    System.out.println("");
}
```

Create a BST

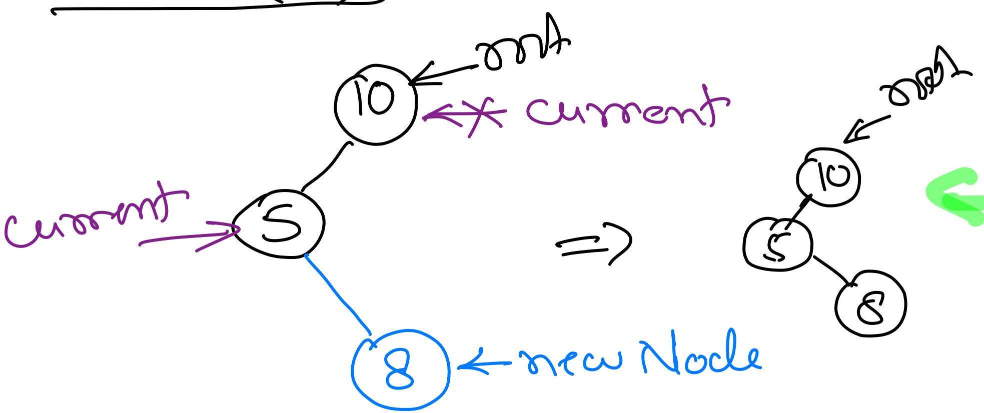
insert (10)



insert (5)



insert (8)



insert : 20, 15, 30, 1, 12

Insert (element)
- Make space for the new element, say newNode.
- Store the element in newNode's data.
- Set newNode's left and right child to empty.

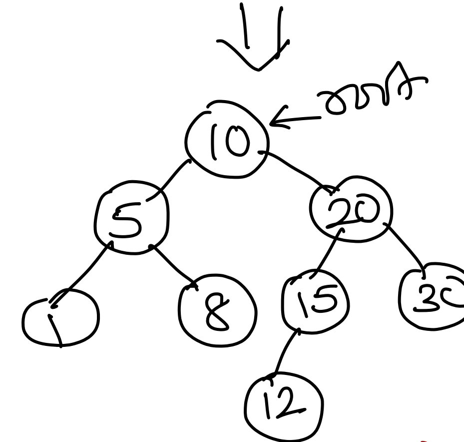
- if (tree is empty) then // root is empty
  - Make newNode as the root node of tree.
  - Stop.

- Set current to the root node.
- while (current is not empty) do
  - if (element < current node's data) then
    - if (current's left child is empty) then
      - Set newNode as left child of current.
      - Stop.
    - Move current to current's left child.

  Else
    - if (current's right child is empty) then
      - Set newNode as right child of current.
      - Stop.
    - Move current to current's right child.
- Stop

if (element == current. data)
→ return;

to not allow duplicate elements.

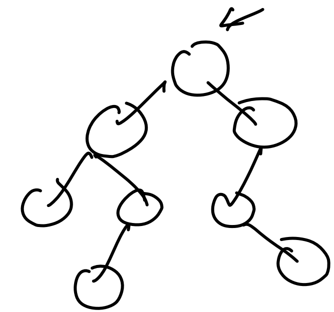


```
class BST {
    :
    private BTNode root;
    :
    private
    ~~public~~ void printUsingInorder(BTNode root)
    {
      :
    }
    public void print() {
        printUsingInorder(root);
    }
}
```

can't be called from outside class, as root is private & needs to be passed as argument

wrapper function

Time complexity of BST operations



Search

Iteration #		# of nodes
1	$\longrightarrow$	n
2	$\longrightarrow$	$\frac{n}{2}$
3	$\longrightarrow$	$\frac{n}{4}$
⋮	⋮	⋮
?	$\longrightarrow$	1

$\Rightarrow \log_2 n$

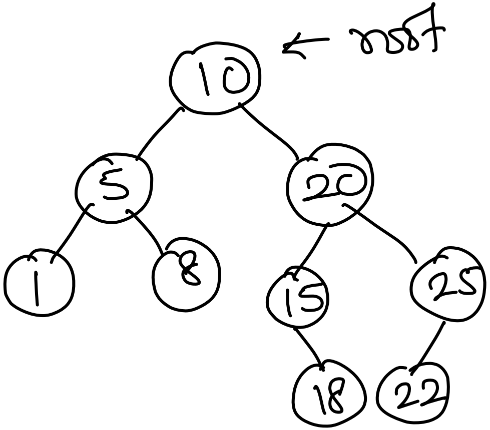
$\log_{10} 1000 = 3$

Search time complexity $= O(\log_2 n)$

lograthmic

Insert $= O(\log_2 n)$

Inorder Successor → Node that gets processed after a node, during inorder traversal.



Inorder traversal

1 5 8 10 15 18 20 22 25

↑

Inorder traversal of a BST give element in sorted order

Steps to find inorder successor (ios)

1. Set ios to right child of node.
2. while (ios has left child)
   → Move ios to ios left child.

Works when inorder successor is in a subtree of node.

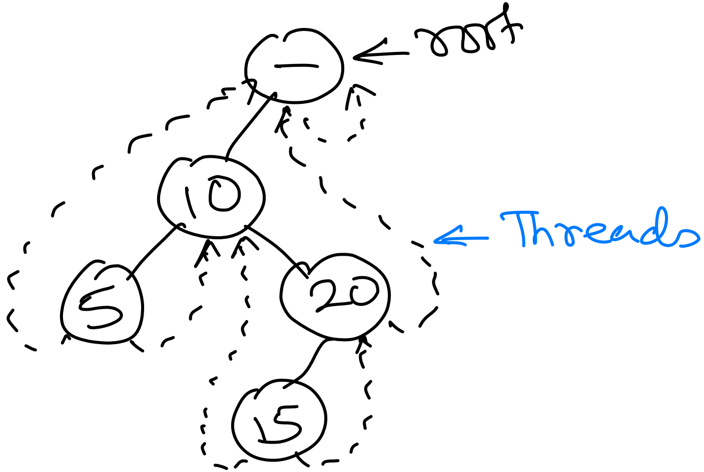
Inorder Predecessor

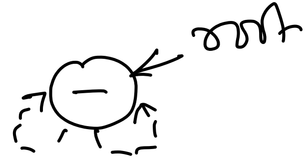
→ Node that gets processed before a node during inorder traversal.

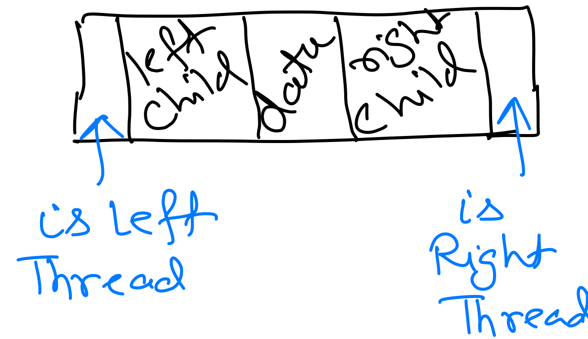
Steps to find inorder predecessor (iop)

1. Set iop to node's left child.
2. while (iop has right child)
   → Move iop to iop's right child.

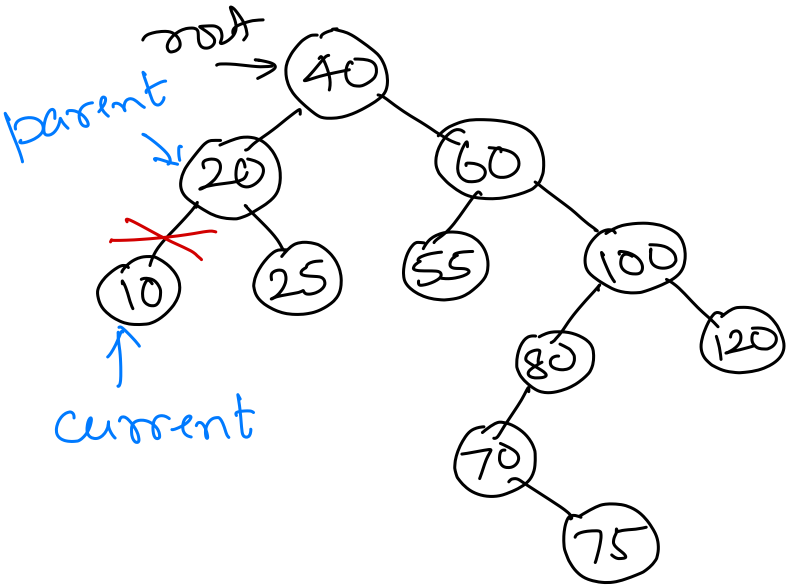
Threaded Binary Tree







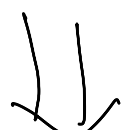
Delete in BST

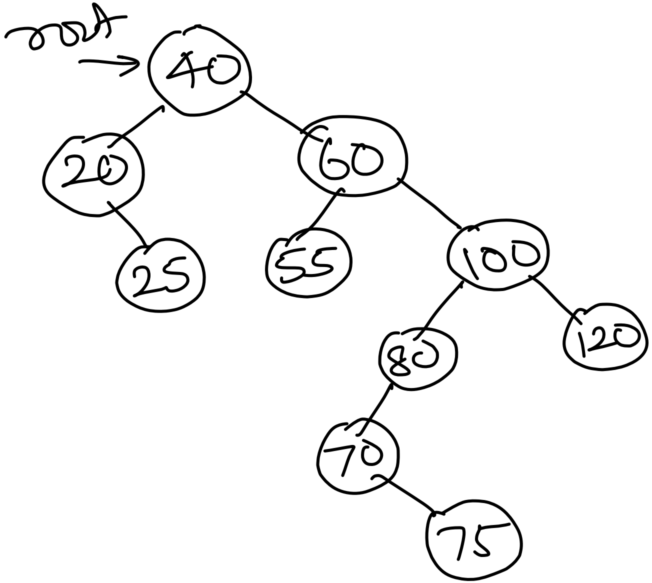


delete (10)

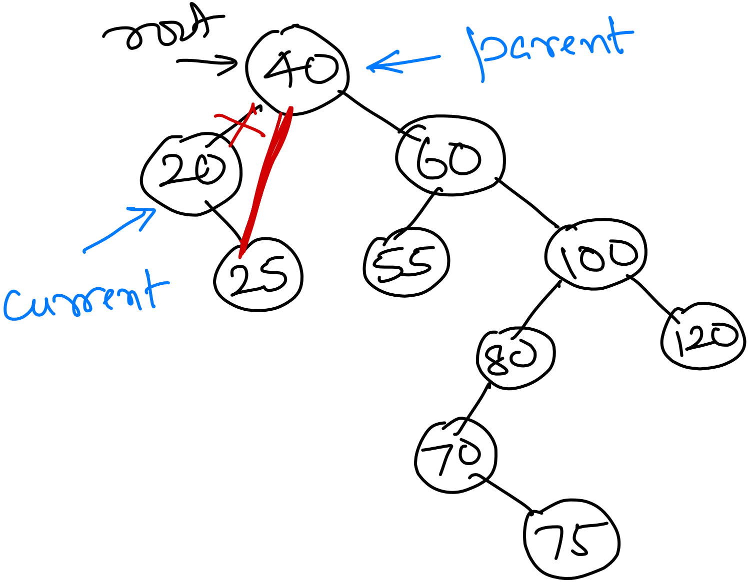
(1) when current node (node to be deleted) is a leaf node.

→ Remove current as a child of its parent.




root
40
20
60
25
55
100
80
120
70
75

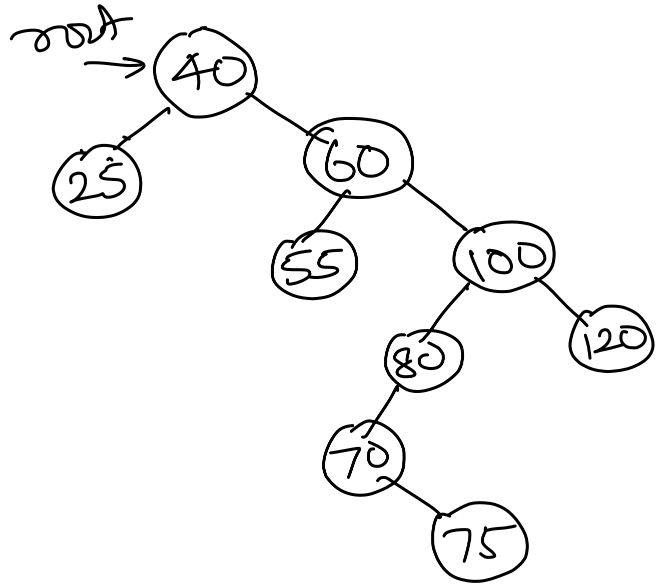
delete (20)

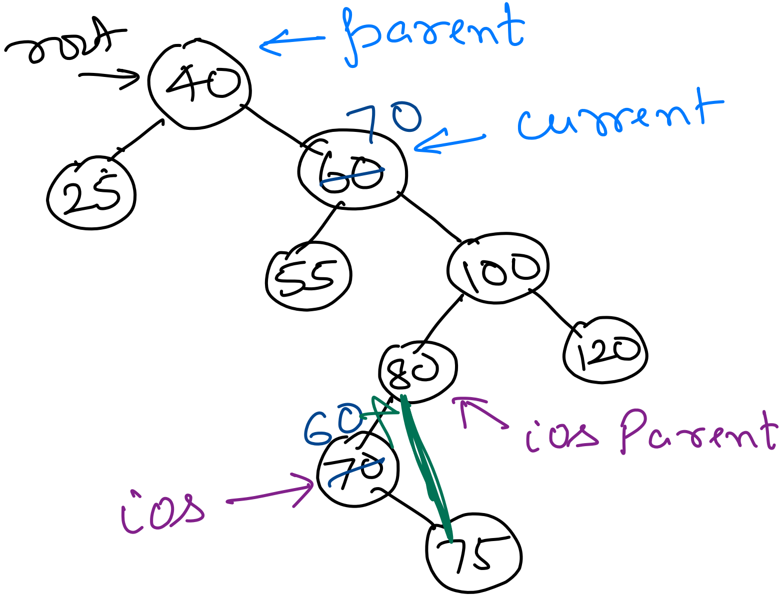


(2) when current node has one children.

→ current node's only child replaces current as child of parent.

⇓


root
40
25
60
55
100
80
120
70
75



delete (60)

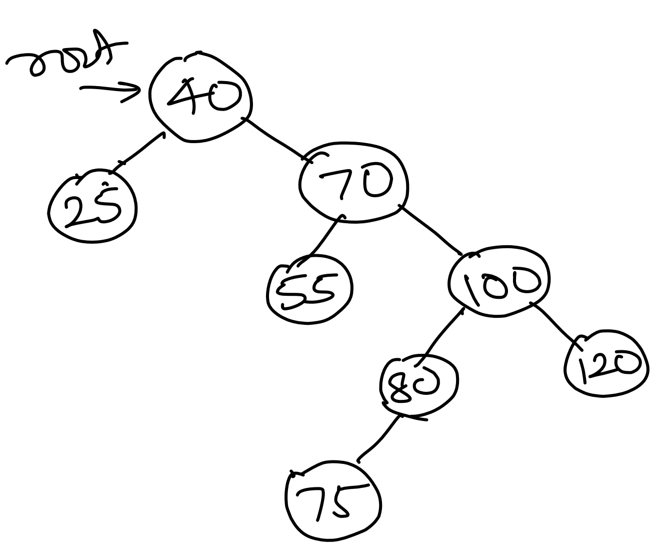
③ when current has both child nodes present.

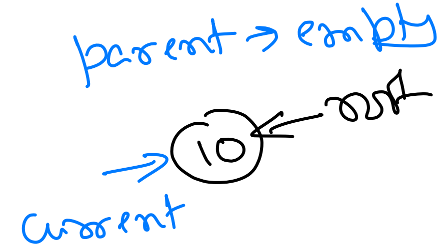
→ Find inorder successor/ predecessor of current.

→ Swap values of current and inorder successor/ predecessor.

→ Delete inorder successor/ predecessor

⇓





delete (10)

root → empty.

Delete(element)
// Find the node to be deleted.
- Set parent to empty
- Set current to root node.
- while (current is not empty) do
  - if (element = current node's data) then
    // Element found.
    - End the traversal.
  - Set parent to current.
  - if (element < current node's data) then
    - Move current to the current node's left child.
  Else
    - Move current to current node's right child.

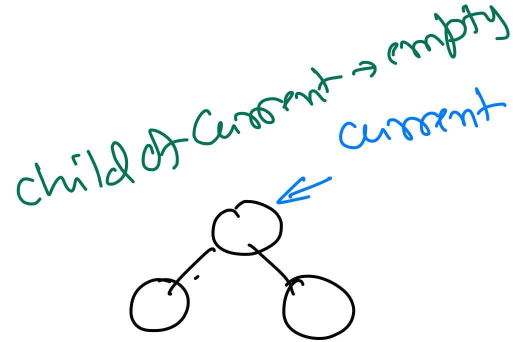
// Is an element found?
- if (current is empty) then
  // Element not found in tree.
  - Stop

Time complexity
?

// Deleting leaf node?
- if (current is leaf node) then // Leaf node => both child are empty
  // Are we deleting root node? => Deleting the only node in the tree.
  - if (current is root node) then
    - Set root to empty.
    - Release memory for the current node. // Not required in JAVA.
    - Stop.

// Delete current node, child of parent. But, which child?
- if (current is left child of parent) then
  - Set left child of parent to empty.

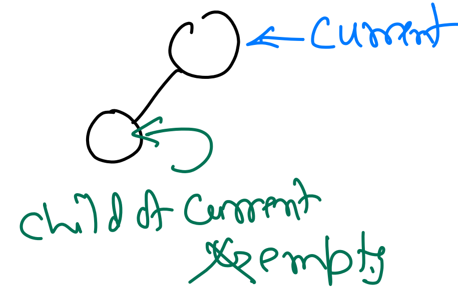
Else
  - Set the right child of the parent to empty.
- Release memory for the current node. // Not required in JAVA.
- Stop



// Deleting node with only one child?
- Set childOfCurrent to empty.
- If (current node's left child is empty) then // current has only right child.
  - Set childOfCurrent to current's right child.
- if (current node's right child is empty) then // current has only left child.
  - Set childOfCurrent to the current's left child.
- if (childOfCurrent is not empty) then // Current has only one child.

  // Set the only child of the current as the child of its parent.
  - if (current is left child of parent) then
    - Set childOfCurrent as left child of parent.

  Else
    - Set childOfCurrent as the right child of the parent.
  - Release memory for the current node. // Not required in JAVA.
  - Stop.

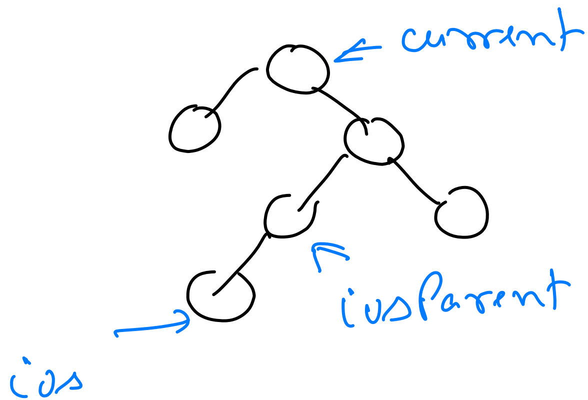


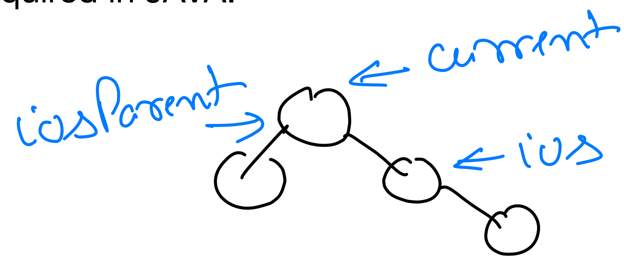
// Deleting node with two children.
// Find inorder successor of the current.
- Set inOrderSuccessorParent to current.

- Set inOrderSuccessor to the current node's right child.
- while (inOrderSuccessor has left child) do
  - Set inOrderSuccessorParent to inOrderSuccessor.
  - Move inOrderSuccessor to the left child of inOrderSuccessor.
- Swap data of current and inOrderSuccessor node.

// Delete inorder successor node.
// Inorder successor has max one child. => It will only be the right child.
- if (inOrderSuccessor is left child of inOrderSuccessorParent) then
  - Set right child of inOrderSuccessor as left child of inOrderSuccessorParent.

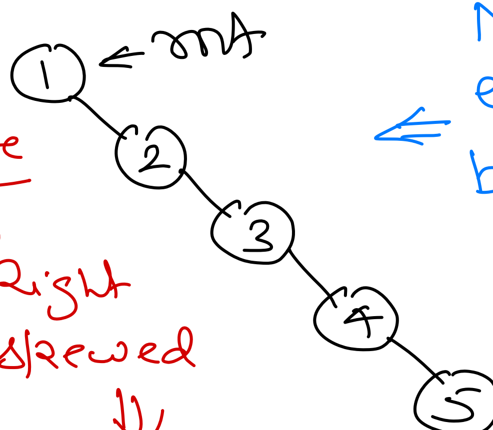
Else
  - Set the right child of inOrderSuccessor as the right child of inOrderSuccessorParent.
- Release memory for inOrderSuccessor node. // Not required in JAVA.
- Stop





Problem with BST

insert → 1 2 3 4 5



Skewed binary tree

- Left skewed ⇓ Each node has only left child
- Right skewed ⇓ Each node has only right child.

⇐ Nodes are NOT evenly distributed between left and right subtrees.

Search in BST of skewed BST

Iteration #	# of nodes
1 ⟶	n
2 ⟶	$(n-1)$
3 ⟶	$(n-2)$

Time complexity
$= O(n)$

?

$n \longrightarrow 1$

How to solve the problem?

→ Use balanced BST / self adjusting BST.

$\Downarrow$

nodes are evenly distributed after inserting in BST.

e.g. 2-3 Tree, Red-Black Tree,
AVL Tree.