SEP 2024

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

Binary Search Tree

'24
34th Week
233-133

AUGUST
20
TUESDAY

	Array (unsorted)	Array (sorted)	linked list	BST (balanced)	Hashtable
Search	O(n)	O(log n)	O(n)	O(log n)	O(1)
Insert	O(1)	O(n)	O(1)	O(log n)	O(1)
Delete	O(n)	O(n)	O(n)	O(log n)	O(1)
Find closest	O(n)	O(log n)	O(n)	O(log n)	O(n)
Sorted Traversal	O(n log n)	O(n)	O(n log n)	O(n)	O(n log n)

O(n) in case of sorted LL.

(*) (*)

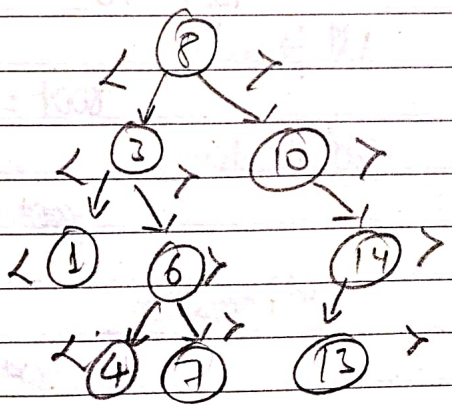
Binary search tree is a node-based binary tree DS which has the following properties :—

(a) The left subtree of a node contains only nodes with keys lesser than or equal to the node's key. [left ⇈ smaller]

(b) The right subtree of a node contains only nodes with keys greater than the node's keys. [right ⇈ greater]

(c) The left & right subtree each must also be a binary search tree.

There must be not 'Duplicate'.
Like LL it is a linked data structure. [Not Cache friendly]



TC : Worst case TC of search & insert operation is O(h)
Where h is height of BST.
In worst case we may have to travel from root to deepest leaf node.

AUGUST
21 '24
34th Week
234-132
WEDNESDAY

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

AUG 2024

Search in BST

Recursive

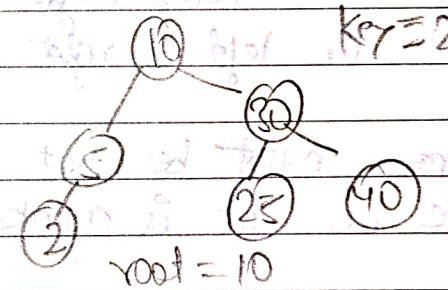
```
def search(root, key):
    if root is None:
        return False
    elif root.key == key:
        return True
    elif root.key > key:
        return search(root.left, key)
    else:
        return search(root.right, key)
```

TC = O(h) → (logn)
AS = O(h)

To search a given key in BST, first compare with root, if the key is present at root, return root. If the key is greater than root's key we recur for the right subtree of root node. otherwise, we recur for the left subtree. That's How it became O(logn) because we are ignoring the 1/2 of tree.

Iterative

```
def searchBST(root, key):
    while root != None:
        if root.key == key:
            return True
        elif root.key > key:
            root = root.left
        else:
            root = root.right
    return False
```



key = 25

root = 10
1st iteration

root = 30
2nd iteration

root = 25
3rd iteration

root True

Insert in BST

A new key is always inserted at the leaf. We start searching a key from the root until we hit a leaf Node. Once a leaf Node is found, the new Node is added as a child of leaf Node.

Approach

1. Start from the root
2. Compare the inserting element with root, if less than root, then recursively call left subtree, else recursively call right subtree.
3. After reaching the end, just insert that Node at left else right.

Recursive

```
def insert(root, key):
    if root == None:
        return Node(key)
    elif root.key == key:
        return root
    elif root.key > key:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root
```

Iterative

```
def insert(root, key):
    parent = None
    curr = root
    while curr != None:
        parent = curr
        if curr.key == key:
            return root
        elif curr.key < key:
            curr = curr.left
        else:
            curr = curr.right
    if parent == None:
        return Node(key)
    if parent.key > key:
        parent.left = Node(key)
    else:
        parent.right = Node(key)
    return root
```

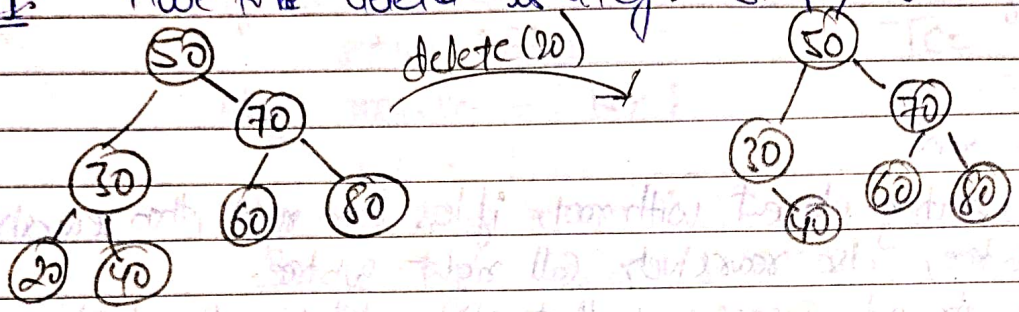
AUGUST
23
FRIDAY
'24
34th Week
236-130

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

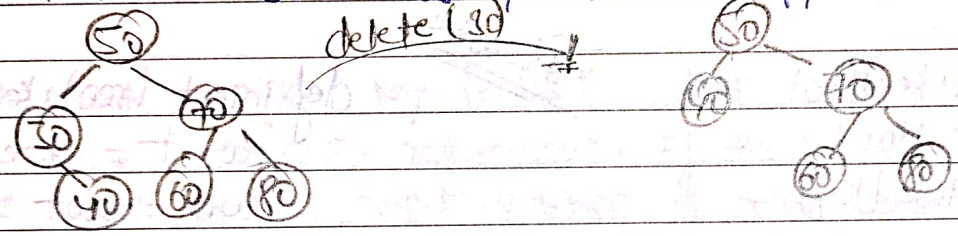
AUG 2024

Delete in BST

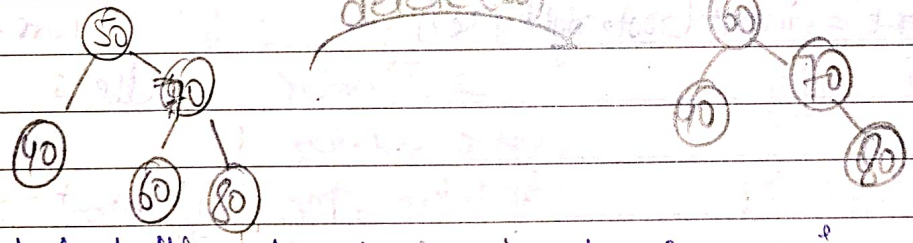
Case 1: Node to be deleted is a leaf: Simply remove from the tree.



Case 2: Node to be deleted has only one child: Copy the child to the node & delete the child



Case 3: Node to be deleted has two children: Find the inorder successor (It has to be greater) of the node. Copy contents of the inorder successor to the Node and delete the inorder successor. Note that inorder predecessor can also be used.



The important thing to note is, inorder successor is needed only when the right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in the right child of Node.

```
def deleteNode(root, key):
    if root == None:
        return
    if root.key > key:
        root.left = deleteNode(root.left, key)
    if root.key < key:
        root.right = deleteNode(root.right, key)
    else:
        if root.left == None:
            return root.right
        elif root.right == None:
            return root.left
        else:
            succ = getSucc(root.right, key)
            root.key = succ
            root.right = deleteNode(root.right, succ)
    return root
```

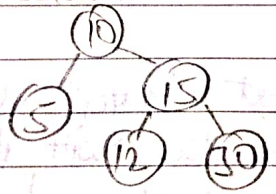
```
def getSucc(curr, key):
    while curr.left != None:
        curr = curr.left
    return curr.key
```

TC = O(h)
AS = O(h)

Floor in BST

→ We need to find out closest smaller (or) equal value.
→ If value is present, return Node with it value. [largest]
→ " " " Not " " largest value which is smaller than given value.

i/p: n=14, root of below tree
o/p: Node 12



i/p: n=4, root of same tree
o/p: None

i/p: n=30, root of same tree
o/p: Node 30

i/p: n=100, root of same tree
o/p: Node 30

AUGUST
25
SUNDAY
'24
35th Week
238-128

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

AUGUST 2024

→ A simple solution is to traverse the tree using (Inorder, Preorder (or) Postorder) and keep track of closest smaller (or) same element. Time Complexity — O(n)

→ Efficient solution is @ start at root Node.
- (b) If root → data == key floor of the key is equal to root
- (c) else if root → data → key, then floor of the key must lie in the left subtree.
- (d) Else floor may lie in the right subtree but only if there is a value lesser than or equal to the key.

If not, then root is key.

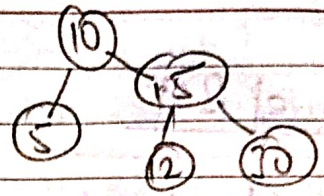
O(h) → (log n)

```
def getFloor(root, n):
    res = None
    while root != None:
        if root.key == n:
            return root
        elif root.key > n:
            root = root.left
        else:
            res = root
            root = root.right
    return res
```

Ceil in BST

→ We need to find out closest largest (or) equal value.
→ If value is present, return Node with its value (smallest)
→ " " " Not ", " smallest value which is largest than given Value

i/p: n = 14, Root of below Tree



o/p: Node 15

i/p: n = 3, root of some BST

o/p: Node 5

i/p: n = 40, root of some BST

o/p: None.

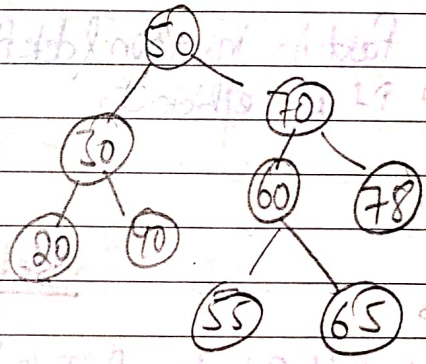
→ Naive soln is same as floor of BST.

Efficient Solution: — O(h) Time
O(1) Aux Space

① If root's key is same as n return root.
② If root's key is smaller, then change root to root's right.
③ If root's key is greater, update the result as root and change root to root's left.

```
def getCeil(root, n):
    res = None
    while root != None:
        if root.key == n:
            return root
        elif root.key < n:
            root = root.right
        else:
            res = root
            root = root.left
    return res
```

O(h)



We initialize result as None and traverse from root to a leaf using three steps.

AUGUST
27
TUESDAY
'24
35th Week
240-126

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

AUG 2024

Applications of BST

08

(1) To maintain stored stream of data (or sorted set of data).
09 (2) To implement doubly ended priority Queue.
(3) To solve problems like :—
10 (a) count smaller / greater in a stream.
(b) floor / ceiling / greater / smaller in a stream.
11 (4) BSTs are used for indexing.

12 ## Real time Applications

(1) BSTs are used for indexing in database.
(ii) It is used to implement huffman coding algorithms.
(iii) It is used to implement Dictionaries.

02

Advantages

03

(1) Fast in insertion & deletion.
(ii) 04 It is efficient.

Disadvantages

(1) Accessing is slightly slower than list.
(ii) we should always implement a balanced binary SearchTree.

05

Self Balancing BST

06

→ Self Balancing BST (or) height balanced BST trees that automatically keeps
07 height as small as possible when insertion and deletion operation are performed on Tree.
The height is typically maintained in order of $\log n$ so that all operation take $O(\log n)$ time on average.

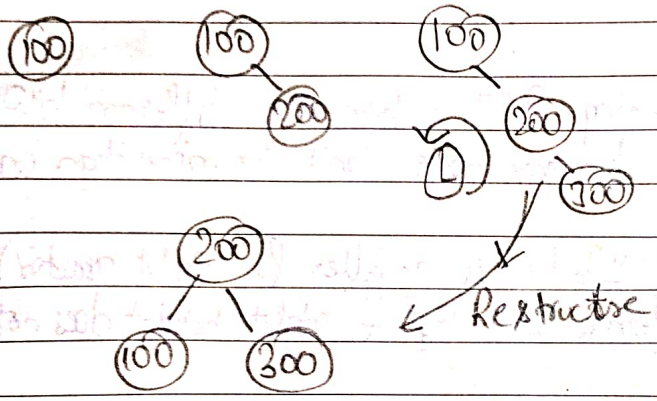
→ $O(\log n)$ [All cases]

Q How do self balancing - Tree maintain height?

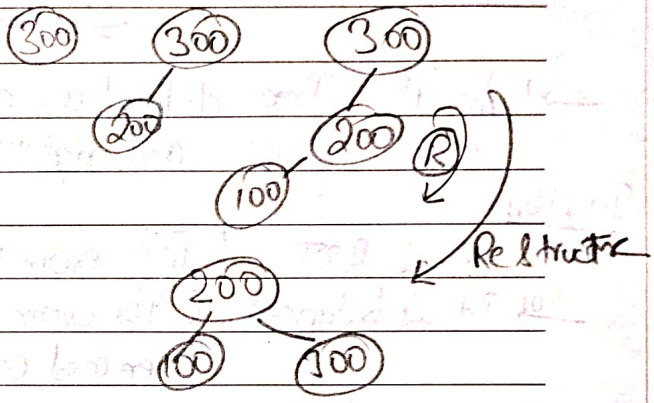
→ The idea is to do some restructuring (or rebalancing) when doing insertion / deletion.

(1) left rotation (2) Right rotation.

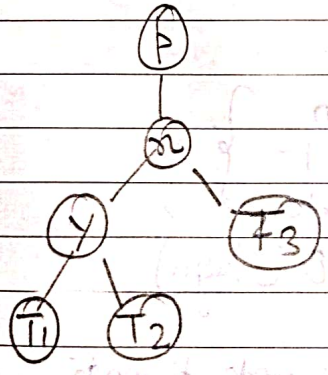
Insert 100, 200, 300 in BST



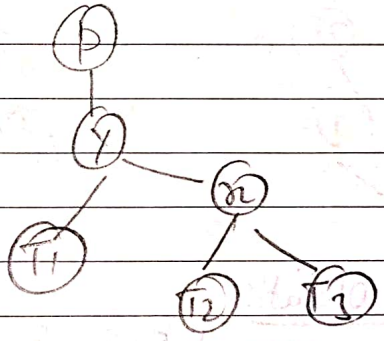
Insert 100, 200, 300 in BST.



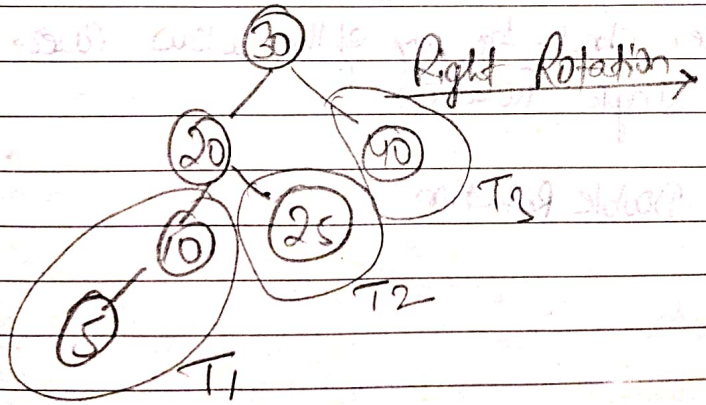
Rotation → O(1) operation

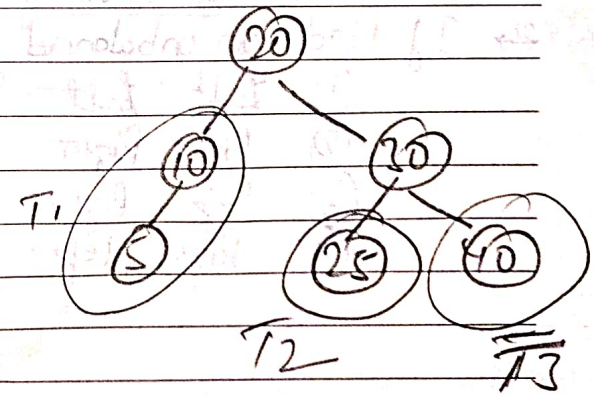


Right Rotation →
← Left Rotation



Eg





AUGUST
29
'24
35th Week
242-124
THURSDAY

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

AUG 2024

Most common Self balancing Trees are -
1. AVL Tree
2. Red black Tree [map & set]
3. splay Tree

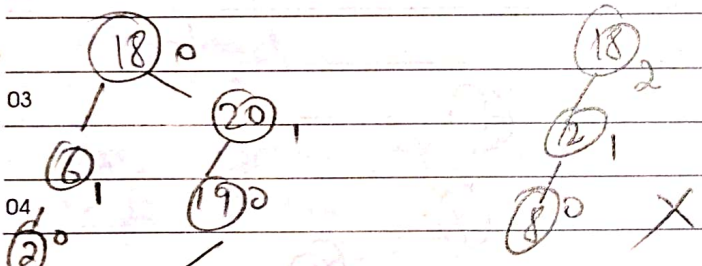
AVL Tree

→ An AVL Tree defined as a self balancing BST where the difference b/w height of left and right subtrees for any node can't be more than one.

In short,
→ It is BST (For every Node, left Subtree is smaller & right greater)
→ It is balanced (For every Node, difference b/w left & right height does not exceed one).

Balance factor = |lh − rh|
Balance factor = ≤ 1

eg

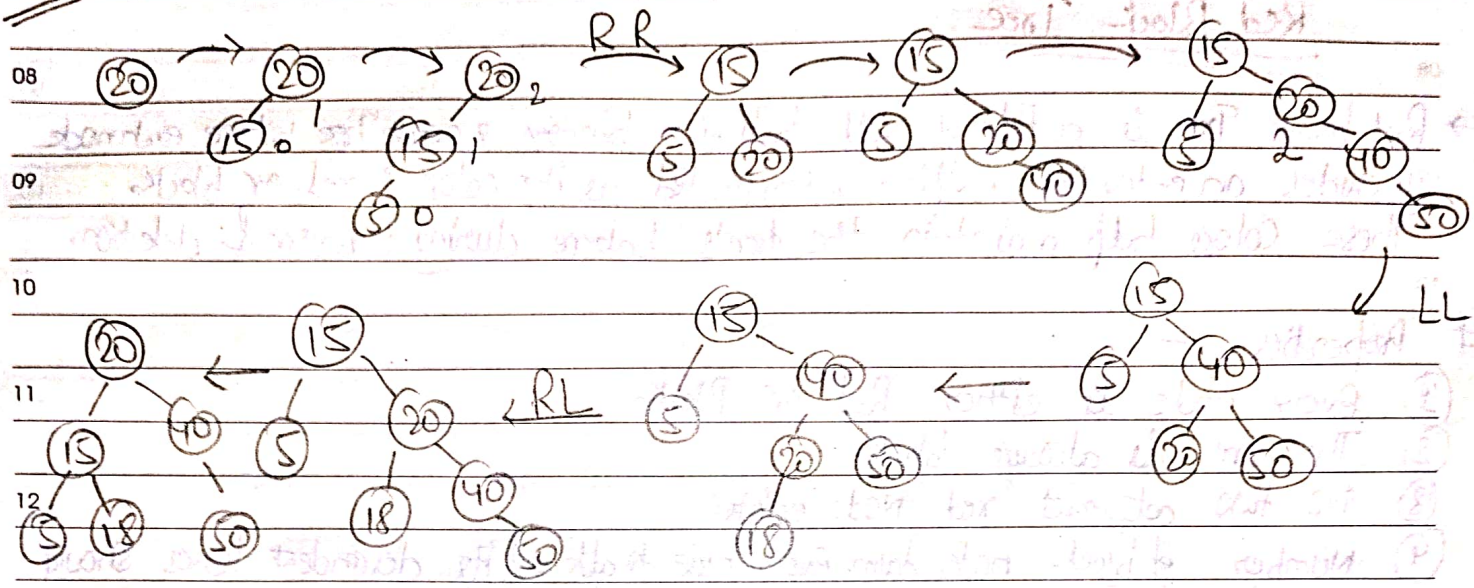


Search
Insert
Delete
TC = O(logn)

Insert operation

→ Perform normal BST Insert.
→ Traverse all ancestors of the newly inserted node from the node to root.
→ If find an unbalanced node, check for any of the below cases.
1. Left Left } Single Rotation
2. Right Right }
3. Left Right } Double Rotation
4. Right left }

eg. Insert 20, 15, 5, 40, 50, 18



height of AVL Tree

$h < c \log_2(n+2) + b$ Where $c \approx 1.4405$, $b \approx -1.3277$

$\Rightarrow O(\log n)$

Q Why AVL Tree?

Ans Most of the BST operations (eg., search, max, min, insert, delete) take O(h) time where h is the height of BST. The cost of these operation may become O(n) for a skewed binary tree. If we make sure that height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations.
The height of AVL Tree is always O(log n) where n is the No. of Nodes in tree.

→ It is used in map, set, multiset, multimap.

→ It is used when frequent lookups are required.

AUGUST
31
SATURDAY
'24
35th Week
244-122

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

AUG 2024

Red Black Tree

→ Red black Tree is a type of self balancing binary search Tree where each node includes an extra bit, often interpreted as its color: red or black. These colors help maintain the tree's balance during insertion & deletion.

* Properties :—

1. Every node is either Red or Black.
2. The root is always black.
3. NO two adjacent red Node exists
4. Number of black nodes from every node to all of its descendent leaves should be same.
5. All leaf Node are black.

Search
Insert
Delete
} O(log n) (or) height of tree.

Applications

1. Database :— MySQL uses Red black tree for indexing, optimizing search & insertion operation.
2. operating system :— In CPU scheduling.
3. Library function :— C++ STL's map & Java's Treemap.
4. Machine learning :— K means clustring algorithms.

→ As universal Data structure
→ used as Database Interaction / Transactions.

Check BST

Naive for every Node:
(1) find maximum in left subtree.
(2) Find minimum in right subtree.
(3) check that the value of current node is greater than the maximum and smaller than the minimum.
TC = O(n²)

Efficient
A binary Tree is BST if the inorder traversal is in sorted increasing order.
TC = O(n)
AS = O(h)

M1

```
INT_MAX = 4294967296
INT_MIN = -4294967296

def isBST(node):
    return (isBSTil(node, INT_MIN, INT_MAX))

def isBSTil(node, mini, maxi):
    if node is None:
        return True
    if node.data < mini (or) node.data > maxi:
        return False
    return (isBSTil(node.left, mini, node.data-1) and
            (isBSTil(node.right, node.data+1, maxi))
```

M2

```
INT_MIN = -4294967296
prev = INT_MIN
def isBST(root):
    global prev
    if root == None:
        return True
    if isBST(root.left) == False:
        return False
    if root.key <= prev:
        return False
    prev = root.key
    return isBST(root.right)
```

Ceiling on the left side

→ Ciel value which is equal to element of greater.

i/p: arr = [2, 8, 30, 15, 25, 12]
o/p: -1 -1 -1 30 30 15 } — smaller = 15 ✓

SEPTEMBER
02
'24
36th Week
246-120
MONDAY

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

SEP 2024

M-1

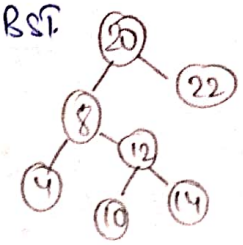
```
def printdel(arr):
    n=len(arr)
    print("-1", end=" ")
    s= Set()
    s.add(arr[0])
    for i in range(1,n):
        it =[x for x in s if x >=arr[i]]
        if len(it) ==0:
            print("-1", end=" ")
        else:
            print(min(it), end=" ")
        s.add(arr[i])
```

TC: (nlogn)

happiness is not perfected until it is shared

Find Kth Smallest in BST

→ Given the root of a BST and K as input, find Kth smallest element in BST.



eg if K=3 → 10
if K=5 → 14

M-1 Using Inorder Traversal [TC = O(n), SC = O(h)]

```
def PrintKth (root, K):
    if root != None:
        PrintKth (root.left, K)
        PrintKth.count += 1
        if PrintKth.count == K:
            Print (root.key)
            return
        PrintKth (root.right, K)
```

M-2 The idea is to maintain the rank of each node. We can keep track of element in the left subTree of every node while building the tree. Since, we need the kth smallest element, we can maintain the no. of element of the left subtree in every node.

TC = O(h)

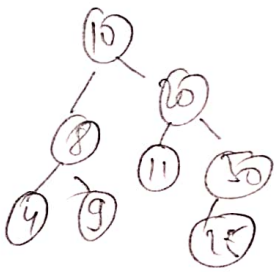
```
def update-size (node):
    if node is not None:
        node.size = 1 + (node.left.size
            if node.left else 0) +
            (node.right.size if node.right
                else 0)

def PrintKth (root, K):
    if root is None:
        return None
    left-size = root.left.size if root.left
                else 0
    if K == left-size + 1:
        Print (root.key)
        return
    elif K <= left-size:
        PrintKth (root.left, K)
    else:
        PrintKth (root.right, K-left-size-1)
```

Pair Sum with Given BST

Given a Balanced BST and a target sum, write a function that returns True if there is a pair with sum equals to target sum, otherwise return False.

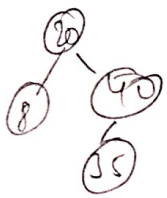
Using Hashing

```
def isPairSum (root, sum, S):
    if root == None:
        return false
    if isPairSum (root.left, sum, S):
        return True
    if sum - root.key in S:
        Print ("Pair found: {} + {} = {}".
            format (sum - root.key,
                root.key, sum))
    else:
        S.add (root.key)
    return isPairSum (root.right, sum, S)
```



sum = 33

(8, 25)

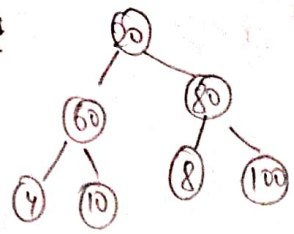


sum = 49

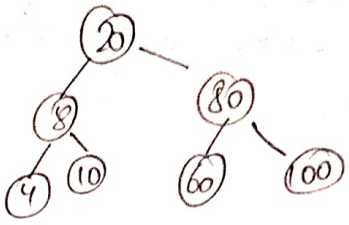
False

Find the BST for two Nodes Swapped

eg1

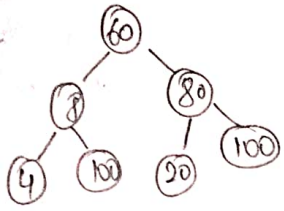


Swap 8 & 60 then BST →

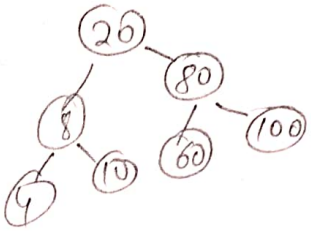


Not a BST

eg2



Swap 20 & 60 the BST →



Not a BST

Approach : The inorder traversal of a BST produce a sorted Array. So a simple method is to store inorder traversal of the input tree in an auxiliary array. Sort the Auxiliary Array. Finally, insert the auxiliary array elements back to the BST, keeping the structure of BST same.

TC=O(nlogn)

```
def CorrectBSTutil(root, first, middle, last, Prev):
    if (root):
        CorrectBSTutil(root.left, first, middle, left, Prev)
        if (Prev[0] and root.key < Prev[0].key):
            if (not first[0]):
                first[0] = Prev[0]
                middle[0] = root
            else:
                last[0] = root
        Prev[0] = root
        CorrectBSTutil(root.right, first, middle, last, Prev)
```

```
def CorrectBST(root):
    first, middle, last, Prev = [None][None]...
    CorrectBSTutil(root, first, middle, last
                   Prev)
    if (first[0] and last[0]):
        first[0].key, last[0].key =
            (last[0].key, first[0].key)
    elif (first[0] and middle[0]):
        first[0].key, middle[0].key =
        (middle[0].key, first[0].key)
```

```
def PrintInOrder(root):
    if (root):
        PrintInOrder(root.left)
        print(root.key, end=" ")
        PrintInOrder(root.right)
    else:
        return.
```