Today's Content

1. BST Intro
2. Search in BST
3. Insert in BST
4. Construct Balanced BST from sorted array.

1. Binary Search Tree

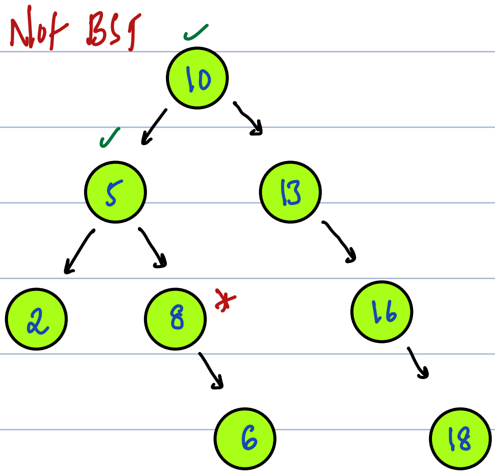
A BT is BST if

For all nodes {All nodes in LST < Node < All nodes in RST}

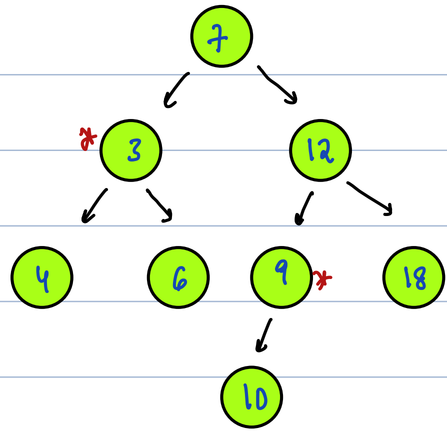
#Note1: If we have a null, assume it holds property

#Note2: In BST, values are distinct

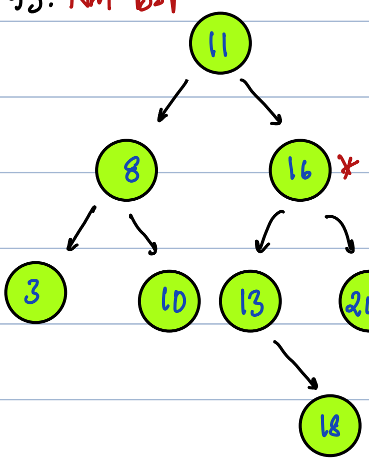
Eg1: Not BST



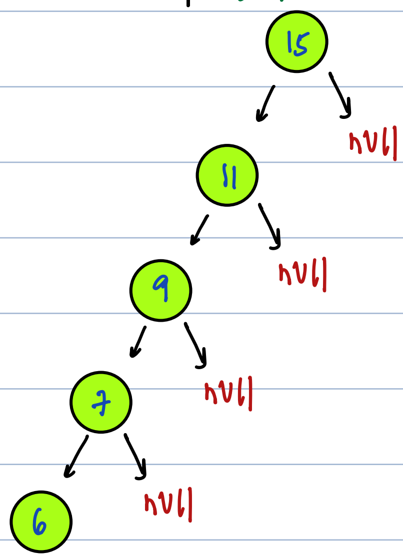
Eg2 Not BST



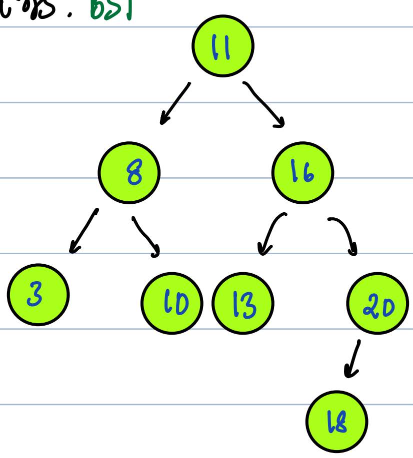
Eg3: Not BST



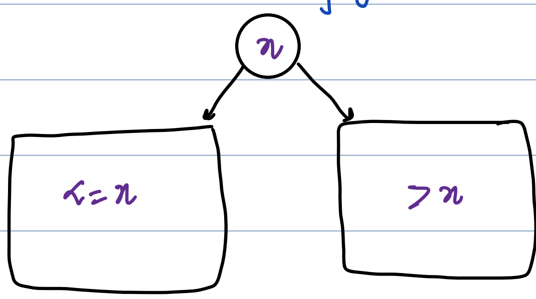
Eg4: BST



Eg5: BST

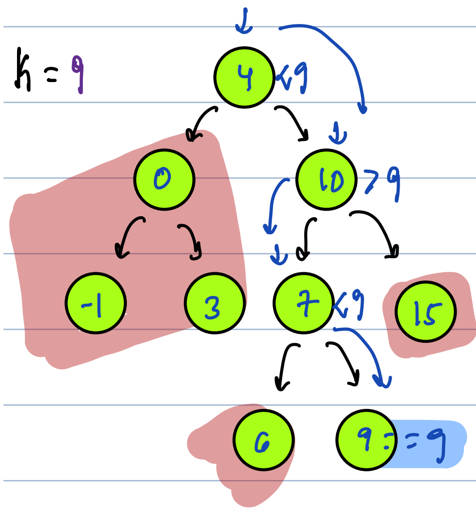


Note: In BST ideally repeating not allowed, but if needed stick to one side {left}

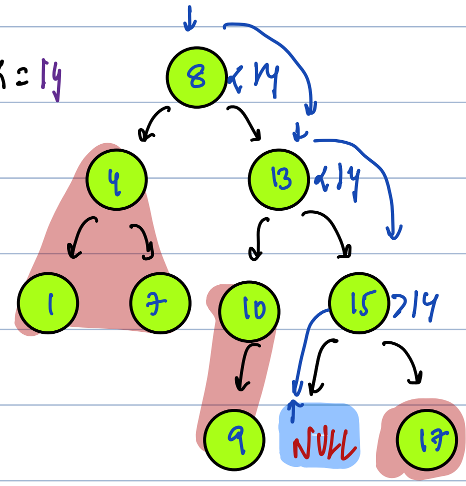


Search k in BST

k = 9



k = 14




```
bool search(Node *root, int k){    TC: O(H)  # H is height of BST.
    while(root != nullptr){
        if(root->data == k){return true;}
        if(root->data > k){ # Discard right
            root = root->left;
        }
        else{ # Discard left
            root = root->right;
        }
    }
    return false;
}
```

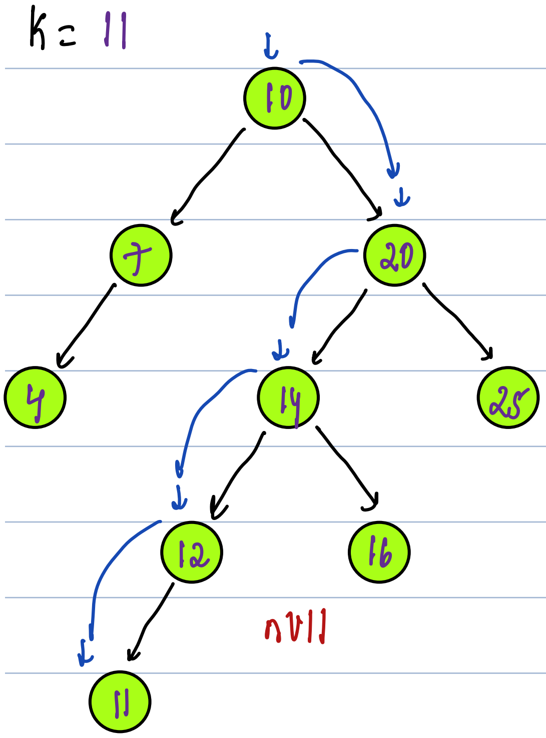
Insert in BST

Note1: After Insertion BST property should still hold.

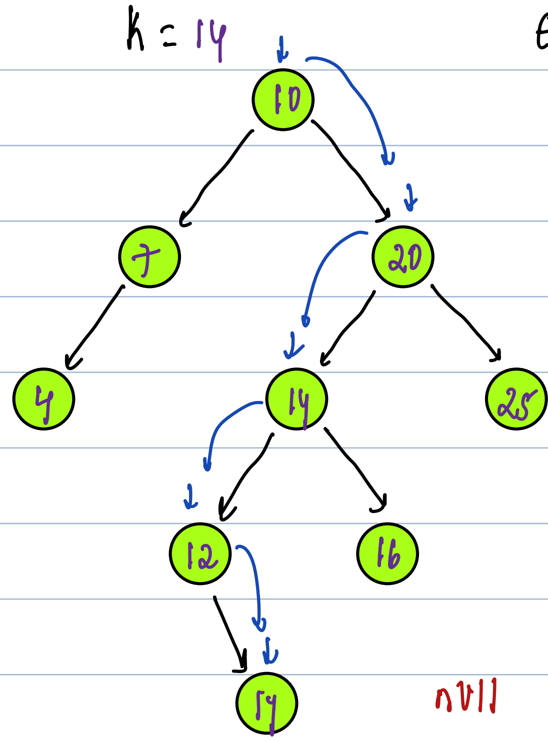
Note2: We always insert new node at null/Empty slot

Ex1:

k = 11



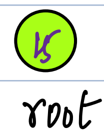
k = 14



Ex3:

root = null

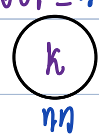
k = 15

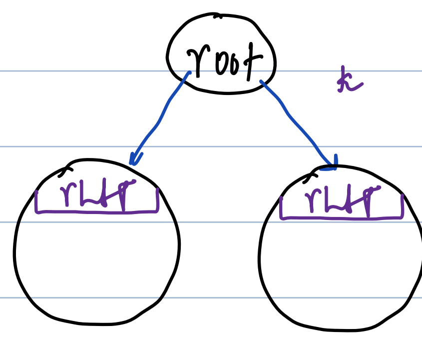


Ans: Given root node of BST, Insert k at correct pos & return root node.

```
Node* insert(Node* root, int k){  TC: O(H)
    if(root==null){
        Node *nn = new Node(k);   ] Node creation
        return nn;
    }
    if(root->data>=k){  #Insert k in LT
        root->left = insert(root->left, k);
    }                                          ] linking
    else{
        root->right = insert(root->right, k);
    }
    return root;
}
```

root = null





10

insert(root=10, k=13){

    10 → right = insert(10 → right, k); #20
    return root; 10

insert(root=20, k=13){

    20 → left = insert(20 → left, k); #14
    return root; 20

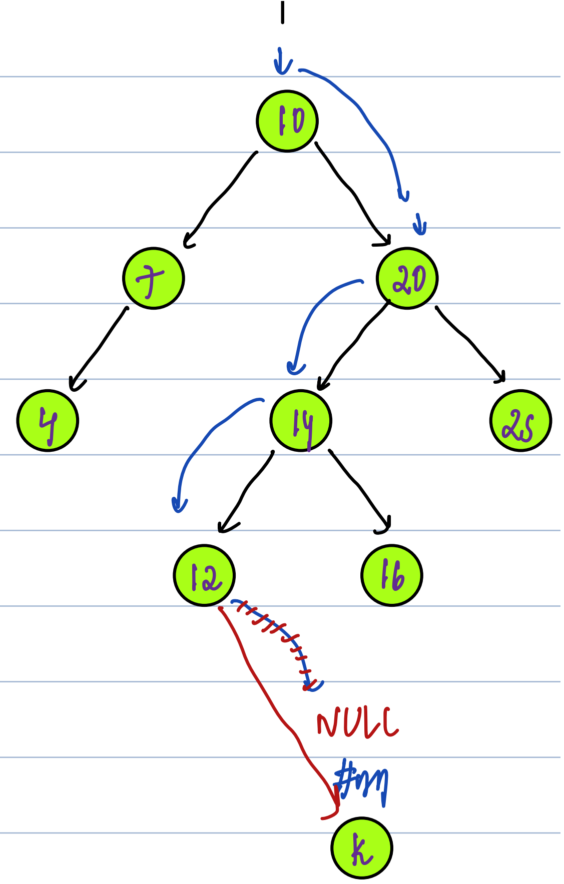
insert(root=14, k=13){

    14 → left = insert(14 → left, k); #12
    return root; 14

insert(root=12, k=13){

    12 → right = insert(12 → right, k); #nn
    return root; #12

insert(root=null, k=13){
    Node* nn = new Node(nn);
    return nn;



Balanced Binary Tree:

A Binary Tree is said to be balanced.

If for every node $abs|height(LST) - height(RST)| <= 1$

#Claim: In Balanced BT, height $= \log_2^N$ Proof later?

Balanced Binary Search Tree: $H = \log_2^N$ # N = number of nodes

Operations in BST

Search() / Insert() / Floor() / Ceil() .. | TC : $O(H)$

#Con;

If BST is Balanced $H = \log_2^N$

Search() / Insert() / Floor() / Ceil() .. | TC : $O(\log_2^N)$

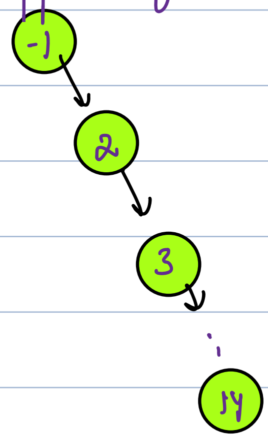
#Note: As we insert in BBST, Balance factor can be effected,

It can be maintained using AVL rotations

48 Given an sorted arr[], create a BBST & return it's head node

	0	1	2	3	4	5	6	7	8	9
Ex arr[] = {	-1	2	3	4	6	7	8	10	13	14 }

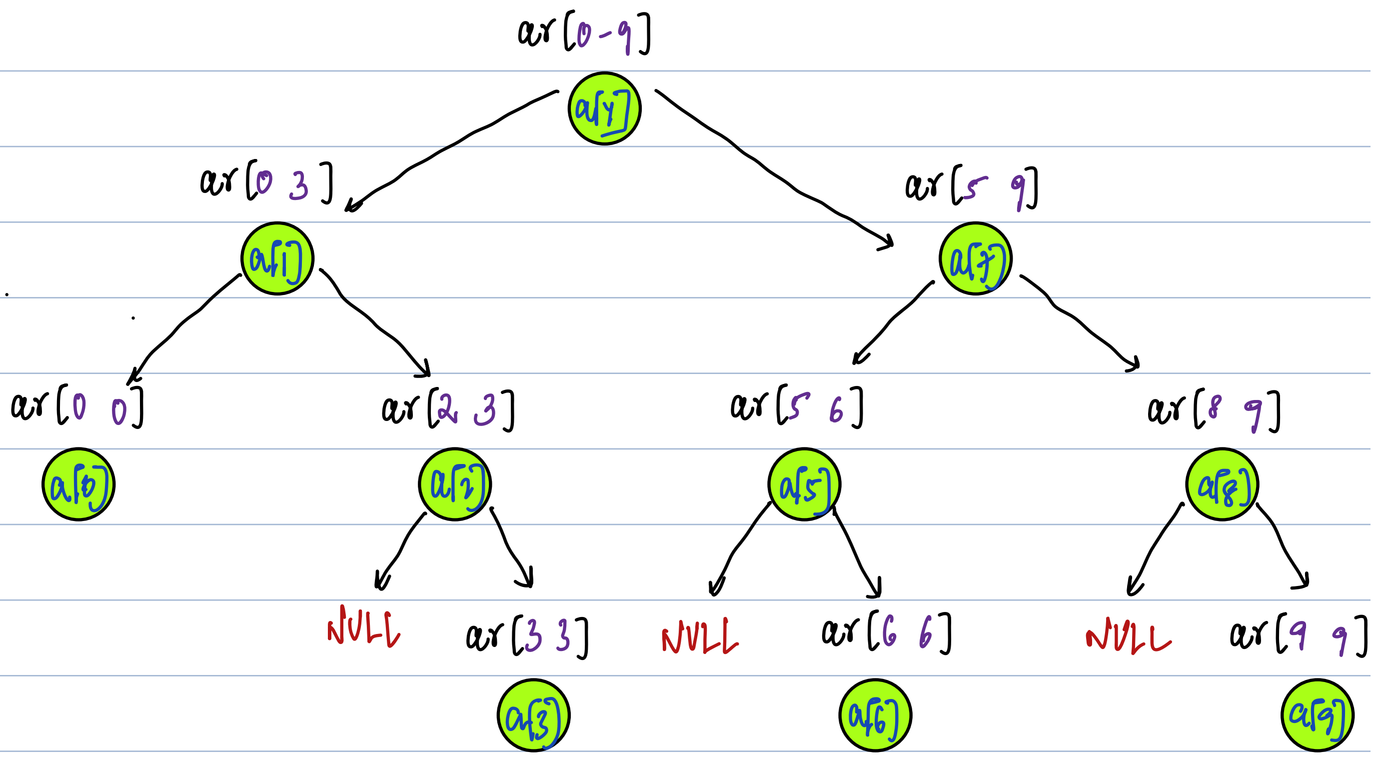
Idea1: Approach of Inserting node by node will create skewed Trees ✗



Idea2:

	0	1	2	3	4	5	6	7	8	9
arr[] = {	-1	2	3	4	$\boxed{6}$	7	8	10	13	14 }

2:10



```
Node* Solve(int arr[], int N){
    Node *root = createBBST(arr, 0, N-1);
    return root;
}
```

Ass: Create BBST from arr[s..e] & return root node of BBST

```
Node* createBBST(int arr[], int s, int e){ TC: O(N)
    if(s>e){ return nullptr; }
    int m = (s+e)/2;
    Node *root = new Node(arr[m]); #
    root->left = createBBST(arr, s, m-1);
    root->right = createBBST(arr, m+1, e);
}
```