Day - 45

(Binary Tree & Binary search Tree)

problem 1: Construct tree from Inorder & preorder

Inorder : 1 3 3 2 4 7 9
LNR
left | right

preorder : 4 3 1 2 3 7 9
NLR
Node

4

start end start end
{1 3 3 2} 4 {7 9}
↑
present in left

4
/
start/end start end 3
{1} 3 {3, 2}
↑
present in left

4
/
3
/
1

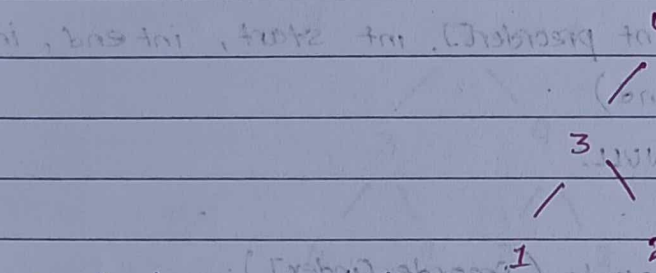
start/end start/end
{{}} {{}} 3
left right

All NULL

return.

8 present in right of 3

{ 1 } 3 { 8, 2 }



start/end

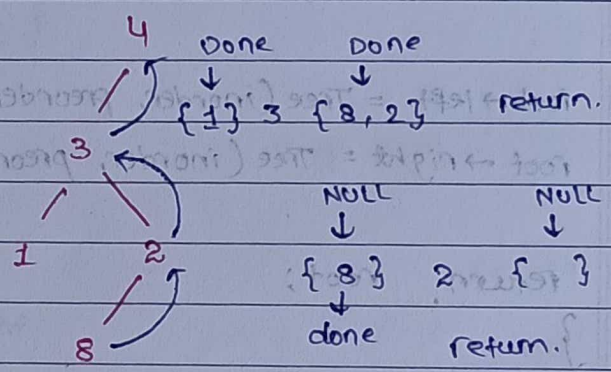
start/end.

{ 8 } 2 { }

left

right

8 present in left of 2



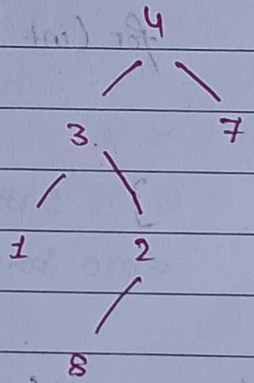
NULL NULL

{ } 8 { }

return.

4 { 7, 9 }

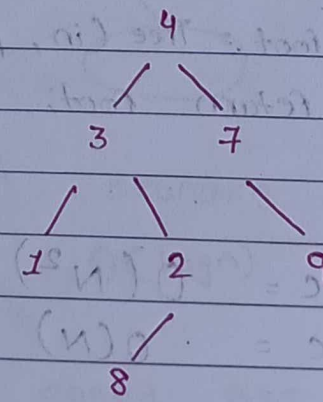
present right side.



NULL start/end

{ } 7 { 9 }

present right side of 7.



required tree:

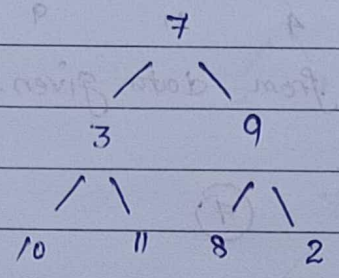
Code:

```
Node* Tree(int inorder[], int preorder[], int start, int end, int index){
        if (start > end)
            return NULL;

    Node* root = new Node(preorder[index]);
    int i = find(inorder, preorder[index], start, end);

root->left = Tree(inorder, preorder, start, i-1, index+1);
root->right = Tree(inorder, preorder, i+1, end, index+(i-start)+1);

return root;
}

int find(int inorder[], int num, int start, int end){
        for(int i = start; i<=end; i++){
            if(inorder[i] == num)
                return i;
        }
}

Node* buildTree(int in[], int pre[], int n){
        Node* root;
        root = Tree(in, pre, 0, n-1, 0);
        return root;
}
```

T.C = $O(N^2)$

S.C = $O(N)$

Binary Search Tree (BST)



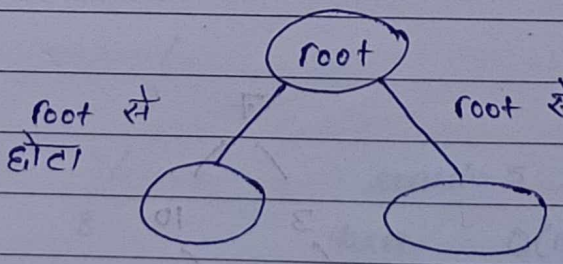
How we search any element ?
(search 2)

root
↓
root → left
↓
root → right

By this we can traverse whole tree.

Time Complexity : O(N)

But, we want to search data faster :
we store the data in sorted order.



इस तरह से searching average case में O(logn) ही होगा।

This type of tree is called BST (Binary search Tree).

Let's understand With an Example:

7 3 10 8 2 4 11 9 1

Construct a BST from data given.

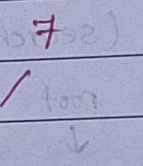
7 → 7

left = smaller than 7
right = larger than 7

3 →

3 < 7

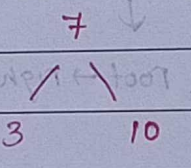
left of 7



10 →

10 > 7

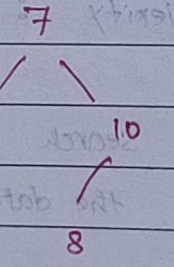
right of 7



8 →

8 > 7

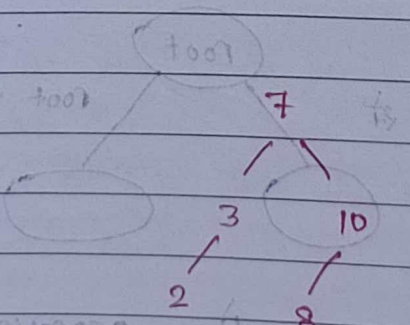
8 < 10



2 →

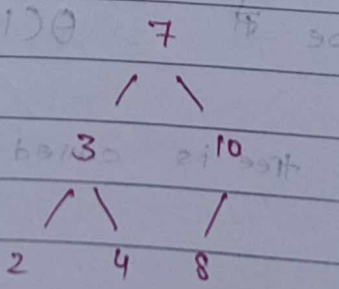
2 < 7

2 < 3



4 →

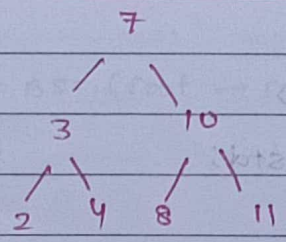
4 > 3



11 →

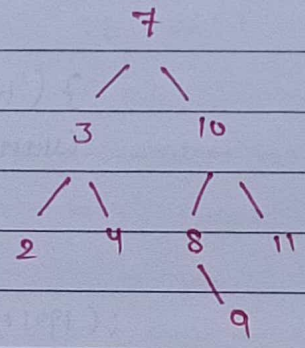
11 > 7

11 > 10



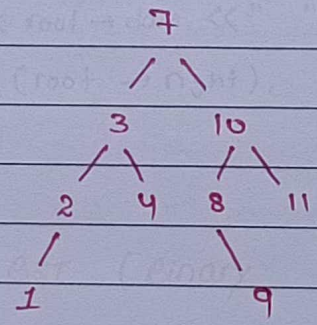
9 →

9 > 8



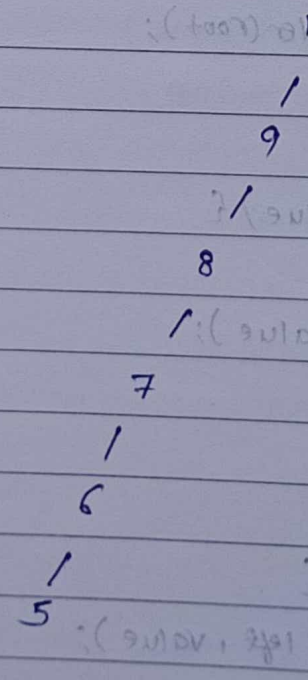
1 →

1 < 2



On Average case searching will take logn time.

For Worst case O(N)



search 5 takes O(N) time.

Code :

```
#include <iostream>
using namespace std;

class Node {
  public:
    int data;
    Node * left;
    Node * right;

    Node (int x) {
      data = x;
      left = NULL;
      right = NULL;
    }
}
int main () {
  int arr[10] = { 10, 13, 4, 8, 11, 19, 2, 7, 18, 23};
              Node * root = NULL;
      for (int i = 0;  i < 10;  i++) {
              root = BST (root, arr [i];
              ~~return 0;~~  inorder (root);
            } return 0;
  }
Node * BST ( Node * root,  int value) {
        if (root == NULL) {
            root = new Node (value);
            return root;
          }

        if (root -> data > value) {
        root -> left = BST (root -> left , value);
            return root;
```

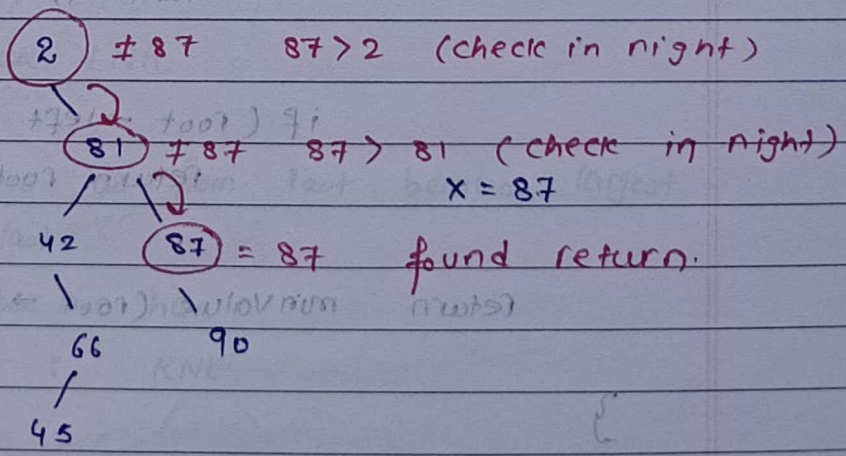
```
else {
    root->right = BST(root->right, value);
    return root;
}
```

```
Void inorder(Node * root) {
    if (root == NULL)
        return;

    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}
```

Problems on BST (Binary search Tree).

problem 1 : Search a Node in BST.

Example :



Code :

```
bool search (Node *root, int x) {
    if (root == NULL)
        return 0;

    if (root->data == x)
        return 1;

    if (root->data > x)
    return search (root->left, x);

    if (root->data < x)
    return search (root->right, x);
}
```

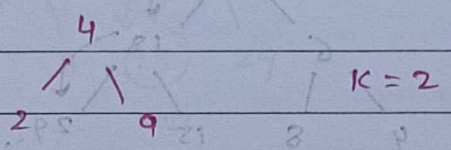
problem 2 : Minimum element in BST

Code:

```
int minValue (Node * root) {
    if (root == NULL)
        return -1;

    if (root->left == NULL)
        return root->data;

    return minValue (root->left);
}
```

problem 3: K^{th} largest Element in BST:

Example:



Ans = 4

Approach 1:

store the tree in an array

arr = [2, 4, 9];

K = 2

find total Node -(2)+1

and return.

But here an Extra space for array is required.

Approach 2:

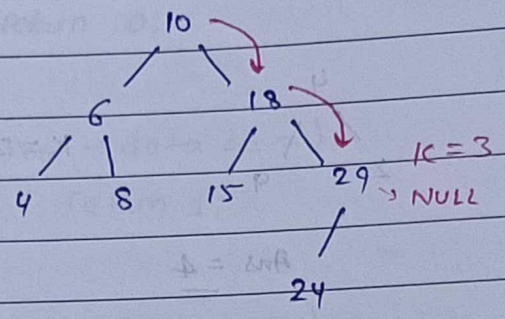
Inorder: ~~LR~~ LNR

But we have to do from last because largest present at last.

So, we use here

Inorder: RNL

Example:



4th largest

RNL

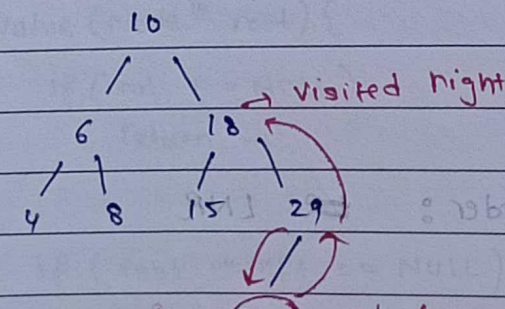
K = 4

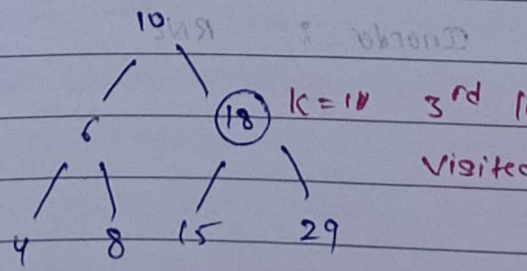
traverse to right until found NULL

1 st larget = 29

k-- => k = 3

then it go to left.





then visit left.

Problem 4 : Check for BST

Approach 1 :

Traverse using inorder

store in array

if sorted in ascending order

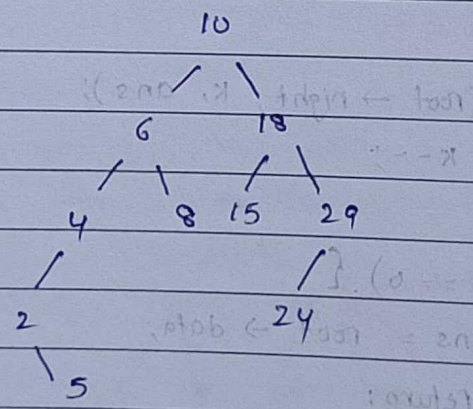
↳ thus it is BST

otherwise

↳ not BST.

In, this Extra space required.

Approach 2 :

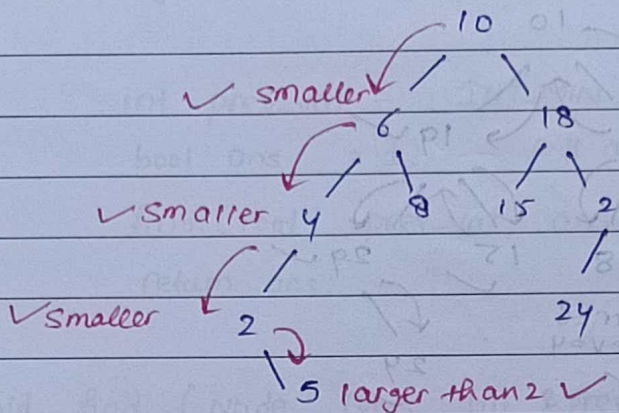


Traverse

↳ if we go left check element is smallest

↳ if we go right check element is larger.

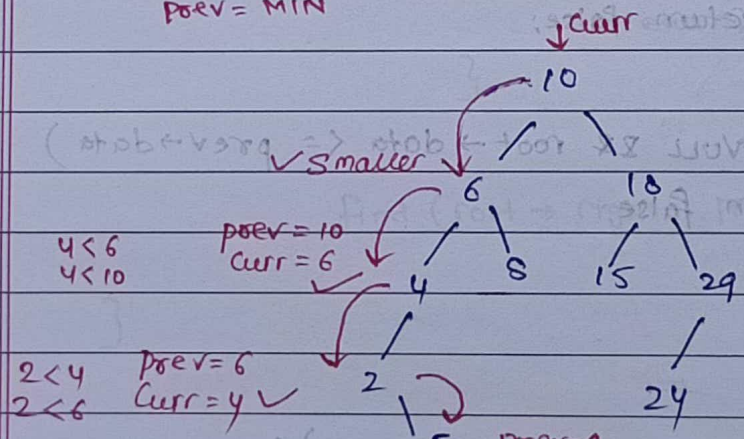
Example:



But this is not BST
because it also less than 4
But here it is greater.

This approach also fails

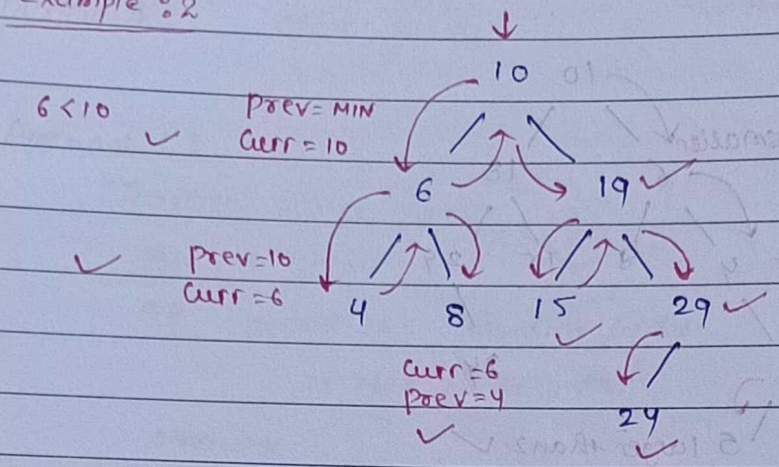
In next Approach we also check with prev also.



In this Example, 5 is greater than 4
but in BST it is not possible.

So, it is not BST.

Example : 2



Code :

```
bool isValidBSTUtil (Node* root, Node * & prev){
        if (root == nullptr)
            return true;

        if (!isValidBSTUtil (root→left, prev))
            return false;

        if ( prev != NULL && root→data <= prev→data )
            return false;

        prev = root;

        return isValidBSTUtil (root→right, prev);
}

bool isBST( Node * root){
        Node * prev = NULL;

        return isValidBSTUtil (root, prev);
}
```