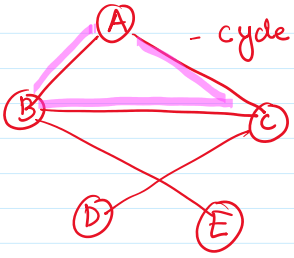
8) Lecture 15-19 Tree, Their Traversal Algorithms, BST

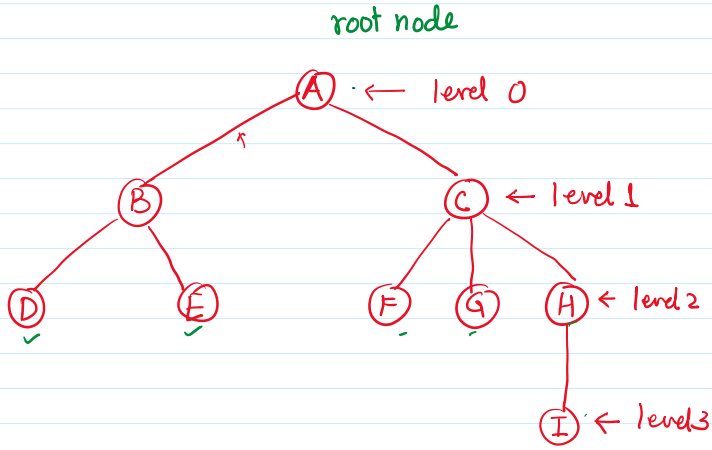
27 August 2024 18:36

$G = \{V, E\}$

$V = \{A, B, C, D, E\}$

$E = \{(A,B), (C,D), (B,E), (A,C), (B,C)\}$





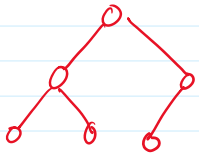
A - root node
B is parent of D & E
A is ancestor of D, E
D & E are descendent of A
C is sibling of B

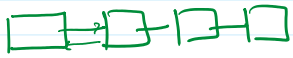
Height of tree is 3.

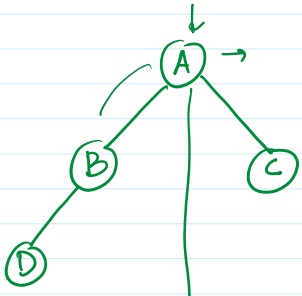
D, E, F, G, I leaf node

B, C, H, A are non-leaf (Internal node)

Binary tree in which a node is either leaf or can have atmost 2 children

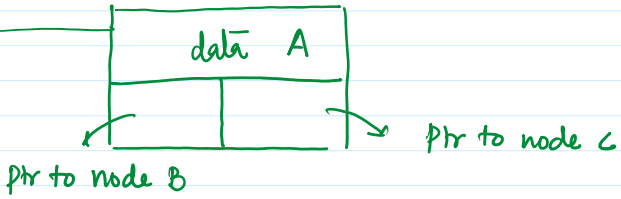






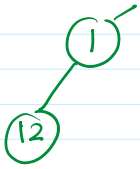
- Insert (key, val)
- Search (key)
- Delete (key)

```
struct node
{
int data;
struct node *left;
struct node *right;
};
```

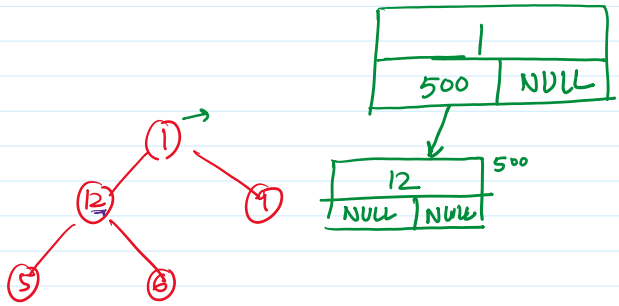



```
#include <stdio.h>
#include <stdlib.h>
```

```c
struct node {
  int item;
  struct node* left;
  struct node* right;
};
```



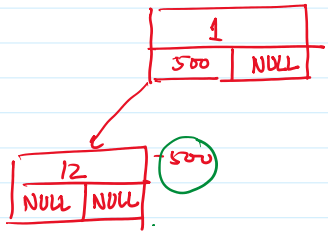

```c
// Create a new Node
struct node* createNode(value) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;
  return newNode;
}
```



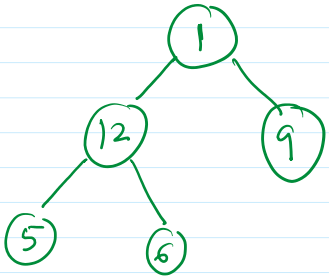

```c
// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
  root->left = createNode(value);
  return root->left;
}
```

ptr to node 1

```c
// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
  root->right = createNode(value);
  return root->right;
}
```



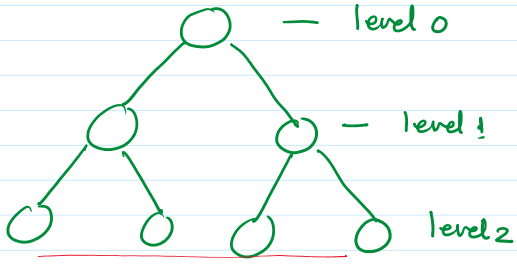

```c
int main() {
  struct node* root = createNode(1);
  insertLeft(root, 12);
  insertRight(root, 9);

  insertLeft(root->left, 5);
  insertRight(root->left, 6);

}
```



Complete Binary tree · Each internal node will have exactly two children & all the leaves should be at the same level

level i has 2^i nodes

Let h be the height of tree



# of leaves is 2^h —(I)

# of internal node $2^0 + 2^1 + 2^2 + \cdots 2^{h-1} = 2^h - 1$ —(II)

$\Rightarrow$ **# of internal node = # of leaves − 1**

# of leaves = # of internal node + 1

total # nodes = n

$\Rightarrow$ total # of nodes $= 2^h + 2^h - 1 = 2^{h+1} - 1$

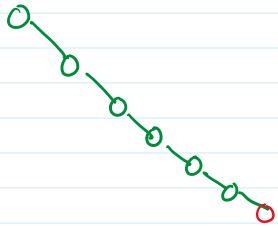
$\Rightarrow$ total # of nodes $= 2^h + 2^h - 1 = 2^{h+1} - 1$

$n = 2^{h+1} - 1$

$\Rightarrow \boxed{h = \log_2 (n+1) - 1}$

Binary Tree

Binary tree of n nodes



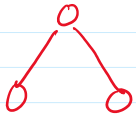
Max height of Binary tree $n-1$

$\log_2 (n+1) - 1 \leq h \leq n-1$

Th: For a binary tree: # of leaves $\leq$ 1 + # of internal nodes — (I)

Proof: Proof by induction on # of internal nodes.

Base Case when # of internal node = 1. This is possible only in the following two scenarios:



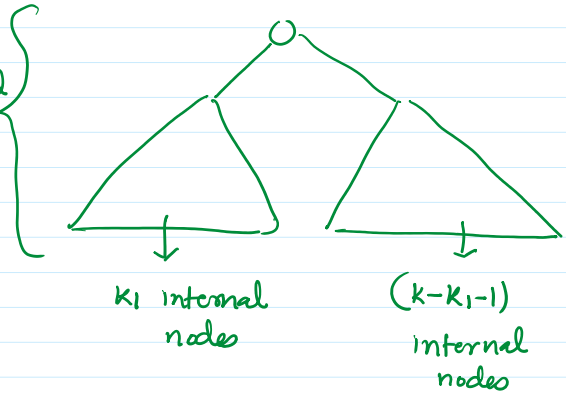
Eqn (1) holds true for both the cases.

Inductive Case:

Suppose stmt is true for $(k-1)$ internal nodes.

Total no. of leaves $\leq k-1+1 = k$

Now we need to prove for tree with k internal nodes



# of leaves in left subtree $\leq k_1+1$

" " right " $\leq k-k_1-1+1$

$= k-k_1$

# of leaves $\leq (k_1+1)+(k-k_1)$

$= k+1$

Th For a binary tree with n nodes

$1 \leq$ # of leaves $\leq \frac{n+1}{2}$

Proof

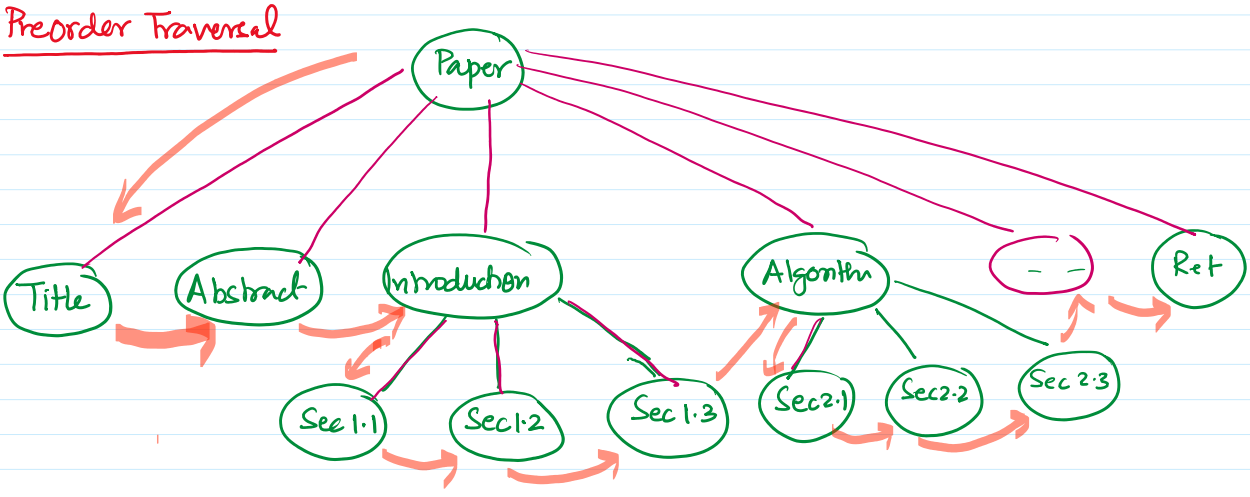
# of leaves + # of internal node $= n$ — ①

# of leaf $\leq 1+$ # of internal nodes — ②

①&② $\Rightarrow$ # of leaves $\leq \frac{n+1}{2}$

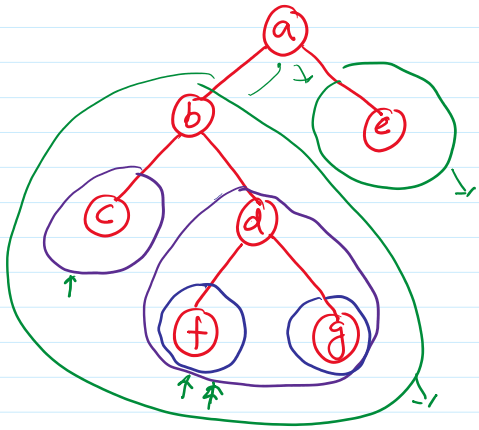
Tree Traversal: Is a way to visit all nodes of tree in a specified order.

Preorder Traversal



Root → left Subtree → Right Subtree

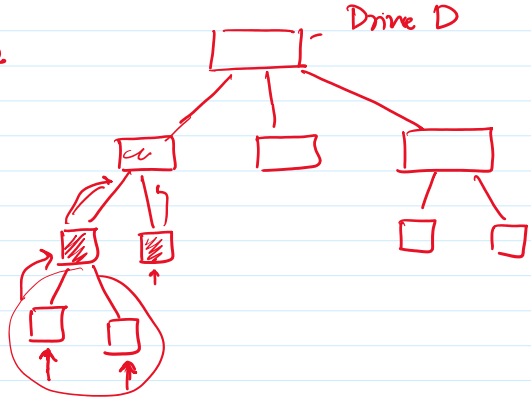
a b c d f g e



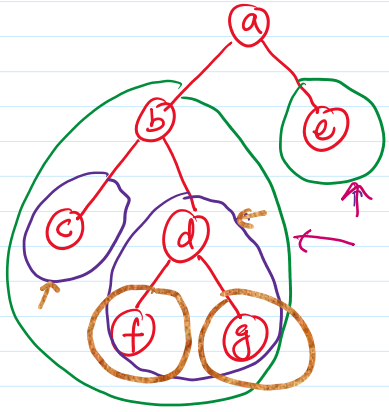

```
preorder ( v)
{  if (v == NULL)
      then return
   else
         visit(v)          | print v→key
         preorder ( v→left)
         preorder ( v→ right)
}
```

Postorder Traversal

left subtree → right subtree → root node



c f g d b e a



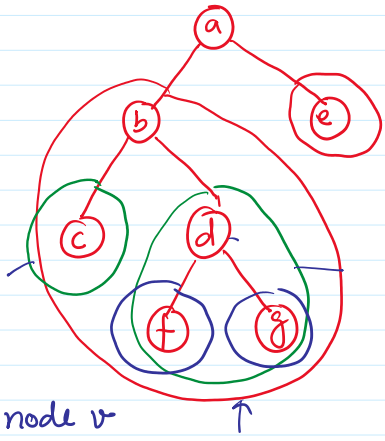

```
postorder (v)
{  if (v == NULL)
      return
```

```
else
        postorder(v→left)
        postorder(v→right)
        visit(v)        /* print v→key
}
```

Inorder Traversal

left subtree → root → right subtree

c b f d g a e




```
inorder(v)
{   if (v == NULL)
      then return
    else
        inorder(v→left)
        visit(v)        /* Printing key value of node v
        inorder(v→right)
}
```

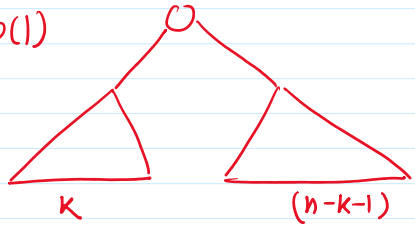
Time complexity

Since inorder traversal visit each node ⇒ running time $\Omega(n)$ —(i)

Upper bound: T(n) : worst case running time of inorder traversal

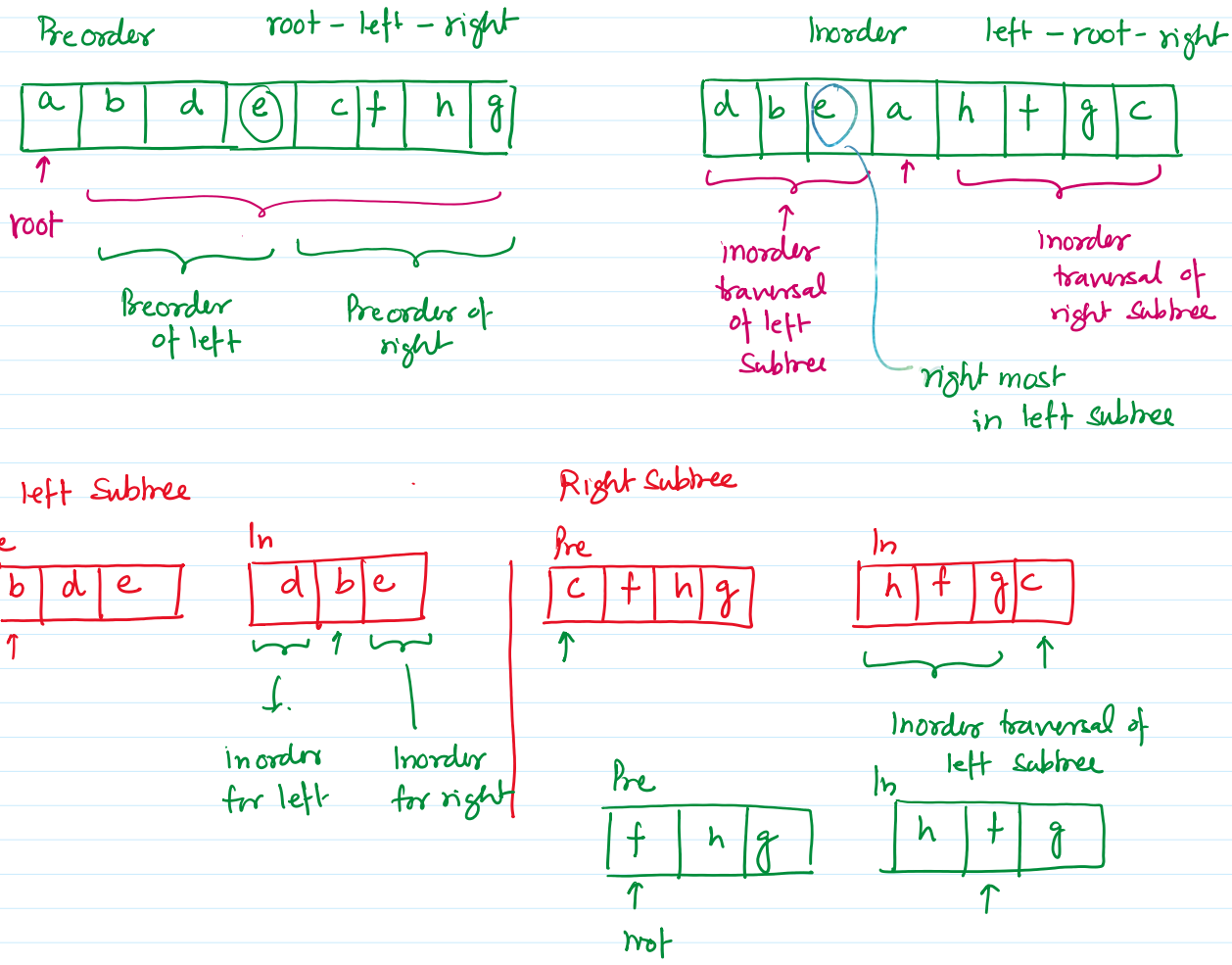
$$T(n) = T(k) + T(n-k-1) + O(1)$$

$$\Rightarrow T(n) = O(n) \quad \text{—(ii)}$$



(i) & (ii) ⇒ $T(n) = \Theta(n)$.

Buiding tree from pre & inorder traversal

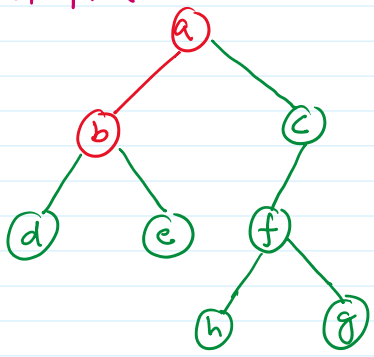


Pseudocode

① Take the first element of pre order traversal - root of tree

② Search for that element in inorder traversal
- splits into inorder traversal of left/right subtree

③ Count # of elements in left subtree, inorder traversal of



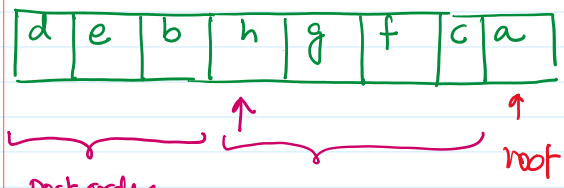
④ Split the preorder traversal after those many elements
- Split preorder traversal into preorder traversal of left & right subtree

⑤ Recurse on left & right subtree

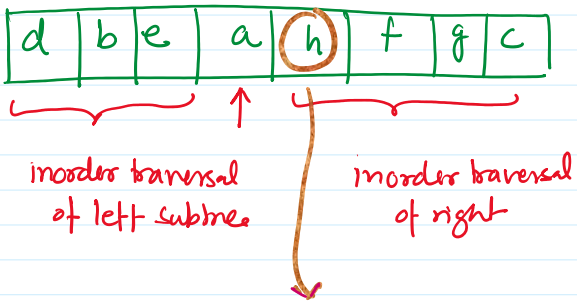
Building tree from post & inorder traversal

Postorder (left → right - root) Inorder (left - root - right)

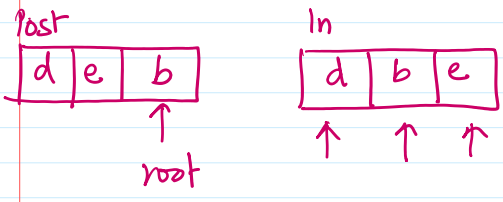
Postorder (left → right - root)



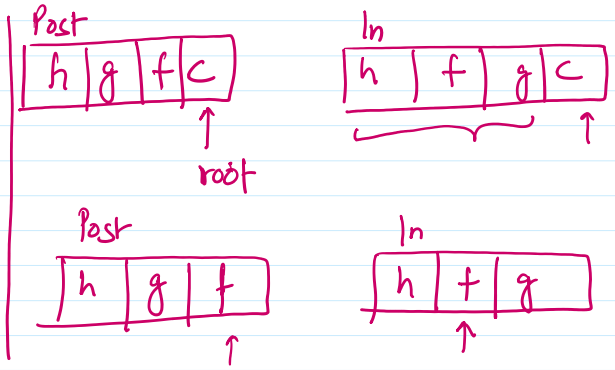
Inorder (left - root - right)

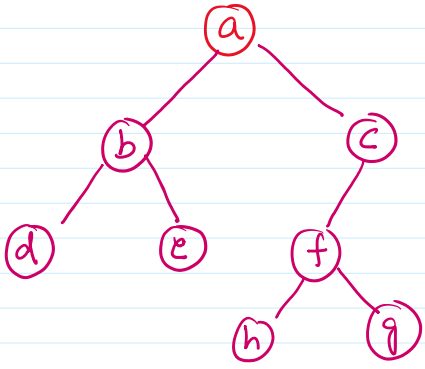


left Subtree



right Subtree





Buiding tree from post order & pre order traversal

Post order

(left - right - root)

Pre order

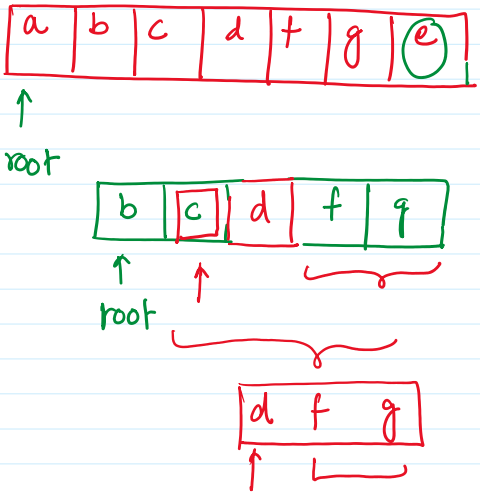
(root → left → right)

Post c b a

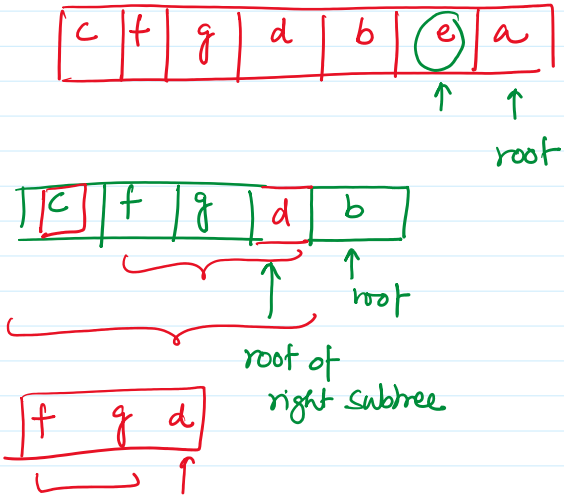
Pre a b c

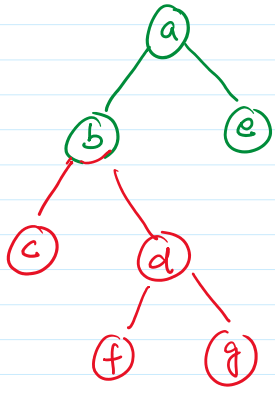
It each internal node should have exactly two children then tree can be uniquely determined by post + pre order traversal.

Preorder Root left right



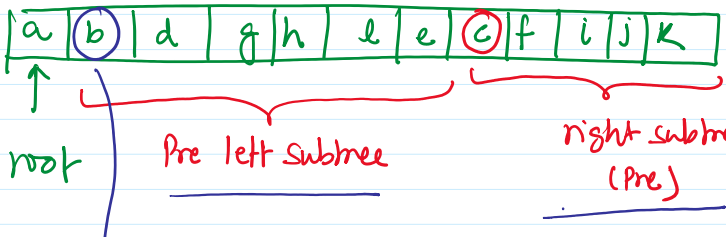
Postorder left right root



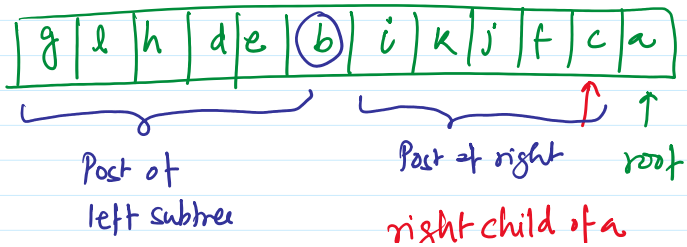


Preorder Root left right

Pre



Post



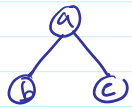
root | Pre left Subtree | (Pre) | Post of left subtree | Post of right | root

left child of a | right child of a

① How to prepare tree data structure

② Traversal algorithm

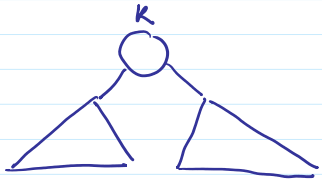
③ Constructing tree from traversals



a
b c

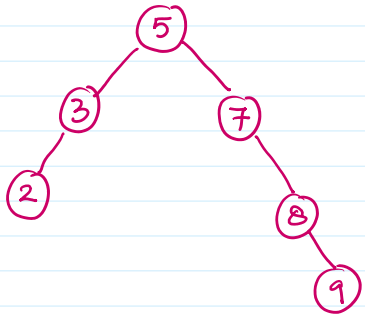
Binary Search tree (BST)

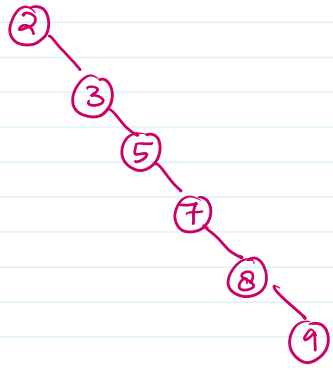
- Key value in integer
- Each internal node v stores (K, val)
  - All nodes in left subtree of v have value at most K



  - All nodes in right subtree of v have values at least K

2, 3, 5, 7, 8, 9





① Search ② Insertion ③ Deletion

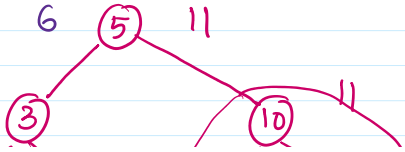
① Searching in BST

Input - ptr to root node, key := K

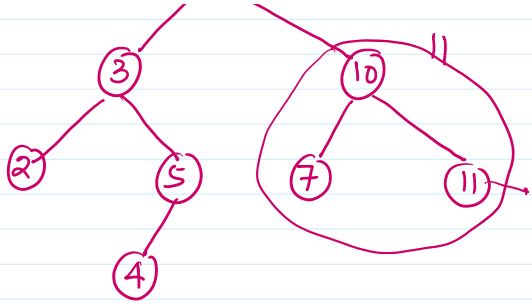
Output - Yes/No if K is present in BST

Search for 11

Search for 6



Search for 6




```
Search (root, k)
    x ← root
    while (k ≠ x→key) and (x ≠ NIL)
        { if k < x→key
            then x ← x→left
            else x ← x→right
        }
    return x
```

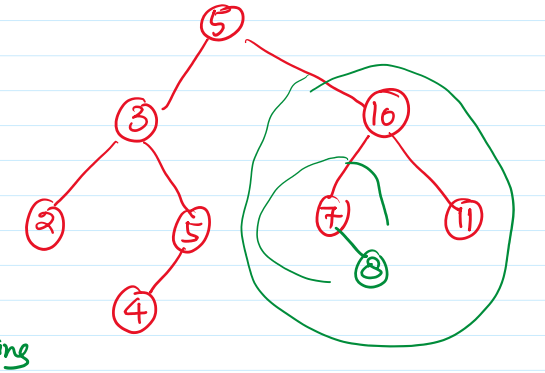
time complexity O(h)

h := height of BST.

n nodes in BST

② Inserting in BST

Insert 8



- Insertion is done at leaf level
- Create a node z (left/right ptr are NIL
  key ← k)
- Find the place where z belong (as if searching for z)
- add z

Running time O(h)

③ Finding minimum of BST

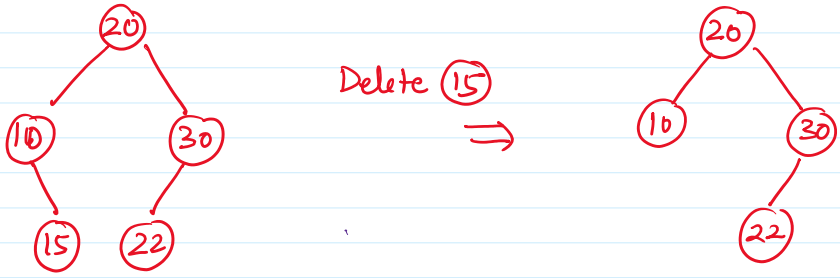
```
Tree-min (root)
    { x ← root
      while (x→left ≠ NIL)
          x ← x→left
      return x
    }
```

Time complexity O(h)

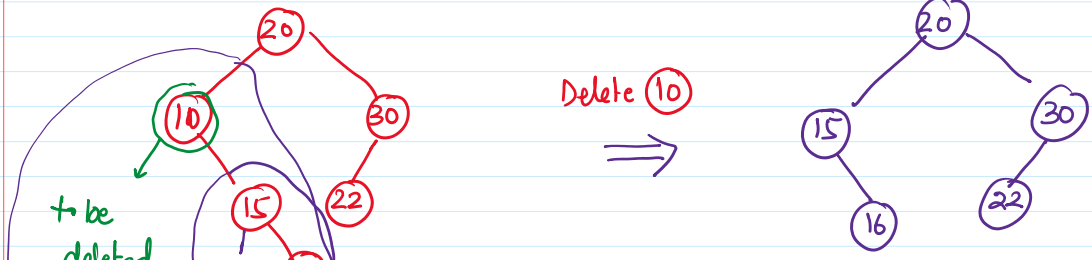
④ Deletion in BST

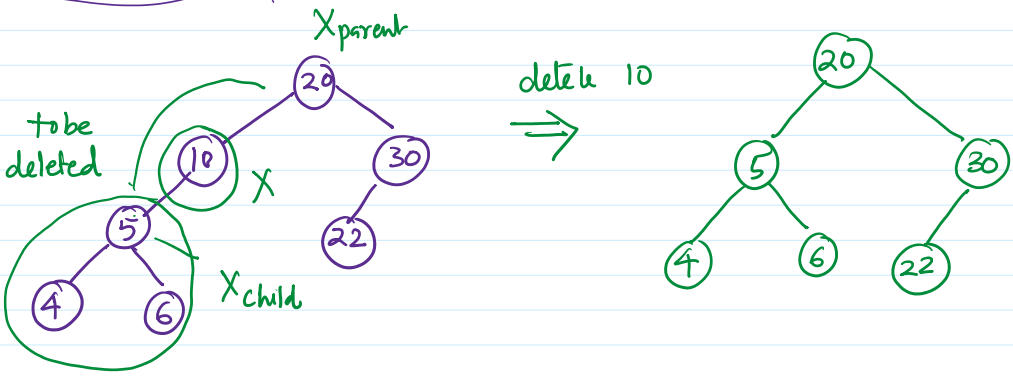
x is node to be deleted

Case (i) x is leaf node

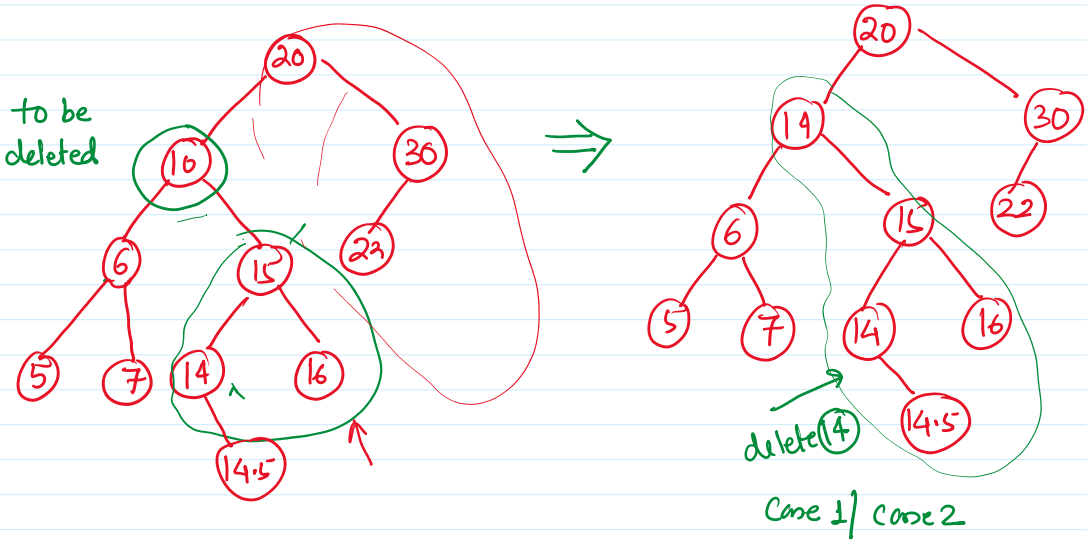


Case (ii) x has only one child.





Case (iii) x has both left & right child



key of node to be deleted

Delete (root, x)

{

1. Search for x

2. If x is leaf, delete x & its connecting edges

3. If x has exactly one child := X_{child}

and x is left (right) child of X_{parent}

Then delete x and its corresponding edges

Make X_{child} as left (right) child of X_{parent}

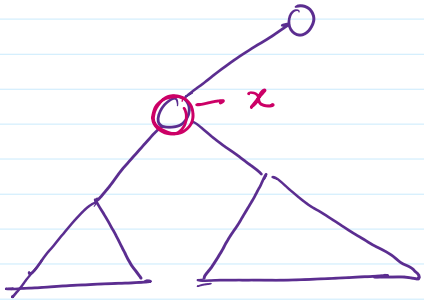
4. If x has both left & right child.

└ find sucessor of x := smallest element in right subtree of x (say x')

- delete (x, x')
- copy x' to x

Problem Finding successor of a given node.

Given a node x find the node with the smallest key greater than x.

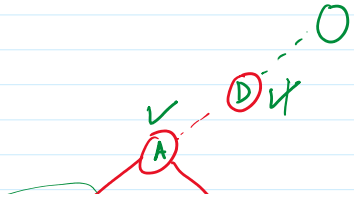


Case (i) - Right subtree of x is non-empty

- Successor is leftmost node in rightsubtree
- computed by calling Tree-min(x→right)

Case (ii) Right subtree of x is empty

Sucessor of x is lowest ancestor of x whose



Successor of x is lowest ancestor of x whose left child is also ancestor of x.

$y := x \rightarrow parent$

while $y \neq NIL$ and $x = y \rightarrow right$

$x \leftarrow y$

$y \leftarrow (y \rightarrow parent)$

return y

Time complexity $O(h)$