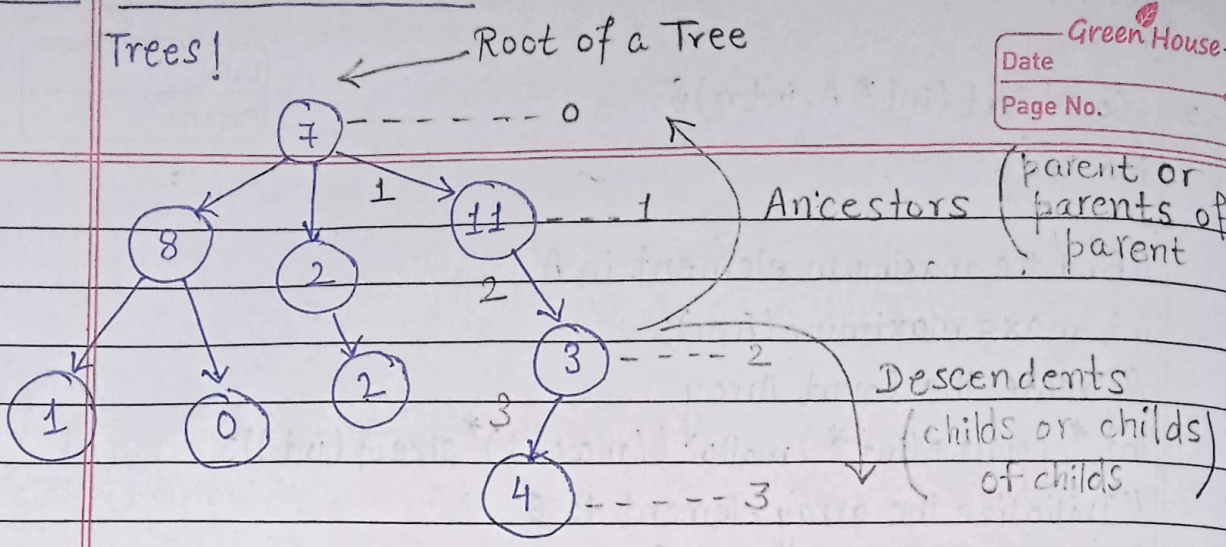
lec-61 Introduction to Trees:

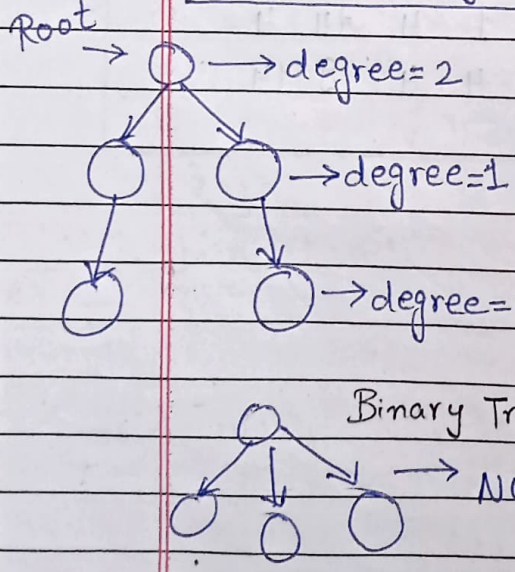
Trees!



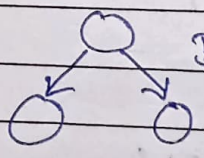
Common Terminologies:

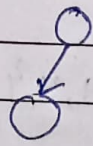
1. Root → Topmost Node
2. Parent → Node which connects to the child
3. child → Node which is connected by another Node is it's child.
4. Leaf/External Node: Nodes with no children.
5. Internal Node: → Node with atleast one child.
6. Depth → No. of edges from root to the Node.
7. Height → No. of edges from node to the depest leaf!
8. Sibling → Nodes belonging to the same parent!
9. Ancestors/Descendents

lec-62 What is a Binary Tree?



→ Binary Tree is a tree, which has atmost 2 children for all the Nodes.



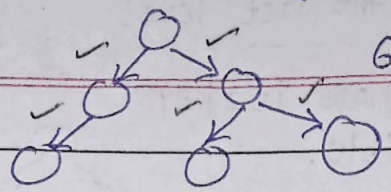


→ Yes, It is Binary Tree

Quick Revision: ① Tree is made up of nodes & Edges!

② n nodes → n-1 edges

Green House
Date
Page No.



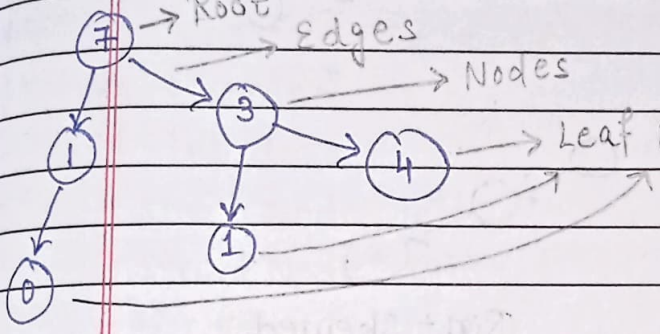
③ Dedgee?

Degree → No. of Direct children. (for a Node)

④ Degree of a Tree is the highest degree of a Node among all the nodes present in the Tree.

⑤ Binary Tree → Tree of degree ≤ 2

Nodes can have 0, 1 or 2 children.



Tree → Non-linear Data Structure.

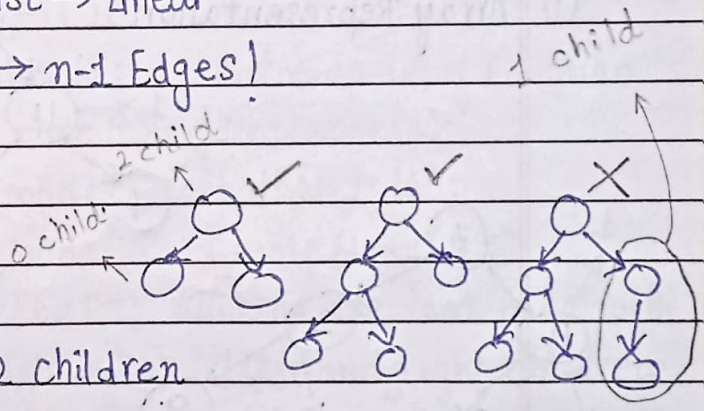
Lec-63 Types of Binary Trees:

Imp Tree → Non Linear → Ideal for represending hierarchical Data

Array, Stack, Queue, LinkedList → Linear

In a Tree with n Nodes → n-1 Edges!

Types of Binary Trees:

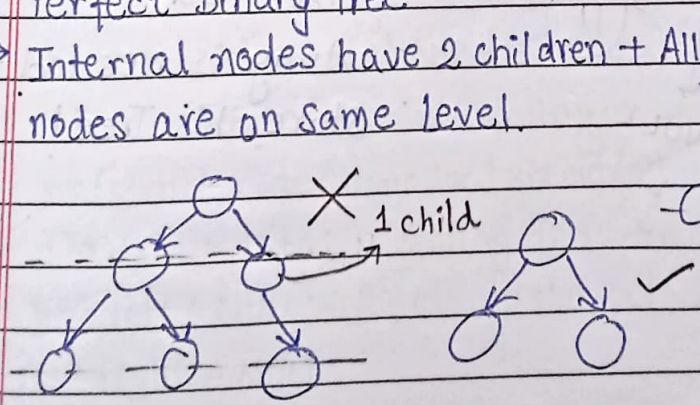


① Full or Strict Binary Tree

→ All Nodes have either 0 or 2 children

② Perfect Binary Tree

→ Internal nodes have 2 children + All leaf nodes are on same level.

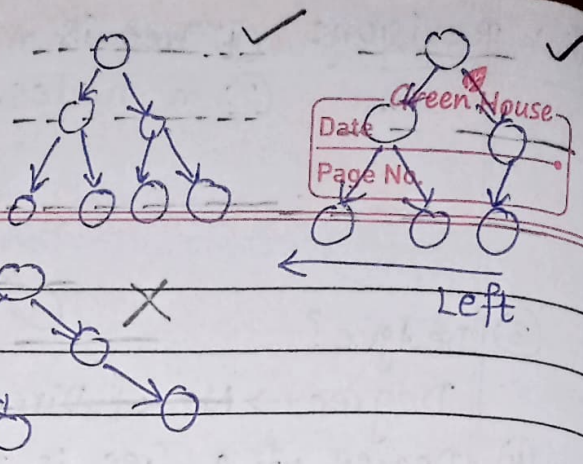


③ Complete Binary Tree

→ All levels are completely filled except possibly the last level

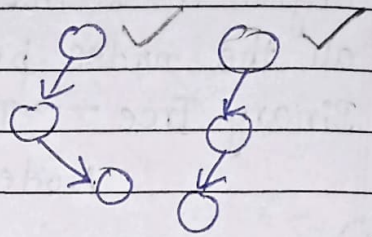
+

Last level must have it's keys as left as possible.

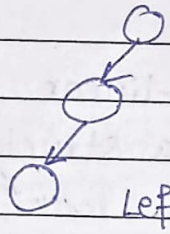


④ Degenerate Tree:

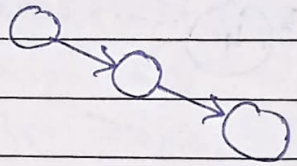
→ Every Parent Node has exactly one child



⑤ Skewed Binary Trees:



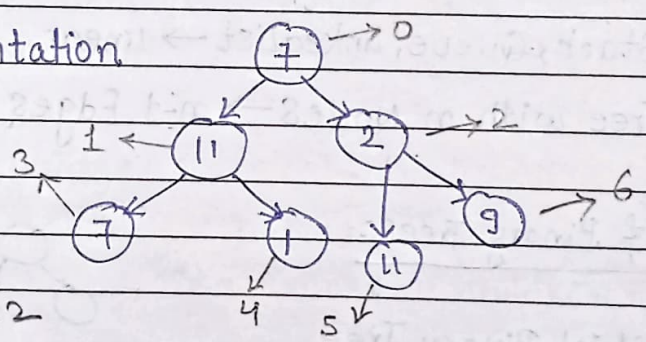
Left Skewed

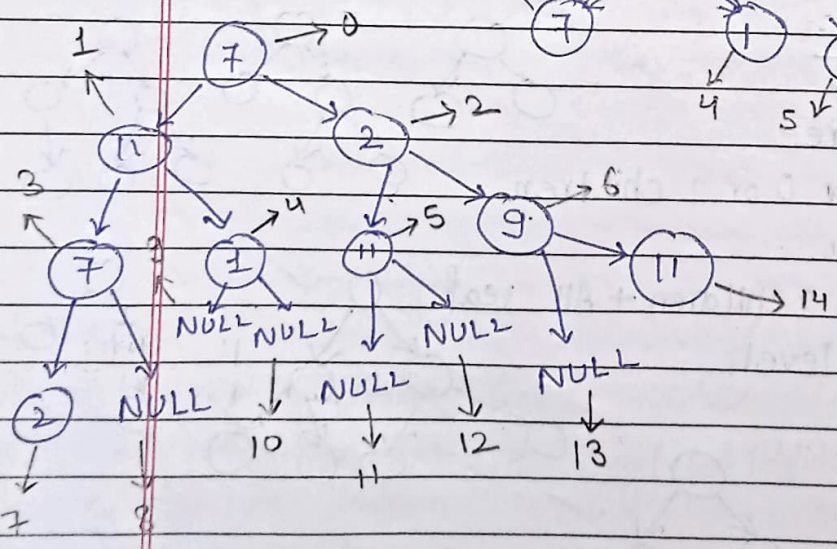


Right Skewed

Lec-64 Representation of a Binary Tree:

① Array Representation

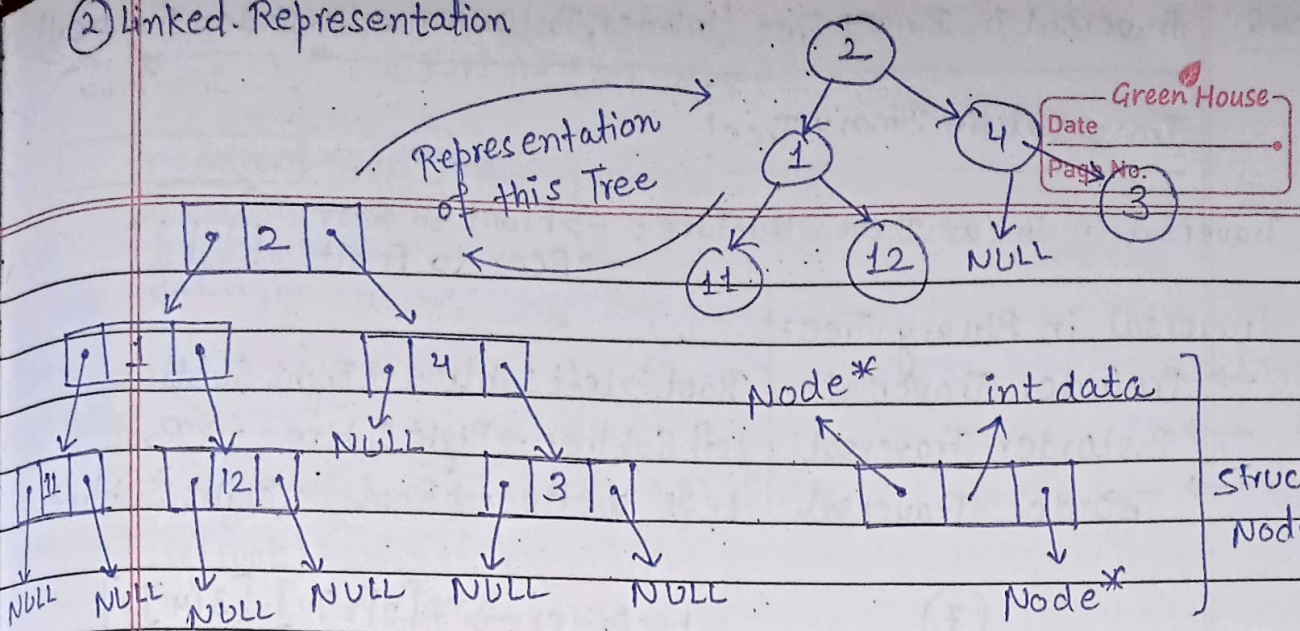




Size → 15

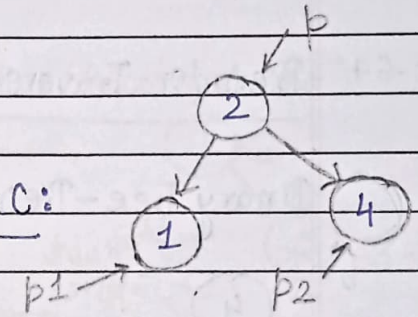
Array is Needed to store this Tree!

(2) Linked Representation




```
Struct Node{
  int data;
Struct Node* left;
Struct Node* right;
};
```

Lec-65 Linked Representation of Binary Tree in C:




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

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

struct node* createNode (int data){
  Struct node *n; // Creating a node pointer
n=(struct node*)malloc(sizeof(struct node)); // Allocating memory in heap
n->data=data; // Setting the data
n->left=NULL; // Setting the left and right children to NULL
n->right=NULL;
return n; // Finally returning the created node
}

int main(){
//Constructing the root node-using Function
Struct node *P=createNode(2);
Struct node* P1=createNode(1);
Struct node* P2=createNode(4);
 p->left=P1; //Linking the root node with
 P->right=P2; // Left and right children
 return 0; }
```

Lec-66 Traversal in Binary Tree (InOrder, PostOrder & PreOrder Traversal)

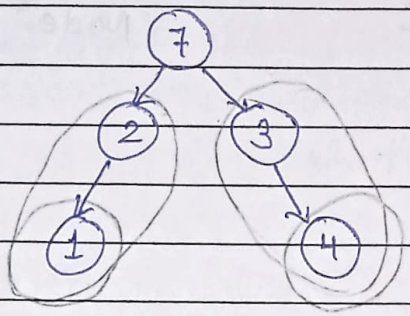
Traversal in Binary Tree!

Green House
Date
Page No.

Traversal in Linear Data Structure: → Front to Rear
→ Rear to front

Imp Traversal in Binary Tree:

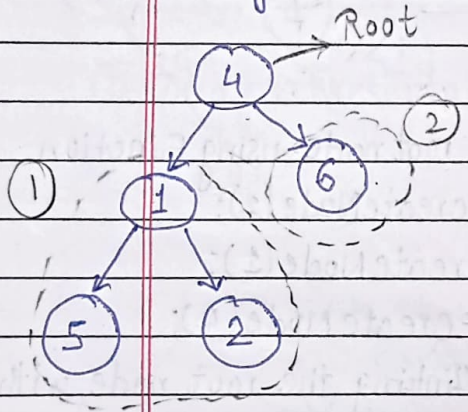
→ PreOrder Traversal — Root → Left Subtree → Right Subtree
→ Postorder Traversal — Left Subtree → Right Subtree → Root
→ InOrder Traversal — Left Subtree → Root → Right Subtree



preOrder → 7 [2[1]] [3[4]]
→ 7 2 1 3 4

Lec-67 PreOrder Traversal in a Binary Tree (with C Code):

Binary Tree - PreOrder Traversal



Root → Left Subtree → Right Subtree

4 → [1 5 2] → 6
4 1 5 2 6

```
void preorder(struct node *root){
  if(root) = NULL){
    printf("%d", root->data);
    preorder(root->left);
    preorder(root->right);
  }
}
```

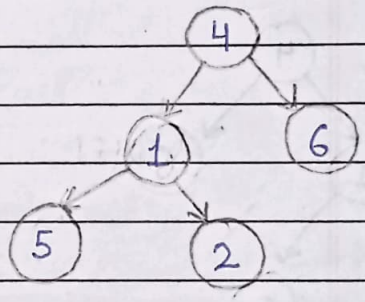
C-Code:

```
#include<stdio.h>
#include<stdlib.h>
struct node{
    int data;
    struct node *left;
    struct node *right;
```

```
};
struct node* createNode (int data){
    struct node *n;
    n=(struct node*)malloc(sizeof(struct node));
    n->data = data;
    n->left = NULL;
    n->right = NULL;
    return n;
}
```

4 1 5 2 6

```
void preOrder(struct node* root){
    if(root!=NULL){
        printf("%d ", root->data);
        preorder(root->left);
        preOrder(root->right);
    }
}
int main(){
    //Constructing the root node-using Function (Recommended)
    struct node *p = createNode(4);
    struct node* p1 = createNode(1);
    struct node* p2 = createNode(6);
    struct node* p3 = createNode(5);
    struct node* p4 = createNode(2);

    //Linking the root node with left and right children
    p->left = P1;
    p->right = P2;
    p1->left = P3;
    p1->right = p4;

    PreOrder(p);
    return 0;
}
```

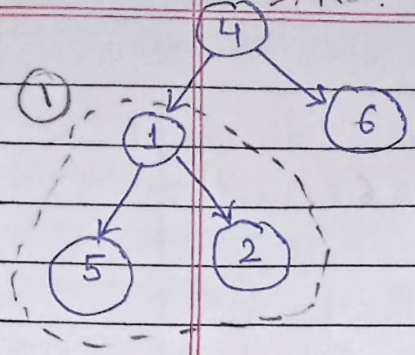


Lec-68 Postorder Traversal in a Binary Tree (with C Code):

Green House
Date
Page No.

Binary Tree – Postorder Traversal:

Left Subtree → Right Subtree → Root



(5 2 1) 6 4

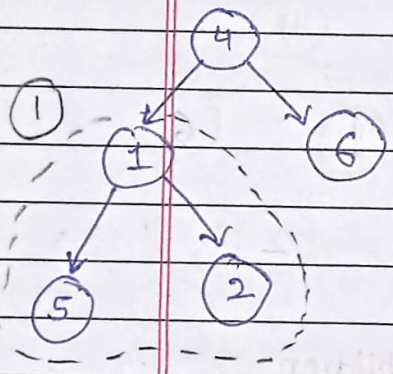
→ 5 2 1 6 4

```
void PostOrder(struct node* root){
    if(root!=NULL){
        postorder(root->left);
        postorder(root->right);
        printf("%d", root->data);
    }
}
```

Lec-69 InOrder Traversal in a Binary Tree (with C Code):

Binary Tree – InOrder Traversal:

Left Subtree → Root → Right Subtree



[5 1 2] 4 6

→ 5 1 2 4 6

```
void InOrder(struct node *root){
    if(root!=NULL){
        InOrder(root->left);
        printf("%d", root->data);
        InOrder(root->right);
    }
}
```

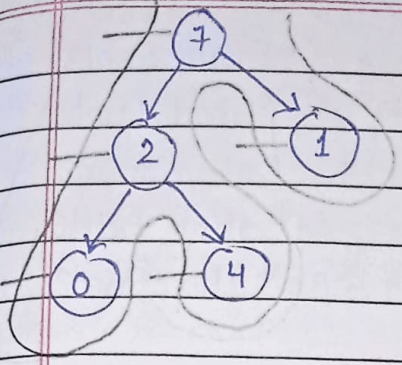
5 1 2 4 6

In(5) → print(5)
In(1) → print(1)
In(4) → Root → Right 2 4 6

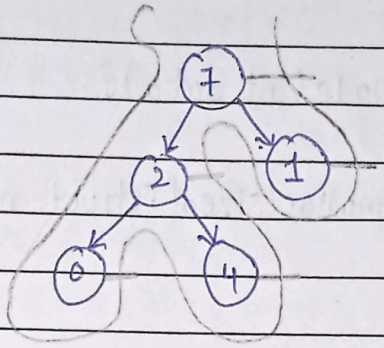
Lec-70 *Best* Trick to Find PreOrder, InOrder & PostOrder Traversal:

→ Trick to find InOrder, PreOrder, & PostOrder Traversal:

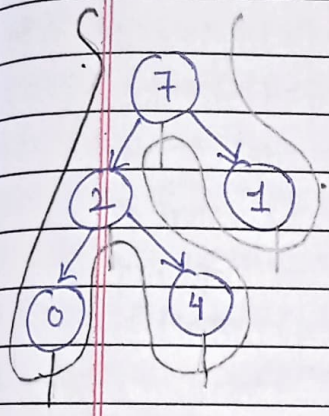
Green House
Date
Page No.



7 2 0 4 1 → PreOrder Traversal



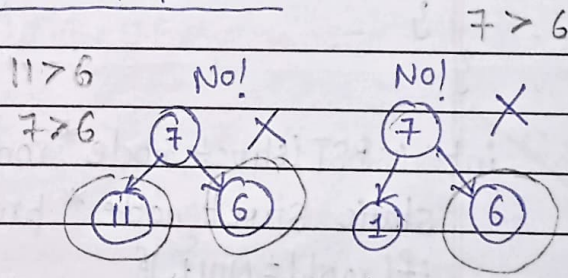
0 4 2 1 7
→ PostOrder Traversal



0 2 4 7 1 → InOrder Traversal

Lec-71 Binary Search Trees: Introduction and properties:

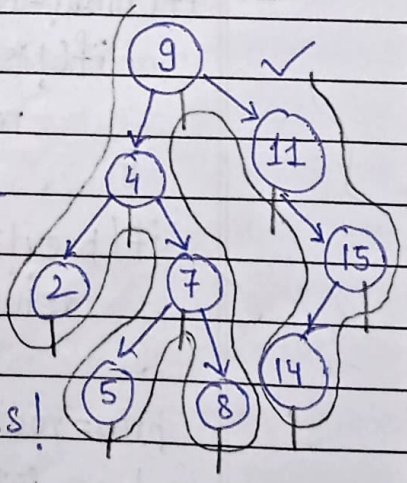
Binary Search Tree:



1. It is a Type of Binary Tree!

Imp Properties:

→ All Nodes of the left Subtree are lesser.
→ All nodes of the right Subtree are greater.
→ left & Right Seubtrees are also BST.
→ There are no duplicate nodes.
Imp → InOrder Traversal of a BST gives an ascending Sorted Array!



yes!

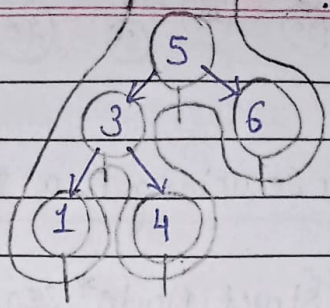
2, 4, 5, 7, 8, 9, 11, 14, 15
Yes Ascending!

Lec-72 Checking if a Binary Tree is a BST or Not!

Green House
Date
Page No.

```
#include<stdio.h>
#include<stdlib.h>
struct node{
    int data;
    struct node *left;
    struct node *right;
};
struct node *createNode(int data){
struct node*n;
n=(structnode*)malloc(sizeof(struct node));
n->data=data;
n->left=NULL;
n->right=NULL;
return n;
}
void inOrder(struct node*root){
    if(root!=NULL){
        inOrder(root->left);
        printf("%d ",root->data);
        inOrder(root->right);
    }
}

int isBST(struct node *root){
static struct node * prev=NULL;
if(root!=NULL){
    if(!isBST(root->left)){
        return 0;
    }
    if(prev!=NULL && root->data<=prev->data){
        return 0;
    }
    prev=root;
    return isBST(root->right);
}
```

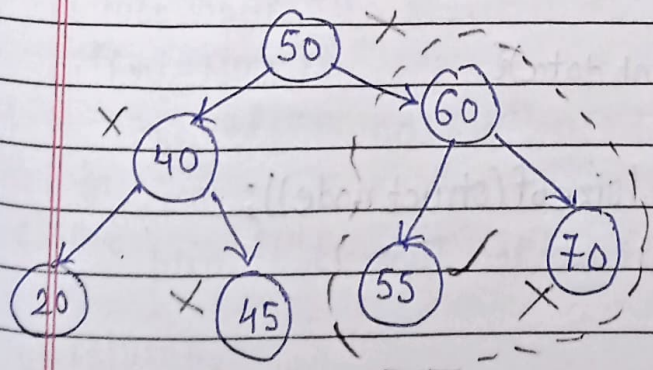
```
else {
    return 1;
}
}
int main(){
    struct node *p = createNode(5);
    struct node *p1 = createNode(3);
    struct node *p2 = createNode(6);
    struct node *p3 = createNode(1);
    struct node *p4 = createNode(4);

// Linking the root node with left and right children
    p->left = p1;
    p->right = p2;
    p1->left = p3;
    p1->right = p4;
    inOrder(p);
    printf("\n");
    if(isBST(p)){
        printf("This is a BST");
    }
else{
        printf("This is not a BST");
}
    return 0;
}
```

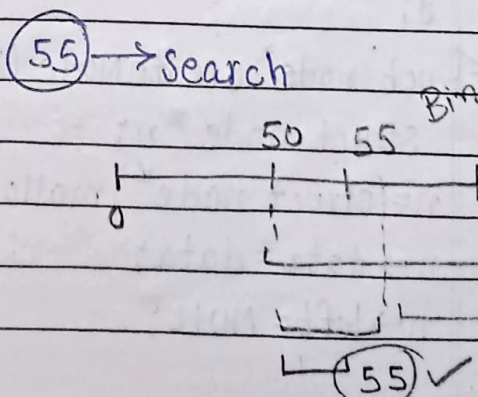


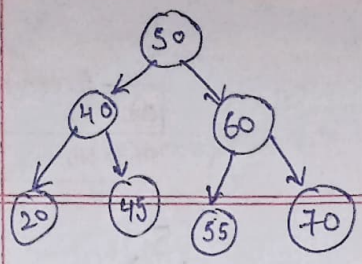

```
1 3 4 5 6
This is a BST
```

c-73 Searching in a Binary Search Trees (Search Operation):

Searching in a Binary Search Tree





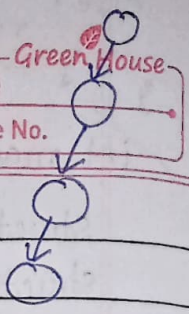


$T(n) = h \times t$ → height of Tree

$\log n \leq h \leq n$

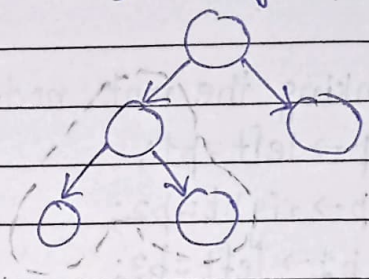
$O(\log n)$ Best case

$O(n)$ Worst case

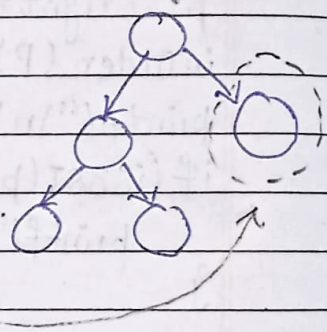


Code for Searching in a BST!

```
Struct Node* search (Struct Node *root, int key){
    if (root == NULL){
        return NULL;
    }
    if (root->data == key){
        return root;
    }
    else if (root->data > key){
        return Search (root->left, key);
    }
    else {
        return Search (root->right, key);
    }
}
```



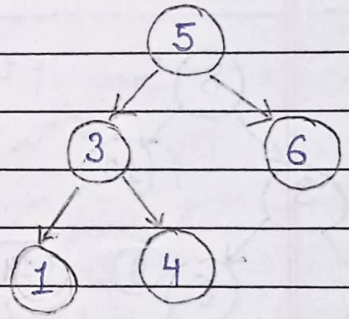
root->data > key



Lec-74 C Code For Searching in a BST:

```
#include<stdio.h>
#include<stdlib.h>
Struct node{
    int data;
    Struct node * left;
    Struct node * right;
};
Struct node* CreateNode (int data){
    Struct node *n;
    n=(struct node*)malloc(sizeof(struct node));
    n->data = data;
    n->left = NULL;
```

```
    n->right = NULL;
    return n;
}
struct node *search(struct node* root, int key){
    if(root==NULL){
        return NULL;
    }
    if(key == root->data){
        return root;
    }
    else if(key<root->data){
      return search(root->left, key);
    }
    else{
        return search(root->right, key);
    }
}
int main(){
    struct node* p = CreateNode(5);
    struct node* p1=CreateNode(3);
    struct node* p2=CreateNode(6);
    struct node* p3=CreateNode(1);
    struct node* p4= createNode(4);

    p->left=p1;
    p->right=p2;
    p1->left=p3;
    p1->right=p4;

    struct node *n = search(p,10);
    if(n!=NULL){
        printf("Found:%d", n->data);
    }
    else{ printf("Element not found");
    }
    return 0;
}
```



Element not found

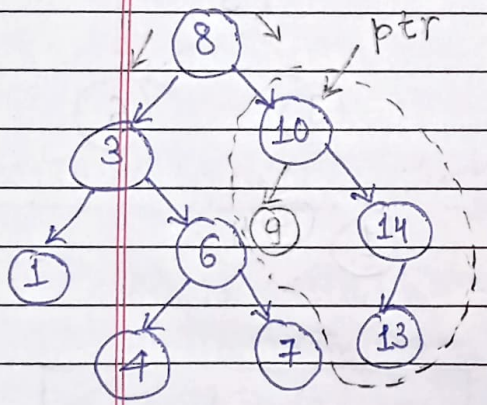
lec-75 Intera-

Iterative Search in a Binary Search Tree:

Green House
Date
Page No.

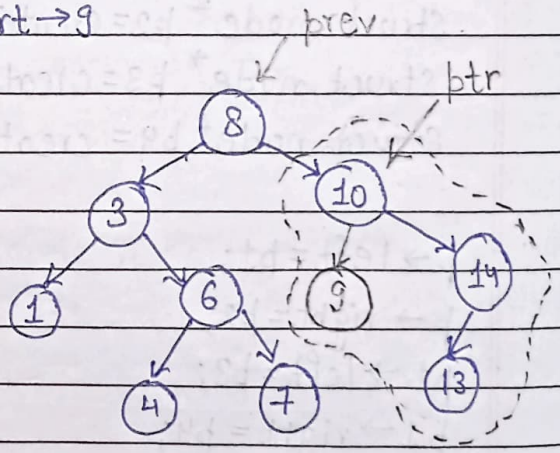
```
Struct node * SearchIter(struct node *root, int key){
    while(root != NULL){
        if(key == root->data){
            return root;
        }
        else if (key<root->data.){
            root = root->left;
        }
        else {
            root=root->right;
        }
    }
    return NULL;
}
```

lec-76 **Insertion in a Binary Search Tree:**



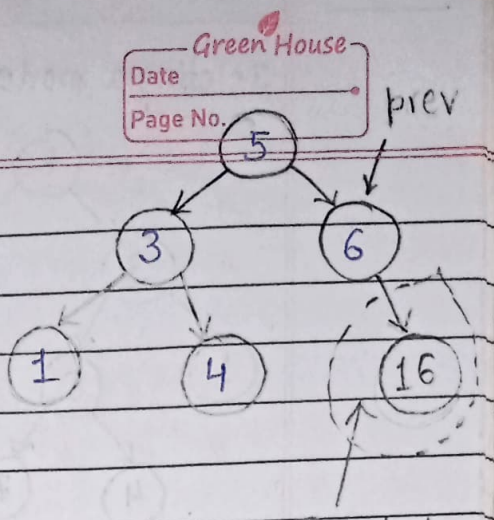
BST → No Duplicates!

Insert → 9




```
#include<stdio.h>
#include< stdlib.h>
Struct node{
    int data;
    struct node* left;
    Struct node* right;
};
struct node* createNode(int data){
    Struct node* n;
```

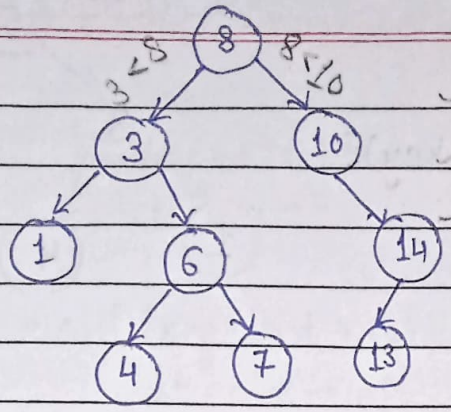
```
    n=(struct node*)malloc(sizeof(struct node));
    n->data=data;
    n->left=NULL;
    n->right=NULL;
    return n;
}

void insert(struct node*root,int key){
    struct node*prev=NULL;
    while(root!=NULL){
        prev=root;
        if(key==root->data){
            printf("Cannot Insert %d, already in BST",key);
            return n;
        }
        else if(key<root->data){
            root=root->left;
        }
        else{
            root=root->right;
        }
    }
    Struct node*new=createNode(key);
    if(key<prev->data){
        prev->left=new;
    else{
        prev->right=new;
    }
}

int main(){
    Struct node*p=createNode(5);
    struct node*p1=CreateNode(3);
    Struct node*p2=createNode(6);
    struct node*p3=createNode(1);
    Struct node*p4=createNode(4);
    p->left=p1;
    p->right=p2;
    p1->left=p3;
    p1->right=p4;

    insert(p,16);
    printf("%d",p->right->
                right->data);
    return 0;
}
```



16

Lec-77 | Deletion in a Binary Search Tree

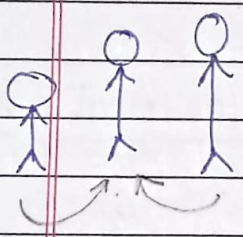
Deleting a node in a BST



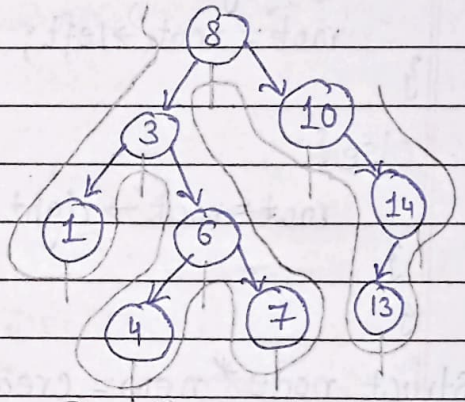
Case1: The node is a leaf node.
Case2: The node is a non-leaf node
Case3: The node is the root node.
Case4: The node is not in Tree

Case:1. The node is a leaf Node
step1 → Search the node
step2 → Delete the node

Case:2. The node is a non-leaf node.

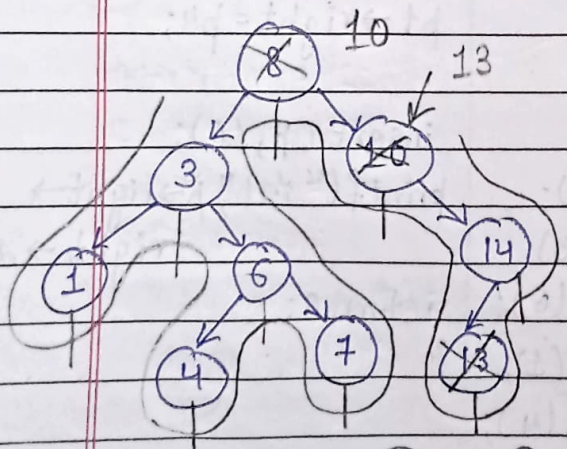


→ Inorder pre
→ Inorder post



1 3 4 6 7 .....
Inorder pre of 6
Inorder post of 6

Case:3 The node is the root node / Node in Between BST



step1:→Search for the node
step2→ Search for Inorder pre & Inorder post.
step3 → keep doing this until the tree has no Empty node

Inorder → 1 3 4 6 7 8 10 13 14

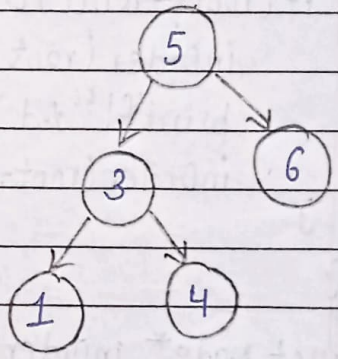
Lec-78

C Code for Deletion in BST:

Green House
Date
Page No.

```
#include<stdio.h>
#include<stdlib.h>
struct node{
    int data;
    struct node *left;
    struct node *right;
};
struct node* createNode(int data){
struct node* n;
n=(struct node*)malloc(sizeof(struct node));
n->data=data;
n->left=NULL;
n->right=NULL;
return n;
}
void inOrder(struct node*root){
if(root!=NULL){
    inOrder(root->left);
    printf("%d ",root->data);
    inOrder(root->right);
}
}
struct Node* inOrderPredecessor(struct node* root){
root=root->left;
while(root->right!=NULL){
    root=root->right;
}
return root;
}
struct node *deleteNode(struct node *root, int value){
struct node *iPre;
if(root==NULL){
    return NULL;
}
```

```
if(root->left==NULL && root->right==NULL){
    free(root);
    return NULL;
}
// search for the node to be deleted
if(value<root->data){
  delete root->left=deleteNode(root->left, value);
}
else if(value>root->data){
   root->right = deleteNode(root->right, value);
}
// Deletion Strategy when the node is found
else{
    iPre=inOrderPredecessor(root);
    root->data= iPre->data;
    root->left = deleteNode(root->left, iPre->data);
}
return root;
}

int main(){
struct node *p = CreateNode(5);
Struct node* p1 = CreateNode(3);
Struct node * p2 = CreateNode(6);
Struct node * p3 = CreateNode(1);
Struct node *p34=CreateNode(4);
//Linking the root node with left and right children
p->left = p1;
p->right =p2;
p1->left=p3;
p1->right =p4;
inOrder(p);
deleteNode(p,5);
printf("\n");
printf("|Data is %d |", p->data);
inOrder(p);
return 0;
}
```



1 3 4 5 6
|Data is 4|1 3 4 6

100-79 AVL Trees - Introduction:

AVL Trees : Height Balanced Binary Tree.

Green House
Date
Page No.

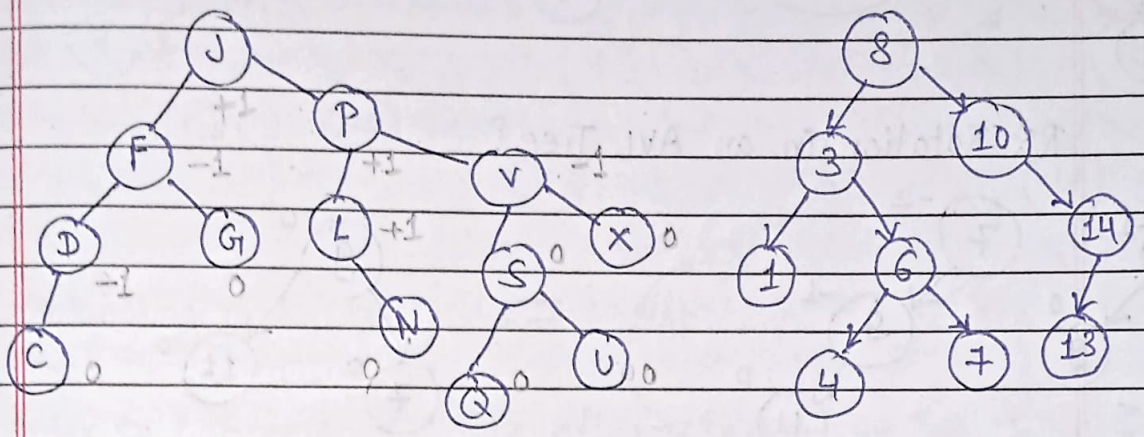
why AVL Trees?

why do we need an AVL Tree?

→ All Almost all the operations in a binary search tree are of order O(h) where h is the height of the tree.

→ If we don't plan our tree properly, this heigh can get as high as n where n is the number of nodes in a BST (skewed tree)

→ To guarantee an upper bound of O(logn) for all these operations, we use balanced trees.



what is an AVL Tree?

→ Height balanced binary Search Trees.

→ Height difference between heights of left and right subtrees for every node is less than or equal to ± 1.

→ Balanced factor = Height of right Subtree - Height of left Subtree

→ Can be -1, 0 or 1 for a node to be balanced in a BST.

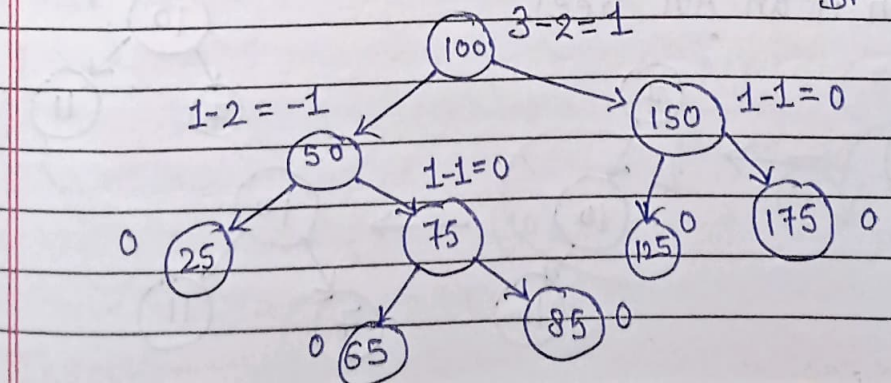
→ Can be -1, 0 or 1 for all nodes of an AVL Tree.

hl → Height of left Subtree
hr → Height of right subtree

BF → Balanced Factor.

→ For a tree to be balanced $|BF| \leq 1$

$BF = hl - hr = \{-1, 0, 1\}$

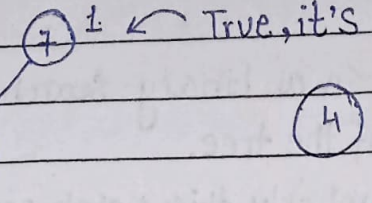


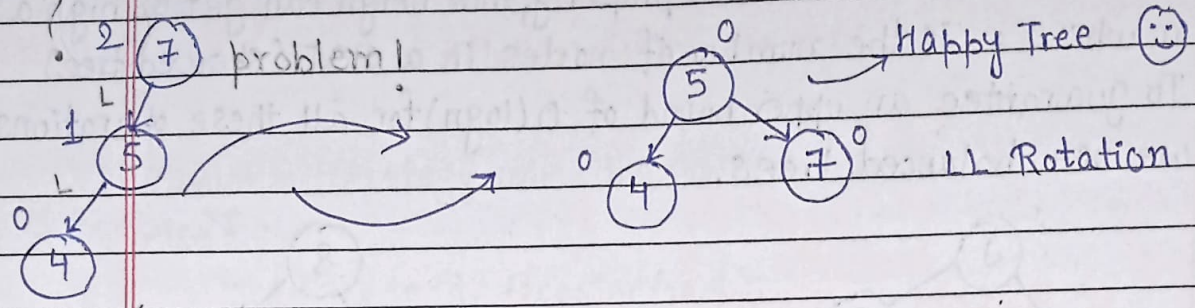
Lec-80 Insertion and Rotation in AVL Tree:

Green House
Date
Page No.

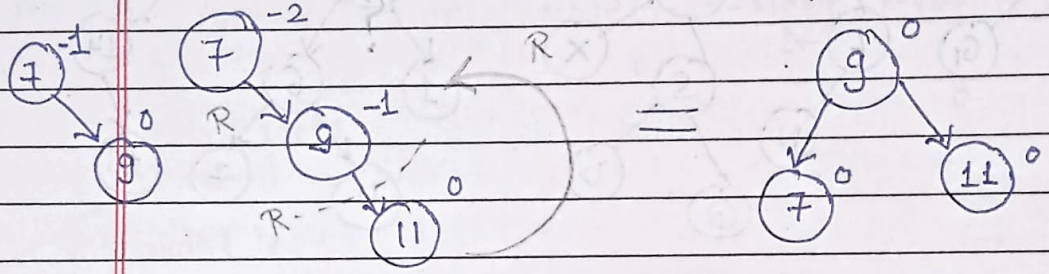
Rotation in AVL Trees

LL Rotation in an AVL Tree?

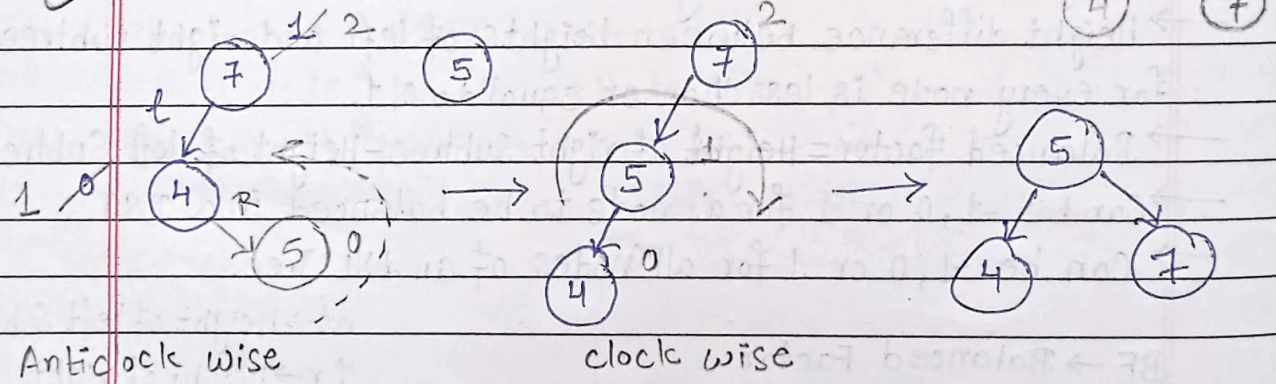




RR Rotation in an AVL Tree?



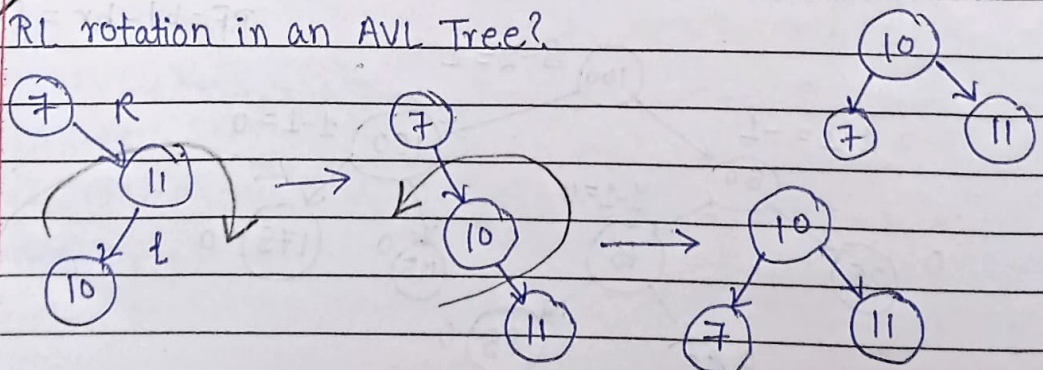
LR rotation in an AVL Tree?



Anticlock wise

clock wise

RL rotation in an AVL Tree?



dc-81

AVL Trees – LL, LR, RL and RR rotations:

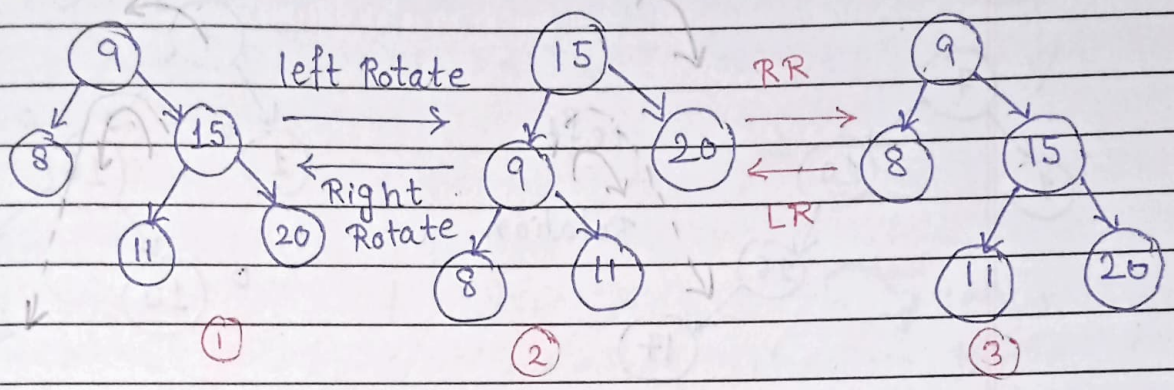
Rotation in AVL Trees with multiple Nodes

Green House
Date
Page No.

Rotate Operations:

we can perform rotate operations to balance a binary search tree such that the newly formed tree satisfies all the properties of a BST. Following are two basic rotate Operations:

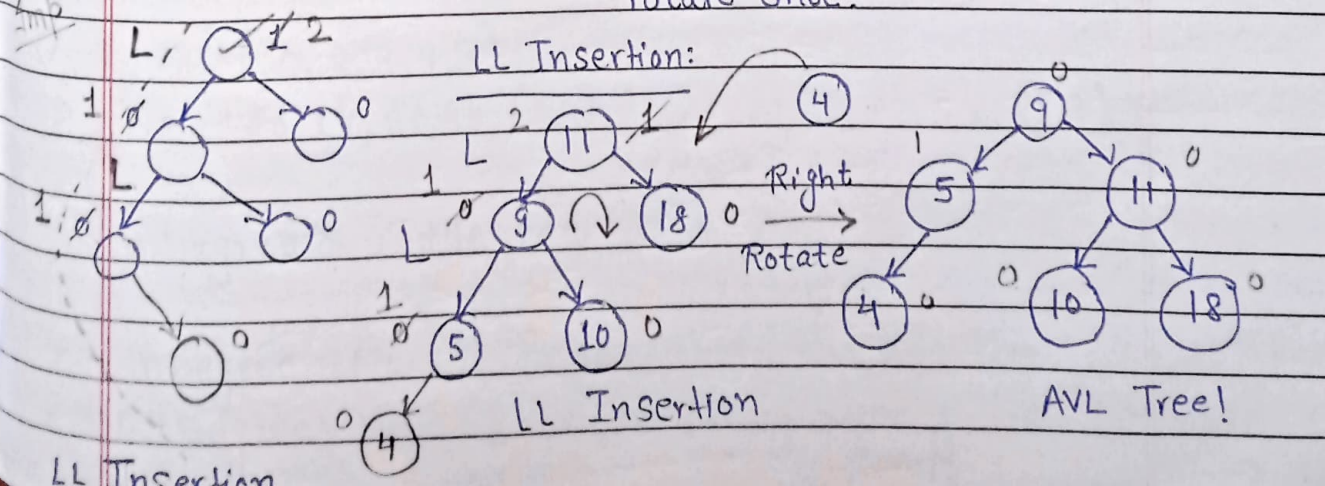
1. Left Rotate w.r.t a node → Node is moved towards the left.
2. Right Rotate w.r.t a node → Node is moved towards the right.



Balancing a AVL tree after Insertion:

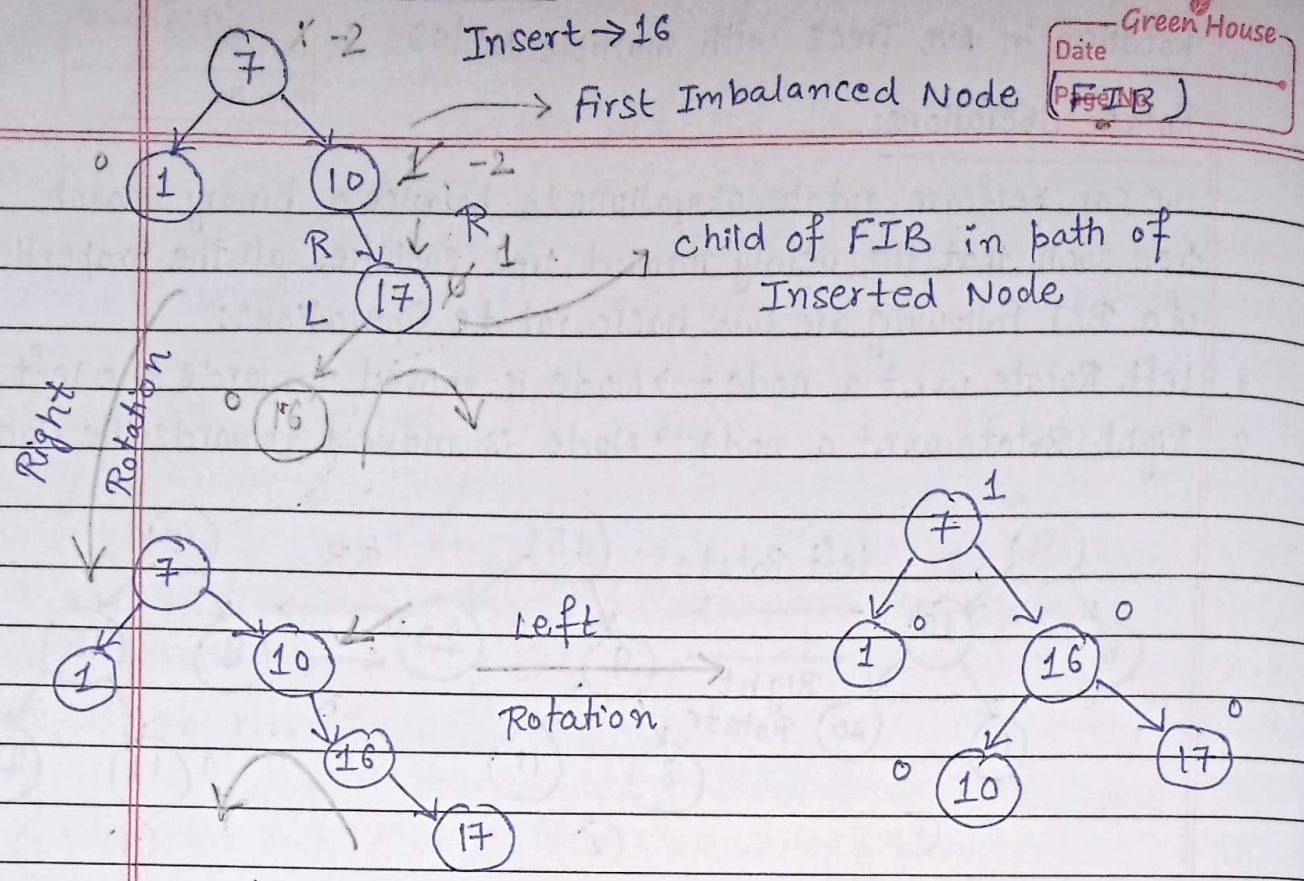
In order to balance an AVL tree after insertion, we can follow the following rules:

1. For a Left-Left Insertion → Right rotate once w.r.t the first Imbalanced node.
2. For a Right-Right Insertion → Left rotate once w.r.t the first Imbalanced node.
3. For a Left-Right Insertion → Left rotate once and then Right rotate Once.
4. For a Right-Left Insertion → Right rotate once and then left rotate once.

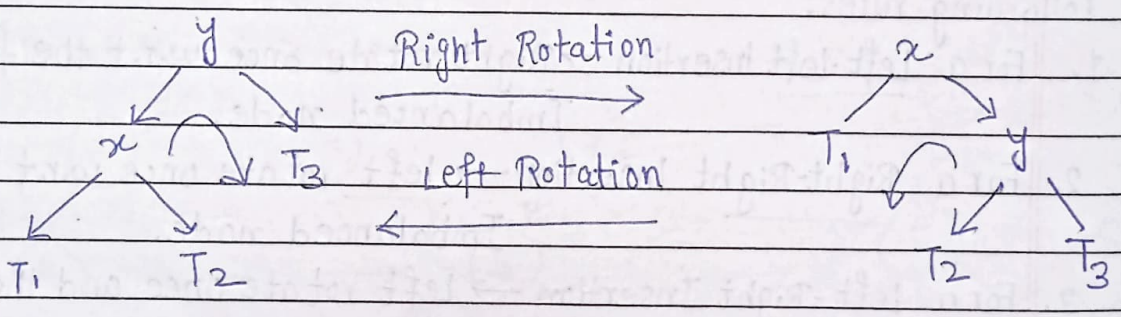


LL Insertion

Right-Left Insertion



Lec-82. C Code for AVL Tree Insertion & Rotation (LL, RR, LR and RL Rotation):-



Lec-83 Introduction to Graphs (Graph Data Structure)

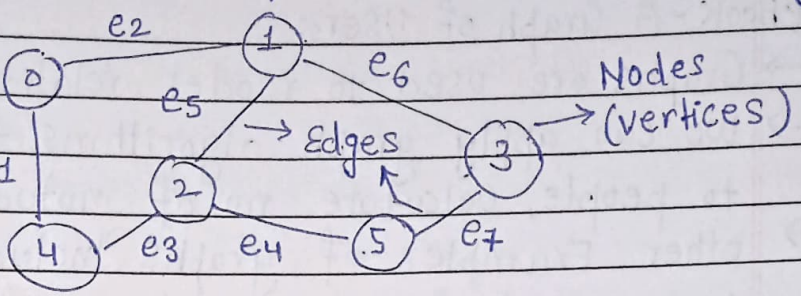
What is a Graph?

→ Array / Linked-Lists & Stacks → Linear Data Structure.

→ BST & AVL Trees → Non-linear Data Structure.

Graph is an Example of Non-Linear DS.

A Graph is a Collection of Nodes Connected through edges.

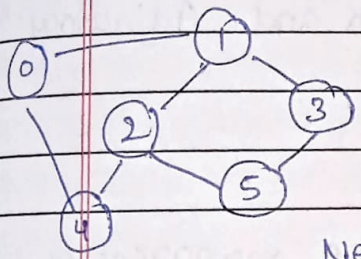


G(V, E).

V = {0, 1, 2, 3, 4, 5}

E = {e1, e2, e3, e4, e5, e6, e7}

Formal Definition of a Graph:



→ A Graph G = (V, E) is a collection of vertices and edges connecting these vertices.

→ Used to model Paths in a City, Social Networks, website backlinks, Internal employee network etc.

→ A vertex or a Node is one of fundamental unit / entity of which graphs are formed.

→ A Edge is uniquely defined by it's 2 endpoints.

→ Directed edges – One way Connection (A → B)

→ Undirected edges – Two way Connection (A — B)

→ Directed Graph – All directed Edges

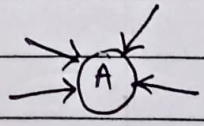
→ Undirected Graph – All Undirected Edges.

Here,

V = {1, 0, 2, 3, 4, 5}

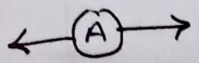
E = { {0, 1}, {0, 4}, {2, 4}, {1, 2}, {1, 3}, {2, 5}, {5, 3} }

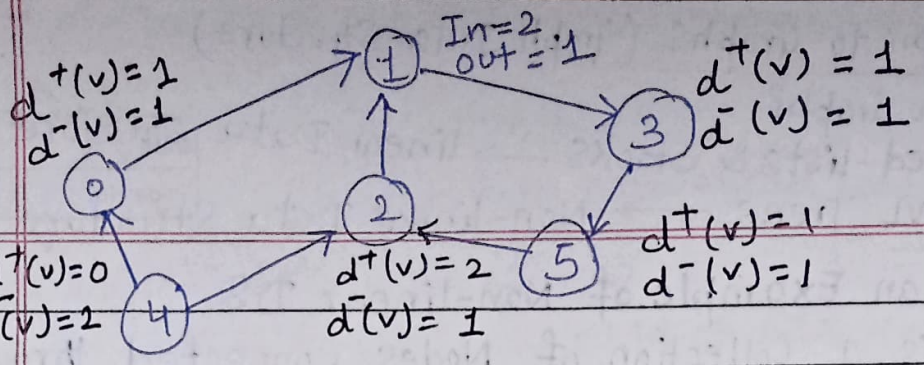
InDegree & Outdegree of a Node:



→ InDegree = No. of Edges ~~going out of~~ entering the Node.

→ OutDegree = No. of Edges ~~going coming into~~ Leaving the Node.





Green House
Date
Page No.

Facebook - A Graph of Users:

→ Graphs are used to model relationship between Nodes
→ we can apply graph algorithms to suggest friends to people, calculate no of mutual friends etc.
→ other Example of graphs include result of a web crawl for a website or for the entire world wide web, city routes etc.

Lec-84 Representation of Graph - Adjancency Link and Adjacency Matrix and other Representations:-

- Graphs = Nodes & Edges

→ used to model real world problems like managing a Social network, website links etc.
→ Helps to Solve problems like is there a path between Delhi and California by road. If yes which one is shortest?

Adjacency Matrix:-