Data Structure

5. Tree

Handwritten by pankaj kumar

Tree (BT & BST)

Date
Page No. 212

* Binary Tree Theory :*

(45) Flatten a B.T to LL (254) (46) Serialize & Deserialize a B.T (256)

(5) Binary Search Trees Introduction (258)

(6) Operation on Binary search Tree (Find min^m/max^m/Insert/Delete) (258)

(7) * BST problem * (263)

(1) Find LCA (263) (2) Shortest path B/w Two nodes (263)

(3) Check BST or not (263) (4) sorted array to BST (265)

(5) sorted DLL to BST (265) (6) kth smallest element (266)

(7) Floor & ceil (267) (8) Union & intersection (269)

(9) print all element b/w a rage: k1 & k2 (269)

(10) check elements of two BST same or not (270)

(11) For key value 1..n, how many unique BST are possible (270)

(12) BST From Preorder (271) (13) Inorder successor/predecessor (272)

(14) Two sum in BST (272) (15) BST Iterator (274)

(16) Recover BST (274) (17) largest BST in BT (276)

(8) Balanced Binary Search Trees (278)

(9) AVL (Adelson-Velskii and Landis) Trees (278)

(10) Generic Trees (N-ary Trees) (284).

(11) Threaded Binary Tree Traversal (285)

(12) Ex-pression Trees (290).

(13) XOR Trees (291)

(14) Red-Black Trees (292)

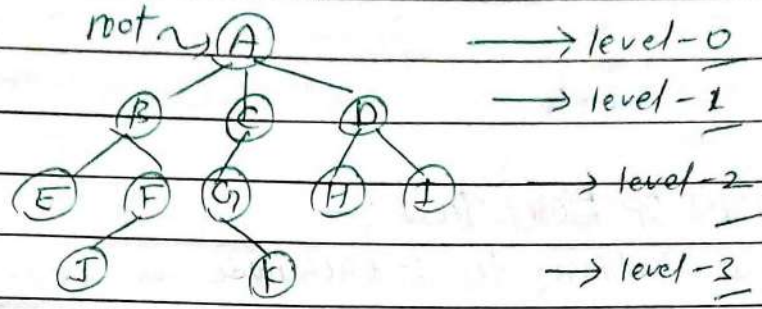
(14) Splay Trees (293)

(15)

① Introduction :-

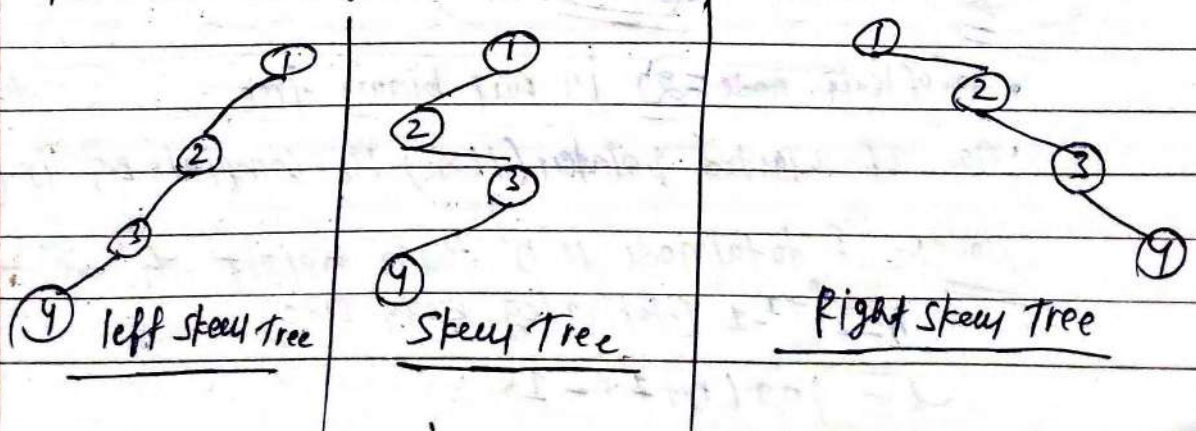
• A tree is a data structure similar to linked list but unlike LL. Here each node points to a number of nodes. Tree is a example of non-linear data structure is a way of representing the hierarchical nature of a structure in a graphical form.

* Glossary



- root : node with no parent (can be at most one in a tree). (A)
- edge : link from parent to child.
- leaf node : a node with no children (E, J, K, H and I).
- siblings : children of same parent (B, C, D are siblings of A).
- ancestor & descendant :- 'p' is ancestor of 'q' if there exist a path from root to 'q' and p appears on the path. and 'q' is called decendant of 'p'
- depth :- depth of a node is the length of the path from root to the node of a node (Ex :- depth of G is 2, A—C—G)
- level :- The set of all nodes at a given deth (ex :- B, C, D → level 1)
- Height of a node :- length of path from that node to deepest node. (EX :- Height of node 'B' is '2', B → F—J)
- Height of tree :- maxm Height among all nodes in the tree.
- depth of tree :- maxm depth among all nodes nodes in the tree
- Size of a node :- no. of descendants including itself (For C=3, C, G, K)

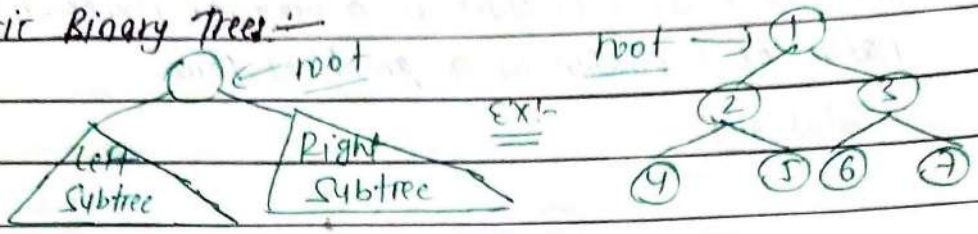
* left skew, Right skew and skew tree.



② Binary Tree

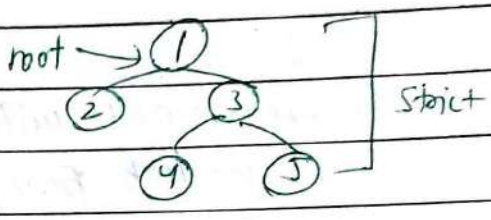
- A tree is called binary tree if each node has zero child, one child or two child. Empty tree is called a valid binary tree.

*Generic Binary Trees :-

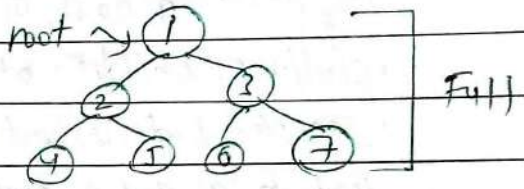


*Types of Binary Trees :-

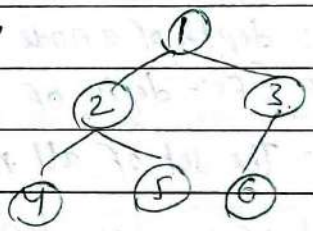
(i) Strict Binary tree :- each node has two children exactly or no children.



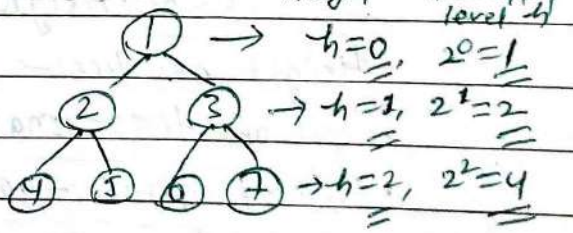
(ii) Full Binary tree :- each node has exactly two children and all leaf nodes are at same level



(iii) Complete Binary tree :- if all leaf nodes are at height h or $h-1$ and also without any missing no. in the sequence.



*Properties of Binary Trees :-



Full binary tree:
no. of nodes = $(2^{h+1}-1)$
$[2^0+2^1+2^2+\dots+2^h = 2^{h+1}-1] = 2^{h+1}-1$

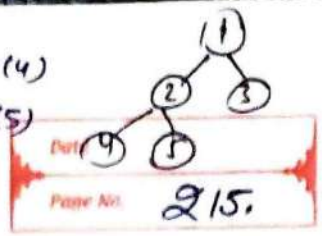
complete binary tree:
2^h (minimum) and $(2^{h+1}-1)$ (max^m)

- no. of leaf node = 2^h in full binary tree
- no. of wasted pointers (None) in complete B.T is $(n+1)$

Note :- if total node is 'n' then height 'h' will be
$n = 2^{h+1}-1$ (taking log both side)

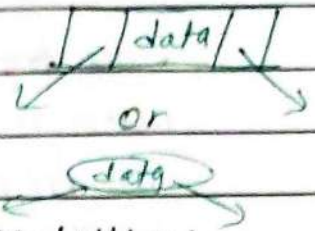
$$h = \log(n+1) - 1$$

② root->left->left = new(4)
root->left->right = new(5)
return 0;
}



* Structure of Binary Trees

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



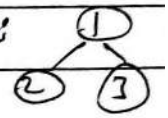
* Operation on Binary Tree

Basic Operation:-
- Inserting an element into a tree
- Deleting an element from a tree
- Searching for an element
- Traversing the tree

Auxiliary Operation:-
- Finding the size of tree
- Finding height of tree
- Finding level which have maxm sum
- Finding least common ancestor (LCA) for a given pair of nodes and more

Ex:- build a tree

```
struct BinaryTreeNode* new(int data)
{
    struct BinaryTreeNode* node =
        (struct BinaryTreeNode*) malloc
        (sizeof(struct BinaryTreeNode));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}
int main()
{
    struct BinaryTreeNode = new(1);
    root->left = new(2);
    root->right = new(3);
```



* Application of Binary Tree

- Expression tree are use in compiler
- Huffman coding tree used in data compression algorithm
- BST, which supports search, insertion and deletion on a collection of items in O(logn) (average)
- Priority Queue (PQ), which supports search and deletion of minm (or maxm) on a collection of item in logarithmic time (in worst case).

③ Binary Tree Traversals

- The process of visiting all nodes of a tree is called tree traversal
- In linear data structure the elements are visited in sequential order. But in tree data structure there are many different way

* clarify the traversal

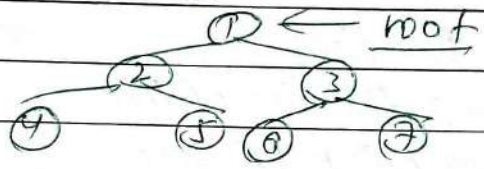
current node → D | left child node → L | Right child node → R

- The traversal is based on the order in which current node is processed.

(i) Preorder (DLR) Traversal
(ii) Inorder (LDR) Traversal
(iii) Postorder (LRD) Traversal

Note:- there is another traversal method which does not depend on the above orders and it is:

- Level order Traversal: Inspired from Breadth First Traversal (BFS of Graph algorithms)



(i) Preorder Traversal :-

Steps for Preorder Traversal

a.) Visit the root
b.) Traverse the left subtree in Preorder
c.) Traverse the right subtree in Preorder

Note:- after finishing the processing of left subtree. It must return to process the right subtree. For this we must maintain the root information. For this we use stack, Bcz of its LIFO structure. we can get right subtrees back in the reverse order.

- preorder traversal = 1 2 4 5 3 6 7

* code (Recursive) :-

```
void preOrder (struct BinaryTreeNode *root){
    if (root){
        printf(root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

time = O(n)
space = O(n)

* Non-Recursive :-

```
void preOrderNonRecursive (struct BinaryTreeNode *root){
    struct Stack *s = createStack();
    while (1){
        while (root){
            print "root->data" // (current.node
            push (s, root);
            root = root->left;
        } // compelete left
        if (isEmpty(s))
            break;
        root = pop (s);
        root = root -> right; //goes to right now
    }
    deleteStack(s);
}
```

print left

when left compelete take top of stack and goes to right again repeat above

Time = O(n)
Space = O(n)

(ii) In Order Traversal :-

Steps

a.) Traverse the left subtree in inorder
b.) Visit the root
c.) Traverse the right subtree in inorder

* Recursive Code:-

```
void inOrder (struct BinaryTreeNode *root){
    if (root){
        inOrder (root->left);
        print "root->data";
        inOrder(root->right);
    }
}
```

Time: O(n) | Space : O(n)

visited order :- 4 2 5 1 6 3 7

* Non-Recursive InOrder:

- instead of processing the node before (like preorder) going to left subtree, process it after popping (which indicate completion of left subtree)

```
void inOrderNonRecursive(struct BinaryTreeNode *root){
    Stack<BinaryTreeNode*> S;
    while(1){
        while(root){
            push(S, root);
            root = root->left;
        }
        if(isEmpty(S))
            break;
        root = pop(S);
        print" root->data";        } after popping process current node
        root = root->right;        } completion of left subtree & current node now go right subtree
    }
```

Time: O(n)
Space: O(n)

(iii) Post Order Traversal :-

Steps :-

a.) Traverse the left subtree in Post order
b.) Traverse the right subtree in Post order
c.) Visit the root

visited order of tree node :- 4 5 2 6 7 3 1.

* Recursive Code :-

```
void postOrder(struct BinaryTreeNode *root){
    if(root){
        postOrder(root->left);
        postOrder(root->right);
        print" root->data";
    }
}
```

Time: O(n)
Space: O(n)

* Non-Recursive Post Order :-

- In preorder & inorder traversals, after popping the stack element we do not need to visit them again. But in post-order traversal, each node is visited twice. that means after processing the left subtree we will visit the current node and after processing the right subtree we will visit the same current node. But we processing the node during the second visit.

```
void postOrderNonRecursive(struct BinaryTreeNode *root){
    struct stack *s = CreateStack();
    struct BinaryTreeNode *previous = NULL;
    do{
        while(root!=NULL){
            push(s, root);
            root = root->left;
        }
        while(root==NULL && !isEmpty(s)){
            root = top(s);
            if(root->right==NULL || root->right==previous){
                print(root->data);
                pop(s);
                previous = root;
                root = NULL;
            }
            else
                root = root->right;
        } //while end
    }while(!isEmpty(s)); //End of do-while
}
```

previous — used to keep track of the earlier traversed node, to check whether we are returning from left or right

if we are returning from the right subtree then print node, pop them & update previous.

or, if right of root is NULL

we are not returning from right neither root->right is NULL so go to the right subtree

we are making null to process again the parent node of current node

Time: O(n) | space: O(n)

Note:-

(iv) Level Order Traversal:-

Steps:-

a.) Visit the root

b.) While traversing level 'l', keep all the element at level 'l+1' in queue.

c.) Go to next level and visit all the node at that level.

d.) Repeat untill all levels are completed.

visited order in level order: 1 2 3 4 5 6 7

code:-

```
void levelOrder(struct BinaryTreeNode * root){
    struct BinaryTreeNode * temp;
    struct Queue * Q = createQueue();
    if (!root)
        return;
    enQueue(Q, root);
    while (!isEmpty(Q)){
        temp = deQueue(Q);
        print("temp->data");
        if (temp->left)
            enQueue(Q, temp->left);
        if (temp->right)
            enQueue(Q, temp->right);
    }
    deleteQueue(Q);
}
```

Time: O(n) | Space: O(n) (in worst case)

(4) Binary Tree Problems & solution :-

(P1) Find maxm element in Binary Tree.

Soln:- Find maxm ele in left subtree, Find maxm ele in right subtree compare them with root select the one which is maxm.

```
int FindMax (struct BinaryTree *root){
    int root_val, left, right, max = INT_MIN;
    if (root!=NULL){
        root_val = root->data;
        left = FindMax(root->left);
        right = Findmax (root->right);
        if (left>right) max=left;
        else max=right;
        if (root_val >max) max= root_val;
    }
    return max;
}
```

Time: O(n)
Space: O(n)

(P2) Find maxm without recursion

Sol:- using level order traversal just observe the element while deleting.

```
int FindMaxUsingLevelOrder (struct BinaryTreeNode * root){
    struct BinaryTreeNode * temp;
    int max= INT_MIN;
    struct Queue *Q = createQueue();
    enQueue (Q, root);
    while (! isEmpty(Q)){
        temp= deQueue (Q);
        if (max<temp->data) max= temp-> data;
        if (temp->left) enQueue(Q, temp->left);
        if (temp->right) enQueue (Q, temp->right);
    }
    deleteQueue (Q);
    return max;
}
```

time: O(n), space: O(n).

(P3) Search an element in Binary Tree.

Recursion:- Check node & recurse down to left & right subtree.

```
int FindInBinaryTree(struct BinaryTreeNode *root, int data){
    int temp;
    if(root==NULL) return 0;
    else{
        if(data==root->data) return 1;
        else{
            temp=FindInBinaryTree(root->left,data);
            if(temp!=0) return temp;
            else return(FindInBinaryTree(root->right,data));
        }
    }
    return 0;
}
```

Time: O(n)
Space: O(n)

(P4) Search an element in B.T without recursion.

Soln:- use level order traversal. The only change is that instead of printing the data, check whether the root data is equal to the ele. we want to search. Time: O(n), Space: O(n)

(P5) Insert an element into B.T.

- Create the newnode with given (in C with malloc) data/ele.
- we can insert the node wherever we want. To insert an ele. we can use level order traversal and insert the element wherever we find the node whose left or right child is NULL & return from there.

Time: O(n), Space: O(n)

(P6) Finding the size of B.T.

Soln:- cal. size of left & right recursive, add 1 (current node) return.

```
int sizeOfBT(struct BinaryTreeNode *root){
    if(root==NULL) return 0;
    else
        return(sizeOfBT(root->left)+1+sizeOfBT(root->right));
}
```

Time: O(n), Space (O(n))

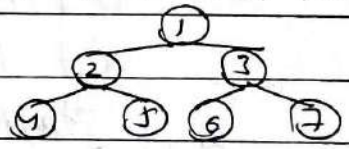
(P7) size of B.T without recursion

Soln:- using level order traversal. when delete increment '1' count.

```
int sizeofBT (struct BinaryTreeNode *root) {
    Struct BinaryTreeNode * temp;
    if (!root) return 0;
    Struct Queue *Q = create Queue ();
    int count = 0;
    enQueue (Q, root);
    while (! isEmpty (Q)) {
        temp = deQueue (Q);
        count ++;
        if (temp->left) enQueue (Q, temp->left);
        if (temp->right) enQueue (Q, temp->right);
    }
    deleteQueue (Q);
    return count;
}
```

Time: O(n), space: O(n)

(P8) Print level order data in reverse order i.e 4 5 6 7 2 3 1 for



Soln:- in level order traversal when deleting from queue insert that into stack (st2) and when traversal complete print the all stack data (with popping). Time: O(n), space O(n)

(at end of each level pop from st1 to st2) (from st2)

(P9) deleting the tree

Soln:- to delete a ~~tree~~ tree. we must traverse all the nodes of the tree and delete them one by one. so which traversal should we use; Inorder, Preorder, Postorder or level order?

- Before deleting the parent node we should delete its children node first. so we can use postorder traversal. we can use other traversal but we have to take extra space.

Code:-

```
void deleteBT (struct BinaryTreeNode *root) {
    if (root == NULL) return;
    deleteBT (root->left); deleteBT (root->right);
    Free (root);
}
```

(P10.) Find height (or depth) of the B.T

Soln:- Recursively cal. left & right subtree height of a node and assign height to max of both plus 1. (Similar to Preorder of DFS)

Code:-
```
int heightOfBT (struct BinaryTreeNode *root){
    int leftheight, rightheight;
    if (root == NULL) return 0;
    else{
        leftheight = heightOfBT (root->left);
        rightheight = heightOfBT (root->right);
        return max(leftheight, rightheight)+1;
    }
}
// Time: O(n), Space: O(n)
```

(P11) P-10 without recursion

Soln:- Using level order traversal same as BFS graph algo.
- End of level is identified with NULL.

```
int FindheightofBT (struct BinaryTreeNode * root){
    if (!root) return 0;
    int level = 0;
    struct Queue * Q = createQueue();
    enQueue(Q, root) → enQueue(Q, NULL); //end of first level
    while (!isEmpty(Q)){
        root = deQueue(Q);
        if (root == NULL){ //completion of current node
            if (!isEmpty(Q)) //put another for next level
                enQueue(Q, NULL);
            level++;
        }
        else{
            if (root->left) enQueue(Q, root->left);
            if (root->right) enQueue(Q, root->right);
        }
    }
    return level;
}
// time: O(n) | Space: O(n).
```

(P.12) Find deepest node of B.T

- deepest node of B.T is the last node of level order traversal

```
enQueue (Q, root);
while (!isEmpty(Q)){
    temp = deQueue(Q);
    if (temp->left) enQueue(Q, temp->left);
    if (temp->right) enQueue(Q, temp->right);
}
deleteQueue(Q);
return temp; //deepest node
```

Time : O(n)
Space : O(n)

(P.13) deleting a node from BST (data is given)

(a) Starting at root, find the node which to be delete.
(b) Find the deepest node in the tree
(c) Replace deepest node's data with node to be delete.
(d) delete deepest node.

(P.14) Find no. of leaves in B.T without recurrion.

use level order traversal if temp node i.e when deleting from Q. check if both left & right of that is NULL increment count plus '1'

Time: O(n), Space: O(n)

(P.15) Find no. of full nodes in BT without recurrion.

use level order traversal. when deleting a node from Queue check if both left & right are present count +1.

Time: O(n), Space: O(n)

(P.16) Find no. of half nodes (either left or right child) not both. in level order traversal. when deleting a node from Queue check

```
if (!temp->left && temp->right || temp->left &&
    !temp->right)
    count++;
```

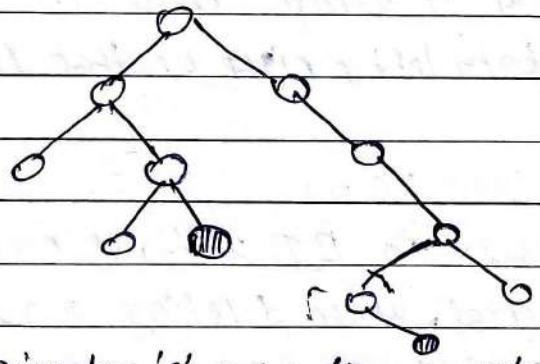
time: O(n)
space: O(n)

P.17 Given two binary trees, return true if they are structurally identical.

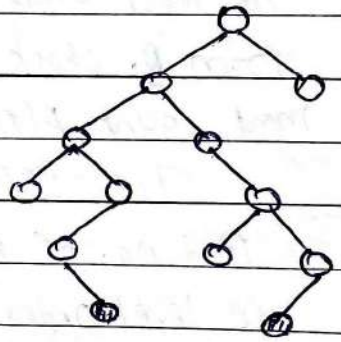
Sol^n:-
- If both tree are NULL then return true.
- If both tree are not NULL, then compare data and recursively check left and right subtree structure.

```
int areStructurallySameBT(struct BinaryTreeNode * root1,
                          struct BinaryTreeNode * root2){
    if (root1==NULL && root2==NULL) return 1;
    if (root1==NULL || root2==NULL) return 0;
    //both non-empty -> compare them
    return (root1->data == root2->data &&
            areStructurallySameBT(root1->left, root2->left) &&
            areStructurallySameBT(root1->right, root2->right));
}
```

P.18 Find diameter of B.T (sometime called the width). Diameter of tree is the no. of nodes on the longest path. Buy two leaves.



Diameter 'd' nodes through roots



Diameter 'd' nodes not through roots

Steps:-
- 1st calculate the diameter of left subtree and right subtrees recursively.
- among these two values, send max^m value along with current level (+1).

code: (1)

```
int height(struct Node* root){
    if (node==NULL) return 0;
    return 1+max(height(node->left), height(node->right));
}
```

```
int diameter (struct Node* root){
    if(root == NULL) return 0;
    int lHeight = height(root->left);
    int rHeight = height(root->right);
    int ldiameter = diameter(root->left);
    int rdiameter = diameter(root->right);
    return max(lHeight+rHeight+1, max(ldiameter, rdiameter));
}
// Time: O(n^2), space: O(n) for call stack.
```

Code ② : optimized

The above implementation can be optimized by calculating the height in the same recursion rather than calling a height() seperately. it reduce time to O(n)

```
int diameterOfTree(struct Node* root, int *diameter){
    int left, right;
    if(!root) return 0;
    left = diameterOfTree(root->left, diameter); // left height
    right = diameterOfTree(root->right, diameter); // right height
    *diameter = max(left+right, *diameter);
    // assign new diameter if left+right > diameter
    return max(left, right)+1;
}
```

(D.19) Finding the level that has the maxm sum in the binary tree.

Soln:- The logic is similar to finding the no. of levels ~~that has the maxm sum in the binary tree~~. the only change is we need to keep track of the sum as well.

```
int FindlevelwithMaxSum (struct Node* root){
    if(!root) return 0;
    struct Node* temp;
    struct Queue *Q=CreateQueue();
    int level=0, maxlevel=0, currentSum=0, maxSum=0;
    enQueue(Q, root);
    enQueue(Q, NULL);          //End of first level
    while(!isEmpty(Q)){
        temp=deQueue(Q);
        if(temp==NULL){ //if current level completed then
            [redacted]  compare sums
        if(maxSum < currentSum){
            maxSum = currentSum;
            maxlevel = level
        }
        currentSum=0;
        if(!isEmpty(Q)) //place Indicator for next level end
            enQueue(Q, NULL);
        level++;
    }
    else{
        currentSum += temp->data;
        if(temp->left) enQueue(Q, temp->left);
        if(temp->right) enQueue(Q, temp->right);
    }
}//while end
    return maxlevel;
}
```

when current level completed.

Time: O(n) | Space: O(n)

(P. 20) Given a binary tree, print out all its root-to-leaf paths

```
void printPaths(struct Node* root, int path[], int len){
    if(root==NULL) return;
```

```
    path[len] = root->data   //append this node to path array.
    len++;
    if (root->left == NULL && root->right == NULL)
        printArray(path, len);
    else{
        printPaths(root->left, path, len);
        printPath(root->right, path, len);
    }
}
```

It is leaf so print the path led to here

(P.21) Check the existance of path with given sum in BT

```
int hasPathSum(struct Node* root, int sum){
    if (root == NULL) return 0;
    sum -= root->data;
    if (sum == 0 && root->left == NULL && root->right == NULL)
        return 1;
    return hasPathSum(root->left, sum) || hasPathSum(root->right, sum);
} //Time: O(n), space: O(n)
```

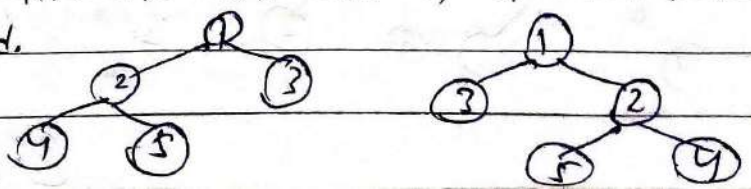
(P.22) Sum of all element in B.T

```
int add(struct Node* root){
    if (root == NULL) return 0;
    else return (root->data + add(root->left) + add(root->right));
} //Time: O(n), space: O(n)
```

(P.23) sum of all element in B.T without recurrion

use level order traverral. every time after deleting node from queue add the node data to sum variable.

(P.24) Convert a tree to its mirror. mirror of a tree is another tree with left and right children of all non-leaf nodes interchanged.



```
struct Node* mirrorOfBT (struct Node* root){
    struct Node* temp;
    if (root){
        mirrorOfBT (root->left);
        mirrorOfBT (root->right);
        /* swap the pointer in this node */
        temp = root->left;
        root->left = root->right;
        root->right = temp;
    }
    return root;                    // Time: O(n)
}                                   // Space: O(n)
```

(P.25) Check whether Two Given trees are mirrors of each other.

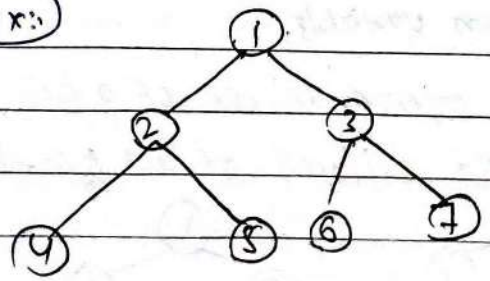
Sol^n:

```
int areMirrors (struct BinaryTreeNode* root1, struct
                                    BinaryTreeNode* root2){
    if (root1 == NULL && root2 == NULL) return 1;
    if (root1 == NULL || root2 == NULL) return 0;
    if (root1->data != root2->data) return 0;
    else return areMirrors (root1->left, root2->right) &&
                areMirrors (root1->right, root2->left);
}
// Time : O(n), space : O(n)
```

(P.26) Find LCA (least Common Ancestor) of two nodes in B.T

The lowest common ancestor b/w two nodes n1 & n2 is defined as the lowest node in tree that has both n1 & n2 as descendants (where we allow a node to be a descendant of itself).

Ex:



LCA(4,5) = 2
LCA(4,6) = 1
LCA(3,4) = 1
LCA(2,4) = 2

Note:- Computation of LCA may be useful for instance, to cal. the distance b/w pairs of nodes in tree :-

dist from root to n1 + dist from root to n2 − 2 * dist from root to LCA

Code:- (Using Single Traversal) Efficient:-

```
struct BinaryTreeNode *LCA(struct BinaryTreeNode *root,
                           struct BinaryTreeNode *n1, struct BinaryTreeNode *n2){
    struct BinaryTreeNode *left, *right;
    if (root == NULL) return root; // root is null
    if (root == n1 || root == n2) return root;
    left = LCA(root->left, n1, n2);
    right = LCA(root->right, n1, n2);
    if (left && right) return root;
    else return (left ? left : right);
}
```

Time: O(n) | Space: O(n)

Note:- if LCA is not present it will return NULL. and if only one n1 or n2 is present then it will return only one as LCA n1 or n2

Code:- Method 2) :- (By storing root to n1 and root to n2 path and then choose last common node in path)

- it is not better bcz it uses two times loop to store path.

ex :- LCA(4,6) ⇒ for 4 path is 1 → 2 → 4
for 6 path is 1 → 3 → 6
(1) common

- (Time) and (Space) for this is also O(n):

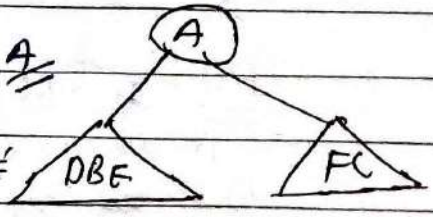
P.27 Construct Binary Tree from given Inorder & Preorder Traversal

Soln:- InOrder Sequence : D B E A F C
PreOrder Sequence : A B D E C F

(i) • In Preorder sequence, leftmost denote root i.e A

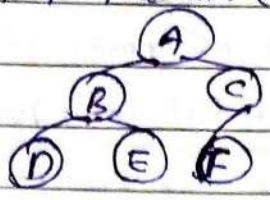
(ii) • A is root of given tree

(iii) • search A in inorder left side of 'A' i.e D B E, right of 'A' i.e F C



is left subtree & right of 'A' i.e F C is right subtree.

(iv). We recursively follow the above step and get the following tree.



Algorithm :-

1. Pick an element from preorder. Increment a Preorder Index variable to pick the next element in the next recursive call.
2. Create a new treeNode with the data as picked element (tNode)
3. Find the picked element's index in inorder. Let (inIndex).
4. Call buildTree for element before inIndex and make built tree as a left subtree of tNode
5. Call buildTree for element ~~before~~ after inIndex and make the builttree as a right subtree of tNode
6. return tNode.

Code :-

```
node * buildTree (char in[], char pre[], int inStrt, int inEnd){
    static int preIndex = 0;
    if (inStrt > inEnd) return NULL;
    Node * tNode = newNode (pre[preIndex++]); //step 1 & 2
    if (inStrt == inEnd) return tNode; //If No children
    int inIndex = search (in, inStrt, inEnd, tNode->data) //Step (3)
    //using inIndex in Inorder traversal, construct left &
    // right subtree. Step (4) & step (5).
    tNode->left = buildTree (in, pre, inStrt, inIndex-1);
    tNode->right = buildTree (in, pre, inIndex+1, inEnd);
    return tNode;
}
```

Note:- (i) Time & space complexity depend on search function if we use linear search Time = O(n) & space = O(n). → O(1) search
(ii) or we can store index & data of Inorder in unordered map then Time will be = O(n) & space = O(n) [O(2n)]

P.28 If we are given two traversal sequences, can we construct the binary tree uniquely?

Soln:- If one of the traversal method is InOrder. Then tree can be constructed uniquely, otherwise not.

⇒ For uniquely tree combination can be.

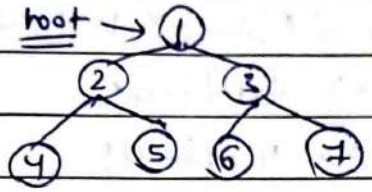
- InOrder & preorder
- InOrder & Post order
- InOrder & Level-Order

do not uniquely tree-

- post & pre
- pre & level
- post & level.

P.29 Print all the ancestors of a node in a Binary tree.

Ex:-

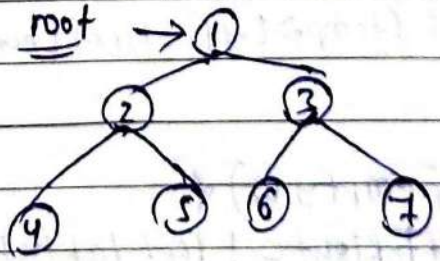


here for 7 the ancestors are: 1,3,7

```
Int printAllAncestors (struct BinaryTreeNode *root, struct BinaryTreeNode
                                                    *node) {
    if (root==NULL) return 0;
    if (root->left==node || root->right==node ||
        printAllAncestors (root->left, node) || printAllAncestors (root->
                                                    right, node)
    {
        printf ("%d", root->data);
        return 1;
    }
    return 0;
}
//Time complexity : O(n) and space complexity : O(n) for recursion
```

P.30 zigzag Tree Traversal:

ex:-



zig-zag Traversal

1 3 2 4 5 6 7

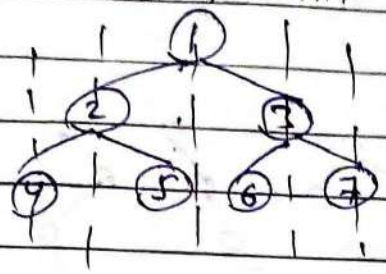
(i) Use 2 stack - currentLevel & nextLevel.
(ii) A variable LeftToRight (whether it is left to right or, right to left).
(iii) if LeftToRight (push left child then right to next level stack).
else, (push right child then left to nextLevel stack)
(iv) swap currentLevel & nextLevel after end of each level

Note:- in (iii) step the currentLevel stack is used, in which we pop element from currentLevel, print them & then check for condition of currentLevel order, leftToRight,

code:- (C++)

```
void zigzagTraversal (Node* root) {
    Stack <Node*> currLevel;
    Stack <Node*> nextLevel;
    bool leftToRight = true;
    currLevel.push(root);
    while (!currLevel.empty()) {
        Node* temp = currLevel.top();
        currLevel.pop();
        if (temp) {
            cout << temp->data << " ";
        }
        if (leftToRight) {
            if (temp->left) nextLevel.push(temp->left);
            if (temp->right) nextLevel.push(temp->right);
        }
        else {
            if (temp->right) nextLevel.push(temp->right);
            if (temp->left) nextLevel.push(temp->left);
        }
        if (currLevel.empty()) {
            leftToRight = !leftToRight;
            swap (currLevel, nextLevel)
        }
    }
}
```

P.31 Find vertical sum of a B.T



vertical-1: 4 , sum = 4
vertical-2: 2 , sum = 2
vertical-3: 1, 5, 6 , sum = 12
vertical-4: 3 , sum = 3
vertical-5: 7 , sum = 7
o/p: 4 2 12 3 7

- This can be solved with the help of hashing tables. the idea is to create a empty map where each key represents the horizontal distance of a node from the root node (root node dis=0) and value in the map maintain sum of all nodes present at same horizontal dism. For each node, we recur for its left subtree by decreasing horizontal distance by 1 and recur for right by increasing horizontal dism by 1.

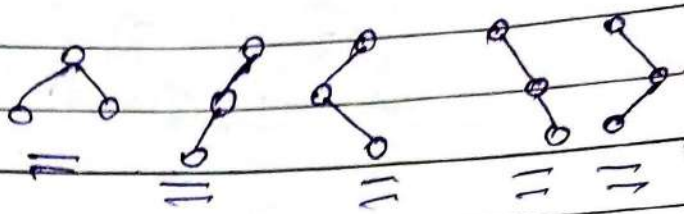
```
void verticalSumUtil (Node *node, hd, map<int, int> &mp){
        if (node == NULL) return;
        verticalSumUtil (node->left, hd-1, mp);
        mp[hd] += node->data;
        verticalSumUtil (node->right, hd+1, mp);
}
void verticalSum (Node *root){
        map<int, int> mp;
        map<int, int>:: iterator it;
        verticalSumUtil (root, 0, mp);
        for (it=mp.begin(); it!=mp.end(); ++it){
                cout<<it->second<<" ";
        }
}
```

P.32 vertical level traversal of B.T.
we will use same approach of vertical sum but instead of sum we push all the node data in vector.
i.e map <int, vector<int>> mp;

Q.33 How many different binary tree are possible with 'n' nodes

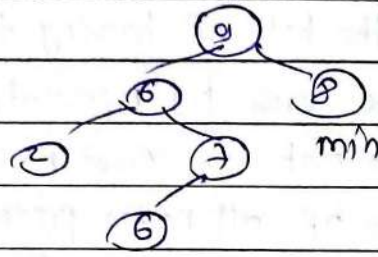
Sol^n:- $2^n - n$ different tree

Ex:- $n=3$, $2^3-3 = 5$ trees.



Q.34 Find the minm depth of Binary Tree

Ex:-



minm Depth = 2

(i) int minDepth (BinaryTreeNode * root){
if (!root) return 0;
if (root->left == NULL || root->right == NULL)
return minDepth (root->left) + minDepth (root->right) + 1;
int left = minDepth (root->left) + 1;
int right = minDepth (root->right) + 1;
return (left < right ? left : right);
}

(ii) Using level order traversal

(i) if root is equal to NULL, minm depth would be zero.
(ii) with every level we keep track depth = level + 1
(iii) we will find a node with leaf node the minm depth will be at that level + 1. which is required ans. return them.

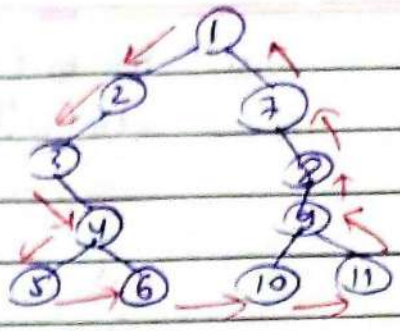
Symmetric question: maximum depth

in (i) case:- we will check for 'left > right'

in (ii) case:- don't return from traversal count all the level of Binary Tree the maxm depth = last level + 1

Date
Page No. 257.

(P35) Boundary Traversal :-



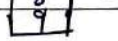
anticlockwise :-

1, 2, 3, 4, 5, 6, 10, 11, 9, 8, 7

steps :-

(I) Left boundary excluding leaf
(II) leaf node
(III) Right boundary in the reverse exclude the leaf.

- for left boundary take the root (I) & goes to left (ii) iterative if left no exist goes to right until leaf reached required sequence [1, 2, 3, 4]
- for leaf node do In order not level order bcz order of sequence should be left first fill now. [1, 2, 3, 4, 5, 6, 10, 11]
- for right boundary goes to right of root & start to right iteratively if right not exist go to left until leaf reached and store in stack then pop all, [1, 2, 3, 4, 5, 6, 10, 11, 9, 8, 7]



code :- (C++)

```
void leftBoundary(node* root, vector<int> &res){
    node * cur = root->left;
    while (cur){
        if (!isLeaf(cur)) res.push_back(cur->data);
        if (cur->left) cur = cur->left;
        else cur = cur->right;
    }
}
void rightBoundary(node *root, vector<int> &res){
    node * cur = root->right;
    vector<int> tmp;
    while (cur){
        if (!isLeaf(cur)) tmp.push_back(cur->data);
        if (cur->right) cur = cur->right;
        else cur = cur->left;
    }
    for (int i = tmp.size()-1; i>=0; --i) res.push_back(tmp[i]);
}
```

```
void leaves (node * root, vector <int> & res) {
    if (isLeaf (root)) {
        res.push_back (root->data);
        return;
    }
    if (root->left) leaves (root->left, res);
    if (root->right) leaves (root->right, res);
}

vector <int> printBoundary (node* root) {
    vector <int> res;
    if (!root) return res;
    if (!isLeaf (root)) res.push_back (root->data);
    leftBoundary (root, res);
    leaves (root, res);
    rightBoundary (root, res);
    return res;
}
```

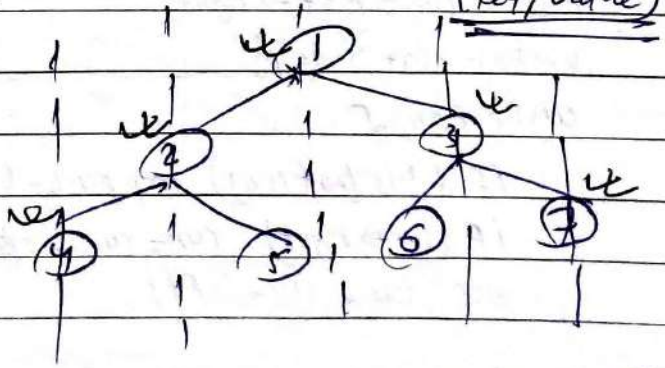
P:36 :- Top view of B.T

like vertical sum and vertical level traversal instead of storing all same vertical level element, we take only one element (First occurrence according to level order traversal) i.e with recursion we use pre-order. at every stage we check if key is not present then we add element with the key.

(key, value)

ex:-

(2,7)
(-1,3)
(-2,4)
(1,2)
(0,1)



key will be in increasing order :. o/p = 4, 2, 1, 3, 7

Code :- (Recursion)

["horizontal distance" = key]

```
void Topview (Node *node, hd, map<int, int> &mp){
    if(!node) return;
    if (mp.find(hd) == mp.end()) mp[hd] = node->data;
    Topview (node->left, hd-1, mp);
    Topview (node->right, hd+1, mp);
}
Topview (root, 0, mp); // Function call
```

Time : O(n)
Space : O(n)

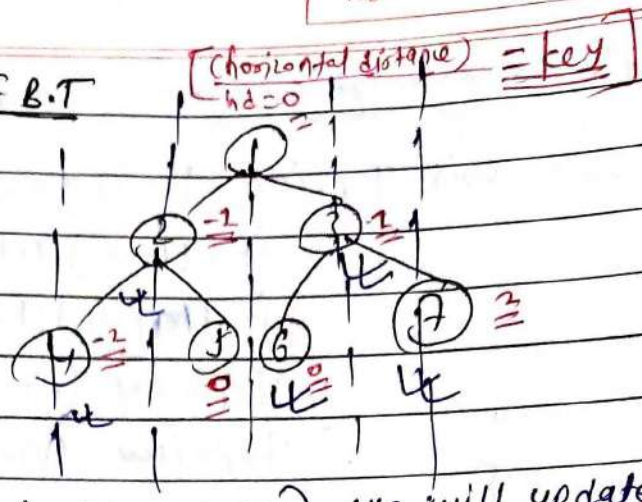
Code :- Iterative using level order traversal

```
vector <int> topView (Node *root){
    vector <int> ans;
    if (root == NULL) return ans;
    map<int, int> mp;
    queue <pair< Node*, int>> q;
    q.push ({root, 0});
    while ( !q.empty() ){
        auto it = q.front();
        q.pop();
        Node* node = it.first;
        int hd = it.second;
        if (mp.find (hd) == mp.end()) mp[hd] = node->data;
        if (node->left != NULL)
            q.push ({node->left, hd-1});
        if (node->right != NULL)
            q.push ({node->right, line+1});
    }
    for(auto it : mp){
        ans.push_back(it.second);
    }
    return ans;
}
```

Time : O(n)
Space : O(n)

P:37 Bottom view of B.T

O/P:- 4, 2, 6, 3, 7

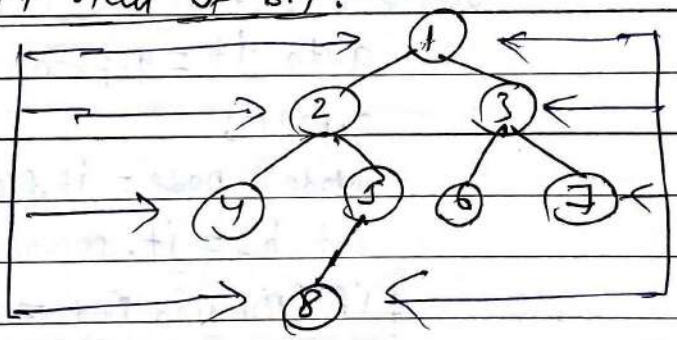


• In top view problem we didn't need to check the condition if key is present or, not" (highlighted red box P:36). we will update every time the value with key, and according to this we will have bottom element for every vertical level which will be our bottom view element

Ex:- above ex: A/c to recursion :-

(2,7)
(1,3)
(-2,4)
(-1,2)
(0,1) → (0,5) → (0,6)

P:38 Right/left view of B.T:-



left view
(1,2,4,8)

right view
(1,3,7,8)

⇒ Iterative :-

• For solving "left view" problem, we take "every first element of level order traversal"

(To visualize if this is first or, not in the queue we insert Null at every end of level and we check if top element is null after popping a node element then that will be last element for that level and if current element is null then the next top element will be first element for next level).

- For solving "right view" problem, we will take "last element of every level order traversal"

Ex:-

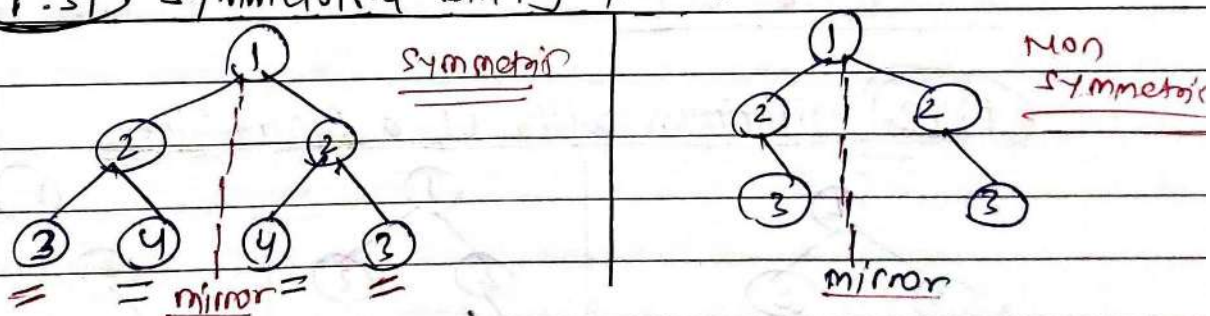
Level 0: 1
Level 1: 2 3
Level 2: 4 5 6 7
Level 3: 8

Left view = (1, 2, 4, 8)
Right view = (1, 3, 7, 8)

⇒ Recursive :- (Right view) :-

```
void recursion(Node* root, int level, vector<int> &res)
{
    if(root == NULL) return;
    if(res.size() == level) res.push_back(root->data);
    recursion(root->right, level+1, res);
    recursion(root->right, level+1, res);
}
vector<int> rightView(Node* root){
    vector<int> res;
    recursion(root, 0, res);
    return res;
}
```

P:39 Symmetrical Binary Tree :-



- It is like mirror property. we can ignore the root node as it is lying on the mirror line. in the next level, for a symmetric tree the node at root's left should be equal to the node at the root's right. again left node of root's left node should be equal to right node of root's right node. ex:- 3 nodes

and right node of root's left node should be equal to the left node of root's right node for ex:- node 4

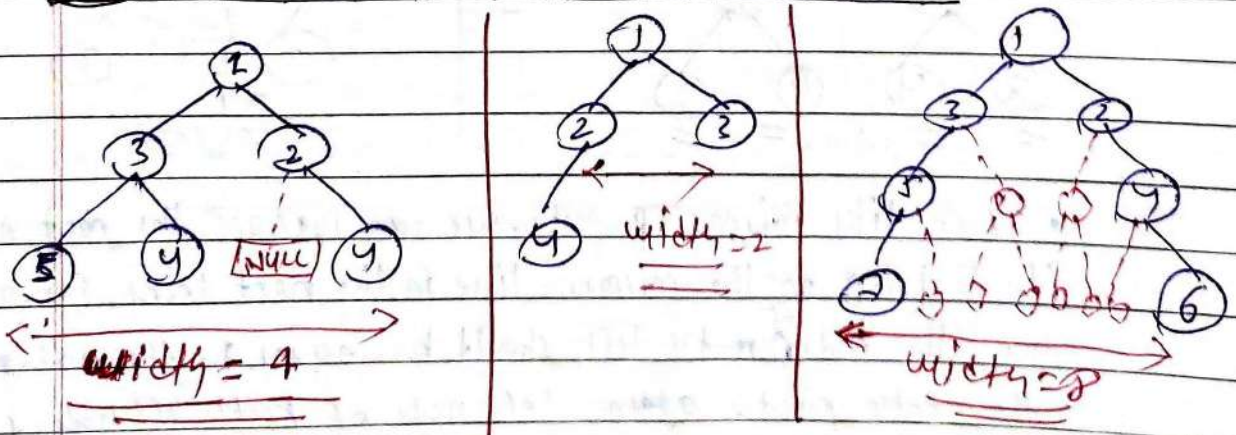
traversal on left-subtree → root left right

traversal on right-subtree → root right left

```
bool isSymmetricUtil(node* root1, node* root2){
    if(root1==NULL && root2==NULL) return true;
    else if(root1==NULL || root2==NULL) return false;
    else{
        return (root1->data == root2->data) &&
               isSymmetricUtil(root1->left, root2->right) &&
               isSymmetricUtil(root1->right, root2->left);
    }
}
bool isSymmetric(node* root){
    if(!root) return true;
    return isSymmetricUtil(root->left, root->right);
}
```

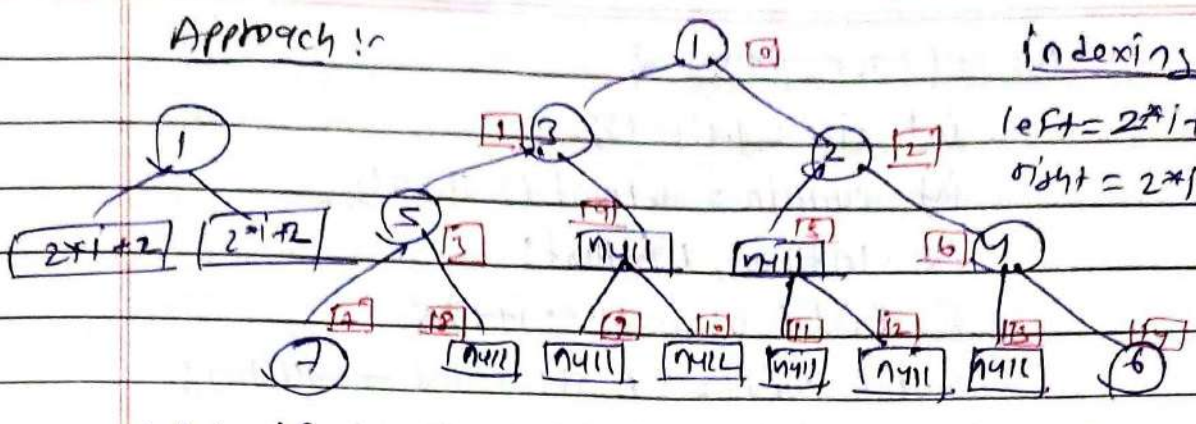
Time & Space → O(n)

P:40 Maximum width of a Binary Tree



width = at the same level rightmostNode - leftmostNode + 1

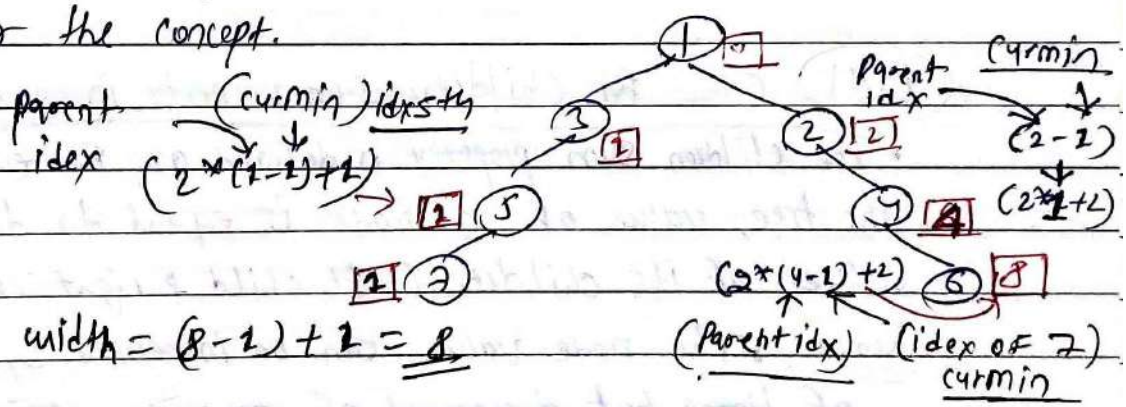
Approach :-



Note :- if we index the tree we can easily calculate width, (rightMostNode - leftmostNode + 1). To store the index we use level order traversal. it is 0 based indexing

Prevention of Integer overflow :-

- Third approach has a problem, as we are multiplying 2 to the current index, it can happen in a tree that we overshoot the bound of an integer. Therefore we need to find a strategy
- Before starting a level, we can store the left-most index in a variable (say curMin). Now whenever we assign the index for its children, we take the parent node index as (i - curMin) rather than i. The below illustration will clear the concept.



Code :-

```
int widthOfBT(node* root){
    if(!root) return 0;
    int ans=0;
    queue<pair<node*, int>> q;
    q.push({root, 0});
```

```
while(!q.empty()){
    int size = q.size();
    int curMin = q.front().second;
    int leftMost, rightMost;
    for(int i=0; i<size; i++){
        int cur_id = q.front().second - curMin;
        node* temp = q.front().first;
        q.pop();
        if(i==0) leftMost = cur_id;
        if(i==size-1) rightMost = cur_id;
        if(temp->left) q.push({temp->left, cur_id*2+1});
        if(temp->right) q.push({temp->right, cur_id*2+2});
    }
    ans = max(ans, rightMost-leftMost+1);
}
return ans;
}
```

Time Complexity: O(N) | Space Complexity: O(N)

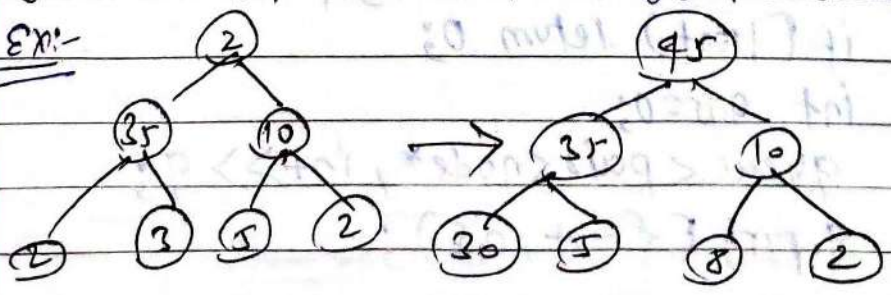
P:41 Check for children sum property in a binary tree.

- The children sum property is defined as, For every node of the tree, value of a node is equal to the sum of values of its children (left child & right child)

Note:- The node value can be increased by 1 any number of times but decrement of any node value is not allowed.

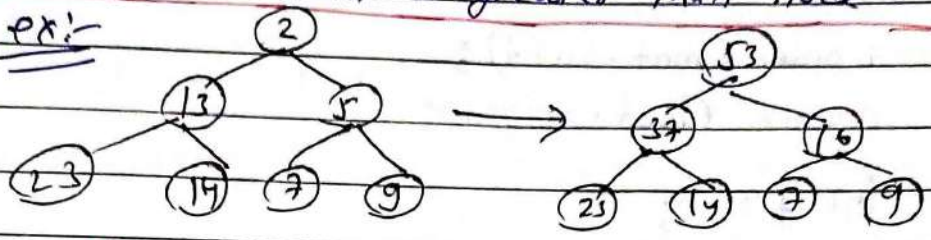
- A value for a NULL node can be assumed as 0.
- you are not allowed to change the structure of B.T.

Ex:-

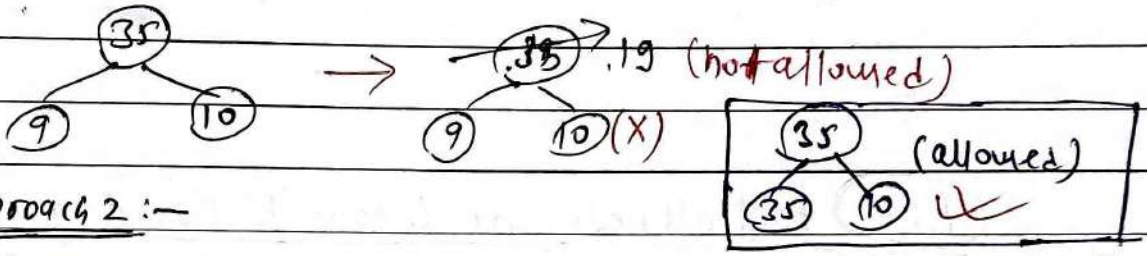


Approach 1 :- The first approach is that at every node make the node value equal to the sum of value(s) of its children. But it work in only that case "if children node sum is greater than node value!!

ex:-



- But "it will fail in that case if root node's value is greater than the sum of children node!! in such case we can not assign node value to children sum (because we are not allowed to decrement a value).



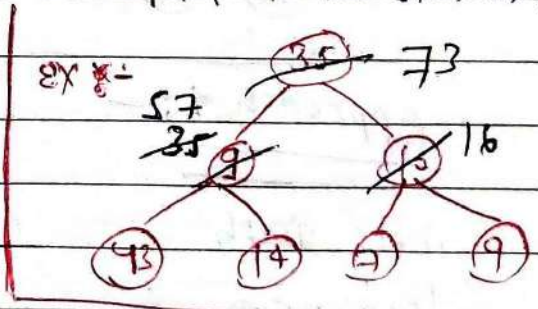
Approach 2 :-

- At every node, first we find the sum of children value (for a NULL child, value is assumed to be 0)
- if node's value > sum of children node value, we assign both the children's value to their parent's node value.
- Then we visit the children using recursion.
- After we return to the node after visiting the children, we explicitly set its value to be equal to the sum of its value of its children.

we can also assign either one

Code :-

```
void reorder (node * root){
    if (root == NULL) return;
    int child = 0;
    if (root -> right) child += root -> right -> data;
    if (root -> left) child += root -> left -> data;
```



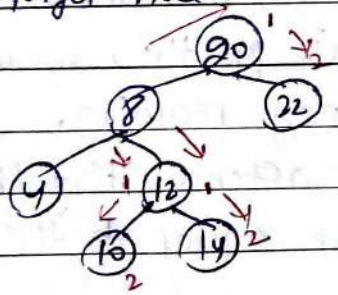
```
if ( child < root -> data){
   if (root -> left)  root -> left -> data = root -> data;
   else if (root -> right) root -> right -> data = root -> data;
}
reorder (root -> left);
reorder (root -> right);

int tot = 0;
if (root -> left) tot += root -> left -> data;
if (root -> right) tot += root -> right -> data;
if (root -> left || root -> right) root -> data = tot;
}
```

Time complexity : O(N)

Space complexity : O(N)

P:42 Print all nodes at distance 'k' from a given node.

Given a binary tree, a target node in the B.T, and a integer value k, print all the nodes that are at kth distance from target node.

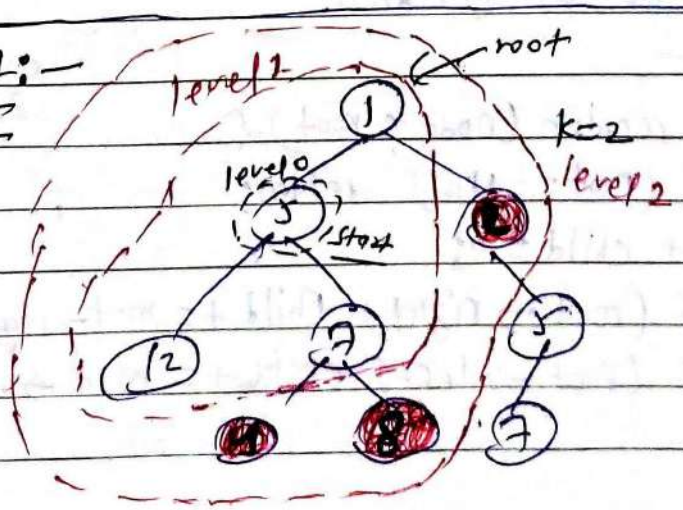


, if k=2, and target node = 8

output: 20, 14, 22

Approach 1:—

make graph then BFS



So, at level 2 will be our answers

step 2 making undirected :-

- so, it's look like a Breadth First search, But in the tree case we have given only one direction root to child then how we can go upward. for this first we make a datastructure which will store the parent node
- what can be that data structure so that we can look up the parent in o(1) time. so it can be Hash-table.
- so by using this we are making here directed tree or undirected graph. so that we can easily do breadth First search (BFS)

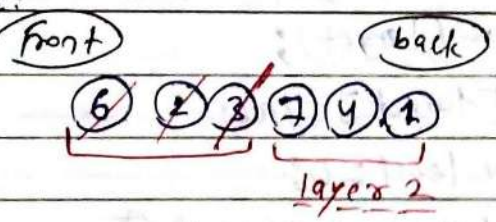
step 3 BFS :-

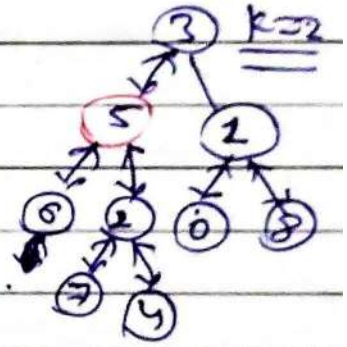
- at every node we push the left, right, and parent to the queue. before insert we check if we have already seen this or not, if not seen then push/insert

current level = ~~0 1~~ 2

seen of (5), (6), (2), (3), (7), (4)

queue:





complexity :- BFS = (m+n) , ∵ in binary tree

Time = (m+n) (m = no. of edge, n = no. of node)

m = n-2

∴ = (n-1+n) = (2n-1) = O(n)

space = O(n)

code :-

```
void markParents(TreeNode* root, unordered_map<TreeNode*,
                 TreeNode*>& parent_track, TreeNode* target){
    queue<TreeNode*> queue;
    queue.push(root);
    while(!queue.empty()){
```

```
        TreeNode* current = queue.front();
        queue.pop();
        if (current->left){
            parent_track[current->left] = current;
            queue.push(current->left);
        }
        if (current->right){
            parent_track[current->right] = current;
            queue.push(current->right);
        }
    }
}
vector<int> distanceK(TreeNode* root, TreeNode* target, int k){
    unordered_map<TreeNode*, TreeNode*> parent_track;
    markParents(root, parent_track, target);

    unordered_map<TreeNode*, bool> visited;
    queue<TreeNode*> queue;
    queue.push(target);
    visited[target] = true;
    int curr_level = 0;
    while(!queue.empty()){
        int size = queue.size();
        if(curr_level++ == k) break;
        for(int i=0; i<size; i++){
            TreeNode* current = queue.front(); queue.pop();
            if(current->left && !visited[current->left]){
                queue.push(current->left);
                visited[current->left] = true;
            }
            if(current->right && !visited[current->right]){
                queue.push(current->right);
                visited[current->right] = true;
            }
```

```
if (parent_track[current] && !visited[parent_track[current]]){
    queue.push(parent_track[current]);
    visited[parent_track[current]] = true;
}
}

vector<int> result;
while (!queue.empty()){
    TreeNode* current = queue.front(); queue.pop();
    result.push_back(current->val);
}
return result;
}
```

Approach 2 :- Without B.F.S

There are two types of nodes to be considered.

1) Nodes in the subtree rooted with target node.
2) other nodes, may be an ancestor of target, or a node in some other subtree.

- Finding the first type of node is easy to implement. Just traverse subtrees rooted with the target node and decrement k in recursive call. When the k becomes 0, print the node currently being traversed and return from there.
- For the second case, output node not lying in the subtree with the target node or the root, we must go through all ancestor. For every ancestor, we find its distance from target node, let the distance be 'd', now we go to other subtree (if target was found in left subtree, then we go to right subtree & vice versa) of the ancestor and find all nodes at (k-d) distance from the ancestor.

code :-

```
void printkdistanceNodeDown(node* root, int k)
{ if(root==NULL || k<0) return;
  if(k==0){
      cout<<root->data<<endl;
      return;
  }
  printkdistanceNodeDown(root->left, k-1);
  printkdistanceNodeDown(root->right, k-1);
}
```

Print all node at distance 'k' from root node

```
int printkdistanceNode(node* root, node* target, int k){
  if(root==NULL) return -1; //if target not present
  if(root==target){
      printkdistanceNodeDown(root, k);
      return 0;
  }
```

// case 1: all node at distance k rooted with target

```
  int dl = printkdistanceNode(root->left, target, k);
  if(dl!=-1){
      if(dl+1==k) //if root is at distance k
          cout<<root->Data<<endl;
      else //else go to right subtree & print node at k-dl-2
          printkdistanceNodeDown(root->right, k-dl-2);
      return dl+1; //add 1 to the distance and return
                   //value for parents call
  }
```

if target node was found in left subtree

here dist is k-dl-2 bcz right child is 2 dege away from left child

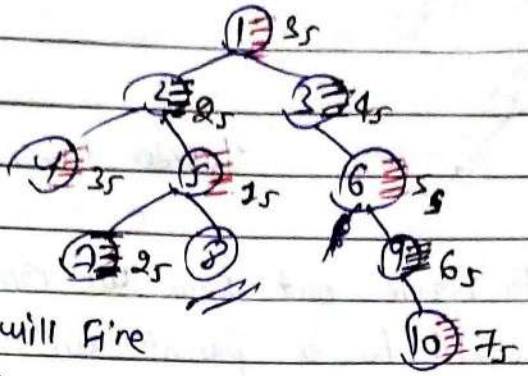
```
  int dr = printkdistanceNode(root->right, target, k);
  if(dr!=-1){
      if(dr+1==k) cout<<root->data<<endl;
      else
          printkdistanceNodeDown(root->left, k-dr-2);
      return 1+dr;
  }
  return -1;
}
```

we reach here only if target node was not found in left subtree

Time: O(h) (single traversal)
Space: O(h) //recursion

(P: 43) Minimum time to burn binary tree from a given node

ex:-



Target Node = 8.

at 1 sec = 5 will fire
at 2 " = 2 & 7
at 3 " = 4 & 1
at 4 " = 3
at 5 " = 6
at 6 " = 9
at 7 " = 10

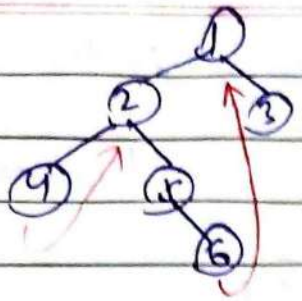
∴ output = 7

- we can solve this problem by doing ~~b~~readth first search. we can use same process as follow to solve this problem: 42 (Print all nodes at distance 'k').
- Make a graph (using hashtable for upward movement) and then do BFS.
- Code:-

(P: 42) Approach: 1 :- There is little change in the (i.e. the BFS), level (Breadth) any node, code that at every ~~parent~~ we check if it is able to burn or not a node (it will burn a node if the left, right or up any one is previously not burned). then we increment curr_level. The final answer will be curr_level when queue will be empty.

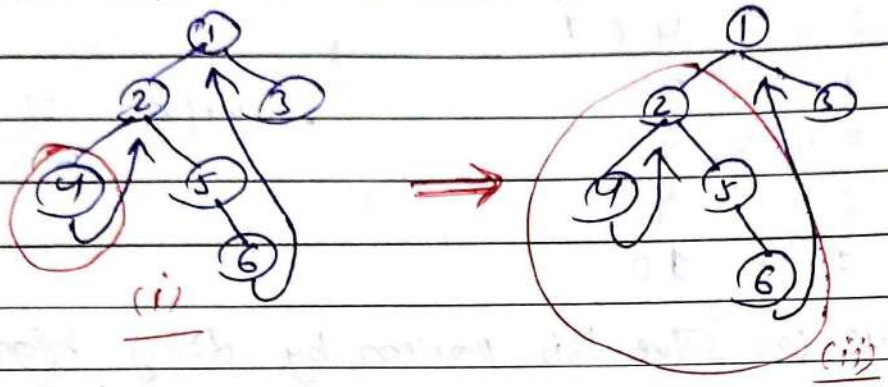
(P: 44) Morris Traversal :-

- This is traversal algorithm, without recursion, without stack.
- This take time complexity O(n) & space as O(1).



Inorder traversal :- 4, 2, 5, 6, 1, 3

- we need to figure out how we can go back in the tree from a child to a parent without using recursion. we break this tree into small sub-trees



- in fig(i) we can see that, we are at node '4', and at this node there is no right child, we move to the parent of this subtree, similarly in case (ii) at node '6'.

- So, we observe a pattern that whenever we are at last node of a subtree such that right child is pointing to none, we move to the parent of this subtree.

Algorithm :-

case1:- when current node has no left subtree. there is nothing to be traversed on the left side, so we simply print the value of current node and move to right of current.

case2:- when there is left subtree and the right-most child of this left subtree is pointing to null, in this case we need to set the right most to point to the current node, instead of null and move to the left of current node.

case3:- when there is a left subtree and the right most child of this left subtree is already pointing to the current node. In this case, we know left subtree is visited,

So, we print the current node, remove the link and goes to the right of the current node.

Code:—

```
vector<int> inorderTraversal (node * root){
    vector<int> inorder;
    node * cur = root;
    while (cur != NULL){
        if (cur->left == NULL){
            inorder.push_back (cur->data);
            cur = cur->right;
        }else{
            node *prev = cur->left;
            while (prev->right != NULL && prev->right != curr){
                prev = prev->right;
            }
            if (prev->right == NULL){
                prev->right = cur;
                cur = cur->left;
            }else{
                prev->right = NULL;
                inorder.push_back (cur->data);
                cur = cur->right;
            }
        }
    }
    return inorder;
}
```

* Morris traversal Preorder :-

Changing In Inorder morris traversal

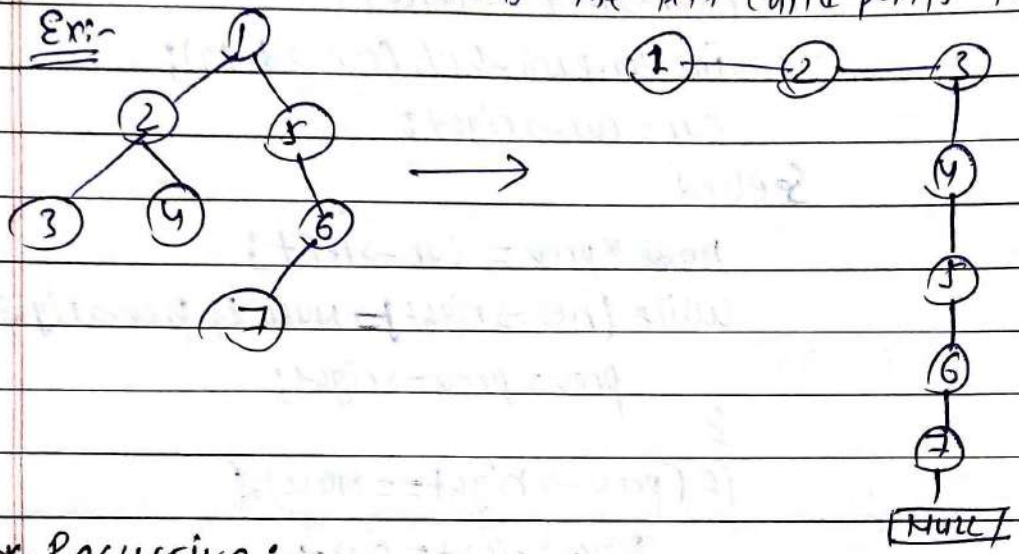
- In case 1: we print current node and then goes to right subtree
- In case 2: before moving to left of current node print current node
- In case 3: do not need to print current node, remove the link and goes to right of current node.

P:45 Flatten a Binary Tree to Linked list :-

Note:-

- The sequence of nodes in the linked list should be the same as that of the preorder
- The linked list nodes are the same binary tree nodes. you are not allowed to create extra nodes.
- The right child of a node points to the next node of the linked list whereas the left child points to NULL.

Ex:-



* Recursive:-

Approach:- • if we do a preorder traversal (root, left, right) A starting node left at (1) we point it right to (2) to its current left child. Now there is no way to reach node (3), (4), (5).

- so we need to modify our traversal technique. If we some-how start from its most right, we need not traverse it right child as it is NULL. then we set its right to left and its left to NULL
- i.e we will make linked list from end which is (7) in this case then we go to (6) → (5) and after (5) we need to go at the node (4) then (3) then (2) then (1) so, we can see that we need to follow- reverse postorder way : i.e right, left, root

Date
Page No. 255.

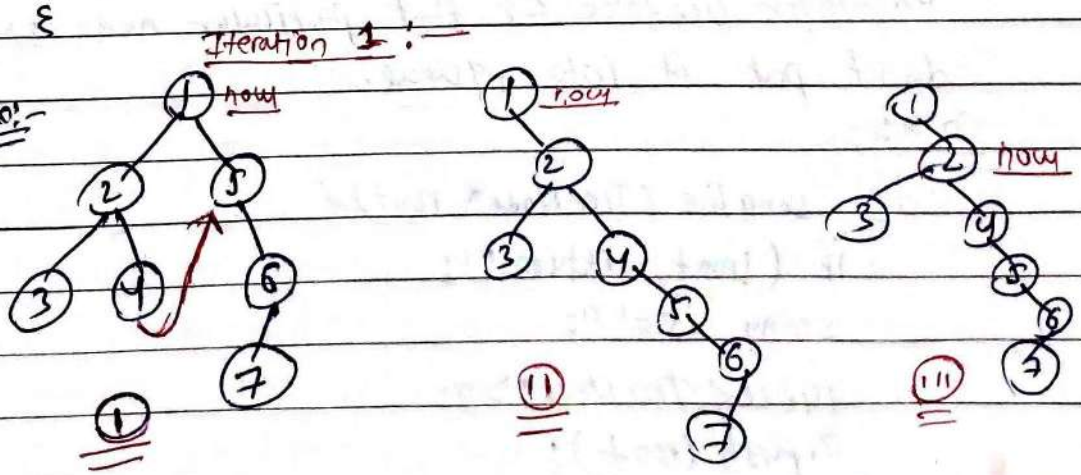
```
class Solution{
    Node* prev = NULL;
public:
    void flatten(Node* root){
        if(root==NULL) return;
        Flatten(root->right);
        Flatten(root->left);
        root->right = prev;
        root->left = NULL;
        prev = root;
    }
};
```

* Non-recursive :- like morris Traversal

```
void flatten(Node* root){
    Node* now = root;
    while(now){
        if(now->left){
            Node* pre = now->left;
            while(pre->right) pre = pre->right;
            pre->right = now->right;
            now->right = now->left;
            now->left = NULL;
        }
        now = now->right;
    }
}
```

(I) (II) (III)

ex:-

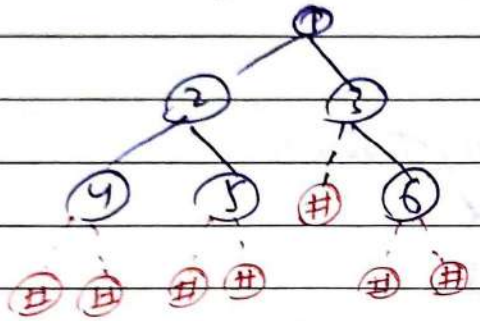


P:46 Serialize and Deserialize a Binary Tree.

- Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization / deserialization algorithm should work. you should just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree.

Approach:-

- there are lot of ways to solve this particular problem like inorder traversal, preorder traversal, post-order traversal,
- we will solving this by using level order traversal.



→ "1,2,3,4,5,#,6,#,#,#,#,#,#"

- If there is not a child for a node then we add a character which is not belong to node data, it is only for convenience to know it is NULL

Deserialize:-

now for deserialize we will take 1 from string and put it into a queue and make this as root. now next 2 will be left of 1 and also put it into queue, now next 3 is right of 1 and put into queue. now 1 is done we will take '2' and repeat same process. when '#' is encounter we take for that particular node as NULL and don't put it into queue.

Code:-

```
string serialize (TreeNode* root){
    if (!root) return "";
    string s="";
    queue<TreeNode*>q;
    q.push(root);
```

```
while (!q.empty()){
    TreeNode* curNode = q.front();
    q.pop();
    if (curNode == NULL) s.append("#,");
    else s.append(to_string(curNode->val) + ',');
    if (curNode != NULL){
        q.push(curNode->left);
        q.push(curNode->right);
    }
}
return s;
}
```

Time: O(n)
Space: O(n)

```
TreeNode* deserialize(string data){
    if(data.size()==0) return NULL;
    stringstream s(data);
    string str;
    getline(s, str, ',');
    TreeNode* root = new TreeNode(stoi(str));
    queue<TreeNode*> q;
    q.push(root);
    while(!q.empty()){
        TreeNode *node = q.front();
        q.pop();
        getline(s, str, ',');
        if(str=="#"){
            node->left = NULL;
        }else{
            TreeNode* leftNode = new TreeNode(stoi(str));
            node->left = leftNode;
            q.push(leftNode);
        }
        getline(s, str, ',');
        if(str=="#") node->right = NULL;
        else{
            TreeNode* rightNode = new TreeNode(stoi(str));
            node->right = rightNode;
            q.push(rightNode);
        }
    }
    return root;
}
```

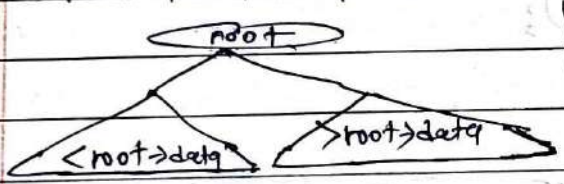
(5) Binary Search Trees (BSTs):-

* **Why BSTs:-**

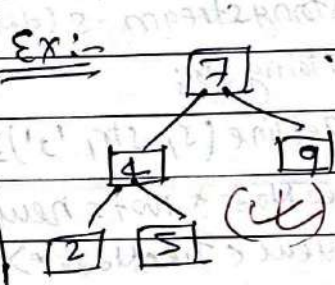
- in a tree to search for an element we need to check both in left subtree & in right subtree. Due to this time complexity is O(n)
- Binary Search Trees is another variation of Binary Trees. the main use of this representation is for searching. At every node in BST there is restriction on kind of data and we make only one subtree towards As a result it reduce time to O(logn) for search operation.

* **Binary Search Tree Property:-**

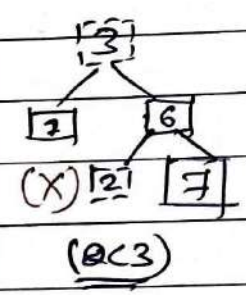
- The left subtree of a node contains key less than the key of node
- The right subtree of a node contains key greater than the key of node
- Both the left & right subtree must also have BSTs property.



Ex:-



(L)



(BST)

* **Declaration of BSTs :-**

it is same as Binary tree the only difference is that in data not in structure.

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

(6) Operations on Binary Search Trees.

* **main Operation:**

(i) Find / Find minm / Find maxm element in BSTs
(ii) Inserting an element in BST.
(iii) Deleting an element from BSTs

* Important point:-

- since root data is always b/w left subtree data and right subtree data, performing inorder traversal on BST produces a sorted list.
- while inserting/deleting an element in BST height of BST will be change and time complexity in best, average, worst case will change.
- $T \propto h$, where $h = (\log n)$, For a complete B.T node(n)
- if tree is linear chain of 'n' nodes (Skew-tree), same will take O(n)

(1)* Finding an Element in B.S.T

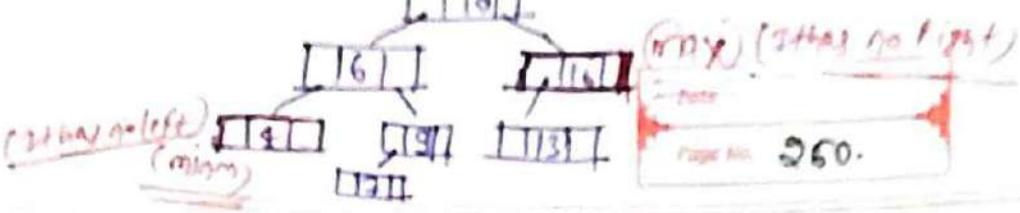
```
struct BSTNode * find (struct BSTNode* root, int data){
    if(root==NULL) return NULL;
    if (data < root->data) return find (root->left, data);
    else if (data>root->data) return find (root->right, data);
    return root;
}
```

Time Complexity: O(logn), Worst case: O(n) (Skew tree)
Space Complexity: O(logn) " "

* Non-Recursive :-

```
struct BSTNode* find (struct BSTNode * root, int data){
    if(root==NULL) return NULL;
    while (root){
        if (data==root->data) return data;
        else if (data>root->data) root = root->right;
        else root = root->left;
    }
    return NULL;
}
```

Same time as Recursive / Space: O(1)



* Find minm (left most node)

```
struct node* FindMin(node* root){
    if(root==NULL) return NULL;
    else if(root->left==NULL) return root;
    else return FindMin(root->left);
}
```

Non-recursive

```
Struct node* FindMin(node* root)
{ if(!root) return NULL;
  while(root->left!=NULL)
      root=root->left;
  return root;
}
```

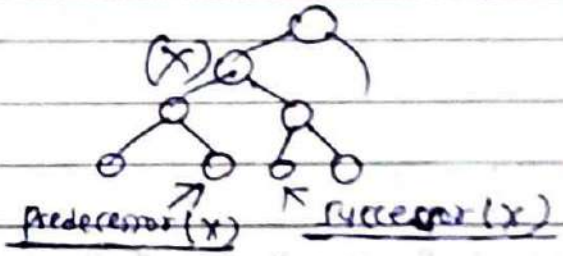
* Find Maxm (Right most node)

```
struct node* FindMax(node* root){
    if(root==NULL) return NULL;
    else if(root->right==NULL) return root;
    else return FindMax(root->right);
}
```

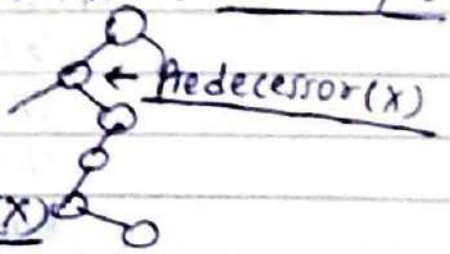
```
while(root->right!=NULL)
    root=root->right;
```

(ii) Where is Inorder predecessor and Successor :-

- If a node x has two children then its inorder predecessor is the maximum value in its left subtree and its inorder successor is the minimum value in its right subtree

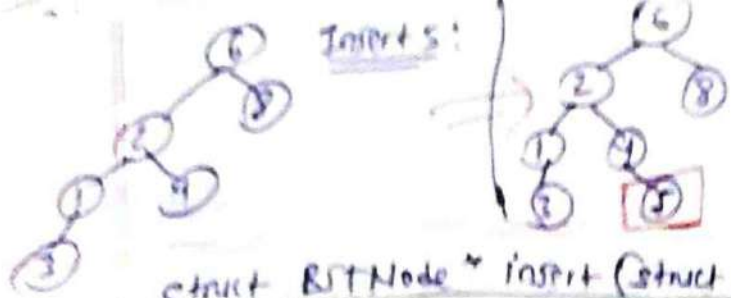


- If it does not have a left child, then a node's inorder predecessor is its first left ancestor.



(iii) Inserting an Element ~~from~~ in Binary Search Tree

- For insertion first we need to find the location for that we can use same find operation. when finding the location if data is already there then simply neglect & come out else insert data at last location path traversed




```
struct BSTNode * insert (struct BSTNode * root, int data) {
    if (root == NULL) {
        root = (struct BSTNode *) malloc (sizeof (struct BSTNode));
        if (root == NULL) {
            printf ("Memory Error"); return;
        }
        else {
            root -> data = data;
            root -> left = root -> right = NULL;
        } }
    else {
        if (data < root -> data) root -> left = insert (root -> left, data);
        else if (data > root -> data) root -> right = insert (root -> right, data);
    }
    return root;
}
```

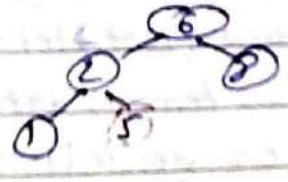
Time: O(logn) for complete B.S.T , worst case: O(n)

space: O(logn) / O(n) (recursive stack)

(iv) Deleting an element from BST

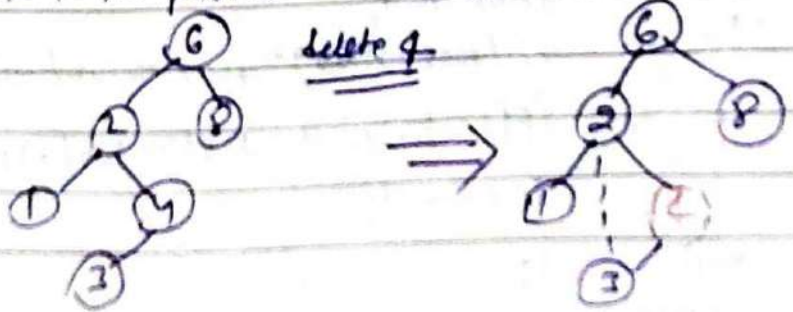
- It is complicated than other operation. bcz the element to be deleted may not be the leaf node. To this first we need to find the location of element which we want to delete.
- Once location is found consider following cases: if element

(i) if leaf node:- make the corresponding child pointer NULL.



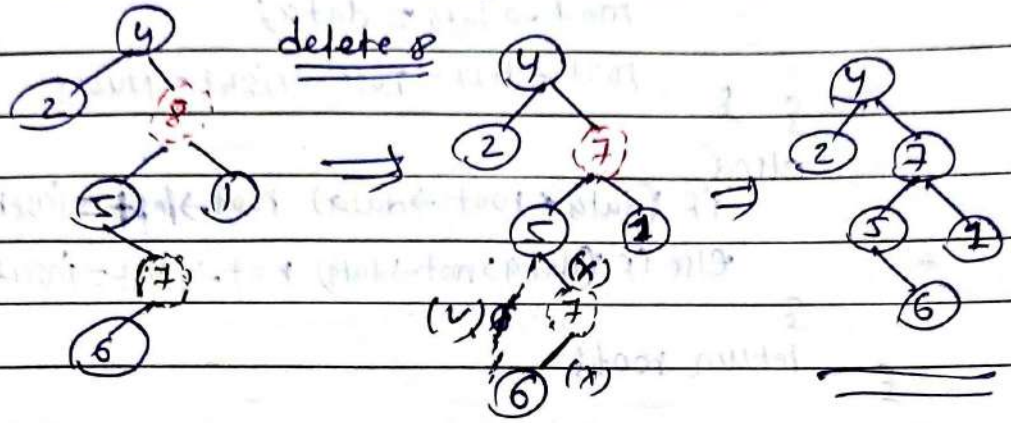
here to delete 5 make 2's child to NULL.

(ii) Element has one child:- Just need to send the current node's child to its parent.



(iii) element has both children :- The general strategy is to replace the key of this node with the largest element of the left subtree and recursively delete that node (which is now empty).

Note:- The largest node in left subtree can't have right child, so we can delete it as easy.



Note:- we can replace element with minm element in right also.

```
struct BSTNode* delete(struct BSTNode* root, int data){
    struct BSTNode *temp;
    if (root==NULL)
        printf("Element is not there");
    else if (data < root->data) root->left=delet(root->left,data);
    else if (data > root->data) root->right= delete(root->right,data);
    else{
        if(root->left && root right){
            temp= findmax(root->left);
            root->data=temp->data;
            root->left = delete(root->left, root->data);
        }
        else{
            temp=root;
            if(root->left) root=root->left;
            if(root->right) root=root->right;
            free(temp);
        }}
    return root;
}
```

(7) BST Problems & solution

P:1 Find LCA of two nodes in BST.

Approach :- while traversing BST, the first node encounter with value b/w α and β, is the LCA of α and β,

- if the value is greater than both α and β then LCA lies on left subtree of the node, and
- if value is smaller than both α and β, then the LCA lies on the right side.

```
struct BSTNode* FindLCA(BSTNode* root, BSTNode* α,
                                       BSTNode* β){
    while(1){
        if ((α->data < root->data && β->data > root->data) ||
            (α->data > root->data && β->data < root->data))
            return root;
        if (α->data < root->data) root = root->left;
        else root = root->right;
    }
}
```

Time : O(h), For Skew trees

P:2 shortest path b/w two nodes in a BST

It is nothing but finding the LCA of two nodes in BST.

P:3 Check is a BT is BST or not

For each node, check if the node on its is smaller and check if the node on its right is greater.

(1)

```
int isBST(struct BTNode* root){
    if(!root) return 1;
    if(root->left && root->left->data > root->data)
        return 0;
    if(root->right && root->right->data < root->data)
        return 0;
    if(!isBST(root->left) || !isBST(root->right))
        return 0;
    return 1;
}
```

(X) not give correct answer

Note:- this above algorithm is wrong, it will check only at current node is not enough.

(ii) For each node, check if max value in left is smaller than the current node data and minm value in right subtree is greater than the node data.

Ex:- Change in above code for checking left & right :-

```
if (root->left && findMax(root->left) > root->data)
    return 0;
if (root->right && findMin(root->right > root->data)
    return 0;
```

Time: O(n²)

(iii) Improve above time complexity :-

- The trick is to write a utility helperfunction isBSTUtil(node*, root, intmin, int max) that traverse down the tree keeping track of the narrowing min and max allowed value as it goes, looking at each node only once.

initial call: isBST(root, INT_MIN, INT_MAX);

```
int isBST(struct BinaryTreeNode* root, int min, int max){
    if(!root) return 1;
    return (root->data > min && root->data < max) && isBST
            (root->left, min, root->data) && isBST
            (root->right, root->data, max);
}
```

Time: O(n)

(iv) Improve above algorithm :-

we know that inorder traversal on BST is sorted list, so we traverse inorder on the BST and check if prev node data is less than the current node.

Code:- int prev = INT_MIN;

Time: O(n), Space: O(1)

```
int isBST(struct BinaryTreeNode *root, int *prev){
    if(!root) return 1;
    if(!isBST(root->left, prev)) return 0;
    if(root->data < *prev) return 0;
    *prev = root->data;
    return isBST(root->right, prev);
}
```

P:4 Convert a sorted array to BST.

• The middle element of an array will be root of BST and again left of tree & left part of array will be same problem. similarly right part of tree & right of array from middle will be same sub problem.

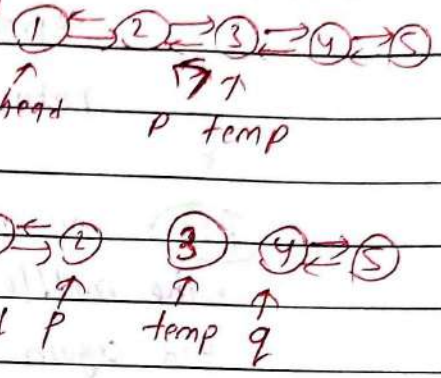
```
struct node* buildBST(int A[], int left, int right){
    struct node* newNode;
    int mid;
    if(left>right) return NULL;
    newNode = (struct node*) malloc(sizeof(struct node*));
    if(left==right){
        newNode->data = A[left];
        newNode->left = newNode->right = NULL;
    }else{
        mid = left+(right-left)/2;
        newNode->data = A[mid];
        newNode->left = buildBST(A, left, mid-1);
        newNode->right = buildBST(A, mid+1, right);
    }
    return newNode;
}
```

Time & Space O(n)

P:5 convert a sorted DLL to BST

Approach: Find the middle node and adjust the pointer.

```
struct Lnode* DLLToBST (struct Lnode* head){
    struct LNode *temp, *p, *q;
    if (!head || !head->next) return head;
    temp = findMiddleNode(head);
    p=head;
    while (p->next != temp)
        p=p->next;
    p->next = NULL;
    q = temp->next;
    temp->next = NULL;
    temp->prev = DLLToBST(head);
    temp->next = DLLToBST(q);
    return temp;
}
```



Time = O(n log n)
Find mid → Binary traversal.
for each traversal

Note:- if we have given LL or DLL and we have to make a tree then we can convert it into a list and the we can find middle in constant time.

P:6 ~~Convert BST to circular DLL~~

Find kth smallest element in BST.

- The idea behind the solution is that, inorder traversal of BST produces sorted lists. while traverring the BST in inorder, keep track of the number of element visited.

⇒
```
struct BSTNode * kthSmallestBST (struct BSTNode* root, int k,
                                 int *count){
    if (!root) return NULL;
    struct BSTNode *left = kthSmallestBST(root->left, k, count);
    if (left) return left;
```

```
if (++count == k)· return root;
return kthSmallestBST (root->right, k, count);
}
```

Time : O(n) / Space : O(n)

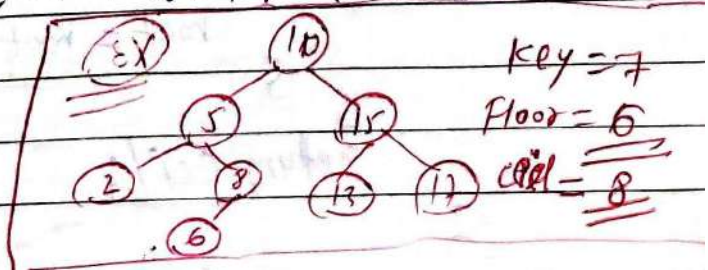
Q.P:7 Floor and ceiling :-

- Floor is the largest key in the BST (which is) less than or equal to the given key.
- ceiling is the smallest key in the BST which is greater than the given key.

→ If given key is less than the key at the root of a BST then the Floor of the key must be in the left subtree. If the key is greater than the key at the root, the Floor of the key could be in right subtree; if not (or if the key is equal to the key at the root) then the key at the root is the Floor of the key.

*Floor :- Recursive

```
struct node* Floor (struct node* root, int data){
    struct node* prev = NULL;
    return FloorUtil(root, prev, data);
}
struct node* FloorUtil (struct node* root, struct node* prev, int data){
    if (!root) return NULL;
struct node* abs = FloorUtil (root->left, prev, data);
if (abs) return abs;
    if (root->data == data) return root;
    if (root->data > data) return prev;
    prev = root
    return FloorUtil(root->right, prev, data);
}
```



* Floor: Non-recursive

```
int Floor(node<int>* root, int key){
    int floor = -1;
    while (root){
        if (root->val == key){
            floor = root->val;
            return floor;
        if (key > root->val){
            floor = root->val;
            root = root->right;
        }
        else{
            root = root->left;
        }
    }
    return floor;
}
```

* Ceil: Non-recursive

```
int Ciel(node<int>* root, int key){
    int ciel = -1;
    while (root){
        if (root->data == key){
            ceil = root->data;
            return ceil;
        }
        if (key > root->data){
            root = root->right;
        }
        else{
            ceil = root->data;
            root = root->left;
        }
    }
    return ceil;
}
```

(P:8) Find the union and intersection of BSTs.

- Perform inorder traversal on one of the BSTs.
- While performing the traversal store them in hash-table.
- After completion of the traversal of first BST, start traversal of 2nd BST and compare them with hash-table content.

Time: O(m+n), space O(max(m,n)).

(P:9) Given a BST and two no. k1 and k2, print all element of BST in rage k1 and k2.

```
void rangePrinter (struct node* root, int k1, int k2){
    if (!root) return;
    if (root->data >= k1) rangePrinter (root->left, k1, k2);
    if (root->data >= k1 && root->data <= k2)
        printf ("%d", root->data);
    if (root->data <= k2) rangePrinter (root->right, k1, k2);
}
```

Time: O(n) | Space O(n)

Another Approach:

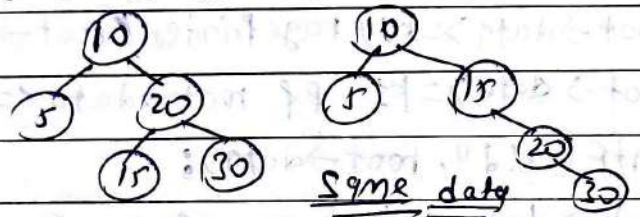
We can use level order traversal: while adding the element to queue check for the range.

```
void rangePrinter(struct node* root, int k1, int k2){
    struct node* temp;
    struct Queue *Q = createQueue();
    if (!root) return NULL;
    Q = enQueue(Q, root);
    while (!isEmpty(Q)){
        temp = deQueue(Q);
⇒       if (temp->data >= k1 && temp->data <= k2)
            printf ("%d", temp->data);
⇒       if (temp->left && temp->data >= k1) enQueue(Q, temp->left);
⇒       if (temp->right && temp->data <= k2) enQueue(Q, temp->right);
    }
}
```

Another Approach:
First locate k2 with normal Binary search and after that use inorder successor until we encounter k2. to solve this we can follow threaded Binary Tree property.

Alternative Question:. Given root of Binary Tree, trim the tree, so that all elements returned in the new tree are b/w the inputs A & B.

P:10 Given two BSTs, check whether the elements of them are the same or not



Same data

i.e BSTs data can be in any order

1st Approach:- perform inorder on first BST store element in hash table, again perform inorder on 2nd BST and check whether it is present or not in Hash table.

Time: O(max(m,n))
Space: O(max(m,n))

2nd Approach:- Instead of performing traversal one after another, we can perform in-order traversal of both tree in parallel. since the in-order traversal gives the sorted list, we can check whether both the trees or generating same sequence or not.

Time: O(max(m,n))
Space: O(1)

P:11 For the key-values 1...n, how many structurally unique BSTs are possible

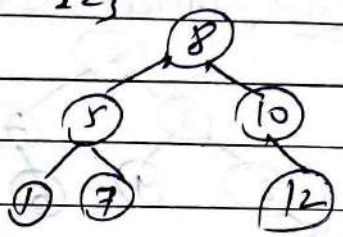
```
int countTrees(int n){
    if (n<=1) return 1;
    else{ int sum=0,left,right,root;
        for (root=1; root<=n; root++){
```

```
        left = countTrees(root - 1);
        right = countTrees(n - root);
        sum += left * right;
    }
    return sum;
}
```

(P.12) Construct BST from preorder traversal

preorder → {8 5 1 7 10 12}

(root, left, right)



Approach 1:- Take a element '8' and make this root now we know right of preorder traversal from '8' can be left subtree or right subtree so we will take every element and check if it is less then goes to left of root, and if greater then goes to right of root, it will take total $O(n^2)$ time.

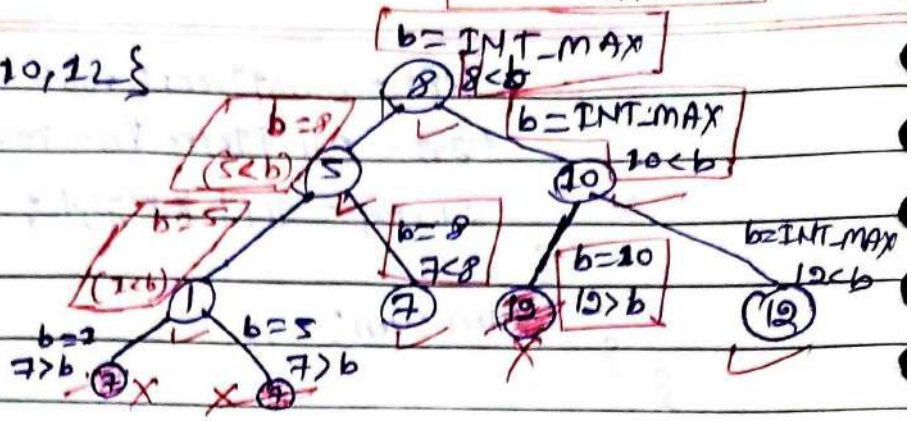
Approach 2:- since we have preorder traversal, so by sorting this we can get in-order traversal and now we can generate our tree by using both traversal easily. time complexity: $O(n \log n)$

Approach 3:- we will follow Approach '1' but we have called a upper bound for the every element so that we can make child or, not of a specific root

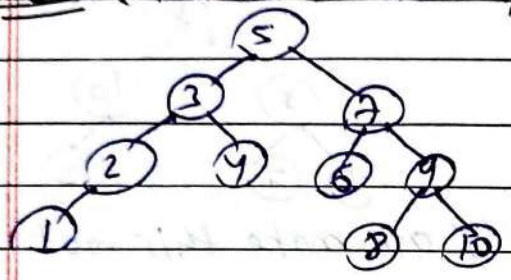
time complexity = $O(n)$

```
⇒ TreeNode* bstFromPreorder(vector<int>& A) {
    return build(A, 0, INT_MAX);
}
TreeNode* build(vector<int>& A, int& i, int bound) {
    if(i == A.size() || A[i] > bound) return NULL;
    TreeNode *root = new TreeNode(A[i++]);
    root->left = build(A, i, root->val);
    root->right = build(A, i, bound);
    return root;
}
```

preorder = {8, 5, 1, 7, 10, 12}



P:13 InOrder Successor/Predecessor in BST



Inorder: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Val = 8	Val = 10
Inorder Successor = 9	IS = (X) NULL
Inorder Predecessor = 7	IP = 9

Approach 1: Apply Inorder traversal the element which is first greater than given value that will be our required inorder successor.

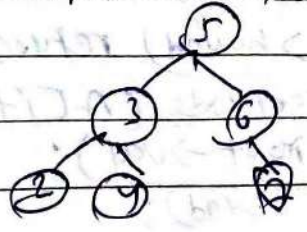
Time: O(n) | space: O(n)

Approach 2:- Applying binary search.

```
TreeNode * inorderSuccessor(TreeNode* root, TreeNode * p){
    TreeNode * successor = NULL;
    while (root != NULL){
        if (p->val >= root->val)
            root = root->right
        else{
            successor = root;
            root = root->left;
        }
    }
    return successor;
}
```

Time: O(H) | space: O(1) (H = height)

P:14 Two Sum in (BST) + (using BST iterator)



k = 9

Inorder = 2 3 4 5 6 7

Approach 1: Store inorder in an array, takes that two number from first & last number. if sum is is greater move towards left (right pointer) and if sum is less then move left pointer to right

Time: O(n) / space: O(n)

Approach 2: we have two iterator next() and before() next will give next element before will give before element

Time: O(H) / SC: O(H) x 2

⇒ Next() & Before() iterator class

```
class BSTIterator{
        stack <TreeNode*> myStack;
        bool reverse = true;        // reverse=true ⇒ before
                                    // reverse = false ⇒ next
public:
    BSTIterator(TreeNode* root, bool isReverse){
            reverse = isReverse;
            pushAll(root);
    }
    bool hasNext(){
        return !myStack.empty();
    }
    int next(){
        TreeNode *tmpNode = myStack.top();
        myStack.pop();
        if(!reverse) pushAll(tmpNode->right);
        else pushAll(tmpNode->left);
        return tmpNode->val;
    }
private:
    void pushAll(TreeNode * node){
        for(; node!=NULL;){
            myStack.push(node);
            if(reverse==true) node = node->right;
            else node = node->left;
        }
    }
};
```

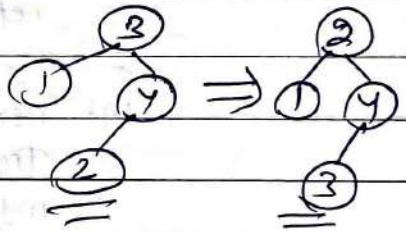
```
class Solution{
public:
    bool FindTarget(TreeNode* root, int k){
        if(!root) return false;
        BSTIterator l(root, false);
        BSTIterator r(root, true);
        int i = l.next();
        int j = r.next();
        while(i<j){
            if(i+j==k) return true;
            else if(i+j<k) i=l.next();
            else j=r.next();
        }
        return false;
    }
};
```

P:15 BST Iterator

- In previous question already explained.

P:16 Recover BST :.

- you have given root of BST in which two nodes are swaped, recover original BST.



Brute force:- (i) do inorder traversal and store them
(ii) sort the element
(iii) check Inorder traversal with sort element. and make changes if required.

T.C: $2n + n\log n$
S.C: $O(n)$

Efficient Approach:-

(I) Swapped nodes are not adjacent:- 3 [25] 7 8 10 15 20 [5]
(1st = 25, 2nd = 7, middle (x), 2nd = 5)

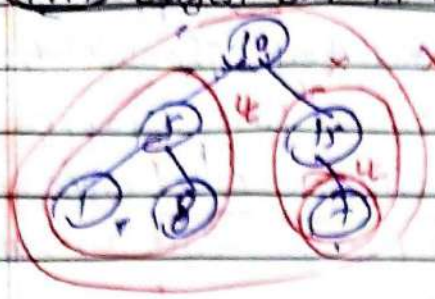
(II) Swapped nodes are adjacent:- 3 5 [8] [7] 10 15 20 25
(1st = 8, 2nd = 7, middle)

• since there are only two nodes are swapped in sorted. so we can do a traversal and check the violation of sorting, if we find mark them as 'first swapped node' and the next node as 'second swapped node' again start from there if any other node are present which violate the sorting that will be '2nd swapped node'

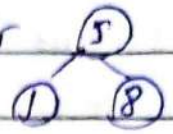
~~T.C → O(N) | S.C → O(1) (by using Morris traversal)~~
~~else → O(N)~~

```
class Solution {
private:
    TreeNode * first, * prev, * middle, * last;
private:
    void inorder (TreeNode * root) {
        if (root == NULL) return;
        inorder (root -> left);
        if (prev != NULL && root -> val < prev -> val) {
            if (first == NULL) {
                first = prev;
                middle = root;
            }
            else
                last = root;
        }
        prev = root;        //mark this node as previous
        inorder (root -> right);
    }
public:
    void recoverTree (TreeNode * root) {
        first = middle = last = NULL;
        prev = new TreeNode (INT_MIN);
        inorder (root);
        if (first && last) swap (first -> val, last -> val);
        else if (first && middle) swap (first -> val, middle -> val);
    }
};
```

P:17 Largest BST in Binary Tree



so, largest BST is

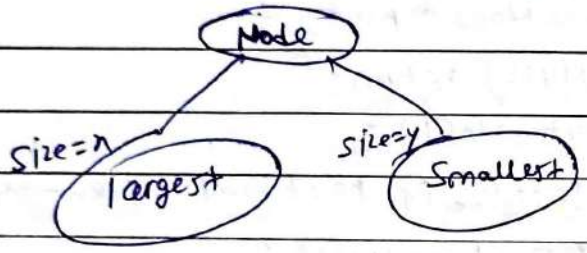


Brute Force :- take every node and check the subtree by making that node as root is BST or not.

Time: $O(N) * O(N) = O(N^2)$

$O(N)$ ← Traverse every node; $O(N)$ → for checking BST

Efficient Approach :-

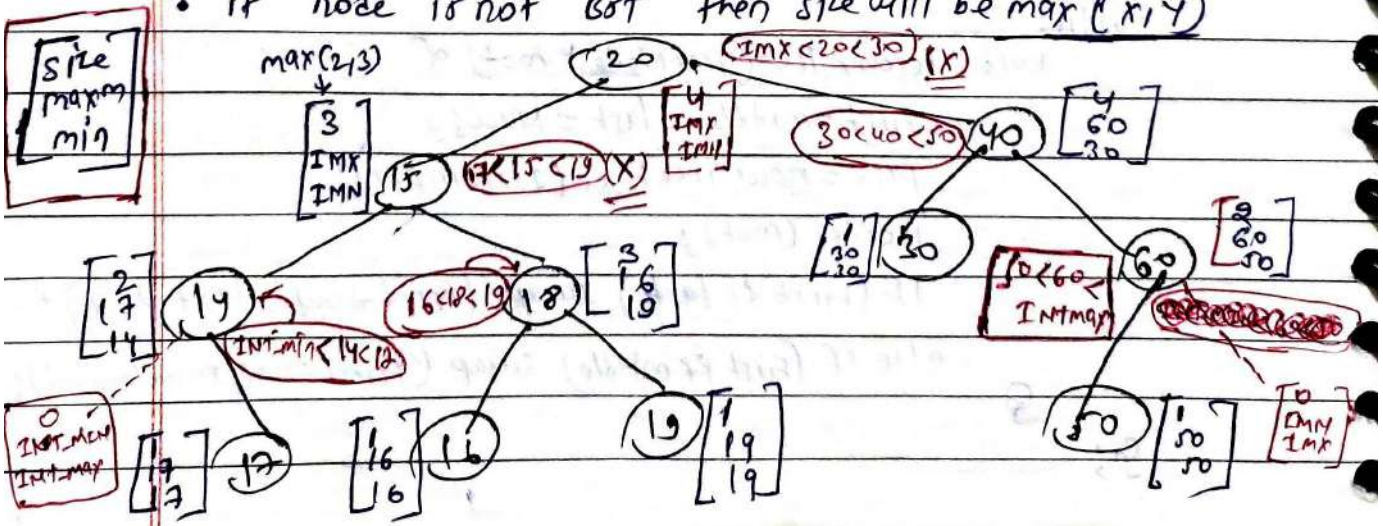


if, largest < node < smallest then this is valid BST and size = $1+x+y$

Note:-
- Since we have to check also largest & smallest (left & right) subtree should also be BST.

If it is not a BST then it will contain very large value (INT_MAX) as largest in left so that node cannot BST. and in right it will contain very minm value INT_MIN so that node can not BST.

- here x and y is largest BST size in left & right side
- if node is not BST then size will be max (x, y)



```
class NodeValue{
public:
    int maxNode, minNode, maxSize;
    NodeValue(int minNode, int maxNode, int maxSize){
        this->maxNode = maxNode;
        this->minNode = minNode;
        this->maxSize = maxSize;
    }
};
class Solution{
private:
    NodeValue largestBSTHelper(TreeNode* root){
        if(!root){
            return NodeValue(INT_MAX, INT_MIN, 0);
        }
        auto left = largestBSTHelper(root->left);
        auto right = largestBSTHelper(root->right);
        if(left.maxNode < root->val && root->val < right.minNode){
            return NodeValue(min(root->val, left.minNode),
                             max(root->val, right.maxNode),
                             left.maxSize + right.maxSize + 1);
        }
        return NodeValue(INT_MIN, INT_MAX, max(left.maxSize,
                                               right.maxSize));
    }
public:
    int largestBST(TreeNode* root){
        return largestBSTHelper(root).maxSize;
    }
};
```

using post-order

if current node is greater than node in left and smaller than node in right then it is BST

otherwise, return [-∞, +∞], so that parent can't be valid BST

T.C: O(N)
S.P: O(N)

Date
Page No. 28

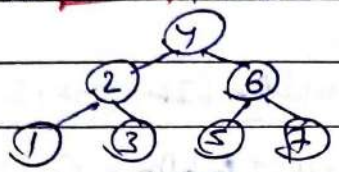
(8) Balanced Binary Search Tree

- if a Binary search Tree is skew then it will take O(n) time (n = no. of nodes). So to reduce it to O(log n) we can impose restriction on heights.
- In general, the height balanced trees are represented with HB(k), where k is different b/w left subtree height & right subtree height, sometimes [k] is called 'balance factor'.

* Full Balanced BSTs

In HB(k), if k=0, then it is called Full balanced BST.

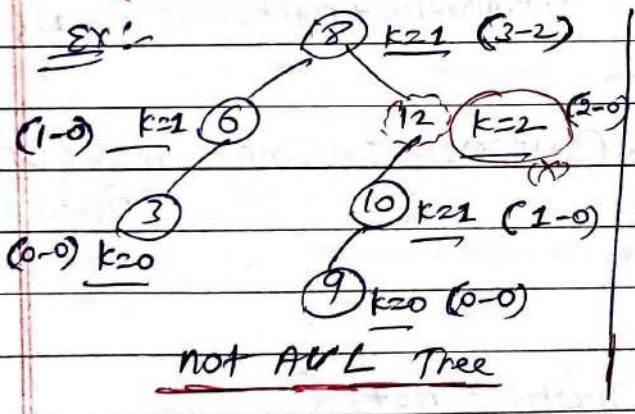
Ex:-



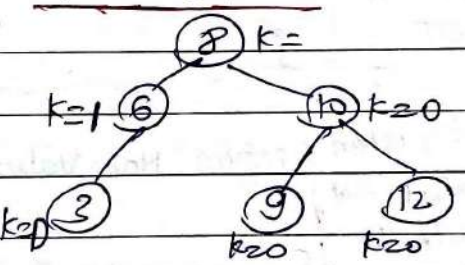
(9) AVL (Adelson-Velskii Landis) Tree

- if in HB(k), if k=1 (balance factor is one), such BST is called an AVL tree.
- For a AVL the difference b/w left subtree height & right subtree height of every node of a BST is at most '1'

Ex:-



not AVL Tree



AVL tree

(*) minimum/maximum no. of nodes in AVL tree.

- if h = height, and N(h) = no. of nodes in AVL with height h
- To get minm no. of nodes at height 'h', we should fill left subtree with height 'h-1' & right subtree with height 'h-2'

$$\therefore N(h) = \underbrace{N(h-1)}_{\text{minm no. of nodes with height } (h-1)} + N(h-2) + 1 \leftarrow \text{current node}$$

minm no. of nodes with height (h-2)

Note:- we can give $N(h-2)$ either for left subtree or right subtree.

by solving above recurrence its gives:-

$$N(h) = O(1.618^h)$$ min^m node in BST

Or, $h = 1.44 \log n \approx O(\log n)$

- It also says that the max^m height in AVL is $O(\log n)$

* Similarly for max^m no. of nodes we need to fill left & right subtree with height $(h-1)$

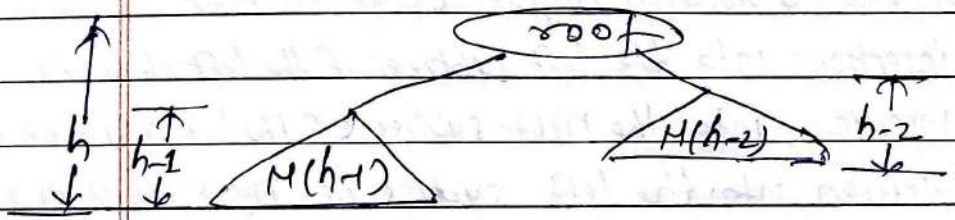
$$N(h) = N(h-1) + N(h-1) + 1 = 2N(h-1) + 1$$

by solving recurrence we get.

$$N(h) = O(2^h)$$ max^m node in BST

Or, $h = \log n \approx O(\log n)$

Note In both case height of AVL tree is $O(\log n)$



⊛ AVL Tree Declaration

- it is same as BST But to simplify the operation, we also include the height.

```
struct AVLTreeNode{
        struct AVLTreeNode *left;
        int data;
        struct AVLTreeNode *right;
        int height;
};
```

⊛ Rotations:-

- When the tree structure changes (e.g., with insertion or deletion), we need to modify the tree to restore the AVL tree property. This can be done using 'single rotations or double

since insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of a subtree by 1.

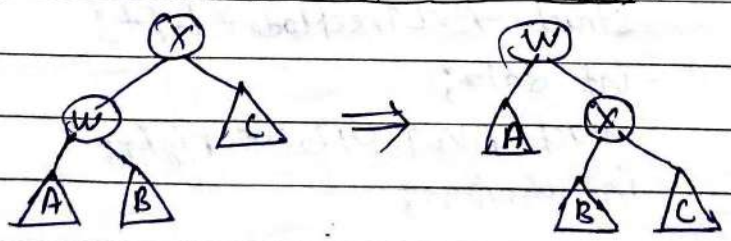
→ observation:- after an insertion, only node that are on
• path from the insertion to the root might have their balances altered. to restore the AVL property, we start at the insertion point and keep going to the root of tree. we need to consider the first node that is not satisfying the AVL property. From that node onwards, every node on the path to the root will have the issue.
• Also, if we fix the issue for that first node, then all other nodes on path to the root will automatically satisfy property.

✱ Types of Violations:-

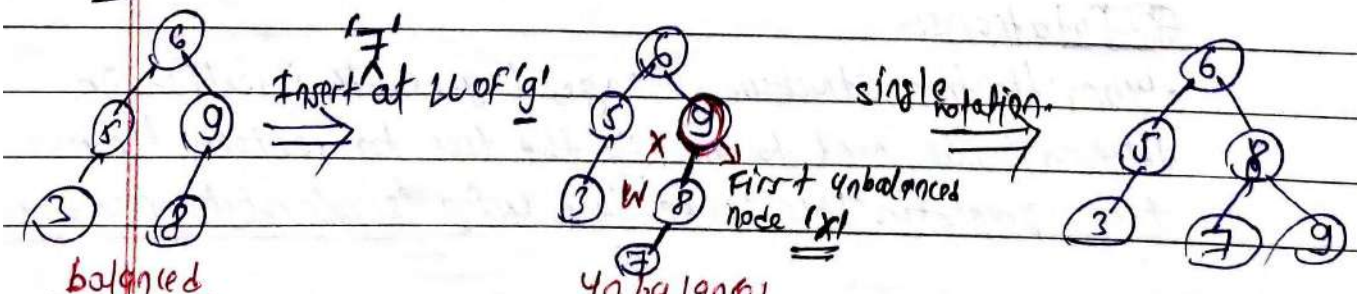
• let's 'x' is to be rebalanced. since any node at most two children, and hight for imbalance is differ by two, so we can observe that a violation might occur in four cases:
1. An insertion into the left subtree of the left child of x.
2. An insertion into the right subtree of the left child of x.
3. An insertion into the left subtree of right child of x.
4. An insertion into the right subtree of right child of x.
• Cases 1 and 4 are symmetric and easily solved with single rotations. similary 2 and 4. can be solve with double. rotation.

✱ Single Rotations:-

Case 1:- Left Left Rotation (LL Rotation):



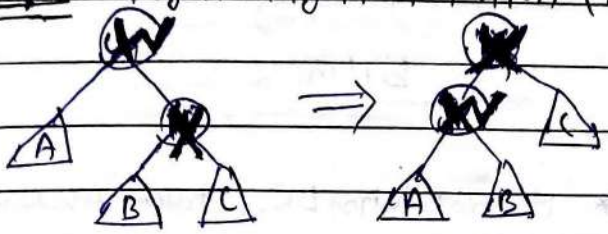
Ex:-



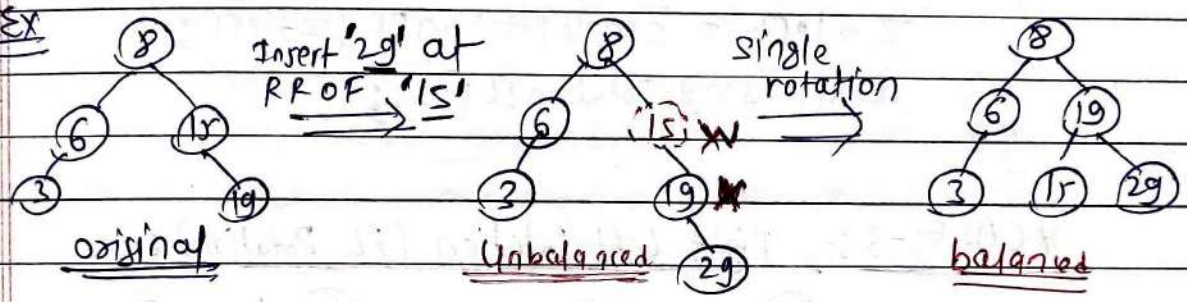
```
Struct AVLNode * singleRotationLL (Struct AVLNode* x){
    Struct AVLNode *w = x->left;
    x->left = w->right;
    w->right = x;
    x->height = max(height(x->left), height(x->right))+1;
    w->height = max(height(w->left), x->height)+1;
    return w;
}
```

Time.C : O(1) / S.C : O(1)

Case 1: Right Right Rotation (RR Rotation) :-



Ex

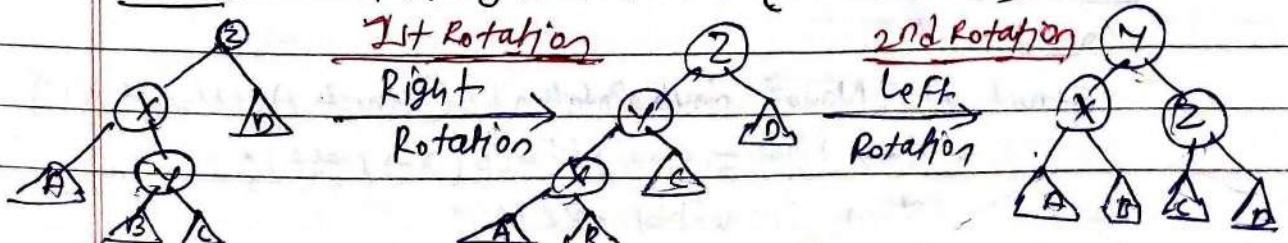



```
Struct AVLNode * singleRotationRR (Struct AVLNode *w){
    Struct AVLNode *x = w->right;
    w->right = x->left;
    x->left = w;
    w->height = max(height(w->right), height(w->left)) + 1
    x->height = max(height(x->right), w->height) + 1;
    return x;
}
```

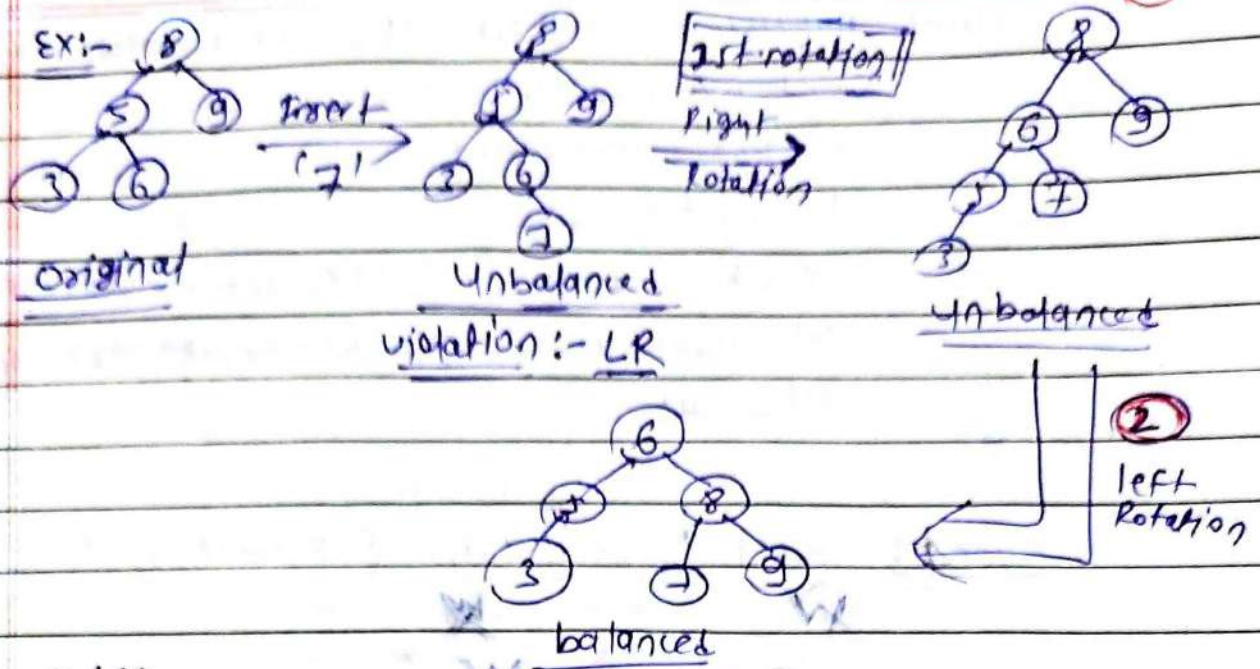
T.C : O(1) / S.C : O(1)

⊛ Double Rotations :-

CASE 2 :- Left-Right Rotation (LR-Rotation)



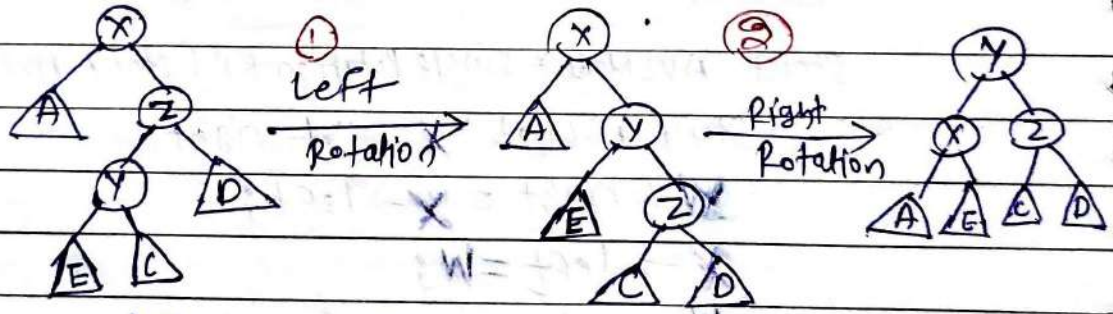
Ex:-



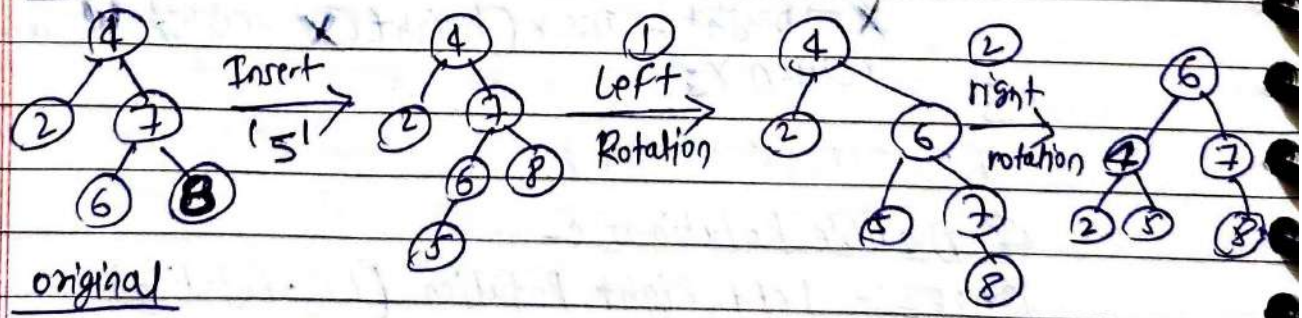
code:-

```
Struct AVLNode * doubleRotationLR (Struct AVLNode* z){
    z->left = singleRotationRR (z->left);
    return singleRotationLL (z);
}
```

*CASE-3:- Right Left Rotation (RL Rotation) :-



Ex:-



original

code:-

```
Struct AVLNode* doubleRotationRL (Struct AVLNode* X){
    X-> right = singleRotationLL (X-> right);
    return singleRotationRR (X);
}
```

(*) Insertion into an AVL Tree.

it is similar to BST insertion. After inserting, we just need to check whether there are any height imbalance, if yes then call appropriate rotation function.

code:-

```
struct AVLNode* insert(struct AVLNode *root, struct AVLNode
                                    *parent, int data){
    if(!root){
        root = (struct AVLNode*)malloc(sizeof(struct
                                        AVLNode));
        root->data = data;
        root->height=0;
        root->left=root->right=NULL;
    }
    else if(data < root->data){
        root->left=insert(root->left, root, data);
        if((height(root->left)-height(root->right))==2){
            if(data < root->left->data)
                root=singleRotateLL(root);
            else root=doubleRotateLR(root);
        }
    }
    else if(data > root->data){
        root->right=insert(root->right, root, data);
        if((height(root->right)-height(root->left))==2){
            if(data < root->right->data)
                root=singleRotateRR(root);
            else root=doubleRotateRL(root);
        }
    }
    //else data is in the tree already, we'll do nothing
    root->height=max(height(root->left), height(root->right))
                                                    +1;
    return root;
}
```

T:C - O(log n) / S:C - O(log n)

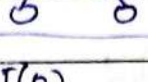
Note:-

- how many different shapes can there be of a minimal AVL tree of height h?

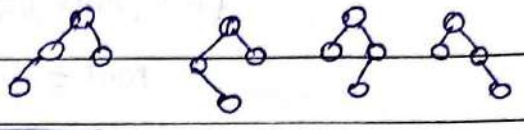
Let NS(h) is minm different shape of a minmal AVL tree 'h' height

then NS(0) = 1

NS(1) = 2



NS(2) = 2 * NS(1) * NS(0)
= 2 * 2 * 1 = 4



$$NS(h) = 2 * NS(h-1) * NS(h-2)$$

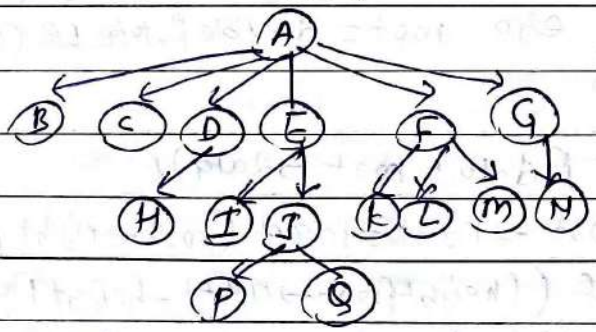
(10) Generic Trees (N-ary Trees)

- In generic trees there can be many children at every node and we don't know how many children a node can have.

(*) representation:-

How can we represent them

ex:-



- In the above tree, there are nodes with 6, 3, 2, 2, 0 children. To represent this tree we have to consider (6 children) and allocate that many child, for each node. i.e

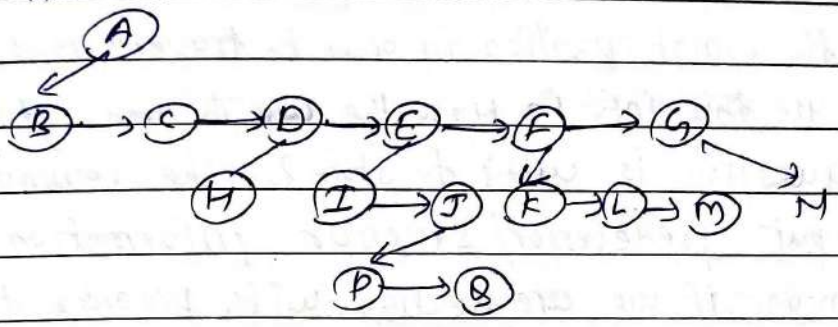
```
Struct TreeNode {        Pointer
    int data;
    Struct TreeNode *FirstChild;
    Struct TreeNode * secondChild;
    Struct TreeNode * thirdChild;
    Struct TreeNode * FourthChild;
    Struct TreeNode * FifthChild;
    Struct TreeNode * SixthChild;
};
```

- since we are not using all the pointers in all the cases, there is a lot of memory wastage. another problem we don't know no. of children for each in advance
- To solve this problem & minimize wastage we use generic tree.

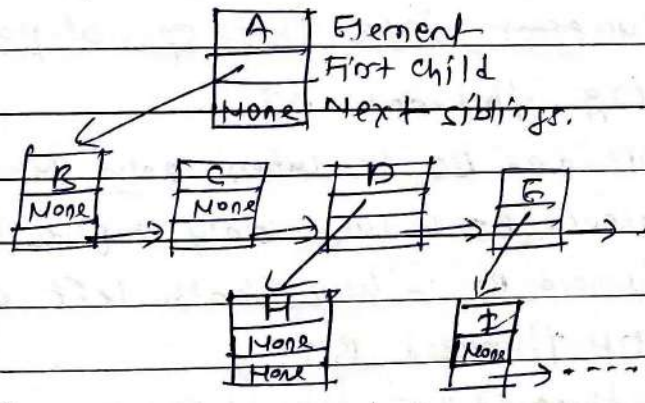
Representation of Generic Trees:-

representations:-

- At each node link children of same parent (siblings) from left to right.
- remove the links from parent to all children except the first child.



- This representation is also known as First child/next sibling representation.



Code :-

```
struct TreeNode{
    int data;
    struct TreeNode *FirstChild;
    struct TreeNode *nextsiblings;
};
```

(11.) Threaded Binary Tree Traversals (stack or queue less Traversals)

- in Threaded Binary Tree Traversals we do not need both stacks and queues.

Issues with Regular Binary Tree Traversals

- The storage space required for the stack & queue is large.
- The majority of pointer in BT are NULL. a BT can have (n+1) NULL with 'n' node. and these are wasted.
- It is difficult to find Successor Node (preorder, inorder and post order successor) for a given node.

Motivation for Threaded Binary Trees.

- The idea is to store some useful info. in NULL pointers.
- since in BT, we use stack/queue bcz we have to record the current position in order to traverse right after left. if we use this info in NULL the we do need stack/queue.
- The question is what to store?, The common convention is to put predecessor/successor information.
- That means, if we are dealing with preorder traversals, then for a given node, NULL left pointer will contain preorder predecessor info and NULL right pointer will contain preorder successor info. These special pointer are thread.

Classifying threaded BT.

- left threaded BT :- when only NULL left pointer used (Predecessor info)
- right threaded BT :- when only right left pointer used (Successor info)
- Fully threaded BT :- when both left & right pointer used (Predecessor & successor)

Types of Threaded B.T.

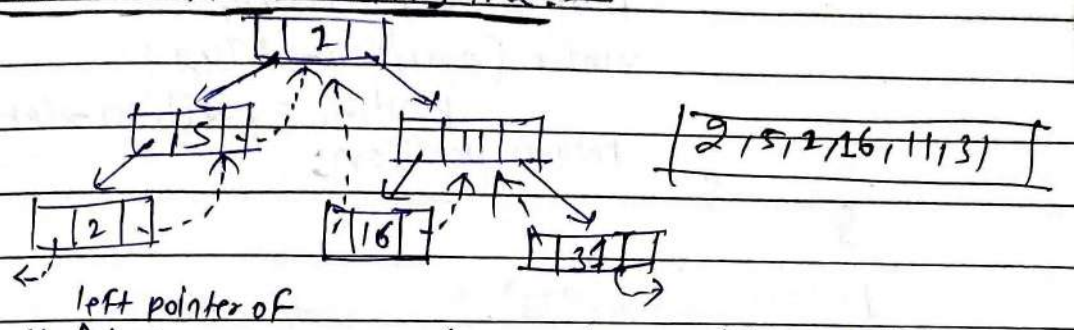
- Preorder threaded BT :- NULL left = preorder predecessor, NULL Right = preorder successor
- Inorder threaded BT :- Inorder predecessor & successor used
- Postorder threaded BT :- Postorder predecessor & successor used.

Threaded Binary Tree structure.

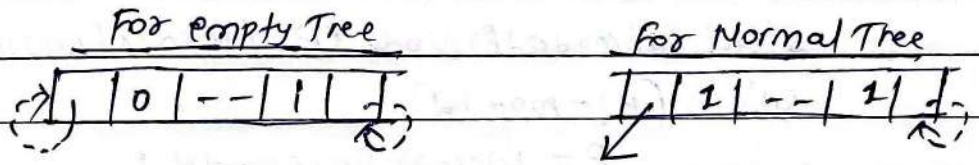
```
struct ThreadedBinaryTreeNode{
    struct ThreadedBinaryTreeNode *left;
    int LTag;
    int data;
    int RTag;
    struct ThreadedBinaryTreeNode* right;
};
```

	Regular BT	Threaded BT
if LTag==0	NULL	left points to the predecessor
if LTag==1	Left point to the left child	Left points to left child
if RTag==0	NULL	right points to the successor
if RTag==1	Right point to Right child	right points to the right child

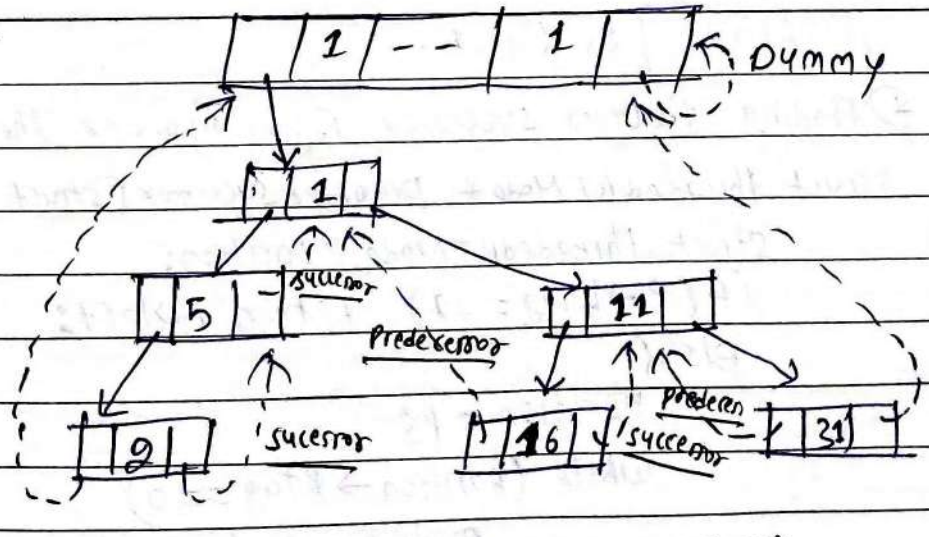
Ex:- Inorder threaded Binary Tree:—

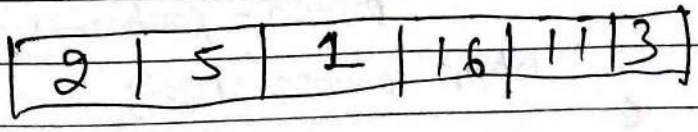


- here the left most node (2) and right pointer of right most node (3)) will point to a dummy node which is always present even for an empty tree. Note that RTag of Dummy node is 1 and its right child points to itself.



- with this convention the above tree can be represented as:





④ Finding Inorder Successor in Inorder Threaded BT.

```
struct ThreadedBTNode * inorderSuccessor(struct ThreadedBTNode *P){
    struct ThreadedBTNode *position;
    if (P->Rtag == 0) return P->right;
    else{
        position = P->right;
        while (position->Ltag)
            position = position->left;
        return position;
    }
}
```

TC: O(n) | SC: O(1)

⑤ Inorder Traversal in Inorder threaded Binary Tree

- We can start with dummy node and call inorderSuccessor() to visit each node until we reach dummy node.

```
→ void InorderTraversal(struct ThreadedBTNode* root){
    struct ThreadedBTNode *P = inorderSuccessor(root);
    while (P != root){
        P = inorderSuccessor(P);
        printf("%d", P->data);
    }
}
```

TC: O(n) | SC: O(1).

⑥ Finding PreOrder successor from inorder Threaded BT.

```
struct ThreadedBTNode * preorderSuccessor(struct ThreadedBTNode *P){
    struct ThreadedBTNode *position;
    if (P->Ltag == 1) return P->left;
    else{
        position = P;
        while (position->Rtag == 0)
            position = position->right;
        return position->right;
    }
}
```

TC: O(n) | SC: O(1)

⊛ preorder traversal of inorder threaded B.T.

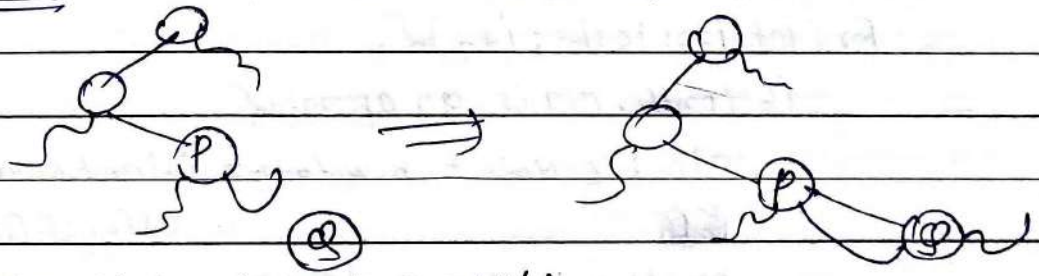
```
void preorderTraversal(struct threadedBTNode *root){
    struct threadedBTNode* p;
    p = preorderSuccessor(root);
    while(p != root){
        p = preorderTraversal(p);
        printf("%d", p->data);
    }
}
```

Note :- Finding preorder & inorder successor from threaded BT is very easy. But difficult to find postorder successor.

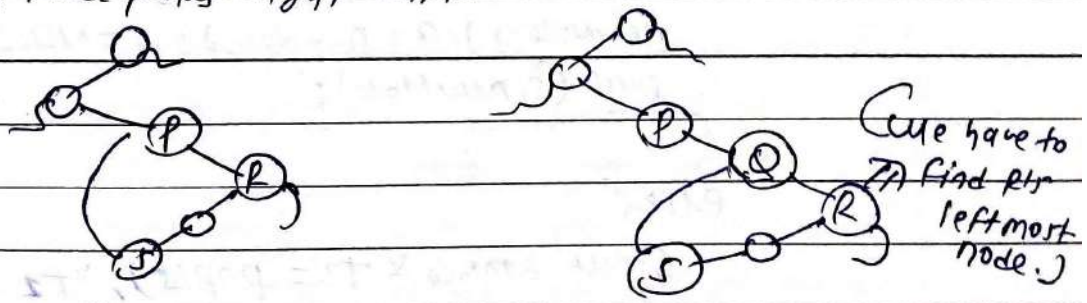
⊛ Insertion of Nodes in Inorder Threaded BT

For simplicity, let us assume that we have two nodes P & Q and we want to attach Q to right of P.

Case 1: Node P does not have right child.



Case 2: Node p has right child.




```
void insert(struct threadedBTNode*P, struct threadedBTNode*Q){
    struct threadedBTNode *temp;
    Q->right = P->right; Q->RTag = P->RTag;
    Q->left = P; Q->LTag = 0;
    P->right = Q; P->RTag = 1;
    if(Q->RTag == 1){
        temp = Q->right;
        while(temp->LTag) temp = temp->left;
        temp->left = Q;
    }
}
```

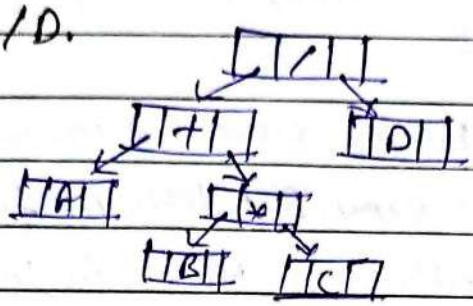
TC: O(n)
SC: O(1)

Date
Page No. 290.

(12) Expression Trees:

- a tree representing an expression is called an expression tree. In this tree leaf node are operand & non-leaf is operators i.e it is a binary tree type.
- an expression tree consists of binary expression. But for a unary operator, one subtree will be empty.

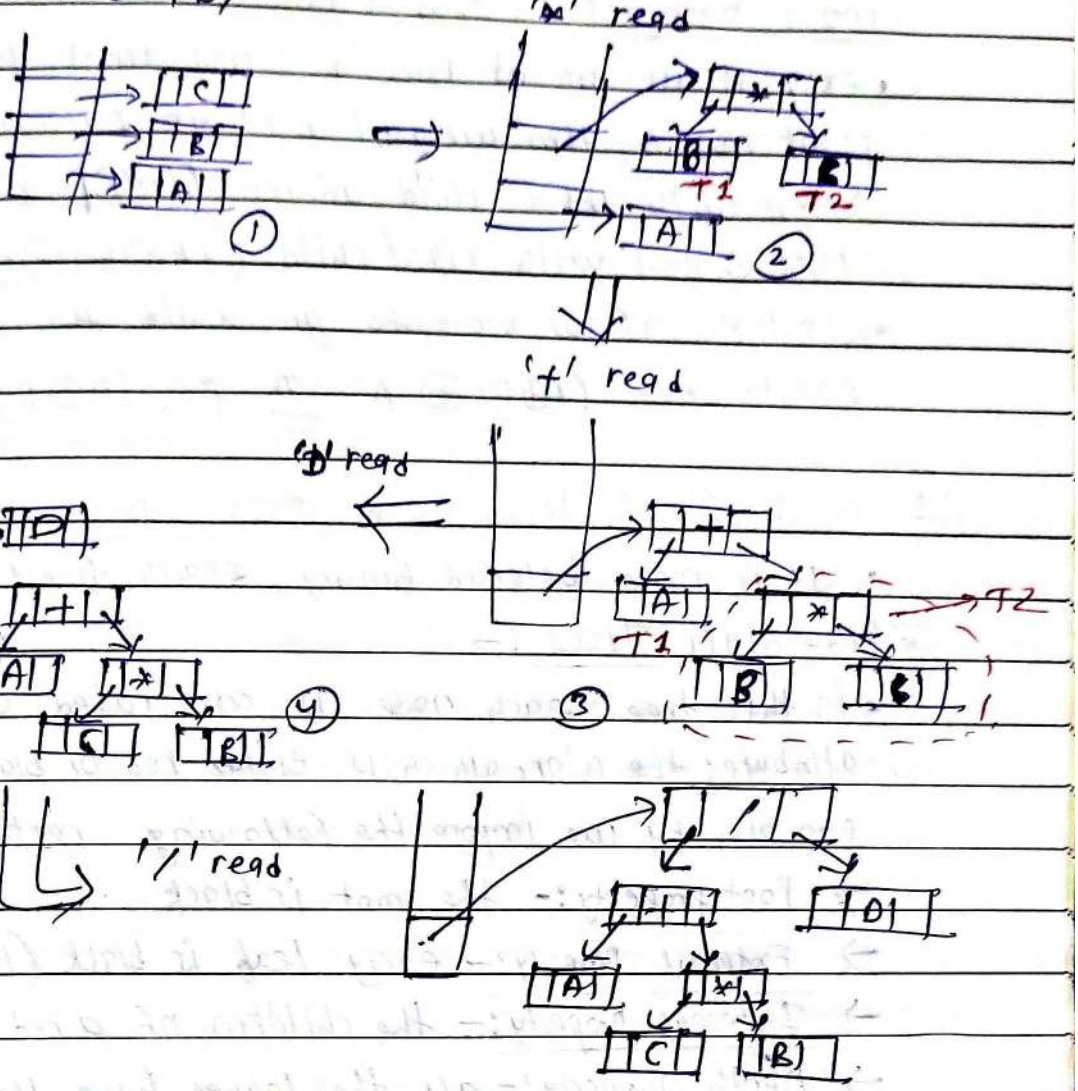
Ex:- (A+B*C)/D.



* Build Expression tree from Postfix Expression :-

```
struct BTNode* buildET (char postfix[], int size){
    struct Stack *S = Stack(size);
    For(int i=0; i<size; i++){
        if (postfix[i] is an operand){
            struct BTNode * newNode = (struct BTNode*) malloc
                                      (sizeof (struct BTNode));
            newNode -> data = postfix[i];
            newNode -> left = newNode -> right = NULL;
            push (S, newNode);
        }
        else{
            struct BTNode *T2 = pop(S), *T1 = pop(S)
            struct BTNode* newNode = (struct BTNode *) malloc (sizeof
                                      (struct BTNode));
            newNode -> data = postfix[i];
            newNode -> left = T1;
            newNode -> right = T2;
            push (S, newNode);
        }
    }
    return S;
}
```

Ex:- ABC*+D/

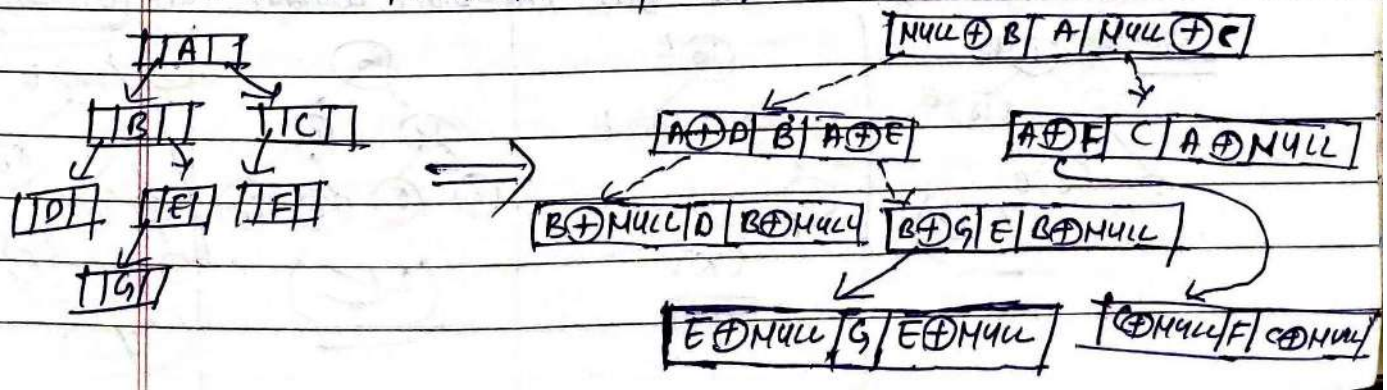


(13) XOR Trees:

- this concept is memory effiient, this representation does not need stack or queues for traversing the trees.
- we can go back and forth from a node by using ⊕ operation.

Representation of XOR trees:-

- Each node left will have the ⊕ of its parent & left children
- Each node right will have the ⊕ of its parent & right children
- Root node parent is Null & leaf node children are Null nodes.



move back & forth from a node :-

- ex:, if we are at node 'B' and want to move to its parent node A, then we just need to perform ⊕ on its left content with left child address $(A \oplus D) \oplus D = A$, or right content with right child $(A \oplus E) \oplus E = A$.
- similarly if we want to go child the xor content with parent. ex:- $(A \oplus D) \oplus A = D$ & $(A \oplus E) \oplus A = E$

(14) Red-Black Trees and Splay Tree

- It is more balanced binary search trees like AVL tree.

(*) Red-Black Trees :-

- in this tree each node is associated with an extra attribute: the color, which is either red or black. To get logarithmic complexity we impose the following restrictions.

→ Root property :- the root is black

→ External property :- every leaf is black (leaf can be NULL)

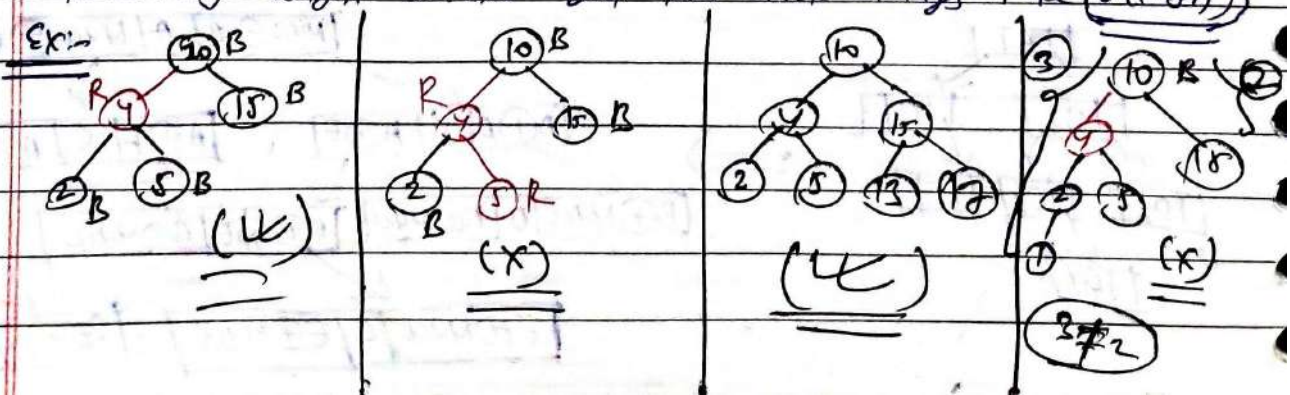
→ Internal property :- the children of a red node are black

→ Depth property :- all the leaves have the same black (Path from root to leave)

- Similar to AVL trees, if the Red-black trees become imbalanced, then we perform rotations to reinfore the balancing property. with Red-black trees, we can perform the following operation in O(logn) in worst case, n = no. of nodes.
  - Insertion / deletion / Finding predecessor, successor / Finding minm, maxm

Note :- Red-black tree is roughly height balance. and AVL trees is strictly height balance but Red-black always take O(logn) (guarantee)

Ex :-



Splay Tree :-

- Splay-Tree are BSTs with a self-adjusting property.
- it is starting with an empty tree, any sequence of k operation with max^m of n nodes takes $O(k \log n)$ time complexity in worst case.
- splay trees are easier to program and also ensure faster access to recently accessed items.
- similar to AVL & Red-Black tree at any point if splay tree becomes imbalanced, we can perform rotations.
- It can not guarantee the $O(\log n)$ complexity in worst case. But it gives amortized $O(\log n)$ complexity. i.e individual operations can be expensive, any sequence of operations gets the complexity of logarithmic behaviour. (average operation)

(15) B-Trees

- B-Tree is like other self-balancing trees such as AVL and Red-black tree such that it maintain its balance of node while operations are performed against it. B-Tree has the following properties:
  - min^m degree 't' where, except root node, all other nodes must have no less than 't-1' keys.
  - Each node with n keys has n+1 children
  - keys in each node are lined up where $k_1 < k_2 < k_3 \cdots < k_n$
  - Each node cannot have more than 2t-1 key, thus 2t children.
  - Root node at least must contain one key. there is no root node if the tree is empty.
  - Tree grows in depth only when root node is split.
- b-tree may have a variable no. of keys and children
- keys are stored in non-decreasing order