```
        else {  currentSum    += temp→data;
                if(temp→left)
                        EnQueue(temp, temp→left);
                if(root→right)
                        EnQueue(temp, temp→right);
        }
    }
    return maxLevel;
}
```

Time Complexity: O(n). Space Complexity: O(n).

Problem-20 Given a binary tree, print out all of its root-to-leaf paths.

Solution: Refer comments in functions.

```
void PrintPathsRecur(struct BinaryTreeNode *root, int path[], int pathLen) {
    if(root ==NULL)
        return;
    // append this node to the path array
    path[pathLen] = root→data;
    pathLen++;
    // it's a leaf, so print the path that led to here
    if(root→left==NULL && root→right==NULL)
        PrintArray(path, pathLen);
    else {  // otherwise try both subtrees
        PrintPathsRecur(root→left, path, pathLen);
        PrintPathsRecur(root→right, path, pathLen);
    }
}
// Function that prints out an array on a line.
void PrintArray(int ints[], int len) {
    for (int i=0; i<len; i++)
        printf("%d",ints[i]);
}
```

Time Complexity: O(n). Space Complexity: O(n), for recursive stack.

Problem-21 Give an algorithm for checking the existence of path with given sum. That means, given a sum check whether there exists a path from root to any of the nodes.

Solution: For this problem, the strategy is: subtract the node value from the sum before calling its children recursively, and check to see if the sum is 0 when we run out of tree.

```
int HasPathSum(struct BinaryTreeNode * root, int sum) {
    // return true if we run out of tree and sum==0
    if(root == NULL) return(sum == 0);
    else {  // otherwise check both subtrees
        int remainingSum = sum - root→data;
        if((root→left && root→right)||(!root→left && !root→right))
            return(HasPathSum(root→left, remainingSum) || HasPathSum(root→right, remainingSum));
        else if(root→left)
            return HasPathSum(root→left, remainingSum);
        else    return HasPathSum(root→right, remainingSum);
    }
```



```
}
```

Time Complexity: O(n). Space Complexity: O(n).

Problem-22 Give an algorithm for finding the sum of all elements in binary tree.

Solution: Recursively, call left subtree sum, right subtree sum and add their values to current nodes data.

```
int Add(struct BinaryTreeNode *root) {
    if(root == NULL) return 0;
    else return (root→data + Add(root→left) + Add(root→right));
}
```

Time Complexity: O(n). Space Complexity: O(n).

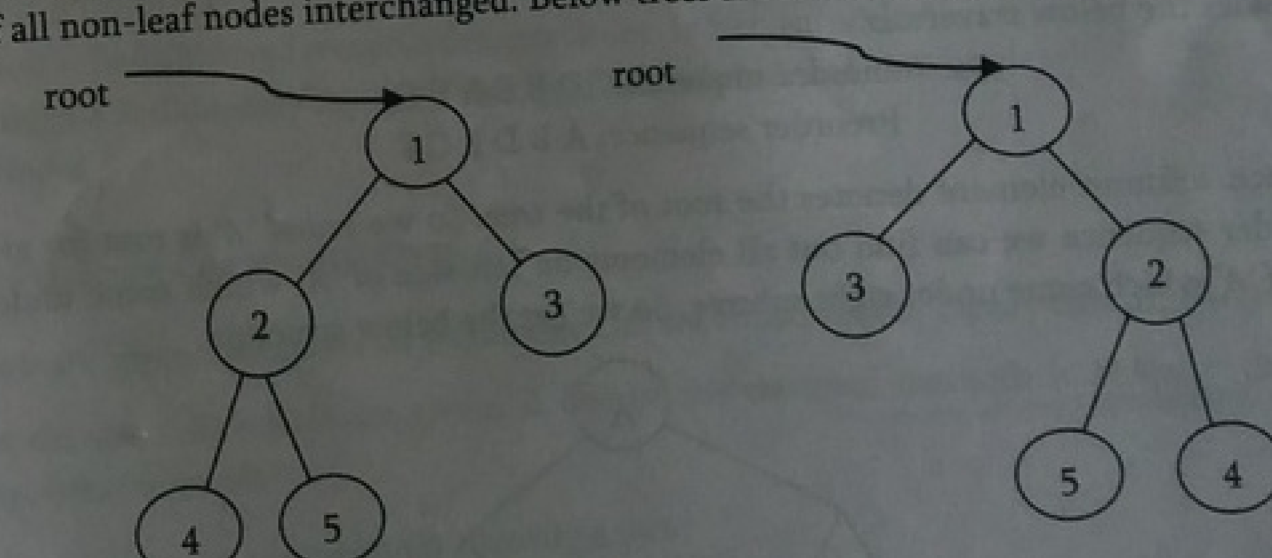
Problem-23 Can we solve Problem-22 without recursion?

Solution: We can use level order traversal with simple change. Every time after deleting an element from queue, add the nodes data value to *sum* variable.

```
int SumofBTusingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    int sum = 0;
    if(!root) return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        sum += temp→data;
        if(temp→left)
            EnQueue (Q, temp→left);
        if(temp→right)
            EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return sum;
}
```

Time Complexity: O(n). Space Complexity: O(n).

Problem-24 Give an algorithm for converting a tree to its mirror. Mirror of a tree is another tree with left and right children of all non-leaf nodes interchanged. Below trees are mirrors to each other.



Solution:

```
struct BinaryTreeNode *MirrorOfBinaryTree(struct BinaryTreeNode *root){
    struct BinaryTreeNode * temp;
    if(root) {
```

```
        MirrorOfBinaryTree(root→left);
        MirrorOfBinaryTree(root→right);
        /* swap the pointers in this node */
        temp = root→left;
        root→left = root→right;
        root→right = temp;
    }
    return root;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-25 Given two trees, give an algorithm for checking whether they are mirrors of each other.

Solution:

```
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 Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-26 Give an algorithm for finding LCA (Least Common Ancestor) of two nodes in a Binary Tree.

Solution:

```
struct BinaryTreeNode *LCA(struct BinaryTreeNode *root, struct BinaryTreeNode *α, struct BinaryTreeNode *β){
    struct BinaryTreeNode *left, *right;
    if(root == NULL) return root;
    if(root == α || root == β) return root;
    left = LCA (root→left, α, β );
    right = LCA (root→right, α, β );
    if(left && right)
        return root;
    else
        return (left? left: right)
}
```

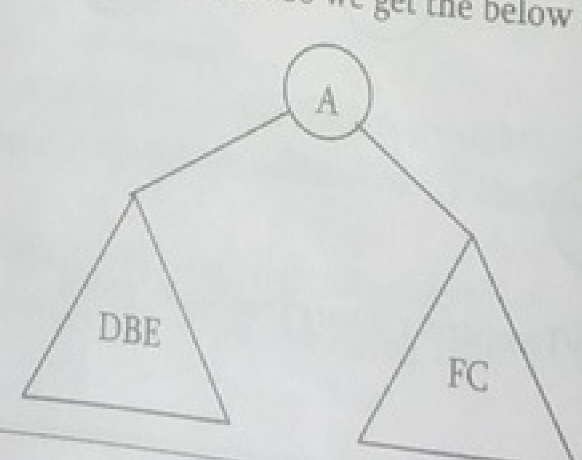
Time Complexity: $O(n)$. Space Complexity: $O(n)$ for recursion.

Problem-27 Give an algorithm for constructing binary tree from given Inorder and Preorder traversals.

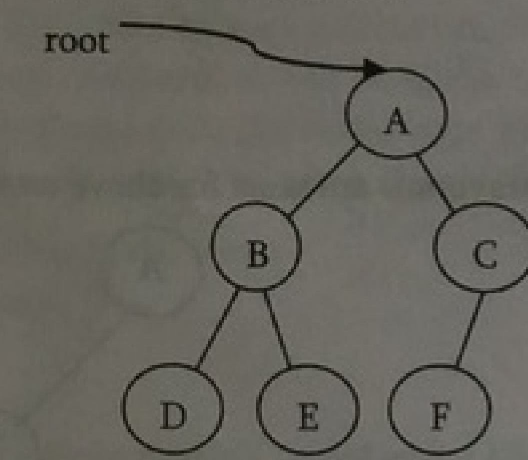
Solution: Let us consider the below traversals:

Inorder sequence: D B E A F C
Preorder sequence: A B D E C F

In a Preorder sequence, leftmost element denotes the root of the tree. So we know '*A*' is root for given sequences. By searching '*A*' in Inorder sequence we can find out all elements on left side of '*A*' which come under left subtree and elements right side of '*A*' which come under right subtree. So we get the below structure.



We recursively follow above steps and get the following tree.



Algorithm: BuildTree()

1 Select an element from Preorder. Increment a Preorder index variable (preIndex in below code) to pick next element in next recursive call.
2 Create a new tree node (newNode) with the data as selected element.
3 Find the selected elements index in Inorder. Let the index be inIndex.
4 Call BuildBinaryTree for elements before inIndex and make the built tree as left subtree of newNode.
5 Call BuildBinaryTree for elements after inIndex and make the built tree as right subtree of newNode.
6 return newNode.

```
struct BinaryTreeNode *BuildBinaryTree(int inOrder[], int preOrder[], int inStrt, int inEnd){
    static int preIndex = 0;
    struct BinaryTreeNode *newNode
    if(inStrt > inEnd) return NULL;
    newNode = (struct BinaryTreeNode *) malloc  (sizeof(struct BinaryTreeNode));
    if(!newNode) {
        printf("Memory Error");
        return NULL;
    }
    // Select current node from Preorder traversal using preIndex
    newNode→data = preOrder[preIndex];
    preIndex++;
    if(inStrt == inEnd)                 /* if this node has no children then return */
        return newNode;
    /* else find the index of this node in Inorder traversal */
    int inIndex = Search(inOrder, inStrt, inEnd, newNode→data);
    /* Using index in Inorder traversal, construct left and right subtress */
    newNode→left = BuildBinaryTree(inOrder, preOrder, inStrt, inIndex-1);
    newNode→right = BuildBinaryTree(inOrder, preOrder, inIndex+1, inEnd);
    return newNode;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

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

Solution: It depends on what traversals are given. If one of the traversal methods is *Inorder* then the tree can be constructed uniquely, otherwise not.

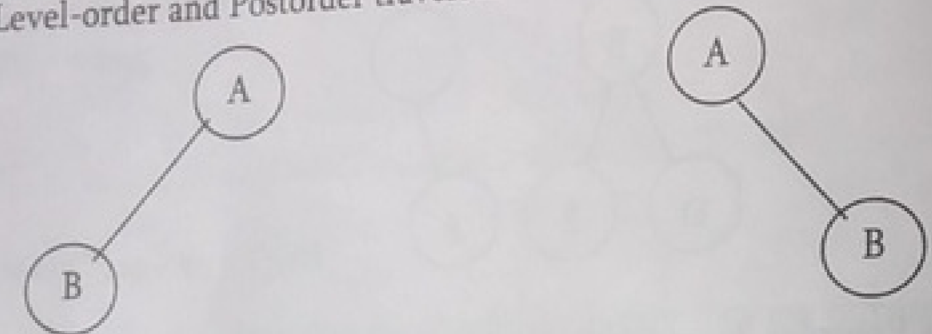
Therefore, following combination can uniquely identify a tree:

- Inorder and Preorder
- Inorder and Postorder
- Inorder and Level-order

The following combinations do not uniquely identify a tree.

- Postorder and Preorder
- Preorder and Level-order
- Postorder and Level-order

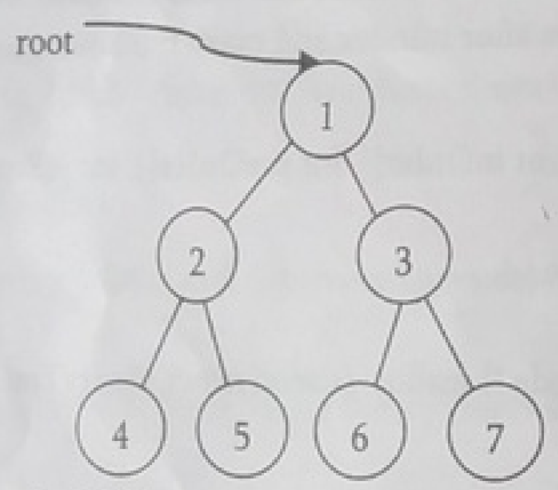
For example, Preorder, Level-order and Postorder traversals are same for above trees:



Preorder Traversal = AB Postorder Traversal = BA Level-order Traversal = AB

So, even if three of them (PreOrder, Level-Order and PostOrder) are given, tree cannot be constructed uniquely.

Problem-29 Give an algorithm for printing all the ancestors of a node in a Binary tree. For the below tree, for 7 the ancestors are 1 3 7.

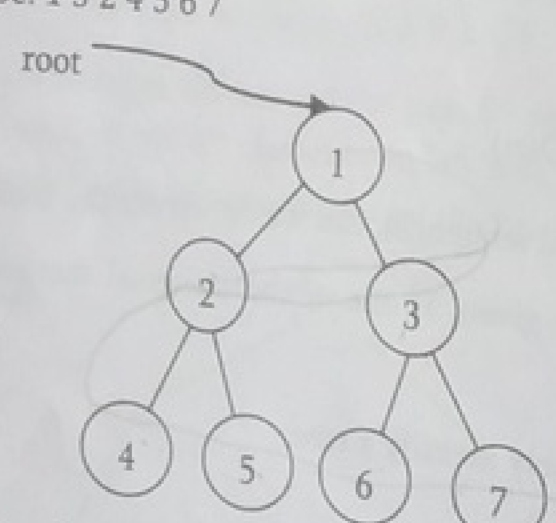


Solution: Apart from the Depth First Search of this tree, we can use the following recursive way to print the ancestors.

```
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: $O(n)$. Space Complexity: $O(n)$ for recursion.

Problem-30 **Zigzag Tree Traversal:** Give an algorithm to traverse a binary tree in Zigzag order. For example, the output for the below tree should be: 1 3 2 4 5 6 7



Solution: This problem can be solved easily using two stacks. Assume the two stacks are: *currentLevel* and *nextLevel*. We would also need a variable to keep track of the current level order (whether it is left to right or right or left).

We pop from *currentLevel* stack and print the nodes value. Whenever the current level order is from left to right, push the nodes left child, then its right child to stack *nextLevel*. Since a stack is a Last In First OUT (*LIFO*) structure, next time when nodes are popped off nextLevel, it will be in the reverse order. On the other hand, when the current level order is from right to left, we would push the nodes right child first, then its left child. Finally, don't forget to swap those two stacks at the end of each level (*i.e.*, when *currentLevel* is empty).

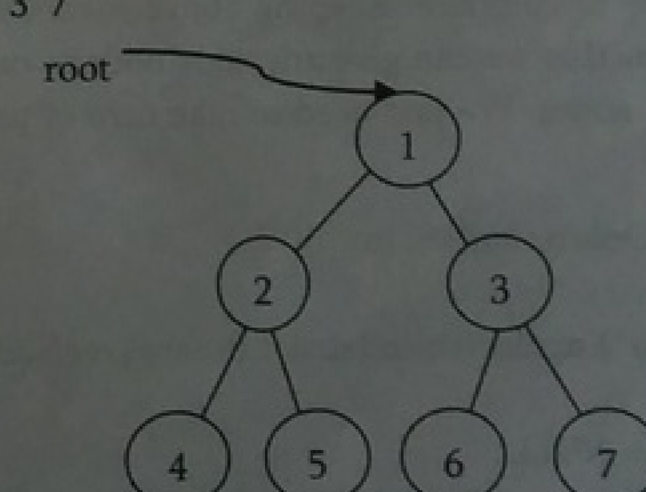
```
void ZigZagTraversal(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    int leftToRight = 1;
    if(!root) return;
    struct Stack *currentLevel = CreateStack(), *nextLevel = CreateStack();
    Push(currentLevel, root);
    while(!IsEmptyStack(currentLevel)) {
        temp = Pop(currentLevel);
        if(temp) {
            printf("%d",temp→data);
            if(leftToRight) {
                if(temp→left)   Push(nextLevel, temp→left);
                if(temp→right) Push(nextLevel, temp→right);
            }
            else {  if(temp→right) Push(nextLevel, temp→right);
                    if(temp→left)   Push(nextLevel, temp→left);
            }
        }
        if(IsEmptyStack(currentLevel)) {
            leftToRight = 1-leftToRight;
            swap(currentLevel, nextLevel);
        }
    }
}
```

Time Complexity: $O(n)$. Space Complexity: Space for two stacks = $O(n) + O(n) = O(n)$.

Problem-31 Give an algorithm for finding the vertical sum of a binary tree. For example,

The tree has 5 vertical lines

Vertical-1: nodes-4 => vertical sum is 4
Vertical-2: nodes-2 => vertical sum is 2
Vertical-3: nodes-1,5,6 => vertical sum is 1 + 5 + 6 = 12
Vertical-4: nodes-3 => vertical sum is 3
Vertical-5: nodes-7 => vertical sum is 7
We need to output: 4 2 12 3 7

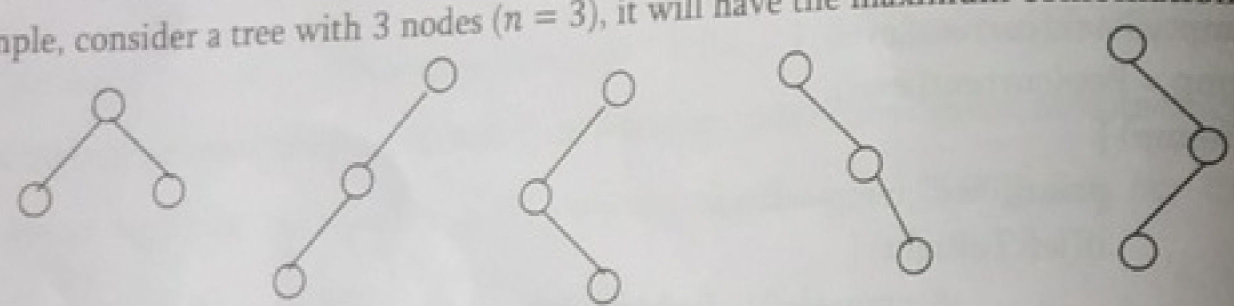


Solution: We can do an inorder traversal and hash the column. We call VerticalSumInBinaryTree(root, 0) which means the root is at column 0. While doing the traversal, hash the column and increase its value by *root → data*.

```
void VerticalSumInBinaryTree (struct BinaryTreeNode *root, int column){
        if(root==NULL) return;
        VerticalSumInBinaryTree(root→left, column-1);
        //Refer Hashing chapter for implementation of hash table
        Hash[column] += root→data;
        VerticalSumInBinaryTree(root→right, column+1);
}
VerticalSumInBinaryTree(root, 0);
Print Hash;
```

Problem-32 How many different binary trees are possible with n nodes?

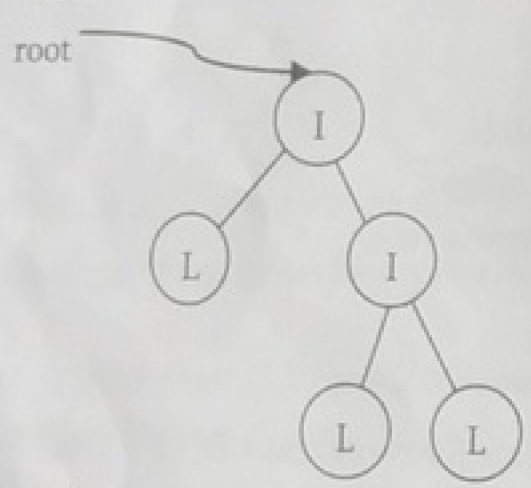
Solution: For example, consider a tree with 3 nodes ($n = 3$), it will have the maximum combination of 5 different (i.e., $2^3 - 3 = 5$) trees.



In general, if there are n nodes, there exist $2^n - n$ different trees.

Problem-33 Given a tree with a special property where leaves are represented with 'L' and internal node with 'I'. Also, assume that each node has either 0 or 2 children. Given preorder traversal of this tree, construct the tree.

Example: Given preorder string => ILILL



Solution: First, we should see how preorder traversal is arranged. Pre-order traversal means first put root node, then pre-order traversal of left subtree and then pre-order traversal of right subtree. In normal scenario, it's not possible to detect where left subtree ends and right subtree starts using only pre-order traversal. Since every node has either 2 children or no child, we can surely say that if a node exists then its sibling also exists. So every time when we are computing a subtree, we need to compute its sibling subtree as well.

Secondly, whenever we get 'L' in the input string, that is a leaf and we can stop for a particular subtree at that point. After this 'L' node (left child of its parent 'L'), its sibling starts. If 'L' node is right child of its parent, then we need to go up in the hierarchy to find next subtree to compute. Keeping above invariant in mind, we can easily determine when a subtree ends and next start. It means that we can give any start node to our method and it can easily complete the subtree it generates going outside of its nodes. We just need to take care of passing correct start nodes to different sub-trees.

```
struct BinaryTreeNode *BuildTreeFromPreOrder(char* A, int *i){
        struct BinaryTreeNode *newNode;
        newNode = (struct BinaryTreeNode *) malloc(sizeof(struct BinaryTreeNode));
        newNode→data = A[*i];
        newNode→left = newNode→right = NULL;
        if(A == NULL){                               //Boundary Condition
                free(newNode);
```



```
                return NULL;
        }
        if(A[*i] == 'L')                             //On reaching leaf node, return
                return newNode;
        *i = *i + 1;                                 //Populate left sub tree
        newNode→left = BuildTreeFromPreOrder(A, i);
        *i = *i + 1;                                 //Populate right sub tree
        newNode→right = BuildTreeFromPreOrder(A, i);
        return newNode;
}
```

Time Complexity: $O(n)$.

Problem-34 Given a binary tree with three pointers (left, right and nextSibling), give an algorithm for filling the *nextSibling* pointers assuming they are NULL initially.

Solution: We can use simple queue (similar to the solution of Problem-11). Let us assume that the structure of binary tree is:

```
struct BinaryTreeNode {
        struct BinaryTreeNode* left;
        struct BinaryTreeNode* right;
        struct BinaryTreeNode* nextSibling;
};
int FillNextSiblings(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        struct Queue *Q;
        if(!root) return 0;
        Q = CreateQueue();
        EnQueue(Q,root);
        EnQueue(Q,NULL);
        while(!IsEmptyQueue(Q)) {
                temp =DeQueue(Q);
                // Completion of current level.
                if(temp ==NULL) { //Put another marker for next level.
                        if(!IsEmptyQueue(Q))
                                EnQueue(Q,NULL);
                }
                else {  temp→nextSibling = QueueFront(Q);
                        if(root→left) EnQueue(Q, temp→left);
                        if(root→right) EnQueue(Q, temp→right);
                }
        }
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-35 For Problem-34, is there any otherway of solving?

Solution: The trick is to re-use the populated *nextSibling* pointers. As mentioned earlier, we just need one more step for it to work. Before we passed the *left* and *right* to the recursion function itself, we connect the right childs *nextSibling* to the current nodes nextSibling left child. In order for this to work, the current node *nextSibling* pointer must be populated, which is true in this case.

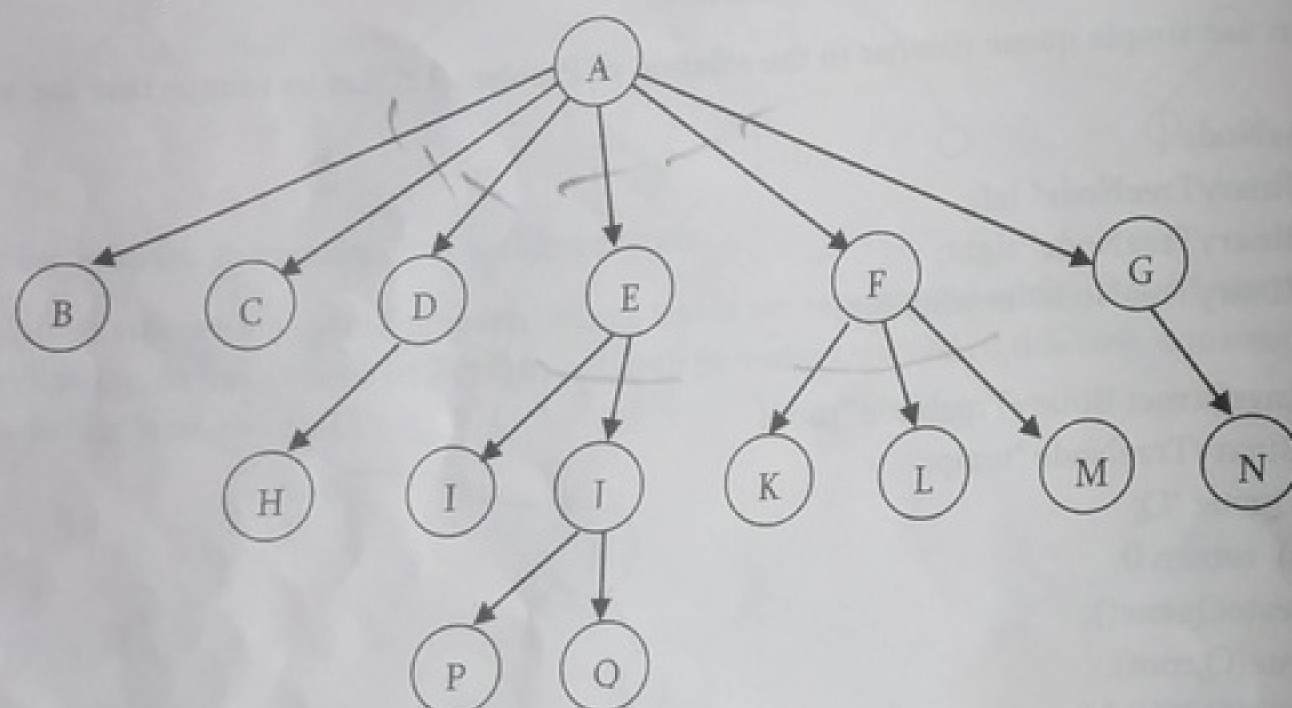
```
void FillNextSiblings(struct BinaryTreeNode* root) {
        if (!root) return;
```

```
        if (root→left) root→left→nextSibling = root→right;
        if (root→right)
                root→right→nextSibling = (root→nextSibling) ? root→nextSibling→left : NULL;
        FillNextSiblings(root→left);
        FillNextSiblings(root→right);
}
```

Time Complexity: $O(n)$.

6.7 Generic Trees (N-ary Trees)

In the previous section we have discussed binary trees where each node can have maximum of two children only and represented them easily with two pointers. But suppose if we have a tree with many children at every node and also if we do not know how many children a node can have, how do we represent them? For example, consider the tree shown above.



For a tree like this, how do we represent the tree?

In the above tree, there are nodes with 6 children, with 3 children, 2 children, with 1 child, and with zero children (leaves). To present this tree we have to consider the worst case (6 children) and allocate those many child pointers for each node. Based on this, the node representation can be given as:

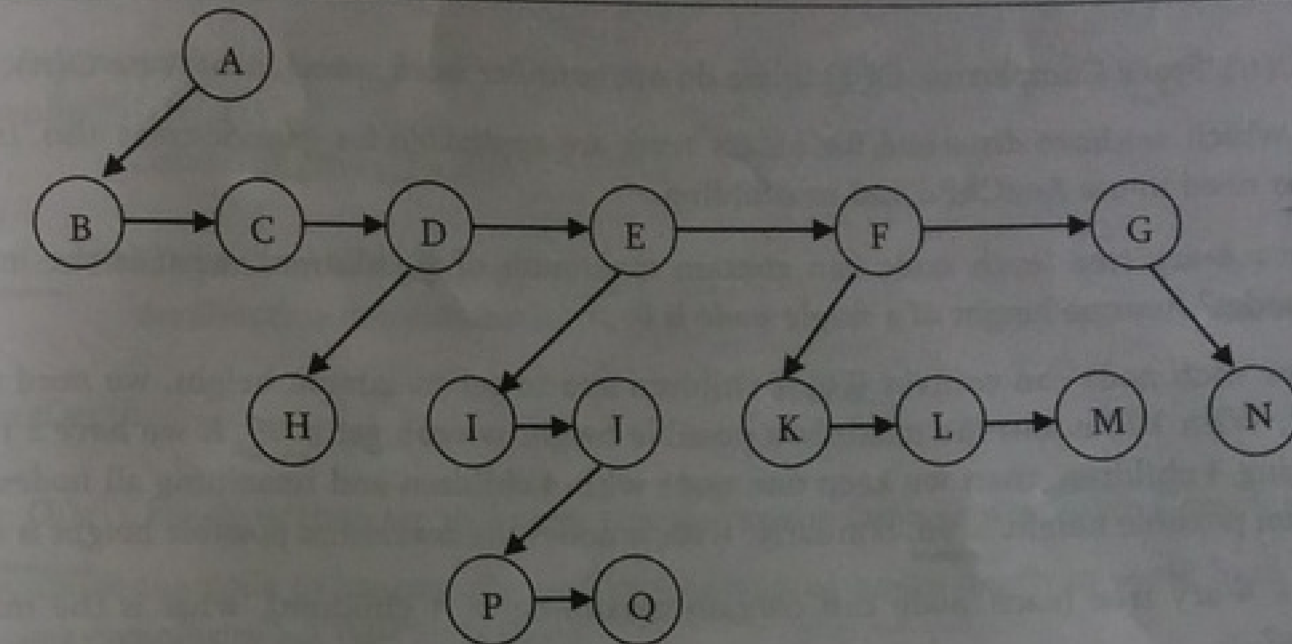
```
struct TreeNode{
        int data;
        struct TreeNode *firstChild;
        struct TreeNode *secondChild;
        struct TreeNode *thirdChild;
        struct TreeNode *fourthChild;
        struct TreeNode *fifthChild;
        struct TreeNode *sixthChild;
};
```

Since we are not using all the pointers in all the cases there is a lot of memory wastage. Also, another problem is that, n advance we do not know the number of children for each node. In order to solve this problem we need a epresentation that minimizes the wastage and also accept nodes with any number of children.

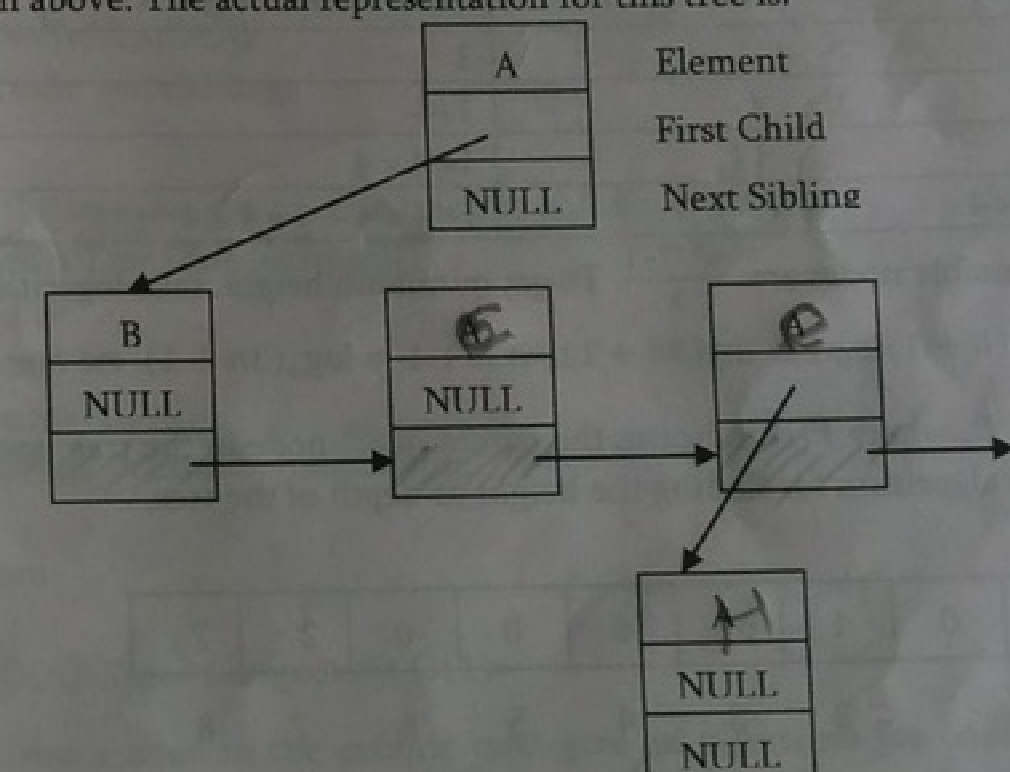
epresentation of Generic Trees

nce our objective is to reach all nodes of the tree, a possible solution to this is as follows:

- 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.



What these above statements say is if we have a link between childrens then we do not need extra links from parent to all children. This is because we can traverse all the elements by starting at the first child of the parent. So if we have link between parent and first child and also links between all children of same parent then it solves our problem. This representation is sometimes called first child/next sibling representation. First child/next sibling representation of the generic tree is shown above. The actual representation for this tree is:



Based on this discussion, the tree node declaration for general tree can be given as:

```
struct TreeNode {
        int data;
        struct TreeNode *firstChild;
        struct TreeNode *nextSibling;
};
```

Note: Since we are able to represent any generic tree with binary representation, in practice we use only binary tree.

Problems on Generic Trees

Problem-36 Given a tree, give an algorithm for finding the sum of all the elements of the tree.

Solution: The solution is similar to what we have done for simple binary trees. That means, traverse the complete list and keep on adding the values. We can either use level order traversal or simple recursion.

```
int FindSum(struct TreeNode *root){
        if(!root) return 0;
        return root→data + FindSum(root→firstChild) + FindSum(root→nextSibling);
}
```

Time Complexity: O(n). Space Complexity: O(1) (if we do not consider stack space), otherwise O(n).

Note: All problems which we have discussed for binary trees are applicable for generic trees also. Instead of left and right pointers we just need to use firstChild and nextSibling.

Problem-37 For a 4-ary tree (each node can contain maximum of 4 children), what is the maximum possible height with 100 nodes? Assume height of a single node is 0.

Solution: In 4-ary tree each node can contain 0 to 4 children and to get maximum height, we need to keep only one child for each parent. With 100 nodes the maximum possible height we can get is 99. If we have a restriction that at least one node is having 4 children, then we keep one node with 4 children and remaining all nodes with 1 child. In this case, the maximum possible height is 96. Similarly, with n nodes the maximum possible height is $n - 4$.

Problem-38 For a 4-ary tree (each node can contain maximum of 4 children), what is the minimum possible height with n nodes?

Solution: Similar to above discussion, if we want to get minimum height, then we need to fill all nodes with maximum children (in this case 4). Now let's see the following table, which indicates the maximum number of nodes for a given height.

Height, h	Maximum Nodes at height, $h = 4^h$	Total Nodes height $h = \frac{4^{h+1}-1}{3}$
0	1	1
1	4	1+4
2	4×4	$1+ 4 \times 4$
3	$4 \times 4 \times 4$	$1+ 4 \times 4 + 4 \times 4 \times 4$

For a given height h the maximum possible nodes are: $\frac{4^{h+1}-1}{3}$. To get minimum height, take logarithm on both sides:

$$n = \frac{4^{h+1}-1}{3} \Rightarrow 4^{h+1} = 3n + 1 \Rightarrow (h+1)log4 = \log(3n+1) \Rightarrow h+1 = \log_4(3n+1) \Rightarrow h = \log_4(3n+1) - 1$$

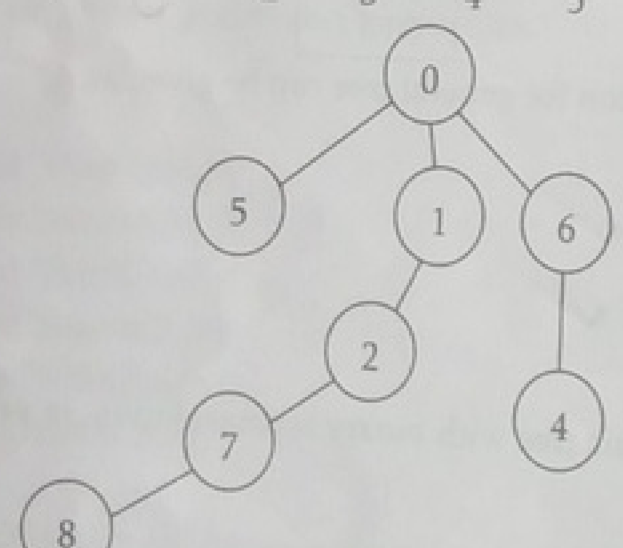
Problem-39 Given a parent array P, where $P[i]$ indicates the parent of i^{th} node in the tree (assume parent of root node is indicated with -1). Give an algorithm for finding the height or depth of the tree.

Solution:

For example: if the P is

-1	0	1	6	6	0	0	2	7
0	1	2	3	4	5	6	7	8

Its corresponding tree is:



From the problem definition, the given array is representing the parent array. That means, we need to consider the tree for that array and find the depth of the tree. The depth of this given tree is 4. If we carefully observe, we just need to start at every node and keep going to its parent until we reach -1 and also keep track of the maximum depth among all nodes.

```
int FindDepthInGenericTree(int P[], int n){
    int maxDepth =-1, currentDepth =-1, j;
    for (int i = 0; i < n; i++)   {
```



```
            currentDepth = 0; j = i;
            while(P[j] != -1) {
                    currentDepth++; j = P[j];
            }
            if(currentDepth > maxDepth)
                    maxDepth = currentDepth;
    }
    return maxDepth;
}
```

Time Complexity: O(n^2). For skew trees we will be re-calculating the same values. Space Complexity: O(1).

Note: We can optimize the code by storing the previous calculated nodes depth in some hash table or other array. This reduces the time complexity but uses extra space.

Problem-40 Given a node in the generic tree, give an algorithm for counting the number of siblings for that node.

Solution: Since tree is represented with first child/next sibling method, the tree structure can be given as:

```
struct TreeNode{
    int data;
    struct TreeNode *firstChild;
    struct TreeNode *nextSibling;
};
```

For a given node in the tree, we just need to traverse all its nextsiblings.

```
int SiblingsCount(struct TreeNode *current){
    int count = 0;
    while(current) {
        count++;
        current = current→nextSibling;
    }
    reutrn count;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-41 Given a node in the generic tree, give an algorithm for counting the number of children for that node.

Solution: Since the tree is represented as first child/next sibling method, the tree structure can be given as:

```
struct TreeNode{
    int data;
    struct TreeNode *firstChild;
    struct TreeNode *nextSibling;
};
```

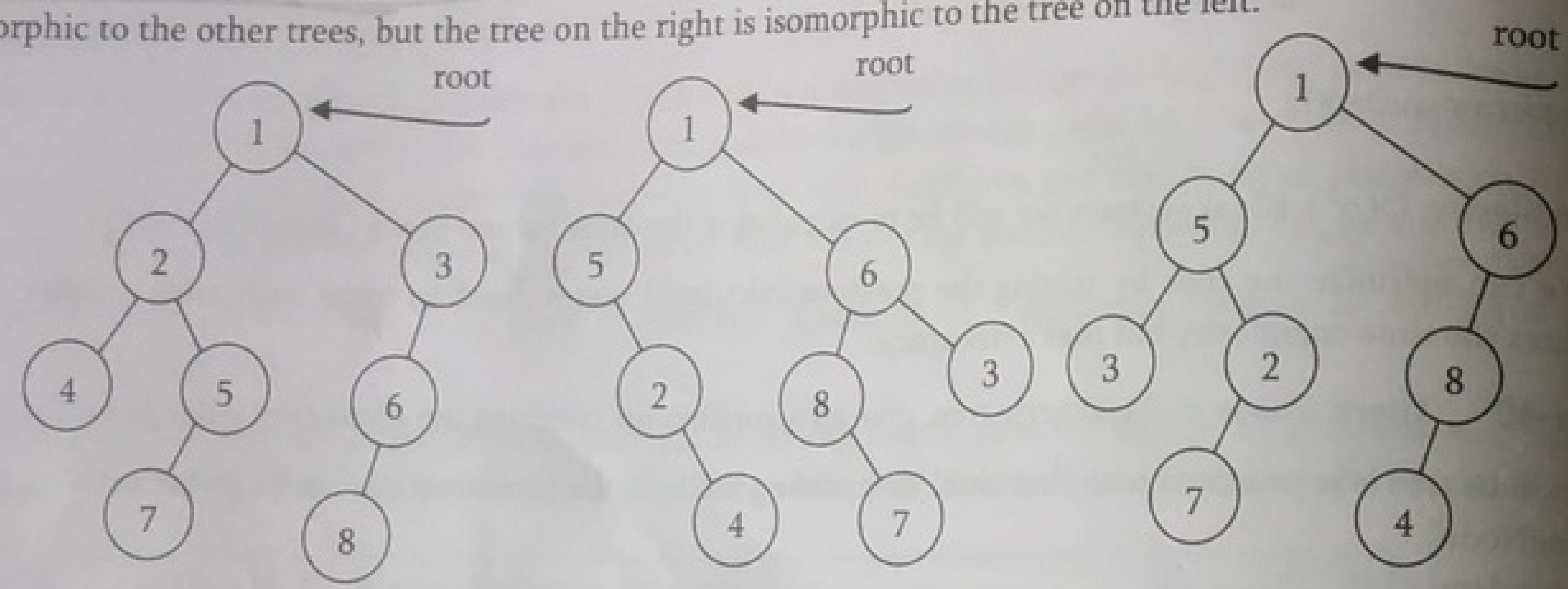
For a given node in the tree, we just need to point to its first child and keep traversing all its nextsiblings.

```
int ChildCount(struct TreeNode *current){
    int count = 0;
    current = current→firstChild;
    while(current)   {
        count++;
        current = current→nextSibling;
    }
    reutrn count;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Problem-42 Given two trees how do we check whether the trees are isomorphic to each other or not?

Solution: Two binary trees *root*1 and *root*2 are isomorphic if they have the same structure. The values of the nodes does not affect whether two trees are isomorphic or not. In the diagram below, the tree in the middle is not isomorphic to the other trees, but the tree on the right is isomorphic to the tree on the left.

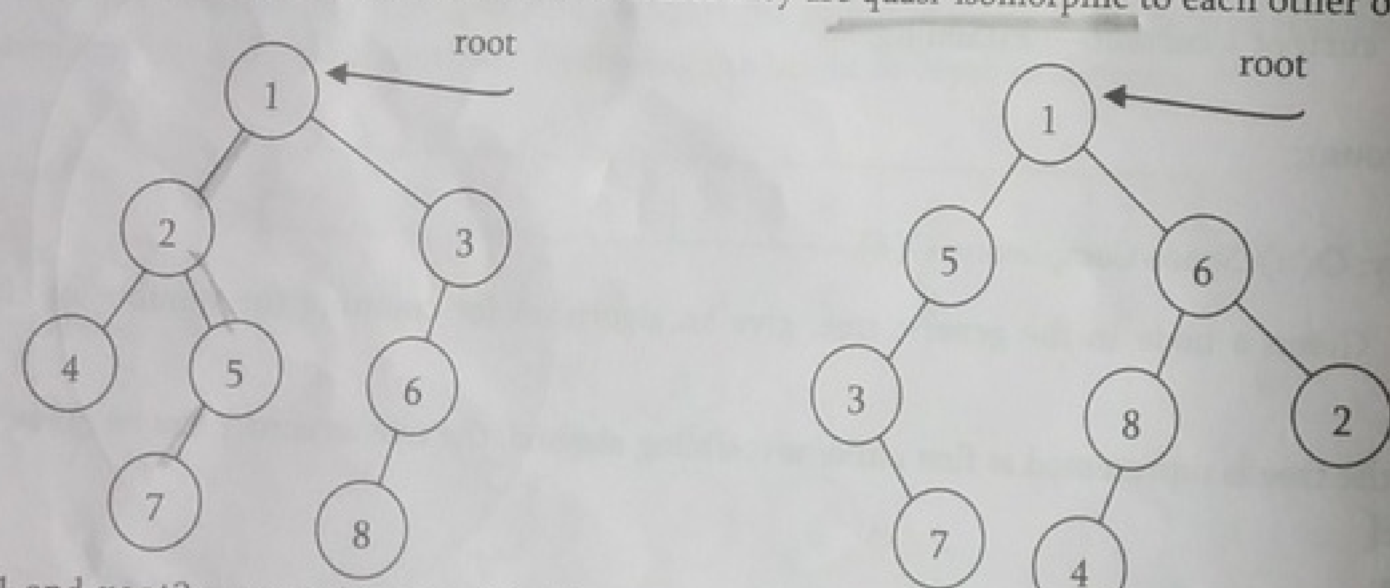



```
int IsIsomorphic(struct TreeNode *root1, struct TreeNode *root2){
    if(!root1 && !root2) return 1;
    if((!root1 && root2) || (root1 && !root2))
        return 0;
    return (IsIsomorphic(root1→left, root2→left) && IsIsomorphic(root1→right, root2→right));
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-43 Given two trees how do we check whether they are quasi-isomorphic to each other or not?

Solution:



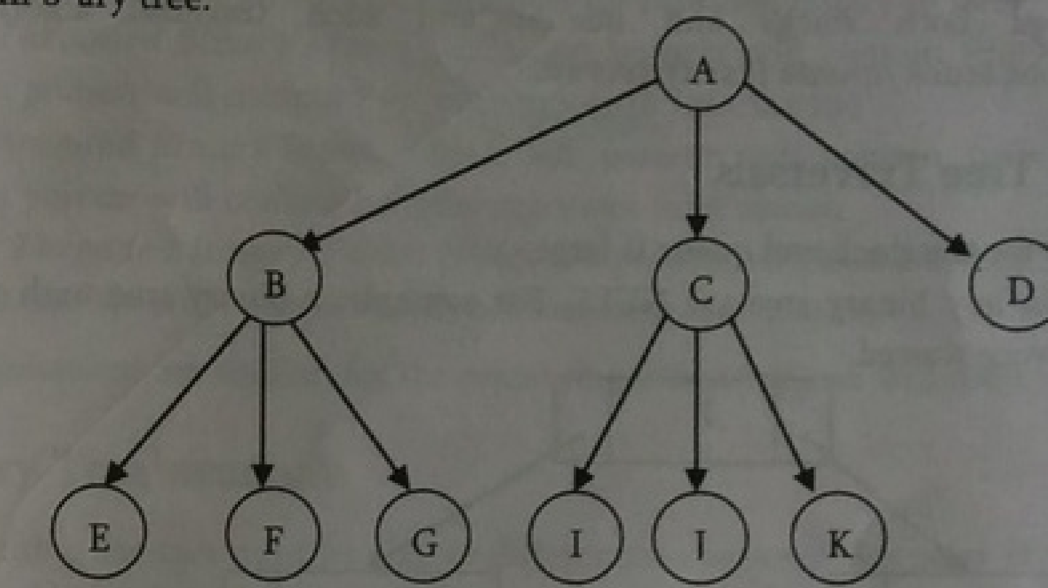
Two trees *root*1 and *root*2 are quasi-isomorphic if *root*1 can be transformed into *root*2 by swapping left and right children of some of the nodes of *root*1. The data in the nodes are not important in determining quasi-isomorphism, only the shape is important. The trees below are quasi-isomorphic because if the children of the nodes on the left are swapped, the tree on the right is obtained.

```
int QuasiIsomorphic(struct TreeNode *root1, struct TreeNode *root2){
    if(!root1 && !root2) return 1;
    if((!root1 && root2) || (root1 && !root2))
        return 0;
    return (QuasiIsomorphic(root1→left, root2→left) && QuasiIsomorphic(root1→right, root2→right)
        || QuasiIsomorphic(root1→right, root2→left) && QuasiIsomorphic(root1→left, root2→right));
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

Problem-44 A full k –ary tree is a tree where each node has either 0 or k children. Given an array which contains the preorder traversal of full k –ary tree, give an algorithm for constructing the full k –ary tree.

Solution: In k –ary tree, for a node at i^{th} position its children will be at $k * i + 1$ to $k * i + k$. For example, the below is an example for full 3-ary tree.



As we have seen, in preorder traversal first left subtree is processed then followed by root node and right subtree. Because of this, to construct a full k-ary, we just need to keep on creating the nodes without bothering about the previous constructed nodes. We can use this trick to build the tree recursively by using one global index. Declaration for k-ary tree can be given as:

```
struct K-aryTreeNode{
    char data;
    struct K-aryTreeNode *child[];
};
int *Ind = 0;
struct K-aryTreeNode *BuildK-aryTree(char A[], int n, int k){
    if(n<=0)
        return NULL;
    struct K-aryTreeNode *newNode = (struct K-aryTreeNode*) malloc(sizeof(struct K-aryTreeNode));
    if(!newNode) {
        printf("Memory Error");
        return;
    }
    newNode→child = (struct K-aryTreeNode*) malloc( k * sizeof(struct K-aryTreeNode));
    if(!newNode→child) {
        printf("Memory Error");
        return;
    }
    newNode→data = A[Ind];
    for (int i = 0; i<k; i++) {
        if(k * Ind + i <n) {
            Ind++;
            newNode→child[i] = BuildK-aryTree(A, n, k,Ind );
        }
        else newNode→child[i] =NULL;
    }
    return newNode;
}
```

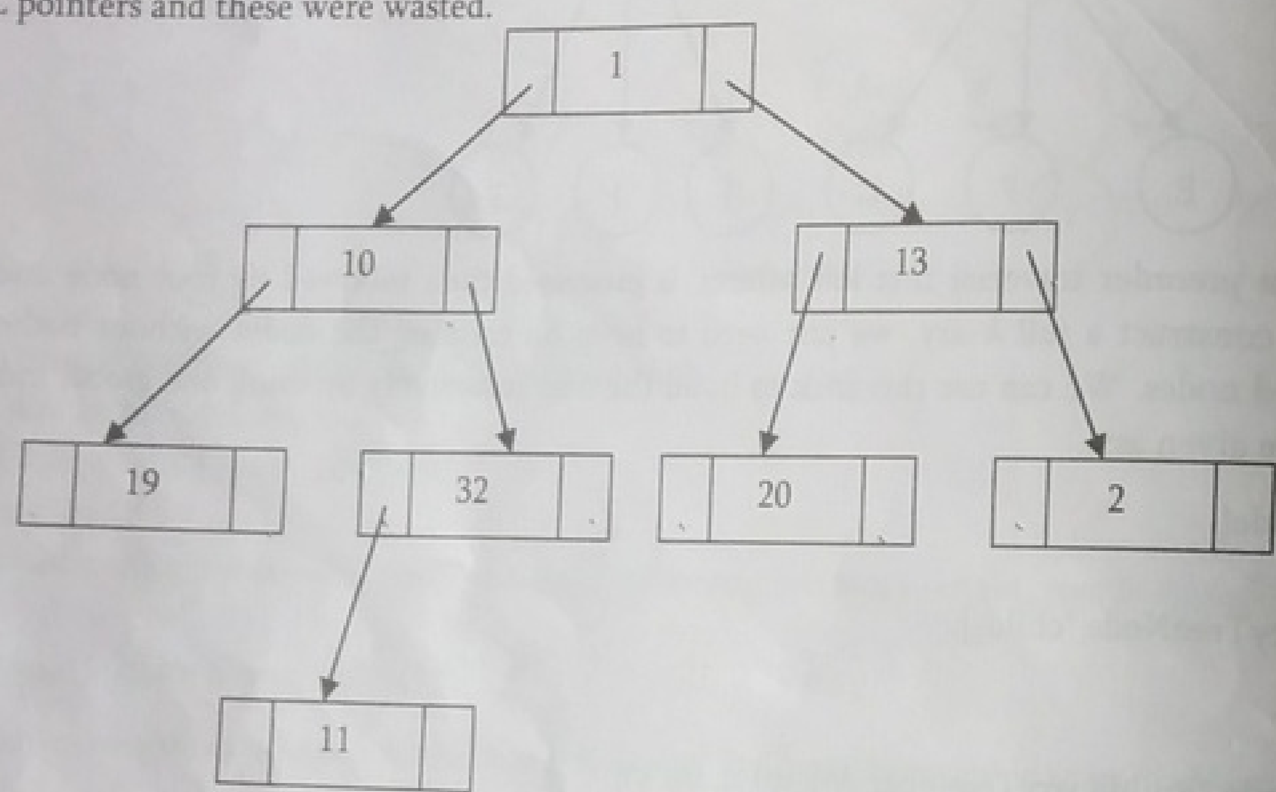
Time Complexity: $O(n)$, where n is the size of the pre-order array. This is because we are moving sequentially and not visiting the already constructed nodes.

6.8 Threaded Binary Tree Traversals [Stack or Queue less Traversals]

In earlier sections we have seen that, *preorder, inorder and postorder* binary tree traversals used stacks and *level order* traversal used queues as an auxiliary data structure. In this section we will discuss new traversal algorithms which do not need both stacks and queues and such traversal algorithms are called *threaded binary tree traversals* or *stack/queue less traversals*.

Issues with Regular Binary Tree Traversals

- The storage space required for the stack and queue is large.
- The majority of pointers in any binary tree are NULL. For example, a binary tree with n nodes has $n + 1$ NULL pointers and these were wasted.



- It is difficult to find successor node (preorder, inorder and postorder successors) for a given node.

Motivation for Threaded Binary Trees

To solve these problems, one idea is to store some useful information in NULL pointers. If we observe previous traversals carefully, stack/queue is required because we have to record the current position in order to move to right subtree after processing the left subtree. If we store the useful information in NULL pointers, then we don't have to store such information in stack/queue. The binary trees which store such information in NULL pointers are called *threaded binary trees*. From the above discussion, let us assume that we have decided to store some useful information in NULL pointers. The next question is what to store?

The common convention is 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 information and NULL right pointer will contain preorder successor information. These special pointers are called *threads*.

Classifying Threaded Binary Trees

The classification is based on whether we are storing useful information in both NULL pointers or only in one of them.

- If we store predecessor information in NULL left pointers only then we call such binary trees as *left threaded binary trees*.
- If we store successor information in NULL right pointers only then we call such binary trees as *right threaded binary trees*.
- If we store predecessor information in NULL left pointers only then we call such binary trees as *fully threaded binary trees* or simply *threaded binary trees*.

Note: For the remaining discussion we consider only (*fully*) *threaded binary trees*.

Types of Threaded Binary Trees

Based on above discussion we get three representations for threaded binary trees.

- *Preorder Threaded Binary Trees*: NULL left pointer will contain PreOrder predecessor information and NULL right pointer will contain PreOrder successor information
- *Inorder Threaded Binary Trees*: NULL left pointer will contain InOrder predecessor information and NULL right pointer will contain InOrder successor information
- *Postorder Threaded Binary Trees*: NULL left pointer will contain PostOrder predecessor information and NULL right pointer will contain PostOrder successor information

Note: As the representations are similar, for the remaining discussion, we will use InOrder threaded binary trees.

Threaded Binary Tree structure

Any program examining the tree must be able to differentiate between a regular *left/right* pointer and a *thread*. To do this, we use two additional fields into each node giving us, for threaded trees, nodes of the following form:

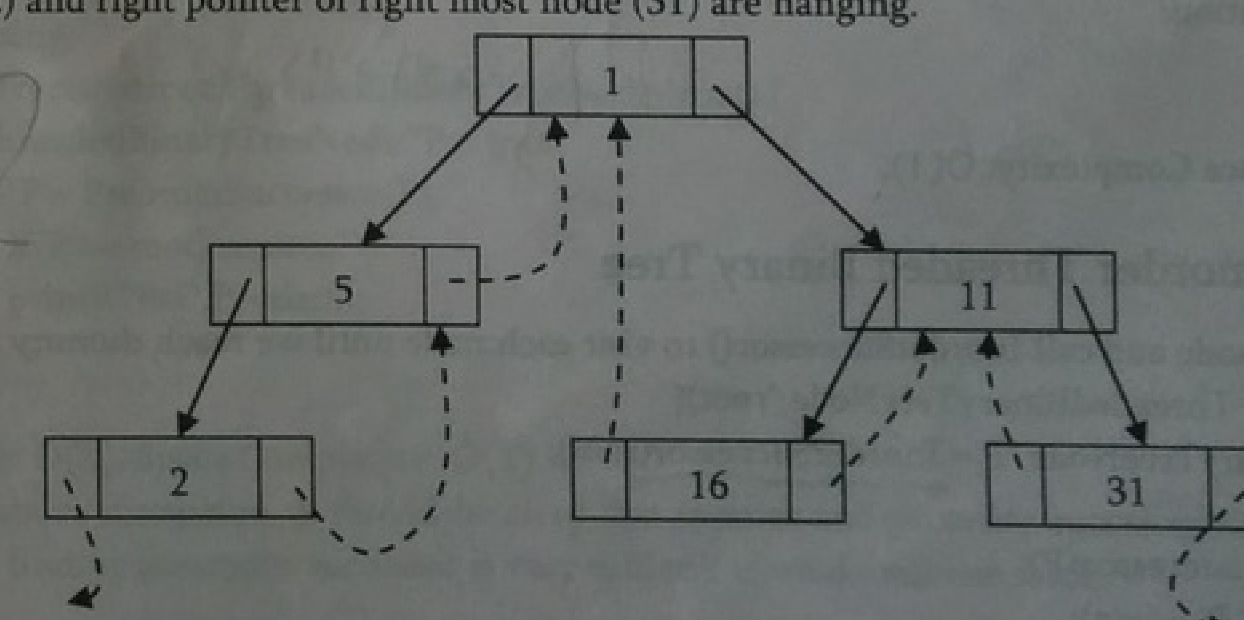

```
struct ThreadedBinaryTreeNode{
        struct ThreadedBinaryTreeNode *left;
        int LTag;
        int data;
        int RTag;
        struct ThreadedBinaryTreeNode *right;
};
```

Difference between Binary Tree and Threaded Binary Tree Structures

	Regular Binary Trees	Threaded Binary Trees
if LTag == 0	NULL	left points to the in-order predecessor
if LTag == 1	left points to the left child	left points to left child
if RTag == 0	NULL	right points to the in-order successor
if RTag == 1	right points to the right child	right points to the right child

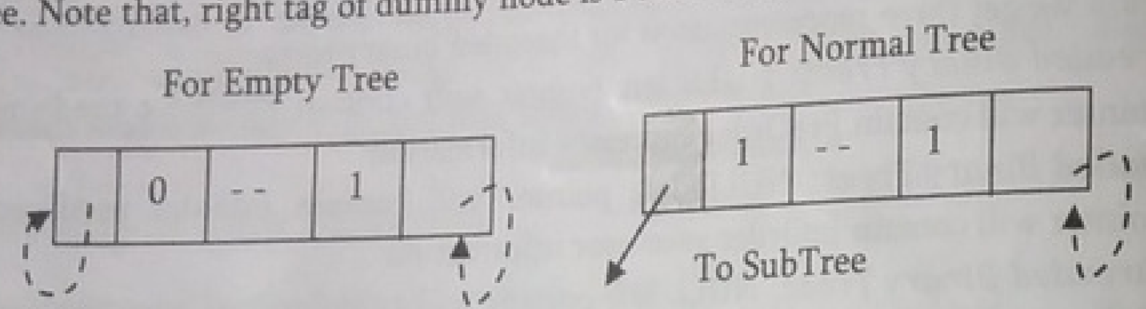
Note: Similarly, we can define for preorder/postorder differences as well.

As an example, let us try representing a tree in inorder threaded binary tree form. The below tree shows how an inorder threaded binary tree will look like. The dotted arrows indicate the threads. If we observe, the left pointer of left most node (2) and right pointer of right most node (31) are hanging.

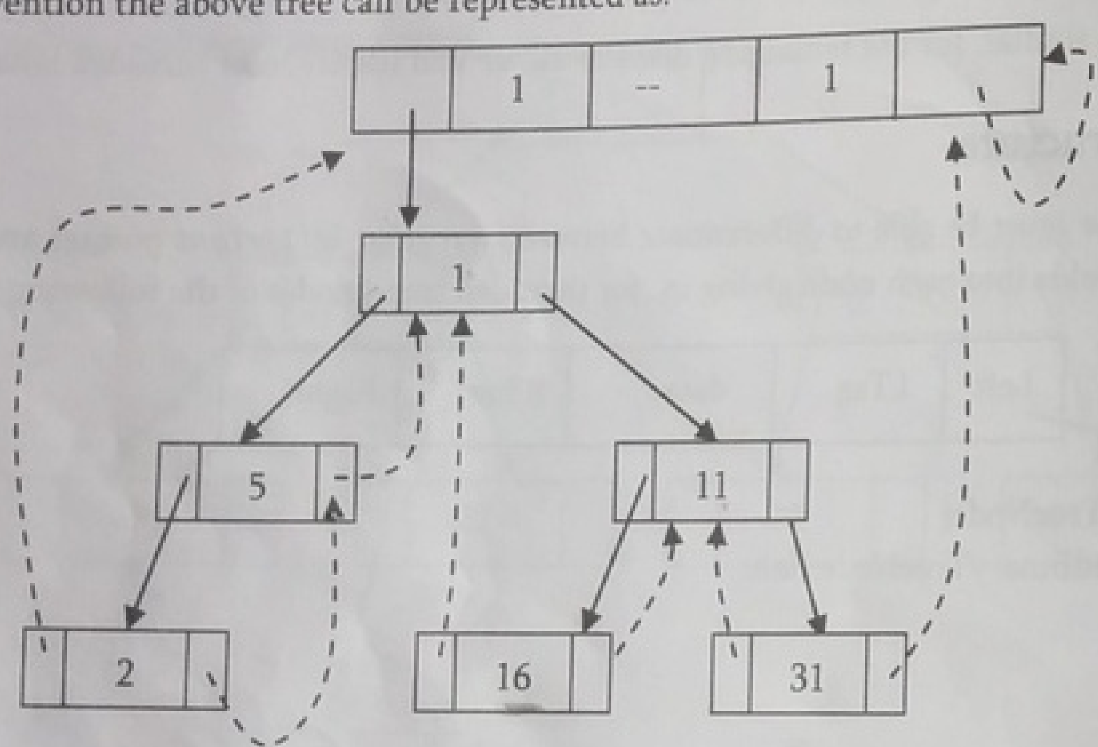


What should leftmost and rightmost pointers point to?

In the representation of a threaded binary tree, it is convenient to use a special node *Dummy* which is always present even for an empty tree. Note that, right tag of dummy node is 1 and its right child points to itself.



With this convention the above tree can be represented as:



Finding Inorder Successor in Inorder Threaded Binary Tree

To find inorder successor of a given node without using a stack, assume that the node for which we want to find the inorder successor is *P*.

Strategy: If *P* has a no right subtree, then return the right child of *P*. If *P* has right subtree, then return the left of the nearest node whose left subtree contains *P*.

```
struct ThreadedBinaryTreeNode* InorderSuccessor(struct ThreadedBinaryTreeNode *P){
        struct ThreadedBinaryTreeNode *Position;
        if(P→RTag == 0) return P→right;
        else {    Position = P→right;
                  while(Position→LTag == 1)
                          Position = Position→left;
                  return Position;
        }
}
```

predessor
if(P->LTag == 0) return P->LTag
else{ pos = p->Left
while(pos->RTag == 1) {
pos = pos->Right
}
ret pos }
}

Time Complexity: O(*n*). Space Complexity: O(1).

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 ThreadedBinaryTreeNode *root){
        struct ThreadedBinaryTreeNode *P = InorderSuccessor(root);
        while(P != root) {
                P = InorderSuccessor(P);
                printf("%d",P→data);
```



```
        }
}
```

Other way of coding:

```
void InorderTraversal(struct ThreadedBinaryTreeNode *root){
        struct ThreadedBinaryTreeNode *P = root;
        while(1) {  P = InorderSuccessor(P);
                    if(P == root) return;
                    printf("%d",P→data);
        }
}
```

Time Complexity: O(*n*). Space Complexity: O(1).

Finding PreOrder Successor in InOrder Threaded Binary Tree

Strategy: If *P* has a left subtree, then return the left child of *P*. If *P* has no left subtree, then return the right child of the nearest node whose right subtree contains *P*.

```
struct ThreadedBinaryTreeNode* PreorderSuccessor(struct ThreadedBinaryTreeNode *P){
        struct ThreadedBinaryTreeNode *Position;
        if(P→LTag == 1) return P→left;
        else {    Position = P;
                  while(Position→RTag == 0)
                          Position = Position→right;
                  return Position→right;
        }
}
```

Time Complexity: O(*n*). Space Complexity: O(1).

PreOrder Traversal of InOrder Threaded Binary Tree

As similar to inorder traversal, start with *dummy* node and call PreorderSuccessor() to visit each node until we get *dummy* node again.

```
void PreorderTraversal(struct ThreadedBinaryTreeNode *root){
        struct ThreadedBinaryTreeNode *P;
        P = PreorderSuccessor(root);
        while(P != root) {
                P = PreorderSuccessor(P);
                printf("%d",P→data);
        }
}
```

Other way of coding:

```
void PreorderTraversal(struct ThreadedBinaryTreeNode *root) {
        struct ThreadedBinaryTreeNode *P = root;
        while(1){P = PreorderSuccessor(P);
                 if(P == root) return;
                 printf("%d",P→data);
        }
}
```

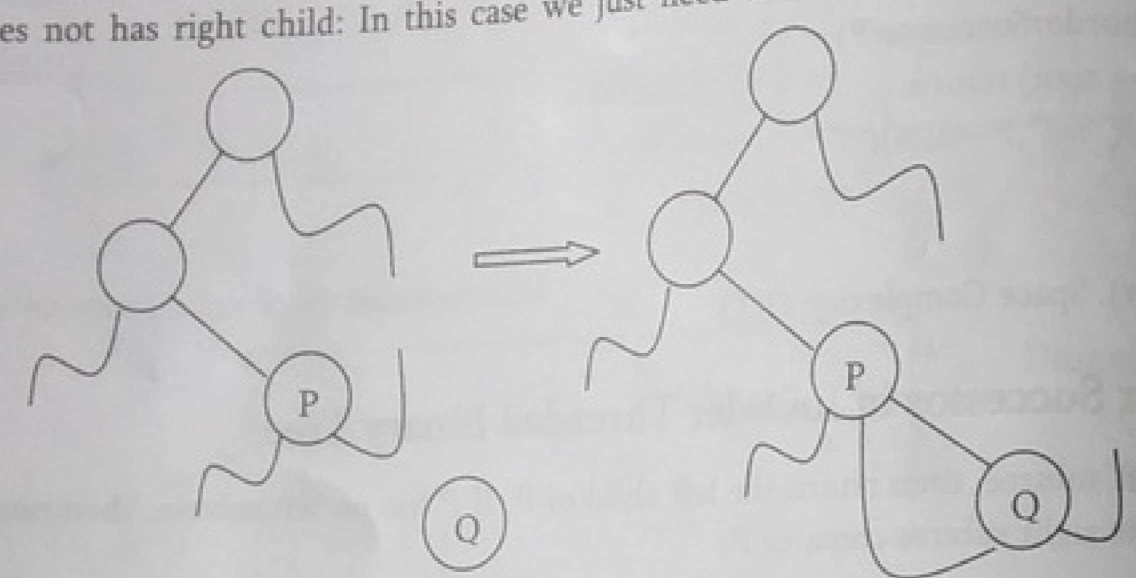
Time Complexity: O(*n*). Space Complexity: O(1).

Note: From the above discussion, it should be clear that inorder and preorder successor finding is easy with threaded binary trees. But finding postorder successor is very difficult if we do not use stack.

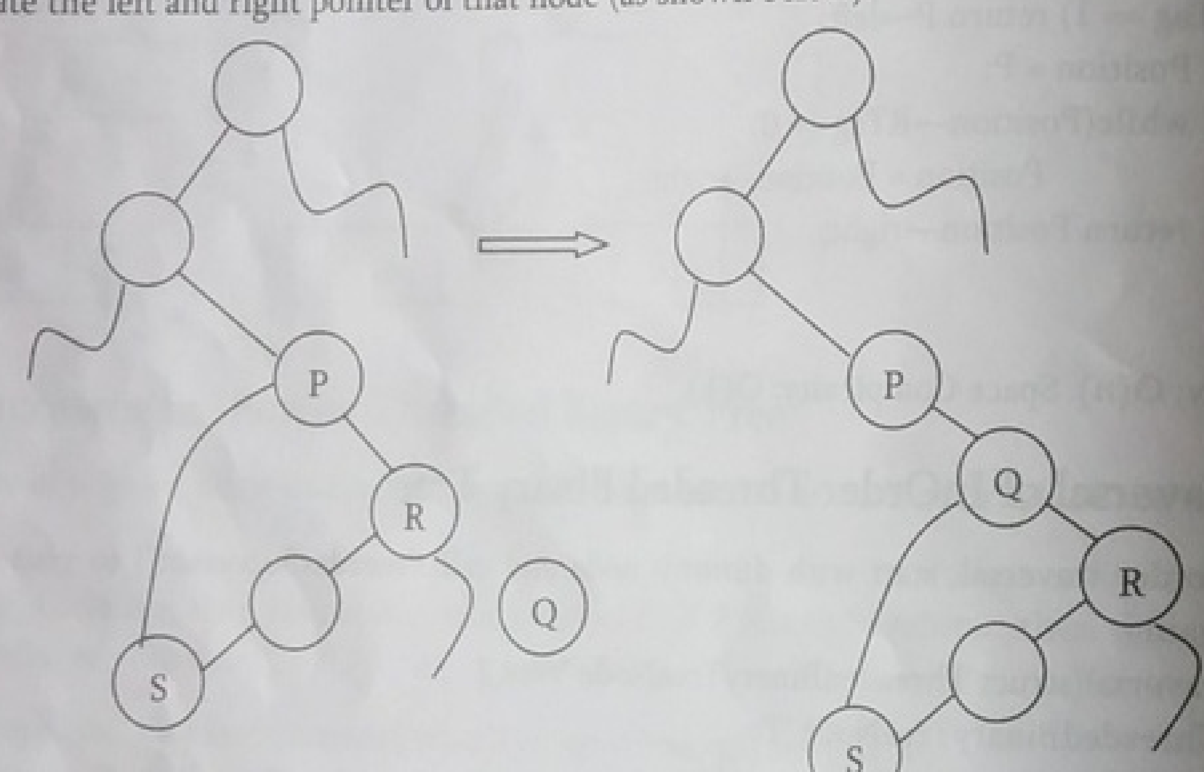
Insertion of Nodes in InOrder Threaded Binary Trees

For simplicity, let us assume that there are two nodes *P* and *Q* and we want to attach *Q* to right of *P*. For this we will have two cases.

- Node *P* does not has right child: In this case we just need to attach *Q* to *P* and change its left and right pointers.



- Node P has right child (say, *R*): In this case we need to traverse *R's* left subtree and find the left most node and then update the left and right pointer of that node (as shown below).




```
void InsertRightInInorderTBT(struct ThreadedBinaryTreeNode *P, struct ThreadedBinaryTreeNode *Q){
    struct ThreadedBinaryTreeNode *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) {                    //Case-2
        Temp = Q→right;
        while(Temp→LTag)
            Temp = Temp→left;
        Temp→left = Q;
    }
}
```

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Problems on Threaded binary Trees

Problem-45 For a given binary tree (not threaded) how do we find the preorder successor?

Solution: For solving this problem, we need to use an auxiliary stack *S*. On the first call, the parameter node is a pointer to the head of the tree, thereafter its value is NULL. Since we are simply asking for the successor of the node we got last time we called the function. It is necessary that the contents of the stack *S* and the pointer *P* to the last node "visited" are preserved from one call of the function to the next, they are defined as static variables.

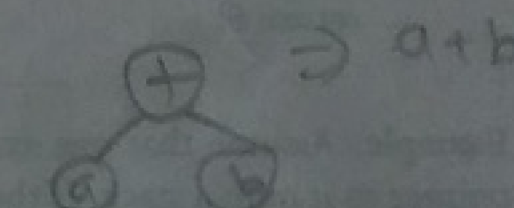
```
// pre-order successor for an unthreaded binary tree
struct BinaryTreeNode *PreorderSuccssor(struct BinaryTreeNode *node){
    static struct BinaryTreeNode *P;
    static Stack *S = CreateStack();
    if(node != NULL)
        P = node;
    if(P→left != NULL) {
        Push(S,P);
        P = P→left;
    }
    else {  while (P→right == NULL)
                P = Pop(S);
        P = P→right;
    }
    return P;
}
```

Problem-46 For a given binary tree (not threaded) how do we find the inorder successor?

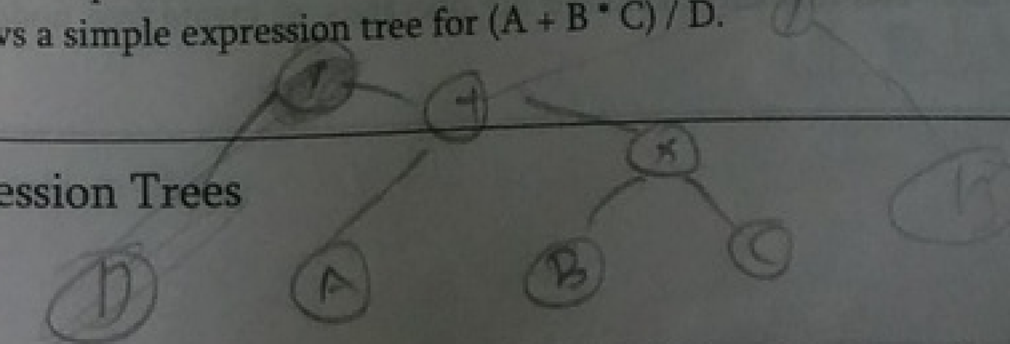
Solution: Similar to above discussion, we can find the inorder successor of a node as:

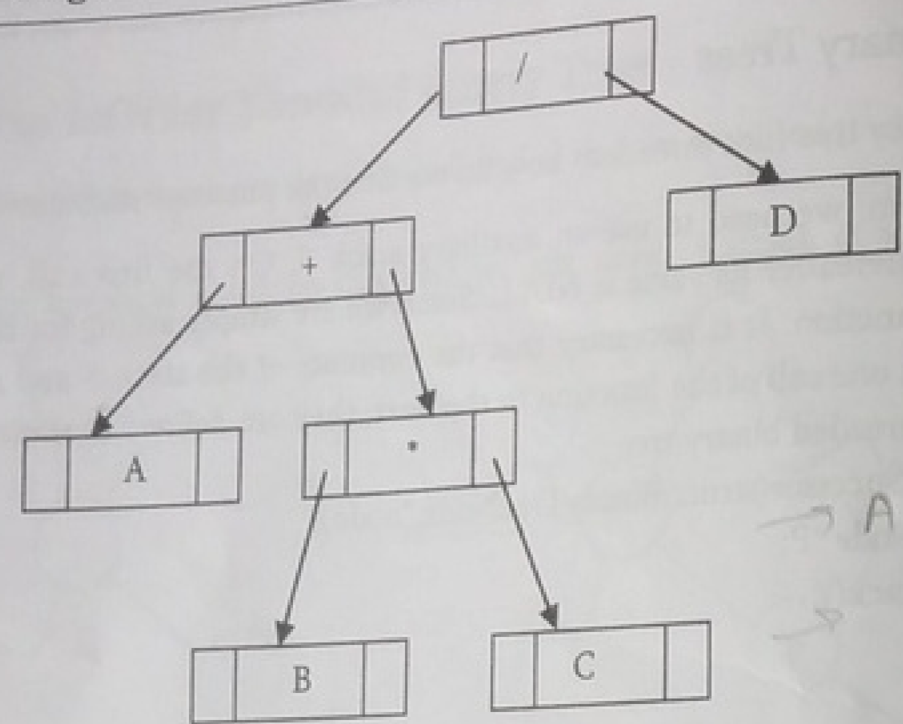
```
// In-order successor for an unthreaded binary tree
struct BinaryTreeNode *InorderSuccssor(struct BinaryTreeNode *node){
    static struct BinaryTreeNode *P;
    static Stack *S = CreateStack();
    if(node != NULL)
        P = node;
    if(P→right == NULL)
        P = Pop(S);
    else {  P = P→right;
        while (P→left != NULL)
            Push(S, P);
        P = P→left;
    }
    return P;
}
```

6.9 Expression Trees



A tree representing an expression is called as an expression tree. In expression trees leaf nodes are operands and non-leaf nodes are operators. That means, an expression tree is a binary tree where internal nodes are operators and leaves are operands. Expression tree consists of binary expression. But for a unary operator, one subtree will be empty. Below figure shows a simple expression tree for (A + B * C) / D.





→ A B C * + D /

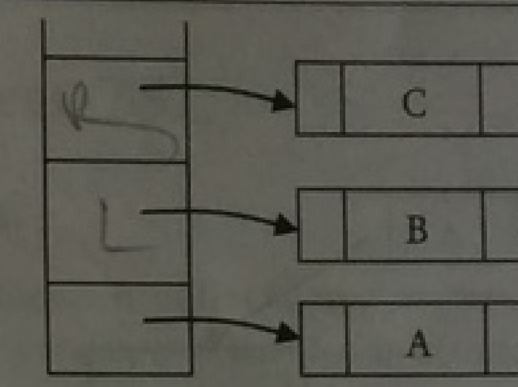
Algorithm for Building Expression Tree from Postfix Expression

```
struct BinaryTreeNode *BuildExprTree(char postfixExpr[], int size){
	struct Stack *S = Stack(size);
	for (int i = 0; i< size; i++) {
		if(postfixExpr[i] is an operand) {
			struct BinaryTreeNode newNode = (struct BinaryTreeNode*)
					malloc( sizeof (struct BinaryTreeNode));
			if(!newNode) {
				printf("Memory Error");
				return NULL;
			}
			newNode→data =postfixExpr[i];
			newNode→left = newNode→right = NULL;
			Push(S, newNode);
		}
		else {
			struct BinaryTreeNode *T2 = Pop(S), *T1 = Pop(S);
			struct BinaryTreeNode newNode = (struct BinaryTreeNode*)
					malloc(sizeof(struct BinaryTreeNode));
			if(!newNode) {
				printf("Memory Error"); return NULL;
			}
			newNode→data = postfixExpr[i];
			newNode→left = T1; newNode→right = T2;
			Push(S, newNode);
		}
	}
	return S;
}
```

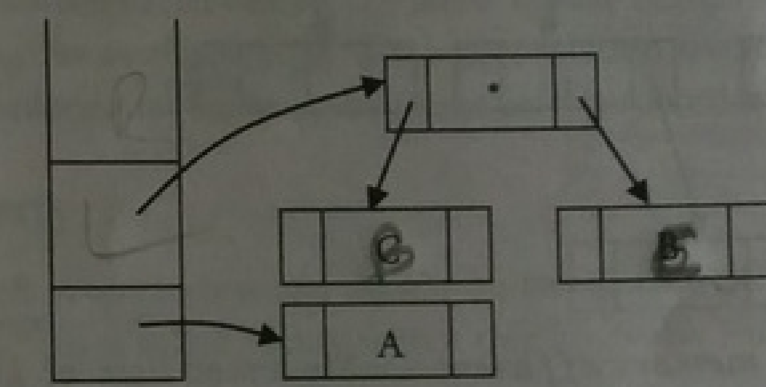
Example: Assume that one symbol is read at a time. If the symbol is an operand, we create a tree node and push a pointer to it onto a stack. If the symbol is an operator, pop pointers to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively. A pointer to this new tree is then pushed onto the stack.

As an example, assume the input is A B C * + D /. The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below.

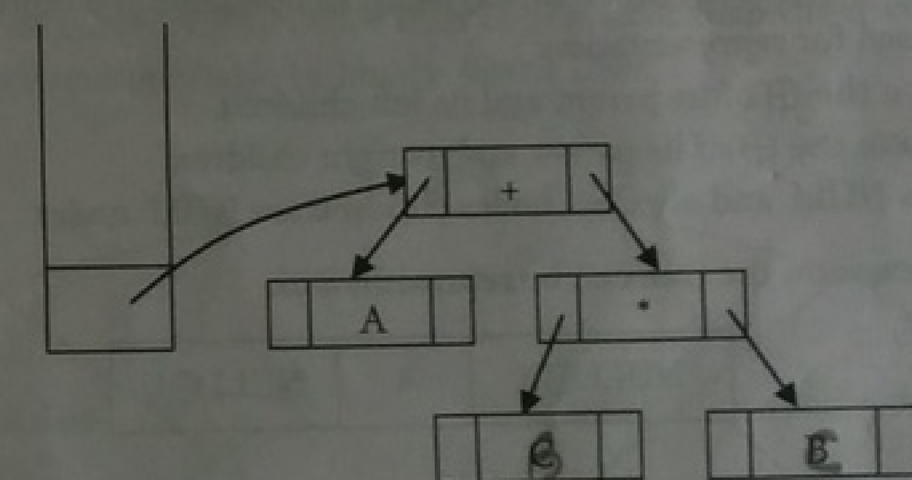
Centroid → 23 → 15 →



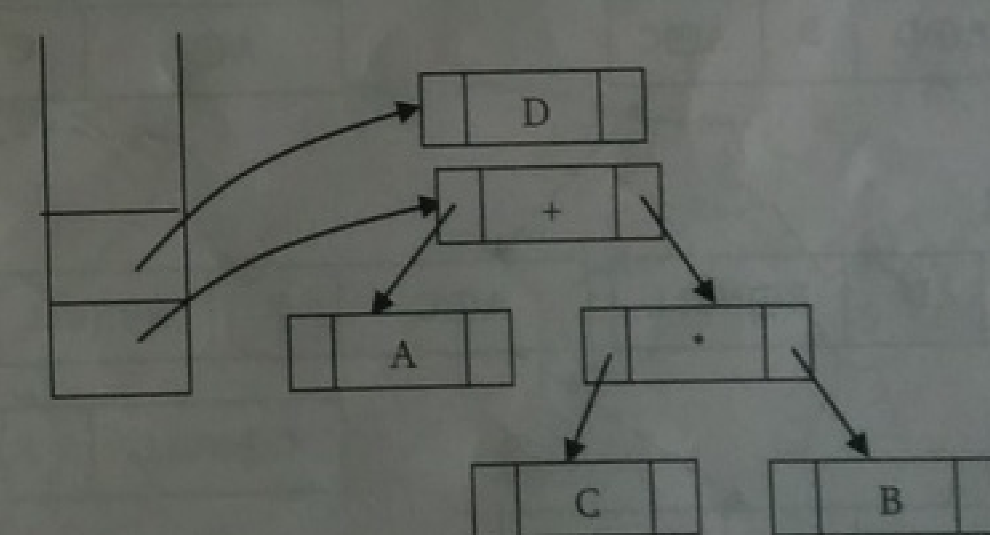
Next, an operator '*' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



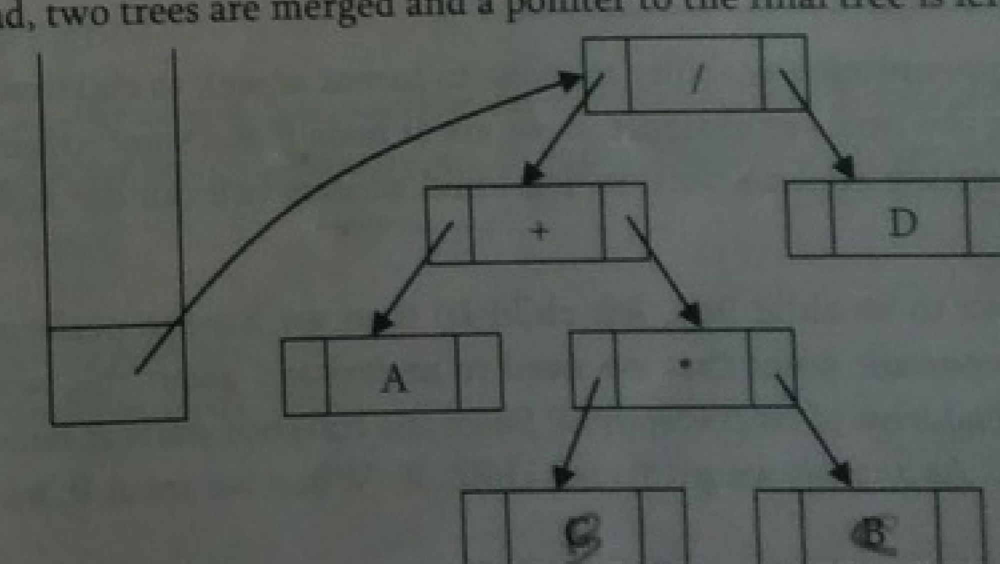
Next, an operator '+' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



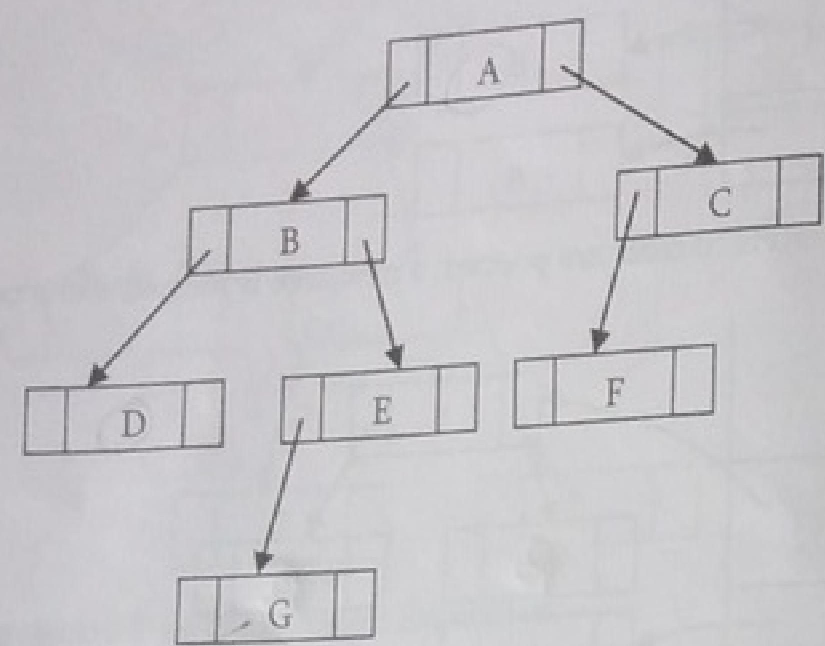
Next, an operand 'D' is read, a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



Finally, the last symbol ('/') is read, two trees are merged and a pointer to the final tree is left on the stack.



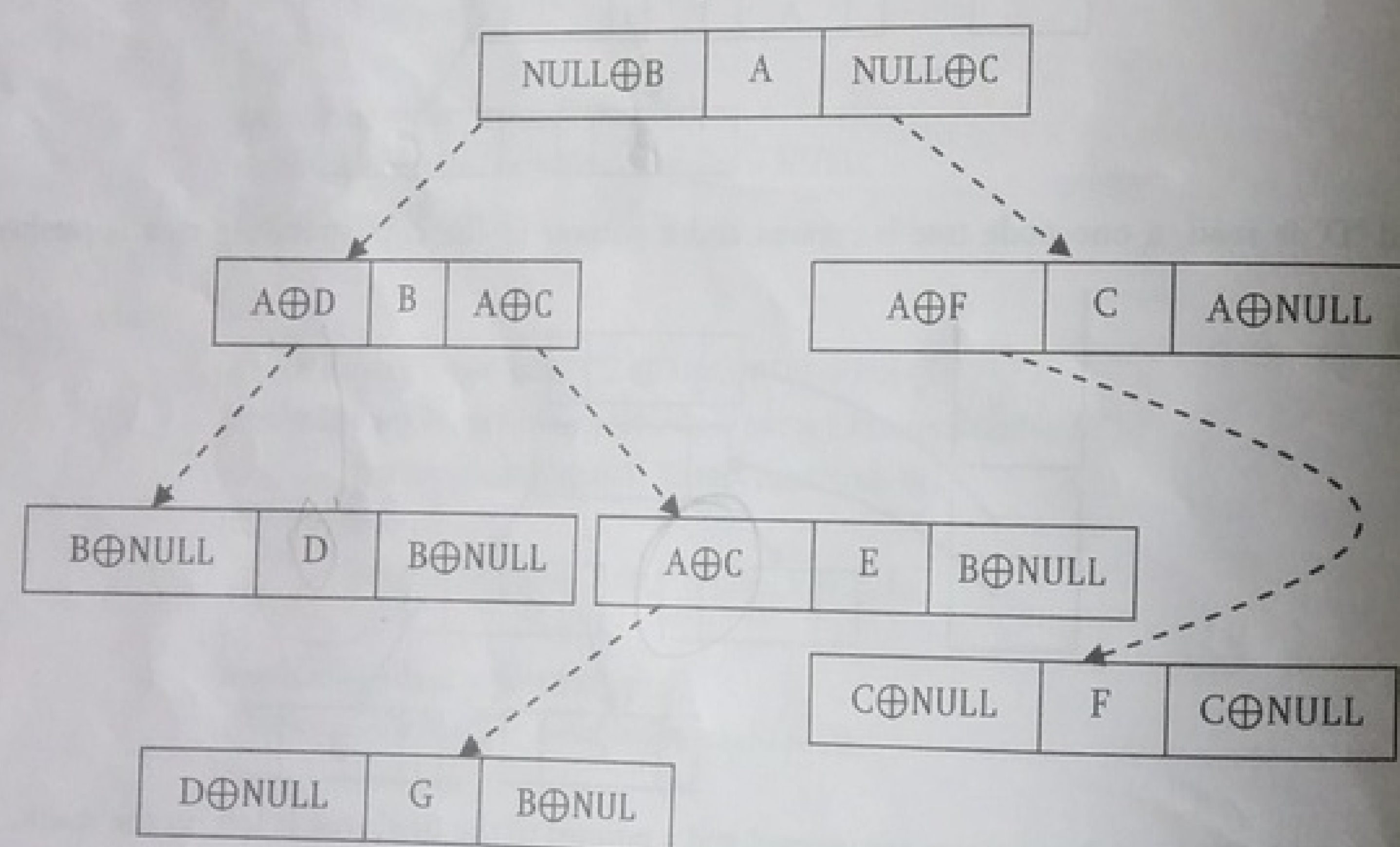
6.10 XOR Trees



This concept is very much similar to *memory efficient doubly linked lists* of *Linked Lists* chapter. Also, like threaded binary trees this representation does not need stacks or queues for traversing the trees. This representation is used for traversing back (to parent) and forth (to children) using ⊕ operation. To represent the same in XOR trees, for each node below are the rules used for representation:

- Each nodes left will have the ⊕ of its parent and its left children.
- Each nodes right will have the ⊕ of its parent and its right children.
- The root nodes parent is NULL and also leaf nodes children are NULL nodes.

Based on the above rules and discussion the tree can be represented as:



The major objective of this presentation is ability to move to parent as well to children. Now, let us see how to use this representation for traversing the tree. For example, 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 its left child address (we can use right child also for going to parent node).

Similarly, if we want to move to its child (say, left child D) then we have to perform ⊕ on its left content with its parent node address. One important point that we need to understand about this representation is: When we are at node B how do we know the address of its children D? Since the traversal starts at node root node, we can apply ⊕ on roots left content with NULL. As a result we get its left child, B. When we are at B, we can apply ⊕ on its left content with A address.

6.11 Binary Search Trees (BSTs)

Why Binary Search Trees?

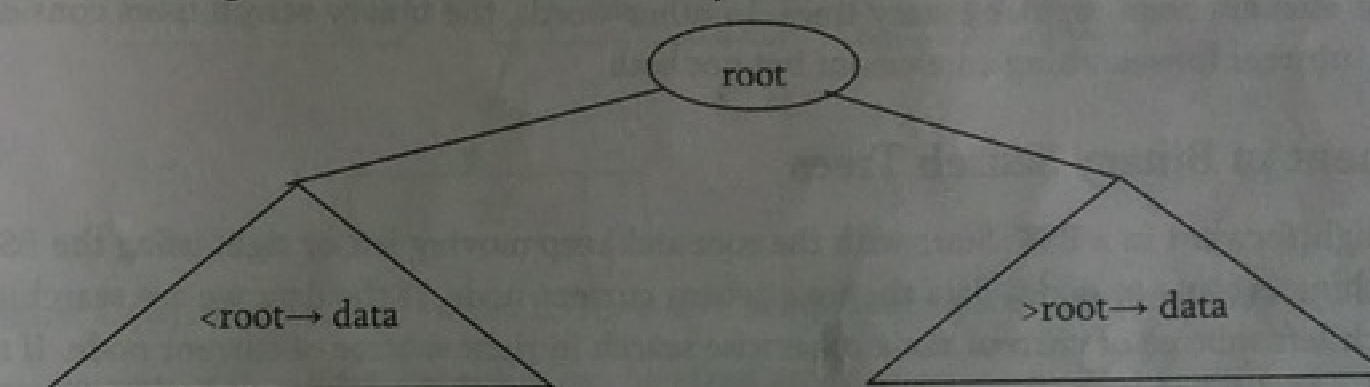
In previous sections we have discussed different tree representations and in all of them we did not impose any restriction on the nodes data. As a result, to search for an element we need to check both in left subtree and also right subtree. Due to this, the worst case complexity of search operation is O(n).

In this section, we will discuss another variant of binary trees: Binary Search Trees (BSTs). As the name suggests, the main use of this representation is for *searching*. In this representation we impose restriction on the kind of data a node can contain. As a result, it reduces the worst case average search operation to O($logn$).

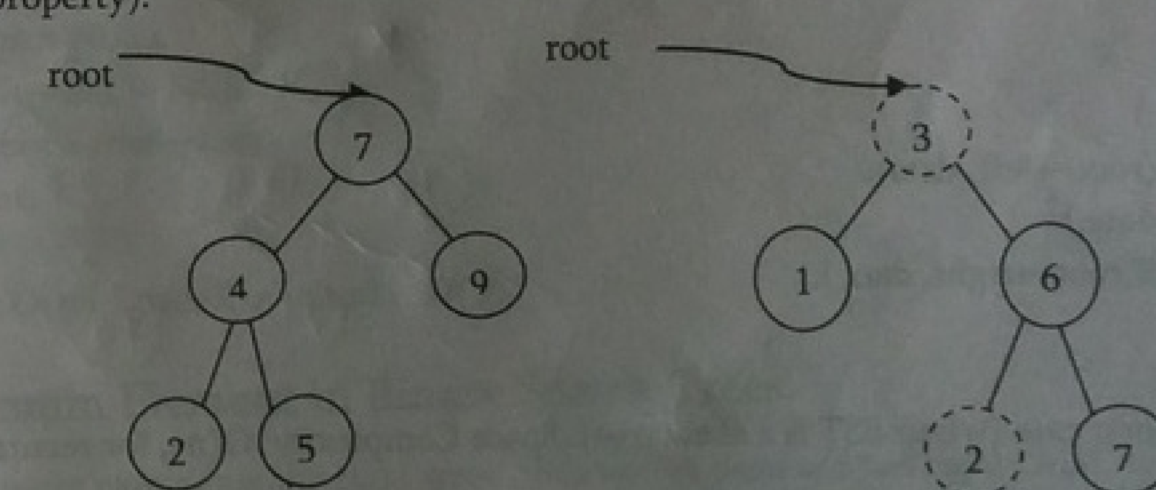
Binary Search Tree Property

In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called binary search tree property. Note that, this property should be satisfied at every node in the tree.

- The left subtree of a node contains only nodes with keys less than the nodes key.
- The right subtree of a node contains only nodes with keys greater than the nodes key.
- Both the left and right subtrees must also be binary search trees.



Example: The left tree is a binary search tree and right tree is not binary search tree (at node 6 it's not satisfying the binary search tree property).



Binary Search Tree Declaration

There is no difference between regular binary tree declaration and binary search tree declaration. The difference is only in data but not in structure. But for our convenience we change the structure name as:

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

Operations on Binary Search Trees

Main operations: The main operations that were supported by binary search trees are:

- Find/ Find Minimum / Find Maximum element in binary search trees
- Inserting an element in binary search trees
- Deleting an element from binary search trees

Auxiliary operations: Checking whether the given tree is a binary search tree or not

- Finding k^{th}-smallest element in tree
- Sorting the elements of binary search tree and many more

Important Notes on Binary Search Trees

- Since root data is always in between left subtree data and right subtree data, performing inorder traversal on binary search tree produces a sorted list.
- While solving problems on binary search trees, most of the time, first we process left subtree, process root data and then process right subtree. That means, depending on the problem only the intermediate step (processing root data) changes and we will not touch first and third steps.
- If we are searching for an element and if the left subtree roots data is less than the element we want to search then skip it. Same is the case with right subtree as well. Because of this binary search trees takes less time for searching an element than regular binary trees. In other words, the binary search trees consider only either left or right subtrees for searching an element but not both.

Inorder

Finding an Element in Binary Search Trees

Find operation is straightforward in a BST. Start with the root and keep moving left or right using the BST property. If the data we are searching is same as nodes data then we return current node. If the data we are searching is less than nodes data then search left subtree of current node otherwise search in right subtree of current node. If the data is not present, we end up in a NULL link.

```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *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: $O(n)$, in worst case (when BST is a skew tree). Space Complexity: $O(n)$, for recursive stack.

Non recursive version of the above algorithm can be given as:

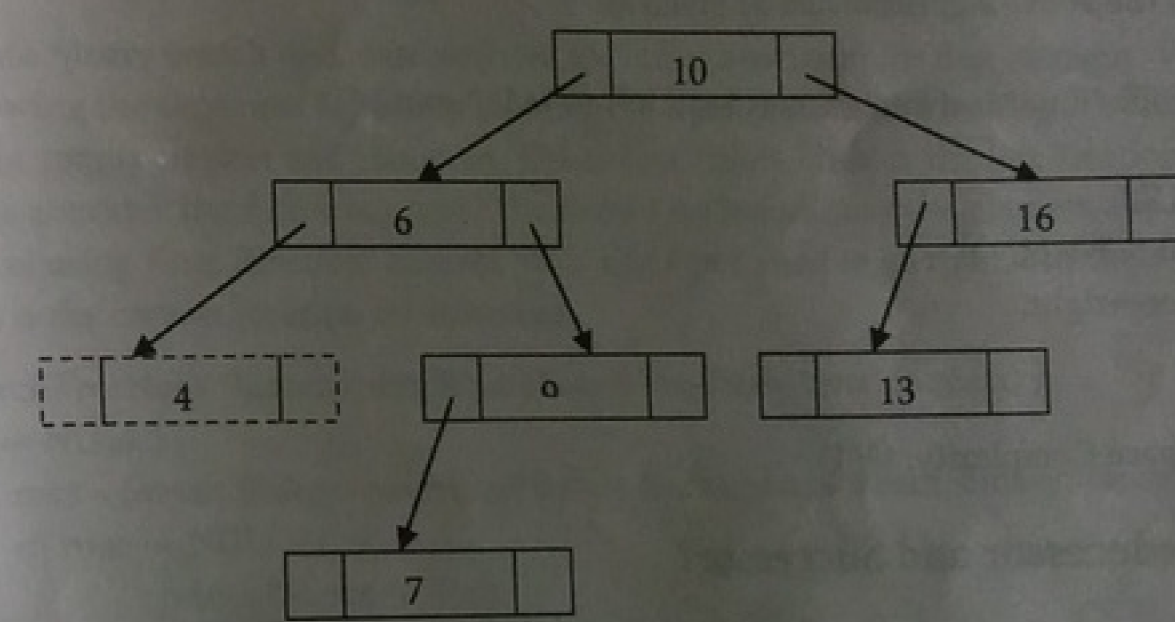
```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
    if( root == NULL ) return NULL;
    while (root) {
        if(data == root→data)
            return root;
        else if(data > root→data)
            root = root→right;
        else    root = root→left;
    }
    return NULL;
}
```


Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Finding Minimum Element in Binary Search Trees

In BSTs, the minimum element is the left most node which does not has left child. In the below BST, the minimum element is 4.

```
struct BinarySearchTreeNode *FindMin(struct BinarySearchTreeNode *root){
    if(root == NULL) return NULL;
    else    if( root→left == NULL )
                return root;
            else    return FindMin( root→left );
}
```



Time Complexity: $O(n)$, in worst case (when BST is a *left skew* tree). Space Complexity: $O(n)$, for recursive stack.

Non recursive version of the above algorithm can be given as:

```
struct BinarySearchTreeNode *FindMin(struct BinarySearchTreeNode * root ) {
    if( root == NULL )
        return NULL;
    while( root→left != NULL )
        root = root→left;
    return root;
}
```

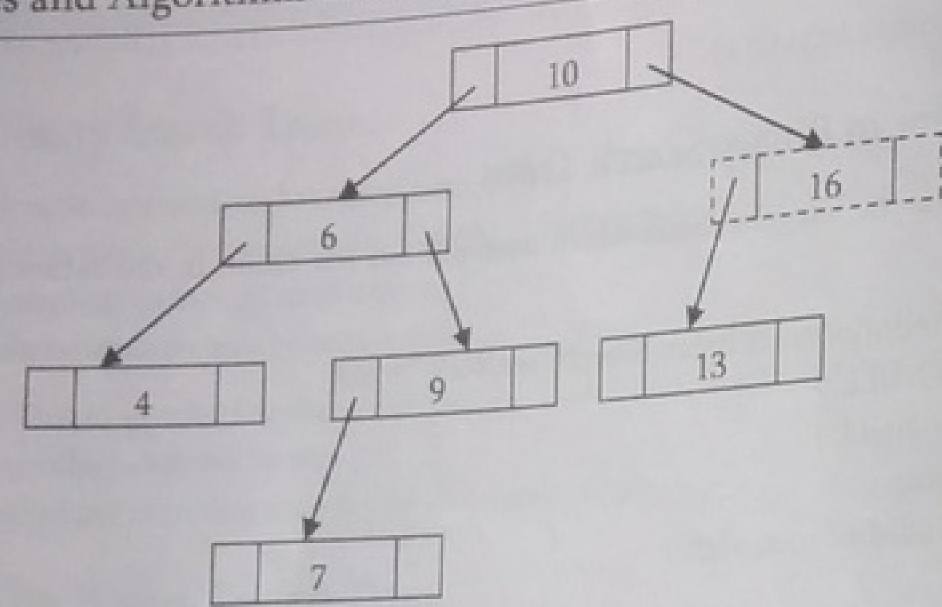
Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Finding Maximum Element in Binary Search Trees

In BSTs, the maximum element is the right most node which does not has right child. In the below BST, the maximum element is 16.

```
struct BinarySearchTreeNode *FindMax(struct BinarySearchTreeNode *root) {
    if(root == NULL)
        return NULL;
    else    if( root→right == NULL )
                return root;
            else    return FindMax( root→right );
}
```

Time Complexity: $O(n)$, in worst case (when BST is a *right skew* tree). Space Complexity: $O(n)$, for recursive stack.



Non recursive version of the above algorithm can be given as:

```
struct BinarySearchTreeNode *FindMax(struct BinarySearchTreeNode * root ) {
    if( root == NULL )
        return NULL;
    while( root→right != NULL )
        root = root→right;
    return root;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

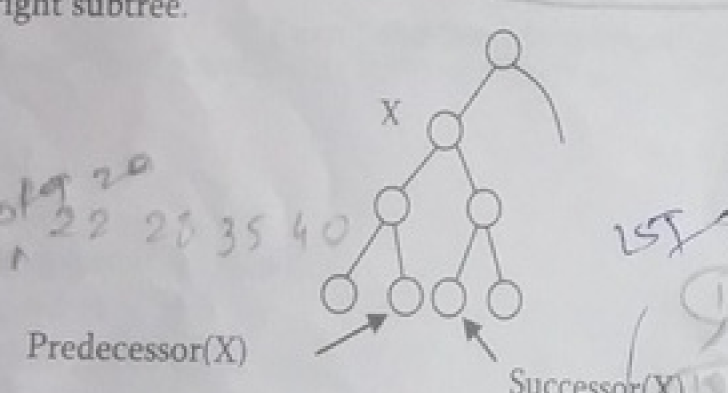
Where is Inorder Predecessor and Successor?

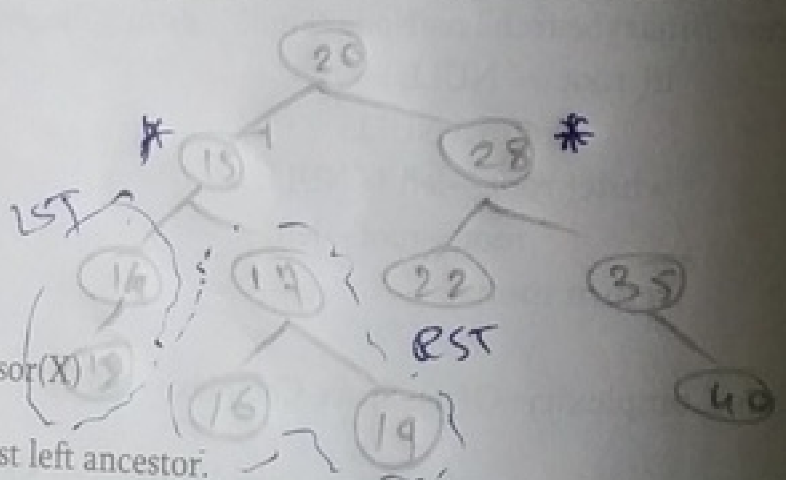
Where is the inorder predecessor and successor of a node X in a binary search tree assuming all keys are distinct?

If X has two children then its inorder predecessor is the maximum value in its left subtree and its inorder successor the minimum value in its right subtree.

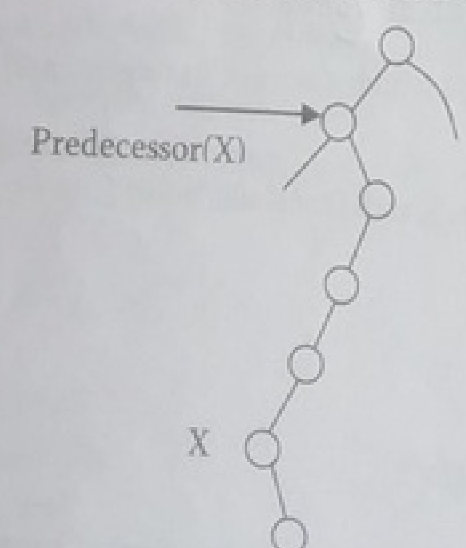
Inorder =>

max in LST min RST

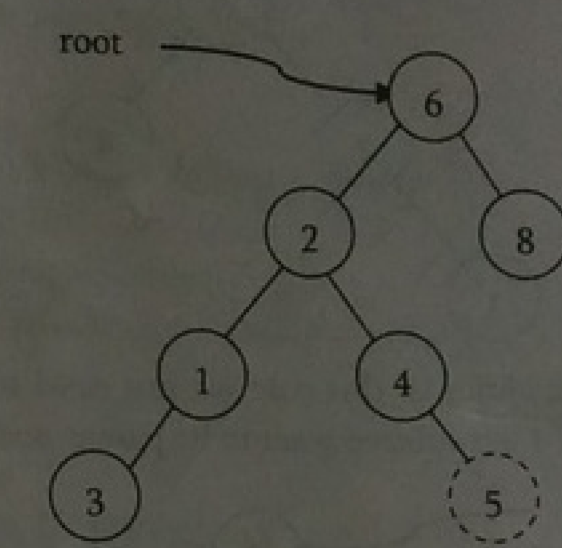




If it does not have a left child a nodes inorder predecessor is its first left ancestor.



Inserting an Element from Binary Search Tree



To insert *data* into binary search tree, first we need to find the location for that element. We can find the location of insertion by following the same mechanism as that of *find* operation. While finding the location if the *data* is already there then we can simply neglect and come out. Otherwise, insert *data* at the last location on the path traversed. As an example let us consider the following tree. The dotted node indicates the element (5) to be inserted. To insert 5, traverse the tree as using *find* function. At node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct location for insertion.

```
struct BinarySearchTreeNode *Insert(struct BinarySearchTreeNode *root, int data) {
    if( root == NULL ) {
        root = (struct BinarySearchTreeNode *) malloc(sizeof(struct BinarySearchTreeNode));
        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;
}
```

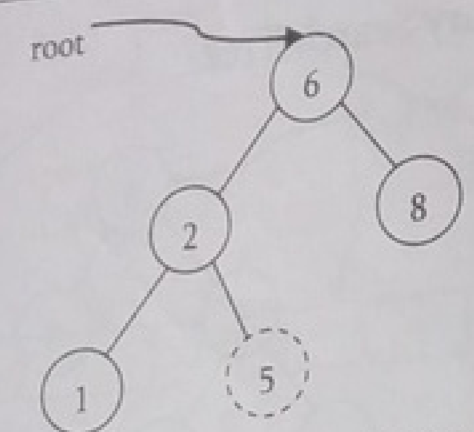
Note: In the above code, after inserting an element in subtrees the tree is returned to its parent. As a result, the complete tree will get updated.

Time Complexity:$O(n)$. Space Complexity:$O(n)$, for recursive stack. For iterative version, space complexity is $O(1)$.

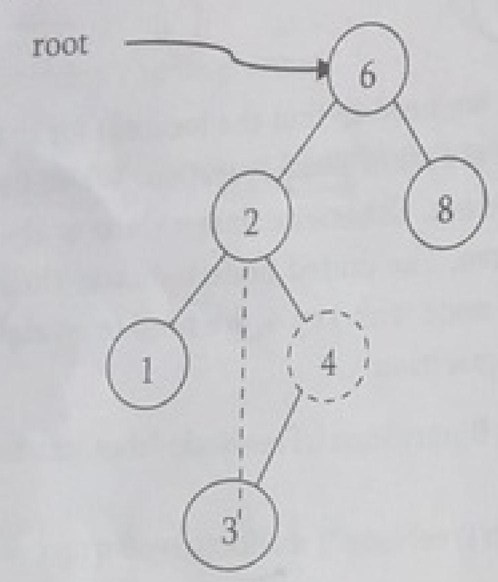
Deleting an Element from Binary Search Tree

The delete operation is little complicated than other operations. This is because the element to be deleted may not be the leaf node. In this operation also, first we need to find the location of the element which we want to delete. Once we have found the node to be deleted, consider the following cases:

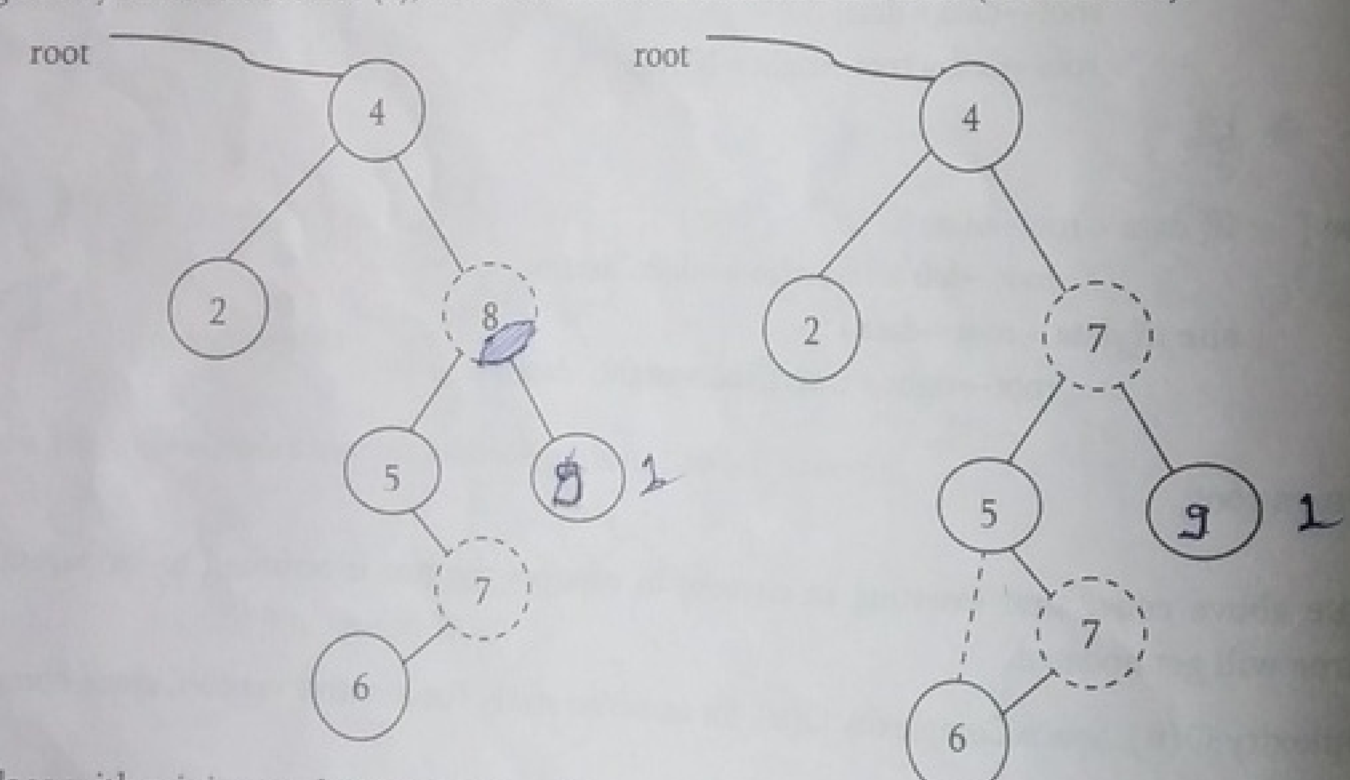
- If the element to be deleted is a leaf node: return NULL to its parent. That means make the corresponding child pointer NULL. In the below tree to delete 5, set NULL to its parent node 2.



- If the element to be deleted has one child: In this case we just need to send the current nodes child to its parent. In the below tree, to delete 4, 4 left subtree is set to its parent node 2.



- If the element to be deleted 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). The largest node in the left subtree cannot have a right child, the second *delete* is an easy one. As an example, let us consider the following tree. In the below tree, to delete 8, it is the right child of root. The key value is 8. It is replaced with the largest key in its left subtree (7), and then that node is deleted as before (second case).



Note: We can replace with minimum element in right subtree also.

```
struct BinarySearchTreeNode *Delete(struct BinarySearchTreeNode *root, int data) {
    struct BinarySearchTreeNode *temp;
    if( root == NULL )
        printf("Element not there in tree");
    else if(data < root→data)
        root→left = Delete(root→left, data);
```



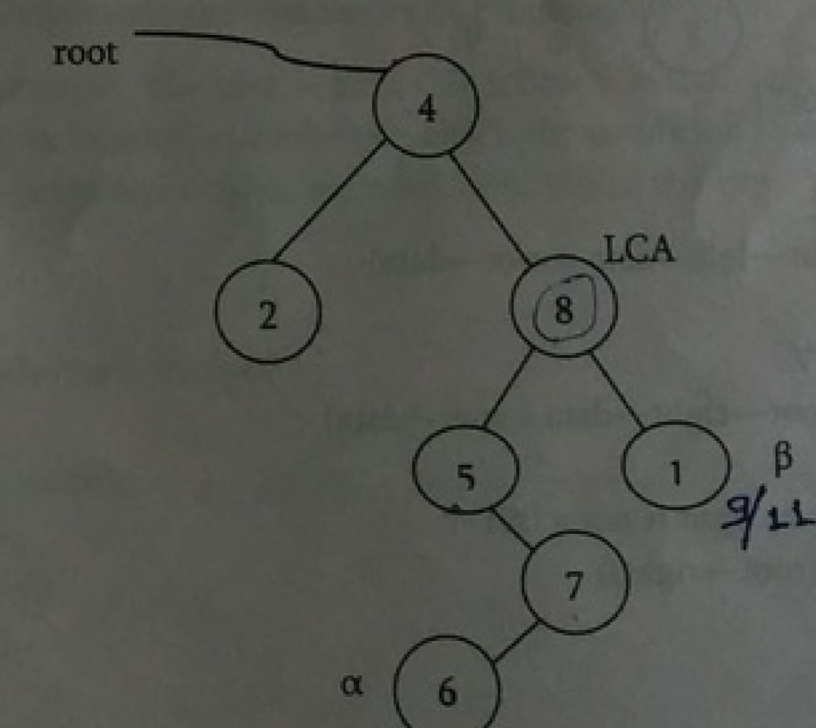
```
    else if(data > root→data )
        root→right = Delete(root→right, data);
    else {  //Found element
        if( root→left && root→right ) {
            /* Replace with largest in left subtree */
            temp = FindMax( root→left );
            root→data = temp→data;
            root→left = Delete(root→left, root→data);
        }
        else {  /* One child */
            temp = root;
            if( root→left == NULL )
                root = root→right;
            if( root→right == NULL )
                root = root→left;
            free( temp );
        }
    }
    return root;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$ for recursive stack. For iterative version, space complexity is $O(1)$.

Problems on Binary Search Trees

Problem-46 Given pointers to two nodes in a binary search tree, find lowest common ancestor (*LCA*). Assume that both values already exist in the tree.

Solution:



4 2 8 5 7 6 9

The main idea of the solution is: while traversing BST from root to bottom, the first node we encounter with value between α and β, i.e., $\alpha < node \rightarrow data < \beta$ is the Least Common Ancestor(LCA) of α and β (where $\alpha < \beta$). So just traverse the BST in pre-order, if we find a node with value in between α and β then that node is the LCA. If its value is greater than both α and β then LCA lies on left side of the node and if its value is smaller than both α and β then LCA lies on right side.

```
struct BinarySearchTreeNode *FindLCA(struct BinarySearchTreeNode *root,
                        struct BinarySearchTreeNode *α, struct BinarySearchTreeNode * 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 complexity: $O(n)$. Space complexity: $O(n)$, for skew trees.

Problem-47 Give an algorithm for finding the shortest path between two nodes in a BST.

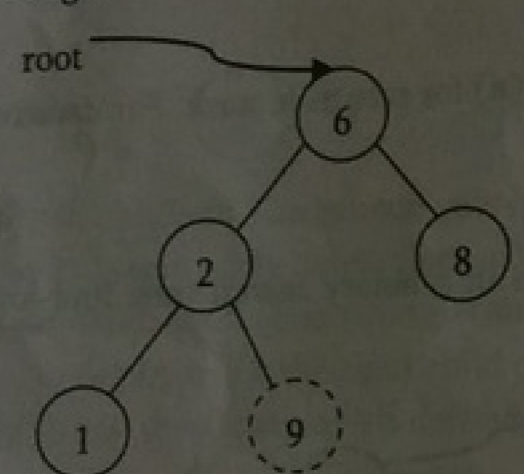
Solution: It's nothing but finding the LCA of two nodes in BST.

Problem-48 Give an algorithm for counting the number of BSTs possible with n nodes.

Solution: This is a DP problem and refer *Dynamic Programming* chapter for algorithm.

Problem-49 Give an algorithm to check whether the given binary tree is a BST or not.

Solution: Consider the following simple program. For each node, check if left node of it is smaller than the node and right node of it is greater than the node. This approach is wrong as this will return true for below binary tree. Checking only at current node is not enough.




```
int IsBST(struct BinaryTreeNode* root) {
    if(root == NULL) return 1;
    /* false if left is > than root */
    if(root→left != NULL && root→left→data > root→data)
        return 0;
    /* false if right is < than root */
    if(root→right != NULL && root→right→data < root→data)
        return 0;
    /* false if, recursively, the left or right is not a BST */
    if(!IsBST(root→left) || !IsBST(root→right))
        return 0;
    /* passing all that, it's a BST */
    return 1;
}
```

Problem-50 Can we think of getting the correct algorithm?

Solution: For each node, check if max value in left subtree is smaller than the current node data and min value in right subtree greater than the node data. It is assumed that we have helper functions *FindMin*() and *FindMax*() that return the min or max integer value from a non-empty tree.

```
/* Returns true if a binary tree is a binary search tree */
int IsBST(struct BinaryTreeNode* root) {
    if(root == NULL) return 1;
```



```
    /* false if the max of the left is > than root */
    if(root→left != NULL && FindMax(root→left) > root→data)
        return 0;
    /* false if the min of the right is <= than root */
    if(root→right != NULL && FindMin(root→right) < root→data)
        return 0;
    /* false if, recursively, the left or right is not a BST */
    if(!IsBST(root→left) || !IsBST(root→right)) return 0;
    /* passing all that, it's a BST */
    return 1;
}
```

Time complexity: $O(n^2)$. Space Complexity: $O(n)$.

Problem-51 Can we improve the complexity of Problem-50?

Solution: Yes. A better solution looks at each node only once. The trick is to write a utility helper function IsBSTUtil(struct BinaryTreeNode* root, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX — they narrow from there.

```
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 &&
            IsBSTUtil(root→left, min, root→data) && IsBSTUtil(root→right, root→data, max));
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

Problem-52 Can we further improve the complexity of Problem-50?

Solution: Yes, using inorder traversal. The idea behind this solution is that, inorder traversal of BST produces sorted lists. While traversing the BST in inorder, at each node check the condition that its key value should be greater than the key value of its previous visited node. Also, we need to initialize the prev with possible minimum integer value (say, INT_MIN).

```
int prev = INT_MIN;
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);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

Problem-53 Give an algorithm for converting BST to circular DLL with space complexity $O(1)$.

Solution: Convert left and right subtrees to DLLs and maintain end of those lists. Then, adjust the pointers.

```
struct BinarySearchTreeNode *BST2DLL(struct BinarySearchTreeNode *root, struct BinarySearchTreeNode **Ltail) {
    struct BinarySearchTreeNode *left, *ltail, *right, *rtail;
    if(!root) {
        * ltail = NULL;
```

```
            return NULL;
      }
      left = BST2DLL(root→left, &ltail);
      right = BST2DLL(root→right, &rtail);
      root→left = ltail;
      root→right = right;
      if(!right)
            *ltail = root;
      else {   right→left = root;
            *ltail = rtail;
      }
      if(!left)
            return root;
      else {   ltail→right = root;
            return left;
      }
}
```

Time Complexity: $O(n)$.

Problem-54 For Problem-53, is there any other way of solving?

Solution: Yes. There is an alternative solution based on divide and conquer method which is quite neat.

```
struct BinarySearchTreeNode *Append(struct BinarySearchTreeNode *a, struct BinarySearchTreeNode *b) {
   struct BinarySearchTreeNode *aLast, *bLast;
      if (a==NULL)
            return b;
      if (b==NULL)
            return a;
      aLast = a->left;
      bLast = b->left;
      aLast->right = b;
      b->left = aLast;
      bLast->right = a;
      a->left = bLast;
      return a;
}
struct BinarySearchTreeNode* TreeToList(struct BinarySearchTreeNode *root) {
      struct BinarySearchTreeNode *aList, *bList;
      if (root==NULL)
            return NULL;
      aList = TreeToList(root->left);
      bList = TreeToList(root->right);
      root->left = root;
      root->right = root;
      aList = Append(aList, root);
      aList = Append(aList, bList);
      return(aList);
}
```

Time Complexity: $O(n)$.

Problem-55 Given a sorted doubly linked list, give an algorithm for converting it to balanced binary search tree.

Solution: Find the middle node and adjust the pointers.

```
struct DLLNode * DLLtoBalancedBST(struct DLLNode *head) {
      struct DLLNode *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 = DLLtoBalancedBST(head);
      temp→next = DLLtoBalancedBST(q);
      return temp;
}
```

Time Complexity: $2T(n/2) + O(n)$ [for finding the middle node] = $O(nlogn)$.

Note: For *FindMiddleNode* function refer *Linked Lists* chapter.

Problem-56 Given a sorted array, give an algorithm for converting the array to BST.

Solution: If we have to choose an array element to be the root of a balanced BST, which element we should pick? The root of a balanced BST should be the middle element from the sorted array. We would pick the middle element from the sorted array in each iteration. We then create a node in the tree initialized with this element. After the element is chosen, what is left? Could you identify the sub-problems within the problem?

There are two arrays left — The one on its left and the one on its right. These two arrays are the sub-problems of the original problem, since both of them are sorted. Furthermore, they are subtrees of the current node's left and right child.

The code below creates a balanced BST from the sorted array in $O(n)$ time (n is the number of elements in the array). Compare how similar the code is to a binary search algorithm. Both are using the divide and conquer methodology.

```
struct BinaryTreeNode *BuildBST(int A[], int left, int right) {
      struct BinaryTreeNode *newNode;
      int mid;
      if(left > right)
            return NULL;
      newNode = (struct BinaryTreeNode *)malloc(sizeof(struct BinaryTreeNode));
      if(!newNode) {
            printf("Memory Error");
            return;
      }
      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 Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

Problem-57 Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Solution: A naive way is to apply the Problem-55 solution directly. In each recursive call, we would have to traverse half of the list's length to find the middle element. The run time complexity is clearly $O(nlogn)$, where n is the total number of elements in the list. This is because each level of recursive call requires a total of $n/2$ traversal steps in the list, and there are a total of $logn$ number of levels (ie, the height of the balanced tree).

Problem-58 For Problem-57, can we improve the complexity?

Solution: Hint: How about inserting nodes following the list order? If we can achieve this, we no longer need to find the middle element, as we are able to traverse the list while inserting nodes to the tree.

Best Solution: As usual, the best solution requires us to think from another perspective. In other words, we no longer create nodes in the tree using the top-down approach. Create nodes bottom-up, and assign them to its parents. The bottom-up approach enables us to access the list in its order while creating nodes [42].

Isn't the bottom-up approach neat? Each time we are stucked with the top-down approach, give bottom-up a try. Although bottom-up approach is not the most natural way we think, it is extremely helpful in some cases. However, we should prefer top-down instead of bottom-up in general, since the latter is more difficult to verify in correctness.

Below is the code for converting a singly linked list to a balanced BST. Please note that the algorithm requires the list length to be passed in as the function parameters. The list length could be found in $O(n)$ time by traversing the entire list once. The recursive calls traverse the list and create tree nodes by the list order, which also takes $O(n)$ time. Therefore, the overall run time complexity is still $O(n)$.

```
struct BinaryTreeNode* SortedListToBST(struct ListNode *& list, int start, int end) {
        if(start > end)
                return NULL;
        // same as (start+end)/2, avoids overflow
        int mid = start + (end - start) / 2;
        struct BinaryTreeNode *leftChild = SortedListToBST(list, start, mid-1);
        struct BinaryTreeNode * parent;
        parent = (struct BinaryTreeNode *)malloc(sizeof(struct BinaryTreeNode));
        if(!parent) {
                printf("Memory Error");
                return;
        }
        parent→data=list→data;
        parent→left = leftChild;
        list = list→next;
        parent→right = SortedListToBST(list, mid+1, end);
        return parent;
}
struct BinaryTreeNode * SortedListToBST(struct ListNode *head, int n) {
        return SortedListToBST(head, 0, n-1);
}
```

Problem-59 Give an algorithm for finding the k^{th} smallest element in BST.

Solution: The idea behind this solution is that, inorder traversal of BST produces sorted lists. While traversing the BST in inorder, keep track of the number of elements visited.


```
struct BinarySearchTreeNode *kthSmallestInBST(struct BinarySearchTreeNode *root, int k, int *count){
        if(!root) return NULL;
        struct BinarySearchTreeNode *left = kthSmallestInBST(root→left, k, count);
        if( left ) return left;
        if(++count == k)
                return root;
        return kthSmallestInBST(root→right, k, count);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Problem-60 **Floor and ceiling:** If a given key is less than the key at the root of a BST then floor of key (the largest key in the BST less than or equal to key) must be in the left subtree. If key is greater than the key at the root then floor of key could be in the right subtree, but only if there is a key smaller than or equal to key in the right subtree; if not (or if key is equal to the key at the root) then the key at the root is the floor of key. Finding the ceiling is similar with interchanging right and left.For example, if the sorted with input array is {1, 2, 8, 10, 10, 12, 19}, then
For $x = 0$: floor doesn't exist in array, ceil $= 1$, For $x = 1$: floor $= 1$, ceil $= 1$
For $x = 5$: floor $= 2$, ceil $= 8$, For $x = 20$: floor $= 19$, ceil doesn't exist in array

Solution: The idea behind this solution is that, inorder traversal of BST produces sorted lists. While traversing the BST in inorder, keep track of the values being visited. If the roots data is greater than the given value then return the previous value which we have maintained during traversal. If the roots data is equal to the given data then return root data.

```
struct BinaryTreeNode *FloorInBST(struct BinaryTreeNode *root, int data){
        struct BinaryTreeNode *prev=NULL;
        return FloorInBSTUtil(root, prev, data);
}
struct BinaryTreeNode  *FloorInBSTUtil(struct BinaryTreeNode *root, struct BinaryTreeNode *prev, int data){
        if(!root)
                return NULL;
        if(!FloorInBSTUtil(root→left, prev, data))
                return 0;
        if(root→data == data)
                return root;
        if(root→data > data)
                return prev;
        prev = root;
        return FloorInBSTUtil(root→right, prev, data);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

For ceiling, we just need to call the right subtree first and then followed by left subtree.

```
struct BinaryTreeNode *CeilingInBST(struct BinaryTreeNode *root, int data){
        struct BinaryTreeNode *prev=NULL;
        return CeilingInBSTUtil(root, prev, data);
}
struct BinaryTreeNode  *CeilingInBSTUtil(struct BinaryTreeNode *root, struct BinaryTreeNode *prev, int data){
        if(!root)
                return NULL;
        if(!CeilingInBSTUtil(root→right, prev, data))
                return 0;
        if(root→data == data) return root;
```

```
        if(root→data < data) return prev;
        prev = root;
        return CeilingInBSTUtil(root→left, prev, data);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

Problem-61 Give an algorithm for finding the union and intersection of BSTs. Assume parent pointers are available (say threaded binary trees). Also, assume the lengths of two BSTs are m and n respectively.

Solution: If parent pointers are available then the problem is same as merging of two sorted lists. This is because if we call inorder successor each time we get the next highest element. It's just a matter of which InorderSuccessor to call.

Time Complexity: $O(m + n)$. Space complexity: $O(1)$.

Problem-62 For Problem-61, what if parent pointers are not available?

Solution: If parent pointers are not available then, one possibility is converting the BSTs to linked lists and then merging.

1 Convert both the BSTs into sorted doubly linked lists in $O(n + m)$ time. This produces 2 sorted lists.
2 Merge the two double linked lists into one and also maintain the count of total elements in $O(n + m)$ time.
3 Convert the sorted doubly linked list into height balanced tree in $O(n + m)$ time.

Problem-63 For Problem-61, is there any alternative way of solving the problem?

Solution: Yes, using inorder traversal.

- Perform inorder traversal on one of the BST.
- While performing the traversal store them in table (hash table).
- After completion of the traversal of first *BST*, start traversal of the second *BST* and compare them with hash table contents.

Time Complexity: $O(m + n)$. Space Complexity: $O(Max(m, n))$.

Problem-64 Given a *BST* and two numbers $K1$ and $K2$, give an algorithm for printing all the elements of *BST* in the range $K1$ and $K2$.

Solution:

```
void RangePrinter(struct BinarySearchTreeNode *root, int K1, int K2) {
        if(root == NULL) 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 Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

Problem-65 For Problem-64, is there any alternative way of solving the problem?

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

```
void RangeSeachLevelOrder(struct BinarySearchTreeNode *root, int K1, int K2){
        struct BinarySearchTreeNode *temp;
        struct Queue *Q = CreateQueue();
        if(!root) return NULL;
        Q = EnQueue(Q, root);
        while(!IsEmptyQueue(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);
        }
        DeleteQueue(Q);
        return NULL;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for queue.

Problem-66 For Problem-64, can we still think of alternative way for solving the problem?

Solution: First locate $K1$ with normal binary search and after that use InOrder successor until we encounter $K2$. For algorithm, refer problems section of threaded binary trees.

Problem-67 Given root of a Binary Search tree, trim the tree, so that all elements in the new tree returned are between the inputs A and B.

Solution: It's just another way of asking the Problem-64.

Problem-68 Given two BSTs, check whether the elements of them are same or not. For example: two BSTs with data 10 5 20 15 30 and 10 20 15 30 5 should return true and the dataset with 10 5 20 15 30 and 10 15 30 20 5 should return false. **Note:** BSTs data can be in any order.

Solution: One simple way is performing an inorder traversal on first tree and storing its data in hash table. As a second step perform inorder traversal on second tree and check whether that data is already there in hash table or not (if it exists in hash table then mark it with -1 or some unique value). During the traversal of second tree if we find any mismatch return false. After traversal of second tree check whether it has all -1s in the hash table or not (this ensures extra data available in second tree).

Time Complexity: $O(max(m, n))$, where m and n are the number of elements in first and second BST. Space Complexity: $O(max(m, n))$. This depends on the size of the first tree.

Problem-69 For Problem-68, can we reduce the time complexity?

Solution: Instead of performing the traversals one after the other, we can perform $in-order$ traversal of both the trees in parallel. Since the $in-order$ traversal gives the sorted list, we can check whether both the trees are generating the same sequence or not.

Time Complexity: $O(max(m, n))$. Space Complexity: $O(1)$. This depends on the size of the first tree.

Problem-70 For the key values $1 \ldots n$, how many structurally unique BSTs are possible that store those keys.

Solution: Strategy: consider that each value could be the root. Recursively find the size of the left and right subtrees.

```
int CountTrees(int n) {
        if (n <= 1) return 1;
        else {  // there will be one value at the root, with whatever remains on the left and right
                // each forming their own subtrees.Iterate through all the values that could be the root...
                int sum = 0;
                int left, right, root;
                for (root=1; root<=n; root++) {
                        left = CountTrees(root - 1);
                        right = CountTrees(numKeys - root);
                        // number of possible trees with this root == left*right
                        sum += left*right;
```

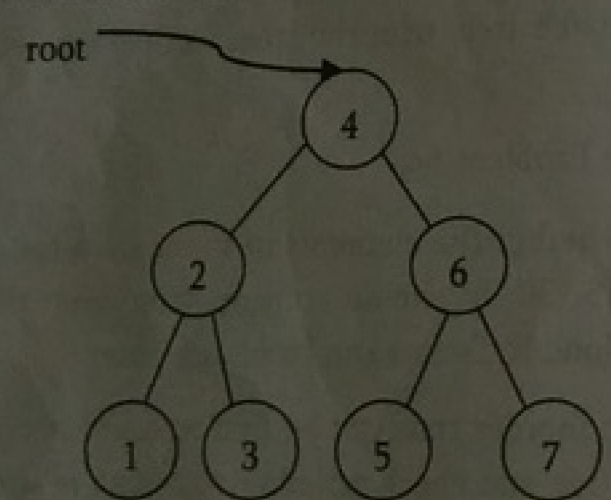
```
        }
        return(sum);
    }
}
```

6.12 Balanced Binary Search Trees

In earlier sections we have seen different trees whose worst case complexity is $O(n)$, where n is the number of nodes in the tree. This happens when the trees are skew trees. In this section we will try to reduce this worst case complexity to $O(logn)$ by imposing restrictions on the heights. In general, the height balanced trees are represented with $HB(k)$, where k is the difference between left subtree height and right subtree height. Sometimes k is called balance factor.

Full Balanced Binary Search Trees

In $HB(k)$, if $k = 0$ (if balance factor is zero), then we call such binary search trees as *full* balanced binary search trees. That means, in $HB(0)$ binary search tree, the difference between left subtree height and right subtree height should be at most zero. This ensures that the tree is a full binary tree. For example,



Note: For constructing $HB(0)$ tree refer problems section.

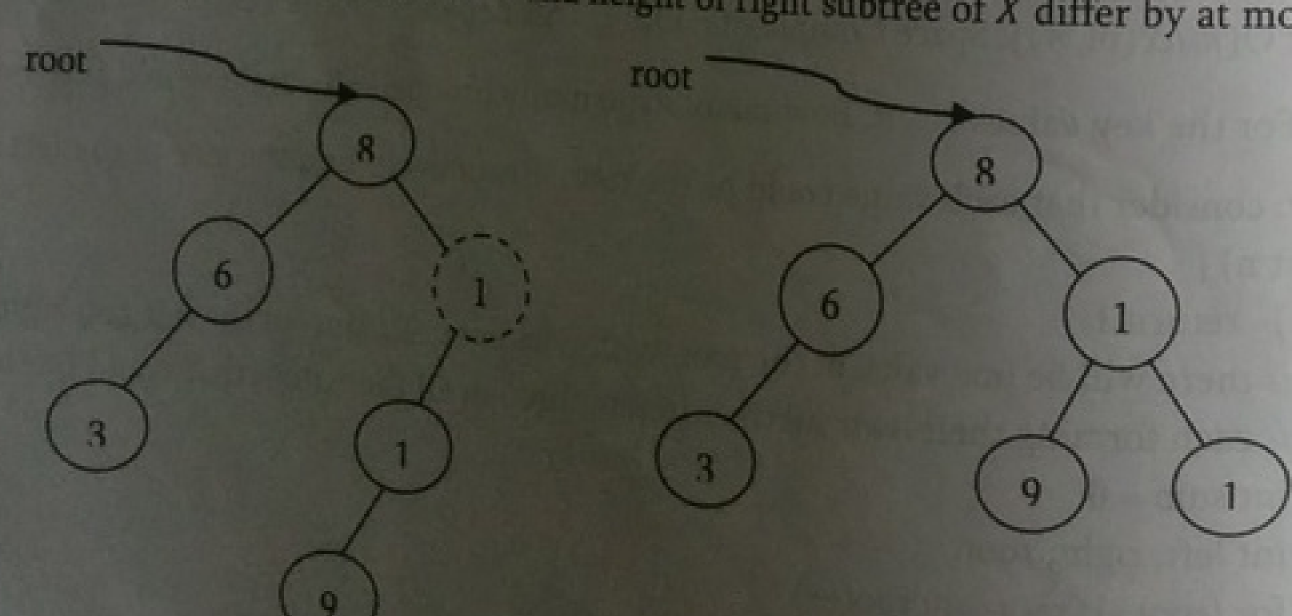
6.13 AVL (Adelson-Velskii and Landis) Trees

In $HB(k)$, if $k = 1$ (if balance factor is one), such binary search tree is called an AVL tree. That means an AVL tree is a binary search tree with a *balance* condition: the difference between left subtree height and right subtree height is at most 1.

Properties of AVL Trees

A binary tree is said to be an AVL tree, if:

- It is a binary search tree, and
- For any node X, the height of left subtree of X and height of right subtree of X differ by at most 1.



As an example among the above binary search trees, the left one is not an AVL tree, whereas the right binary search tree is an AVL tree.

Minimum/Maximum Number of Nodes in AVL Tree

For simplicity let us assume that the height of an AVL tree is h and $N(h)$ indicates the number of nodes in AVL tree with height h. To get minimum number of nodes with height h, we should fill the tree with as minimum nodes as possible. That means if we fill the left subtree with height $h-1$ then we should fill the right subtree with height $h-2$. As a result, the minimum number of nodes with height h is:

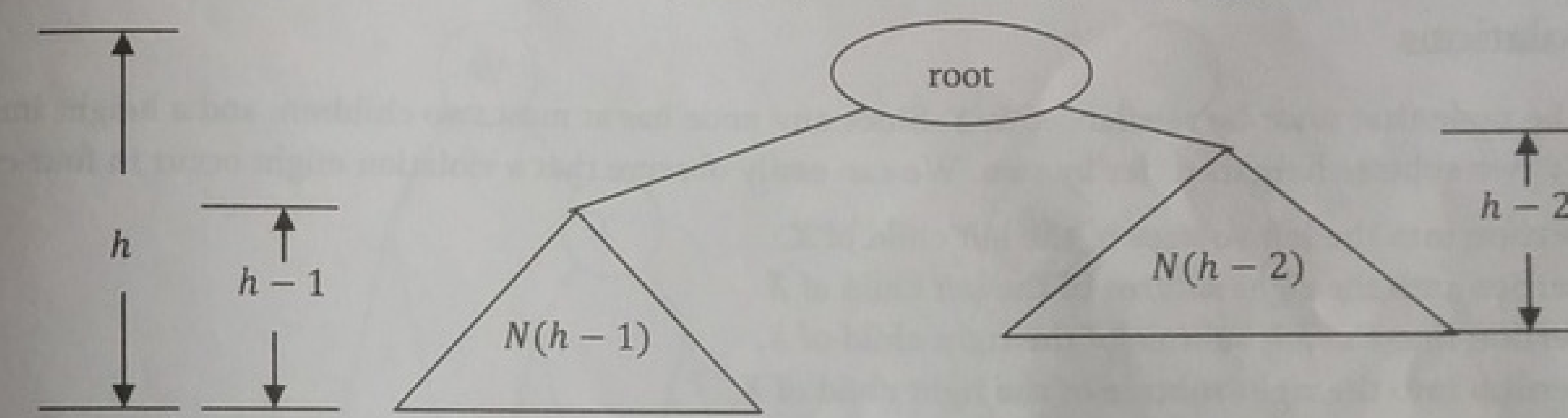
$$N(h) = N(h-1) + N(h-2) + 1$$

In the above equation:

- $N(h-1)$ indicates the minimum number of nodes with height $h-1$.
- $N(h-2)$ indicates the minimum number of nodes with height $h-2$.
- In the above expression, "1" indicates the current node.

We can give $N(h-1)$ either for left subtree or right subtree. Solving the above recurrence gives:

$$N(h) = O(1.618^h) \Rightarrow h = 1.44logn \approx O(logn)$$



Where n is the number of nodes in AVL tree. Also, the above derivation says that the maximum height in AVL trees is $O(logn)$. Similarly, to get maximum number of nodes, we need to fill both left and right subtrees with height $h-1$. As a result, we get

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

The above expression defines the case of full binary tree. Solving the recurrence we get:

$$N(h) = O(2^h) \Rightarrow h = logn \approx O(logn)$$

∴ In both the cases, AVL tree property is ensuring that the height of an AVL tree with n nodes is $O(logn)$.

AVL Tree Declaration

Since AVL tree is a BST, the declaration of AVL is similar to that of BST. But just to simplify the operations, we include the height also as part of declaration.

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

Finding Height of an AVL tree

```
int Height(struct AVLTreeNode *root ){
        if( !root) return -1;
        else return root→height;
}
```

Time Complexity: O(1).

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 rotations. Since an insertion/deletion involves