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 × 4
3	4 × 4 × 4	1+ 4 × 4 + 4 × 4 × 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:

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

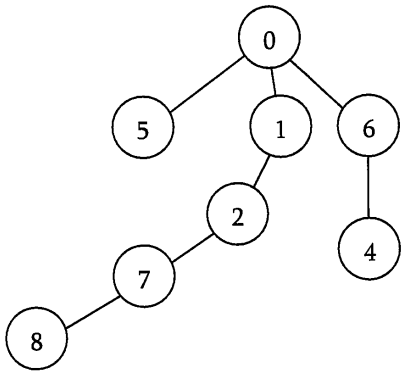
Problem-39 Given a parent array P, where $P[i]$ indicates the parent of i^{th} node in the tree (assume parent of root node is indicated with -1). Give an algorithm for finding the height or depth of the tree.

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;
};
```

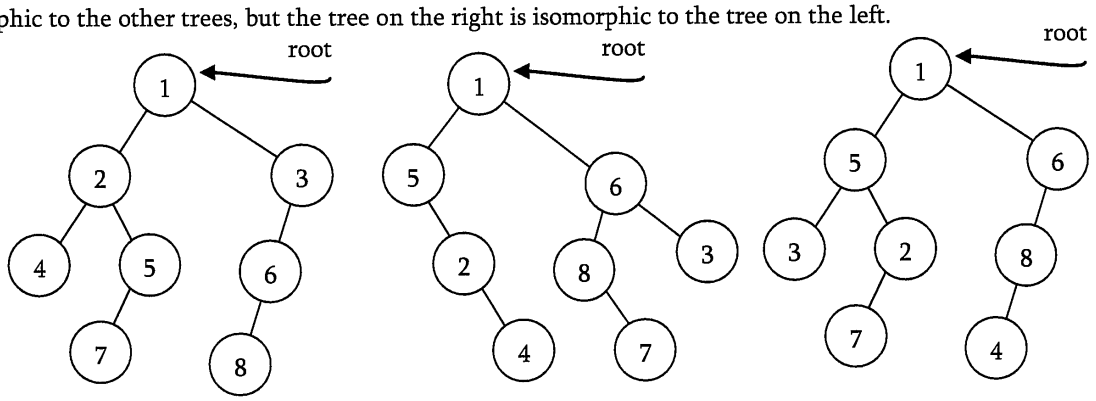
For a given node in the tree, we just need to point to its first child and keep traversing all its nextsiblings.

```
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.

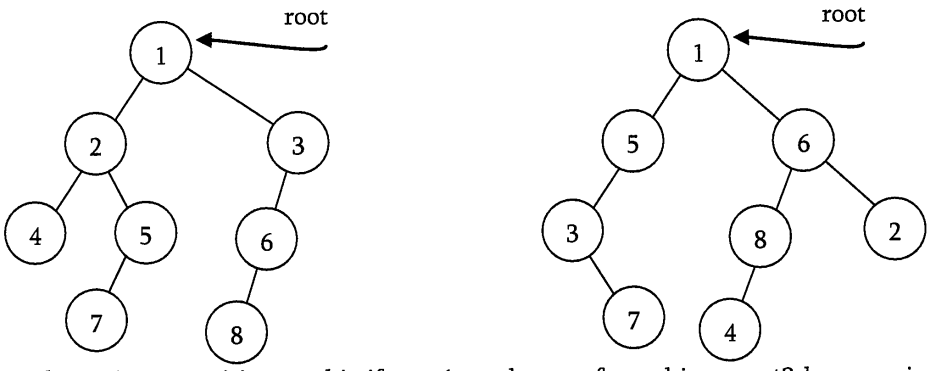



```
int IsIsIsomorphic(struct TreeNode *root1, struct TreeNode *root2){
        if(!root1 && !root2) return 1;
        if((!root1 && root2) || (root1 && !root2))
                return 0;
        return (IsIsomorphic(root1→left, root2→left) && IsIsomorphic(root1→right, root2→right));
}
```

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:



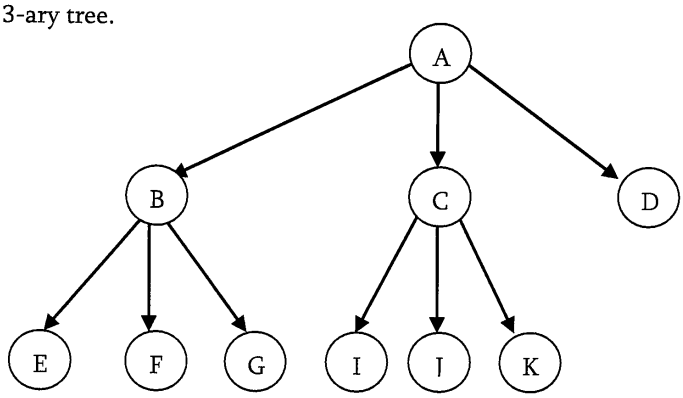
Two trees *root*1 and *root*2 are quasi-isomorphic if *root*1 can be transformed into *root*2 by swapping left and right children of some of the nodes of *root*1. The data in the nodes are not important in determining quasi-isomorphism, only the shape is important. The trees below are quasi-isomorphic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

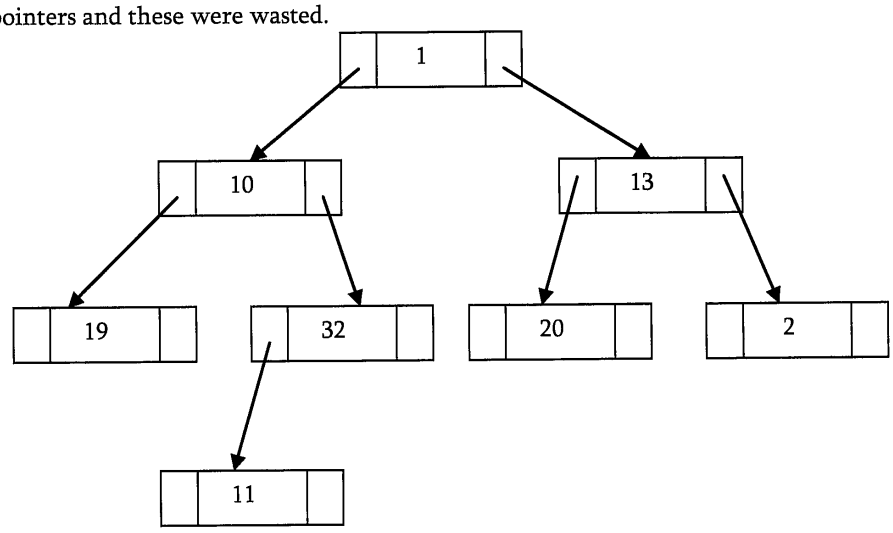
Time Complexity: $O(n)$, where n is the size of the pre-order array. This is because we are moving sequentially and not visiting the already constructed nodes.

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

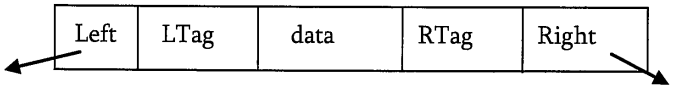
Cased on above discussion we get three representations for threaded binary trees.

- *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:



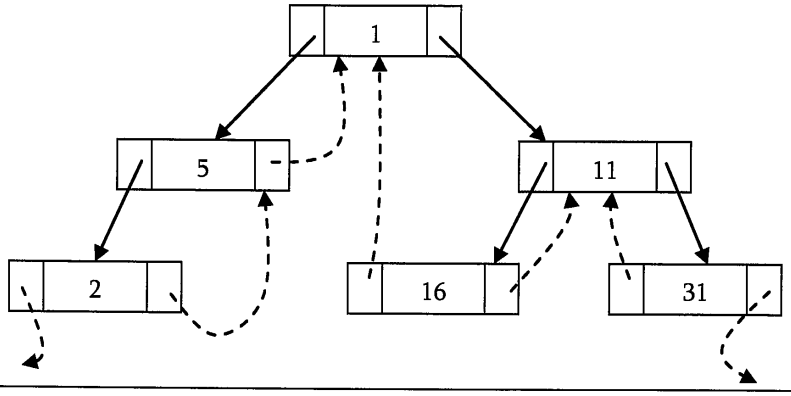

```
struct ThreadedBinaryTreeNode{
        struct ThreadedBinaryTreeNode *left;
        int LTag;
        int data;
        int RTag;
        struct ThreadedBinaryTreeNode *right;
};
```

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

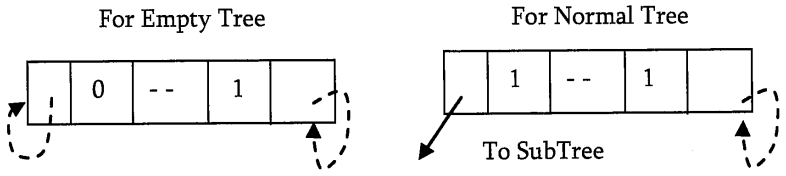
Note: Similarly, we can define for preorder/postorder differences as well.

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.

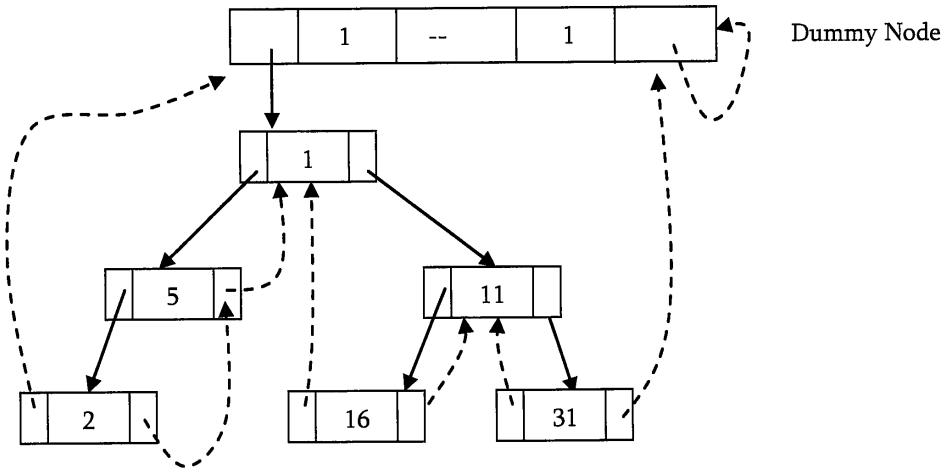


What should leftmost and rightmost pointers point to?

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;
        }
}
```

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);
        }
}
```

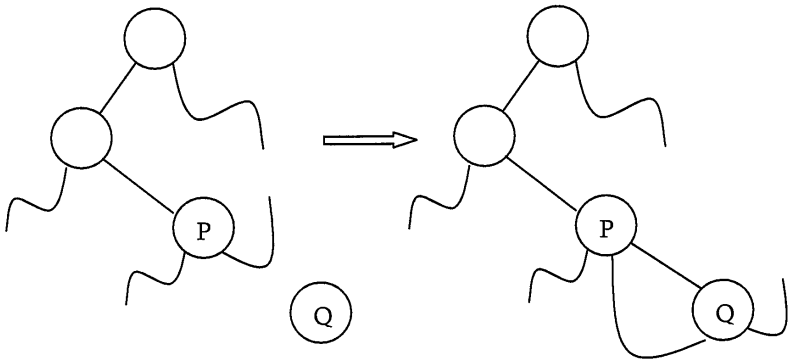
Time Complexity: O(n). Space Complexity: O(1).

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.

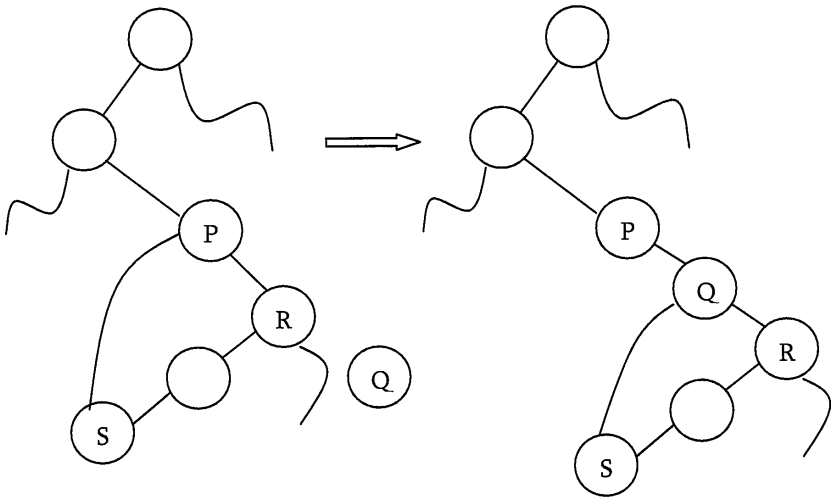
Insertion of Nodes in InOrder Threaded Binary Trees

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;
}
```

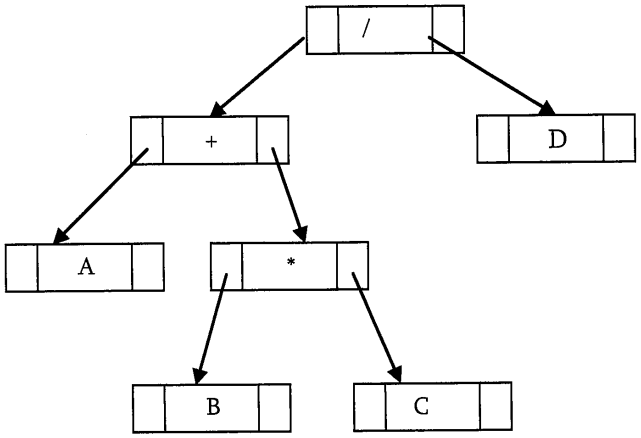
Problem-46 For a given binary tree (not threaded) how do we find the inorder successor?

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.

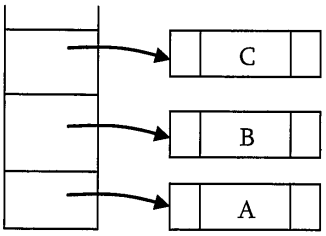


Algorithm for Building Expression Tree from Postfix Expression

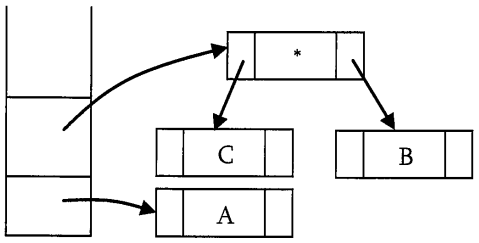
```
struct BinaryTreeNode *BuildExprTree(char postfixExpr[], int size){
        struct Stack *S = Stack(size);
        for (int i = 0; i< size; i++) {
                if(postfixExpr[i] is an operand) {
                        struct BinaryTreeNode newNode = (struct BinaryTreeNode*)
                                malloc( sizeof (struct BinaryTreeNode));
                        if(!newNode) {
                                printf("Memory Error");
                                return;
                        }
                        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;
                        }
                        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.

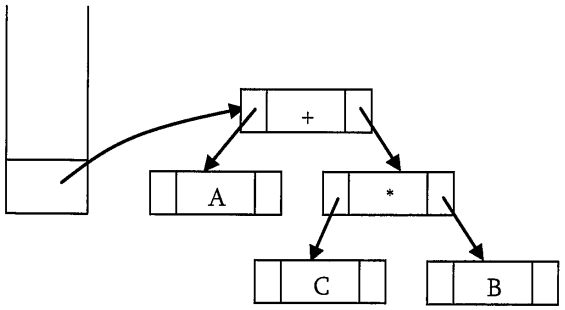
As an example, assume the input is A B C * + D /. The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below.



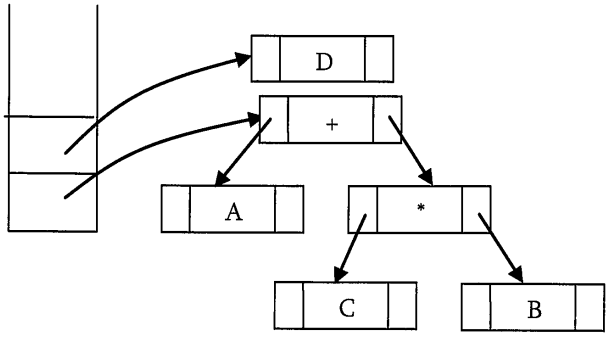
Next, an operator '*' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



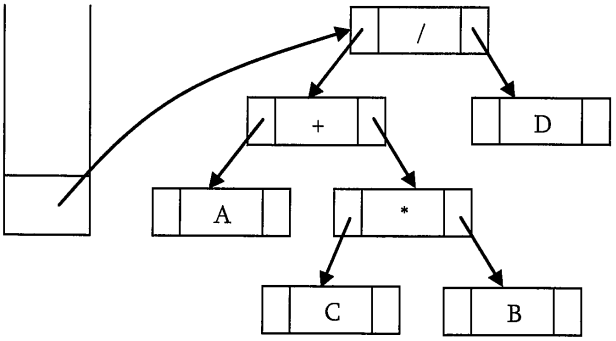
Next, an operator '+' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



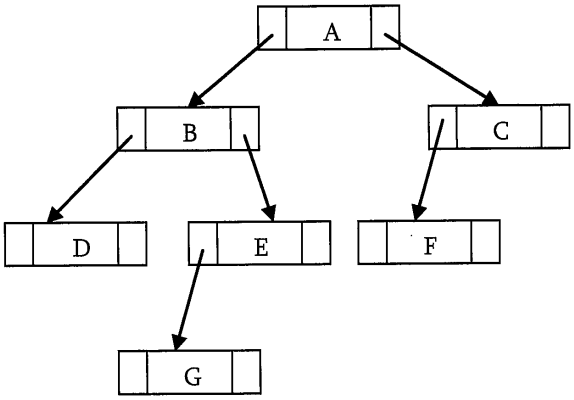
Next, an operand 'D' is read, a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



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



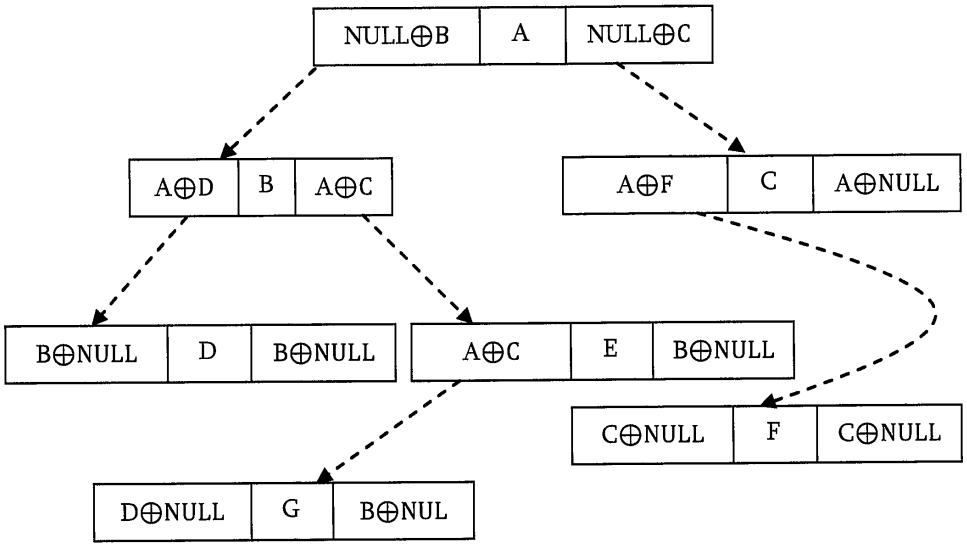
6.10 XOR Trees



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?

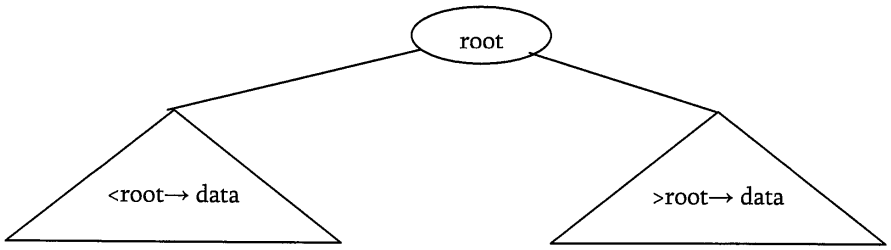
In previous sections we have discussed different tree representations and in all of them we did not impose any restriction on the nodes data. As a result, to search for an element we need to check both in left subtree and also right subtree. Due to this, the worst case complexity of search operation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

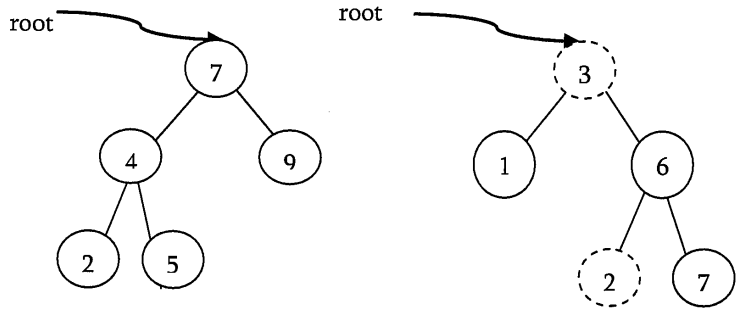
Binary Search Tree Property

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.

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:

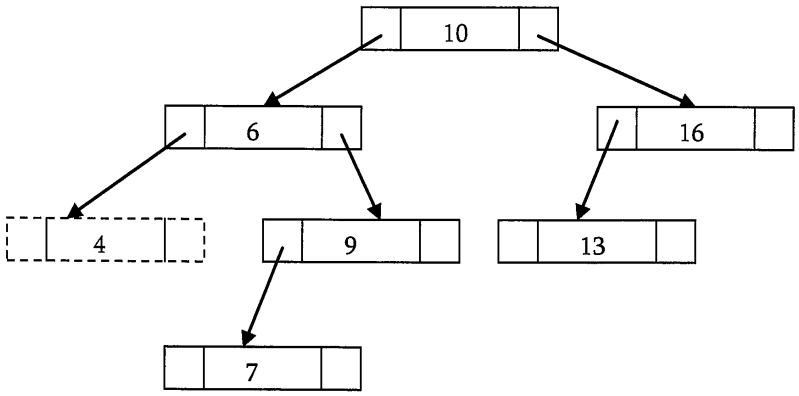
```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
        if( root == NULL ) return NULL;
        while (root) {
                if(data == root→data)
                        return root;
                else if(data > root→data)
                        root = root→right;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

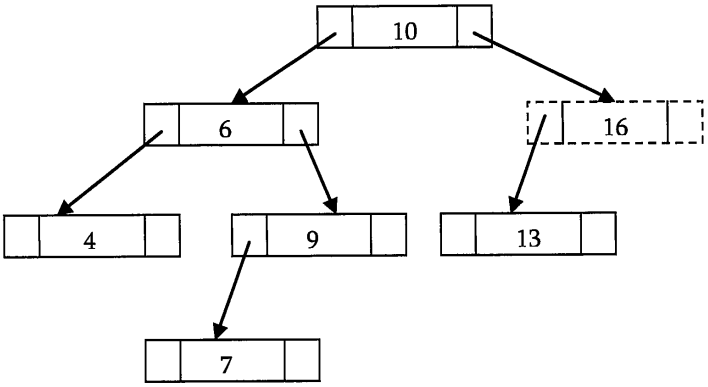
Time Complexity: O(n). Space Complexity: O(1).

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:

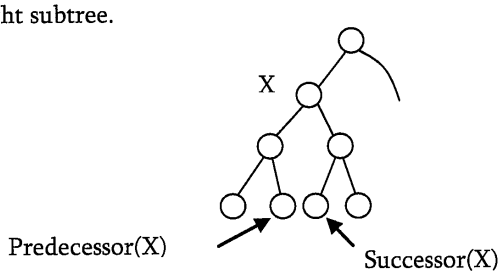
```
struct BinarySearchTreeNode *FindMax(struct BinarySearchTreeNode * root ) {
        if( root == NULL )
                return NULL;
        while( root→right != NULL )
                root = root→right;
        return root;
}
```

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

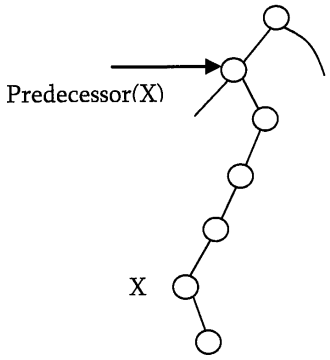
Where is Inorder Predecessor and Successor?

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

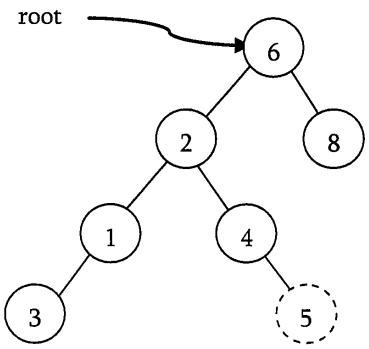
If X has two children then its inorder predecessor is the maximum value in its left subtree and its inorder successor the minimum value in its right subtree.



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;
}
```

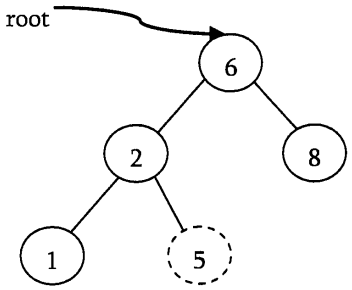
Note: In the above code, after inserting an element in subtrees the tree is returned to its parent. As a result, the complete tree will get updated.

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

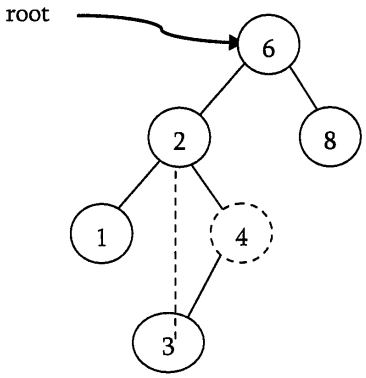
Deleting an Element from Binary Search Tree

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:

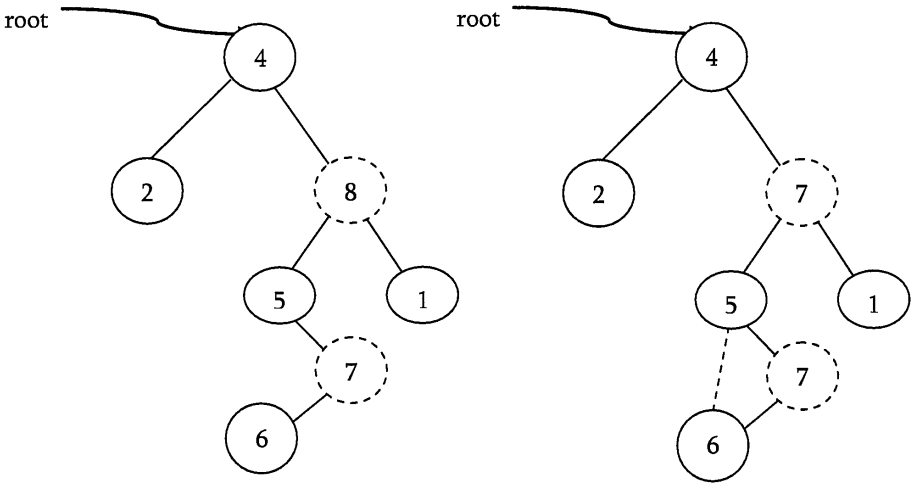
- If the element to be deleted is a leaf node: return NULL to its parent. That means make the corresponding child pointer NULL. In the below tree to delete 5, set NULL to its parent node 2.



- 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→element )
                root→left = Delete(root→left, data);
```

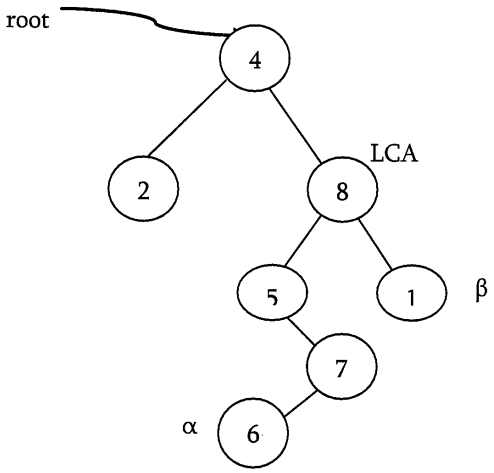
```
        else if(data > root→element )
                root→right = Delete(root→right, data);
        else {  //Found element
                if( root→left && root→right ) {
                        /* Replace with largest in left subtree */
                        temp = FindMax( root→left );
   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



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.

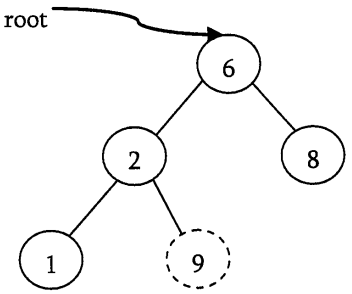
Solution: It's nothing but finding the LCA of two nodes in BST.

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's 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's length to be passed in as the function's parameters. The list's length could be found in O(n) time by traversing the entire list's once. The recursive calls traverse the list and create tree's nodes by the list's 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.