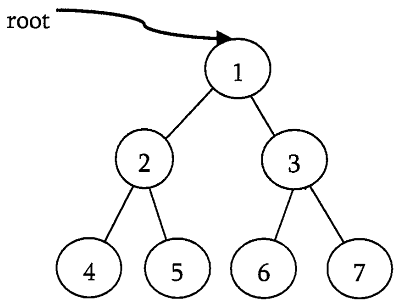
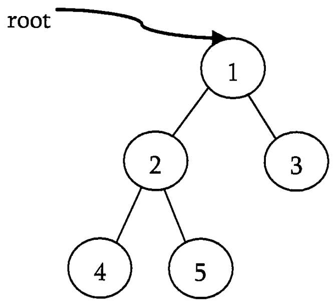
Full Binary Tree: A binary tree is called *full binary tree* if each node has exactly two children and all leaf nodes are at same level.

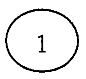
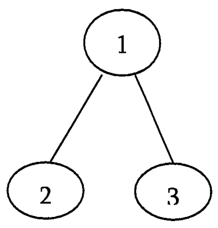
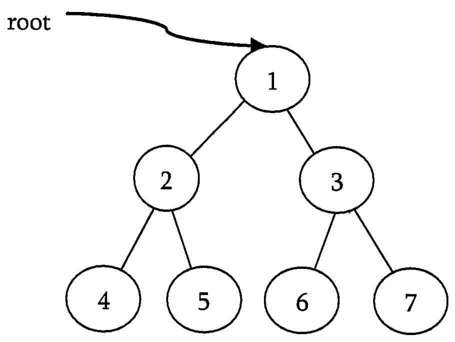


Complete Binary Tree: Before defining the *complete binary tree*, let us assume that the height of the binary tree is h. In complete binary trees, if we give numbering for the nodes by starting at root (let us say the root node has 1) then we get a complete sequence from 1 to number of nodes in the tree. While traversing we should give numbering for NULL pointers also. A binary tree is called complete binary tree if all leaf nodes are at height h or $h-1$ and also without any missing number in the sequence.



6.5 Properties of Binary Trees

For the following properties, let us assume that the height of the tree is h. Also, assume that root node is at height zero.

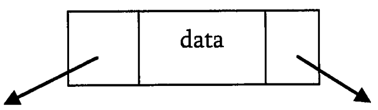
	Height	**Number of nodes at level h**
	$h = 0$	$2^0 = 1$
	$h = 1$	$2^1 = 2$
	$h = 2$	$2^2 = 4$

From the diagram we can infer the following properties:

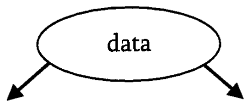
- The number of nodes n in a full binary tree is $2^{h+1} - 1$. Since, there are h levels we need to add all nodes at each level $[2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1]$.
- The number of nodes n in a complete binary tree is between 2^h (minimum) and $2^{h+1} - 1$ (maximum). For more information on this, refer *Priority Queues* chapter.
- The number of leaf nodes in a full binary tree are 2^h.
- The number of NULL links (wasted pointers) in a complete binary tree of n nodes are $n + 1$.

Structure of Binary Trees

Now let us define structure of the binary tree. For simplicity, assume that the data of the nodes are integers. One way to represent a node (which contains the data) is to have two links which points to left and right children along with data fields as shown below:

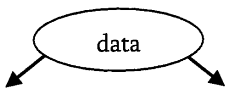


Or



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

Note: In trees, the default flow is from parent to children and showing directed branches is not compulsory. For our discussion, we assume both the below representations are same.



Operations on Binary Trees

Basic Operations

- Inserting an element in to a tree
- Deleting an element from a tree
- Searching for an element
- Traversing the tree

Auxiliary Operations

- Finding size of the tree
- Finding the height of the tree
- Finding the level which has maximum sum
- Finding least common ancestor (LCA) for a given pair of nodes and many more.

Applications of Binary Trees

Following are the some of the applications where *binary trees* play important role:

- Expression trees are used in compilers.
- Huffman coding trees which are used in data compression algorithms.
- Binary Search Tree (BST), which supports search, insertion and deletion on a collection of items in $O(logn)$ (average).
- Priority Queues (PQ), which supports search and deletion of minimum(or maximum) on a collection of items in logarithmic time (in worst case),

6.6 Binary Tree Traversals

In order to process trees, we need a mechanism for traversing them and that forms the subject of this section. The process of visiting all nodes of a tree is called *tree traversal*. Each of the nodes is processed only once but they may be visited more than once. As we have already seen that in linear data structures (like linked lists, stacks, queues, etc...), the elements are visited in sequential order. But, in tree structures there are many different ways.

Tree traversal is like searching the tree except that in traversal the goal is to move through the tree in some particular order. In addition, all nodes are processed in the traversal but searching stops when the required node is found.

Traversal Possibilities

Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node (referred to as "visiting" the node and denotes with "D"), traversing to the left child node (denotes with "L"), and traversing to the right child node (denotes with "R"). This process can be easily described through recursion. Based on the above definition there are 6 possibilities:

1. LDR: Process left subtree, process the current node data and then process right subtree
2. LRD: Process left subtree, process right subtree and then process the current node data
3. DLR: Process the current node data, process left subtree and then process right subtree
4. DRL: Process the current node data, process right subtree and then process left subtree
5. RDL: Process right subtree, process the current node data and then process left subtree
6. RLD: Process right subtree, process left subtree and then process the current node data

Classifying the Traversals

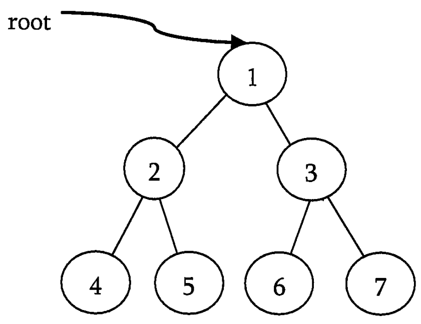
The sequence in which these entities processed defines a particular traversal method. The classification based on the order in which current node is processed. That means, if we are classifying based on current node (D) and if D comes in the middle then it does not matter whether L on left side of D or R is on left side of D. Similarly, it does not matter whether L is on right side of D or R is on right side of D. Due to this, the total 6 possibilities were reduced to 3 and they are:

- Preorder (DLR) Traversal
- Inorder (LDR) Traversal
- Postorder (LRD) Traversal

There is another traversal method which does not depend on above orders and it is:

- Level Order Traversal: This method is inspired from Breadth First Traversal (BFS of Graph algorithms).

Let us use the below diagram for remaining discussion.



PreOrder Traversal

In pre-order traversal, each node is processed before (pre) either of its sub-trees. This is the simplest traversal to understand. However, even though each node is processed before the subtrees, it still requires that some information must be maintained while moving down the tree. In the example above, the 1 is processed first, then the left sub-tree followed by the right subtree. Therefore, processing must return to the right sub-tree after finishing the processing of the left subtree. To move to right subtree after processing left subtree, we must maintain the root information. The obvious ADT for such information is a stack. Because of its LIFO structure, it is possible to get the information about the right subtrees back in the reverse order.

Preorder traversal is defined as follows:

- Visit the root.
- Traverse the left subtree in Preorder.
- Traverse the right subtree in Preorder.

The nodes of tree would be visited in the order: 1 2 4 5 3 6 7

```
void PreOrder(struct BinaryTreeNode *root){
        if(root) {
                printf("%d",root→data);
                PreOrder(root→left);
                PreOrder (root→right);
        }
}
```

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

Non-Recursive Preorder Traversal

In recursive version a stack is required as we need to remember the current node so that after completing the left subtree we can go to right subtree. To simulate the same, first we process the current node and before going to left subtree, we store the current node on stack. After completing the left subtree processing, *pop* the element and go to its right subtree. Continue this process until stack is nonempty.

```
void PreOrderNonRecursive(struct BinaryTreeNode *root){
        struct Stack *S = CreateStack();
        while(1) {
                while(root) {
                        //Process current node
                        printf("%d",root→data);
                        Push(S,root);
                        //If left subtree exists, add to stack
                        root = root→left;
                }
                if(IsEmptyStack(S))
                        break;
                root = Pop(S);
                //Indicates completion of left subtree and current node, now go to right subtree
                root = root→right;
        }
        DeleteStack(S);
}
```

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

InOrder Traversal

In Inorder traversal the root is visited between the subtrees. Inorder traversal is defined as follows:

- Traverse the left subtree in Inorder.
- Visit the root.

- Traverse the right subtree in Inorder.

The nodes of tree would be visited in the order: 4 2 5 1 6 3 7

```
void InOrder(struct BinaryTreeNode *root){
        if(root) {
                InOrder(root→left);
                printf("%d",root→data);
                InOrder(root→right);
        }
}
```

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

Non-Recursive Inorder Traversal

Non-recursive version of Inorder traversal is very much similar to Preorder. The only change is, instead of processing the node before going to left subtree, process it after popping (which indicates after completion of left subtree processing).

```
void InOrderNonRecursive(struct BinaryTreeNode *root){
        struct Stack *S = CreateStack();
        while(1) {
                while(root) {
                        Push(S,root);
                        //Got left subtree and keep on adding to stack
                        root = root→left;
                }
                if(IsEmptyStack(S))
                        break;
                root = Pop(S);
                printf("%d", root→data);        //After popping, process the current node
                //Indicates completion of left subtree and current node, now go to right subtree
                root = root→right;
        }
        DeleteStack(S);
}
```

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

PostOrder Traversal

In postorder traversal, the root is visited after both subtrees. Postorder traversal is defined as follows:

- Traverse the left subtree in Postorder.
- Traverse the right subtree in Postorder.
- Visit the root.

The nodes of tree would be visited in the order: 4 5 2 6 7 3 1

```
void PostOrder(struct BinaryTreeNode *root){
        if(root) {
                PostOrder(root→left);
                PostOrder(root→right);
                printf("%d",root→data);
        }
}
```

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

Non-Recursive Postorder Traversal

In preorder and inorder traversals, after poping the stack element we do not need to visit the same vertex again. But in postorder traversal, each node is visited twice. That means, after processing left subtree we will be visiting the current node and also after processing the right subtree we will be visiting the same current node. But we should be processing the node during the second visit. Here the problem is how to differentiate whether we are returning from left subtree or right subtree?

Trick for this problem is: after popping an element from stack, check whether that element and right of top of the stack are same or not. If they are same then we are done with processing of left subtree and right subtree. In this case we just need to pop the stack one more time and print its data.

```
void PostOrderNonRecursive(struct BinaryTreeNode *root){
        struct Stack *S = CreateStack();
        while (1) {
                if  (root) {
                        Push(S,root);
                        root=root→left;
                }
                else {  if(IsEmptyStack(S)) {
                                printf("Stack is Empty");
                                return;
                        }
                        else    if(Top(S)→right == NULL) {
                                        root = Pop(S);
                                        printf("%d",root→data);
                                        if(root == Top(S)→right) {
                                                printf("%d",Top(S)→data);
                                                Pop(S);
                                        }
                                }
                        if(!IsEmptyStack(S))
                                root=Top(S)→right;
                        else    root=NULL;
                }
        }
        DeleteStack(S);
}
```

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

Level Order Traversal

Level order traversal is defined as follows:

- Visit the root.
- While traversing level l, keep all the elements at level $l + 1$ in queue.
- Go to the next level and visit all the nodes at that level.
- Repeat this until all levels are completed.

The nodes of tree would be visited in the order: 1 2 3 4 5 6 7

```
void LevelOrder(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        struct Queue *Q = CreateQueue();
        if(!root)
                return;
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                //Process current node
                printf("%d", temp→data);
                if(temp→left)
                        EnQueue(Q, temp→left);
                if(temp→right)
                        EnQueue(Q, temp→right);
        }
        DeleteQueue(Q);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$. Since, in the worst case, all the nodes on the entire last level could be in the queue simultaneously.

Problems on Binary Trees

Problem-1 Give an algorithm for finding maximum element in binary tree.

Solution: One simple way of solving this problem is: find the maximum element in left subtree, find maximum element in right sub tree, compare them with root data and select the one which is giving the maximum value. This approach can be easily implemented with recursion.

```
int FindMax(struct BinaryTreeNode  *root) {
        int root_val, left, right, max = INT_MIN;
        if(root !=NULL) {
                root_val = root→data;
                left = FindMax(root→left);
                right = FindMax(root→right);

                // Find the largest of the three values.
                if(left > right)
                        max = left;
                else    max = right;
                if(root_val > max)
                        max = root_val;
        }
        return max;
}
```

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

Problem-2 Give an algorithm for finding maximum element in binary tree without recursion.

Solution: Using level order traversal: just observe the elements data while deleting.

```
int FindMaxUsingLevelOrder(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        int max = INT_MIN;
        struct Queue *Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                // largest of the three values
                if(max < temp→data)
                        max =  temp→data;
                if(temp→left)
                        EnQueue (Q, temp→left);
                if(temp→right)
                        EnQueue (Q, temp→right);
        }
        DeleteQueue(Q);
        return max;
}
```

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

Problem-3 Give an algorithm for searching an element in binary tree.

Solution: Given a binary tree, return true if a node with the data is found in the tree. Recurse down the tree, choose the left or right branch by comparing the data with each nodes data.

```
int FindInBinaryTreeUsingRecursion(struct BinaryTreeNode *root, int data) {
        int temp;
        // Base case == empty tree, in that case, the data is not found so return false
        if(root == NULL)
                return 0;
        else {  //see if found here
                if(data == root→data)
                        return 1;
                else {  // otherwise recur down the correct subtree
                        temp = FindInBinaryTreeUsingRecursion (root→left, data)
                        if(temp != 0)
                                return temp;
                        else    return(FindInBinaryTreeUsingRecursion(root→right, data));
                }
        }
        return 0;
}
```

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

Problem-4 Give an algorithm for searching an element in binary tree without recursion.

Solution: We can use level order traversal for solving this problem. The only change required in level order traversal is, instead of printing the data we just need to check whether the root data is equal to the element we want to search.

```
int SearchUsingLevelOrder(struct BinaryTreeNode *root, int data){
        struct BinaryTreeNode *temp;
        struct Queue *Q;
        if(!root)
                return -1;
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                //see if found here
                if(data == root→data)
                        return 1;
                if(temp→left)
                        EnQueue (Q, temp→left);
                if(temp→right)
                        EnQueue (Q, temp→right);
        }
        DeleteQueue(Q);
        return 0;
}
```

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

Problem-5 Give an algorithm for inserting an element into binary tree.

Solution: Since the given tree is a binary tree, we can insert the element wherever we want. To insert an element, we can use the level order traversal and insert the element wherever we found the node whose left or right child is NULL.

```
void InsertInBinaryTree(struct BinaryTreeNode *root, int data){
        struct Queue *Q;
        struct BinaryTreeNode *temp;
        struct BinaryTreeNode *newNode;
        newNode = (struct BinaryTreeNode *) malloc(sizeof(struct BinaryTreeNode));
        newNode→left = newNode→right = NULL;
        if(!newNode) {
                printf("Memory Error");
                return;
        }
        if(!root) {
                root = newNode;
                return;
        }
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                if(temp→left)
                        EnQueue(Q, temp→left);
                else {  temp→left=newNode;
                        DeleteQueue(Q);
                        return;
                }
                if(temp→right)
                        EnQueue(Q, temp→right);
                else {  temp→right=newNode;
                        DeleteQueue(Q);
                        return;
                }
        }
        DeleteQueue(Q);
}
```

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

Problem-6 Give an algorithm for finding the size of binary tree.

Solution: Calculate the size of left and right subtrees recursively, add 1 (current node) and return to its parent.

```
// Compute the number of nodes in a tree.
int SizeOfBinaryTree(struct BinaryTreeNode *root) {
        if(root==NULL)
                return 0;
        else    return(SizeOfBinaryTree(root→left) + 1 + SizeOfBinaryTree(root→right));
}
```

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

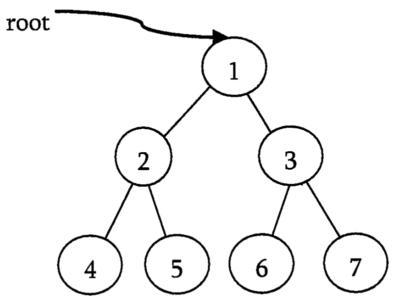
Problem-7 Can we solve the Problem-6 without recursion?

Solution: Yes, using level order traversal.

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

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

Problem-8 Give an algorithm for printing the level order data in reserve order. For example, the output for the below tree should be: 4 5 6 7 2 3 1



Solution:

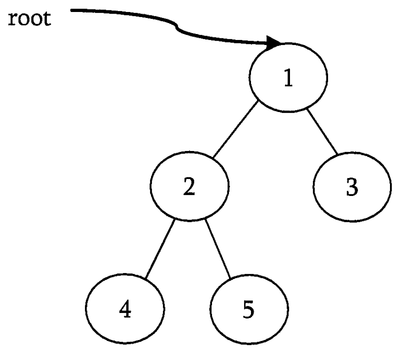
```
void LevelOrderTraversalInReverse(struct BinaryTreeNode *root){
        struct Queue *Q;
        struct Stack *s = CreateStack();
        struct BinaryTreeNode *temp;
        if(!root) return;
        Q = CreateQueue();
        EnQueue(Q, root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                if(temp→right)
                        EnQueue(Q, temp→right);
                if(temp→left)
                        EnQueue (Q, temp→left);
                Push(s, temp);
        }
        while(!IsEmptyStack(s))
                printf("%d",Pop(s)→data);
}
```

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

Problem-9 Give an algorithm for deleting the tree.

Solution: To delete a tree we must traverse all the nodes of the tree and delete them one by one. So which traversal we should use Inorder, Preorder, Postorder or Level order Traversal?

Before deleting the parent node we should delete its children nodes first. We can use postorder traversal as it does the work without storing anything. We can delete tree with other traversals also with extra space complexity. For the following tree nodes are deleted in order – 4, 5, 2, 3, 1.



```
void DeleteBinaryTree(struct BinaryTreeNode *root){
        if(root == NULL)
                return;
        /* first delete both subtrees */
        DeleteBinaryTree(root→left);
        DeleteBinaryTree(root→right);
        //Delete current node only after deleting subtrees
        free(root);
}
```

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

Problem-10 Give an algorithm for finding the height (or depth) of the binary tree.

Solution: Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. This is similar to *PreOrder* tree traversal (and *DFS* of Graph algorithms).

```
int HeightOfBinaryTree(struct BinaryTreeNode   *root){
        int leftheight, rightheight;
        if(root == NULL)
                return 0;
        else {    /* compute the depth of each subtree */
```

```
                leftheight = HeightOfBinaryTree(root→left);
                rightheight = HeightOfBinaryTree(root→right);
                if(leftheight > rightheight)
                        return(leftheight + 1);
                else    return(rightheight + 1);
        }
}
```

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

Problem-11 Can we solve the Problem-10 without recursion?

Solution: Yes. Using level order traversal. This is similar to *BFS* of Graph algorithms. End of level is identified with NULL.

```
int FindHeightofBinaryTree(struct BinaryTreeNode *root){
        int level=1;
        struct Queue *Q;
        if(!root) return 0;
        Q = CreateQueue();
        EnQueue(Q,root);
        // End of first level
        EnQueue(Q,NULL);
        while(!IsEmptyQueue(Q)) {
                root=DeQueue(Q);
                // Completion of current level.
                if(root==NULL) {
                        //Put another marker for next level.
                        if(!IsEmptyQueue(Q))
                                EnQueue(Q,NULL);
                        level++;
                }
                else {  if(root→left)
                                EnQueue(Q, root→left);
                        if(root→right)
                                EnQueue(Q, root→right);
                }
        }
        return level;
}
```

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

Problem-12 Give an algorithm for finding the deepest node of the binary tree.

Solution:

```
struct BinaryTreeNode *DeepestNodeinBinaryTree(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        struct Queue *Q;
        if(!root) return NULL;
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                if(temp→left)
```

```
                        EnQueue(Q, temp→left);
                if(temp→right)
                        EnQueue(Q, temp→right);
        }
        DeleteQueue(Q);
        return temp;
}
```

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

Problem-13 Give an algorithm for deleting an element from binary tree.

Solution: The deletion of a node in binary tree can be implemented as

- Find the node which we want to delete.
- Find the deepest node in the tree.
- Replace the deepest nodes data with node to be deleted.
- Then delete the deepest node.

Problem-14 Give an algorithm for finding the number of leaves in the binary tree without using recursion.

Solution: The set of all nodes whose both left and either right are NULL are called leaf nodes.

```
void NumberOfLeavesInBTusingLevelOrder(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        struct Queue *Q;
        int count = 0;
        if(!root) return 0;
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                if(!temp→left && !temp→right)
                        count++;
                else {  if(temp→left)
                                EnQueue(Q, temp→left);
                        if(temp→right)
                                EnQueue(Q, temp→right);
                }
        }
        DeleteQueue(Q);
        return count;
}
```

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

Problem-15 Give an algorithm for finding the number of full nodes in the binary tree without using recursion.

Solution: The set of all nodes with both left and right children are called full nodes.

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

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

Problem-16 Give an algorithm for finding the number of half nodes (nodes with only one child) in the binary tree without using recursion.

Solution: The set of all nodes with either left or either right child (but not both) are called half nodes.

```
void NumberOfHalfNodesInBTusingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    int count = 0;
    if(!root)
            return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
            temp = DeQueue(Q);
            //we can use this condition also instead of two temp→left ^ temp→right
            if(!temp→left && temp→right || temp→left && !temp→right)
                    count++;
            if(temp→left)
                    EnQueue (Q, temp→left);
            if(temp→right)
                    EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return count;
}
```

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

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

Solution:

Algorithm:

- If both trees are NULL then return true.
- If both trees are not NULL, then compare data and recursively check left and right subtree structures.

```
//Return true if they are structurally identical.
int AreStructurullySameTrees(struct BinaryTreeNode *root1, struct BinaryTreeNode *root2) {
    // both empty→1
    if(root1==NULL && root2==NULL)
            return 1;
    if(root1==NULL || root2==NULL)
```

```
                return 0;
        // both non-empty→compare them
        return(root1→data == root2→data && AreStructurullySameTrees(root1→left, root2→left) &&
                AreStructurullySameTrees(root1→right, root2→right));
}
```

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

Problem-18 Give an algorithm for finding the diameter of the binary tree. The diameter of a tree (sometimes called the *width*) is the number of nodes on the longest path between two leaves in the tree.

Solution: To find the diameter of a tree, first calculate the diameter of left subtree and right sub trees recursively. Among these two values, we need to send maximum along with current level (+1).

```
int DiameterOfTree(struct BinaryTreeNode *root, int *ptr){
        int left, right;
        if(!root)
                return 0;
        left = DiameterOfTree(root→left, ptr);
        right = DiameterOfTree(root→right, ptr);
        if(left + right > *ptr)
                *ptr = left + right;
        return Max(left, right)+1;
}
```

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

Problem-19 Give an algorithm for finding the level which is having maximum sum in the binary tree.

Solution: The logic is very much similar to finding number of levels. The only change is, we need to keep track of sums as well.

```
int FindLevelwithMaxSum(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        int level=0, maxLevel=0;
        struct Queue *Q;
        int currentSum = 0, maxSum = 0;
        if(!root) return 0;
        Q=CreateQueue();
        EnQueue(Q,root);
        EnQueue(Q,NULL);                        //End of first level.
        while(!IsEmptyQueue(Q)) {
                temp =DeQueue(Q);
                // If the current level is completed then compare sums
                if(temp == NULL) {
                        if(currentSum> maxSum) {
                                maxSum = currentSum;
                                maxLevel = level;
                        }
                        currentSum = 0;
                        //place the indicator for end of next level at the end of queue
                        if(!IsEmptyQueue(Q))
                                EnQueue(Q,NULL);
                        level++;
                }
                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).

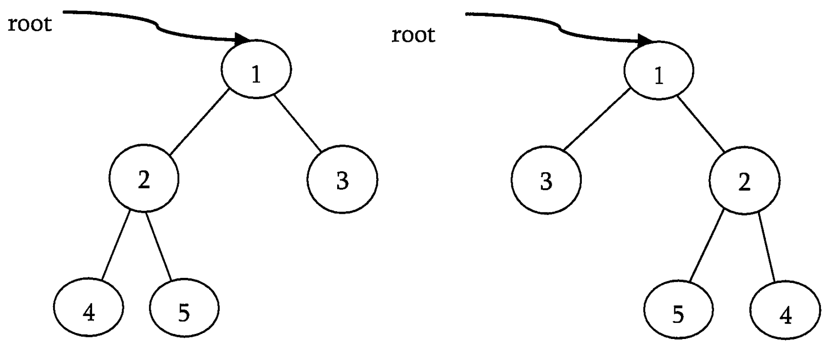
Problem-23 Can we solve Problem-22 without recursion?

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.



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)
}
```

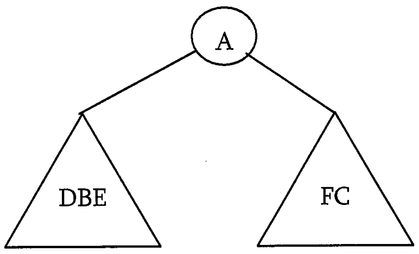
Time Complexity: O(n). Space Complexity: O(n) for recursion.

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

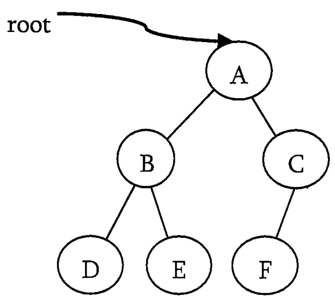
Solution: Let us consider the below traversals:

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;
        }
        // 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.

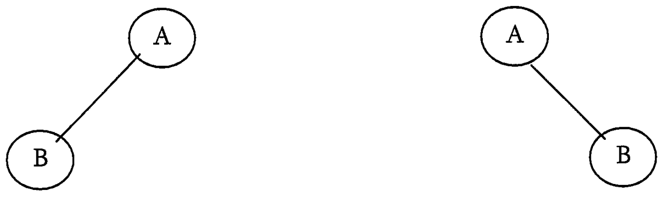
Therefore, following combination can uniquely identify a tree:

- 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

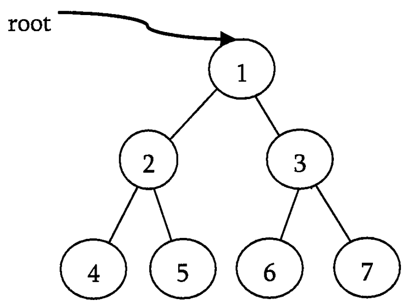
For example, Preorder, Level-order and Postorder traversals are same for above trees:



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.

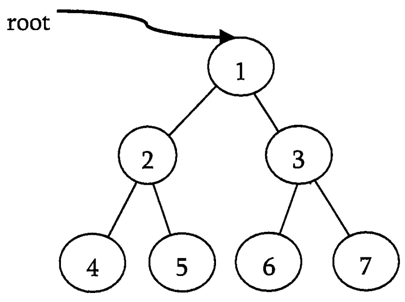


Solution: Apart from the Depth First Search of this tree, we can use the following recursive way to print the ancestors.

```
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).

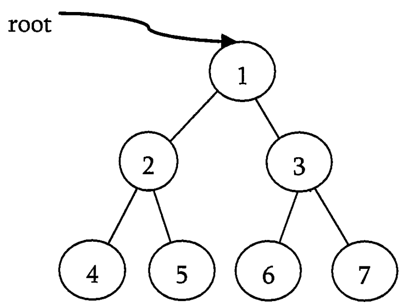
```
void ZigZagTraversal(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        int leftToRight = 1;
        if(!root) return;
        struct Stack *currentLevel = CreateStack(), *nextLevel = CreateStack();
        Push(currentLevel, root);
        while(!IsEmptyStack(currentLevel)) {
                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

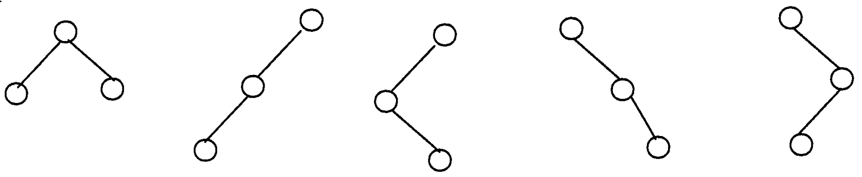


Solution: We can do an inorder traversal and hash the column. We call VerticalSumInBinaryTree(root, 0) which means the root is at column 0. While doing the traversal, hash the column and increase its value by $root \rightarrow data$.

```
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?

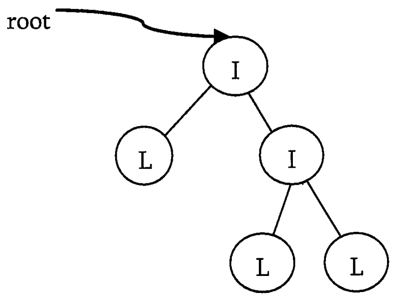
Solution: For example, consider a tree with 3 nodes ($n = 3$), it will have the maximum combination of 5 different (i.e., $2^3 - 3 = 5$) trees.



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 [1].

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 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?

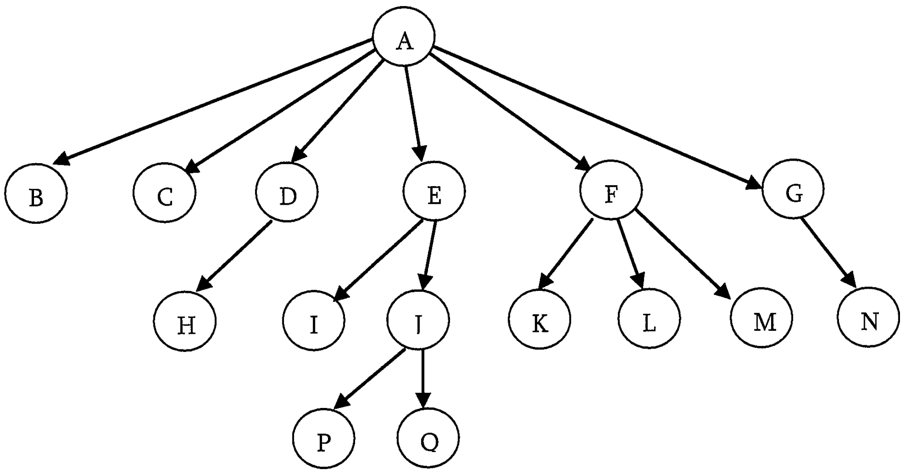
Solution: The trick is to re-use the populated *nextSibling* pointers. As mentioned earlier, we just need one more step for it to work. Before we passed the *left* and *right* to the recursion function itself, we connect the right childs *nextSibling* to the current nodes nextSibling left child. In order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

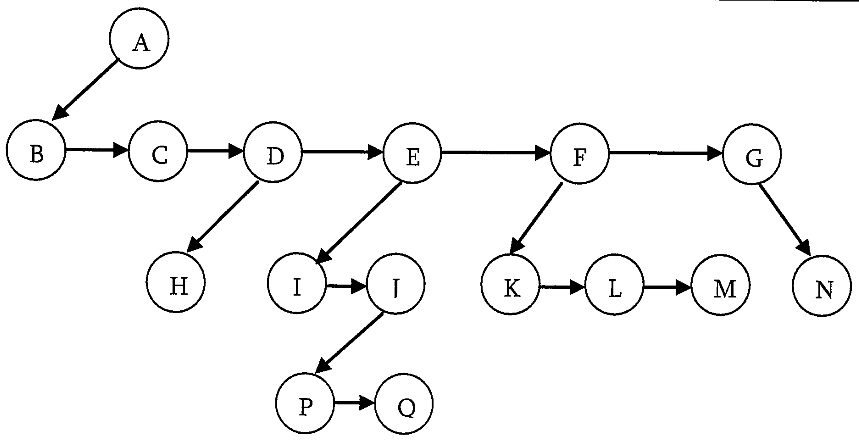
```
        struct TreeNode{
                int data;
                struct TreeNode *firstChild;
                struct TreeNode *secondChild;
                struct TreeNode *thirdChild;
                struct TreeNode *fourthChild;
                struct TreeNode *fifthChild;
                struct 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, in advance we do not know the number of children for each node. In order to solve this problem we need a representation that minimizes the wastage and also accept nodes with any number of children.

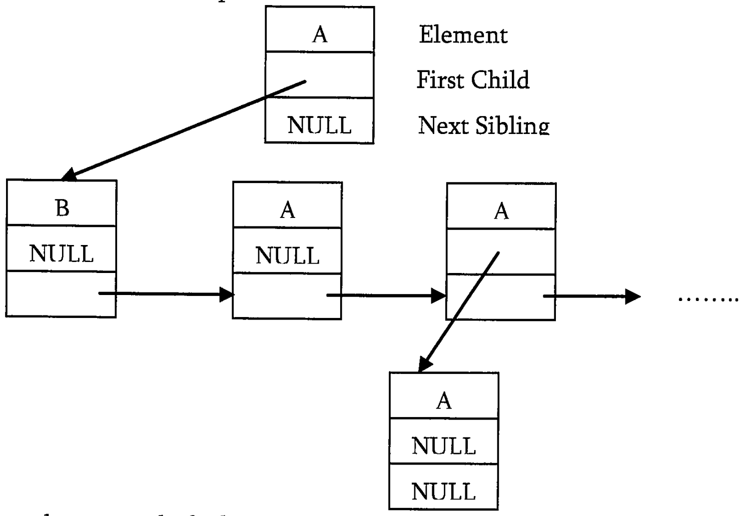
Representation of Generic Trees

Since 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→sibling);
}
```