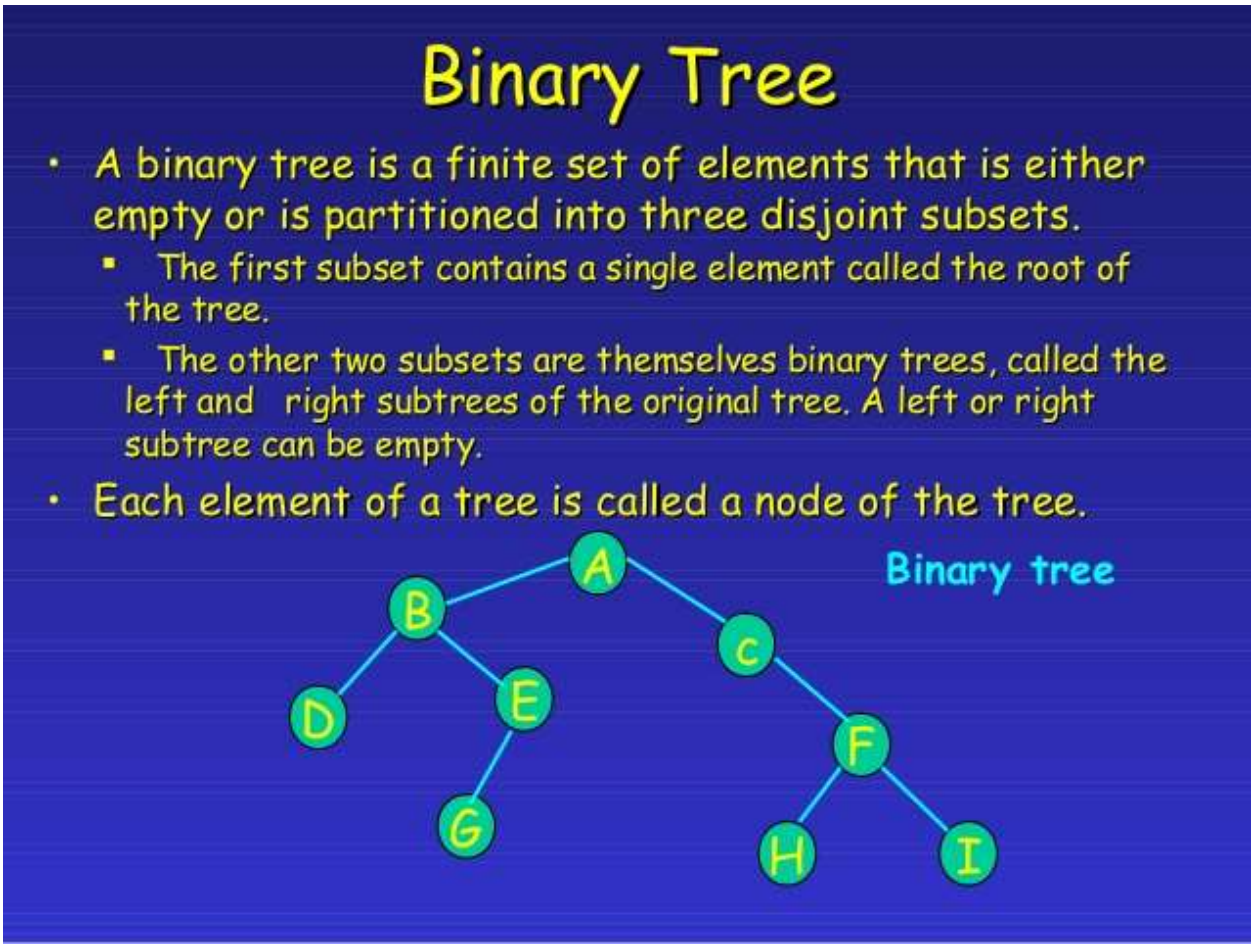
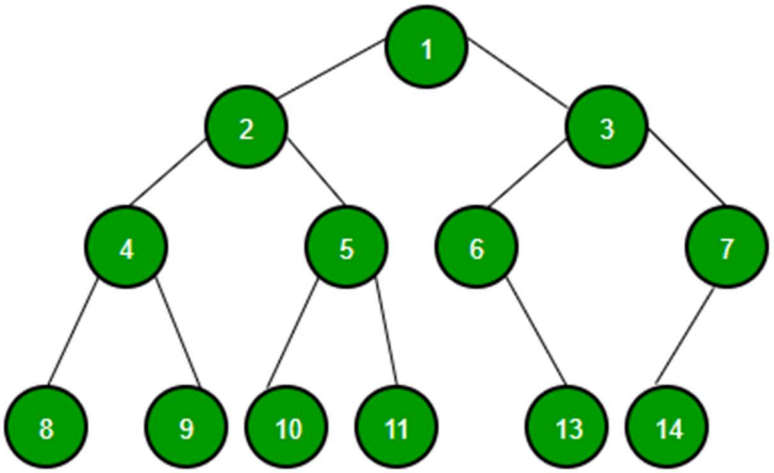
Binary Tree as DS

- A **binary tree** is a special type of **tree** in which every node or vertex has either no child node or one child node or two child nodes.
- A **binary tree** is an important class of a **tree data structure** in which a node can have at most two children. ...
- A **binary tree** is either an empty **tree**.



- Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

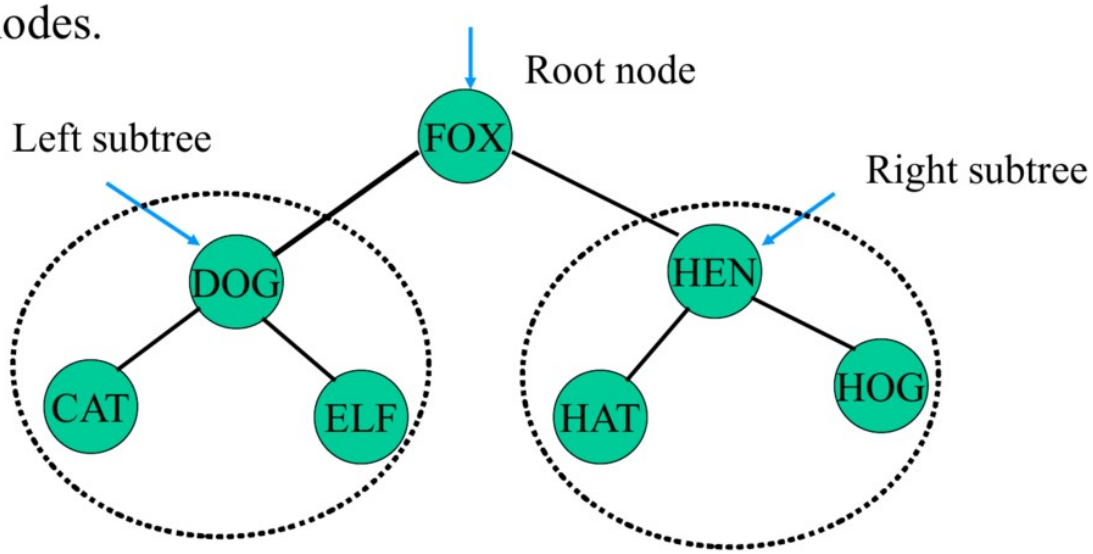


A Binary Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child

Binary Tree

- A data structure whose nodes contain two pointer fields.
- One or both pointers can have the value NULL.
- Each node in a binary tree can have 0, 1, or 2 successor nodes.



There are two ways to represent binary trees. These are:

- **Using arrays**
- **Using Linked lists**

- One of the most important nonlinear data structure is the tree.
- Trees are mainly used to represent data containing the hierarchical relationship between elements, example: records, family trees, and table of contents.
- A tree may be defined as a finite set 'T' of one or more nodes such that there is a node designated as the root of the tree and the other nodes are divided into n>=0 disjoint sets T_1, T_2, T_3, T_4 …. T_n are called the subtrees or children of the root.
- Child node in a binary tree on the left is termed as 'left child node' and node in the right is termed as the 'right child node.'

A binary tree may also be defined as follows:

- A binary tree is either an empty tree
- Or a binary tree consists of a node called the root node, a left subtree and a right subtree, both of which will act as a binary tree once again.
- Binary trees are used to represent a nonlinear data structure.
- There are various forms of Binary trees. Binary trees play a vital role in a software application.
- One of the most important applications of the Binary tree is in the searching algorithm.

A **general tree** is defined as a nonempty finite set T of elements called nodes such that:

- The tree contains the root element
- The remaining elements of the tree form an ordered collection of zeros and more disjoint trees T_1, T_2, T_3, T_4 …. T_n which are called subtrees.

- A **full binary tree** which is also called as **proper binary tree** or **2-tree** is a tree in which **all the node other than the leaves has exact two children.**

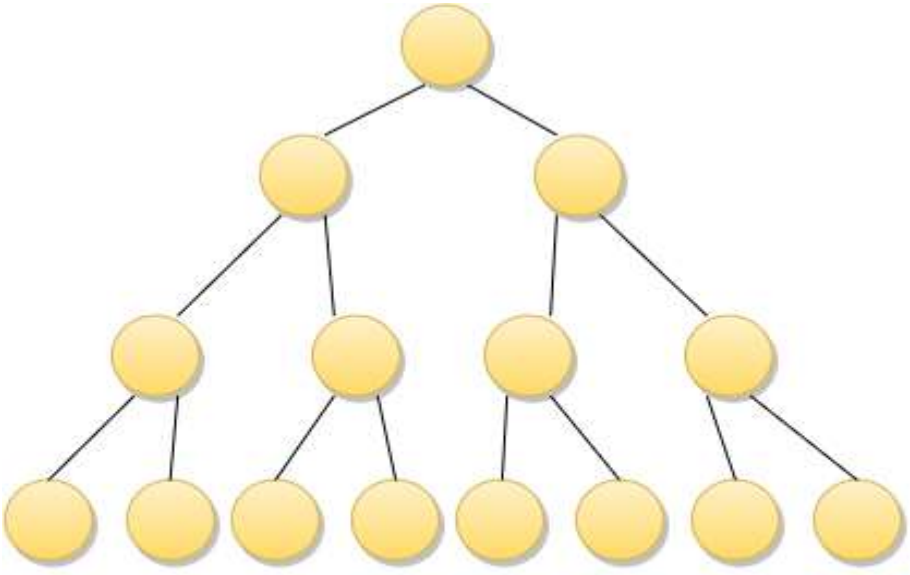


Fig: Full Binary Tree

- A **complete binary tree** is a binary tree in which at every level, except possibly the last, has to be filled and all nodes are as far left as possible.

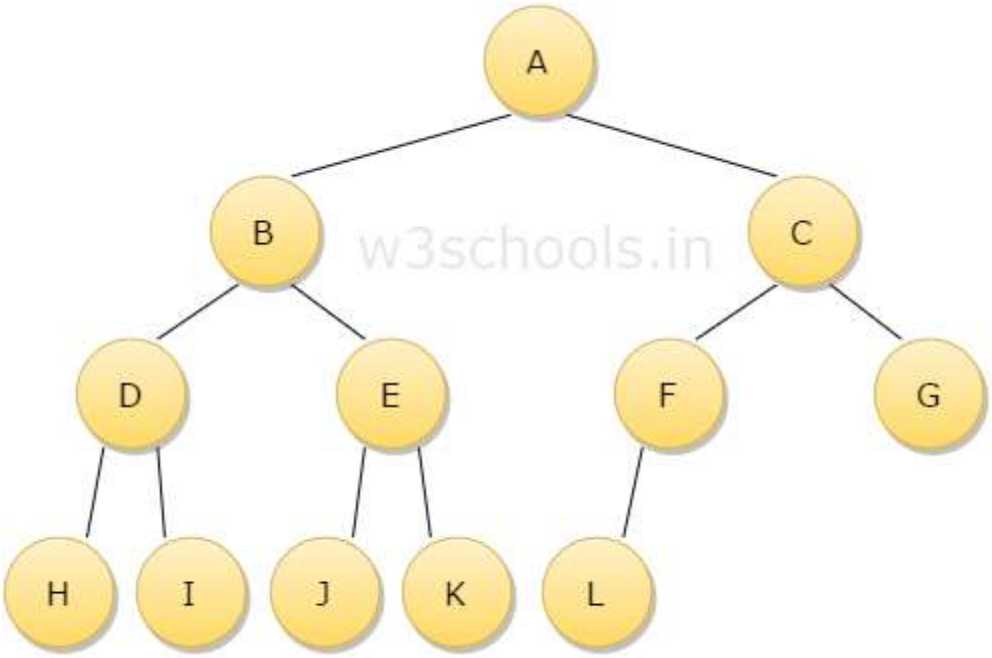


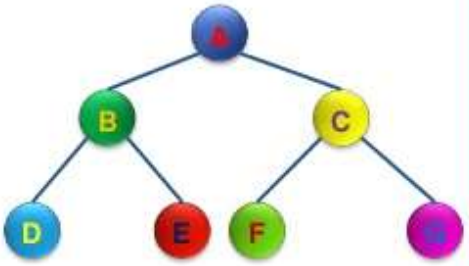
Fig: Complete Binary Tree

- A binary tree can be converted into an **extended binary tree** by adding new nodes to its leaf nodes and to the nodes that have only one child. These new nodes are added in such a way that all the nodes in the resultant tree have either zero or two children. It is also called 2 - tree.
- **The threaded Binary tree** is the tree which is represented using pointers the empty subtrees are set to NULL, i.e. 'left' pointer of the node whose left child is empty subtree is normally set to NULL. These large numbers of pointer sets are used in different ways.

Difference between FBT vs BT

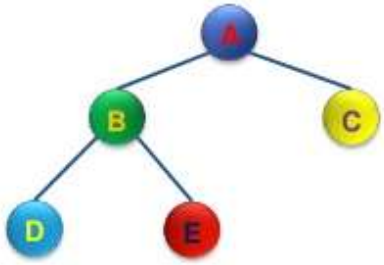
Types of Binary Tree

Full Binary Tree



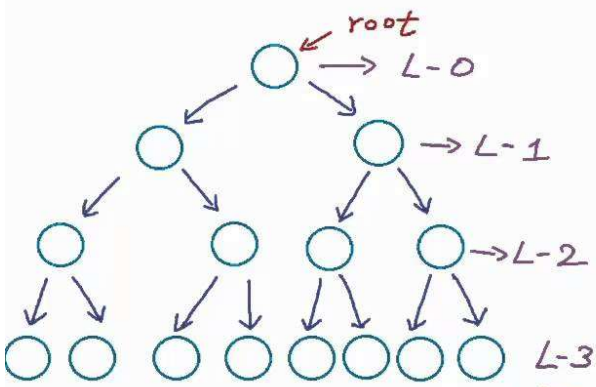
- All nodes (except leaf) have two children.
- Each subtree has same length of path.

Complete Binary Tree



- All nodes (except leaf) have two children.
- Each subtree can has different length of path.

Binary Tree



Perfect Binary tree

Maximum no. of nodes in a binary tree with height h

$= 2^{h+1} - 1$

Height of perfect binary tree with n nodes

$= \log_2(n+1) - 1$

Height of complete binary tree

$= \lfloor \log_2 n \rfloor$

$3 \Leftarrow \lfloor 3.906891 \rfloor \Leftarrow$

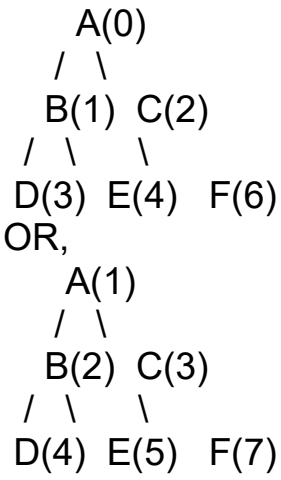
E-1: Binary Tree (Array implementation)

Talking about representation, trees can be represented in two ways:
1) Dynamic Node Representation **(Linked Representation).**

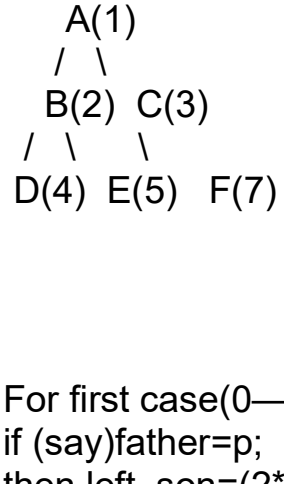
2) Array Representation (**Sequential Representation**).

We are going to talk about the sequential representation of the trees.

To represent tree using an array, the numbering of nodes can start either from 0–(n-1) or 1– n.



OR,



For first case(0—n-1),
if (say)father=p;
then left_son=(2*p)+1;
and right_son=(2*p)+2;
For second case(1—n),
if (say)father=p;
then left_son=(2*p);
and right_son=(2*p)+1;

where father, left_son and right_son are the values of indices of the array.

```
// C++ implementation of tree using array
// numbering starting from 0 to n-1.
#include<bits/stdc++.h>
using namespace std;
char tree[10];
int root(char key)
{
    if(tree[0] != '\0')
        cout << "Tree already had root";
    else
        tree[0] = key;
    return 0;
}

int set_left(char key, int parent)
{
    if(tree[parent] == '\0')
        cout << "\nCan't set child at"
             << (parent * 2) + 1
             << " , no parent found";
    else
        tree[(parent * 2) + 1] = key;
```

```
    return 0;
}

int set_right(char key, int parent)
{
    if(tree[parent] == '\0')
        cout << "\nCan't set child at"
             << (parent * 2) + 2
             << " , no parent found";
    else
        tree[(parent * 2) + 2] = key;
    return 0;
}

int print_tree()
{
    cout << "\n";
    for(int i = 0; i < 10; i++)
    {
        if(tree[i] != '\0')
            cout << tree[i];
        else
            cout << "-";
    }
    return 0;
```

```
}

// Driver Code
int main()
{
    root('A');
    //insert_left('B',0);
    set_right('C', 0);
    set_left('D', 1);
    set_right('E', 1);
    set_right('F', 2);
    print_tree();
    return 0;
}

// This code is contributed by
// Gaurav_Kumar_Raghav
```

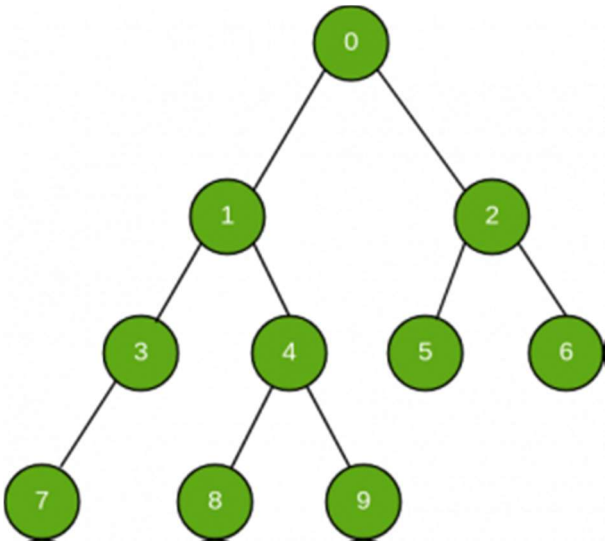
Output:

```
Can't set child at 3, no parent found
Can't set child at 4, no parent found
A-C---F---
```

E-2: Search a node in Binary Tree

Given a Binary tree and a node. The task is to search and check if the given node exists in the binary tree or not. If it exists, print YES otherwise print NO.

Given Binary Tree:



Examples:

Input: Node = 4
Output: YES

Input: Node = 40
Output: NO

The idea is to use any of the tree traversals to traverse the tree and while traversing check if the current node matches with the given node.

Print YES if any node matches with the given node and stop traversing further and if the tree is completely traversed and none of the node matches with the given node then print NO.

Below is the implementation of the above approach:

```
// C++ program to check if a node exists
// in a binary tree
#include <iostream>
using namespace std;

// Binary tree node
struct Node {
    int data;
    struct Node *left, *right;
    Node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
};

// Function to traverse the tree in preorder
// and check if the given node exists in it
bool ifNodeExists(struct Node* node, int key)
{
    if (node == NULL)
        return false;
```

```
    if (node->data == key)
        return true;

    /* then recur on left sutree */
    bool res1 = ifNodeExists(node->left, key);
    // node found, no need to look further
    if(res1) return true;

    /* node is not found in left,
    so recur on right subtree */
    bool res2 = ifNodeExists(node->right, key);

    return res2;
}

// Driver Code
int main()
{
    struct Node* root = new Node(0);
    root->left = new Node(1);
    root->left->left = new Node(3);
    root->left->left->left = new Node(7);
    root->left->right = new Node(4);
    root->left->right->left = new Node(8);
    root->left->right->right = new Node(9);
```

```
    root->right = new Node(2);
    root->right->left = new Node(5);
    root->right->right = new Node(6);

    int key = 4;

    if (ifNodeExists(root, key))
        cout << "YES";
    else
        cout << "NO";

    return 0;
}
```

Output

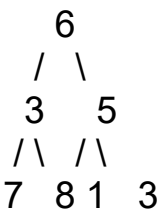
YES

E-3: Check if two nodes are cousins in a Binary Tree

Given the binary Tree and the two nodes say ‘a’ and ‘b’, determine whether the two nodes are cousins of each other or not.

Two nodes are cousins of each other if they are at same level and have different parents.

Example:



Say two node be 7 and 1, result is TRUE.
Say two nodes are 3 and 5, result is FALSE.
Say two nodes are 7 and 5, result is FALSE.

- The idea is to find level of one of the nodes.
- Using the found level, check if ‘a’ and ‘b’ are at this level.
- If ‘a’ and ‘b’ are at given level, then finally check if they are not children of same parent.

Following is the implementation of the above approach.

```
// C program to check if two Nodes in a binary tree are cousins

#include <stdio.h>

#include <stdlib.h>


// A Binary Tree Node

struct Node

{

    int data;

    struct Node *left, *right;

};


// A utility function to create a new Binary Tree Node

struct Node *newNode(int item)

{
```

```
    struct Node *temp =  (struct Node *)malloc(sizeof(struct Node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to check if two Nodes are siblings
int isSibling(struct Node *root, struct Node *a, struct Node *b)
{
    // Base case
    if (root==NULL)  return 0;

    return ((root->left==a && root->right==b)||
            (root->left==b && root->right==a)||
            isSibling(root->left, a, b)||
            isSibling(root->right, a, b));
}

// Recursive function to find level of Node 'ptr' in a binary tree
int level(struct Node *root, struct Node *ptr, int lev)
{
    // base cases
    if (root == NULL) return 0;
    if (root == ptr)  return lev;
```

```
    // Return level if Node is present in left subtree
    int l = level(root->left, ptr, lev+1);
    if (l != 0)  return l;

    // Else search in right subtree
    return level(root->right, ptr, lev+1);
}


// Returns 1 if a and b are cousins, otherwise 0
int isCousin(struct Node *root, struct Node *a, struct Node *b)
{
    //1. The two Nodes should be on the same level in the binary tree.
    //2. The two Nodes should not be siblings (means that they should
    // not have the same parent Node).
    if ((level(root,a,1) == level(root,b,1)) && !(isSibling(root,a,b)))
        return 1;
    else return 0;
}

// Driver Program to test above functions
int main()
{
    struct Node *root = newNode(1);
    root->left = newNode(2);
```

```
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->right->right = newNode(15);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    struct Node *Node1,*Node2;
    Node1 = root->left->left;
    Node2 = root->right->right;

    isCousin(root,Node1,Node2)? puts("Yes"): puts("No");

    return 0;
}
```

Output:

Yes

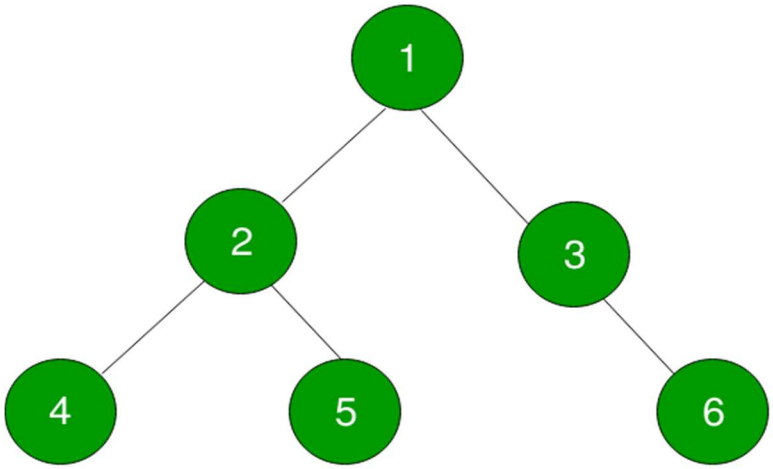
Time Complexity of the above solution is O(n) as it does at most three traversals of binary tree.

E-4: Inorder Successor of a node in Binary Tree

Given a binary tree and a node, we need to write a program to find inorder successor of this node.

Inorder Successor of a node in binary tree is the next node in Inorder traversal of the Binary Tree.

Inorder Successor is NULL for the last node in Inoorder traversal.



In the above diagram, inorder successor of node **4** is **2** and node **5** is **1**.

We need to take care of 3 cases for any node to find its inorder successor as described below:

1. Right child of node is not NULL. If the right child of the node is not NULL then the inorder successor of this node will be the leftmost node in it's right subtree.
2. Right Child of the node is NULL. If the right child of node is NULL. Then we keep finding the parent of the given node x, say p such that p->left = x. For example in the above given tree, inorder successor of node **5** will be **1**. First parent of 5 is 2 but 2->left != 5. So next parent of 2 is 1, now 1->left = 2. Therefore, inorder successor of 5 is 1. Below is the algorithm for this case:
    - Suppose the given node is **x**. Start traversing the tree from **root** node to find **x** recursively.

- If **root** == **x**, stop recursion otherwise find x recursively for left and right subtrees.
- Now after finding the node **x**, recursion will backtrack to the **root**. Every recursive call will return the node itself to the calling function, we will store this in a temporary node say **temp**.Now, when it backtracked to its parent which will be root now, check whether root.left = temp, if not , keep going up

  .

3. If node is the rightmost node. If the node is the rightmost node in the given tree. For example, in the above tree node 6 is the right most node. In this case, there will be no inorder successor of this node. i.e. Inorder Successor of the rightmost node in a tree is NULL.

Below is the implementation of above approach:

```
// CPP program to find inorder successor of a node
#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Temporary node for case 2
Node* temp = new Node;
```

```
// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// function to find left most node in a tree
Node* leftMostNode(Node* node)
{
    while (node != NULL && node->left != NULL)
        node = node->left;
    return node;
}

// function to find right most node in a tree
Node* rightMostNode(Node* node)
{
    while (node != NULL && node->right != NULL)
        node = node->right;
    return node;
}
```

```
// recursive function to find the Inorder Scuccessor
// when the right child of node x is NULL
Node* findInorderRecursive(Node* root, Node* x )
{
    if (!root)
        return NULL;

    if (root==x || (temp = findInorderRecursive(root->left,x)) ||
                (temp = findInorderRecursive(root->right,x)))
    {
        if (temp)
        {
            if (root->left == temp)
            {
                cout << "Inorder Successor of " << x->data;
                cout << " is "<< root->data << "\n";
                return NULL;
            }
        }

        return root;
    }

    return NULL;
```

```
}

// function to find inorder successor of
// a node
void inorderSuccesor(Node* root, Node* x)
{
    // Case1: If right child is not NULL
    if (x->right != NULL)
    {
        Node* inorderSucc = leftMostNode(x->right);
        cout<<"Inorder Successor of "<<x->data<<" is ";
        cout<<inorderSucc->data<<"\n";
    }

    // Case2: If right child is NULL
    if (x->right == NULL)
    {
        int f = 0;

        Node* rightMost = rightMostNode(root);

        // case3: If x is the right most node
        if (rightMost == x)
            cout << "No inorder successor! Right most node.\n";
        else
```

```
        findInorderRecursive(root, x);
    }
}

// Driver program to test above functions
int main()
{
    // Let's construct the binary tree
    // as shown in above diagram

    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

    // Case 1
    inorderSuccesor(root, root->right);

    // case 2
    inorderSuccesor(root, root->left->left);

    // case 3
    inorderSuccesor(root, root->right->right);
```

```
    return 0;
}
```

Output:

```
Inorder Successor of 3 is 6
Inorder Successor of 4 is 2
No inorder successor! Right most node.
```

Another approach:

- We will do a reverse inorder traversal and keep the track of current visited node.
- Once we found the element, last tracked element would be our answer.

Below is the implementation of above approach:

```
// C++ Program to find inorder successor.
#include<bits/stdc++.h>
using namespace std;

// structure of a Binary Node.
struct Node
{
    int data;
    Node* left;
    Node* right;
};
```

```
// Function to create a new Node.
Node* newNode(int val)
{
    Node* temp = new Node;
    temp->data = val;
    temp->left = NULL;
    temp->right = NULL;

    return temp;
}

// function that prints the inorder successor
// of a target node. next will point the last
// tracked node, which will be the answer.
void inorderSuccessor(Node* root,
                      Node* target_node,
                      Node* &next)
{
    // if root is null then return
    if(!root)
        return;

    inorderSuccessor(root->right, target_node, next);
```

```
    // if target node found then enter this condition
    if(root->data == target_node->data)
    {
        // this will be true to the last node
        // in inorder traversal i.e., rightmost node.
        if(next == NULL)
            cout << "inorder successor of "
                << root->data << " is: null\n";
        else
            cout << "inorder successor of "
                << root->data << " is: "
                << next->data << "\n";
    }
    next = root;
    inorderSuccessor(root->left, target_node, next);
}

// Driver Code
int main()
{

    // Let's construct the binary tree
    // as shown in above diagram.
    Node* root = newNode(1);
    root->left = newNode(2);
```

```
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

    // Case 1
    Node* next = NULL;
    inorderSuccessor(root, root->right, next);

    // case 2
    next = NULL;
    inorderSuccessor(root, root->left->left, next);

    // case 3
    next = NULL;
    inorderSuccessor(root, root->right->right, next);

    return 0;
}

// This code is contributed by AASTHA VARMA
```

Output:

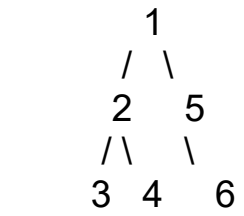
```
inorder successor of 3 is: 6
inorder successor of 4 is: 2
inorder successor of 6 is: null
```

Time Complexity: O(n), where n is the number of nodes in the tree.

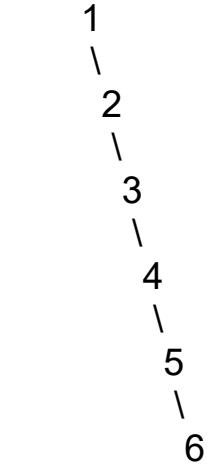
E-5: Flatten a binary tree into linked list

- Given a binary tree, flatten it into linked list in-place.
- Usage of auxiliary data structure is not allowed.
- After flattening, left of each node should point to NULL and right should contain next node in preorder.

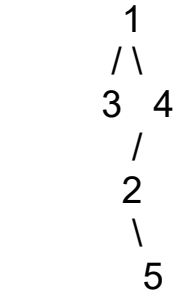
Examples:
Input :



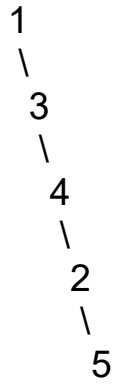
Output :



Input :



Output :



Simple Approach:

A simple solution is to use **Level Order Traversal using Queue.**

- In level order traversal, keep track of previous node.
- Make current node as right child of previous and left of previous node as NULL.

This solution requires queue, but question asks to solve without additional data structure.

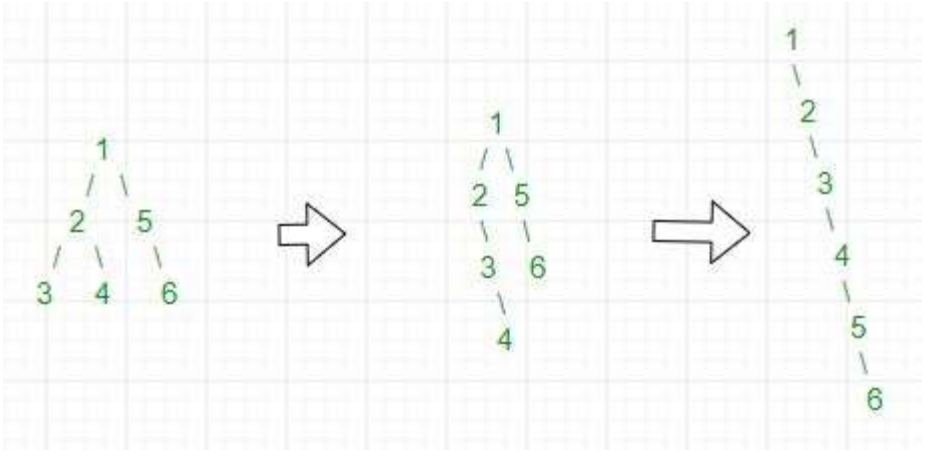
Efficient Without Additional Data Structure:

Recursively look for the node with no grandchildren and both left and right child in the left sub-tree.

Then store node->right in temp and make node->right=node->left. Insert temp in first node NULL on right of node by node=node->right.

Repeat until it is converted to linked list.

For Example,




```
/* Program to flatten a given Binary
   Tree into linked list */
#include <iostream>
using namespace std;

struct Node {
    int key;
    Node *left, *right;
};

/* utility that allocates a new Node
   with the given key  */
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
```

```
    return (node);
}

// Function to convert binary tree into
// linked list by altering the right node
// and making left node point to NULL
void flatten(struct Node* root)
{
    // base condition- return if root is NULL
    // or if it is a leaf node
    if (root == NULL || root->left == NULL &&
                        root->right == NULL) {
        return;
    }

    // if root->left exists then we have
    // to make it root->right
    if (root->left != NULL) {

        // move left recursively
        flatten(root->left);

        // store the node root->right
        struct Node* tmpRight = root->right;
        root->right = root->left;
```

```
        root->left = NULL;

        // find the position to insert
        // the stored value
        struct Node* t = root->right;
        while (t->right != NULL) {
            t = t->right;
        }

        // insert the stored value
        t->right = tmpRight;
    }

    // now call the same function
    // for root->right
    flatten(root->right);
}

// To find the inorder traversal
void inorder(struct Node* root)
{
    // base condition
    if (root == NULL)
        return;
    inorder(root->left);
```

```
    cout << root->key << " ";
    inorder(root->right);
}

/* Driver program to test above functions*/
int main()
{
    /*    1
        /   \
       2     5
      / \     \
     3   4     6 */
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(3);
    root->left->right = newNode(4);
    root->right->right = newNode(6);

    flatten(root);

    cout << "The Inorder traversal after "
            "flattening binary tree ";
    inorder(root);
    return 0;
```

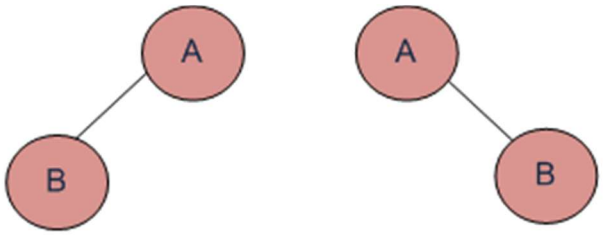
}

Output:

The Inorder traversal after flattening
binary tree 1 2 3 4 5 6

E-6: If you are given two traversal sequences, can you construct the binary tree?

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



Trees having Preorder, Postorder and Level-Order and traversals

Therefore, following combination can uniquely identify a tree.

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

And following do not.
Postorder and Preorder.
Preorder and Level-order.
Postorder and Level-order.

For example, Preorder, Level-order and Postorder traversals are same for the trees given in above diagram.

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

So, even if three of them (Pre, Post and Level) are given, the tree can not be constructed.