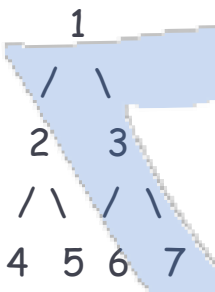
Trees
Session 6

Print path from root to a given node in a binary tree

Given a binary tree with distinct nodes(no two nodes have the same data values). The problem is to print the path from root to a given node **x**. If node **x** is not present then print "No Path".

Examples:

Input :



x = 5

Output : 1->2->5

1. If **root** = **NULL**, return false.
2. push the root's data into **arr[]**.
3. if **root's data** = **x**, return true.
4. if node **x** is present in root's left or right subtree, return true.
5. Else remove root's data value from **arr[]** and return false.

This recursive function can be accessed from other function to check whether node **x** is present or not and if it is present, then the path nodes can be accessed from **arr[]**.

```
1.  bool hasPath(Node *root, vector<int>& arr, int x)
2.  {
3.      if (!root)
4.          return false;
5.
6.      arr.push_back(root->data);
7.      if (root->data == x)
8.          return true;
9.
10.     if (hasPath(root->left, arr, x) ||hasPath(root->right, arr, x))
11.         return true;
12.
13.     arr.pop_back();
14.     return false;
15. }
16. void printPath(Node *root, int x)
17. {
18.     vector<int> v;
19.     if (hasPath(root, arr, x))
20.     {
21.         for (int i=0; i<arr.size()-1; i++)
22.             cout << arr[i] << "->";
23.         cout << arr[arr.size() - 1];
24.     }
25.     else
26.         cout << "No Path";
27. }
```

Special Case:

1. Step 1:
   Write the data of the node as its decimal equivalent example 12=1100

2. Step 2:
   Now leave the most significant digit i.e. leftmost 1 and we are left with 100

3. Step 3:
   Now start with left and whenever 1 is encountered , move right , else move left.

Hence ,

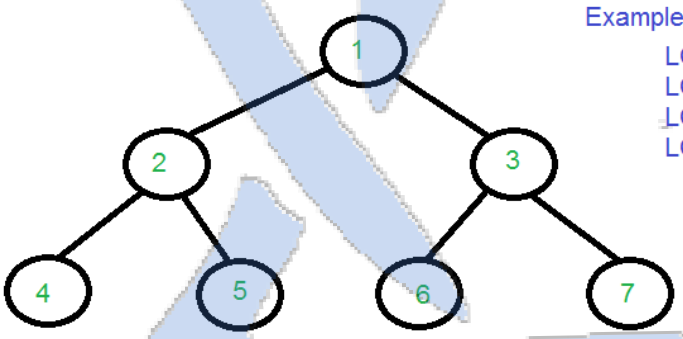
1-> 3-> 6-> 12

This is how the path of the node is traced.

Lowest Common Ancestor in a Binary Tree

Given a binary tree (not a binary search tree) and two values say n1 and n2, write a program to find the least common ancestor.

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).



Examples

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

Method 1 (By Storing root to n1 and root to n2 paths):

Following is a simple O(n) algorithm to find LCA of n1 and n2.

1) Find a path from the root to n1 and store it in a vector or array.

2) Find a path from the root to n2 and store it in another vector or array.

3) Traverse both paths till the values in arrays are the same. Return the common element just before the mismatch.

Following is the implementation of the above algorithm.

```
1.    bool findPath(Node *root, vector<int> &path, int k)
2.    {
3.        if (root == NULL)
4.                return false;
5.
6.        path.push_back(root->key);
7.        if (root->key == k)
8.          return true;
9.
10.       if ( (root->left && findPath(root->left, path, k)) || (root->right &&
findPath(root->right, path, k)) )
11.         return true;
12.
13.       path.pop_back();
14.     return false;
15.   }
16.
17.   int findLCA(Node *root, int n1, int n2)
18.   {
19.       vector<int> path1, path2;
20.       if ( !findPath(root, path1, n1) || !findPath(root, path2, n2))
21.          return -1;
22.      int i;
23.     for (i = 0; i < path1.size() && i < path2.size() ; i++)
24.        if (path1[i] != path2[i])
25.           break;
26.     return path1[i-1];
27.   }
```

Time Complexity: The time complexity of the above solution is O(n). The tree is traversed twice, and then path arrays are compared.

Method 2 (Using Single Traversal)

Method 1 finds LCA in O(n) time but requires three tree traversals plus extra spaces for path arrays. If we assume that the keys n1 and n2 are present in Binary Tree, we can find LCA using a single traversal of Binary Tree and without extra storage for path arrays.

The idea is to traverse the tree starting from the root. If any of the given keys (n1 and n2) matches with the root, then the root is LCA (assuming that both keys are present). If the root doesn't match with any of the keys, we recur for the left and right subtree. The node which has one key present in its left subtree and the other key present in the right subtree is the LCA. If both keys lie in the left subtree, then the left subtree has LCA also, otherwise, LCA lies in the right subtree.

```
1.    struct Node *findLCA(struct Node* root, int n1, int n2)
2.    {
3.         if (root == NULL)
4.               return NULL;
5.
6.         if (root->key == n1 || root->key == n2)
7.          return root;
8.
9.        Node *left_lca  = findLCA(root->left, n1, n2);
10.       Node *right_lca = findLCA(root->right, n1, n2);
11.
12.        if (left_lca && right_lca)
13.   return root;
14.
15.        return (left_lca != NULL)? left_lca: right_lca;
16.   }
```

Note that the above method assumes that keys are present in Binary Tree. If one key is present and the other is absent, then it returns the present key as LCA (Ideally should have returned NULL).

Extend this method to handle all cases bypassing two boolean variables v1 and v2. v1 is set as true when n1 is present in the tree and v2 is set as true if n2 is present in the tree.

Special case :

Step 1:

Write the binary equivalent of the nodes of which LCA is to be calculated

Example 8 and 11 :

8 : 1000

11: 1011

Step 2:

Remove the most significant digit

Step 3:

Now keep moving from left most digit and the moment you get the different values that is lca

0 0 1

0 1 1

  ^

Min distance between two given nodes of a Binary Tree

Given a binary tree and two node values your task is to find the minimum distance between them.

Example 1:

Input:

```
      1
    /  \
   2    3
```

a = 2, b = 3

Output: 2

Explanation: The tree formed is:

```
     1
   /   \
  2     3
```

We need the distance between 2 and 3.Being at node 2, we need to take two steps ahead in order to reach node 3.The path followed will be:2 -> 1 -> 3. Hence, the result is 2.

Expected Time Complexity: O(N).

Expected Auxiliary Space: O(Height of the Tree).

Constraints:

1 <= Number of nodes <= 104

1 <= Data of a node <= 105

Dist(n1, n2) = Dist(root, n1) + Dist(root, n2) - 2*Dist(root, lca)

'n1' and 'n2' are the two given keys

'root' is root of given Binary Tree.

'lca' is lowest common ancestor of n1 and n2

Dist(n1, n2) is the distance between n1 and n2.

Time Complexity: Time complexity of the above solution is O(n) as the method does a single tree traversal.

Better Solution :

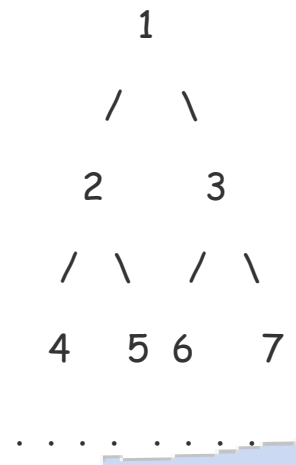
We first find the LCA of two nodes. Then we find the distance from LCA to two nodes.

```
1.    Node* LCA(Node * root, int n1,int n2)
2.    {
3.      // Your code here
4.      if (root == NULL)
5.        return root;
6.      if (root->data == n1 || root->data == n2)
7.        return root;
8.
9.      Node* left = LCA(root->left, n1, n2);
10.     Node* right = LCA(root->right, n1, n2);
11.
12.     if (left != NULL && right != NULL)
13.         return root;
14.     if (left == NULL && right == NULL)
15.           return NULL;
16.     if (left != NULL)
17.        return LCA(root->left, n1, n2);
18.
```

```
19.       return LCA(root->right, n1, n2);
20.    }
21.
22.    int findLevel(Node *root, int k, int level)
23.    {
24.       if(root == NULL) return -1;
25.       if(root->data == k) return level;
26.
27.       int left = findLevel(root->left, k, level+1);
28.      if (left == -1)
29.          return findLevel(root->right, k, level+1);
30.       return left;
31.    }
32.
33.    int findDist(Node* root, int a, int b)
34.    {
35.       Node* lca = LCA(root, a , b);
36.
37.       int d1 = findLevel(lca, a, 0);
38.       int d2 = findLevel(lca, b, 0);
39.
40.       return d1 + d2;
41.       // Your code here
42.    }
```

Shortest distance in infinite tree

Consider you have an infinitely long binary tree having the pattern as below



Given two nodes with values x and y, the task is to find the length of the shortest path between the two nodes.

Example 1:

Input:
X = 1
Y = 3
Output:
1
Explanation:
3 is the child of 1 so, distance between them is 1.

Expected Time Complexity: O(log (y - x))

Expected Auxiliary Space: O(1)

Root to Leaf Paths

Given a Binary Tree of size N, you need to find all the possible paths from root node to all the leaf node's of the binary tree.

Example 1:

Input:

Output: 1 2 #1 3 #

Explanation:

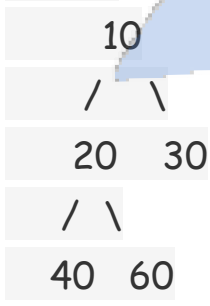
All possible paths:

1->2

1->3

Example 2:

Input:



Output: 10 20 40 #10 20 60 #10 30 #

```
1.  void vectors(Node*root , vector<int>v , vector<vector<int>>&V)
2.  {
3.     if(root == NULL)
4.        return;
5.     v.push_back(root->data);
6.     if(!root->left && !root->right)
7.       {
8.          V.push_back(v);
9.       }
10.    else
11.    {
12.       vectors(root->left , v , V);
13.      vectors(root->right , v , V);
14.     //  v.pop_back();
15.    }
16. }
17.
18. vector<vector<int>> Paths(Node* root)
19. {
20.    vector<vector<int>>V;
21.    vector<int>v;
22.    vectors(root , v , V);
23.    return V;
24.    // Code here
25. }
```

Root to leaf paths sum

Given a binary tree of N nodes, where every node value is a number. Find the sum of all the numbers which are formed from root to leaf paths.

Example 1:

Input :

Output: 13997

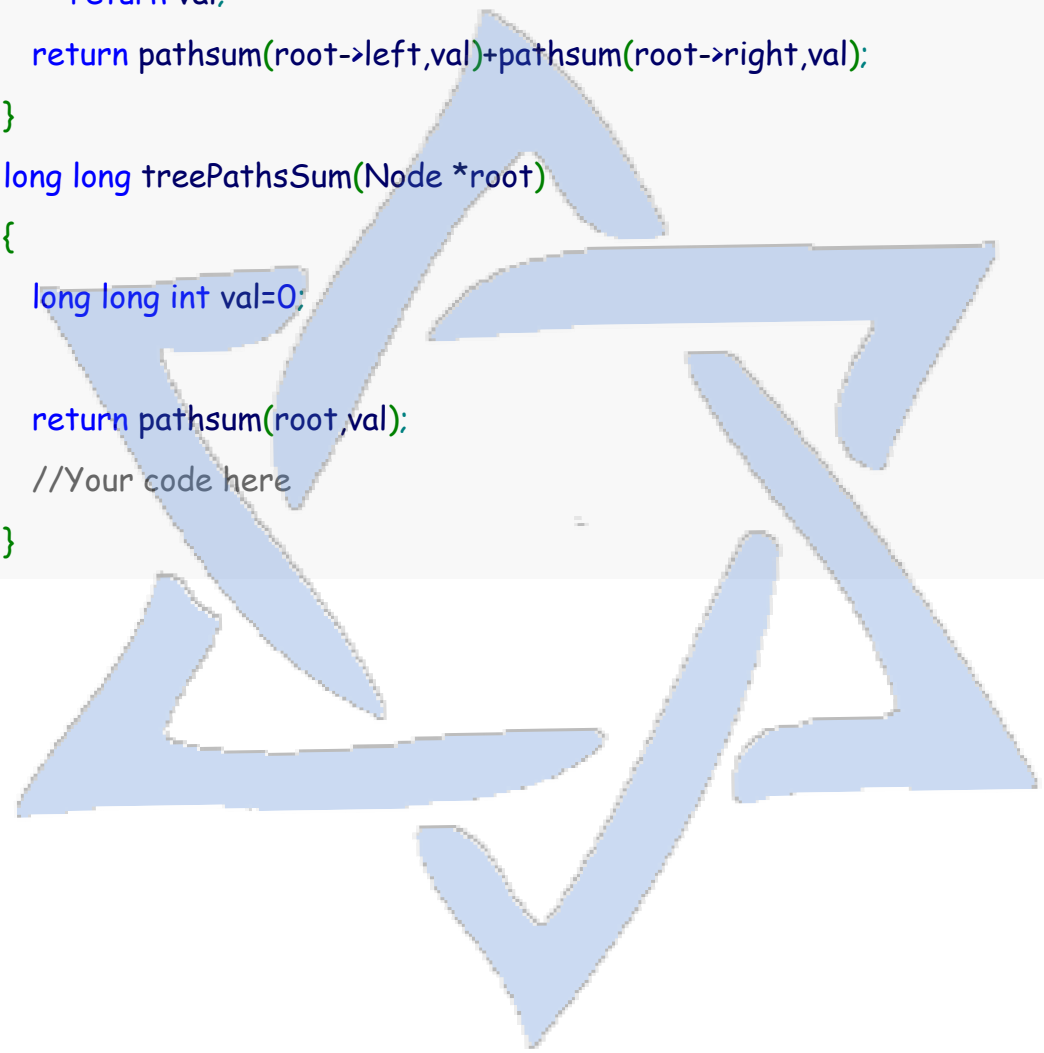
Explanation :

There are 4 leaves, hence 4 root to leaf paths:

Path	Number
6->3->2	632
6->3->5->7	6357
6->3->5->4	6354
6->5>4	654

Answer = 632 + 6357 + 6354 + 654 = 13997

```
1.  long long pathsum(Node*root,long long int val)
2.  {
3.    if(root==NULL)
4.       return 0;
5.    val=val*10+root->data;
6.
7.    if(!root->left&&!root->right)
8.       return val;
9.    return pathsum(root->left,val)+pathsum(root->right,val);
10. }
11. long long treePathsSum(Node *root)
12. {
13.   long long int val=0;
14.
15.   return pathsum(root,val);
16.   //Your code here
17. }
```



Deletion in a Binary Tree

Given a Binary Tree of size **N**, your task is to complete the function **deletionBT()**, that should delete a given node from the tree by making sure that tree shrinks from the bottom (the deleted node is replaced by bottommost and rightmost node).

Example:

Example 1:

Input:

Key=1

```
      1
    /   \
   4     7
  / \
 5   6
```

Output:

5 4 6 7

Explanation:

Modified Tree after deletion the node with key = 1

```
   7
  /
 4
/ \
5  6
```

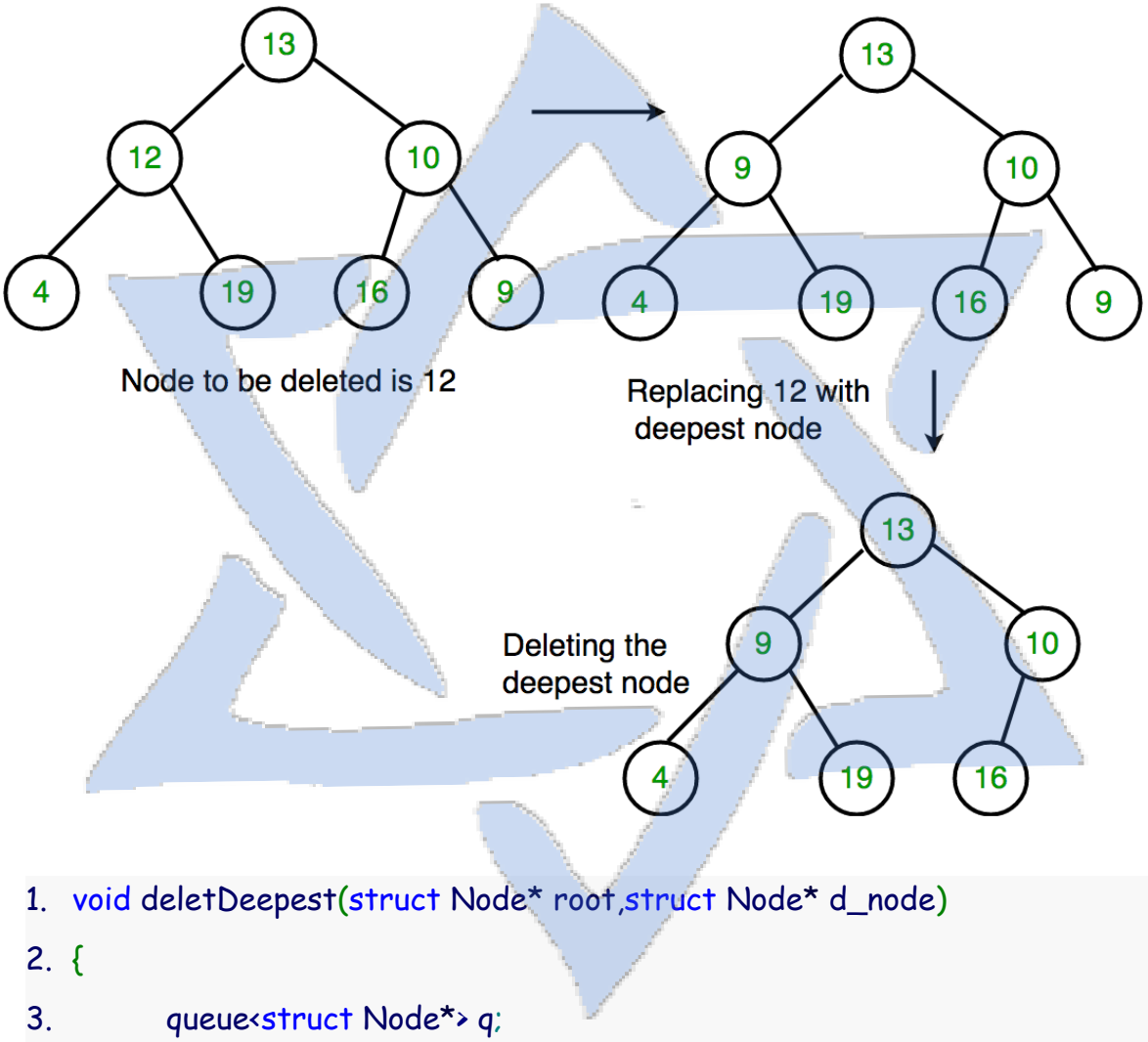
The Inorder traversal of the modified tree is 5 4 6 7

Algorithm

1. Starting at root, find the deepest and rightmost node in binary tree and node which we want to delete.

2. Replace the deepest rightmost node's data with node to be deleted.

3. Then delete the deepest rightmost node.




```
1.  void deletDeepest(struct Node* root,struct Node* d_node)
2.  {
3.          queue<struct Node*> q;
4.          q.push(root);
5.
6.          // Do level order traversal until last node
7.          struct Node* temp;
8.          while (!q.empty()) {
```

```
9.                temp = q.front();
10.               q.pop();
11.               if (temp == d_node) {
12.                       temp = NULL;
13.                       delete (d_node);
14.                       return;
15.               }
16.               if (temp->right) {
17.                       if (temp->right == d_node) {
18.                               temp->right = NULL;
19.                               delete (d_node);
20.                               return;
21.                       }
22.                       else
23.                               q.push(temp->right);
24.               }
25.
26.               if (temp->left) {
27.                       if (temp->left == d_node) {
28.                               temp->left = NULL;
29.                               delete (d_node);
30.                               return;
31.                       }
32.                       else
33.                               q.push(temp->left);
34.               }
35.       }
36.}
37.
38./* function to delete element in binary tree */
```

```
39.Node* deletion(struct Node* root, int key)
40.{
41.         if (root == NULL)
42.                 return NULL;
43.
44.         if (root->left == NULL && root->right == NULL) {
45.                 if (root->key == key)
46.                         return NULL;
47.                 else
48.                         return root;
49.         }
50.
51.         queue<struct Node*> q;
52.         q.push(root);
53.
54.         struct Node* temp;
55.         struct Node* key_node = NULL;
56.
57.
58.         while (!q.empty()) {
59.                 temp = q.front();
60.                 q.pop();
61.
62.                 if (temp->key == key)
63.                         key_node = temp;
64.
65.                 if (temp->left)
66.                         q.push(temp->left);
67.
68.                 if (temp->right)
```

```
69.                         q.push(temp->right);
70.         }
71.
72.         if (key_node != NULL) {
73.                 int x = temp->key;
74.                 deletDeepest(root, temp);
75.                 key_node->key = x;
76.         }
77.         return root;
78.}
```