* Longest Common Ancestor (LCA)
  + In BST O(h) :-
    - struct node \*lca(struct node\* root, int n1, int n2)

{

if (root == NULL) return NULL;

if (root->data > n1 && root->data > n2)

return lca(root->left, n1, n2);

if (root->data < n1 && root->data < n2)

return lca(root->right, n1, n2);

return root;

}

* + In General Binary Tree :-  
    <https://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/>
    - Following is simple O(n) algorithm to find LCA of n1 and n2.

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

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

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

* + - In Single Traversal O(n)

-> Assumes that the keys are present in the tree.  
-> Can be modified to include keys absent cases also.  
struct Node \*findLCA(struct Node\* root, int n1, int n2)

{

if (root == NULL) return NULL;

if (root->key == n1 || root->key == n2)

return root;

Node \*left\_lca = findLCA(root->left, n1, n2);

Node \*right\_lca = findLCA(root->right, n1, n2);

if (left\_lca && right\_lca) return root;

return (left\_lca != NULL)? left\_lca: right\_lca;

}

* + - Using parent Pointer :-

<https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-tree-set-2-using-parent-pointer/>

-> O(h) | O(h)   
Node \*LCA(Node \*n1, Node \*n2)

{

map <Node \*, bool> ancestors;

while (n1 != NULL)

{

ancestors[n1] = true;

n1 = n1->parent;

}

while (n2 != NULL)

{

if (ancestors.find(n2) != ancestors.end())

return n2;

n2 = n2->parent;

}

return NULL;

}

-> O(h) | O(1)

Based on depth difference

Node \*LCA(Node \*n1, Node \*n2)

{

// Find depths of two nodes and differences

int d1 = depth(n1), d2 = depth(n2);

int diff = d1 - d2;

if (diff < 0)

{

Node \* temp = n1;

n1 = n2;

n2 = temp;

diff = -diff;

}

// Move n1 up until it reaches the same level as n2

while (diff--)

n1 = n1->parent;

// Now n1 and n2 are at same levels

while (n1 && n2)

{

if (n1 == n2)

return n1;

n1 = n1->parent;

n2 = n2->parent;

}

return NULL;

}

* + - **Important  
      RMQ** :- <https://www.geeksforgeeks.org/find-lca-in-binary-tree-using-rmq/>  
      **RMQ requires preprocessing in O(n) but then finds LCA in )(logn).  
      So if there are queries on LCA to be performed, converting LCA to RMQ is required.**

We require three arrays for implementation:

-> Nodes visited in order of Euler tour of T

-> Level of each node visited in Euler tour of T

-> Index of the first occurrence of a node in Euler tour of T (since any occurrence would be good, let’s track the first one)

* + - Hashing
* Distance between 2 nodes :-

<https://www.geeksforgeeks.org/find-distance-between-two-nodes-of-a-binary-tree/>  
-> Dist(n1, n2) = Dist(root, n1) + Dist(root, n2) - 2\*Dist(root, lca)

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

* Kth ancestor of a node :-
  + BFS :-

<https://www.geeksforgeeks.org/kth-ancestor-node-binary-tree/>  
The idea to do this is to first traverse the binary tree and store the ancestor of each node in an array of size n. For example, suppose the array is anecestor[n]. Then at index i, ancestor[i] will store the ancestor of ith node.So, the 2nd ancestor of ith node will be ancestor[ancestor[i]] and so on. We will use this idea to calculate the kth ancestor of the given node.

* + DFS :-

<https://www.geeksforgeeks.org/kth-ancestor-node-binary-tree-set-2/>  
first find the node and then backtrack to the kth parent.

Node\* kthAncestorDFS(Node \*root, int node , int &k)

{

if (!root)

return NULL;

if (root->data == node||

(temp = kthAncestorDFS(root->left,node,k)) ||

(temp = kthAncestorDFS(root->right,node,k)))

{

if (k > 0)

k--;

else if (k == 0)

{

// print the kth ancestor

cout<<"Kth ancestor is: "<<root->data;

// return NULL to stop further backtracking

return NULL;

}

// return current node to previous call

return root;

}

}

* + Find the path to the required node and print the kth element from the last.

<https://www.geeksforgeeks.org/k-th-ancestor-of-a-node-in-binary-tree-set-3/>

* Diameter of a tree :-
  + Recursion :-

int diameter(struct node \* tree)

{

if (tree == NULL)

return 0;

int lheight = height(tree->left);

int rheight = height(tree->right);

int ldiameter = diameter(tree->left);

int rdiameter = diameter(tree->right);

return max(lheight + rheight + 1, max(ldiameter, rdiameter));

}

O(n\*n), can be optimised to O(n) by getting height in the same recursive call.

int diameterOpt(struct node \*root, int\* height)

{

int lh = 0, rh = 0;

int ldiameter = 0, rdiameter = 0;

if(root == NULL)

{

\*height = 0;

return 0; /\* diameter is also 0 \*/

}

ldiameter = diameterOpt(root->left, &lh);

rdiameter = diameterOpt(root->right, &rh);

\*height = max(lh, rh) + 1;

return max(lh + rh + 1, max(ldiameter, rdiameter));

}

* + 2 BFS/DFS solution :-

<https://www.geeksforgeeks.org/diameter-tree-using-dfs/>  
Do 2 DFS, First from a random node to get farthest node from it, then from the farthest node to find its farthest node. Diameter will be from these 2 nodes.

* Diameter in N-ary Tree :-
  + Same method as Binary Tree, diameter is max of (sum max 2 heights of child subtrees + 1, max of diameter of a child subtree).  
    <https://www.geeksforgeeks.org/diameter-n-ary-tree/>
  + Using 2 BFS or DFS :-

<https://www.geeksforgeeks.org/diameter-n-ary-tree-using-bfs/>

* **Vertical Order Traversal** :-
  + Using Horizontal distance from root O(w\*n) or O(n\*n) | O(1)

<https://www.geeksforgeeks.org/print-binary-tree-vertical-order/>  
Algorithm :-

findMinMax(tree, min, max, hd)

if tree is NULL then return;

if hd is less than min then

min = hd;

else if hd is greater than max then

\*max = hd;

findMinMax(tree->left, min, max, hd-1);

findMinMax(tree->right, min, max, hd+1);

printVerticalLine(tree, line\_no, hd)

if tree is NULL then return;

if hd is equal to line\_no, then

print(tree->data);

printVerticalLine(tree->left, line\_no, hd-1);

printVerticalLine(tree->right, line\_no, hd+1);

For line\_no from min to max : printVerticalLine(line\_no)

* + Map based Method O(nlogn) can be optimised to O(n) | O(n)

<https://www.geeksforgeeks.org/print-binary-tree-vertical-order-set-2/>  
Algorithm :-

void getVerticalOrder(Node\* root, int hd, map<int, vector<int>> &m)

{

if (root == NULL) return;

m[hd].push\_back(root->key);

getVerticalOrder(root->left, hd-1, m);

getVerticalOrder(root->right, hd+1, m);

}

void printVerticalOrder(Node\* root)

{

map < int,vector<int> > m;

int hd = 0;

getVerticalOrder(root, hd,m);

map< int,vector<int> > :: iterator it;

for (it=m.begin(); it!=m.end(); it++)

{

for (int i=0; i<it->second.size(); ++i)

cout << it->second[i] << " ";

cout << endl;

}

}

-> This solution prints the nodes in a vertical line in the order they appear in the traversal. A node at lower height may be printed before one at higher height. The next solution also maintains the vertical order.

* + Using Hashmap with Level order Traversal :-

<https://www.geeksforgeeks.org/print-a-binary-tree-in-vertical-order-set-3-using-level-order-traversal/>  
The solution is similar to before but instead do BFS instead of DFS. This will maintain the vertical order.

* Top view of a Tree :- O(n)

<https://www.geeksforgeeks.org/print-nodes-in-the-top-view-of-binary-tree-set-3/>

Do vertical order traversal using BFS and print only one node in each vertical level.

* Left View of a Tree :-

<https://www.geeksforgeeks.org/print-left-view-binary-tree/>

* + Do level order traversal and print only 1 node (first) at each level. (Iterative Method)
  + The problem can also be solved using simple recursive traversal. We can keep track of level of a node by passing a parameter to all recursive calls. The idea is to keep track of maximum level also. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the first node in its level (Note that we traverse the left subtree before right subtree). Following is the implementation. (Recursive Method)
* Right View of a Tree :-

<https://www.geeksforgeeks.org/print-right-view-binary-tree-2/>  
Similar to Left View, Just print last node in level order traversal and recur for right child before the left child in Recursive method.

* **Print nodes at k distance from a given node** **O(n)**  
  <https://www.geeksforgeeks.org/print-nodes-distance-k-given-node-binary-tree/>

void printkdistanceNodeDown(node \*root, int k)

{

if (root == NULL || k < 0) return;

if (k==0)

{

cout << root->data << endl;

return;

}

printkdistanceNodeDown(root->left, k-1);

printkdistanceNodeDown(root->right, k-1);

}

int **printkdistanceNode**(node\* root, node\* target , int k)

{

if (root == NULL) return -1;

if (root == target)

{

printkdistanceNodeDown(root, k);

return 0;

}

int dl = printkdistanceNode(root->left, target, k);

if (dl != -1)

{

// If root is at distance k from target, print root

// Note that dl is Distance of root's left child from target

if (dl + 1 == k)

cout << root->data << endl;

// Else go to right subtree and print all k-dl-2 distant nodes

// Note that the right child is 2 edges away from left child

else

printkdistanceNodeDown(root->right, k-dl-2);

// Add 1 to the distance and return value for parent calls

return 1 + dl;

}

int dr = printkdistanceNode(root->right, target, k);

if (dr != -1)

{

if (dr + 1 == k)

cout << root->data << endl;

else

printkdistanceNodeDown(root->left, k-dr-2);

return 1 + dr;

}

return -1;

}

* Print cousins of a given node in Binary Tree :
  + <https://www.geeksforgeeks.org/print-cousins-of-a-given-node-in-binary-tree/>

The idea to first find level of given node using the approach discussed here. Once we have found level, we can print all nodes at a given level using the approach discussed here. The only thing to take care of is, sibling should not be printed. To handle this, we change the printing function to first check for sibling and print node only if it is not sibling.

void printGivenLevel(Node\* root, Node \*node, int level)

{

if (root == NULL || level < 2)

return;

if (level == 2) // 2 is here because level number starts at 1

{

if (root->left == node || root->right == node)

return;

if (root->left)

cout << root->left->data << " ";

if (root->right)

cout << root->right->data;

}

else if (level > 2)

{

printGivenLevel(root->left, node, level - 1);

printGivenLevel(root->right, node, level - 1);

}

}

void printCousins(Node \*root, Node \*node)

{

int level = getLevel(root, node, 1);

printGivenLevel(root, node, level);

}

* + Single Traversal (BFS) :-

<https://www.geeksforgeeks.org/print-cousins-of-a-given-node-in-binary-tree-single-traversal/>  
The idea is to go for level order traversal of the tree, as the cousins and siblings of a node can be found in its level order traversal. Run the traversal till the level containing the node is not found, and if found, print the given level.

void printCousins(Node\* root, Node\* node\_to\_find)

{

if (root == node\_to\_find) {

cout << "Cousin Nodes : None" << endl;

return;

}

queue<Node\*> q;

bool found = false;

int size\_;

Node\* p;

q.push(root);

while (!q.empty() && !found) {

size\_ = q.size();

while (size\_) {

p = q.front();

q.pop();

if ((p->left == node\_to\_find ||

p->right == node\_to\_find)) {

found = true;

}

else {

if (p->left)

q.push(p->left);

if (p->right)

q.push(p->right);

}

size\_--;

}

}

if (found) {

cout << "Cousin Nodes : ";

size\_ = q.size();

// size\_ will be 0 when the node was at the

// level just below the root node.

if (size\_ == 0)

cout << "None";

for (int i = 0; i < size\_; i++) {

p = q.front();

q.pop();

cout << p->data << " ";

}

}

else {

cout << "Node not found";

}

cout << endl;

return;

}

* Print root to leaf paths :-
  + Recursive :-

<https://www.geeksforgeeks.org/given-a-binary-tree-print-out-all-of-its-root-to-leaf-paths-one-per-line/>

* + Iterative (using parent pointer) :-

<https://www.geeksforgeeks.org/print-root-leaf-path-without-using-recursion/>

void printRootToLeaf(Node\* root)

{

if (root == NULL)

return;

stack<Node\*> nodeStack;

nodeStack.push(root);

map<Node\*, Node\*> parent;

parent[root] = NULL;

while (!nodeStack.empty())

{

Node\* current = nodeStack.top();

nodeStack.pop();

if (!(current->left) && !(current->right))

printTopToBottomPath(current, parent);

if (current->right)

{

parent[current->right] = current;

nodeStack.push(current->right);

}

if (current->left)

{

parent[current->left] = current;

nodeStack.push(current->left);

}

}

* **Make a tree with n vertices , d diameter and at most vertex degree k**:-

<https://www.geeksforgeeks.org/make-a-tree-with-n-vertices-d-diameter-and-at-most-vertex-degree-k/>

Approach: Let’s construct the tree with the following algorithm: If (d > n – 1), print “No” and terminate the program. Otherwise, let’s keep the array deg of the length n which will represent degrees of vertices.

The first step is to construct the diameter of the tree. Let the first (d + 1) vertices form it.

Let’s add d edges to the answer and increase degrees of vertices corresponding to these edges, and if some vertex has degree greater than k, print “No” and terminate the program.

The second (and the last) step is to attach the remaining (n – d – 1) vertices to the tree. Let’s call the vertex free if its degree is less than k. Also, let’s keep all free vertices forming the diameter in some data structure which allows us to take the vertex with the minimum maximal distance to any other vertex and remove such vertices. It can be done by, for example, set of pairs (distv, v), where distv is the maximum distance from the vertex v to any other vertex. Now let’s add all the vertices starting from the vertex (d + 1) (0-indexed) to the vertex n?1, let the current vertex be u. One can get the vertex with the minimum maximal distance to any other vertex, let it be v. Now increase the degree of vertices u and v, add the edge between them, and if v still be free, return it to the data structure, otherwise remove it. The same with the vertex u (it is obvious that its maximal distance to any other vertex will be equal to (distv + 1).

If at any step our data structure will be empty or the minimum maximal distance will equal d, the answer is “No”. Otherwise, we can print the answer

* **Possible edges of a tree for given diameter, height and vertices** :-

<https://www.geeksforgeeks.org/print-possible-edges-tree-given-diameter-height-vertices/>

-> Observe that when d = 1, we cannot construct a tree (if tree has more than 2 vertices). Also when d > 2\*h, we cannot construct a tree.

-> As we know that height is the longest path from vertex 1 to another vertex. So build that path from vertex 1 by adding edges up to h. Now, if d > h, we should add another path to satisfy diameter from vertex 1, with a length of d – h.

-> Our conditions for height and diameter are satisfied. But still some vertices may be left. Add the remaining vertices at any vertex other than the end points. This step will not alter our diameter and height. Chose vertex 1 to add the remaining vertices (you can choose any).

-> But when d == h, choose vertex 2 to add the remaining vertices.

* Connect Nodes at same level (towards right):-
  + Using Level order traversal

<https://www.geeksforgeeks.org/connect-nodes-level-level-order-traversal/>

void connect(struct Node\* root)

{

queue<Node\*> q;

q.push(root);

q.push(NULL);

while (!q.empty()) {

Node \*p = q.front();

q.pop();

if (p != NULL) {

p->nextRight = q.front();

if (p->left)

q.push(p->left);

if (p->right)

q.push(p->right);

}

else if (!q.empty())

q.push(NULL);

}

}

* + Extend Preorder traversal (Only for Complete binary tree)  
    <https://www.geeksforgeeks.org/?p=8631/>

In this method we set nextRight in Pre Order fashion to make sure that the nextRight of parent is set before its children. When we are at node p, we set the nextRight of its left and right children. Since the tree is complete tree, nextRight of p’s left child (p->left->nextRight) will always be p’s right child, and nextRight of p’s right child (p->right->nextRight) will always be left child of p’s nextRight (if p is not the rightmost node at its level). If p is the rightmost node, then nextRight of p’s right child will be NULL.

void connect(node \*p)

{

p->nextRight = NULL;

connectRecur(p);

}

Assumption: p is a complete binary tree \*/

void connectRecur(node\* p)

{

if (!p)

return;

if (p->left)

p->left->nextRight = p->right;

if (p->right)

p->right->nextRight = (p->nextRight)?p->nextRight->left: NULL;

connectRecur(p->left);

connectRecur(p->right);

}

* + Modification of above method for every tree :- o(1) space

<https://www.geeksforgeeks.org/connect-nodes-at-same-level-with-o1-extra-space/>

Algorithm:-

void connect (struct node \*p)

{

p->nextRight = NULL;

connectRecur(p);

}

void connectRecur(struct node\* p)

{

if (!p)

return;

if (p->nextRight != NULL)

connectRecur(p->nextRight);

if (p->left)

{

if (p->right)

{

p->left->nextRight = p->right;

p->right->nextRight = getNextRight(p);

}

else

p->left->nextRight = getNextRight(p);

connectRecur(p->left);

}

else if (p->right)

{

p->right->nextRight = getNextRight(p);

connectRecur(p->right);

}

else

**connectRecur(getNextRight(p));**

}

struct node \*getNextRight(struct node \*p)

{

struct node \*temp = p->nextRight;

while(temp != NULL)

{

if(temp->left != NULL)

return temp->left;

if(temp->right != NULL)

return temp->right;

temp = temp->nextRight;

}

return NULL;

}

* + Iterative solution of the above concept O(1) space:-

<https://www.geeksforgeeks.org/connect-nodes-at-same-level-with-o1-extra-space/>

The recursive approach discussed above can be easily converted to iterative. In the iterative version, we use nested loop. The outer loop, goes through all the levels and the inner loop goes through all the nodes at every level. This solution uses constant space.

void connectRecur(node\* p)

{

node \*temp;

if (!p)

return;

p->nextRight = NULL;

while (p != NULL)

{

node \*q = p;

while (q != NULL)

{

if (q->left)

{

if (q->right)

q->left->nextRight = q->right;

else

q->left->nextRight = getNextRight(q);

}

if (q->right)

q->right->nextRight = getNextRight(q);

q = q->nextRight;

}

if (p->left)

p = p->left;

else if (p->right)

p = p->right;

else

p = getNextRight(p);

}

}

* **Sink Odd nodes in Binary Tree** :-

Given a Binary Tree having odd and even elements, sink all its odd valued nodes such that no node with odd value could be parent of node with even value. There can be multiple outputs for a given tree, we need to print one of them

void sink(Node \*&root)

{

if (root == NULL || isLeaf(root))

return;

if (root->left && !(root->left->data & 1))

{

swap(root->data, root->left->data);

sink(root->left);

}

else if(root->right && !(root->right->data & 1))

{

swap(root->data, root->right->data);

sink(root->right);

}

}

void sinkOddNodes(Node\* &root)

{

if (root == NULL || isLeaf(root))

return;

sinkOddNodes(root->left);

sinkOddNodes(root->right);

if (root->data & 1)

sink(root);

}

* To check if Whole of subtreas are same in a question use Tree serialisation/convert tree structure to string/succinct encoding
  + Tree Serialisation (Finding duplicate trees)-

<https://www.geeksforgeeks.org/check-binary-tree-contains-duplicate-subtrees-size-2/>

**s = s + root->key + lStr + rStr;**

Don’t do inorder, as it may give same result for different trees.

Or use this kind of encoding

str.append("|").append(left).append("|");

// append current node data

str.append("|").append(to\_string(root->data)).append("|");

// append right subtree data

str.append("|").append(right).append("|");

<https://www.geeksforgeeks.org/find-largest-subtree-having-identical-left-and-right-subtrees/>

* + Succinct Encoding

<https://www.geeksforgeeks.org/succinct-encoding-of-binary-tree/>

* **Check if 2 binary trees are mirror :-**

bool isMirror(struct Node \*root1, struct Node \*root2)

{

if (root1 == NULL && root2 == NULL)

return true;

if (root1 && root2 && root1->key == root2->key)

return isMirror(root1->left, root2->right) &&

isMirror(root1->right, root2->left);

return false;

}

* **Check mirror in n-ary tree :-**

<https://www.geeksforgeeks.org/check-mirror-n-ary-tree/>

The idea is to use Queue and Stack to check if given N-ary tree are mirror of each other or not.

Let first n-ary tree be t1 and second n-ary tree is t2. For each node in t1, make stack and push its connected node in it. Now, for each node in t2, make queue and push its connected node in it.

Now, for each corresponding node do following:

While stack and Queue is not empty.

a = top element of stack;

b = front of stack;

if (a != b)

return false;

pop element from stack and queue.

bool mirrorUtil(vector<stack<int> >& tree1,

vector<queue<int> >& tree2)

{

for (int i = 1; i < tree1.size(); ++i) {

stack<int>& s = tree1[i];

queue<int>& q = tree2[i];

while (!s.empty() && !q.empty()) {

if (s.top() != q.front())

return false;

s.pop();

q.pop();

}

if (!s.empty() || !q.empty())

return false;

}

return true;

}

void areMirrors(int m, int n, int u1[], int v1[],

int u2[], int v2[])

{

vector<stack<int> > tree1(m + 1);

vector<queue<int> > tree2(m + 1);

for (int i = 0; i < n; i++)

tree1[u1[i]].push(v1[i]);

for (int i = 0; i < n; i++)

tree2[u2[i]].push(v2[i]);

mirrorUtil(tree1, tree2) ? (cout << "Yes" << endl) :

(cout << "No" << endl);

}

* Expression Tree :-

<https://www.geeksforgeeks.org/expression-tree/>

Let t be the expression tree

If t is not null then

If t.value is operand then

Return t.value

A = solve(t.left)

B = solve(t.right)

Return calculate(A, B, t.value)

et\* constructTree(char postfix[])

{

stack<et \*> st;

et \*t, \*t1, \*t2;

for (int i=0; i<strlen(postfix); i++)

{

if (!isOperator(postfix[i]))

{

t = newNode(postfix[i]);

st.push(t);

}

else // operator

{

t = newNode(postfix[i]);

t1 = st.top(); // Store top

st.pop(); // Remove top

t2 = st.top();

st.pop();

**t->right = t1;**

**t->left = t2;**

st.push(t);

}

}

t = st.top();

st.pop();

return t;

}

* **Iterative Postorder** :-
  + Using 2 Stacks :-

<https://www.geeksforgeeks.org/iterative-postorder-traversal/>

Algorithm :-

1. Push root to first stack.

2. Loop while first stack is not empty

2.1 Pop a node from first stack and push it to second stack

2.2 Push left and right children of the popped node to first stack

3. Print contents of second stack

* + Using 1 Stack :-

<https://www.geeksforgeeks.org/iterative-postorder-traversal-using-stack/>

1.1 Create an empty stack

2.1 Do following while root is not NULL

a) Push root's right child and then root to stack.

b) Set root as root's left child.

2.2 Pop an item from stack and set it as root.

a) If the popped item has a right child and the right child

is at top of stack, then remove the right child from stack,

push the root back and set root as root's right child.

b) Else print root's data and set root as NULL.

2.3 Repeat steps 2.1 and 2.2 while stack is not empty.

* + Without Recursion or Stack O(n\*n) :-

void postorder(struct Node\* head)

{

struct Node\* temp = head;

unordered\_set<Node\*> visited;

while (temp && visited.find(temp) == visited.end()) {

if (temp->left &&

visited.find(temp->left) == visited.end())

temp = temp->left;

else if (temp->right &&

visited.find(temp->right) == visited.end())

temp = temp->right;

else {

printf("%d ", temp->data);

visited.insert(temp);

temp = head;

}

}

}

* + Without Recursion or Stack O(n) :-

void postorder(Node\* root)

{

Node\* n = root;

unordered\_map<Node\*, Node\*> parentMap;

parentMap.insert(pair<Node\*, Node\*>(root, nullptr));

while (n) {

if (n->left && parentMap.find(n->left) == parentMap.end()) {

parentMap.insert(pair<Node\*, Node\*>(n->left, n));

n = n->left;

}

else if (n->right && parentMap.find(n->right) == parentMap.end()) {

parentMap.insert(pair<Node\*, Node\*>(n->right, n));

n = n->right;

}

else {

cout << n->data << " ";

n = (parentMap.find(n))->second;

}

}

}

* **Convert a Binary Tree to a Circular Doubly Link List** :-

Node \*concatenate(Node \*leftList, Node \*rightList)

{

if (leftList == NULL)

return rightList;

if (rightList == NULL)

return leftList;

Node \*leftLast = leftList->left;

Node \*rightLast = rightList->left;

leftLast->right = rightList;

rightList->left = leftLast;

leftList->left = rightLast;

rightLast->right = leftList;

return leftList;

}

Node \*bTreeToCList(Node \*root)

{

if (root == NULL)

return NULL;

Node \*left = bTreeToCList(root->left);

Node \*right = bTreeToCList(root->right);

root->left = root->right = root;

return concatenate(concatenate(left, root), right);

}

* **Check if a binary tree is subtree of another binary tree** :-
  + Compare every subtree of first tree with another :-

bool areIdentical(node \* root1, node \*root2)

{

if (root1 == NULL && root2 == NULL)

return true;

if (root1 == NULL || root2 == NULL)

return false;

return (root1->data == root2->data &&

areIdentical(root1->left, root2->left) &&

areIdentical(root1->right, root2->right) );

}

bool isSubtree(node \*T, node \*S)

{

if (S == NULL)

return true;

if (T == NULL)

return false;

if (areIdentical(T, S))

return true;

return isSubtree(T->left, S) || isSubtree(T->right, S);

}

* + O(n) solution :-

1) Find inorder and preorder traversals of T, store them in two auxiliary arrays inT[] and preT[].

2) Find inorder and preorder traversals of S, store them in two auxiliary arrays inS[] and preS[].

3) If inS[] is a subarray of inT[] and preS[] is a subarray preT[], then S is a subtree of T. Else not.

bool isSubtree(Node\* T, Node\* S)

{

if (S == NULL)

return true;

if (T == NULL)

return false;

int m = 0, n = 0;

char inT[MAX], inS[MAX];

storeInorder(T, inT, m);

storeInorder(S, inS, n);

inT[m] = '\0', inS[n] = '\0';

if (strstr(inT, inS) == NULL)

return false;

m = 0, n = 0;

char preT[MAX], preS[MAX];

storePreOrder(T, preT, m);

storePreOrder(S, preS, n);

preT[m] = '\0', preS[n] = '\0';

return (strstr(preT, preS) != NULL);

}

* In Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.

L = T + 1

Where L = Number of leaf nodes

T = Number of internal nodes with two children

* **Handshaking Lemma and Interesting Tree Properties** :-

What is Handshaking Lemma?

Handshaking lemma is about undirected graph. In every finite undirected graph number of vertices with odd degree is always even. The handshaking lemma is a consequence of the degree sum formula (also sometimes called the handshaking lemma)

Sum of degree of nodes is equal to 2\*num edges

1) In a k-ary tree where every node has either 0 or k children, following property is always true.

L = (k - 1)\*I + 1

Where L = Number of leaf nodes

I = Number of internal nodes

* Types of Binary Trees :-
  + degenerate (or pathological) tree

A Tree where every internal node has one child. Such trees are performance-wise same as linked list

* + Balanced Binary Tree

A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes. For Example, AVL tree maintains O(Log n) height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide O(log n) time for search, insert and delete.

* + Full Binary Tree

A Binary Tree is full if every node has 0 or 2 children. Following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.

18

/ \

15 30

/ \ / \

40 50 100 40

* + Perfect Binary Tree

A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level.

Following are examples of Perfect Binary Trees.

18

/ \

15 30

/ \ / \

40 50 100 40

* + Complete Binary Tree

A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

18

/ \

15 30

/ \ / \

40 50 100 40

* **Iteratively check if a tree is symmetric** :-

bool isSymmetric(struct Node\* root)

{

if(root == NULL)

return true;

if(!root->left && !root->right)

return true;

queue <Node\*> q;

q.push(root);

q.push(root);

Node\* leftNode, \*rightNode;

while(!q.empty()){

leftNode = q.front();

q.pop();

rightNode = q.front();

q.pop();

if(leftNode->key != rightNode->key){

return false;

}

if(leftNode->left && rightNode->right){

q.push(leftNode->left);

q.push(rightNode->right);

}

else if (leftNode->left || rightNode->right)

return false;

if(leftNode->right && rightNode->left){

q.push(leftNode->right);

q.push(rightNode->left);

}

else if(leftNode->right || rightNode->left)

return false;

}

return true;

}

* **Serialize and Deserialize a Binary Tree :-**

<https://www.geeksforgeeks.org/serialize-deserialize-binary-tree/>

**If given Tree is Binary Search Tree?**

If the given Binary Tree is Binary Search Tree, we can store it by either storing preorder or postorder traversal. In case of Binary Search Trees, only preorder or postorder traversal is sufficient to store structure information.

**If given Binary Tree is Complete Tree?**

A Binary Tree is complete if all levels are completely filled except possibly the last level and all nodes of last level are as left as possible (Binary Heaps are complete Binary Tree). For a complete Binary Tree, level order traversal is sufficient to store the tree. We know that the first node is root, next two nodes are nodes of next level, next four nodes are nodes of 2nd level and so on.

**If given Binary Tree is Full Tree?**

A full Binary is a Binary Tree where every node has either 0 or 2 children. It is easy to serialize such trees as every internal node has 2 children. We can simply store preorder traversal and store a bit with every node to indicate whether the node is an internal node or a leaf node.

**How to store a general Binary Tree?**

A simple solution is to store both Inorder and Preorder traversals. This solution requires requires space twice the size of Binary Tree.

OR

We can treat the tree as full binary tree and NULL pointers as leaf nodes. We will store -1 for NULL, this will tell if the current node is internal (actual data nodes) or leaf nodes (NULL pointers).

Here only preorder / postorder method will be sufficient.

void serialize(Node \*root, FILE \*fp)

{

if (root == NULL)

{

fprintf(fp, "%d ", MARKER);

return;

}

fprintf(fp, "%d ", root->key);

serialize(root->left, fp);

serialize(root->right, fp);

}

void deSerialize(Node \*&root, FILE \*fp)

{

int val;

if ( !fscanf(fp, "%d ", &val) || val == MARKER)

return;

root = newNode(val);

deSerialize(root->left, fp);

deSerialize(root->right, fp);

}

* **Find uncovered nodes in binary tree** :-

In a binary tree, a node is called Uncovered if it appears either on left boundary or right boundary. Rest of the nodes are called covered

For calculating sum of Uncovered nodes we will follow below steps:

1) Start from root, go to left and keep going until left child is available, if not go to right child and again follow same procedure until you reach a leaf node.

2) After step 1 sum of left boundary will be stored, now for right part again do the same procedure but now keep going to right until right child is available, if not then go to left child and follow same procedure until you reach a leaf node.

int uncoveredSumLeft(Node\* t)

{

if (t->left == NULL && t->right == NULL)

return t->key;

if (t->left != NULL)

return t->key + uncoveredSumLeft(t->left);

else

return t->key + uncoveredSumLeft(t->right);

}

int uncoveredSumRight(Node\* t)

{

if (t->left == NULL && t->right == NULL)

return t->key;

if (t->right != NULL)

return t->key + uncoveredSumRight(t->right);

else

return t->key + uncoveredSumRight(t->left);

}

* **Check if leaf traversal of two Binary Trees is same?** :-

Expected time complexity O(n). Expected auxiliary space O(h1 + h2) where h1 and h2 are heights of two Binary Trees.

public static boolean isSame(Node root1, Node root2)

{

Stack<Node> s1 = new Stack<Node>();

Stack<Node> s2 = new Stack<Node>();

s1.push(root1);

s2.push(root2);

while (!s1.empty() || !s2.empty())

{

if (s1.empty() || s2.empty())

return false;

Node temp1 = s1.pop();

while (temp1!=null && !temp1.isLeaf())

{

if (temp1.right != null)

s1.push(temp1. right);

if (temp1.left != null)

s1.push(temp1.left);

temp1 = s1.pop();

}

Node temp2 = s2.pop();

while (temp2!=null && !temp2.isLeaf())

{

if (temp2.right != null)

s2.push(temp2.right);

if (temp2.left != null)

s2.push(temp2.left);

temp2 = s2.pop();

}

if (temp1==null && temp2!=null)

return false;

if (temp1!=null && temp2==null)

return false;

if (temp1!=null && temp2!=null)

{

if (temp1.data != temp2.data)

return false;

}

}

return true;

}

* **Check whether a given binary tree is perfect or not** :-

/\* This function tests if a binary tree is perfect

or not. It basically checks for two things :

1) All leaves are at same level

2) All internal nodes have two children \*/

bool isPerfectRec(struct Node\* root, int d, int level = 0)

{

if (root == NULL)

return true;

if (root->left == NULL && root->right == NULL)

return (d == level+1);

if (root->left == NULL || root->right == NULL)

return false;

return isPerfectRec(root->left, d, level+1) &&

isPerfectRec(root->right, d, level+1);

}

bool isPerfect(Node \*root)

{

int d = findADepth(root);

return isPerfectRec(root, d);

}

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

int isSibling(struct Node \*root, struct Node \*a, struct Node \*b)

{

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));

}

int level(struct Node \*root, struct Node \*ptr, int lev)

{

if (root == NULL) return 0;

if (root == ptr) return lev;

int l = level(root->left, ptr, lev+1);

if (l != 0) return l;

return level(root->right, ptr, lev+1);

}

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;

}

* **Print root to leaf paths without using recursion** :-

void printTopToBottomPath(Node\* curr, map<Node\*, Node\*> parent)

{

stack<Node\*> stk;

while (curr)

{

stk.push(curr);

curr = parent[curr];

}

while (!stk.empty())

{

curr = stk.top();

stk.pop();

cout << curr->data << " ";

}

cout << endl;

}

void printRootToLeaf(Node\* root)

{

if (root == NULL)

return;

stack<Node\*> nodeStack;

nodeStack.push(root);

map<Node\*, Node\*> parent;

parent[root] = NULL;

while (!nodeStack.empty())

{

Node\* current = nodeStack.top();

nodeStack.pop();

if (!(current->left) && !(current->right))

printTopToBottomPath(current, parent);

if (current->right)

{

parent[current->right] = current;

nodeStack.push(current->right);

}

if (current->left)

{

parent[current->left] = current;

nodeStack.push(current->left);

}

}

}

* **Print cousins of a given node in Binary Tree** :-

void printGivenLevel(Node\* root, Node \*node, int level)

{

if (root == NULL || level < 2)

return;

if (level == 2)

{

if (root->left == node || root->right == node)

return;

if (root->left)

cout << root->left->data << " ";

if (root->right)

cout << root->right->data;

}

else if (level > 2)

{

printGivenLevel(root->left, node, level - 1);

printGivenLevel(root->right, node, level - 1);

}

}

void printCousins(Node \*root, Node \*node)

{

int level = getLevel(root, node, 1);

printGivenLevel(root, node, level);

}

* **Find root of the tree where children id sum for every node is given** :-

Consider a binary tree whose nodes have ids from 1 to n where n is number of nodes in the tree. The tree is given as a collection of n pairs, where every pair represents node id and sum of children ids.

Every node id appears in children sum except root. So if we do sum of all ids and subtract it from sum of all children sums, we get root.

int findRoot(pair<int, int> arr[], int n)

{

int root = 0;

for (int i=0; i<n; i++)

root += (arr[i].first - arr[i].second);

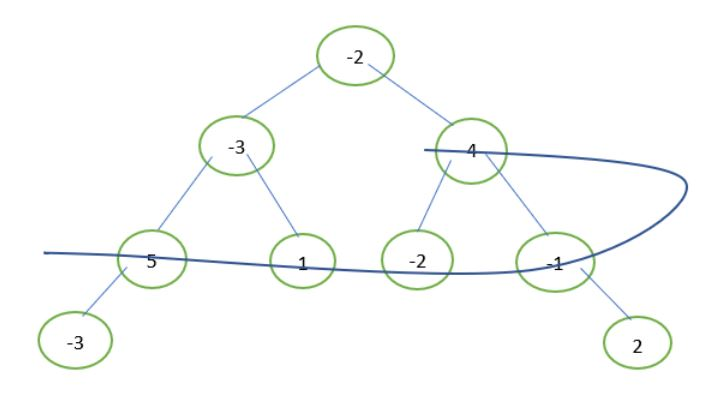
return root;

}

* **Maximum spiral sum in Binary Tree** :-

Given a binary tree containing n nodes. The problem is to find the maximum sum obtained when the tree is spirally traversed. In spiral traversal one by one all levels are being traversed with the root level traversed from right to left, then next level from left to right, then further next level from right to left and so on.

Example:



Maximum spiral sum = 4 + (-1) + (-2) + 1 + 5 = 7

Obtain the level order traversal in spiral form of the given binary tree with the help of two stacks and store it in an array. Find the maximum sum sub-array of the array so obtained.

* **Difference between sums of odd level and even level nodes of a Binary Tree**

Short Recursive solution

int getLevelDiff(node \*root)

{

if (root == NULL)

return 0;

return root->data - getLevelDiff(root->left)- getLevelDiff(root->right);

}

* **Sum of nodes at k-th level in a tree represented as string** :-

Given an integer ‘K’ and a binary tree in string format. Every node of a tree has value in range from 0 to 9. We need to find sum of elements at K-th level from root. The root is at level 0.

Tree is given in the form: (node value(left subtree)(right subtree))

int sumAtKthLevel(string tree, int k)

{

int level = -1;

int sum = 0; // Initialize result

int n = tree.length();

for (int i=0; i<n; i++)

{

if (tree[i] == '(')

level++;

else if (tree[i] == ')')

level--;

else

{

if (level == k)

sum += (tree[i]-'0');

}

}

return sum;

}

* **Print all k-sum paths in a binary tree** :-

A binary tree and a number k are given. Print every path in the tree with sum of the nodes in the path as k.

A path can start from any node and end at any node and must be downward only, i.e. they need not be root node and leaf node; and negative numbers can also be there in the tree.

void printKPathUtil(Node \*root, vector<int>& path, int k)

{

if (!root)

return;

path.push\_back(root->data);

printKPathUtil(root->left, path, k);

printKPathUtil(root->right, path, k);

int f = 0;

for (int j=path.size()-1; j>=0; j--)

{

f += path[j];

if (f == k)

printVector(path, j);

}

path.pop\_back();

}

void printKPath(Node \*root, int k)

{

vector<int> path;

printKPathUtil(root, path, k);

}

* **Maximum sum of nodes in Binary tree such that no two are adjacent** :-

Method 1

We can solve this problem by considering the fact that both node and its children can’t be in sum at same time, so when we take a node into our sum we will call recursively for its grandchildren or when we don’t take this node we will call for all its children nodes and finally we will choose maximum from both of these results.

It can be seen easily that above approach can lead to solving same subproblem many times, for example in above diagram node 1 calls node 4 and 5 when its value is chosen and node 3 also calls them when its value is not chosen so these nodes are processed more than once. We can stop solving these nodes more than once by memoizing the result at all nodes.

In below code a map is used for memoizing the result which stores result of complete subtree rooted at a node in the map, so that if it is called again, the value is not calculated again instead stored value from map is returned directly.

int sumOfGrandChildren(node\* node, map<struct node\*, int>& mp)

{

int sum = 0;

if (node->left)

sum += getMaxSumUtil(node->left->left, mp) +

getMaxSumUtil(node->left->right, mp);

if (node->right)

sum += getMaxSumUtil(node->right->left, mp) +

getMaxSumUtil(node->right->right, mp);

return sum;

}

int getMaxSumUtil(node\* node, map<struct node\*, int>& mp)

{

if (node == NULL)

return 0;

if (mp.find(node) != mp.end())

return mp[node];

int incl = node->data + sumOfGrandChildren(node, mp);

int excl = getMaxSumUtil(node->left, mp) +

getMaxSumUtil(node->right, mp);

mp[node] = max(incl, excl);

return mp[node];

}

int getMaxSum(node\* node)

{

if (node == NULL)

return 0;

map<struct node\*, int> mp;

return getMaxSumUtil(node, mp);

}

Method 2 (Using pair)

Return a pair for each node in the binary tree such that first of the pair indicates maximum sum when the data of node is included and second indicates maximum sum when the data of a particular node is not included.

pair<int, int> maxSumHelper(Node \*root)

{

if (root==NULL)

{

pair<int, int> sum(0, 0);

return sum;

}

pair<int, int> sum1 = maxSumHelper(root->left);

pair<int, int> sum2 = maxSumHelper(root->right);

pair<int, int> sum;

sum.first = sum1.second + sum2.second + root->data;

sum.second = max(sum1.first, sum1.second) +

max(sum2.first, sum2.second);

return sum;

}

int maxSum(Node \*root)

{

pair<int, int> res = maxSumHelper(root);

return max(res.first, res.second);

}

* **Maximum sum from a tree with adjacent levels not allowed** :-

int getSumAlternate(Node\* root)

{

if (root == NULL)

return 0;

int sum = root->data;

if (root->left != NULL)

{

sum += getSum(root->left->left);

sum += getSum(root->left->right);

}

if (root->right != NULL)

{

sum += getSum(root->right->left);

sum += getSum(root->right->right);

}

return sum;

}

int getSum(Node\* root)

{

if (root == NULL)

return 0;

return max(getSumAlternate(root),

(getSumAlternate(root->left) +

getSumAlternate(root->right)));

}

* **Tree Isomorphism Problem** :-

Write a function to detect if two trees are isomorphic. Two trees are called isomorphic if one of them can be obtained from other by a series of flips, i.e. by swapping left and right children of a number of nodes. Any number of nodes at any level can have their children swapped. Two empty trees are isomorphic.

We simultaneously traverse both trees. Let the current internal nodes of two trees being traversed be n1 and n2 respectively. There are following two conditions for subtrees rooted with n1 and n2 to be isomorphic.

1) Data of n1 and n2 is same.

2) One of the following two is true for children of n1 and n2

……a) Left child of n1 is isomorphic to left child of n2 and right child of n1 is isomorphic to right child of n2.

……b) Left child of n1 is isomorphic to right child of n2 and right child of n1 is isomorphic to left child of n2

bool isIsomorphic(Node \*root1,Node \*root2)

{

if(root1 == NULL && root2 == NULL)

return true;

if(root1 && root2)

return root1->data == root2->data && ((isIsomorphic(root1->left, root2->left) && isIsomorphic(root1->right, root2->right)) || (isIsomorphic(root1->left, root2->right) && isIsomorphic(root1->right, root2->left)));

return false;

}

* **Find Height of Binary Tree represented by Parent array** :-

An efficient solution can solve the above problem in O(n) time. The idea is to first calculate depth of every node and store in an array depth[]. Once we have depths of all nodes, we return maximum of all depths.

1) Find depth of all nodes and fill in an auxiliary array depth[].

2) Return maximum value in depth[].

Following are steps to find depth of a node at index i.

1) If it is root, depth[i] is 1.

2) If depth of parent[i] is evaluated, depth[i] is depth[parent[i]] + 1.

3) If depth of parent[i] is not evaluated, recur for parent and assign depth[i] as depth[parent[i]] + 1 (same as above).

void fillDepth(int parent[], int i, int depth[])

{

if (depth[i])

return;

if (parent[i] == -1)

{

depth[i] = 1;

return;

}

if (depth[parent[i]] == 0)

fillDepth(parent, parent[i], depth);

depth[i] = depth[parent[i]] + 1;

}

int findHeight(int parent[], int n)

{

int depth[n];

for (int i = 0; i < n; i++)

depth[i] = 0;

for (int i = 0; i < n; i++)

fillDepth(parent, i, depth);

int ht = depth[0];

for (int i=1; i<n; i++)

if (ht < depth[i])

ht = depth[i];

return ht;

}

* **Find height of a special binary tree whose leaf nodes are connected** :-

Given a special binary tree whose leaf nodes are connected to form a circular doubly linked list.

bool isLeaf(Node\* node)

{

return node->left && node->left->right == node &&

node->right && node->right->left == node;

}

int maxDepth(Node\* node)

{

if (node == NULL)

return 0;

if (isLeaf(node))

return 1;

return 1 + max(maxDepth(node->left), maxDepth(node->right));

}

* **Find mirror of a given node in Binary tree** :-

int findMirrorRec(int target, struct Node\* left, struct Node\* right)

{

if (left==NULL || right==NULL)

return 0;

if (left->key == target)

return right->key;

if (right->key == target)

return left->key;

int mirror\_val = findMirrorRec(target, left->left,right->right);

if (mirror\_val)

return mirror\_val;

findMirrorRec(target, left->right, right->left);

}

int findMirror(struct Node\* root, int target)

{

if (root == NULL)

return 0;

if (root->key == target)

return target;

return findMirrorRec(target, root->left, root->right);

}

* **Closest leaf to a given node in Binary Tree** :-

Given a Binary Tree and a node x in it, find distance of the closest leaf to x in Binary Tree. If given node itself is a leaf, then distance is 0.

void findLeafDown(Node \*root, int lev, int \*minDist)

{

if (root == NULL)

return ;

if (root->left == NULL && root->right == NULL)

{

if (lev < (\*minDist))

\*minDist = lev;

return;

}

findLeafDown(root->left, lev+1, minDist);

findLeafDown(root->right, lev+1, minDist);

}

int findThroughParent(Node \* root, Node \*x, int \*minDist)

{

if (root == NULL) return -1;

if (root == x) return 0;

int l = findThroughParent(root->left, x, minDist);

if (l != -1)

{

findLeafDown(root->right, l+2, minDist);

return l+1;

}

int r = findThroughParent(root->right, x, minDist);

if (r != -1)

{

findLeafDown(root->left, r+2, minDist);

return r+1;

}

return -1;

}

int minimumDistance(Node \*root, Node \*x)

{

int minDist = INT\_MAX;

findLeafDown(x, 0, &minDist);

findThroughParent(root, x, &minDist);

return minDist;

}

Another method :-

int closestDown(struct Node \*root)

{

if (root == NULL)

return INT\_MAX;

if (root->left == NULL && root->right == NULL)

return 0;

return 1 + getMin(closestDown(root->left), closestDown(root->right));

}

int findClosestUtil(struct Node \*root, char k, struct Node \*ancestors[],

int index)

{

if (root == NULL)

return INT\_MAX;

if (root->key == k)

{

int res = closestDown(root);

for (int i = index-1; i>=0; i--)

res = getMin(res, index - i + closestDown(ancestors[i]));

return res;

}

ancestors[index] = root;

return getMin(findClosestUtil(root->left, k, ancestors, index+1),

findClosestUtil(root->right, k, ancestors, index+1));

}

int findClosest(struct Node \*root, char k)

{

struct Node \*ancestors[100];

return findClosestUtil(root, k, ancestors, 0);

}

* **Maximum Consecutive Increasing Path Length in Binary Tree** :-

Given a Binary Tree find the length of the longest path which comprises of nodes with consecutive values in increasing order. Every node is considered as a path of length 1.

Examples:

10

/ \

/ \

11 9

/ \ /\

/ \ / \

13 12 13 8

Maximum Consecutive Path Length is 3 (10, 11, 12)

Note: 10, 9 ,8 is NOT considered since

the nodes should be in increasing order.

int maxPathLenUtil(Node \*root, int prev\_val, int prev\_len)

{

if (!root)

return prev\_len;

int cur\_val = root->val;

if (cur\_val == prev\_val+1)

{

return max(maxPathLenUtil(root->left, cur\_val, prev\_len+1),

maxPathLenUtil(root->right, cur\_val, prev\_len+1));

}

int newPathLen = max(maxPathLenUtil(root->left, cur\_val, 1),

maxPathLenUtil(root->right, cur\_val, 1));

return max(prev\_len, newPathLen);

}

int maxConsecutivePathLength(Node \*root)

{

if (root == NULL)

return 0;

return maxPathLenUtil(root, root->val-1, 0);

}

**Note :- This recursion technique can be used in many questions**

* **Remove nodes on root to leaf paths of length < K :-**

Node \*removeShortPathNodesUtil(Node \*root, int level, int k)

{

if (root == NULL)

return NULL;

root->left = removeShortPathNodesUtil(root->left, level + 1, k);

root->right = removeShortPathNodesUtil(root->right, level + 1, k);

if (root->left == NULL && root->right == NULL && level < k)

{

delete root;

return NULL;

}

return root;

}

* **Number of turns to reach from one node to other in binary tree** :-

Given a binary tree and two nodes. The task is to count the number of turns needs to reach from one node to another node of the Binary tree.

Note :- here turn variable denotes whether the root is left child or right

bool CountTurn(Node\* root, int key, bool turn, int\* count)

{

if (root == NULL)

return false;

if (root->key == key)

return true;

if (turn == true) {

if (CountTurn(root->left, key, turn, count))

return true;

if (CountTurn(root->right, key, !turn, count)) {

\*count += 1;

return true;

}

}

else // Case 2:

{

if (CountTurn(root->right, key, turn, count))

return true;

if (CountTurn(root->left, key, !turn, count)) {

\*count += 1;

return true;

}

}

return false;

}

int NumberOFTurn(struct Node\* root, int first, int second)

{

struct Node\* LCA = findLCA(root, first, second);

if (LCA == NULL)

return -1;

int Count = 0;

if (LCA->key != first && LCA->key != second) {

if (CountTurn(LCA->right, second, false, &Count)

|| CountTurn(LCA->left, second, true,&Count)) ;

if (CountTurn(LCA->left, first, true,&Count)

|| CountTurn(LCA->right, first, false, &Count)) ;

return Count + 1;

}

if (LCA->key == first) {

CountTurn(LCA->right, second, false, &Count);

CountTurn(LCA->left, second, true, &Count);

return Count;

} else {

CountTurn(LCA->right, first, false, &Count);

CountTurn(LCA->left, first, true, &Count);

return Count;

}

}

* **Find first non matching leaves in two binary trees** :-

This solution auxiliary space requirement as O(h1 + h2) where h1 and h2 are heights of trees. We do Iterative Preorder traversal of both the trees simultaneously using stacks. We maintain a stack for each tree. For every tree, keep pushing nodes in the stack till the top node is a leaf node. Compare the two top nodes f both the stack. If they are equal, do further traversal else return.

void findFirstUnmatch(Node \*root1, Node \*root2)

{

if (root1 == NULL || root2 == NULL)

return;

stack<Node\*> s1, s2;

s1.push(root1);

s2.push(root2);

while (!s1.empty() || !s2.empty())

{

if (s1.empty() || s2.empty() )

return;

Node \*temp1 = s1.top();

s1.pop();

while (temp1 && !isLeaf(temp1))

{

s1.push(temp1->right);

s1.push(temp1->left);

temp1 = s1.top();

s1.pop();

}

Node \* temp2 = s2.top();

s2.pop();

while (temp2 && !isLeaf(temp2))

{

s2.push(temp2->right);

s2.push(temp2->left);

temp2 = s2.top();

s2.pop();

}

if (temp1 != NULL && temp2 != NULL )

{

if (temp1->data != temp2->data )

{

cout << "First non matching leaves : "

<< temp1->data <<" "<< temp2->data

<< endl;

return;

}

}

}

}