**Theory Portion:**

**Different Types Of Tree:**

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

**A segment tree is a good example full binary tree.**

**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

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

**A degenerate (or pathological) tree:** A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

**(for finite undirected graph or undirected acyclic graph)**

Handshaking lemma is about undirected graph. In every finite undirected graph number of vertices with odd degree is always even.

(the number of vertices with odd degree is always even)

There is also another lemma saying that the total number of degree (including in degree and odd degree) is always even.

These are lemmas.

How are these lemmas useful?

**Prove that:** 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

Case 1 (Root is Leaf):There is only one node in tree. The above formula is true for single node as L = 1, I = 0.

Case 2 (Root is Internal Node): For trees with more than 1 nodes, root is always internal node. The above formula can be proved using Handshaking Lemma for this case. A tree is an undirected acyclic graph.

Total number of edges in Tree is number of nodes minus 1, i.e., |E| = L + I – 1.

All internal nodes except root in the given type of tree have degree k + 1. Root has degree k. All leaves have degree 1. Applying the Handshaking lemma to such trees, we get following relation.

Sum of all degrees = 2 \* (Sum of Edges)

Sum of degrees of leaves +

Sum of degrees for Internal Node except root +

Root's degree = 2 \* (No. of nodes - 1)

Putting values of above terms,

L + (I-1)\*(k+1) + k = 2 \* (L + I - 1)

L + k\*I - k + I -1 + k = 2\*L + 2I - 2

L + K\*I + I - 1 = 2\*L + 2\*I - 2

K\*I + 1 - I = L

(K-1)\*I + 1 = L (proved)

1. **In Binary tree, number of leaf nodes is always one more than nodes with two children.**

**Now, lemmas or portion of proven facts which we could use:**

**The total number of degrees (including in-degree and out-degree) of a tree is twice of the total number of edges**

**The number of vertices with odd degree (in degree+out degree) is always even.**

**E=L+I-1 (you could draw it to prove it)**

**Now, here, we need to prove L=I+1 (I is the internal node with 2 children)**

**Now, I+J+L-1=E**

**Now, sum of all degrees=2\*E**

**Now, sum of all degrees of leaf node+sum of all degrees of internal node with two children+sum of all degrees of internal node with 1 children=2(I+j+L-1)**

**L+3\*I+2\*j=2\*I+2\*j+2\*L-2**

**I=L-2**

**Now, I did something wrong**

**Case 1: There is only one node, the relationship holds**

**as T = 0, L = 1.**

**Case 2: Root has two children, i.e., degree of root is 2.**

**Sum of degrees of nodes with two children except root +**

**Sum of degrees of nodes with one child +**

**Sum of degrees of leaves + Root's degree = 2 \* (No. of Nodes - 1)**

**Putting values of above terms,**

**(T-1)\*3 + S\*2 + L + 2 = (S + T + L - 1)\*2**

**Cancelling 2S from both sides.**

**(T-1)\*3 + L + 2 = (S + L - 1)\*2**

**T - 1 = L - 2**

**T = L - 1**

**Now, here, I did wrong. Root’s case should be separately handled. Two cases will be based on root. When, root has 1 child and when root has 2 children.**

**Case 3: Root has one child, i.e., degree of root is 1.**

**Sum of degrees of nodes with two children +**

**Sum of degrees of nodes with one child except root +**

**Sum of degrees of leaves + Root's degree = 2 \* (No. of Nodes - 1)**

**Putting values of above terms,**

**T\*3 + (S-1)\*2 + L + 1 = (S + T + L - 1)\*2**

**Cancelling 2S from both sides.**

**3\*T + L -1 = 2\*T + 2\*L - 2**

**T - 1 = L - 2**

**T = L - 1**

**Tree Traversal:**

1. **Depth First Search:**
   1. **Pre order traversal:** root left right
   2. **Post Order Traversal:** left right root
   3. **In order traversal :** left root right

**Note:** I can do this traversals recursively. And there are non recursive linear solution of them too. I will discuss them later

1. **Breadth First Search:**

There are recursive and non recursive solution of this.

**Pre Order Traversal:**

I can easily solve it recursively. So, let’s skip it. And jump to non recursive linear solution.

**Non Recursive Solution Based On Stack:**

1. Create an empty stack nodeStack and push root node to stack.  
   **2)** Do following while nodeStack is not empty.  
   ….**a)** Pop an item from stack and print it.  
   ….**b)** Push right child of popped item to stack  
   ….**c)** Push left child of popped item to stack

Note: note the pushing order. First, right child, then left child.

**Non Recursive Solution WithOut Stack:**

**Now, this is based on Morris traversal. (if first creates a threaded binary tree, finally restores it)**

1. **Initialize current node To root.**
2. **Iterate until current node is not NULL.**
3. **If current node’s** left child is null, print the current node data. Move to right child.  
   (That is easy part)  
     
   4) **Else**, Make the right child of the inorder predecessor point to the current node.

Or, in more simpler language, traverse upto the inorder predecessor of the current node. It is in general the right most node of left subtree. Then, check it’s right child.

Two cases arise:  
………**a)** The right child of the inorder predecessor already points to the current node. Set right child to NULL. Move to right child of current node.  
………**b)** The right child is NULL. Set it to current node. Print current node’s data and move to left child of current node.

**Post order Traversal:**

I can easily solve it recursively. So, let’s discuss the non recursive Solutions.

**Non Recursive Solution Based On 2 Stacks:**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

**Non Recursive Solution Based On 1 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.

**(Notice that: 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.** (why ? because, that indicates the current node’s right child is yet to be traversed. And since, it’s left right root in postorder traversal, we first have to traverse the right child)

**Non Recursive Solution Without Stack:**

There is not any. (No solution based on Morris traversal.

**Inorder Traversal:**

**Non Recursive Solution Based On Stack:**

1) Create an empty stack S.

2) Initialize current node as root

3) Push the current node to S and set current = current->left until current is NULL

4) If current is NULL and stack is not empty then

a) Pop the top item from stack.

b) Print the popped item, set current = popped\_item->right

c) Go to step 3.

5) If current is NULL and stack is empty then we are done.

**Non Recursive Solution Without Stack:  
  
Morris Traversal.**

1. Initialize current as root

2.While current is not NULL

If current is not NULL

1. If current’s left is NULL, print the current node and go to current’s right. I.e. make the current’s right as current node.
2. Else, find the inorder predecessor of current node(the rightmost node of the left subtree).
   1. if right child of Inorder predecessor is NULL, make current node as inorder predecessor’s right node and make current->left as new current
   2. Else, if right child of inorder predecessor is current. We are visiting current node second time. (so, left subtree of current node is already printed ). We will print the current node this time and make current’s right as new current.

**Non Threaded Non Recursive Solution Without Stack:**

Since, Morris traversal is based on threading of binary tree.

But, parent node concept is to be involved.

1. Initialize current node as root

2. Initialize a flag: leftdone = false;

3. Do following while root is not NULL

a) If leftdone is false, set current node as leftmost

child of node.

b) Mark leftdone as true and print current node.

c) If right child of current nodes exists, set current

as right child and set leftdone as false.

d) Else If parent exists, **If current node is left child**

**of its parent, set current node as parent.**

If current node is right child, keep moving to ancestors

using parent pointer while current node is right child

of its parent.

e) Else break (We have reached back to root)

**Breadth First Traversal:(Level Order Traversal)**

**Recursive Solution:**

void printLevelOrder(struct node\* root)

{

int h = height(root);

int i;

for (i=1; i<=h; i++)

{

printGivenLevel(root, i);

printf("\n");

}

}

/\* Print nodes at a given level \*/

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

{

if (root == NULL)

return;

if (level == 1)

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

else if (level > 1)

{

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

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

}

}

**Non Recursive Solution:**

Based on queue. I can do it.

**Special Traversals:**

**Diagonal Traversal**

**Vertical Traversal**

**Reverse Level Order Traversal**

**Boundary Traversals**

**Print Postorder Traversal From Given Inorder and Preorder Traversal:**

**Print Preorder traversal From Given Inorder Traversal and PostOrder Traversal**

**Print Inorder Traversal From Given PreOrder and PostOrder Traversal (Cannot be constructed)**

**Boundary Traversal**

# Perfect Binary Tree Specific Level Order Traversal

**binary_tree-1**

Print the nodes in following manner:

**1 2 3 4 7 5 6 8 15 9 14 10 13 11 12 16 31 17 30 18 29 19 28 20 27 21 26 22 25 23 24**

The standard level order traversal idea will slightly change here. Instead of processing ONE node at a time, we will process TWO nodes at a time. And while pushing children into queue, the enqueue order will be: 1st node’s left child, 2nd node’s right child, 1st node’s right child and 2nd node’s left child.

**Perfect Binary Tree Specific Level order Traversal 2**

**binary_tree-1**

**16 31 17 30 18 29 19 28 20 27 21 26 22 25 23 24 8 15 9 14 10 13 11 12 4 7 5 6 2 3 1**

Now, this is exactly same as the previous problem. Except, here we need to print from bottom to top.

The standard level order traversal idea slightly changes here.

Instead of processing ONE node at a time, we will process TWO nodes at a time.

For dequeued nodes, we push node’s left and right child into stack in following manner – 2nd node’s left child, 1st node’s right child, 2nd node’s right child and 1st node’s left child.

And while pushing children into queue, the enqueue order will be: 1st node’s right child, 2nd node’s left child, 1st node’s left child and 2nd node’s right child. Also, when we process two queue nodes.

Finally pop all Nodes from stack and prints them.

**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.

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.

**Tree Construction And Conversion:**

**Convert A Normal BST To Balanced BST**A Simple Solution is to traverse nodes in Inorder and one by one insert into a self-balancing BST like AVL tree. Time complexity of this solution is O(n Log n) and this solution doesn’t guarantee

An Efficient Solution can construct balanced BST in O(n) time with minimum possible height. Below are steps.

Traverse given BST in inorder and store result in an array. This step takes O(n) time. Note that this array would be sorted as inorder traversal of BST always produces sorted sequence.

Build a balanced BST from the above created sorted array using the recursive approach discussed here. This step also takes O(n) time as we traverse every element exactly once and processing an element takes O(1) time.

Node\* buildTreeUtil(vector<Node\*> &nodes, int start,

int end)

{

// base case

if (start > end)

return NULL;

/\* Get the middle element and make it root \*/

int mid = (start + end)/2;

Node \*root = nodes[mid];

/\* Using index in Inorder traversal, construct

left and right subtress \*/

root->left = buildTreeUtil(nodes, start, mid-1);

root->right = buildTreeUtil(nodes, mid+1, end);

return root;

}

**Change a Binary Tree so that every node stores sum of all nodes in left subtree**

The idea is to traverse the given tree in bottom up manner. For every node, recursively compute sum of nodes in left and right subtrees. Add sum of nodes in left subtree to current node and return sum of nodes under current subtree.

**Write An Efficient Program To Convert A Tree To It’s Mirror Tree:**

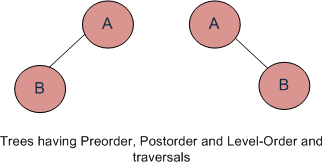
I can do that.

For every node first, recursively call make\_mirror for both it’s left child and right child.

After that, swap these left child and right child using temp node.

**Construct Binary Tree From PreOrder and PostOrder Traversal:**

**Now, It is not possible to construct a binary tree from PreOrder and PostOrder** traversal unless the tree is complete and full. Why?



Now, consider the first tree. It it neither complete nor full. And, think about pre order traversal and post order traversal.

Preorder Traversal = AB  
Postorder Traversal = BA

Now, if you consider second tree, it also has the same pre order and post order traversal.

Preorder Traversal = AB  
Postorder Traversal = BA

So, ambiguity could be there.

However, if the tree is complete and full, then we can construct.

Even if a tree is just full (A full binary tree: where all nodes have either 0 or 2 two children)

**Now, let’s focus on the problem:**

Let us consider the two given arrays as pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7} and post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1};

In pre[], the leftmost element is root of tree. Since the tree is full and array size is more than 1. The value next to 1 in pre[], must be left child of root. So we know 1 is root and 2 is left child. How to find the all nodes in left subtree? We know 2 is root of all nodes in left subtree. All nodes before 2 in post[] must be in left subtree. Now we know 1 is root, elements {8, 9, 4, 5, 2} are in left subtree, and the elements {6, 7, 3} are in right subtree.

**Construct A Tree From Inorder And Level Order Traversals:**

**Construct A Complete Binary Tree From The Level Order Traversal Represented By LinkedList:**

I can do that. It’s almost the same procedure we used for level order traversing a tree using queue.

**Construct a special tree from given preorder traversal:**

Given an array ‘pre[]’ that represents Preorder traversal of a spacial binary tree where every node has either 0 or 2 children. One more array ‘preLN[]’ is given which has only two possible values ‘L’ and ‘N’. The value ‘L’ in ‘preLN[]’ indicates that the corresponding node in Binary Tree is a leaf node and value ‘N’ indicates that the corresponding node is non-leaf node. Write a function to construct the tree from the given two arrays.

**Example:**

Input: pre[] = {10, 30, 20, 5, 15}, preLN[] = {'N', 'N', 'L', 'L', 'L'}

Output: Root of following tree

10

/ \

30 15

/ \

20 5

The first element in pre[] will always be root. So we can easily figure out root. If left subtree is empty, the right subtree must also be empty and preLN[] entry for root must be ‘L’. We can simply create a node and return it. If left and right subtrees are not empty, then recursively call for left and right subtrees and link the returned nodes to root.

**Construct Binary Tree From It’s Ancestor Tree:**

I can do it.

A multimap is required and a parent array indicating that whether parent for this node is found or not.

multimap<int, int> mm;

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

{

int sum = 0; // Initialize sum of this row

for (int j = 0; j < N; j++)

sum += mat[i][j];

// insert(sum, i) pairs into the multimap

mm.insert(pair<int, int>(sum, i));

}

//key part of every key,value pair represents the the children count

And value part the index of the current node in ancestor matrix

// node[i] will store node for i in constructed tree

Node\* node[N];

//this will contain node for ith node.

// Traverse all entries of multimap. Note that values

// are accessed in increasing order of sum

for (auto it = mm.begin(); it != mm.end(); ++it)

{

// create a new node for every value

node[it->second] = newNode(it->second);

// To store last processed node. This node will be

// root after loop terminates

root = node[it->second];

// if non-leaf node

//because, in case of leaf node, nothing much to do

if (it->first != 0)

{

// traverse row 'it->second' in the matrix

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

{

// if parent is not set and ancestor exits

if (!parent[i] && mat[it->second][i])

{

// check for unoccupied left/right node

// and set parent of node i

if (!node[it->second]->left)

node[it->second]->left = node[i];

else

node[it->second]->right = node[i];

parent[i] = 1;

}

}

}

}

return root;

**Construct Binary Tree From Given Parent Array Representation:**

Given an array that represents a tree in such a way that array indexes are values in tree nodes and array values give the parent node of that particular index (or node). The value of the root node index would always be -1 as there is no parent for root. Construct the standard linked representation of given Binary Tree from this given representation.

**Efficient Solution:**

**createTree(parent[], n)**

Create an array of pointers say created[0..n-1]. The value of created[i] is NULL if node for index i is not created, else value is pointer to the created node.

Do following for every index i of given array

createNode(parent, i, created)

createNode(parent[], i, crated[])

If created[i] is not NULL, then node is already created. So return.

Create a new node with value ‘i’.

If parent[i] is -1 (i is root), make created node as root and return.

Check if parent of ‘i’ is created (We can check this by checking if created[parent[i]] is NULL or not.

If parent is not created, recur for parent and create the parent first.

Let the pointer to parent be p. If p->left is NULL, then make the new node as left child. Else make the new node as right child of parent.  
  
**Construct Ancestor Matrix From A Given Binary Tree:**

I can do that.

**Creating a tree with Left-Child Right-Sibling Representation**

Left-Child Right-Sibling Representation is a different representation of an n-ary tree where instead of holding a reference to each and every child node, a node holds just two references, first a reference to it’s first child, and the other to it’s immediate next sibling. This new transformation not only removes the need of advance knowledge of the number of children a node has, but also limits the number of references to a maximum of two, thereby making it so much easier to code.

// CPP program to create a tree with left child

// right sibling representation.

#include<bits/stdc++.h>

using namespace std;

struct Node

{

int data;

struct Node \*next;

struct Node \*child;

};

// Creating new Node

Node\* newNode(int data)

{

Node \*newNode = new Node;

newNode->next = newNode->child = NULL;

newNode->data = data;

return newNode;

}

// Adds a sibling to a list with starting with n

Node \*addSibling(Node \*n, int data)

{

if (n == NULL)

return NULL;

while (n->next)

n = n->next;

return (n->next = newNode(data));

}

// Add child Node to a Node

Node \*addChild(Node \* n, int data)

{

if (n == NULL)

return NULL;

// Check if child list is not empty.

if (n->child)

return addSibling(n->child, data);

else

return (n->child = newNode(data));

}

// Traverses tree in level order

void traverseTree(Node \* root)

{

if (root == NULL)

return;

while (root)

{

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

if (root->child)

traverseTree(root->child);

root = root->next;

}

}

//Driver code

int main()

{

/\* Let us create below tree

\* 10

\* / / \ \

\* 2 3 4 5

\* | / | \

\* 6 7 8 9 \*/

// Left child right sibling

/\* 10

\* |

\* 2 -> 3 -> 4 -> 5

\* | |

\* 6 7 -> 8 -> 9 \*/

Node \*root = newNode(10);

Node \*n1 = addChild(root, 2);

Node \*n2 = addChild(root, 3);

Node \*n3 = addChild(root, 4);

Node \*n4 = addChild(n3, 6);

Node \*n5 = addChild(root, 5);

Node \*n6 = addChild(n5, 7);

Node \*n7 = addChild(n5, 8);

Node \*n8 = addChild(n5, 9);

traverseTree(root);

return 0;

}

**The traversal we are doing here, is depth first. As, we are traversing child first, sibling later.**

**Convert A Binary Tree To Doubly Linked List:**

**Approach 1:**

1. If left subtree exists, process the left subtree

…..1.a) Recursively convert the left subtree to DLL.

…..1.b) Then find inorder predecessor of root in left subtree (inorder predecessor is rightmost node in left subtree).

…..1.c) Make inorder predecessor as previous of root and root as next of inorder predecessor.

2. If right subtree exists, process the right subtree (Below 3 steps are similar to left subtree).

…..2.a) Recursively convert the right subtree to DLL.

…..2.b) Then find inorder successor of root in right subtree (inorder successor is leftmost node in right subtree).

…..2.c) Make inorder successor as next of root and root as previous of inorder successor.

3. Find the leftmost node and return it (the leftmost node is always head of converted DLL).

**Approach 2:**

1) Fix Left Pointers: In this step, we change left pointers to point to previous nodes in DLL. The idea is simple, we do inorder traversal of tree. In inorder traversal, we keep track of previous visited node and change left pointer to the previous node. See fixPrevPtr() in below implementation.

1. Fix Right Pointers: The above is intuitive and simple. How to change right pointers to point to next node in DLL? The idea is to use left pointers fixed in step 1. We start from the rightmost node in Binary Tree (BT). The rightmost node is the last node in DLL. Since left pointers are changed to point to previous node in DLL, we can linearly traverse the complete DLL using these pointers. The traversal would be from last to first node. While traversing the DLL, we keep track of the previously visited node and change the right pointer to the previous node. See fixNextPtr() in below implementation.

**Approach 3:**

The idea is to do inorder traversal of the binary tree. While doing inorder traversal, keep track of the previously visited node in a variable say prev. For every visited node, make it next of prev and previous of this node as prev.

**Approach 4:**

we traverse the tree in inorder fashion. We add nodes at the beginning of current linked list and update head of the list using pointer to head pointer. Since we insert at the beginning, we need to process leaves in reverse order. For reverse order, we first traverse the right subtree before the left subtree. i.e. do a reverse inorder traversal.

**Convert A Binary Tree To Circular Linked List:**

I can do that.

**Convert A Binary Tree To A Tree That Holds The Property Of A Binary Tree:**

I can do that.

**Longest Common Ancestor**

**Find Out Longest Common Ancestor Of Two Nodes Using Vector:**

Find out paths for from root to each of the nodes and store them in two different vectors. Compare the vectors and generate it.

**Find Out Longest Common Ancestors Of Two Nodes: (Without Vector)**

int find\_LCA\_util(struct tree\_node \*root,int node\_val1,int node\_val2)

{

if(root==NULL)

{

return INT\_MIN;

}

if(root->data==node\_val1||root->val==node\_val2)

{

return root->data;

}

int left\_val=find\_LCA\_util(root->left,node\_val1,node\_val2);

int right\_val=find\_LCA\_util(root->right,node\_val1,node\_val2);

if(left\_val!=INT\_MIN&&right\_val!=INT\_MIN)

{

//both tree\_contains one node each

//hence,both left ree and right contains one of them

return root->data;

}

//this assumes the presence of both

if(left\_val==INT\_MIN)

{

return right\_val;

}

if(right\_val==INT\_MIN)

{

return left\_val;

}

}

**Find Distance Between Two Nodes In A Binary Tree:**

This can be done based on LCA.

**Print Common Nodes On Path From Root:**

This can be done based on LCA.

**Misc:**

**How To Delete A Tree?**

**Find Maximum Height Or Depth Of A Tree**

**Write A Program To Calculate the Size Of The Tree**

**Root To Leaf Path Sum Equal To A Given Number**

**How To Determine If A Tree Is Height Balanced**

**Check If A Binary Tree Holds Children Sum Property**

I can do that. Recursive Solution

**Diameter Of A Binary Tree:**

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between any two end nodes.

The diameter of a tree T is the largest of the following quantities:

\* the diameter of T’s left subtree

\* the diameter of T’s right subtree

\* the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

**Top View Of A Binary Tree:**

**Bottom View Of A Binary Tree:**

**Right View Of A Binary Tree:**

**Left View Of A Binary Tree:**

**(**all these are basically application of level order traversal of a tree. However, the top view and bottom view require additional concept of horizontal distance or vertical distance)

**Connect Nodes Of Same Level:**

**Find A Key In Binary Tree: (Iterative)**

**Find Next Right Node Of A Binary Tree:**

**Find If Two Nodes Are Cousins Or Not:**

**Iterative Way To Find The Height Of A Binary Tree:**

**Find If The Binary Tree Is A Perfect Binary Tree Or Not:**

**Get Level Of A Node In The Binary Tree:**

# Print nodes between two given level numbers of a binary tree

Again, all the above mentioned problems are variations of Level order traversals using queue.

**Find Minimum Depth Of A Binary Tree:**

**Find Closest Leaf Of A Binary Tree:**

**Count Leaf nodes Of A Binary Tree:**

**Find depth of the deepest odd level leaf node**

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

**Find Maximum Or Minimum Of A Binary Tree  
  
Find Sum Of All Left Leaf Nodes Of A Binary Tree**

**Check If All Leaf Nodes Are Of Same Level**

**Check if a given Binary Tree is SumTree**

**Find Maximum Sum Root To Leaf Path Of A Binary Tree:**

**Diagonal Sum Of A Binary Tree:**

**Vertical Sum Of A Binary Tree:**

**Deepest Left Leaf Node Of A Binary Tree:**

I can do these problems. These problems are easy.

**Given A Binary Tree, Print All Root To Leaf Node Path**

(use vector. Recursive Solution)

**Double Tree:**

Write a program that converts a given tree to its Double tree. To create Double tree of the given tree, create a new duplicate for each node, and insert the duplicate as the left child of the original node.

So the tree…

2

/ \

1 3

is changed to…

2

/ \

2 3

/ /

1 3

/

1

And the tree

1

/ \

2 3

/ \

4 5

is changed to

1

/ \

1 3

/ /

2 3

/ \

2 5

/ /

4 5

/

4

**Algorithm:**

Recursively convert the tree to double tree in postorder fashion. For each node, first convert the left subtree of the node, then right subtree, finally create a duplicate node of the node and fix the left child of the node and left child of left child.

**Extract Leave Nodes Of A Binary Tree Into DLL:**

Now, I can do it. But careful, postorder or preorder traversal wont work here. As a wrong node can be converted into leaf node. Only In order traversal will work.

**Check if a given Binary Tree is height balanced like a Red-Black Tree:**

In a Red-Black Tree, the maximum height of a node is at most twice the minimum height (The four Red-Black tree properties make sure this is always followed). Given a Binary Search Tree, we need to check for following property.

For every node, length of the longest leaf to node path has not more than twice the nodes on shortest path from node to leaf.

So, basically check if min\_height<=2\*max\_height

**Find Distance B/W Two Keys Of A Binary Tree:**This is application of LCA.

**Check If A Binary Tree Is Subtree of Another Binary Tree:**

I can do it.

You can break it into two functions like the example code given below:

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

{

/\* base cases \*/

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

return true;

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

return false;

/\* Check if the data of both roots is same and data of left and right

subtrees are also same \*/

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

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

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

}

/\* This function returns true if S is a subtree of T, otherwise false \*/

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

{

/\* base cases \*/

if (S == NULL)

return true;

if (T == NULL)

return false;

/\* Check the tree with root as current node \*/

if (areIdentical(T, S))

return true;

/\* If the tree with root as current node doesn't match then

try left and right subtrees one by one \*/

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

isSubtree(T->right, S);

}

**Populate Inorder Successor Of All Nodes:**

Extra space (one extra variable) named inorder\_successor is required for all node.

We can populate it using reverse inorder traversal.

**Tree IsoMorphism Problem:**

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.

(I can)

**Reverse Alternate Levels Of A Binary Tree:**

Application of level order traversal and deque.

**Serialize and Deserialize an N-ary Tree**

Given an N-ary tree where every node has at-most N children. How to serialize and deserialze it? Serialization is to store tree in a file so that it can be later restored. The structure of tree must be maintained. Deserialization is reading tree back from file.

In an N-ary tree, there are no designated left and right children. An N-ary tree is represented by storing an array or list of child pointers with every node.

The idea is to store an ‘end of children’ marker with every node. The following diagram shows serialization where ‘)’ is used as end of children marker.

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

{

// Base case

if (root == NULL) return;

// Else, store current node and recur for its children

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

for (int i = 0; i < N && root->child[i]; i++)

serialize(root->child[i], fp);

// Store marker at the end of children

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

}

This is depth first order traversal. And, insertion of a special character after every leaf node is reached.

Now, as you can see, DFS is followed to put words into a file. And, if

However, marker appending logic might needed to be changed.

Now, I personally feel the following is a better logic

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

{

// Base case

if (root == NULL) return;

// Else, store current node and recur for its children

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

bool presence\_of\_child=false;

for (int i = 0; i < N && root->child[i]; i++)

{

Presence\_of\_child=true;

serialize(root->child[i], fp);

}

// Store marker at the end of children

If(presence\_of\_child==false)

{

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

}

}

Now, deserialize part.

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

{

// Read next item from file. If theere are no more items or next

// item is marker, then return 1 to indicate same

char val;

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

return 1;

//when fscanf(fp,”%c”,&val) returns 0

//On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the *end-of-file*.

// Else create node with this item and recur for children

root = newNode(val);

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

if (deSerialize(root->child[i], fp))

break;

// Finally return 0 for successful finish

return 0;

**How to detect a graph is a tree or not.**

**How To Check For Cycle:**

We can either use BFS or DFS. For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle (See Detect cycle in an undirected graph for more details).

(but, better is BFS)

**How to check for connectivity?**

Since the graph is undirected, we can start BFS or DFS from any vertex and check if all vertices are reachable or not. If all vertices are reachable, then graph is connected, otherwise not.

(must be undirected. Directed graph has a different way of detecting connectivvity)

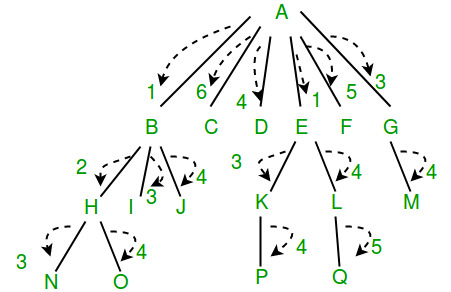
(from any vertex, not from every vertex)

**Minimum Number Of Iterations To Pass Information To All Nodes In The Tree:**

Given a very large n-ary tree. Where the root node has some information which it wants to pass to all of its children down to the leaves with the constraint that it can only pass the information to one of its children at a time (take it as one iteration).

Now in the next iteration the child node can transfer that information to only one of its children and at the same time instance the child’s parent i.e. root can pass the info to one of its remaining children. Continuing in this way we have to find the minimum no of iterations required to pass the information to all nodes in the tree.

Minimum no of iterations for tree below is 6. The root A first passes information to B. In next iteration, A passes information to E and B passes information to H and so on.



This can be done using Post Order Traversal. The idea is to consider height and children count on each and every node.

If a child node i takes ci iterations to pass info below its subtree, then its parent will take (ci + 1) iterations to pass info to subtree rooted at that child i.

If parent has more children, it will pass info to them in subsequent iterations. Let’s say children of a parent takes c1, c2, c3, c4, …, cn iterations to pass info in their own subtree, Now parent has to pass info to these n children one by one in n iterations. If parent picks child i in ith iteration, then parent will take (i + ci) iterations to pass info to child i and all it’s subtree.

In any iteration, when parent passes info a child i+1, children (1 to i) which got info from parent already in previous iterations, will pass info to further down in subsequent iterations, if any child (1 to i) has its own child further down.

Nodes with height = 0: (Trivial case) Leaf node has no children (no information passing needed), so no of iterations would be ZERO.

Nodes with height = 1: Here node has to pass info to all the children one by one (all children are leaf node, so no more information passing further down). Since all children are leaf, node can pass info to any child in any order (pick any child who didn’t receive the info yet). One iteration needed for each child and so no of iterations would be no of children.So node with height 1 with n children will take n iterations.

Take a counter initialized with ZERO, loop through all children and keep incrementing counter.

Nodes with height > 1: Let’s assume that there are n children (1 to n) of a node and minimum no iterations for all n children are c1, c2, …., cn.

To make sure maximum no of nodes participate in info passing in any iteration, parent should 1st pass info to that child who will take maximum iteration to pass info further down in subsequent iterations. i.e. in any iteration, parent should choose the child who takes maximum iteration later on. It can be thought of as a greedy approach where parent choose that child 1st, who needs maximum no of iterations so that all subsequent iterations can be utilized efficiently.

If parent goes in any other fashion, then in the end, there could be some nodes which are done quite early, sitting idle and so bandwidth is not utilized efficiently in further iterations.

If there are two children i and j with minimum iterations ci and cj where ci > cj, then If parent picks child j 1st then no of iterations needed by parent to pass info to both children and their subtree would be:max (1 + cj, 2 + ci) = 2 + ci

If parent picks child i 1st then no of iterations needed by parent to pass info to both children and their subtree would be: max(1 + ci, 2 + cj) = 1 + ci (So picking ci gives better result than picking cj)

So,

sort all ci values decreasing order,  
let’s say after sorting, values are c1 > c2 > c3 > …. > cn  
**take a counter c, set c = 1 + c1 (for child with maximum no of iterations)  
for all children i from 2 to n, c = c + 1 + ci**

**(c is a counter value)**

C is the counter value. Starting from 0 and incremented by 1 after every step

Ci is the needed number of iterations for ith node to pass the information to it’s subtree.

**Print All Nodes At Distance k From A Given Node**

There are two types of nodes to be considered.

1) Nodes in the subtree rooted with target node. For example if the target node is 8 and k is 2, then such nodes are 10 and 14.

2) Other nodes, may be an ancestor of target, or a node in some other subtree. For target node 8 and k is 2, the node 22 comes in this category.

Finding the first type of nodes is easy to implement. Just traverse subtrees rooted with the target node and decrement k in recursive call. When the k becomes 0, print the node currently being traversed (See this for more details). Here we call the function as printkdistanceNodeDown().

How to find nodes of second type? For the output nodes not lying in the subtree with the target node as the root, we must go through all ancestors. For every ancestor, we find its distance from target node, let the distance be d, now we go to other subtree (if target was found in left subtree, then we go to right subtree and vice versa) of the ancestor and find all nodes at k-d distance from the ancestor.

**Expression Tree:**

**Construction Of Expression Tree:**

**Construction For Expression Tree:**

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.  
1) If character is operand push that into stack  
2) If character is operator pop two values from stack make them its child and push current node again.  
At the end only element of stack will be root of expression tree.

Now, visiting this expression tree in different order would convert infix to prefix and infix to postfix.

Construction Tree Can Even be constructed from postfix and prefix expression. We just need to think.

I personally found converting it form postfix is very easy.

And, the algorithm for construction of expression tree which is given here is from postfix expression.

However, there is other ways:

**Convert Infix to Postfix Expression:**

1. Scan the infix expression from left to right.  
   **2.** If the scanned character is an operand, output it.  
   **3.**Else,  
   …..**3.1** If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.  
   …..**3.2** Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.  
   **4.** If the scanned character is an ‘(‘, push it to the stack.  
   **5.** If the scanned character is an ‘)’, pop and output from the stack until an ‘(‘ is encountered.  
   **6.** Repeat steps 2-6 until infix expression is scanned.  
   **7.**Pop and output from the stack until it is not empty.  
     
   **Convert Infix To Prefix Expression:**
2. Reverse the infix expression.
3. Scan the reversed infix expression from left to right
4. If the scanned character is an operand, keep in in a list.
5. Else
   1. If the precedence of the scanned operator is greater than or equal the precedence of the operator in the stack(or the stack is empty), push it.
   2. Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.
6. If the scanned character is an ‘(‘, push it to the stack.
7. If the scanned character is an ‘)’, pop and output from the stack until an ‘(‘ is encountered.
8. Repeat steps 2-6 until reverse infix expression is scanned.
9. The reverse of the generated list or string is the result

**Foldable Binary Trees:**

A tree can be folded if left and right subtrees of the tree are structure wise mirror image of each other. An empty tree is considered as foldable.

**First Approach:**

1) If tree is empty, then return true.

2) Convert the left subtree to its mirror image

mirror(root->left); /\* See this post \*/

3) Check if the structure of left subtree and right subtree is same

and store the result.

res = isStructSame(root->left, root->right); /\*isStructSame()

recursively compares structures of two subtrees and returns

true if structures are same \*/

4) Revert the changes made in step (2) to get the original tree.

mirror(root->left);

5) Return result res stored in step 2.

**Second Approach:**

IsFoldable(root)

1) If tree is empty then return true

2) Else check if left and right subtrees are structure wise mirrors of

each other. Use utility function IsFoldableUtil(root->left,

root->right) for this.

// Checks if n1 and n2 are mirror of each other.

IsFoldableUtil(n1, n2)

1) If both trees are empty then return true.

2) If one of them is empty and other is not then return false.

3) Return true if following conditions are met

a) n1->left is mirror of n2->right

b) n1->right is mirror of n2->left

**Maximum Path Sum In A Tree:**

Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree.

**Solution:**

For each node there can be four ways that the max path goes through the node:

1. Node only

2. Max path through Left Child + Node

3. Max path through Right Child + Node

4. Max path through Left Child + Node + Max path through Right Child

The idea is to keep trace of four paths and pick up the max one in the end. An important thing to note is, root of every subtree need to return maximum path sum such that at most one child of root is involved. This is needed for parent function call. In below code, this sum is stored in ‘max\_single’ and returned by the recursive function.

(ma\_single will contain the max among (node only,max path through left child+Node, node+ max path through right child) (at most of side of child sum is involved)

// This function returns overall maximum path sum in 'res'

// And returns max path sum going through root.

int findMaxUtil(Node\* root, int &res)

{

//Base Case

if (root == NULL)

return 0;

// l and r store maximum path sum going through left and

// right child of root respectively

int l = findMaxUtil(root->left,res);

//This returns maximum path sum from l such that at most one child of l is involved

int r = findMaxUtil(root->right,res);

//this returns maximum path from r such that at most one child of r is involved

// Max path for parent call of root. This path must

// include at-most one child of root

int max\_single = max(max(l, r) + root->data, root->data);

// Max Top represents the sum when the Node under

// consideration is the root of the maxsum path and no

// ancestors of root are there in max sum path

int max\_top = max(max\_single, l + r + root->data);

res = max(res, max\_top); // Store the Maximum Result.

return max\_single;

}

// Returns maximum path sum in tree with given root

int findMaxSum(Node \*root)

{

// Initialize result

int res = INT\_MIN;

// Compute and return result

findMaxUtil(root, res);

return res;

}

**Check If Leaf Traversals Of Both Are Same:**

1. Create empty stacks stack1 and stack2

for iterative traversals of tree1 and tree2

2. insert (root of tree1) in stack1

insert (root of tree2) in stack2

3. Stores current leaf nodes of tree1 and tree2

temp1 = (root of tree1)

temp2 = (root of tree2)

4. Traverse both trees using stacks

while (stack1 and stack2 parent empty)

{

// Means excess leaves in one tree

if (if one of the stacks are empty)

return false

// get next leaf node in tree1

temp1 = stack1.pop()

while (temp1 is not leaf node)

{

push right child to stack1

push left child to stack1

}

// get next leaf node in tree2

temp2 = stack2.pop()

while (temp2 is not leaf node)

{

push right child to stack2

push left child to stack2

}

// If leaves do not match return false

if (temp1 != temp2)

return false

}

1. If all leaves matched, return true

**Threaded Binary Tree:**

We have discussed Threaded Binary Tree. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. In a simple threaded binary tree, the NULL right pointers are used to store inorder successor. Where-ever a right pointer is NULL, it is used to store inorder successor.

(why only NULL right pointers because, for left pointers the successor is the parent node or the node itself)

The following is the structure of threaded binary tree:

**An extra variable is needed:**

**struct Node**

**{**

**int key;**

**Node \*left, \*right;**

**// Used to indicate whether the right pointer is a normal right**

**// pointer or a pointer to inorder successor.**

**bool isThreaded;**

**};**

The following code snippet shows how to do it using queue (there are other ways to do so)

// Helper function to put the Nodes in inorder into queue

void populateQueue(Node \*root, std::queue <Node \*> \*q)

{

if (root == NULL) return;

if (root->left)

populateQueue(root->left, q);

q->push(root);

if (root->right)

populateQueue(root->right, q);

}

// Function to traverse queue, and make tree threaded

void createThreadedUtil(Node \*root, std::queue <Node \*> \*q)

{

if (root == NULL) return;

if (root->left)

createThreadedUtil(root->left, q);

q->pop();

if (root->right)

createThreadedUtil(root->right, q);

// If right pointer is NULL, link it to the

// inorder successor and set 'isThreaded' bit.

else

{

root->right = q->front();

root->isThreaded = true;

}

}

**But more important part is how to traverse it:**

void inOrder(Node \*root)

{

if (root == NULL) return;

// Find the leftmost node in Binary Tree

Node \*cur = leftMost(root);

while (cur != NULL)

{

cout << cur->key << " ";

// If this Node is a thread Node, then go to

// inorder successor

if (cur->isThreaded)

cur = cur->right;

else // Else go to the leftmost child in right subtree

cur = leftMost(cur->right);

}

}