* **Construct Tree from given Inorder and Preorder traversals** :-

Algorithm: buildTree()

1) Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick next element in next recursive call.

2) Create a new tree node tNode with the data as picked element.

3) Find the picked element’s index in Inorder. Let the index be inIndex.

4) Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.

5) Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.

6) return tNode.

node\* buildTree(char in[], char pre[], int inStrt, int inEnd)

{

static int preIndex = 0; **// avoid using static, will get wrong answer in multiple test cases**

if (inStrt > inEnd)

return NULL;

node\* tNode = newNode(pre[preIndex++]);

if (inStrt == inEnd)

return tNode;

int inIndex = search(in, inStrt, inEnd, tNode->data);

tNode->left = buildTree(in, pre, inStrt, inIndex - 1);

tNode->right = buildTree(in, pre, inIndex + 1, inEnd);

return tNode;

}

int search(char arr[], int strt, int end, char value)

{

int i;

for (i = strt; i <= end; i++)

{

if (arr[i] == value)

return i;

}

}

* **Construct a tree from Inorder and Level order traversals** :-

int search(int arr[], int strt, int end, int value)

{

for (int i = strt; i <= end; i++)

if (arr[i] == value)

return i;

return -1;

}

int \*extrackKeys(int in[], int level[], int m, int n)

{

int \*newlevel = new int[m], j = 0;

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

if (search(in, 0, m-1, level[i]) != -1)

newlevel[j] = level[i], j++;

return newlevel;

}

Node\* buildTree(int in[], int level[], int inStrt, int inEnd, int n)

{

if (inStrt > inEnd)

return NULL;

Node \*root = newNode(level[0]);

if (inStrt == inEnd)

return root;

int inIndex = search(in, inStrt, inEnd, root->key);

int \*llevel = extrackKeys(in, level, inIndex, n);

int \*rlevel = extrackKeys(in + inIndex + 1, level, n-inIndex-1, n);

root->left = buildTree(in, llevel, inStrt, inIndex-1, n);

root->right = buildTree(in, rlevel, inIndex+1, inEnd, n);

delete [] llevel;

delete [] rlevel;

return root;

}

* **Construct Full Binary Tree from given preorder and postorder traversals** :-

It is not possible to construct a general Binary Tree from preorder and postorder traversals . But if know that the Binary Tree is Full, we can construct the tree without ambiguity

node\* constructTreeUtil (int pre[], int post[], int\* preIndex,

int l, int h, int size)

{

if (\*preIndex >= size || l > h)

return NULL;

node\* root = newNode ( pre[\*preIndex] );

++\*preIndex;

if (l == h)

return root;

int i;

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

if (pre[\*preIndex] == post[i])

break;

if (i <= h)

{

root->left = constructTreeUtil (pre, post, preIndex, l, i, size);

root->right = constructTreeUtil (pre, post, preIndex, i+1, h, size);

}

return root;

}

node \*constructTree (int pre[], int post[], int size)

{

int preIndex = 0;

return constructTreeUtil (pre, post, &preIndex, 0, size - 1, size);

}

* **Construct Full Binary Tree using its Preorder traversal and Preorder traversal of its mirror tree** :-

If we observe carefully, then reverse of the Preorder traversal of mirror tree will be the Postorder traversal of original tree.

So it can be reduced to the question above.

* **Finding Ancestor Matrix** :-

<https://www.geeksforgeeks.org/construct-ancestor-matrix-from-a-given-binary-tree/>

mat[i][j] = 1 if i is ancestor of j

mat[i][j] = 0, otherwise

The idea is to traverse the tree. While traversing, keep track of ancestors in an array. When we visit a node, we add it to ancestor array and consider corresponding row in adjacency matrix. We mark all ancestors in its row as 1. Once a node and all its children are processed, we remove the node from ancestor array.

int ancestorMatrixRec(Node \*root, vector<int> &anc)

{

if (root == NULL) return 0;;

int data = root->data;

for (int i=0; i<anc.size(); i++)

mat[anc[i]][data] = true;

anc.push\_back(data);

int l = ancestorMatrixRec(root->left, anc);

int r = ancestorMatrixRec(root->right, anc);

anc.pop\_back();

return l+r+1;

}

void ancestorMatrix(Node \*root)

{

vector<int> anc;

int n = ancestorMatrixRec(root, anc);

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

{

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

cout << mat[i][j] << " ";

cout << endl;

}

}

* **Construct tree from ancestor matrix** :-  
  <https://www.geeksforgeeks.org/construct-tree-from-ancestor-matrix/>

There can be many possible trees

The idea is to construct the tree in bottom up manner.

1) Create an array of node pointers node[].

2) Store row numbers that correspond to a given count. We have used multimap for this purpose.

3) Process all entries of multimap from smallest count to largest (Note that entries in map and multimap can be traversed in sorted order). Do following for every entry.

…….a) Create a new node for current row number.

…….b) If this node is not a leaf node, consider all those descendants of it whose parent is not set, make current node as its parent.

4) The last processed node (node with maximum sum) is root of tree.

* **Construct Binary Tree from given Parent Array representation** :-

Node \*createTree(int parent[], int n)

{

Node\* nodes[n];

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

nodes[i] = newNode(i);

Node\* root;

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

{

if(parent[i] == -1)

{

root = nodes[i];

continue;

}

if(nodes[parent[i]]->left == NULL)

nodes[parent[i]]->left = nodes[i];

else

nodes[parent[i]]->right = nodes[i];

}

return root;

}

* **Construct a Binary Tree from Postorder and Inorder** :-

1) We first find the last node in post[]. The last node is “1”, we know this value is root as root always appear in the end of postorder traversal.

2) We search “1” in in[] to find left and right subtrees of root. Everything on left of “1” in in[] is in left subtree and everything on right is in right subtree.

3) We recur the above process for following two.

….b) Recur for in[] = {6, 3, 7} and post[] = {6, 7, 3}

…….Make the created tree as right child of root.

….a) Recur for in[] = {4, 8, 2, 5} and post[] = {8, 4, 5, 2}.

…….Make the created tree as left child of root.

**Note :- First Recur for right subtree**

Node\* buildUtil(int in[], int post[], int inStrt, int inEnd, int\* pIndex)

{

if (inStrt > inEnd)

return NULL;

Node\* node = newNode(post[\*pIndex]);

(\*pIndex)--;

if (inStrt == inEnd)

return node;

int iIndex = search(in, inStrt, inEnd, node->data);

**node->right = buildUtil(in, post, iIndex + 1, inEnd, pIndex);**

**node->left = buildUtil(in, post, inStrt, iIndex - 1, pIndex);**

**// first recur for right subtree**

return node;

}

Node\* buildTree(int in[], int post[], int n)

{

int pIndex = n - 1;

return buildUtil(in, post, 0, n - 1, &pIndex);

}

int search(int arr[], int strt, int end, int value)

{

int i;

for (i = strt; i <= end; i++) {

if (arr[i] == value)

break;

}

return i;

}

* **Create a Doubly Linked List from a Ternary Tree** :-

void push(Node\*\* tail\_ref, Node\* node)

{

if (\*tail\_ref == NULL)

{

\*tail\_ref = node;

node->left = node->middle = node->right = NULL;

return;

}

(\*tail\_ref)->right = node;

node->left = (\*tail\_ref);

node->right = node->middle = NULL;

(\*tail\_ref) = node;

}

Node\* TernaryTreeToList(Node\* root, Node\*\* head\_ref)

{

if (root == NULL)

return NULL;

static Node\* tail = NULL;

Node\* left = root->left;

Node\* middle = root->middle;

Node\* right = root->right;

if (\*head\_ref == NULL)

\*head\_ref = root;

push(&tail, root);

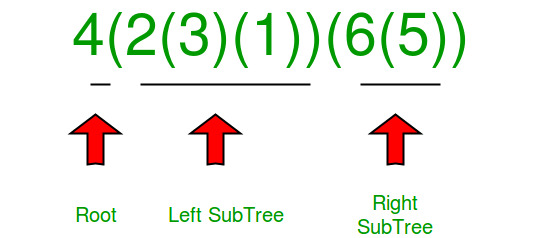
TernaryTreeToList(left, head\_ref);

TernaryTreeToList(middle, head\_ref);

TernaryTreeToList(right, head\_ref);

}

* **Construct Binary Tree from String with bracket representation** :-



int findIndex(string str, int si, int ei)

{

if (si > ei)

return -1;

stack<char> s;

for (int i = si; i <= ei; i++) {

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

s.push(str[i]);

else if (str[i] == ')') {

if (s.top() == '(') {

s.pop();

if (s.empty())

return i;

}

}

}

return -1;

}

Node\* treeFromString(string str, int si, int ei)

{

if (si > ei)

return NULL;

Node\* root = newNode(str[si] - '0');

int index = -1;

if (si + 1 <= ei && str[si + 1] == '(')

index = findIndex(str, si + 1, ei);

if (index != -1) {

root->left = treeFromString(str, si + 2, index - 1);

root->right = treeFromString(str, index + 2, ei - 1);

}

return root;

}

* **Convert a given Binary Tree to Doubly Linked List** :-
  + 2 traversals

void fixPrevPtr(node \*root)

{

static node \*pre = NULL;

if (root != NULL)

{

fixPrevPtr(root->left);

root->left = pre;

pre = root;

fixPrevPtr(root->right);

}

}

node \*fixNextPtr(node \*root)

{

node \*prev = NULL;

while (root && root->right != NULL)

root = root->right;

while (root && root->left != NULL)

{

prev = root;

root = root->left;

root->right = prev;

}

return (root);

}

node \*BTToDLL(node \*root)

{

fixPrevPtr(root);

return fixNextPtr(root);

}

* + Single inorder traversal

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.

void BinaryTree2DoubleLinkedList(node \*root, node \*\*head)

{

if (root == NULL) return;

static node\* prev = NULL;

BinaryTree2DoubleLinkedList(root->left, head);

if (prev == NULL)

\*head = root;

else

{

root->left = prev;

prev->right = root;

}

prev = root;

BinaryTree2DoubleLinkedList(root->right, head);

}

* Another method :-

void BToDLL(Node\* root, Node\*\* head\_ref)

{

if (root == NULL)

return;

BToDLL(root->right, head\_ref);

root->right = \*head\_ref;

if (\*head\_ref != NULL)

(\*head\_ref)->left = root;

\*head\_ref = root;

BToDLL(root->left, head\_ref);

}

* **Convert an arbitrary Binary Tree to a tree that holds Children Sum Property**

void convertTree(node\* node)

{

int left\_data = 0, right\_data = 0, diff;

if (node == NULL || (node->left == NULL && node->right == NULL))

return;

else

{

convertTree(node->left);

convertTree(node->right);

if (node->left != NULL)

left\_data = node->left->data;

if (node->right != NULL)

right\_data = node->right->data;

diff = left\_data + right\_data - node->data;

if (diff > 0)

node->data = node->data + diff;

/\* THIS IS TRICKY --> If node's data

is greater than children sum,

then increment subtree by diff \*/

if (diff < 0)

increment(node, -diff); // -diff is used to make diff positive

}

}

void increment(node\* node, int diff)

{

if(node->left != NULL)

{

node->left->data = node->left->data + diff;

increment(node->left, diff);

}

else if (node->right != NULL) // Else increment right child

{

node->right->data = node->right->data + diff;

increment(node->right, diff);

}

}

* **Minimum swap required to convert binary tree to binary search tree** :-

void inorder(int a[], std::vector<int> &v, int n, int index)

{

if(index >= n)

return;

inorder(a, v, n, 2 \* index + 1);

v.push\_back(a[index]);

inorder(a, v, n, 2 \* index + 2);

}

int minSwaps(std::vector<int> &v)

{

std::vector<pair<int,int> > t(v.size());

int ans = 0;

for(int i = 0; i < v.size(); i++)

t[i].first = v[i], t[i].second = i;

sort(t.begin(), t.end());

for(int i = 0; i < t.size(); i++)

{

if(i == t[i].second)

continue;

else

{

swap(t[i].first, t[t[i].second].first);

swap(t[i].second, t[t[i].second].second);

}

if(i != t[i].second)

--i;

ans++;

}

return ans;

}

* **Convert Ternary Expression to a Binary Tree** :-

string expression = a?b:c

Output : a

/ \

b c

Idea is that we traverse a string make first character as root and do following step recursively .

1. If we see Symbol ‘?’

…….. then we add next character as the left child of root.

2. If we see Symbol ‘:’

…….. then we add it as the right child of current root.

do this process until we traverse all element of “String”.

Node \*convertExpression(string str, int & i)

{

Node \* root =newNode(str[i]);

if(i==str.length()-1) return root;

i++;

if(str[i]=='?')

{

//skip the '?'

i++;

//construct the left subtree

//Notice after the below recursive call i will point to ':' just before the right child of current node since we pass i by reference

root->left = convertExpression(str,i);

//skip the ':' character

i++;

root->right = convertExpression(str,i);

return root;

}

else return root;

}

* Convert a tree to forest of even nodes

Given a tree of n even nodes. The task is to find the maximum number of edges to be removed from the given tree to obtain forest of trees having even number of nodes. This problem is always solvable as given graph has even nodes.

int dfs(vector<int> tree[N], int visit[N], int \*ans, int node)

{

int num = 0, temp = 0;

visit[node] = 1;

for (int i = 0; i < tree[node].size(); i++)

{

if (visit[tree[node][i]] == 0)

{

temp = dfs(tree, visit, ans, tree[node][i]);

(temp%2)?(num += temp):((\*ans)++);

}

}

return num+1;

}

int minEdge(vector<int> tree[N], int n)

{

int visit[n+2];

int ans = 0;

memset(visit, 0, sizeof visit);

dfs(tree, visit, &ans, 1);

return ans;

}

* **Construct the full k-ary tree from its preorder traversal** :-

Given an array which contains the preorder traversal of full k-ary tree, construct the full k-ary tree and print its postorder traversal. A full k-ary tree is a tree where each node has either 0 or k children.

n Preorder traversal, first root node is processed then followed by the left subtree and right subtree. Because of this, to construct a full k-ary tree, we just need to keep on creating the nodes without bothering about the previous constructed nodes. We can use this to build the tree recursively.

Following are the steps to solve the problem:

1. Find the height of the tree.

2. Traverse the preorder array and recursively add each node

Note :- A unique tree can’t be found with only preorder. We are using height as an extra parameter by ourselves.

Node\* BuildKaryTree(int A[], int n, int k, int h, int& ind)

{

if (n <= 0)

return NULL;

Node\* nNode = newNode(A[ind]);

if (nNode == NULL) {

cout << "Memory error" << endl;

return NULL;

}

for (int i = 0; i < k; i++) {

if (ind < n - 1 && h > 1) {

ind++;

nNode->child.push\_back(BuildKaryTree(A, n, k, h - 1, ind));

} else {

nNode->child.push\_back(NULL);

}

}

return nNode;

}

Node\* BuildKaryTree(int\* A, int n, int k, int ind)

{

int height = (int)ceil(log((double)n \* (k - 1) + 1) / log((double)k));

return BuildKaryTree(A, n, k, height, ind);

}

* **Flip Binary Tree :-**

In the flip operation, left most node becomes the root of flipped tree and its parent become its right child and the right sibling become its left child and same should be done for all left most nodes recursively.

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

root->left->right = root;

root->left = NULL;

root->right = NULL;

Node\* flipBinaryTree(Node\* root)

{

if (root == NULL)

return root;

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

return root;

Node\* flippedRoot = flipBinaryTree(root->left);

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

root->left->right = root;

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

return flippedRoot;

}