* Inorder Traversal :-
  + Without recursion (using stack):-

void inOrder(struct Node \*root)

{

stack<Node \*> s;

Node \*curr = root;

while (curr != NULL || s.empty() == false)

{

while (curr != NULL)

{

s.push(curr);

curr = curr->left;

}

curr = s.top();

s.pop();

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

curr = curr->right;

}

}

* + Iterative without stack (Morris Traversal) :-

void MorrisTraversal(struct tNode\* root)

{

struct tNode \*current, \*pre;

if (root == NULL)

return;

current = root;

while (current != NULL) {

if (current->left == NULL) {

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

current = current->right;

}

else {

/\* Find the inorder predecessor of current \*/

pre = current->left;

while (pre->right != NULL && pre->right != current)

pre = pre->right;

/\* Make current as the right child of its inorder

predecessor \*/

if (pre->right == NULL) {

pre->right = current;

current = current->left;

}

/\* Revert the changes made in the 'if' part to restore

the original tree i.e., fix the right child

of predecessor \*/

else {

pre->right = NULL;

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

current = current->right;

} /\* End of if condition pre->right == NULL \*/

} /\* End of if condition current->left == NULL\*/

} /\* End of while \*/

}

* **Iterative Preorder** :-
  + Using stack :-

Following is a simple stack based iterative process to print Preorder traversal.

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

void iterativePreorder(node \*root)

{

if (root == NULL)

return;

stack<node \*> nodeStack;

nodeStack.push(root);

while (nodeStack.empty() == false)

{

struct node \*node = nodeStack.top();

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

nodeStack.pop();

if (node->right)

nodeStack.push(node->right);

if (node->left)

nodeStack.push(node->left);

}

}

* + Morris traversal :-

The algorithm for Preorder is almost similar to Morris traversal for Inorder.

1...If left child is null, print the current node data. Move to right child.

….Else, Make the right child of the inorder predecessor point to the current node. 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.

2...Iterate until current node is not NULL.

void morrisTraversalPreorder(node\* root)

{

while (root)

{

if (root->left == NULL)

{

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

root = root->right;

}

else

{

node\* current = root->left;

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

current = current->right;

if (current->right == root)

{

current->right = NULL;

root = root->right;

}

else

{

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

current->right = root;

root = root->left;

}

}

}

}

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

The idea is to push reverse postorder traversal to a stack. Once we have the reversed postorder traversal in a stack, we can just pop all items one by one from the stack and print them; this order of printing will be in postorder because of the LIFO property of 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

void postOrderIterative(Node\* root)

{

if (root == NULL)

return;

stack<Node \*> s1, s2;

s1.push(root);

Node\* node;

while (!s1.empty()) {

node = s1.top();

s1.pop();

s2.push(node);

if (node->left)

s1.push(node->left);

if (node->right)

s1.push(node->right);

}

while (!s2.empty()) {

node = s2.top();

s2.pop();

cout << node->data << " ";

}

}

* + Using 1 Stack :-

Following is detailed algorithm.

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.

void postOrderIterative(struct Node\* root)

{

if (root == NULL)

return;

struct Stack\* stack = createStack(MAX\_SIZE);

do

{

while (root)

{

if (root->right)

push(stack, root->right);

push(stack, root);

root = root->left;

}

root = pop(stack);

if (root->right && peek(stack) == root->right)

{

pop(stack); // remove right child from stack

push(stack, root); // push root back to stack

root = root->right

}

else // Else print root's data and set root as NULL

{

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

**root = NULL;**

}

} while (!isEmpty(stack));

}

* + Without stack :-
    - O(n\*n) solution :-

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;

}

}

}

* + - O(n) solution (use hashmap to store parent):-

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;

}

}

}

* **Diagonal Traversal of Binary Tree** :-

void diagonalPrintUtil(Node\* root, int d,

map<int, vector<int>> &diagonalPrint)

{

if (!root)

return;

diagonalPrint[d].push\_back(root->data);

diagonalPrintUtil(root->left, d + 1, diagonalPrint);

diagonalPrintUtil(root->right, d, diagonalPrint);

}

void diagonalPrint(Node\* root)

{

map<int, vector<int> > diagonalPrint;

diagonalPrintUtil(root, 0, diagonalPrint);

cout << "Diagonal Traversal of binary tree : n";

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

{

for (auto itr = it->second.begin();

itr != it->second.end(); ++itr)

cout << \*itr << ' ';

cout << '\n';

}

}

* **Diagonal Traversal of Binary Tree** :-
  + Using Hashmap :-

void diagonalPrintUtil(Node\* root, int d,

map<int, vector<int>> &diagonalPrint)

{

if (!root)

return;

diagonalPrint[d].push\_back(root->data);

diagonalPrintUtil(root->left, d + 1, diagonalPrint);

diagonalPrintUtil(root->right, d, diagonalPrint);

}

void diagonalPrint(Node\* root)

{

map<int, vector<int> > diagonalPrint;

diagonalPrintUtil(root, 0, diagonalPrint);

cout << "Diagonal Traversal of binary tree : n";

for (auto it = diagonalPrint.begin();

it != diagonalPrint.end(); ++it)

{

for (auto itr = it->second.begin();

itr != it->second.end(); ++itr)

cout << \*itr << ' ';

cout << 'n';

}

}

* + Simple Recursion :-

void diagonalPrint(Node \*root)

{

queue<Node\*> q;

q.push(root);

while(!q.empty())

{

int s = q.size();

while(s--)

{

Node\* f = q.front();

q.pop();

while(f)

{

cout << f->data << " ";

if(f->left)

q.push(f->left);

f = f->right;

}

}

}

}

* **Print Postorder traversal from given Inorder and Preorder traversals** :-

We can print postorder traversal without constructing the tree. The idea is, root is always the first item in preorder traversal and it must be the last item in postorder traversal. We first recursively print left subtree, then recursively print right subtree. Finally, print root. To find boundaries of left and right subtrees in pre[] and in[], we search root in in[], all elements before root in in[] are elements of left subtree and all elements after root are elements of right subtree. In pre[], all elements after index of root in in[] are elements of right subtree. And elements before index (including the element at index and excluding the first element) are elements of left subtree.

int search(int arr[], int x, int n)

{

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

if (arr[i] == x)

return i;

return -1;

}

void printPostOrder(int in[], int pre[], int n)

{

int root = search(in, pre[0], n);

if (root != 0)

printPostOrder(in, pre + 1, root);

if (root != n - 1)

printPostOrder(in + root + 1, pre + root + 1, n - root - 1);

cout << pre[0] << " ";

}

* **Populate Inorder Successor for all nodes** :-

Either use a prev pointer and do normal inorder traversal or use a next pointer and do reverse inorder traversal.

* **Inorder Successor of a node in Binary Tree** :-

<https://www.geeksforgeeks.org/inorder-succesor-node-binary-tree/>

Node\* findInorderRecursive(Node\* root, Node\* x )

{

if (!root)

return NULL;

if (root==x || (temp = findInorderRecursive(root->left,x)) ||

(temp = findInorderRecursive(root->right,x)))

{

if (temp)

{

if (root->left == temp)

{

cout << "Inorder Successor of " << x->data;

cout << " is "<< root->data << "\n";

return NULL;

}

}

return root;

}

return NULL;

}

void inorderSuccesor(Node\* root, Node\* x)

{

if (x->right != NULL)

{

Node\* inorderSucc = leftMostNode(x->right);

cout<<"Inorder Successor of "<<x->data<<" is ";

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

}

if (x->right == NULL)

{

Node\* rightMost = rightMostNode(root);

if (rightMost == x)

cout << "No inorder successor! Right most node.\n";

else

findInorderRecursive(root, x);

}

}

* **Level order traversal in spiral form** :-

<https://www.geeksforgeeks.org/level-order-traversal-in-spiral-form/>

We can print spiral order traversal in O(n) time and O(n) extra space. The idea is to use two stacks. We can use one stack for printing from left to right and other stack for printing from right to left. In every iteration, we have nodes of one level in one of the stacks. We print the nodes, and push nodes of next level in other stack.

void printSpiral(struct node\* root)

{

if (root == NULL)

return;

stack<struct node\*> s1;

stack<struct node\*> s2;

s1.push(root);

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

while (!s1.empty()) {

struct node\* temp = s1.top();

s1.pop();

cout << temp->data << " ";

if (temp->right)

s2.push(temp->right);

if (temp->left)

s2.push(temp->left);

}

while (!s2.empty()) {

struct node\* temp = s2.top();

s2.pop();

cout << temp->data << " ";

if (temp->left)

s1.push(temp->left);

if (temp->right)

s1.push(temp->right);

}

}

}

* **Reverse Level Order Traversal** :-

The idea is to print last level first, then second last level, and so on. Like Level order traversal, every level is printed from left to right.

Using Queue and Stack :-

void reverseLevelOrder(node\* root)

{

stack <node \*> S;

queue <node \*> Q;

Q.push(root);

while (Q.empty() == false)

{

root = Q.front();

Q.pop();

S.push(root);

if (root->right)

Q.push(root->right); // NOTE: RIGHT CHILD IS ENQUEUED BEFORE LEFT

if (root->left)

Q.push(root->left);

}

while (S.empty() == false)

{

root = S.top();

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

S.pop();

}

}

* **Reverse tree path** :-

Given a tree and a node data, your task to reverse the path till that particular Node.

Node\* reverseTreePathUtil(Node\* root, int data,

map<int, int>& temp, int level, int& nextpos)

{

if (root == NULL)

return NULL;

if (data == root->data) {

temp[level] = root->data;

root->data = temp[nextpos];

nextpos++;

return root;

}

temp[level] = root->data;

Node \*left, \*right;

left = reverseTreePathUtil(root->left, data, temp,

level + 1, nextpos);

if (left == NULL)

right = reverseTreePathUtil(root->right, data,

temp, level + 1, nextpos);

if (left || right) {

root->data = temp[nextpos];

nextpos++;

return (left ? left : right);

}

return NULL;

}

void reverseTreePath(Node\* root, int data)

{

map<int, int> temp;

int nextpos = 0;

reverseTreePathUtil(root, data, temp, 0, nextpos);

}

* **Perfect Binary Tree Specific Level Order Traversal** :-

<https://www.geeksforgeeks.org/perfect-binary-tree-specific-level-order-traversal/>

Given a Perfect Binary Tree like below:

**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**

i.e. print nodes in level order but nodes should be from left and right side alternatively. Here 1st and 2nd levels are trivial.

While 3rd level: 4(left), 7(right), 5(left), 6(right) are printed.

While 4th level: 8(left), 15(right), 9(left), 14(right), .. are printed.

While 5th level: 16(left), 31(right), 17(left), 30(right), .. are printed.

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.

void printSpecificLevelOrder(Node \*root)

{

if (root == NULL)

return;

cout << root->data;

if (root->left != NULL)

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

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

return;

queue <Node \*> q;

q.push(root->left);

q.push(root->right);

Node \*first = NULL, \*second = NULL;

while (!q.empty())

{

first = q.front();

q.pop();

second = q.front();

q.pop();

cout << " " << first->left->data << " " << second->right->data;

cout << " " << first->right->data << " " << second->left->data;

if (first->left->left != NULL)

{

q.push(first->left);

q.push(second->right);

q.push(first->right);

q.push(second->left);

}

}

}

<https://www.geeksforgeeks.org/perfect-binary-tree-specific-level-order-traversal-set-2/>

* **Level order traversal with direction change after every two levels** :-

We make use of queue and stack here, but in a different way. Using macros #define ChangeDirection(Dir) ((Dir) = 1 – (Dir)). In following implementation directs the order of push operations in both queue or stack.

In this way, we will achieve desired modified level order traversal by making use of queue and stack.

void modifiedLevelOrder(struct node \*root)

{

if (!root)

return ;

int dir = LEFT;

struct node \*temp;

queue <struct node \*> Q;

stack <struct node \*> S;

S.push(root);

while (!Q.empty() || !S.empty())

{

while (!S.empty())

{

temp = S.top();

S.pop();

cout << temp->data << " ";

if (dir == LEFT) {

if (temp->left)

Q.push(temp->left);

if (temp->right)

Q.push(temp->right);

}

else {

if (temp->right)

Q.push(temp->right);

if (temp->left)

Q.push(temp->left);

}

}

cout << endl;

while (!Q.empty())

{

temp = Q.front();

Q.pop();

cout << temp->data << " ";

if (dir == LEFT) {

if (temp->left)

S.push(temp->left);

if (temp->right)

S.push(temp->right);

} else {

if (temp->right)

S.push(temp->right);

if (temp->left)

S.push(temp->left);

}

}

cout << endl;

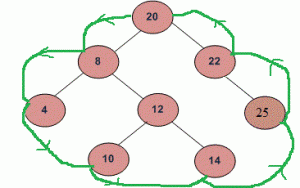
ChangeDirection(dir);

}

}

* Boundary Traversal of binary tree

Given a binary tree, print boundary nodes of the binary tree Anti-Clockwise starting from the root. For example, boundary traversal of the following tree is “20 8 4 10 14 25 22”



We break the problem in 3 parts:

1. Print the left boundary in top-down manner.

2. Print all leaf nodes from left to right, which can again be sub-divided into two sub-parts:

…..2.1 Print all leaf nodes of left sub-tree from left to right.

…..2.2 Print all leaf nodes of right subtree from left to right.

3. Print the right boundary in bottom-up manner.

void printLeaves(struct node\* root)

{

if (root) {

printLeaves(root->left);

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

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

printLeaves(root->right);

}

}

void printBoundaryLeft(struct node\* root)

{

if (root) {

if (root->left) {

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

printBoundaryLeft(root->left);

}

else if (root->right) {

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

printBoundaryLeft(root->right);

}

}

}

void printBoundaryRight(struct node\* root)

{

if (root) {

if (root->right) {

printBoundaryRight(root->right);

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

}

else if (root->left) {

printBoundaryRight(root->left);

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

}

}

}

void printBoundary(struct node\* root)

{

if (root) {

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

printBoundaryLeft(root->left);

printLeaves(root->left);

printLeaves(root->right);

printBoundaryRight(root->right);

}

}