1. **Constructing a binary Tree:**

struct node

{

    int data;

    struct node \*left;

    struct node \*right;

};

struct node\* newNode(int data)

{

  struct node\* node = (struct node\*)malloc(sizeof(struct node));

  node->data = data;

  node->left = NULL;

  node->right = NULL;

  return(node);

}

int main()

{

  /\*create root\*/

  struct node \*root = newNode(1);

  root->left        = newNode(2);

  root->right       = newNode(3);

  root->left->left  = newNode(4);

  getchar();

  return 0;

}

**Tree traversals:**

1. **Inorder Traversal**

void printInorder(struct node\* node)

{

     if (node == NULL)

          return;

     /\* first recur on left child \*/

     printInorder(node->left);

     /\* then print the data of node \*/

     printf("%d ", node->data);

     /\* now recur on right child \*/

     printInorder(node->right);

}

1. **PreOrder traversal**

void printPreorder(struct node\* node)

{

     if (node == NULL)

          return;

     /\* first print data of node \*/

     printf("%d ", node->data);

     /\* then recur on left sutree \*/

     printPreorder(node->left);

     /\* now recur on right subtree \*/

     printPreorder(node->right);

}

1. **Postorder traversal**

void printPostorder(struct node\* node)

{

     if (node == NULL)

        return;

     // first recur on left subtree

     printPostorder(node->left);

     // then recur on right subtree

     printPostorder(node->right);

     // now deal with the node

     printf("%d ", node->data);

}

1. **Level order traversal (method 1 : using queues)**

void printLevelOrder(struct node\* root)

{

    int rear, front;

    struct node \*\*queue = createQueue(&front, &rear);

    struct node \*temp\_node = root;

    while (temp\_node)

    {

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

        /\*Enqueue left child \*/

        if (temp\_node->left)

            enQueue(queue, &rear, temp\_node->left);

        /\*Enqueue right child \*/

        if (temp\_node->right)

            enQueue(queue, &rear, temp\_node->right);

        /\*Dequeue node and make it temp\_node\*/

        temp\_node = deQueue(queue, &front);

    }

}

1. **Level order traversal (Method 2): uses function which prints nodes at a given level**

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);

    }

}

1. **Calculating height of a tree**

int height(struct node\* node)

{

    if (node==NULL)

        return 0;

    else

    {

        /\* compute the height of each subtree \*/

        int lheight = height(node->left);

        int rheight = height(node->right);

        /\* use the larger one \*/

        if (lheight > rheight)

            return(lheight+1);

        else return(rheight+1);

    }

}

1. **Reverse level order traversal : method 1**

void reverseLevelOrder(struct node\* root)

{

    int h = height(root);

    int i;

    for (i=h; i>=1; i--) //THE ONLY LINE DIFFERENT FROM NORMAL LEVEL ORDER

        printGivenLevel(root, i);

}

1. **Reverse level order traversal : method 2 (using stack and queue)**

Do something like normal level order traversal order. Following are the differences with normal level order traversal

    1) Instead of printing a node, we push the node to stack

    2) Right subtree is visited before left subtree

void reverseLevelOrder(node\* root)

{

    stack <node \*> S;

    queue <node \*> Q;

    Q.push(root);

    while (Q.empty() == false)

    {

        /\* Dequeue node and make it root \*/

        root = Q.front();

        Q.pop();

        S.push(root);

        /\* Enqueue right child \*/

        if (root->right)

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

        /\* Enqueue left child \*/

        if (root->left)

            Q.push(root->left);

    }

    // Now pop all items from stack one by one and print them

    while (S.empty() == false)

    {

        root = S.top();

        cout << root->data << " ";

        S.pop();

    }

}

**\*Try to use STL wherever possible**

**10. Preorder without recursion**

void iterativePreorder(node \*root)

{

    // Base Case

    if (root == NULL)

       return;

    // Create an empty stack and push root to it

    stack<node \*> nodeStack;

    nodeStack.push(root);

    /\* Pop all items one by one. Do following for every popped item

       a) print it

       b) push its right child

       c) push its left child

    Note that right child is pushed first so that left is processed first \*/

    while (nodeStack.empty() == false)

    {

        // Pop the top item from stack and print it

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

        printf ("%d ", node->data);

        nodeStack.pop();

        // Push right and left children of the popped node to stack

        if (node->right)

            nodeStack.push(node->right);

        if (node->left)

            nodeStack.push(node->left);

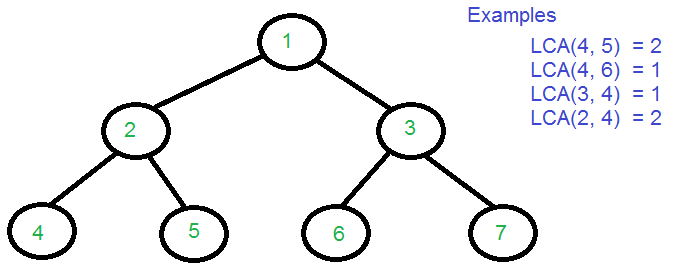
    }

}

**\*similarly look for inorder and postorder without recursion**

1. **Lowest Common Ancestor between two nodes**

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



// This function returns pointer to LCA of two given values n1 and n2.

// This function assumes that n1 and n2 are present in Binary Tree

struct Node \*findLCA(struct Node\* root, int n1, int n2)

{

    // Base case

    if (root == NULL) return NULL;

 /\*If either n1 or n2 matches with root's key, report the presence by returning root (Note that if a key is ancestor of other, then the ancestor key becomes LCA \*/

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

        return root;

    // Look for keys in left and right subtrees

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

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

    /\* If both of the above calls return Non-NULL, then one key is present in once subtree and other is present in other, So this node is the LCA \*/

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

    // Otherwise check if left subtree or right subtree is LCA

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

}

1. **Count the number of Leaf nodes**

unsigned int getLeafCount(struct node\* node)

{

  if(node == NULL)

    return 0;

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

    return 1;

  else

    return getLeafCount(node->left)+ getLeafCount(node->right);

}

1. **Print ancestors of a given node**

/\* If target is present in tree, then prints the ancestors and returns true, otherwise returns false. \*/

bool printAncestors(struct node \*root, int target)

{

  /\* base cases \*/

  if (root == NULL)

     return false;

  if (root->data == target)

     return true;

  /\* If target is present in either left or right subtree of this node,  then print this node \*/

if ( printAncestors(root->left, target) || printAncestors(root->right, target) )

  {

    cout << root->data << " ";

    return true;

  }

  /\* Else return false \*/

  return false;

}