1. Preorder traversal

vector <int> preorder(Node\* root)

{

vector<int> ans, temp;

if(root==NULL) return ans;

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

temp = preorder(root->left);

ans.insert(ans.end(), temp.begin(), temp.end());

temp = preorder(root->right);

ans.insert(ans.end(), temp.begin(), temp.end());

return ans;

}

1. Inorder traversal

vector<int> inOrder(Node\* root) {

vector<int> ans, temp;

if(root==NULL) return ans;

temp = inOrder(root->left);

ans.insert(ans.end(), temp.begin(), temp.end());

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

temp = inOrder(root->right);

ans.insert(ans.end(), temp.begin(), temp.end());

return ans;

}

1. Postorder traversal

vector <int> postOrder(Node\* root)

{

vector<int> ans, temp;

if(root==NULL) return ans;

temp = postOrder(root->left);

ans.insert(ans.end(), temp.begin(), temp.end());

temp = postOrder(root->right);

ans.insert(ans.end(), temp.begin(), temp.end());

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

return ans;

}

1. Height of binary tree

int height(struct Node\* node){

if(node==NULL) return 0;

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

}

1. Determine if two trees are identical

bool isIdentical(Node \*r1, Node \*r2)

{

if(r1==NULL && r2==NULL) return true;

if(r1==NULL || r2==NULL) return false;

return (r1->data==r2->data && isIdentical(r1->left, r2->left) && isIdentical(r1->right, r2->right));

}

int leftsum, rightsum;

if(root->left!=NULL)

leftsum = isSumProperty(root->left);

else

leftsum = 0;

if(root->right!=NULL)

rightsum = isSumPropert(root->right);

else

rightsum = 0;

if(rightsum==-1 || )

1. Children sum parent

int isSumProperty(Node \*root)

{

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

return 1;

else if(root->left==NULL)

return (root->data==root->right->data && isSumProperty(root->right));

else if(root->right==NULL)

return (root->data==root->left->data && isSumProperty(root->left));

else

return (root->data==(root->right->data + root->left->data) && isSumProperty(root->right) && isSumProperty(root->left));

}

1. Level order traversal

vector<int> levelOrder(Node\* node)

{

vector<int> ans;

queue<Node\*> q;

q.push(node);

while(!q.empty())

{

Node \*temp = q.front();

ans.push\_back(temp->data);

if(temp->left!=NULL)

q.push(temp->left);

if(temp->right!=NULL)

q.push(temp->right);

q.pop();

}

return ans;

}

1. Level order traversal line by line

vector<vector<int>> levelOrder(Node\* node)

{

vector<int> part;

vector <vector<int>> ans;

queue<Node\*> q;

q.push(node);

q.push(NULL);

while(q.size()!=1)

{

Node \*temp = q.front();

if(temp==NULL)

{

q.pop();

q.push(NULL);

ans.push\_back(part);

part.clear();

continue;

}

part.push\_back(temp->data);

if(temp->left!=NULL)

q.push(temp->left);

if(temp->right!=NULL)

q.push(temp->right);

q.pop();

}

q.pop();

q.push(NULL);

ans.push\_back(part);

return ans;

}

1. Level order traversal in spiral form

vector<int> findSpiral(Node \*root)

{

vector <int> ans;

if (root == NULL)

return ans;

stack<Node\*> s1;

stack<Node\*> s2;

s1.push(root);

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

{

while (!s1.empty())

{

Node\* temp = s1.top();

s1.pop();

ans.push\_back(temp->data);

if (temp->right!=NULL)

s2.push(temp->right);

if (temp->left!=NULL)

s2.push(temp->left);

}

while (!s2.empty())

{

Node\* temp = s2.top();

s2.pop();

ans.push\_back(temp->data);

if (temp->left!=NULL)

s1.push(temp->left);

if (temp->right!=NULL)

s1.push(temp->right);

}

}

return ans;

}

1. Max width of trees

int getMaxWidth(Node\* root) {

if(root==NULL) return 0;

int wid = 1, count = 0;

queue<Node\*> q;

q.push(root);

q.push(NULL);

while(q.size()!=1)

{

Node \*temp = q.front();

if(temp==NULL)

{

q.pop();

q.push(NULL);

wid = max(wid, count);

count = 0;

continue;

}

count++;

if(temp->left!=NULL)

q.push(temp->left);

if(temp->right!=NULL)

q.push(temp->right);

q.pop();

}

return max(wid, count);

}

1. Check for balanced tree

int cal(Node\*root)

{

if(root==NULL)

return 0;

int lh=cal(root->left);

if(lh==-1)return -1;

int rh=cal(root->right);

if(rh==-1)return -1;

if(abs(lh-rh)>1)

return -1;

else

return max(lh,rh)+1;

}

bool isBalanced(Node \*root)

{

if(cal(root)==-1)

return false;

else

return true;

}

1. Left view of binary tree(not working in time)

vector<int> leftView(Node \*node)

{

vector<int> ans;

queue<Node\*> q;

q.push(node);

ans.push\_back(node->data);

q.push(NULL);

while(q.size()!=1)

{

Node \*temp = q.front();

if(temp==NULL)

{

q.pop();

q.push(NULL);

ans.push\_back(q.front()->data);

continue;

}

if(temp->left!=NULL)

q.push(temp->left);

if(temp->right!=NULL)

q.push(temp->right);

q.pop();

}

return ans;

}

1. Right view of binary tree

vector<int> rightView(Node \*node)

{

vector<int> ans;

queue<Node\*> q;

q.push(node);

ans.push\_back(node->data);

q.push(NULL);

while(q.size()!=1)

{

Node \*temp = q.front();

if(temp==NULL)

{

q.pop();

q.push(NULL);

ans.push\_back(q.front()->data);

continue;

}

if(temp->right!=NULL)

q.push(temp->right);

if(temp->left!=NULL)

q.push(temp->left);

q.pop();

}

return ans;

}

1. Lca in binary tree

Node\* lca(Node\* root ,int n1 ,int n2 )

{

if(!root || root->data==n1 || root->data==n2)

{

return root;

}

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

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

if(left && right)

return root;

else if(!left)

return right;

else

return left;

}

1. Diameter of binary tree

int res = 0;

int sol(Node\* root) {

if(root==NULL)

return 0;

int l=sol(root->left);

int r=sol(root->right);

res=max(res,1+l+r);

return (1+max(l,r));

}

int diameter(Node\* root)

{

sol(root);

return res;

}

1. Vertical width of binary tree

int lmax = 0, rmax = 0;

void cal(Node\* root, int pos)

{

if(root==NULL) return;

lmax = max(pos, lmax);

rmax = max(-pos, rmax);

cal(root->left, pos+1);

cal(root->right, pos-1);

return;

}

int verticalWidth(Node\* root)

{

if(root==NULL) return 0;

lmax = 0, rmax = 0;

cal(root, 0);

return lmax + rmax + 1;

}

1. Mirror tree

void mirror(Node\* node) {

if(node==NULL) return;

Node \*temp = node->left;

node->left = node->right;

node->right = temp;

mirror(node->left);

mirror(node->right);

return;

}

1. Check if subtree

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

{

if(T==NULL && S==NULL)

return true;

else if(T==NULL || S==NULL)

return false;

else if(T->data==S->data)

return (equal(T->left, S->left)&&equal(T->right, S->right));

else

return false;

}

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

{

if(S==NULL) return true;

else if(T==NULL) return false;

if(equal(T, S))

return true;

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

}

1. Make binary tree from linked list

void convert(Node \*head, TreeNode \*&root) {

if(head==NULL) return;

queue<TreeNode\*>q;

Node \*start=head;

root=new TreeNode(start->data);

q.push(root);

while(start->next!=NULL)

{

TreeNode \*temp=q.front();

start=start->next;

temp->left=new TreeNode(start->data);

q.push(temp->left);

if(start->next==NULL)

continue;

start=start->next;

temp->right=new TreeNode(start->data);

q.push(temp->right);

q.pop();

}

}

1. Binary tree to dll

vector<Node\*> res;

void sol(Node \*root)

{

if(root==NULL) return;

sol(root->left);

res.push\_back(root);

sol(root->right);

}

Node \* bToDLL(Node \*root)

{

if(root==NULL) return NULL;

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

res.clear();

sol(root);

int i = 0;

res[i]->left=NULL;

res[i]->right = res[i+1];

for(i = 1; i<res.size()-1; i++)

{

res[i]->left=res[i-1];

res[i]->right = res[i+1];

}

res[i]->left=res[i-1];

res[i]->right = NULL;

return res[0];

}

1. Binary tree to cdll

vector<Node\*> res;

void sol(Node \*root)

{

if(root==NULL) return;

sol(root->left);

res.push\_back(root);

sol(root->right);

}

Node \* bTreeToCList(Node \*root)

{

if(root==NULL) return NULL;

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

res.clear();

sol(root);

int i = 0;

res[i]->left=res[res.size()-1];

res[i]->right = res[i+1];

for(i = 1; i<res.size()-1; i++)

{

res[i]->left=res[i-1];

res[i]->right = res[i+1];

}

res[i]->left=res[i-1];

res[i]->right = res[0];

return res[0];

}

1. Connect nodes at same level

void connect(Node \*root)

{

queue<Node\*> q;

Node \*tail = root;

q.push(root);

q.push(NULL);

while(q.size()!=1)

{

Node \*temp = q.front();

if(temp==NULL)

{

q.pop();

q.push(NULL);

if(q.size()==1) break;

tail = q.front();

continue;

}

if(temp->left!=NULL)

q.push(temp->left);

if(temp->right!=NULL)

q.push(temp->right);

q.pop();

temp->nextRight=q.front();

tail = temp;

}

q.pop();

q.push(NULL);

tail->nextRight=NULL;

}