LINKED LIST+BINARY TREES

LL:

Q1)normal node creation both the node way

```
#include <bits/stdc++.h>
using namespace std;
struct Node{
   int data;
   Node* next;
   Node(int x){
      data=x;
      next=NULL;
   }
};
int main(){
            Node *head=new Node(10);
            Node *temp1=new Node(20);
            Node *temp2=new Node(30);
            head->next=temp1;
            temp1->next=temp2;
            cout<<head->data<<"-->"<<temp1->data<<"-->"<<temp2->data;
            return 0;
}

// second way
#include <bits/stdc++.h>
using namespace std;

struct Node{
   int data;
   Node* next;
   Node(int x){
      data = x;
      next = NULL;
   }
};

void printList(Node* head) {
   Node* temp = head;
   while (temp != NULL) {
      cout << temp->data;
      if (temp->next != NULL){
         cout << "-->";
      }
      temp = temp->next;
   }
   cout << endl;
}

int main()
{
   int n;
   cin >> n;//nodes

   if (n <= 0){
      cout << "List is empty." << endl;
      return 0;
   }

   int value;
   cin >> value;
```

```
    Node *head = new Node(value);
    Node *current = head;

    for (int i = 1; i < n; ++i) {
        cin >> value;
        Node *newNode = new Node(value);
        current->next = newNode;
        current = newNode;
    }

    printList(head);
    return 0;
}
```

Q2)traversing the whole linked list

```
void printlist(Node *head){
    Node *curr=head;
    while(curr!=NULL){
        cout<<curr->data<<" ";
        curr=curr->next;
    }
}
```

Q3)inserting at begin

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node* next;

    // Constructor to initialize a Node
    Node(int x) {
        data = x;
        next = NULL;
    }
};

Node* head = NULL;

void insertAtBegin(int data) {
    // Create a new node with the given data
    Node* newNode = new Node(data);
    newNode->next = head;
    head = newNode;
}

void printList() {
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
}

int main() {
    insertAtBegin(30);
    insertAtBegin(20);
    insertAtBegin(10);

    printList();

    return 0;
}
```

4)insert at the end

```
#include <bits/stdc++.h>
using namespace std;

struct Node{
   int data;
   Node* next;
   Node(int x){
      data=x;
      next=NULL;
   }
};

Node *insertEnd(Node *head,int x){
   Node *temp=new Node(x);//new node created
   if(head==NULL)return temp;
   Node *curr=head;
   while(curr->next!=NULL){
      curr=curr->next;
   }
   curr->next=temp;
   return head;

}

void printlist(Node *head){
      Node *curr=head;
      while(curr!=NULL){
      cout<<curr->data<<" ";
      curr=curr->next;
      }
}
int main()
{
      Node *head=NULL;
      head=insertEnd(head,10);
      head=insertEnd(head,20);
      head=insertEnd(head,30);
      printlist(head);
      return 0;
}
```

5) insert a given position

```
Node *insertAtPos(Node *head, int x, int pos){
   Node *temp=new Node(x);
   if(head==NULL){
      if(pos==1)return temp;
      else return head;
   }
   if(pos==1){
      temp->next=head;
      return temp;
   }
   Node *curr=head;
   for(int i=1;i<pos-1;i++){
      curr=curr->next;
      if(curr==NULL){
         cout<<"Position out of range"<<endl;
         return head;
      }
   }
```

```
    temp->next=curr->next;
    curr->next=temp;
    return head;
}
```

6)Delete first node

```
Node *delHead(Node *head){
    if(head==NULL)return NULL;

        Node *temp=head->next;
        delete(head);
        return temp;

}
```

7)Delete last node

```
Node *delTail(Node *head){
    if(head==NULL)return NULL;
    if(head->next==NULL){
        delete head;
        return NULL;
    }
    Node *curr=head;
    while(curr->next->next!=NULL)
        curr=curr->next;
    delete (curr->next);
    curr->next=NULL;
    return head;
}
```

8) search in singly linkedlist

Normal

```
int search(Node * head, int x){
    int pos=1;
    Node *curr=head;
    while(curr!=NULL){
        if(curr->data==x)
            return pos;
        else{
            pos++;
            curr=curr->next;
        }
    }
    return -1;
}
```

Recusive

```
int search(Node * head, int x){
    if(head==NULL)return -1;
    if(head->data==x)return 1;
    else{
        int res=search(head->next,x);
        if(res==-1)return -1;
        else return res+1;
    }
}
```

9) reverse a singly linked list

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;
        ListNode* next = nullptr;

        while (curr != nullptr) {
            next = curr->next;  // Store the next
node
```

```
        curr->next = prev;  // Reverse the current node's pointer
        prev = curr;       // Move the prev pointer forward
        curr = next;       // Move to the next node in the list
      }
      return prev;  // New head of the reversed list
   }
};
```

DOUBLY LL

10)normal implementation

```
#include <bits/stdc++.h>
using namespace std;

struct Node{
   int data;
   Node* prev;
   Node* next;
   Node(int d){
      data=d;
      prev=NULL;
      next=NULL;
   }
};

void printlist(Node *head){
   Node *curr=head;
   while(curr!=NULL){
      cout<<curr->data<<" ";
      curr=curr->next;
   }
   cout<<endl;
}

int main()
{
        Node *head=new Node(10);
        Node *temp1=new Node(20);
        Node *temp2=new Node(30);
        head->next=temp1;
        temp1->prev=head;
        temp1->next=temp2;
        temp2->prev=temp1;
        printlist(head);
        return 0;
}
```

```
//input method
#include <bits/stdc++.h>
using namespace std;

struct Node {
   int data;
   Node* prev;
   Node* next;
   Node(int d) {
      data = d;
      prev = NULL;
      next = NULL;
   }
};
```

```
void printList(Node* head) {
    Node* curr = head;
    while (curr != NULL) {
        cout << curr->data;
        if (curr->next != NULL) {
            cout << " <-> ";
        }
        curr = curr->next;
    }
    cout << endl;
}

int main() {
    int n;
    cout << "Enter the number of nodes: ";
    cin >> n;

    if (n <= 0) {
        cout << "List is empty." << endl;
        return 0;
    }

    cout << "Enter the values: ";
    int value;
    cin >> value;
    Node* head = new Node(value);
    Node* current = head;

    for (int i = 1; i < n; ++i) {
        cin >> value;
        Node* newNode = new Node(value);
        current->next = newNode;
        newNode->prev = current;
        current = newNode;
    }

    printList(head);
    return 0;
}
```

11)insertion at begin

```
Node *insertbegin(Node *head,int data){
    Node *temp= new Node(data);
    temp->next=head;
    if(head!=NULL){
        head->previous=temp;
    }
    return temp;
}
```

12)Insert at end

```
Node *insertEnd(Node *head,int data){
    Node *temp=new Node(data);
    if(head==NULL)return temp;
    Node *curr=head;
    while(curr->next!=NULL){
        curr=curr->next;
    }
    curr->next=temp;
    temp->prev=curr;
    return head;
```

```
}
```

Q13)Reverse a doubly LL

```
Node *reverse(Node *head){
   Node *temp=NULL;
   Node *curr=head;
   while(curr!=NULL){
      temp=curr->prev;
      curr->prev=curr->next;
      curr->next=temp;
      curr=curr->prev;
   }
   if(temp!=NULL)head=temp->prev;
   return head;
}
```

Q14)Delete head

```
Node *delHead(Node *head){
   if(head==NULL)return NULL;
   if(head->next==NULL){
      delete head;
      return NULL;
   }
   else{
      Node *temp=head;
      head=head->next;
      head->prev=NULL;
      delete temp;
      return head;
   }
}
```

15)delete the last in double LL

```
Node *delLast(Node *head){
   if(head==NULL)return NULL;
   if(head->next==NULL){
      delete head;
      return NULL;
   }
   Node *curr=head;
   while(curr->next!=NULL)
      curr=curr->next;
   curr->prev->next=NULL;
   delete curr;
   return head;
}
```

TREES

Q1) Tree linked representation

```
#include <bits/stdc++.h>
using namespace std;
static Node* prev = NULL;
struct Node {
 int key;
Node *left;
Node *right;
 Node(int k){
    key=k;
    left=right=NULL;
 }
};

int main() {

        Node *root=new Node(10);
        root->left=new Node(5);
        root->right=new Node(20);
        root->right->left=new Node(30);
        root->right->right=new Node(35);

        Node *head=BTToDLL(root);
}
```

Q2) build a tree

You can access the whole tree by the node

```
Node* buildTree() {
   int key;
   cout << "Enter the value for the root node:
";
   cin >> key;

   Node* root = new Node(key);
   queue<Node*> q;
   q.push(root);

   while (!q.empty()) {
      Node* current = q.front();
      q.pop();

      cout << "Enter left child of " << current-
>key << " (or -1 for no left child): ";
      cin >> key;
      if (key != -1) {
         current->left = new Node(key);
         q.push(current->left);
      }

      cout << "Enter right child of " << current-
>key << " (or -1 for no right child): ";
      cin >> key;
      if (key != -1) {
         current->right = new Node(key);
         q.push(current->right);
      }
   }

   return root;
}

int main() {
   Node* root = buildTree();
```

```
    cout << "Inorder traversal of the binary tree is: ";
    inorder(root);
    cout << endl;

    return 0;
}
```

Q3)Inorder traversal

```
void inorder(Node *root){
    if(root!=NULL){
        inorder(root->left);
        cout<<root->key<<" ";
        inorder(root->right);
    }
}
```

Q4)Preorder traversal

```
void preorder(Node *root){
    if(root!=NULL){
        cout<<root->key<<" ";
        preorder(root->left);
        preorder(root->right);
    }
}
```

Q5) Post order traversal

```
class Solution {
public:
    void postOrder(TreeNode* root, vector<int> &ans) {
        if(!root) return;
        postOrder(root->left, ans);
        postOrder(root->right, ans);
        ans.push_back(root->val);
    }
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> ans;
        postOrder(root, ans);
        return ans;
    }
};
```

Q6)

https://leetcode.com/problems/maximum-depth-of-binary-tree/description/

height of the binary tree

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(root==NULL){
            return 0;
        }
        int left=1+maxDepth(root->left);//remember after recursion call is finished 1 is added
        int right=1+maxDepth(root->right);
        return max(left,right);
    }
};
```

Q7)Print nodes at a distance k

```
void printKDist(Node *root,int k){
    if(root==NULL)return;

    if(k==0){cout<<root->key<<" ";}
    else{
    printKDist(root->left,k-1);
```

```
    printKDist(root->right,k-1);
    }
}
```

Q8)LEVEL order traversal

Note-level order traversal means (BFS)

And DFS includes recursion in the form of inorder,preorder,postorder traversal

```
void printLevel(Node *root){
    if(root==NULL)return;
    queue<Node *>q;
    q.push(root);
    while(q.empty()==false){
        Node *curr=q.front();
        q.pop();
        cout<<curr->key<<" ";
        if(curr->left!=NULL)
            q.push(curr->left);
        if(curr->right!=NULL)
            q.push(curr->right);
    }
}
```

Q9)level order traversal -II

```
void printLevel(Node *root){
    if(root==NULL)return;
    queue<Node *>q;
    q.push(root);
    q.push(NULL);
    while(q.size()>1){
        Node *curr=q.front();
        q.pop();
        if(curr==NULL){
            cout<<"\n";
            q.push(NULL);
            continue;
        }
        cout<<curr->key<<" ";
        if(curr->left!=NULL)
            q.push(curr->left);
        if(curr->right!=NULL)
            q.push(curr->right);
    }
}
```

Q10)size of the binary tree

basically you have to count all the nodes in a binary tree

//note how this code returns 2 in 3 t member tree

```
int getSize(Node *root){
    if(root==NULL)
        return 0;
    else
        int left=1+getSize(root->left);
        int right=1+getSize(root->right);
        return left+right;
}
```

//corrected code

```
int getSize(Node *root) {
    if (root == NULL)
        return 0;

        int left = getSize(root->left);
        int right = getSize(root->right);
```

```
        return 1 + left + right;
    }
}
```

Q11)left view of the binary tree

```
// DFS

int maxLevel=0;
void printLeft(Node *root,int level){
    if(root==NULL)
        return;
    if(maxLevel<level){
        cout<<root->key<<" ";
        maxLevel=level;
    }
    printLeft(root->left,level+1);
    printLeft(root->right,level+1);
}

void printLeftView(Node *root){
    printLeft(root,1);
}

// BFS

void printLeft(Node *root){
    if(root==NULL)
        return;
        queue<Node *> q;q.push(root);
    while(q.empty()==false){
        int count=q.size();
        for(int i=0;i<count;i++){
            Node *curr=q.front();
            q.pop();
            if(i==0)
                cout<<curr->key<<" ";
            if(curr->left!=NULL)
                q.push(curr->left);
            if(curr->right!=NULL)
                q.push(curr->right);
        }
    }
}
```

https://leetcode.com/problems/binary-tree-right-side-view/

question based on it

Q12)children sum property

```
bool isCSum(Node *root){
    if(root==NULL)
        return true;
    if(root->left==NULL && root->right==NULL)
        return true;//leaf nodes
    int sum=0;
    if(root->left!=NULL)
    sum+=root->left->key;
    if(root->right!=NULL)
    sum+=root->right->key;
    //The use of && in the return statement of the isCSum function ensures that all conditions must be met for the function to return true.
    return (root->key==sum && isCSum(root->left) && isCSum(root->right));
}
```

Q13)

https://leetcode.com/problems/balanced-binary-tree/

Naïve

```
int height(Node *root){
   if(root==NULL)
      return 0;
   else
      return (1+max(height(root->left),height(root->right)));
}

bool isBalanced(Node *root){
   if(root==NULL)
      return true;
   int lh=height(root->left);
   int rh=height(root->right);
   return (abs(lh-rh)<=1 && isBalanced(root->left) && isBalanced(root->right));
}
```

pro

```
class Solution {
public:
   int height(TreeNode* root) {
      if (root == NULL) {
         return 0;
      }
      int leftHeight = height(root->left);
      if (leftHeight == -1) return -1;
      int rightHeight = height(root->right);
      if (rightHeight == -1) return -1;
      if (abs(leftHeight - rightHeight) > 1) return -1;
      return max(leftHeight, rightHeight) + 1;
   }

   bool isBalanced(TreeNode* root) {
      return height(root) != -1;//if -1 then false;
   }
};
```

Q14)

https://leetcode.com/problems/maximum-width-of-binary-tree/

//through BFS

```
class Solution {
public:
   typedef unsigned long long ll;
   int widthOfBinaryTree(TreeNode* root) {
      if(!root)
         return 0;
      queue<pair<TreeNode*, ll>> que;
      que.push({root, 0});
      ll maxWidth = 0;

      while(!que.empty()){
         int n = que.size();
         ll f = que.front().second;
         ll l = que.back().second;
         maxWidth = max(maxWidth, l-f+1);

         while(n--) {
            TreeNode* curr = que.front().first;
            ll d       = que.front().second;
```

```
            que.pop();
            if(curr->left) {
               que.push({curr->left, 2*d+1});
            }
            if(curr->right) {
               que.push({curr->right, 2*d+2});
            }
         }
      }
      return maxWidth;
   }
};
```

```
int maxWidth(Node *root){
   if(root==NULL)return 0;
   queue<Node *>q;
   q.push(root);
   int res=0;
   while(q.empty()==false){
      int count=q.size();
      res=max(res,count);
      for(int i=0;i<count;i++){
         Node *curr=q.front();
         q.pop();
         if(curr->left!=NULL)
            q.push(curr->left);
         if(curr->right!=NULL)
            q.push(curr->right);
      }
   }
   return res;
}
```

Q15) convert binary tree into doubly LL

```
void printlist(Node *head){
   Node *curr=head;
   while(curr!=NULL){
      cout<<curr->key<<" ";
      curr=curr->right;
   }cout<<endl;
}
```

```
Node *BTToDLL(Node *root){
   if(root==NULL)return root;
   Node *head=BTToDLL(root->left);
   if(prev==NULL){head=root;}
   else{
      root->left=prev;
      prev->right=root;
   }
   prev=root;
   BTToDLL(root->right);
   return head;
}
```

Q16) max value in a node

```
int getMax(Node *root){
   if(root==NULL)
      return INT_MIN;
```

```
    return max(root->key,max(getMax(root->left),getMax(root->right)));
}
```

Q17)

https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/

```
class Solution {
public:
  int preindex=0;
  TreeNode* contree(vector<int>& preorder, vector<int>& inorder,int is,int ie){
    if(is>ie){
     return NULL;
    }
    int inindex;
    TreeNode*root=new TreeNode(preorder[preindex++]);
    for(int i=is;i<=ie;i++){
        if(inorder[i]==root->val){
          inindex=i;
          break;
        }
    }
    root->left=contree(preorder,inorder,is,inindex-1);
    root->right=contree(preorder,inorder,inindex+1,ie);
    return root;
  }

  TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    int i=0;
    int n=preorder.size()-1;
    return contree(preorder,inorder,i,n);
  }
};
```

Q18)

https://leetcode.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/

```
class Solution {
public:
  int postindex; // Index for postorder traversal

  TreeNode* contree(vector<int>& inorder, vector<int>& postorder, int is, int ie) {
    if (is > ie) {
      return nullptr;
    }

    TreeNode* root = new TreeNode(postorder[postindex--]);

    int inindex;
    for (inindex = is; inindex <= ie; ++inindex) {
      if (inorder[inindex] == root->val) {
        break;
      }
    }

    root->right = contree(inorder, postorder, inindex + 1, ie);
```

```
        root->left = contree(inorder, postorder,
is, inindex - 1);


        return root;
    }


    TreeNode* buildTree(vector<int>& inorder,
vector<int>& postorder) {
        postindex = postorder.size() - 1;
        return contree(inorder, postorder, 0,
inorder.size() - 1);
    }
};
```

Q19)
https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/description/

```
void printSpiral(Node *root){
    if(root==NULL)return;
    queue<Node *>q;
    stack<int> s;
    bool reverse=false;
    q.push(root);
    while(q.empty()==false){
        int count=q.size();
        for(int i=0;i<count;i++){
        Node *curr=q.front();
        q.pop();
        if(reverse)
            s.push(curr->key);
        else
            cout<<curr->key<<" ";
        if(curr->left!=NULL)
            q.push(curr->left);
        if(curr->right!=NULL)
            q.push(curr->right);
        }
        if(reverse){
            while(s.empty()==false){
                cout<<s.top()<<" ";
                s.pop();
            }
        }
    reverse=!reverse;
    }
}
```

METHOD-2

Two stack method efficient try yourself

Q20)
https://leetcode.com/problems/diameter-of-binary-tree/

```
class Solution {
public:
    int res=0;
    int height(TreeNode*root){
        if(root==NULL){
            return 0;
        }
        int lh=height(root->left);
        int rh=height(root->right);
        res=max(res,lh+rh);
        return 1+max(lh,rh);
    }
    int diameterOfBinaryTree(TreeNode* root)
{
        res=0;
```

```
    height(root);
    return res;
  }
}
```

Q21)
https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/

```
class Solution {
public:
  TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if(!root)
      return NULL;

    if(root->val == p->val || root->val == q->val)
      return root;

    TreeNode* l = lowestCommonAncestor(root->left, p, q);
    TreeNode* r = lowestCommonAncestor(root->right, p, q);

    if(l && r)//better way in notes
      return root;

    return l?l:r;
  }
};
```

Q22) **https://leetcode.com/problems/add-one-row-to-tree/description/**

```
class Solution {
public:

  TreeNode* add(TreeNode* root, int val, int depth, int curr) {
    if(!root)
      return NULL;

    if(curr == depth-1) {
      TreeNode* lTemp = root->left;
      TreeNode* rTemp = root->right;

      root->left  = new TreeNode(val);
      root->right = new TreeNode(val);
      root->left->left = lTemp;
      root->right->right = rTemp;

      return root;
    }

    root->left  = add(root->left, val, depth, curr+1);
    root->right = add(root->right, val, depth, curr+1);

    return root;
  }

  TreeNode* addOneRow(TreeNode* root, int val, int depth) {
    if(depth == 1) {
      TreeNode* newRoot = new TreeNode(val);
```

```
        newRoot->left = root;
        return newRoot;
      }

      return add(root, val, depth, 1);
    }
};
```

Q23)
https://leetcode.com/problems/maximum-difference-between-node-and-ancestor/

Naïve: see copy

Pro:

```
class Solution {
public:

  int findMaxDiff(TreeNode* root, int minV, int maxV) {
      if(!root)
        return abs(minV-maxV);

      minV = min(root->val, minV);
      maxV = max(root->val, maxV);

      int l = findMaxDiff(root->left,  minV, maxV);
      int r = findMaxDiff(root->right, minV, maxV);

      return max(l, r);

  }

  int maxAncestorDiff(TreeNode* root) {
      int minV = root->val;
      int maxV = root->val;

      return findMaxDiff(root, minV, maxV);

  }
};
```

Q24)
https://leetcode.com/problems/serialize-and-deserialize-binary-tree/

```
class Codec {
public:

  string serialize(TreeNode* root) {
      string result;
      serializeHelper(root, result);
      return result;
  }

  void serializeHelper(TreeNode* root, string &result) {
      if (root == NULL) {
        result += "null,";
        return;
      }
      result += to_string(root->val) + ",";
      serializeHelper(root->left, result);
      serializeHelper(root->right, result);
  }
```

```
TreeNode* deserialize(string data) {
    int index = 0;
    return deserializeHelper(data, index);
}

TreeNode* deserializeHelper(const string &data, int &index) {
    if (index >= data.size()) return NULL;

    int nextIndex = data.find(',', index);
    string val = data.substr(index, nextIndex - index);
    index = nextIndex + 1;

    if (val == "null") return NULL;

    TreeNode* root = new TreeNode(stoi(val));
    root->left = deserializeHelper(data, index);
    root->right = deserializeHelper(data, index);

    return root;
  }
};
```

Q25) in order traversal using BFS

```
void inorderTraversal(TreeNode* root) { stack<TreeNode*> stk; TreeNode* curr = root; while (curr != NULL || !stk.empty()) { // Reach the leftmost node of the current node while (curr != NULL) { stk.push(curr); curr = curr->left; } // Current must be NULL at this point curr = stk.top(); stk.pop(); cout << curr->val << " "; // Visit the node // Visit the right subtree curr = curr->right; } }
```

Q26)

Q27) https://leetcode.com/problems/same-tree/description/

```
class Solution {
public:
  bool isSameTree(TreeNode* p, TreeNode* q) {
    if(!p && !q)
      return true; // both are null, return true

    if(p == NULL || q == NULL)
      return false; // any one of them is null, return false

    if(p->val != q->val)
      return false; // values are different, return false

    return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
  }
};
```

Q28) https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/description/

Q29) https://leetcode.com/problems/find-duplicate-subtrees/

```
class Solution {
public:
```

```
    string DFS(TreeNode* root, unordered_map<string, int>& mp, vector<TreeNode*>& res){
        if(root == NULL)
            return "NULL";

        string s = to_string(root->val) + "," + DFS(root->left, mp, res) + "," + DFS(root->right, mp, res);

        if(mp[s] == 1)
            res.push_back(root);

        mp[s]++;

        return s;
    }

    vector<TreeNode*> findDuplicateSubtrees(TreeNode* root) {
        unordered_map<string, int> mp;

        vector<TreeNode*>res;

        DFS(root, mp, res);

        return res;
    }

};
```

Q30) **https://leetcode.com/problems/symmetric-tree/**

```
class Solution {
public:

    bool check(TreeNode* l, TreeNode* r) {
        if(l == NULL && r == NULL)
            return true;
        if(l == NULL || r == NULL)
            return false;

        if(l->val == r->val && check(l->left, r->right) && check(l->right, r->left))
            return true;

        return false;
    }

    bool isSymmetric(TreeNode* root) {
        if(!root)
            return true;

        return check(root->left, root->right);
    }
};
```

Q31) **https://leetcode.com/problems/check-completeness-of-a-binary-tree/description/**

```
class Solution {
public:
    int countNodes(TreeNode* root) {
```

```
    if(root == NULL)
        return 0;

    return 1 + countNodes(root->left) + countNodes(root->right);
  }

  bool dfs(TreeNode* root, int i, int totalNodes) {
    if(root == NULL)
      return true;

    if(i > totalNodes)
      return false;

    return dfs(root->left, 2*i, totalNodes) &&
        dfs(root->right, 2*i + 1, totalNodes);
  }

  //Using DFS
  bool isCompleteTree(TreeNode* root) {
    int totalNodes = countNodes(root);

    int i = 1;
    return dfs(root, i, totalNodes);
  }
};
```

Q32) https://leetcode.com/problems/delete-nodes-and-return-forest/

```
class Solution {
public:
  TreeNode* deleteHelper(TreeNode* root, unordered_set<int>& st, vector<TreeNode*>& result) {
    if(root == NULL)
      return NULL;

    root->left  = deleteHelper(root->left, st, result);
    root->right = deleteHelper(root->right, st, result);

    if(st.find(root->val) != st.end()) { //if I have to delete this root, then put root->left and root->right in result
      if(root->left != NULL)
        result.push_back(root->left);

      if(root->right != NULL)
        result.push_back(root->right);

      return NULL;
    } else {
      return root;
    }
  }

  vector<TreeNode*> delNodes(TreeNode* root, vector<int>& to_delete) {
    vector<TreeNode*> result;

    unordered_set<int> st;

    for(int &num : to_delete) {
```

```
            st.insert(num);
        }

        deleteHelper(root, st, result); // <-- it will not consider root

        //So, check here if root is to be deleted or not
        if(st.find(root->val) == st.end()) {
            result.push_back(root);
        }

        return result;
    }
};
```

Q33)

Naïve:
https://leetcode.com/problems/binary-tree-pruning/

```
class Solution {
public:

    bool checkOne(TreeNode* root) {
        if(!root)
            return false;

        if(root->val == 1)
            return true;

        return checkOne(root->left) || checkOne(root->right);//checking for both subtrees
    }

    TreeNode* pruneTree(TreeNode* root) {
        if(!root)
            return NULL;

        pruneTree(root->left);
        pruneTree(root->right);

        if(!checkOne(root->left))  root->left = NULL;
        if(!checkOne(root->right)) root->right = NULL;

        if(!root->left && !root->right && root->val == 0)
            return NULL;
        return root;
    }
};
```

Pro:

Using bottom up

```
class Solution {
public:
    TreeNode* pruneTree(TreeNode* root) {
        if(!root)
            return NULL;

        root->left  = pruneTree(root->left);
        root->right = pruneTree(root->right);
```

```
    if(!root->left && !root->right && root->val == 0)
        return NULL;

    return root;
  }
};
```

Q34) https://leetcode.com/problems/delete-leaves-with-a-given-value/
similar question

```
class Solution {
public:
  TreeNode* removeLeafNodes(TreeNode* root, int target) {
    if(!root)
      return NULL;

    // Recursively process the left and right subtrees
    root->left = removeLeafNodes(root->left, target);
    root->right = removeLeafNodes(root->right, target);

    // If the current node is a leaf and its value is equal to the target, remove it
    if(!root->left && !root->right && root->val == target)
      return NULL;

    return root;
  }
};
```

Q35) https://leetcode.com/problems/path-sum-ii/

```
class Solution {
public:
  void collectPaths(TreeNode* root, int curr, vector<int>& temp, vector<vector<int>>& result) {
    if(!root)
      return;
    temp.push_back(root->val);
    if(root->left == NULL && root->right == NULL && root->val == curr) {
      result.push_back(temp);
    }

    collectPaths(root->left, curr-root->val, temp, result);
    collectPaths(root->right, curr-root->val, temp, result);
    temp.pop_back();
  }

  vector<vector<int>> pathSum(TreeNode* root, int sum) {
    vector<vector<int>> result;

    vector<int> temp;
    collectPaths(root, sum, temp, result);
    return result;
  }
};
```

Q36) https://leetcode.com/problems/path-sum/

```
class Solution {
```

```
public:
   bool pathSum(TreeNode* root, int sum, int
curr) {
      if(!root)
         return false;

      if(!root->left && !root->right)
         return ((curr+root->val) == sum);

      bool l = pathSum(root->left,  sum,
curr+root->val);
      bool r = pathSum(root->right, sum,
curr+root->val);
      return l||r;
   }
   bool hasPathSum(TreeNode* root, int sum)
{
      return pathSum(root, sum, 0);
   }
};
```

BINARY SEARCH TREE(BST)

Q1) https://leetcode.com/problems/search-in-a-binary-search-tree/

```
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if (root == nullptr || root->val == val) {
            return root;
        }

        // If the value is less than the root's value, search the left subtree
        if (val < root->val) {
            return searchBST(root->left, val);
        }

        // Otherwise, search the right subtree
        return searchBST(root->right, val);
    }
};
```

Iterative:

```
bool search(Node *root, int x){
    while(root!=NULL){
        if(root->key==x)
            return true;
        else if(root->key<x)
            root=root->right;
        else
            root=root->left;
    }
    return false;
}
```

Q2) https://leetcode.com/problems/insert-into-a-binary-search-tree/description/

```
class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int key) {
        if (root == nullptr)
            return root;

        if (key < root->val) {
            root->left = deleteNode(root->left, key);
        } else if (key > root->val) {
            root->right = deleteNode(root->right, key);
        } else {

            if (root->left == nullptr) {
                TreeNode* temp = root->right;
                delete root;
                return temp;
            } else if (root->right == nullptr) {
                TreeNode* temp = root->left;
                delete root;
                return temp;
            }

            TreeNode* temp = minValueNode(root->right);

            root->val = temp->val;
```

```
        root->right = deleteNode(root->right,
temp->val);
    }

    return root;
  }

private:
  TreeNode* minValueNode(TreeNode*
node) {
    TreeNode* current = node;
    while (current && current->left != nullptr)
      current = current->left;
    return current;
  }
};
```

Q3) https://leetcode.com/problems/closest-nodes-queries-in-a-binary-search-tree/
floor and ceil value

```
class Solution {
public:
  vector<vector<int>>
closestNodes(TreeNode* root, vector<int>&
queries) {
    vector<vector<int>> result;

    for (int x : queries) {
      TreeNode* floorNode = floor(root, x);
      TreeNode* ceilNode = ceil(root, x);

      int floorVal = (floorNode != nullptr) ?
floorNode->val : -1;
      int ceilVal = (ceilNode != nullptr) ?
ceilNode->val : -1;
      result.push_back({floorVal, ceilVal});
    }

    return result;
  }

private:
  TreeNode* floor(TreeNode* root, int x) {
    TreeNode* res = nullptr;
    while (root != nullptr) {
      if (root->val == x)
        return root;
      else if (root->val > x)
        root = root->left;
      else {
        res = root;
        root = root->right;
      }
    }
    return res;
  }

  TreeNode* ceil(TreeNode* root, int x) {
    TreeNode* res = nullptr;
    while (root != nullptr) {
      if (root->val == x)
        return root;
      else if (root->val < x)
        root = root->right;
      else {
```

```
            res = root;
            root = root->left;
          }
        }
        return res;
    }
};
```

Q4)
https://leetcode.com/problems/balance-a-binary-search-tree/description/

```
class Solution {
public:
    void inorderTraversal(TreeNode* root, vector<int>& nodes) {
        if (root == nullptr) return;
        inorderTraversal(root->left, nodes);
        nodes.push_back(root->val);
        inorderTraversal(root->right, nodes);
    }

    TreeNode* buildBalancedBST(const vector<int>& nodes, int start, int end) {
        if (start > end) return nullptr;

        int mid = start + (end - start) / 2;
        TreeNode* root = new TreeNode(nodes[mid]);

        root->left = buildBalancedBST(nodes, start, mid - 1);
        root->right = buildBalancedBST(nodes, mid + 1, end);
        return root;
    }
    TreeNode* balanceBST(TreeNode* root) {
        vector<int> sortedNodes;
        inorderTraversal(root, sortedNodes);
        return buildBalancedBST(sortedNodes, 0, sortedNodes.size() - 1);
    }

};
```

Q5)
https://leetcode.com/problems/convert-sorted-list-to-binary-search-tree/description/

```
class Solution {
public:
    TreeNode* buildBST(ListNode* start, ListNode* end) {
        if (start == end) return nullptr;

        ListNode* slow = start;
        ListNode* fast = start;

        while (fast != end && fast->next != end) {
            slow = slow->next;
            fast = fast->next->next;
        }

        TreeNode* node = new TreeNode(slow->val);
        node->left = buildBST(start, slow);
        node->right = buildBST(slow->next, end);
```

```
        return node;
    }
    TreeNode* sortedListToBST(ListNode* head) {
        if (head==NULL)
        return NULL;
        return buildBST(head, nullptr);
    }

};
```

AVL TREE (try)

Q6) https://leetcode.com/problems/kth-smallest-element-in-a-bst/

```
class Solution {
public:
 void inorder(TreeNode* node, int k, int& count, int& result) {
        if (node==NULL)
        return;

        inorder(node->left, k, count, result);
        count++;
        if (count == k) {
            result = node->val;
            return;
        }

        inorder(node->right, k, count, result);
    }
    int kthSmallest(TreeNode* root, int k) {
        int count = 0;
        int result = -1;
        inorder(root, k, count, result);
        return result;
    }

};
```

Pro approach in the copy

Q7) https://leetcode.com/problems/validate-binary-search-tree/

```
class Solution {
    public:
    bool isBST(TreeNode* node, TreeNode*& prevv) {
        if (node == NULL) {
            return true;
        }

        if (isBST(node->left, prevv)==false) {
            return false;
        }//for input 1,1

        if (prevv != nullptr && node->val <= prevv->val) {
            return false;
        }
        prevv = node;

        return isBST(node->right, prevv);
    }
    bool isValidBST(TreeNode* root) {
        TreeNode* prevv = nullptr;
```

```
        return isBST(root, prevv);
    }
};
```

Q8) https://leetcode.com/problems/recover-binary-search-tree/

```
class Solution {
public:
    TreeNode* prevv = nullptr;
    TreeNode* first = nullptr;
    TreeNode* second = nullptr;

    void fixBST(TreeNode* root) {
        if (root == nullptr) return;

        fixBST(root->left);

        if (prevv != nullptr && root->val < prevv->val) {
            if (first == nullptr) {
                first = prevv;
            }
            second = root;
        }
        prevv = root;

        fixBST(root->right);
    }
    void recoverTree(TreeNode* root) {
        prevv = nullptr;
        first = nullptr;
        second = nullptr;
        fixBST(root);

        if (first && second) {
            swap(first->val, second->val);
        }
    }
};
```

Q9) https://leetcode.com/problems/two-sum-iv-input-is-a-bst/description/

```
class Solution {
    bool find(TreeNode* node, int k, unordered_set<int>& set) {
        if (node == nullptr) {
            return false;
        }

        if (set.count(k - node->val)) {
            return true;
        }

        set.insert(node->val);

        return find(node->left, k, set) || find(node->right, k, set);
    }
public:
    bool findTarget(TreeNode* root, int k) {
        unordered_set<int> set;
        return find(root, k, set);
    }
```

```
};
```

Q10)
https://leetcode.com/problems/maximum-level-sum-of-a-binary-tree/

```
class Solution {
public:
    int maxLevelSum(TreeNode* root) {
        if (root == nullptr) return 0;

        queue<TreeNode*> q;
        q.push(root);

        int maxSum = INT_MIN;
        int maxLevel = 1;
        int currentLevel = 1;

        while (!q.empty()) {
            int levelSum = 0;
            int size = q.size();

            for (int i = 0; i < size; ++i) {
                TreeNode* node = q.front();
                q.pop();
                levelSum += node->val;

                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }

            if (levelSum > maxSum) {
                maxSum = levelSum;
                maxLevel = currentLevel;
            }

            currentLevel++;
        }

        return maxLevel;
    }
};
```

Q11)vertical sum in binary tree

```
void vSumR(Node *root,int hd,map<int,int> &mp){
    if(root==NULL)return;
    vSumR(root->left,hd-1,mp);
    mp[hd]+=root->key;
    vSumR(root->right,hd+1,mp);
}
```

Q12)
https://leetcode.com/problems/vertical-order-traversal-of-a-binary-tree/

```
class Solution {
public:
    vector<vector<int>> verticalTraversal(TreeNode* root) {
        map<int, map<int, vector<int>>> nodes;  // Maps column -> (row -> vector of values)
        queue<pair<TreeNode*, pair<int, int>>> q; // Queue for BFS: node -> (column, row)
        q.push({root, {0, 0}}); // Initialize with root at column 0, row 0

        while (!q.empty()) {
```

```
        auto p = q.front();
        q.pop();
        TreeNode* node = p.first;
        int x = p.second.first, y =
p.second.second;


        nodes[x][y].push_back(node->val); //
Insert node value in the correct position


        if (node->left)
            q.push({node->left, {x - 1, y + 1}}); //
Left child: x - 1, y + 1
        if (node->right)
            q.push({node->right, {x + 1, y + 1}});
// Right child: x + 1, y + 1
      }


      vector<vector<int>> result;
      for (auto p : nodes) { // Traverse columns
in sorted order
        vector<int> col;
        for (auto q : p.second) { // Traverse
rows in sorted order
            sort(q.second.begin(),
q.second.end()); // Sort vector of values
            col.insert(col.end(), q.second.begin(),
q.second.end());
        }
        result.push_back(col); // Add the
column to the result
      }
      return result;
   }
};
```