Binary Search Trees

Declaration :

```
struct BinarySearchTreeNode {
    int data;
    struct BinarySearchTreeNode *left;
    struct BinarySearchTreeNode *right;
}
```

Finding an element in BST :

Recursive :

```
struct BinarySearchTreeNode* Find(BinarySearchTreeNode *root, int
data) {
    if (root == nullptr) {
        return nullptr;
    }

    if (data < root->data) {
        return Find(root->left, data);
    } else if (data > root->data) {
        return Find(root->right, data);
    }

    return root;
}
```

Non Recursive :

```
BinarySearchTreeNode* Find(BinarySearchTreeNode *root, int data) {
    while (root != nullptr) {
        if (data == root->data) {
            return root;
        } else if (data > root->data) {
            root = root->right;
        } else {
            root = root->left;
        }
    }
    return nullptr;
}
```

Finding minimum element in BST :

Recursive :

```
BinarySearchTreeNode* FindMin(BinarySearchTreeNode *root) {
    if (root == nullptr) {
        return nullptr;
```

```
    } else if (root->left == nullptr) {
        return root;
    } else {
        return FindMin(root->left);
    }
}
```

Non Recursive :

```
BinarySearchTreeNode* FindMin(BinarySearchTreeNode *root) {
    if (root == nullptr) {
        return nullptr;
    }
    while (root->left != nullptr) {
        root = root->left;
    }
    return root;
}
```

Finding maximum element in BST :

Recursive :

```
BinarySearchTreeNode* FindMax(BinarySearchTreeNode* root) {
    if (root == NULL)
        return NULL;
    else if (root->right == NULL)
        return root;
    else
        return FindMax(root->right);
}
```

Non Recursive :

```
BinarySearchTreeNode* FindMax(BinarySearchTreeNode* root) {
    if (root == NULL)
        return NULL;
    while (root->right != NULL)
        root = root->right;
    return root;
}
```

Insertion :

```
BinarySearchTreeNode* Insert(BinarySearchTreeNode* root, int data) {
    if (root == NULL) {
        root = new BinarySearchTreeNode();
        if (root == NULL) {
            cout << "Memory Error" << endl;
            return NULL;
```

```
        } else {
            root->data = data;
            root->left = root->right = NULL;
        }
    } else {
        if (data < root->data) {
            root->left = Insert(root->left, data);
        } else if (data > root->data) {
            root->right = Insert(root->right, data);
        }
    }
    return root;
}
```

Deletion :

```
BinarySearchTreeNode* Delete(BinarySearchTreeNode* root, int data) {
    if (root == NULL) {
        cout << "Element not there in tree" << endl;
        return NULL;
    }
    if (data < root->data) {
        root->left = Delete(root->left, data);
    } else if (data > root->data) {
        root->right = Delete(root->right, data);
    } else {
        // Found the element
        if (root->left && root->right) {
            // Replace with the largest in the left subtree
            BinarySearchTreeNode* temp = FindMax(root->left);
            root->data = temp->data;
            root->left = Delete(root->left, root->data);
        } else {
            // One child or no child
            BinarySearchTreeNode* temp = root;
            if (root->left == NULL)
                root = root->right;
            else if (root->right == NULL)
                root = root->left;
            delete temp;
        }
    }
    return root;
}
```

Given pointers to two nodes in a BST, find the LCA

```
BinarySearchTreeNode* FindLCA(BinarySearchTreeNode* root,
BinarySearchTreeNode* a, BinarySearchTreeNode* b) {
    while (1) {
        if ((a->data < root->data && b->data > root->data) ||
            (a->data > root->data && b->data < root->data)) {
            return root;
        }

        if (a->data < root->data)
            root = root->left;
        else
            root = root->right;
    }
}
```

Checking whether a given tree is a BST or not

Checking at one node :

```
int IsBST(BinaryTreeNode* root) {
    if (root == NULL)
        return 1;

    // Return 0 if left child is greater than root
    if (root->left != NULL && root->left->data > root->data)
        return 0;

    // Return 0 if right child is less than root
    if (root->right != NULL && root->right->data < root->data)
        return 0;

    // Recursively check for left and right subtrees
    if (!IsBST(root->left) || !IsBST(root->right))
        return 0;

    // All checks passed, it's a BST
    return 1;
}
```

Complete code :

```
int IsBST(BinaryTreeNode* root) {
    if (root == NULL)
        return 1;

    // Check if the max value in the left subtree is greater than the
root
```

```
    if (root->left != NULL && FindMax(root->left)->data > root->data)
        return 0;

    // Check if the min value in the right subtree is less than or
equal to the root
    if (root->right != NULL && FindMin(root->right)->data <=
root->data)
        return 0;

    // Recursively check the left and right subtrees
    if (!IsBST(root->left) || !IsBST(root->right))
        return 0;

    // If all checks pass, the tree is a BST
    return 1;
}

// Helper functions to find the maximum and minimum node
BinaryTreeNode* FindMax(BinaryTreeNode* root) {
    while (root && root->right != NULL)
        root = root->right;
    return root;
}

BinaryTreeNode* FindMin(BinaryTreeNode* root) {
    while (root && root->left != NULL)
        root = root->left;
    return root;
}
```

O(n2), O(n)

Improving complexity :

```
bool IsBSTUtil(BinaryTreeNode* root, int& prev) {
    if (!root) return true;

    if (!IsBSTUtil(root->left, prev)) return false;

    if (root->data <= prev) return false;

    prev = root->data;

    return IsBSTUtil(root->right, prev);
}

bool IsBST(BinaryTreeNode* root) {
```

```
    int prev = INT_MIN;
    return IsBSTUtil(root, prev);
}
```

O(n), O(n)

Convert BST into circular DLL with space complexity O(1)

```
BinaryTreeNode* BST2DLL(BinaryTreeNode* root, BinaryTreeNode** ltail)
{
    if (!root) {
        *ltail = nullptr;
        return nullptr;
    }

    BinaryTreeNode *left, *right, *rtail;
    left = BST2DLL(root->left, ltail);
    right = BST2DLL(root->right, &rtail);

    root->left = *ltail;
    root->right = right;

    if (right) {
        *ltail = root;
    } else {
        right->left = root;
        *ltail = rtail;
    }

    if (!left) {
        return root;
    } else {
        (*ltail)->right = root;
        return left;
    }
}
```

Another method :

```
BinaryTreeNode* Append(BinaryTreeNode* a, BinaryTreeNode* b,
BinaryTreeNode** aLast, BinaryTreeNode** bLast) {
    if (a == nullptr) return b;
    if (b == nullptr) return a;

    *aLast = a->left;
    *bLast = b->left;

    (*aLast)->right = b;
```

```
    b->left = *aLast;
    (*bLast)->right = a;
    a->left = *bLast;

    return a;
}

BinaryTreeNode* TreeToList(BinaryTreeNode* root) {
    if (root == nullptr) return nullptr;

    BinaryTreeNode *aList, *bList;

    aList = TreeToList(root->left);
    bList = TreeToList(root->right);

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

    aList = Append(aList, root, &aList, &root);
    aList = Append(aList, bList, &aList, &bList);

    return aList;
}
```

Given a sorted DLL, give an algorithm for converting it into balanced BST

```
DLLNode* FindMiddleNode(DLLNode* head) {
    DLLNode* slow = head;
    DLLNode* fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}

DLLNode* DLLtoBalancedBST(DLLNode* head) {
    if (!head || !head->next)
        return head;
    DLLNode* temp = FindMiddleNode(head);
    DLLNode* p = head;
    DLLNode* q;
    while (p->next != temp)
        p = p->next;
    p->next = nullptr;
```

```
    q = temp->next;
    temp->next = nullptr;

    temp->prev = DLLtoBalancedBST(head);
    temp->next = DLLtoBalancedBST(q);

    return temp;
}
```

Given a sorted array, give an algorithm for converting the array to BST :

```
BinaryTreeNode* BuildBST(int A[], int left, int right) {
    BinaryTreeNode* newNode;
    int mid;
    if (left > right)
        return NULL;
    newNode = new BinaryTreeNode();
    if (!newNode) {
        std::cerr << "Memory Error!" << std::endl;
        return NULL;
    }
    if (left == right) {
        newNode->data = A[left];
        newNode->left = newNode->right = NULL;
    } else {
        mid = left + (right - left) / 2;
        newNode->data = A[mid];
        newNode->left = BuildBST(A, left, mid - 1);
        newNode->right = BuildBST(A, mid + 1, right);
    }
    return newNode;
}
```

Given a SLL where elemens are sorted in ascending order, convert it to a height balanced BST :

```
BinaryTreeNode* SortedListToBST(ListNode*& list, int start, int end)
{
    if (start > end)
        return NULL;

    // Avoid overflow by calculating mid as (start + end) / 2
    int mid = start + (end - start) / 2;

    // Recursively build the left subtree
    BinaryTreeNode* leftChild = SortedListToBST(list, start, mid -
1);
```

```
    // Create the parent node
    BinaryTreeNode* parent = new BinaryTreeNode();
    if (!parent) {
        std::cerr << "Memory Error!" << std::endl;
        return NULL;
    }
    parent->data = list->data;
    parent->left = leftChild;

    // Move to the next node in the list
    list = list->next;

    // Recursively build the right subtree
    parent->right = SortedListToBST(list, mid + 1, end);

    return parent;
}

// Wrapper function to initiate BST conversion
BinaryTreeNode* SortedListToBST(ListNode* head, int n) {
    return SortedListToBST(head, 0, n - 1);
}
```

Finding kth smallest element in BST :

```
// Function to find the k-th smallest element in a BST
BinaryTreeNode* kthSmallestInBST(BinaryTreeNode* root, int k, int*
count) {
    if (!root)
        return NULL;

    // Search in the left subtree
    BinaryTreeNode* left = kthSmallestInBST(root->left, k, count);

    // If the k-th smallest element is found in the left subtree,
return it
    if (left)
        return left;

    // Increment the count for the current node
    if (++(*count) == k)
        return root;

    // Search in the right subtree
    return kthSmallestInBST(root->right, k, count);
```

```
}

// Wrapper function to initiate the k-th smallest search
BinaryTreeNode* findKthSmallest(BinaryTreeNode* root, int k) {
    int count = 0;
    return kthSmallestInBST(root, k, &count);
}
```

Floor and Ceiling in a Binary Search Tree (BST):

1. **The floor of a key is the largest key in the BST less than or equal to the given key.**
2. **The ceiling of a key is the smallest key in the BST greater than or equal to the given key.**

```
#include <iostream>
using namespace std;

// Definition of a BST Node
struct BinaryTreeNode {
    int data;
    BinaryTreeNode* left;
    BinaryTreeNode* right;
};

// Function to find the Floor in BST
BinaryTreeNode* FloorInBST(BinaryTreeNode* root, int data) {
    BinaryTreeNode* prev = NULL;
    return FloorInBSTUtil(root, prev, data);
}

BinaryTreeNode* FloorInBSTUtil(BinaryTreeNode* root, BinaryTreeNode*
&prev, int data) {
    if (!root) return NULL;

    if (!FloorInBSTUtil(root->left, prev, data)) return 0;

    if (root->data == data) return root;
    if (root->data > data) return prev;

    prev = root;
    return FloorInBSTUtil(root->right, prev, data);
}

// Function to find the Ceiling in BST
```

```
BinaryTreeNode* CeilingInBST(BinaryTreeNode* root, int data) {
    BinaryTreeNode* prev = NULL;
    return CeilingInBSTUtil(root, prev, data);
}

BinaryTreeNode* CeilingInBSTUtil(BinaryTreeNode* root,
BinaryTreeNode* &prev, int data) {
    if (!root) return NULL;

    if (!CeilingInBSTUtil(root->right, prev, data)) return 0;

    if (root->data == data) return root;
    if (root->data < data) return prev;

    prev = root;
    return CeilingInBSTUtil(root->left, prev, data);
}

// Function to create a new BST node
BinaryTreeNode* newNode(int data) {
    BinaryTreeNode* node = new BinaryTreeNode();
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Function to insert a new key in BST
BinaryTreeNode* insert(BinaryTreeNode* root, int key) {
    if (root == NULL) return newNode(key);

    if (key < root->data)
        root->left = insert(root->left, key);
    else if (key > root->data)
        root->right = insert(root->right, key);

    return root;
}

// Driver Code
int main() {
    BinaryTreeNode* root = NULL;
    root = insert(root, 8);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 2);
```

```
    root = insert(root, 6);
    root = insert(root, 10);
    root = insert(root, 14);

    int key = 5;
    BinaryTreeNode* floor = FloorInBST(root, key);
    BinaryTreeNode* ceiling = CeilingInBST(root, key);

    if (floor)
        cout << "Floor of " << key << " is " << floor->data << endl;
    else
        cout << "Floor of " << key << " doesn't exist\n";

    if (ceiling)
        cout << "Ceiling of " << key << " is " << ceiling->data <<
endl;
    else
        cout << "Ceiling of " << key << " doesn't exist\n";

    return 0;
}
```

Given a BST and two numbers k1 and k2, give an algorithm for printing all the elements of BST in the range K1 and K2.

Approach 1 :

```
void RangePrinter(BinarySearchTreeNode* root, int K1, int K2) {
    if (root == NULL)
        return;

    if (root->data >= K1)
        RangePrinter(root->left, K1, K2);

    if (root->data >= K1 && root->data <= K2)
        cout << root->data << " ";

    if (root->data <= K2)
        RangePrinter(root->right, K1, K2);
}
```

Approach 2 :

```
void RangeSearchLevelOrder(BinarySearchTreeNode* root, int K1, int
K2) {
    if (root == NULL)
        return;
```

```
    queue<BinarySearchTreeNode*> Q;
    Q.push(root);

    while (!Q.empty()) {
        BinarySearchTreeNode* temp = Q.front();
        Q.pop();

        if (temp->data >= K1 && temp->data <= K2)
            cout << temp->data << " ";

        if (temp->left && temp->data >= K1)
            Q.push(temp->left);

        if (temp->right && temp->data <= K2)
            Q.push(temp->right);
    }
}
```

For the key values 1, 2, ..., n how many structurally unique BSTs are possible that storage those keys :

```
int CountTrees(int n) {
    if (n <= 1)
        return 1;
    else {
        int sum = 0;
        int left, right, root;
        for (root = 1; root <= n; root++) {
            left = CountTrees(root - 1);
            right = CountTrees(n - root);
            sum += left * right;
        }
        return sum;
    }
}
```

Given a BST of size n, in which each node r has an additional field r -> size, the number of the keys in the sub-tree rooted at r (including the root node r). Give an O(h) algorithm GreaterthanConstant(r, k) to find the number of keys that are strictly greater than k (h is the height of the BST) :

```
int GreaterThanConstant(struct BinarySearchTreeNode *r, int k) {
    int keysCount = 0;
    while (r != NULL) {
        if (k < r->data) {
            keysCount += r->right->size + 1;
            r = r->left;
```

```
        } else if (k > r->data) {
            r = r->right;
        } else {
            keysCount += r->right->size;
            break;
        }
    }
    return keysCount;
}
```

AVL Trees

Declaration :

```
struct AVLTreeNode {
    AVLTreeNode *left;
    int data;
    AVLTreeNode *right;
    int height;
};
```

Finding Height of an AVL Tree :

```
int Height(AVLTreeNode *root) {
    if (!root)
        return -1;
    else
        return root->height;
}
```

Single Rotations :

Left Left Rotation :

```
AVLTreeNode* SingleRotateLeft(AVLTreeNode* X) {
    AVLTreeNode* W = X->left;
    X->left = W->right;
    W->right = X;
    X->height = max(Height(X->left), Height(X->right)) + 1;
    W->height = max(Height(W->left), Height(X)) + 1;
    return W;
}
```

Right Right Rotation :

```
AVLTreeNode* SingleRotateRight(AVLTreeNode* W) {
    AVLTreeNode* X = W->right;
    W->right = X->left;
    X->left = W;
    W->height = max(Height(W->right), Height(W->left)) + 1;
```

```
    X->height = max(Height(X->right), Height(W)) + 1;
    return X;
}
```

Double Rotations :

Left Right Rotation :

```
AVLTreeNode* DoubleRotateWithLeft(AVLTreeNode* Z) {
    Z->left = SingleRotateRight(Z->left);
    return SingleRotateLeft(Z);
}
```

Right Left Rotation :

```
AVLTreeNode* DoubleRotateWithRight(AVLTreeNode* Z) {
    Z->right = SingleRotateLeft(Z->right);
    return SingleRotateRight(Z);
}
```

Instruction :

```
AVLTreeNode* Insert(AVLTreeNode* root, AVLTreeNode* parent, int data)
{
    if (!root) {
        root = new AVLTreeNode();
        if (!root) {
            std::cerr << "Memory Error\n";
            return nullptr;
        } else {
            root->data = data;
            root->height = 0;
            root->left = root->right = nullptr;
        }
    } else if (data < root->data) {
        root->left = Insert(root->left, root, data);
        if (Height(root->left) - Height(root->right) == 2) {
            if (data < root->left->data)
                root = SingleRotateLeft(root);
            else
                root = DoubleRotateLeft(root);
        }
    } else if (data > root->data) {
        root->right = Insert(root->right, root, data);
        if (Height(root->right) - Height(root->left) == 2) {
            if (data > root->right->data)
                root = SingleRotateRight(root);
            else
                root = DoubleRotateWithRight(root);
```

```
        }
    }
    root->height = std::max(Height(root->left), Height(root->right))
+ 1;
    return root;
}
```

Problems

Given a height hhh, write an algorithm to generate a full binary tree HB(h), where the tree has 2^(h+1)−1 nodes. Each node in the tree is numbered sequentially from 1 to 2^(h+1)−1, assuming the height of a tree with one node is 0 :

Approach 1 :

```
BinarySearchTreeNode* BuildHB0(int h) {
    if (h == 0) return nullptr;
    BinarySearchTreeNode* temp = new BinarySearchTreeNode();
    temp->left = BuildHB0(h - 1);
    temp->data = ++count;
    temp->right = BuildHB0(h - 1);
    return temp;
}
```

Approach 2 :

```
BinarySearchTreeNode* BuildHB0(int l, int r) {
    if (l > r) return nullptr;
    int mid = l + (r - l) / 2;
    BinarySearchTreeNode* temp = new BinarySearchTreeNode();
    temp->data = mid;
    temp->left = BuildHB0(l, mid - 1);
    temp->right = BuildHB0(mid + 1, r);
    return temp;
}
```

Given a BST, check whether it is an AVL or not :

```
int IsAVL(BinarySearchTreeNode* root) {
    if (!root) return 0;
    int left = IsAVL(root->left);
    if (left == -1) return -1;
    int right = IsAVL(root->right);
    if (right == -1) return -1;
    if (std::abs(left - right) > 1) return -1;
    return std::max(left, right) + 1;
}
```

Given a height h, give an algorithm to generate an AVL tree with minimum number of nodes :

```
int count = 1;
AVLTreeNode* GenerateAVLTree(int h) {
    if (h == 0) return NULL;
    AVLTreeNode* temp = new AVLTreeNode();
    temp->left = GenerateAVLTree(h - 1);
    temp->data = count++;
    temp->right = GenerateAVLTree(h - 2);
    temp->height = h;
    return temp;
}
```

Given an AVL tree with n integer items and two integers a and b, where a and b can be any integers with a <= b. Implement an algorithm to count the number of nodes in the range [a, b] :

```
int RangeCount(AVLNode* root, int a, int b) {
    if (root == NULL)
        return 0;
    else if (root->data > b)
        return RangeCount(root->left, a, b);
    else if (root->data < a)
        return RangeCount(root->right, a, b);
    else if (root->data >= a && root->data <= b)
        return RangeCount(root->left, a, b) + RangeCount(root->right,
a, b) + 1;
    return 0;
}
```

Given a BST (applicable to AVL trees as well) where each node contains two data elements (its data and also the number of nodes in its subtrees) as shown below. Convert the tree to another BST by replacing the second data element (number of nodes in its subtrees) with previous node data in inorder traversal. Note that each node is merged with inorder previous node data. Also make sure that conversion happens in-place :

```
void TreeCompression(BST* root) {

    if (!root) return;

    Queue q;

    q.EnQueue(root);
```

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

        BST* temp = q.DeQueue();

        if (!temp) continue;

        BST* temp2 = nullptr;

        if (temp->left && temp->right && temp->left->data >
temp->right->data) {

            temp2 = FindMax(temp->left);

        } else if (temp->left || temp->right) {

            temp2 = FindMin(temp);

        }

        if (temp2) {

            temp->data = temp2->data;

            DeleteNodeInBST(temp, temp2);

        }

        if (temp->left) q.EnQueue(temp->left);

        if (temp->right) q.EnQueue(temp->right);

    }

}
```

Improved Complexity :

```
BinarySearchTreeNode* TreeCompression(BinarySearchTreeNode* root,
int* previousNodeData) {

    if (!root) return nullptr;



    TreeCompression(root->left, previousNodeData);



    if (*previousNodeData == INT_MIN) {
```

```
        *previousNodeData = root->data1;
        free(root); // Delete the node
    } else if (*previousNodeData != INT_MIN) {
        // Process current node
        root->data2 = *previousNodeData;
        *previousNodeData = INT_MIN;
    }

    return TreeCompression(root->right, previousNodeData);
}
```

Given a BST and a key, find the element in the BST which is closest to the given key :

Non Recursive :

```
int ClosestInBST(BinaryTreeNode* root, int key) {
    if (!root) return 0;
    queue<BinaryTreeNode*> Q;
    BinaryTreeNode* temp = nullptr;
    BinaryTreeNode* element = nullptr;
    int difference = INT_MAX;
    Q.push(root);
    while (!Q.empty()) {
        temp = Q.front();
        Q.pop();
        if (difference > std::abs(temp->data - key)) {
            difference = std::abs(temp->data - key);
            element = temp;
```

```
        }
        if (temp->left) Q.push(temp->left);
        if (temp->right) Q.push(temp->right);
    }
    return element->data;
}
```

Recursive :

```
BinaryTreeNode* ClosestInBST(BinaryTreeNode* root, int key) {
    if (root == nullptr)
        return nullptr;
    if (root->data == key)
        return root;
    if (key < root->data) {
        if (!root->left)
            return root;
        BinaryTreeNode* temp = ClosestInBST(root->left, key);
        return abs(temp->data - key) > abs(root->data - key) ? root :
temp;
    } else {
        if (!root->right)
            return root;


        BinaryTreeNode* temp = ClosestInBST(root->right, key);
        return abs(temp->data - key) > abs(root->data - key) ? root :
temp;
    }
```

```
}
```

Given a binary tree, remove all half nodes (which have only one child). Do not touch the leaves :

```
BinaryTreeNode* removeHalfNodes(BinaryTreeNode* root) {
    if (!root)
        return nullptr;
    root->left = removeHalfNodes(root->left);
    root->right = removeHalfNodes(root->right);
    if (root->left == nullptr && root->right == nullptr)
        return root;
    if (root->left == nullptr)
        return root->right;
    if (root->right == nullptr)
        return root->left;
    return root;
}
```

Given a binary tree, how do you remove its leaves :

```
struct BinaryTreeNode* removeLeaves(struct BinaryTreeNode* root) {
    if (root != NULL) {
        if (root->left == NULL && root->right == NULL) {
            delete root;
            return NULL;
        } else {
            root->left = removeLeaves(root->left);
            root->right = removeLeaves(root->right);
```

```
        }

    }

    return root;

}
```

Given a BST and two integers (min and max integers) as parameters, how do you remove (prune) elements that are not within that range?

```
BinarySearchTreeNode* pruneBST(BinarySearchTreeNode* root, int A, int
B) {

    if (!root) return NULL;

    root->left = pruneBST(root->left, A, B);

    root->right = pruneBST(root->right, A, B);

    if (A <= root->data && root->data <= B) return root;

    if (root->data < A) return root->right;

    if (root->data > B) return root->left;

    return NULL;

}
```

Given a binary tree, how do you connect all the adjacent nodes at the same level? Assume that given binary tree has next pointer along with left and right pointers.

```
void linkingNodesOfSameLevel(BinaryTreeNode* root) {

    Queue* Q = createQueue();

    BinaryTreeNode* prev = NULL;

    BinaryTreeNode* temp;

    int currentLevelNodeCount = 1, nextLevelNodeCount = 0;

    if (!root) return;

    enQueue(Q, root);

    while (!isEmptyQueue(Q)) {
```

```
        temp = deQueue(Q);
        if (temp->left) {
            enQueue(Q, temp->left);
            nextLevelNodeCount++;
        }
        if (temp->right) {
            enQueue(Q, temp->right);
            nextLevelNodeCount++;
        }
        if (prev) prev->next = temp;
        prev = temp;
        currentLevelNodeCount--;
        if (currentLevelNodeCount == 0) {
            currentLevelNodeCount = nextLevelNodeCount;
            nextLevelNodeCount = 0;
            prev = NULL;
        }
    }
}
```