Cây nhị phân :

#include <iostream>

class Node {

public:

    int data;

    Node\* left;

    Node\* right;

    Node(int value) {

        data = value;

        left = nullptr;

        right = nullptr;

    }

    ~Node() {

        delete left;

        delete right;

    }

    void insertLeft(Node\* p) {

        if (left != nullptr) {

            std::cout << "Error: Left child already exists" << std::endl;

        } else {

            left = p;

        }

    }

    void insertRight(Node\* p) {

        if (right != nullptr) {

            std::cout << "Error: Right child already exists" << std::endl;

        } else {

            right = p;

        }

    }

    void removeLeft() {

        delete left;

        left = nullptr;

    }

    void removeRight() {

        delete right;

        right = nullptr;

    }

    static void deleteTree(Node\* root) {

        if (root != nullptr) {

            deleteTree(root->left);

            deleteTree(root->right);

            delete root;

        }

    }

    static Node\* search(Node\* root, int value) {

        if (root == nullptr || root->data == value) {

            return root;

        }

        if (value < root->data) {

            return search(root->left, value);

        } else {

            return search(root->right, value);

        }

    }

    static void preorderTraversal(Node\* root) {

        if (root != nullptr) {

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

            preorderTraversal(root->left);

            preorderTraversal(root->right);

        }

    }

    static void inorderTraversal(Node\* root) {

        if (root != nullptr) {

            inorderTraversal(root->left);

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

            inorderTraversal(root->right);

        }

    }

    static void postorderTraversal(Node\* root) {

        if (root != nullptr) {

            postorderTraversal(root->left);

            postorderTraversal(root->right);

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

        }

    }

};

int main() {

    Node\* root = new Node(5);

    Node\* leftChild = new Node(3);

    Node\* rightChild = new Node(7);

    Node\* leftGrandchild = new Node(2);

    Node\* rightGrandchild = new Node(4);

    root->insertLeft(leftChild);

    root->insertRight(rightChild);

    leftChild->insertLeft(leftGrandchild);

    leftChild->insertRight(rightGrandchild);

    std::cout << "Preorder Traversal: ";

    Node::preorderTraversal(root);

    std::cout << std::endl;

    std::cout << "Inorder Traversal: ";

    Node::inorderTraversal(root);

    std::cout << std::endl;

    std::cout << "Postorder Traversal: ";

    Node::postorderTraversal(root);

    std::cout << std::endl;

    Node\* searchResult = Node::search(root, 4);

    if (searchResult != nullptr) {

        std::cout << "Found node with value 4" << std::endl;

    } else {

        std::cout << "Node with value 4 not found" << std::endl;

    }

    root->removeLeft();

    root->removeRight();

    Node::deleteTree(root);

    return 0;

}

Ứng dụng : DSA11010 Cây nhị phân hoàn hảo :

Phân tích tính phù hợp : Trong bài toán trên, ta có thể sử dụng cấu trúc cây nhị phân để lưu trữ các node của cây. Mỗi node trong cây nhị phân có thể được lưu trữ dưới dạng một đối tượng (object) với các thuộc tính như giá trị của node, con trái và con phải của node. Trong trường hợp này, để kiểm tra xem cây nhị phân có phải là một cây hoàn hảo hay không, ta cần duyệt toàn bộ cây và kiểm tra mức của tất cả các node lá. Nếu tất cả các node lá đều có cùng một mức và tất cả các node trung gian đều có hai node con, thì cây được coi là hoàn hảo.

Code :

#include <bits/stdc++.h>

#define ll long long

#define II pair<int, int>

using namespace std;

struct Node {

    int data;

    Node \*left, \*right;

    Node(int data) {

        this->data = data;

        this->left = NULL;

        this->right = NULL;

    }

};

bool isPerfect;

void MakeNode(Node \*root, int data, char c) {

    if (c == 'L')

        root->left = new Node(data);

    else

        root->right = new Node(data);

}

void Insert(Node \*root, int x, int y, char c) {

    if (root == NULL)

        return;

    if (root->data == x)

        MakeNode(root, y, c);

    else {

        Insert(root->left, x, y, c);

        Insert(root->right, x, y, c);

    }

}

int CountLeaf(Node \*root) {

    if (root == NULL)

        return 0;

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

        return 1;

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

        isPerfect = false;

        return 0;

    }

    return CountLeaf(root->left) + CountLeaf(root->right);

}

void TestCase() {

    isPerfect = true;

    int n; cin >> n;

    Node \*root = NULL;

    while (n--) {

        int x, y; char c;

        cin >> x >> y >> c;

        if (root == NULL) {

            root = new Node(x);

        }

        Insert(root, x, y, c);

    }

    int cntLeft = CountLeaf(root->left), cntRight = CountLeaf(root->right);

    if (isPerfect && cntLeft == cntRight)

        cout << "Yes";

    else

        cout << "No";

}

int main() {

    ios\_base::sync\_with\_stdio(false);

    cin.tie(NULL); cout.tie(NULL);

    int T = 1; cin >> T;

    while (T--) {

        TestCase();

        cout << endl;

    }

    return 0;

}

Cây nhị phân tìm kiếm :

#include <iostream>

using namespace std;

struct Node {

    int value;

    Node\* left;

    Node\* right;

};

Node\* createNode(int value) {

    Node\* newNode = new Node();

    newNode->value = value;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

Node\* insert(Node\* root, int value) {

    if (root == NULL) {

        return createNode(value);

    }

    if (value < root->value) {

        root->left = insert(root->left, value);

    }

    else {

        root->right = insert(root->right, value);

    }

    return root;

}

Node\* findMin(Node\* node) {

    while (node->left != NULL) {

        node = node->left;

    }

    return node;

}

Node\* remove(Node\* root, int value) {

    if (root == NULL) {

        return NULL;

    }

    if (value < root->value) {

        root->left = remove(root->left, value);

    }

    else if (value > root->value) {

        root->right = remove(root->right, value);

    }

    else {

        if (root->left == NULL) {

            Node\* temp = root->right;

            delete root;

            return temp;

        }

        else if (root->right == NULL) {

            Node\* temp = root->left;

            delete root;

            return temp;

        }

        Node\* temp = findMin(root->right);

        root->value = temp->value;

        root->right = remove(root->right, temp->value);

    }

    return root;

}

Node\* search(Node\* root, int value) {

    if (root == NULL || root->value == value) {

        return root;

    }

    if (root->value < value) {

        return search(root->right, value);

    }

    return search(root->left, value);

}

void inorderTraversal(Node\* root) {

    if (root == NULL) {

        return;

    }

    inorderTraversal(root->left);

    cout << root->value << " ";

    inorderTraversal(root->right);

}

void preorderTraversal(Node\* root) {

    if (root == NULL) {

        return;

    }

    cout << root->value << " ";

    preorderTraversal(root->left);

    preorderTraversal(root->right);

}

void postorderTraversal(Node\* root) {

    if (root == NULL) {

        return;

    }

    postorderTraversal(root->left);

    postorderTraversal(root->right);

    cout << root->value << " ";

}

Node\* leftRotate(Node\* root) {

    Node\* newRoot = root->right;

    root->right = newRoot->left;

    newRoot->left = root;

    return newRoot;

}

Node\* rightRotate(Node\* root) {

    Node\* newRoot = root->left;

    root->left = newRoot->right;

    newRoot->right = root;

    return newRoot;

}

int main() {

    Node\* root = NULL;

    // Insertion

    root = insert(root, 50);

    insert(root, 30);

    insert(root, 20);

    insert(root, 40);

    insert(root, 70);

    insert(root, 60);

    insert(root, 80);

    // Traversal

    cout << "Inorder traversal: ";

    inorderTraversal(root);

    cout << endl;

    cout << "Preorder traversal: ";

    preorderTraversal(root);

    cout << endl;

    cout << "Postorder traversal: ";

    postorderTraversal(root);

    cout << endl;

    // Search

    int searchValue = 40;

    Node\* searchResult = search(root, searchValue);

    if (searchResult == NULL) {

        cout << searchValue << " not found in tree." << endl;

    }

    else {

        cout << searchValue << " found in tree." << endl;

    }

    // Deletion

    int deleteValue = 20;

    root = remove(root, deleteValue);

    // Rotation

    root = leftRotate(root);

    // Traversal after deletion and rotation

    cout << "Inorder traversal: ";

    inorderTraversal(root);

    cout << endl;

    system("pause");

    return 0;

}

Ứng dụng : DSA11017 Duyệt cây nhị phân tìm kiếm 1

Phân tích tính phù hợp : Cây nhị phân tìm kiếm là một cấu trúc dữ liệu rất hữu ích trong việc lưu trữ và tìm kiếm dữ liệu có thứ tự. Trong bài toán này, chúng ta đã cho một mảng biểu diễn phép duyệt theo thứ tự trước của một cây nhị phân tìm kiếm và yêu cầu đưa ra phép duyệt theo thứ tự sau của cây nhị phân tìm kiếm. Các bước để giải quyết bài toán này như sau:

1.Tạo cây nhị phân tìm kiếm từ mảng được cho. Vì mảng được cho biểu diễn phép duyệt theo thứ tự trước của cây, nên chúng ta có thể sử dụng phương pháp đệ quy để tạo cây từ mảng. Các bước để tạo cây nhị phân tìm kiếm từ mảng được cho như sau:

-Nếu mảng rỗng, trả về null. Lấy phần tử đầu tiên của mảng làm gốc của cây.

-Chia mảng thành hai phần, phần đầu tiên là các phần tử nhỏ hơn gốc, phần thứ hai là các phần tử lớn hơn gốc.

-Đệ quy tạo cây nhị phân tìm kiếm cho cả hai phần của mảng và gán chúng vào node trái và node phải của gốc.

2.Duyệt cây theo thứ tự sau và đưa ra kết quả. Vì phép duyệt theo thứ tự sau sẽ trả về các node lá trước, sau đó đến các node trung gian, nên chúng ta có thể sử dụng đệ quy để duyệt cây theo thứ tự sau và đưa ra kết quả.

Code :

#include <iostream>

#include <vector>

using namespace std;

void postorder(vector<int>& A, int left, int right, bool last) {

    if (left > right) {

        return;

    }

    int root = A[left];

    int i = left + 1;

    while (i <= right && A[i] < root) {

        i++;

    }

    postorder(A, left + 1, i - 1, false);

    postorder(A, i, right, last || left == right);

    if (last) {

        cout << root << " ";

    } else {

        cout << root << " ";

    }

}

int main() {

    int T;

    cin >> T;

    while (T--) {

        int n;

        cin >> n;

        vector<int> A(n);

        for (int i = 0; i < n; i++) {

            cin >> A[i];

        }

        postorder(A, 0, n - 1, true);

        cout << endl;

    }

    system("pause");

    return 0;

}

3.Cài đặt các thao tác trên cây nhị phân tìm kiếm tự cân bằng :

#include <iostream>

using namespace std;

struct Node {

    int key;

    Node\* left;

    Node\* right;

    int height;

};

// Tính chiều cao của một node

int height(Node\* node) {

    if (node == nullptr) {

        return 0;

    }

    return node->height;

}

// Tạo một node mới với giá trị key

Node\* createNode(int key) {

    Node\* node = new Node();

    node->key = key;

    node->left = nullptr;

    node->right = nullptr;

    node->height = 1;

    return node;

}

// Xoay trái cây con có chiều cao lệch

Node\* rotateLeft(Node\* node) {

    Node\* newParent = node->right;

    node->right = newParent->left;

    newParent->left = node;

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

    newParent->height = max(height(newParent->left), height(newParent->right)) + 1;

    return newParent;

}

// Xoay phải cây con có chiều cao lệch

Node\* rotateRight(Node\* node) {

    Node\* newParent = node->left;

    node->left = newParent->right;

    newParent->right = node;

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

    newParent->height = max(height(newParent->left), height(newParent->right)) + 1;

    return newParent;

}

// Thêm node mới vào cây

Node\* insertNode(Node\* node, int key) {

    if (node == NULL) {

        return node;

    }

    if (key < node->key) {

        node->left = insertNode(node->left, key);

    } else if (key > node->key) {

        node->right = insertNode(node->right, key);

    } else {

        // Key đã tồn tại trong cây

        return node;

    }

    // Cập nhật chiều cao của node hiện tại

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

    // Kiểm tra sự cân bằng của cây

    int balance = height(node->left) - height(node->right);

    if (balance > 1 && key < node->left->key) {

        // Trường hợp đặc biệt: xoay phải khi chèn node vào cây con trái của node cha

        return rotateRight(node);

    }

    if (balance > 1 && key > node->left->key) {

        // Trường hợp thường: xoay trái khi chèn node vào cây con phải của node con trái

        node->left = rotateLeft(node->left);

        return rotateRight(node);

    }

    if (balance < -1 && key > node->right->key) {

        // Trường hợp đặc biệt: xoay trái khi chèn node vào cây con phải của node cha

        return rotateLeft(node);

    }

    if (balance < -1 && key < node->right->key) {

        // Trường hợp thường: xoay phải khi chèn node vào cây con trái của node con phải

        node->right = rotateRight(node->right);

        return rotateLeft(node);

    }

    // Trả về node hiện tại

    return node;

}

int main()

{

    int n; cin >> n;

    Node\* root = NULL;

    for(int i=0 ;i<n ;i++)

    {

        int x; cin >> x;

        if(root == NULL)

        {

            root = createNode(x);

            insertNode(root, x);

        }

        else

        {

            //Node\* p = new Node();

            insertNode(root , x);

        }

    }

    cout << root->key << endl;

    return 0;

}