# **PHẦN 1: CÂY NHỊ PHÂN TÌM KIẾM BST**

### **1. Biên dịch đoạn chương trình nêu trên.**

- Node để đại diện cho một nút trong cây nhị phân tìm kiếm (BST) và BST để quản lý cây nhị phân tìm kiếm, bao gồm các thao tác như chèn, tìm kiếm, duyệt và xóa nút. Chương trình tạo và quản lý một cây nhị phân tìm kiếm và sau đó in ra các giá trị của cây theo thứ tự Pre-order, thể hiện cấu trúc của BST đã xây dựng.

### **2. Vẽ hình cây nhị phân tìm kiếm theo dữ liệu được câu 1.**

10

/ \

9 19

/ /

3 15

\

8

/

4

/

1

### **3. Thực hiện hoàn thiện các hàm: có chú thích //sinh vien code**

BST.cpp

#include "BST.h"

#include <iostream>

using namespace std;

BST::BST() {

 //ctor

 this->root=nullptr;

}

BST::~BST() {

 //dtor }

bool BST::InsertNode(Node\* n){

 Node \*p=this->root;

 Node \*T;

 if(root==nullptr)

{

 this->root=n;

 return true;

}

 while(p!=nullptr){

 T=p;

 if(p->Getkey()>n->Getkey())

 p=p->Getleft();

 else

 if(p->Getkey()<n->Getkey())

 p=p->Getright();

 else

 if(p->Getkey()==n->Getkey())

 return false;

}

 if(T->Getkey()>n->Getkey())

T->Setleft(n);

 else T->Setright(n);

n->Setparent(T);

 return true;

 }

bool BST::InsertNodeRe(Node\* root,Node\*p){

 if(root==nullptr){

 root=p;

 return true;

}

 if(root->Getkey()==p->Getkey())

 return false;

 else if(root->Getkey()>p->Getkey())

 return InsertNodeRe(root->Getleft(),p);

 else return InsertNodeRe(root->Getright(),p);

}

void BST::NLR(Node\*r){

 if(r!=nullptr){

 cout<<r->Getkey()<<"\n";

 NLR(r->Getleft());

 NLR(r->Getright());

}

}

void BST::LNR(Node\*r)

{

if (r != nullptr)

{

LNR(r->Getleft());

cout << r->Getkey() << "\n";

LNR(r->Getright());

}

}

void BST::LRN(Node\*r)

{

    if (r != nullptr)

    {

        LRN(r->Getleft());

        cout << r->Getkey() << "\n";

        LRN(r->Getright());

    }

}

void BST::TravelNLR()

{

 NLR(this->root);

}

void BST::TravelLNR()

{

 LNR(this->root);

}

void BST::TravelLRN()

{

    LRN(this->root);

}

Node\* BST::search\_x(int k)

{

{

    Node\* p = this->root;

    while (p != NULL)

    {

        if (k == p->Getkey()) return p;

        else if(k < current->Getkey())

            p = p->Getleft();

else

p = p->Getright();

}

    return NULL; //Khong tim dc nut nao co khoa x

}

void BST::deleteNode(Node\* n){

 Node\* p=n;

 if(p->Getleft()==nullptr&&n->Getright()==nullptr){

        // Case 1: Node is a leaf

        // Find the parent

        Node\* parent = nullptr;

        Node\* current = this->root;

        while (current != nullptr && current != n)

        {

            parent = current;

            if (n->Getkey() < current->Getkey())

            {

                current = current->Getleft();

            }

            else

            {

                current = current->Getright();

            }

        }

        // Update the parent pointer

        if (parent != nullptr)

        {

            if (parent->Getleft() == n)

                parent->Setleft(nullptr);

            else parent->Setright(nullptr);

        }

        else

        {

            root = nullptr;

        }

 delete n;

}

 else{

 if(p->Getright()!=nullptr){

 p=p->Getright();

 while(p->Getleft()!=nullptr)//

 p=p->Getleft();

n->Setkey(p->Getkey());

             if (parentRightSubtree->Getleft() == p) // set pointer of p parents to child of p

            {

                parentRightSubtree->Setleft(p->Getright());

            }

            else

            {

                parentRightSubtree->Setright(p->Getright());

            }

**delete** p;

        }

        else

        {

            // Case 2: Node has left child only

            // Find largest node in the left subtree

            Node\* parentLeftSubtree = p;

            p = p->Getleft();

            while (p->Getright() != nullptr)

            {

                parentLeftSubtree = p;

                p = p->Getright();

            }

 }else{

 p=p->Getleft();

 while(p->Getright()!=nullptr)//

 p=p->Getright();

n->Setkey(p->Getkey());

            if (parentLeftSubtree->Getright() == p)

            {

                parentLeftSubtree->Setright(p->Getleft());

            }

            else

            {

                parentLeftSubtree->Setleft(p->Getleft());

            }

**delete** p;

        }

    }

}

## **Áp dụng – Nâng cao**

### **1. Bổ sung chương trình mẫu cho phép tính tổng giá trị các nút trên cây nhị phân gồm các giá trị nguyên.**

BST.h

#ifndef BST\_H

#define BST\_H

#include "Node.h"

class BST

{

public:

    BST();

    virtual ~BST();

    Node\* Getroot() { return root; }

    void Setroot(Node\* val) { root = val; }

    bool InsertNode(Node\*);

    bool InsertNodeRe(Node\*, Node\* );

    void deleteNode(Node\*);

    void TravelNLR();

    void TravelLNR();

    void TravelLRN();

    void NLR(Node \*);

    void LNR(Node \*);

    void LRN(Node \*);

    Node\* search\_x(int);

    int SumTree();

    void SumNodeTree(Node\*, int&);

protected:

private:

    Node\* root;

};

#endif // BST\_H

BST.cpp

int BST::SumTree() {

    int totalSum = 0;

    SumNodeTree(this->root, totalSum);

    return totalSum;

}

void BST::SumNodeTree(Node\* r, int& totalSum) {

    if (r != nullptr) {

        totalSum += r->Getkey();

        SumNodeTree(r->Getleft(), totalSum);

        SumNodeTree(r->Getright(), totalSum);

    }

}

main.cpp

    cout << "Tong gia tri cua cac nut: " << tree->SumTree() << endl;

### **2. Bổ sung chương trình mẫu cho phép tìm giá trị nguyên lớn nhất và nhỏ nhất trong số các phần tử nguyên trên cây nhị phân tìm kiếm gồm các giá trị nguyên.**

**Gợi ý: dựa vào tính chất 1, 2 của cây nhị phân tìm kiếm.**

BST.h

#ifndef BST\_H

#define BST\_H

#include "Node.h"

class BST

{

public:

…

    Node\* FindMin();

    Node\* FindMax();

    Node\* FindMinNode(Node\*);

    Node\* FindMaxNode(Node\*);

…

BST.cpp

Node\* BST::FindMin()

{

    return FindMinNode(this->root);

}

Node\* BST::FindMinNode(Node\* r)

{

    if ( r->Getleft()== nullptr)

    {

        return r;

    }

    FindMinNode(r->Getleft());

}

Node\* BST::FindMax()

{

    return FindMaxNode(this->root);

}

Node\* BST::FindMaxNode(Node\* r)

{

    if (r->Getright() == nullptr)

    {

        return r;

    }

    FindMaxNode(r->Getright());

}

main.cpp

    cout << "Gia tri nguyen nho nhat: " << tree->FindMin()->Getkey() << endl;

    cout << " Gia tri Nguyen lon nhat: " << tree->FindMax()->Getkey() << endl;

### **3. Bổ sung chương trình mẫu cho phép tính số lượng các nút của cây nhị phân gồm các giá trị nguyên.**

**Gợi ý: tham khảo hàm NLR để viết hàm CountNode.**

BST.h

    int CountAllNode();

    void CountAllNode(Node\*, int&);

    int CountNode();

    void CountNode(Node\*, int&);

BST.cpp

int BST::CountAllNode()

{

    int countNode = 0;

    CountNode(this->root, countNode);

    return countNode;

}

void BST::CountNode(Node\* r, int& countNode)

{

    if (r != nullptr)

    {

        countNode++;

        CountNode(r->Getleft(), countNode);

        CountNode(r->Getright(), countNode);

    }

}

main.cpp

    cout << "So luong cac nut: " << tree->CountAllNode() << endl;

### **4. Bổ sung chương trình mẫu cho biết số lượng các nút lá trên cây nhị phân.**

Gợi ý: tham khảo thao tác duyệt cây nhị phân NLR.

BST.h

    int CountLeafNode();

    void CountLeafNode(Node\*, int&);

BST.cpp

int BST::CountLeafNode() {

    int leafCount = 0;

    CountLeafNode(this->root, leafCount);

    return leafCount;

}

void BST::CountLeafNode(Node\* r, int& leafCount) {

    if (r != nullptr) {

        if (r->Getleft() == nullptr && r->Getright() == nullptr) {

            leafCount++;

        }

        CountLeafNodeHelper(r->Getleft(), leafCount);

        CountLeafNodeHelper(r->Getright(), leafCount);

    }

}

main.cpp

    cout << "So luong cac nut la: " << tree->CountLeafNodes() << endl;

### **5. Sử dụng cây nhị phân tìm kiếm để giải bài toán:**

a. Đếm có bao nhiêu giá trị phân biệt trong dãy số cho trước

b. Với mỗi giá trị phân biệt, cho biết số lượng phần tử

Node.h

#ifndef NODE\_H

#define NODE\_H

class Node

{

public:

    Node();

    Node(int);

    virtual ~Node();

    Node \*Getleft() { return left; }

    void Setleft(Node\* val) { left = val; }

    Node \*Getright() { return right; }

    void Setright(Node\* val) { right = val; }

    Node \*Getparent() { return parent; }

    void Setparent(Node\* val) { parent = val; }

    int Getkey() { return key; }

    void Setkey(int val) { key = val; }

    int GetCount() const { return count; }

    void IncrementCount() { count++; }

protected:

private:

    Node \*left;

    Node \*right;

    Node \*parent;

    int key;

    int count;

};

#endif // NODE\_H

Node.cpp

#include "Node.h"

Node::Node()

{

    // ctor

    this->key = 0;

    this->left = nullptr;

    this->right = nullptr;

    this->parent = nullptr;

}

Node::Node(int k)

{

    // ctor

    this->key = k;

    this->left = nullptr;

    this->right = nullptr;

    this->parent = nullptr;

    this->count = 1;

}

Node::~Node()

{

    // dtor

}

BST.h

#ifndef BST\_H

#define BST\_H

#include "Node.h"

class BST

{

public:

…

    int CountDistinctValues();

    void CountDistinctValues(Node\* r, int& count);

    void PrintFrequencies();

    void PrintFrequencies(Node\*);

…

#endif // BST\_H

BST.cpp

a.

int BST::CountDistinctValues() {

    int count = 0;

    CountDistinctValues(this->root, count);

    return count;

}

void BST::CountDistinctValues(Node\* r, int& count) {

    if (r != nullptr) {

        count++;  // Count this node (represents a distinct value)

        CountDistinctValues(r->Getleft(), count);

        CountDistinctValues(r->Getright(), count);

    }

}

b.

void BST::PrintFrequencies()

{

    PrintFrequencies(this->root);

}

void BST::PrintFrequencies(Node\* r)

{

    if (r != nullptr)

    {

        PrintFrequencies(r->Getleft());

        cout << "Gia tri: " << r->Getkey() << ",So luong phan tu: " << r->GetCount() << endl;

        PrintFrequencies(r->Getright());

    }

}

main.cpp

a.

    cout << "Gia tri phan biet: " << tree->CountDistinctValues() << endl;

b.

    cout << "So luong phan tu cua moi gia tri phan biet:" << endl;

    tree->PrintFrequencies();

## **BÀI TẬP ỨNG DỤNG**

### **1. Sử dụng cây nhị phân tìm kiếm để giải bài toán đếm (thống kê) số lượng ký tự có trong văn bản (Không dấu).**

a. Xây dựng cây cho biết mỗi ký tự có trong văn bản xuất hiện mấy lần

b. Nhập vào 1 ký tự. Kiểm tra ký tự đó xuất hiện bao nhiêu lần trong văn bản

Node.h

#ifndef NODE\_H

#define NODE\_H

class Node

{

public:

    Node();

    Node(char); // Thay đổi từ int thành char

    virtual ~Node();

    Node \*Getleft() { return left; }

    void Setleft(Node\* val) { left = val; }

    Node \*Getright() { return right; }

    void Setright(Node\* val) { right = val; }

    Node \*Getparent() { return parent; }

    void Setparent(Node\* val) { parent = val; }

    char Getkey() { return key; }

    void Setkey(char val) { key = val; }

    int GetCount() const { return count; }

    void IncrementCount() { count++; }

protected:

private:

    Node \*left;

    Node \*right;

    Node \*parent;

    char key; // Thay đổi từ int thành char

    int count;

};

#endif // NODE\_H

Node.cpp

#include "Node.h"

Node::Node()

{

    // ctor

    this->key = '\0'; // Khởi tạo ký tự rỗng

    this->left = nullptr;

    this->right = nullptr;

    this->parent = nullptr;

    this->count = 0;

}

Node::Node(char k)

{

    // ctor

    this->key = k;

    this->left = nullptr;

    this->right = nullptr;

    this->parent = nullptr;

    this->count = 1;

}

Node::~Node()

{

    // dtor

}

BST.h

#ifndef BST\_H

#define BST\_H

#include "Node.h"

#include <iostream>

class BST

{

public:

    BST();

    virtual ~BST();

    Node\* Getroot() { return root; }

    void Setroot(Node\* val) { root = val; }

    bool InsertNode(Node\*);

    bool InsertNodeRe(Node\* , Node\* );

    void deleteNode(Node\*);

    void TravelNLR();

    void TravelLNR();

    void TravelLRN();

    void NLR(Node \*);

    void LNR(Node \*);

    void LRN(Node \*);

    Node\* search\_x(char); // Cập nhật tham số thành char

protected:

private:

    Node\* root;

    void DeleteAllNodes(Node\*);

};

#endif // BST\_H

BST.cpp

#include "BST.h"

using namespace std;

BST::BST()

    : root(nullptr)

{

    // Constructor

}

BST::~BST()

{

    DeleteAllNodes(root);

}

void BST::DeleteAllNodes(Node\* r)

{

    if (r != nullptr)

    {

        DeleteAllNodes(r->Getleft());

        DeleteAllNodes(r->Getright());

**delete** r;

    }

}

bool BST::InsertNode(Node\* newNode)

{

    if (root == nullptr)

    {

        root = newNode;

    }

    Node\* current = root;

    Node\* parent = nullptr;

    while (current != nullptr)

    {

        parent = current;

        if (newNode->Getkey() == current->Getkey())

        {

            current->IncrementCount();

**delete** newNode;

            return true;

        }

        else if (newNode->Getkey() < current->Getkey())

        {

            current = current->Getleft();

        }

        else

        {

            current = current->Getright();

        }

    }

    if (newNode->Getkey() < parent->Getkey())

    {

        parent->Setleft(newNode);

    }

    else

    {

        parent->Setright(newNode);

    }

    return true;

}

Node\* BST::search\_x(char key)

{

    Node\* current = root;

    while (current != nullptr)

    {

        if (key == current->Getkey())

        {

            return current;

        }

        else if (key < current->Getkey())

        {

            current = current->Getleft();

        }

        else

        {

            current = current->Getright();

        }

    }

    return nullptr; // Không tìm thấy

}

void BST::TravelLNR()

{

    LNR(root);

}

void BST::LNR(Node\* r)

{

    if (r != nullptr)

    {

        LNR(r->Getleft());

       cout << "" << r->Getkey() << " : " << r->GetCount() << " lan.\n";

        LNR(r->Getright());

    }

}

main.cpp

#include <iostream>

#include <string>

#include "BST.h"

#include "Node.h"

using namespace std;

int main()

{

    BST\* tree = **new** BST();

    string text;

    cout << "Nhap van ban(khong dau): ";

    getline(cin, text);

    for(char ch : text)

    {

        if(ch == ' ') continue;

        Node\* n = **new** Node(ch);

        tree->InsertNode(n);

    }

    // a.

    cout << "\nThong ke so luong ky tu trong van ban:\n";

    tree->TravelLNR(); // Duyet cay theo thu tu LNR (In-order traversal)

    // b.

    char searchChar;

    cout << "\nNhap vao mot ky tu de kiem tra so lan xuat hien: ";

    cin >> searchChar;

    Node\* foundNode = tree->search\_x(searchChar);

    if(foundNode != nullptr)

    {

        cout << "Ky tu '" << foundNode->Getkey() << "' xuat hien " << foundNode->GetCount() << " lan trong van ban.\n";

    }

    else

    {

        cout << "Ky tu '" << searchChar << "' khong co trong van ban.\n";

    }

    // Clean up

**delete** tree;

    return 0;

}

### **2. Bài toán tương tự như trên nhưng thống kê số lượng tiếng có trong văn bản (không dấu)**

Ví dụ:

Văn bản có nội dung như sau: “hoc sinh di hoc mon sinh hoc”

Kết quả cho thấy như sau:

di: 1

hoc: 3

mon: 1

sinh: 2

Node.h

#ifndef NODE\_H

#define NODE\_H

#include <string>

using namespace std;

class Node

{

public:

    Node();

    Node(const string&);

    virtual ~Node();

    Node \*Getleft() { return left; }

    void Setleft(Node\* val) { left = val; }

    Node \*Getright() { return right; }

    void Setright(Node\* val) { right = val; }

    Node \*Getparent() { return parent; }

    void Setparent(Node\* val) { parent = val; }

    string Getkey() { return key; }

    void Setkey(string val) { key = val; }

    int GetCount() const { return count; }

    void IncrementCount() { count++; }

protected:

private:

    Node \*left;

    Node \*right;

    Node \*parent;

    string key;

    int count;

};

#endif // NODE\_H

Node.cpp

#include "Node.h"

Node::Node()

{

    // ctor

    this->key = '\0'; // Khởi tạo ký tự rỗng

    this->left = nullptr;

    this->right = nullptr;

    this->parent = nullptr;

    this->count = 0;

}

Node::Node(const string& k)

{

    // ctor

    this->key = k;

    this->left = nullptr;

    this->right = nullptr;

    this->parent = nullptr;

    this->count = 1;

}

Node::~Node()

{

    // dtor

}

BST.h

#ifndef BST\_H

#define BST\_H

#include "Node.h"

#include <iostream>

using namespace std;

class BST

{

public:

    BST();

    virtual ~BST();

    Node\* Getroot() { return root; }

    void Setroot(Node\* val) { root = val; }

    bool InsertNode(Node\*);

    bool InsertNodeRe(Node\* , Node\* );

    void deleteNode(Node\*);

    void TravelNLR();

    void TravelLNR();

    void TravelLRN();

    void NLR(Node \*);

    void LNR(Node \*);

    void LRN(Node \*);

    Node\* search\_x(const string&);

protected:

private:

    Node\* root;

    void DeleteAllNodes(Node\*);

};

#endif // BST\_H

BST.cpp

#include "BST.h"

BST::BST()

    : root(nullptr)

{

}

BST::~BST()

{

    DeleteAllNodes(root);

}

void BST::DeleteAllNodes(Node\* r)

{

    if (r != nullptr)

    {

        DeleteAllNodes(r->Getleft());

        DeleteAllNodes(r->Getright());

**delete** r;

    }

}

bool BST::InsertNode(Node\* newNode)

{

    if (root == nullptr)

    {

        root = newNode;

        return true;

    }

    Node\* current = root;

    Node\* parent = nullptr;

    while (current != nullptr)

    {

        parent = current;

        if (newNode->Getkey() == current->Getkey())

        {

            current->IncrementCount();

**delete** newNode;

            return true;

        }

        else if (newNode->Getkey() < current->Getkey())

        {

            current = current->Getleft();

        }

        else

        {

            current = current->Getright();

        }

    }

    if (newNode->Getkey() < parent->Getkey())

    {

        parent->Setleft(newNode);

    }

    else

    {

        parent->Setright(newNode);

    }

    return true;

}

Node\* BST::search\_x(const std::string& key)

{

    Node\* current = root;

    while (current != nullptr)

    {

        if (key == current->Getkey())

        {

            return current;

        }

        else if (key < current->Getkey())

        {

            current = current->Getleft();

        }

        else

        {

            current = current->Getright();

        }

    }

    return nullptr;

}

void BST::TravelLNR()

{

    LNR(root);

}

void BST::LNR(Node\* r)

{

    if (r != nullptr)

    {

        LNR(r->Getleft());

        cout << "\"" << r->Getkey() << "\" : " << r->GetCount() << " lan.\n";

        LNR(r->Getright());

    }

}

main.cpp

#include <bits/stdc++.h>

#include "BST.h"

#include "Node.h"

using namespace std;

int main()

{

    BST\* tree = **new** BST();

    string text;

    cout << "Nhap van ban (Khong dau): ";

    getline(cin, text);

    stringstream ss(text);

    string word;

    while (ss >> word)

    {

        Node\* n = **new** Node(word);

        tree->InsertNode(n);

    }

    cout << "\nThong ke so luong tu trong van ban:\n";

    tree->TravelLNR();

    string searchWord;

    cout << "\nNhap vao mot tu de kiem tra so lan xuat hien: ";

    cin >> searchWord;

    Node\* foundNode = tree->search\_x(searchWord);

    if(foundNode != nullptr)

    {

        cout << "\"" << foundNode->Getkey() << "\" xuat hien " << foundNode->GetCount() << " lan.\n";

    }

    else

    {

        cout << "\"" << searchWord << "\" khong co trong van ban.\n";

    }

**delete** tree;

    return 0;

}

# **PHẦN 2: CÂY NHỊ PHÂN CÂN BẰNG AVL**

### **1. Xây dựng cấu trúc cây AVL**

Node.h

#ifndef NODE\_H

#define NODE\_H

class Node

{

public:

    Node();

    Node(int);

    virtual ~Node();

    Node \*Getleft() { return left; }

    void Setleft(Node \*val) { left = val; }

    Node \*Getright() { return right; }

    void Setright(Node \*val) { right = val; }

    Node \*Getparent() { return parent; }

    void Setparent(Node \*val) { parent = val; }

    int Getkey() { return key; }

    void Setkey(int val) { key = val; }

    int Getheight() { return height; }

    void Setheight(int val) { height = val; }

protected:

private:

    Node \*left;

    Node \*right;

    Node \*parent;

    int key;

    int height;

};

#endif // NODE\_H

AVL\_tree.h

#pragma once

#ifndef AVL\_TREE\_H

#define AVL\_TREE\_H

#include "Node.h"

class AVL\_tree

{

public:

    AVL\_tree();

    virtual ~AVL\_tree();

    Node \*Getroot() { return root; }

    void Setroot(Node \*val) { root = val; }

    bool InsertNode(Node \*);

    Node \*InsertNode(Node \*, Node \*);

    void InsertNodeRe(Node \*);

    void deleteNode(Node \*);

    void TravelNLR();

    void TravelLNR();

    void TravelLRN();

    void NLR(Node \*);

    void LNR(Node \*);

    void LRN(Node \*);

    void LeftRotate(Node \*&);

    void RightRotate(Node \*&);

    int CheckBal(Node \*);

    int GetHeight(Node \*);

    Node \*search\_x(int);

    void deleteNodeRe(Node\*);

    Node\* deleteNode(Node\*, Node\*);

    Node\* search\_x(int, int&);

    void printAVL(Node\*, int);

    void printAVLTree();

protected:

private:

    Node \*root;

// int bal; // -1 0 1

    int nNum;

    int height;

};

#endif // AVL\_TREE\_H

### **2. Xây dựng cây AVL, khi người dùng nhập vào các dữ liệu sau:**

**50 20 30 10 -5 7 15 35 57 65 55 -1**

Node.cpp

#include "Node.h"

Node::Node()

{

    // ctor

    this->key = 0;

    this->left = nullptr;

    this->right = nullptr;

    this->parent = nullptr;

    this->height = 0;

}

Node::Node(int k)

{

    // ctor

    this->key = k;

    this->left = nullptr;

    this->right = nullptr;

    this->parent = nullptr;

    this->height = 0;

}

Node::~Node()

{

    // dtor

}

AVL\_tree.cpp

#include "AVL\_tree.h"

#include <iostream>

#include "Node.h"

using namespace std;

AVL\_tree::AVL\_tree()

{

    //ctor

    this->root = nullptr;

}

AVL\_tree::~AVL\_tree()

{

    //dtor

}

Node\* AVL\_tree::deleteNode(Node\* currentNode, Node\* n)

{

    if (currentNode == nullptr)

        return currentNode;

    if (n->Getkey() < currentNode->Getkey())

        currentNode->Setleft(deleteNode(currentNode->Getleft(), n));

    else if (n->Getkey() > currentNode->Getkey())

        currentNode->Setright(deleteNode(currentNode->Getright(), n));

    else

    {

        if ((currentNode->Getleft() == nullptr) || (currentNode->Getright() == nullptr))

        {

            Node\* temp = currentNode->Getleft() ? currentNode->Getleft() : currentNode->Getright();

            if (temp == nullptr)

            {

                temp = currentNode;

                currentNode = nullptr;

            }

            else

            {

                \*currentNode = \*temp;

            }

**delete** temp;

        }

        else

        {

            Node\* temp = currentNode->Getright();

            while (temp->Getleft() != nullptr)

                temp = temp->Getleft();

            currentNode->Setkey(temp->Getkey());

            currentNode->Setright(deleteNode(root->Getright(), temp));

        }

    }

    if (currentNode == nullptr)

        return currentNode;

    currentNode->Setheight(1 + max(GetHeight(currentNode->Getleft()), GetHeight(currentNode->Getright())));

    int balance = CheckBal(currentNode);

    // Balance AVL tree

    // Left Left Case

    if (balance > 1 && CheckBal(currentNode->Getleft()) >= 0)

    {

        RightRotate(currentNode);

    }

    // Left Right Case

    if (balance > 1 && CheckBal(currentNode->Getleft()) < 0)

    {

        Node\* nLeft = currentNode->Getleft();

        LeftRotate(nLeft);

        RightRotate(root);

    }

    // Right Right Case

    if (balance < -1 && CheckBal(currentNode->Getright()) <= 0)

    {

        LeftRotate(currentNode);

    }

    // Right Left Case

    if (balance < -1 && CheckBal(currentNode->Getright()) > 0)

    {

        Node\* nRight = currentNode->Getright();

        RightRotate(nRight);

        LeftRotate(currentNode);

    }

    return currentNode;

}

void AVL\_tree::deleteNodeRe(Node\* key)

{

    this->root = deleteNode(this->root, key);

}

bool AVL\_tree::InsertNode(Node\* n)

{

    Node\* p = this->root;

    Node\* T = **new** Node();

    if (root == nullptr)

    {

        this->root = n;

        return true;

    }

    while (p != nullptr)

    {

        T = p;

        if (p->Getkey() > n->Getkey())

            p = p->Getleft();

        else

            if (p->Getkey() < n->Getkey())

                p = p->Getright();

            else

                if (p->Getkey() == n->Getkey())

                    return false;

    }

    if (T->Getkey() > n->Getkey())

        T->Setleft(n);

    else T->Setright(n);

    n->Setparent(T);

    // Rebalance the tree

    Node\* x = n->Getparent();

    while (x != nullptr)

    {

        int bal = this->CheckBal(x);

        switch (bal)

        {

            case 0:

            case 1:

            case -1:

                break;

            case 2:

                if (this->CheckBal(x->Getleft()) >= 0)

                {

                    // Left-Left case

                    if (x == this->root)

                    {

                        this->RightRotate(this->root);

                    }

                    else

                    {

                        Node\* parentX = x->Getparent();

                        this->RightRotate(x);

                        if (parentX->Getleft() == x)

                        {

                            parentX->Setleft(x);

                        }

                        else

                        {

                            parentX->Setright(x);

                        }

                    }

                }

                else

                {

                    // Left-Right case

                    Node\* leftChild = x->Getleft();

                    this->LeftRotate(leftChild);

                    if (x == this->root)

                    {

                        this->RightRotate(this->root);

                    }

                    else

                    {

                        Node\* parentX = x->Getparent();

                        this->RightRotate(x);

                        if (parentX->Getleft() == x)

                        {

                            parentX->Setleft(x);

                        }

                        else

                        {

                            parentX->Setright(x);

                        }

                    }

                }

                break;

            case -2:

                if (this->CheckBal(x->Getright()) <= 0)

                {

                    // Right-Right case

                    if (x == this->root)

                    {

                        this->LeftRotate(this->root);

                    }

                    else

                    {

                        Node\* parentX = x->Getparent();

                        this->LeftRotate(x);

                        if (parentX->Getright() == x)

                        {

                            parentX->Setright(x);

                        }

                        else

                        {

                            parentX->Setleft(x);

                        }

                    }

                }

                else

                {

                    // Right-Left case

                    Node\* rightChild = x->Getright();

                    this->RightRotate(rightChild);

                    if (x == this->root)

                    {

                        this->LeftRotate(this->root);

                    } else

                    {

                        Node\* parentX = x->Getparent();

                        this->LeftRotate(x);

                        if (parentX->Getright() == x)

                        {

                            parentX->Setright(x);

                        }

                        else

                        {

                            parentX->Setleft(x);

                        }

                    }

                }

                break;

        }

        // Move up to the parent node

        x = x->Getparent();

    }

    return true;

}

Node\* AVL\_tree::InsertNode(Node\* r, Node\* p)

{

    // Perform BST insertion

    if (r == nullptr) {

        r = p;

        return r;

    }

    if (p->Getkey() < r->Getkey())

    {

        r->Setleft(InsertNode(r->Getleft(), p));

    }

    else if (p->Getkey() > r->Getkey())

    {

        r->Setright(InsertNode(r->Getright(), p));

    }

    else

    {

        // Duplicate keys are not allowed in this implementation

        return r;

    }

    // Update height of current node

    r->Setheight(1 + std::max(GetHeight(r->Getleft()), GetHeight(r->Getright())));

    // Check balance factor

    int balance = CheckBal(r);

    // Perform rotations if tree becomes unbalanced

    if (balance > 1 && p->Getkey() < r->Getleft()->Getkey()) {

        RightRotate(r);

    }

    if (balance < -1 && p->Getkey() > r->Getright()->Getkey()) {

        LeftRotate(r);

    }

    if (balance > 1 && p->Getkey() > r->Getleft()->Getkey()) {

        Node\* rLeft = r->Getleft();

        LeftRotate(rLeft);

        r->Setleft(rLeft);

        RightRotate(r);

    }

    if (balance < -1 && p->Getkey() < r->Getright()->Getkey()) {

        Node\* rRight = r->Getright();

        RightRotate(rRight);

        r->Setright(rRight);

        LeftRotate(r);

    }

    return r;

}

void AVL\_tree::InsertNodeRe(Node\* p)

{

    this->root = InsertNode(this->root, p);

}

void AVL\_tree::NLR(Node\* r) {

    if (r != nullptr)

    {

        cout << r->Getkey() << "\n";

        NLR(r->Getleft());

        NLR(r->Getright());

    }

}

void AVL\_tree::LNR(Node\* r)

{

    if(r != nullptr)

    {

        LNR(r->Getleft());

        cout << r->Getkey() << "\n";

        LNR(r->Getright());

    }

}

void AVL\_tree::LRN(Node\* r)

{

    if (r != nullptr)

    {

        LRN(r->Getleft());

        LRN(r->Getright());

        cout << r->Getkey() << "\n";

    }

}

void AVL\_tree::TravelNLR() {

    NLR(this->root);

}

void AVL\_tree::TravelLNR() {

    LNR(this->root);

}

void AVL\_tree::TravelLRN() {

    LRN(this->root);

}

void AVL\_tree::deleteNode(Node \*n)

{

    Node \*p = n;

    if (n->Getleft() == nullptr && n->Getright() == nullptr)

**delete** n;

    else

    {

        if (p->Getright() != nullptr)

        {

            while (p->Getleft() != nullptr) //

                p = p->Getleft();

            n->Setkey(p->Getkey());

            // Code

**delete** p;

        }

        else

        {

            while (p->Getright() != nullptr) //

                p = p->Getright();

            n->Setkey(p->Getkey());

            // Code

**delete** p;

        }

    }

}

Node\* AVL\_tree::search\_x(int k, int &compareCount)

{

    Node\* p = this->root;

    while (p != nullptr)

    {

        compareCount++;

        if (p->Getkey() == k)

        {

            return p;

        }

        else if (k < p->Getkey())

        {

            p = p->Getleft();

        }

        else

        {

            p = p->Getright();

        }

    }

    return nullptr;

}

void AVL\_tree::RightRotate(Node\*& P)

{

    Node\* Q = P->Getleft();

    Node\* T = Q->Getright();

    Q->Setright(P);

    P->Setleft(T);

    // Update parents

    Q->Setparent(P->Getparent());

    P->Setparent(Q);

    if (T != nullptr)

    {

        T->Setparent(P);

    }

    // Update the parent's child pointer

    if (Q->Getparent() != nullptr)

    {

        if (Q->Getparent()->Getleft() == P)

        {

            Q->Getparent()->Setleft(Q);

        }

        else

        {

            Q->Getparent()->Setright(Q);

        }

    }

    else

    {

        // if parent of Q is nullptr, Q becomes the root

        this->root = Q;

    }

    P = Q;

}

void AVL\_tree::LeftRotate(Node\*& P)

{

    Node\* Q = P->Getright();

    Node\* T = Q->Getleft();

    Q->Setleft(P);

    P->Setright(T);

    // Update parents

    Q->Setparent(P->Getparent());

    P->Setparent(Q);

    if (T != nullptr)

    {

        T->Setparent(P);

    }

    // Update the parent's child pointer

    if (Q->Getparent() != nullptr)

    {

        if (Q->Getparent()->Getleft() == P)

        {

            Q->Getparent()->Setleft(Q);

        }

        else

        {

            Q->Getparent()->Setright(Q);

        }

    }

    else

    {

        // if parent of Q is nullptr, Q becomes the root

        this->root = Q;

    }

    P = Q;

}

int AVL\_tree::CheckBal(Node\* p)

{

    int bal = this->GetHeight(p->Getleft()) - this->GetHeight(p->Getright());

    return bal;

}

int AVL\_tree::GetHeight(Node\* p)

{

    if (p == nullptr) return 0;

    else

        return 1 + max(GetHeight(p->Getleft()), GetHeight(p->Getright()));

}

void AVL\_tree::printAVL(Node\* root, int iSpace)

{

    if (root == NULL)

        return;

    iSpace += 8;

    printAVL(root->Getright(), iSpace);

    cout << endl;

    for (int i = 8; i < iSpace; i++)

        cout << " ";

    cout << root->Getkey() << "\n";

    printAVL(root->Getleft(), iSpace);

}

void AVL\_tree::printAVLTree()

{

    printAVL(this->root, 0);

}

main.cpp

#include <iostream>

#include "AVL\_tree.h"

#include "Node.h"

using namespace std;

int main()

{

    AVL\_tree tree;

    int values[] = {50, 20, 30, 10, -5, 7, 15, 35, 57, 65, 55, -1};

    for(int val : values)

    {

        Node\* n = **new** Node(val);

        tree.InsertNodeRe(n);

    }

    // Hiển thị cây AVL theo thứ tự in NLR

    cout << "Cay AVL sau khi chen cac gia tri:\n";

    tree.TravelNLR();

    // Hiển thị cây AVL dạng hình ảnh

    cout << "\nCay AVL (hien thi hinh anh):\n";

    tree.printAVLTree();

    return 0;

}

### **3. Vẽ hình cây AVL được tạo ra từ phần nhập liệu ở câu 2.**

30

/ \

10 50

/ \ / \

-1 20 35 57

/ \ / / \

-5 7 15 55 65

### **4. Hãy ghi chú các thông tin bằng cách trả lời các câu hỏi ứng với các dòng lệnh có yêu cầu ghi chú (//Ghi chú) trong các hàm InsertNode,.**

bool AVL\_tree::InsertNode(Node\* n)

{

    Node\* p = this->root;

    Node\* T = **new** Node();

    if (root == nullptr)

    {

        this->root = n;

        return true;

    }

    while (p != nullptr)

    {

        T = p;

        if (p->Getkey() > n->Getkey())

            p = p->Getleft();

        else if (p->Getkey() < n->Getkey())

            p = p->Getright();

        else

            return false;

    }

    if (T->Getkey() > n->Getkey())

        T->Setleft(n);

    else

        T->Setright(n);

    n->Setparent(T);

    // Rebalance the tree : cân bằng lại của cây AVL

    Node\* x = n->Getparent();

    while (x != nullptr)

    {

        int bal = this->CheckBal(x);

        switch (bal)

        {

            case 0:

            case 1:

            case -1:

                break;

            case 2:

                if (this->CheckBal(x->Getleft()) >= 0)

                {

                    // Case Left-Left: Xoay phải tại nút x

                    if (x == this->root)

                    {

                        this->RightRotate(this->root);

                    }

                    else

                    {

                        Node\* parentX = x->Getparent();

                        this->RightRotate(x);

                        if (parentX->Getleft() == x)

                        {

                            parentX->Setleft(x);

                        }

                        else

                        {

                            parentX->Setright(x);

                        }

                    }

                }

                else

                {

                    // Case Left-Right: Xoay trái tại nút con trái rồi xoay phải tại nút x

                    Node\* leftChild = x->Getleft();

                    this->LeftRotate(leftChild);

                    if (x == this->root)

                    {

                        this->RightRotate(this->root);

                    }

                    else

                    {

                        Node\* parentX = x->Getparent();

                        this->RightRotate(x);

                        if (parentX->Getleft() == x)

                        {

                            parentX->Setleft(x);

                        }

                        else

                        {

                            parentX->Setright(x);

                        }

                    }

                }

                break;

            case -2:

                if (this->CheckBal(x->Getright()) <= 0)

                {

                    // Case Right-Right: Xoay trái tại nút x

                    if (x == this->root)

                    {

                        this->LeftRotate(this->root);

                    }

                    else

                    {

                        Node\* parentX = x->Getparent();

                        this->LeftRotate(x);

                        if (parentX->Getright() == x)

                        {

                            parentX->Setright(x);

                        }

                        else

                        {

                            parentX->Setleft(x);

                        }

                    }

                }

                else

                {

                    // Case Right-Left: Xoay phải tại nút con phải rồi xoay trái tại nút x

                    Node\* rightChild = x->Getright();

                    this->RightRotate(rightChild);

                    if (x == this->root)

                    {

                        this->LeftRotate(this->root);

                    }

                    else

                    {

                        Node\* parentX = x->Getparent();

                        this->LeftRotate(x);

                        if (parentX->Getright() == x)

                        {

                            parentX->Setright(x);

                        }

                        else

                        {

                            parentX->Setleft(x);

                        }

                    }

                }

                break;

        }

        // Di chuyển lên nút cha để tiếp tục kiểm tra cân bằng

        x = x->Getparent();

    }

    return true;

}

Node\* AVL\_tree::InsertNode(Node\* r, Node\* p)

{

    // Thuc hien chen theo BST

    if (r == nullptr) {

        r = p;

        return r;

    }

    if (p->Getkey() < r->Getkey())

    {

        r->Setleft(InsertNode(r->Getleft(), p));

    }

    else if (p->Getkey() > r->Getkey())

    {

        r->Setright(InsertNode(r->Getright(), p));

    }

    else

    {

        // Khong cho phep chen key trung nhau

        return r;

    }

    // Cap nhat chieu cao cua node hien tai

    r->Setheight(1 + std::max(GetHeight(r->Getleft()), GetHeight(r->Getright())));

    // Kiem tra chi so can bang

    int balance = CheckBal(r);

    // Thuc hien xoay neu cay tro nen mat can bang

    if (balance > 1 && p->Getkey() < r->Getleft()->Getkey()) {

        RightRotate(r);

    }

    if (balance < -1 && p->Getkey() > r->Getright()->Getkey()) {

        LeftRotate(r);

    }

    if (balance > 1 && p->Getkey() > r->Getleft()->Getkey()) {

        Node\* rLeft = r->Getleft();

        LeftRotate(rLeft);

        r->Setleft(rLeft);

        RightRotate(r);

    }

    if (balance < -1 && p->Getkey() < r->Getright()->Getkey()) {

        Node\* rRight = r->Getright();

        RightRotate(rRight);

        r->Setright(rRight);

        LeftRotate(r);

    }

    return r;

}

### **5. Sinh viên cài đặt lại các hàm dùng cho cây nhị phân và cây NPTK để áp dụng cho cây AVL.**

Node\* AVL\_tree::InsertNode(Node\* r, Node\* p)

{

if (r == nullptr) {

r = p;

return r;

}

if (p->Getkey() < r->Getkey())

{

r->Setleft(InsertNode(r->Getleft(), p));

}

else if (p->Getkey() > r->Getkey())

{

r->Setright(InsertNode(r->Getright(), p));

}

else

{

return r;

}

r->Setheight(1 + std::max(GetHeight(r->Getleft()), GetHeight(r->Getright())));

int balance = CheckBal(r);

if (balance > 1 && p->Getkey() < r->Getleft()->Getkey()) {

RightRotate(r);

}

if (balance < -1 && p->Getkey() > r->Getright()->Getkey()) {

LeftRotate(r);

}

if (balance > 1 && p->Getkey() > r->Getleft()->Getkey()) {

Node\* rLeft = r->Getleft();

LeftRotate(rLeft);

r->Setleft(rLeft);

RightRotate(r);

}

if (balance < -1 && p->Getkey() < r->Getright()->Getkey()) {

Node\* rRight = r->Getright();

RightRotate(rRight);

r->Setright(rRight);

LeftRotate(r);

}

return r;

}

## **Áp dụng – Nâng cao**

### **1. Sinh viên tự cài đặt thêm chức năng cho phép người dùng nhập vào khóa x và kiểm tra xem khóa x có nằm trong cây AVL hay không.**

**Cho dãy A như sau: 1 3 5 7 9 12 15 17 21 23 25 27**

a. Tạo cây AVL từ dãy A. Cho biết số phép so sánh cần thực hiện để tìm phần tử 21 trên cây AVL vừa tạo.

b. Tạo cây nhị phân tìm kiếm từ dãy A dùng lại đoạn code tạo cây của bài thực hành trước). Cho biết số phép so sánh cần thực hiện để tìm phần tử 21 trên cây nhị phân tìm kiếm vừa tạo.

c. So sánh 2 kết quả trên và rút ra nhận xét?

a.

AVL\_tree.h

    Node\* search\_x(int, int&);

AVL\_tree.cpp

Node\* AVL\_tree::search\_x(int k, int &compareCount)

{

    Node\* p = this->root;

    compareCount = 0;

    while (p != nullptr)

    {

        compareCount++;

        if (p->Getkey() == k)

        {

            return p;

        }

        else if (k < p->Getkey())

        {

            p = p->Getleft();

        }

        else

        {

            p = p->Getright();

        }

    }

    return nullptr;

}

main.cpp

#include <iostream>

#include "AVL\_tree.h"

#include "Node.h"

using namespace std;

int main()

{

    AVL\_tree tree;

    int values[] = {1, 3, 5, 7, 9, 12, 15, 17, 21, 23, 25, 27};

    for(int val : values)

    {

        Node\* n = **new** Node(val);

        tree.InsertNodeRe(n);

    }

    cout << "Cay AVL sau khi chen cac gia tri:\n";

    tree.TravelNLR();

    cout << "\nCay AVL (hien thi hinh anh):\n";

    tree.printAVLTree();

    int x;

    cout << "\nNhap vao khoa x de tim kiem: ";

    cin >> x;

    int compareCount = 0;

    Node\* result = tree.search\_x(x, compareCount);

    if(result != nullptr)

    {

        cout << "Tim thay khoa " << x << " trong cay AVL.\n";

    }

    else

    {

        cout << "Khong tim thay khoa " << x << " trong cay AVL.\n";

    }

    cout << "So phep so sanh thuc hien: " << compareCount << "\n";

    return 0;

}

Số phép so sánh cần thực hiện để tìm phần tử 21 trên cây AVL là 3

b.

BST.h

    Node\* search\_x(Node\*, int);

    Node\* search\_x(int);

BST.cpp

Node\* BST::search\_x(int k)

{

    return search\_x(this->root, k);

}

Node\* BST::search\_x(Node\* root, int k)

{

    if (root == nullptr) return nullptr;

    if (root->Getkey() == k) return root;

    if (root->Getkey() > k) return search\_x(root->Getleft(), k);

    return search\_x(root->Getright(), k);

}

Số phép so sánh cần thực hiện để tìm phần tử 21 trên cây nhị phân là 9

c.

Nhận xét:

* Số phép so sánh cần thực hiện để tìm phần tử 21 trên cây AVL là 3
* Số phép so sánh cần thực hiện để tìm phần tử 21 trên cây nhị phân là 9

Cây AVL vì cái tính tự cân bằng nên thành ra nó làm cái cây AVL trở nên cân bằng do đó khi tìm kiếm phần tử 21 chỉ cần 3 phép so sánh, còn Cây BST thì lại không cân bằng nên khi tìm kiếm phần tử 21 cần tới 9 phép so sánh.

### **2. Cài đặt chương trình đọc các số nguyên từ tập tin input.txt (không biết trước số lượng số nguyên trên tập tin) và tạo cây AVL từ dữ liệu đọc được**

main.cpp

#include <iostream>

#include <fstream>

#include "AVL\_tree.h"

#include "Node.h"

using namespace std;

int main()

{

    AVL\_tree tree;

    ifstream infile("input.txt");

    if (!infile.is\_open())    {

        cerr << "Khong mo dc input.txt\n";

        return 1;

    }

    int val;

    while (infile >> val)    {

        Node\* n = **new** Node(val);

        tree.InsertNodeRe(n);

    }

    infile.close();

    cout << "Cay AVL sau khi chen cac gia tri:\n";

    tree.TravelNLR();

    cout << "\nCay AVL (hien thi hinh anh):\n";

    tree.printAVLTree();

    int x;

    cout << "\nNhap vao khoa x de tim kiem: ";

    cin >> x;

    int compareCount = 0;

    Node\* result = tree.search\_x(x, compareCount);

    if(result != nullptr)

    {

        cout << "Tim thay khoa " << x << " trong cay AVL.\n";

    }

    else

    {

        cout << "Khong tim thay khoa " << x << " trong cay AVL.\n";

    }

    cout << "So phep so sanh thuc hien: " << compareCount << "\n";

    return 0;

}

### **3. Cài đặt cây cân bằng AVL trong đó mỗi node trên cây lưu thông tin sinh viên**

Node.h

#ifndef NODE\_H

#define NODE\_H

#include "Student.h"

class Node

{

public:

    Node() : left(nullptr), right(nullptr), height(1) {}

    Node(const Student& s) : student(s), left(nullptr), right(nullptr), height(1) {}

    // Getters and setters

    const Student& GetStudent() const { return student; }

    void SetStudent(const Student& s) { student = s; }

    Node \*Getleft() const { return left; }

    void Setleft(Node \*val) { left = val; }

    Node \*Getright() const { return right; }

    void Setright(Node \*val) { right = val; }

    Node \*Getparent() const { return parent; }

    void Setparent(Node \*val) { parent = val; }

    int Getkey() const { return key; }

    void Setkey(int val) { key = val; }

    int getHeight() const { return height; }

    void Setheight(int val) { height = val; }

protected:

private:

    Node \*left;

    Node \*right;

    Node \*parent;

    int key;

    int height;

    Student student;

};

#endif // NODE\_H

Student.h

#pragma once

#ifndef STUDENT\_H

#define STUDENT\_H

#include <iostream>

#include <algorithm>

using namespace std;

class Student

{

private:

    int studentID;

    string name;

    string dateOfBirth;

    float gpa;

public:

    Student() : studentID(0), name(""), dateOfBirth(""), gpa(0.0) {}

    Student(int id, const string& n, const string& d, float g) :

        studentID(id), name(n), dateOfBirth(d), gpa(g) {}

    int GetStudentID() const { return studentID; }

    void SetStudentID(int id) { studentID = id; }

    const string& GetName() const { return name; }

    void SetName(const string& n) { name = n; }

    const string& GetDateOfBirth() const { return dateOfBirth; }

    void SetDateOfBirth(const string& d) { dateOfBirth = d; }

    float GetGPA() const { return gpa; }

    void SetGPA(float g) { gpa = g; }

};

#endif // STUDENT\_H

AVL\_tree.h

#pragma once

#ifndef AVL\_TREE\_H

#define AVL\_TREE\_H

#include "Node.h"

class AVL\_tree

{

public:

    AVL\_tree() : root(nullptr) {}

    virtual ~AVL\_tree();

    Node \*Getroot() { return root; }

    void Setroot(Node \*val) { root = val; }

    bool InsertNode(Node \*);

    Node \*InsertNode(Node \*, Node \*);

    void InsertNodeRe(Node \*);

    void deleteNode(Node \*);

    int CheckBal(Node \*);

    int GetHeight(Node \*);

    void insert(const Student& student);

    void deleteNode(int key);

    bool search(int key);

    void LNR(Node\* root);

    void TravelLNR();

    void NLR(Node\* root);

    void TravelNLR();

protected:

private:

    Node \*root;

    int getHeight(Node\* p);

    int checkBalance(Node\* p);

    void rotateLeft(Node\*& x);

    void rotateRight(Node\*& y);

    Node\* insert(Node\* node, const Student& student);

    bool search(Node\* root, int key);

};

#endif // AVL\_TREE\_H

AVL\_tree.cpp

#include "AVL\_tree.h"

#include "Node.h"

#include <iostream>

using namespace std;

AVL\_tree::~AVL\_tree()

{

    //dtor

}

int AVL\_tree::getHeight(Node \*p)

{

    if (p == nullptr)

        return 0;

    return p->getHeight();

}

int AVL\_tree::checkBalance(Node \*p)

{

    if (p == nullptr)

        return 0;

    return getHeight(p->Getleft()) - getHeight(p->Getright());

}

void AVL\_tree::rotateLeft(Node\*& x)

{

    Node\* y = x->Getright();

    Node\* T2 = y->Getleft();

    y->Setleft(x);

    x->Setright(T2);

    x->Setheight(1 + max(getHeight(x->Getleft()), getHeight(x->Getright())));

    y->Setheight(1 + max(getHeight(y->Getleft()), getHeight(y->Getright())));

    x = y;

}

void AVL\_tree::rotateRight(Node\*& y)

{

    Node\* x = y->Getleft();

    Node\* T2 = x->Getright();

    x->Setright(y);

    y->Setleft(T2);

    y->Setheight(1 + max(getHeight(y->Getleft()), getHeight(y->Getright())));

    x->Setheight(1 + max(getHeight(x->Getleft()), getHeight(x->Getright())));

        y = x;

}

    Node\* AVL\_tree::deleteNode(Node\* root, int key)

{

    if (root == nullptr) return root;

    if (key < root->GetStudent().GetStudentID())

        root->Setleft(deleteNode(root->Getleft(), key));

    else if (key > root->GetStudent().GetStudentID())

        root->Setright(deleteNode(root->Getright(), key));

    else

    {

        if (root->Getleft() == nullptr && root->Getright() == nullptr)

        {

                Node\* temp = root->Getleft() ? root->Getleft() : root->Getright();

                if (temp == nullptr)

                {

                    temp = root;

                    root = nullptr;

                }

                else

                {

                    \*root = \*temp;

                }

**delete** temp;

            }

            else

            {

                Node\* temp = root->Getright();

                while (temp->Getleft() != nullptr)

                    temp = temp->Getleft();

                root->SetStudent(temp->GetStudent());

                root->Setright(deleteNode(root->Getright(), temp->GetStudent().GetStudentID()));

            }

        }

        if (root == nullptr)

            return root;

        root->Setheight(1 + max(getHeight(root->Getleft()), getHeight(root->Getright())));

        int balance = checkBalance(root);

        if (balance > 1 && checkBalance(root->Getleft()) >= 0)

            rotateRight(root);

        if (balance > 1 && checkBalance(root->Getleft()) < 0)

        {

            Node\* nLeft = root->Getleft();

            rotateLeft(nLeft);

            rotateRight(root);

        }

        if (balance < -1 && checkBalance(root->Getright()) <= 0)

            rotateLeft(root);

        if (balance < -1 && checkBalance(root->Getright()) > 0)

        {

            Node\* nRight = root->Getright();

            rotateRight(nRight);

            rotateLeft(root);

        }

        return root;

    }

    Node\* AVL\_tree::insert(Node\* node, const Student& student)

    {

        if (node == nullptr)

            return **new** Node(student);

        if (student.GetStudentID() < node->GetStudent().GetStudentID())

            node->Setleft(insert(node->Getleft(), student));

        else if (student.GetStudentID() > node->GetStudent().GetStudentID())

            node->Setright(insert(node->Getright(), student));

        else

            return node;

        node->Setheight(1 + max(getHeight(node->Getleft()), getHeight(node->Getright())));

        int balance = checkBalance(node);

        if (balance > 1 && student.GetStudentID() < node->Getleft()->GetStudent().GetStudentID())

            rotateRight(node);

        if (balance < -1 && student.GetStudentID() > node->Getright()->GetStudent().GetStudentID())

            rotateLeft(node);

        if (balance > 1 && student.GetStudentID() > node->Getleft()->GetStudent().GetStudentID())

        {

            Node\* nLeft = node->Getleft();

            rotateLeft(nLeft);

            rotateRight(node);

        }

        if (balance < -1 && student.GetStudentID() < node->Getright()->GetStudent().GetStudentID())

        {

            Node\* nRight = node->Getright();

            rotateRight(nRight);

            rotateLeft(node);

        }

        return node;

    }

bool AVL\_tree::search(Node\* root, int key) {

        if (root == nullptr)

            return false;

        if (key == root->GetStudent().GetStudentID())

            return true;

        else if (key < root->GetStudent().GetStudentID())

            return search(root->Getleft(), key);

        else

            return search(root->Getright(), key);

    }

void AVL\_tree::insert(const Student& student)

{

    root = insert(root, student);

}

void AVL\_tree::deleteNode(int key)

    {

        root = deleteNode(root, key);

    }

    bool AVL\_tree::search(int key)

{

    return search(root, key);

}

void AVL\_tree::LNR(Node\* root)

    {

        if (root != nullptr)

        {

            LNR(root->Getleft());

            cout << "Student ID: " << root->GetStudent().GetStudentID() << ", Name: " << root->GetStudent().GetName()

                << ", dateOfBirth: " << root->GetStudent().GetDateOfBirth() << ", GPA: " << root->GetStudent().GetGPA() << endl;

            LNR(root->Getright());

        }

    }

    void AVL\_tree::TravelLNR()

    {

        LNR(root);

    }

    void AVL\_tree::NLR(Node\* root)

    {

        if (root != nullptr)

        {

            cout << "Student ID: " << root->GetStudent().GetStudentID() << ", Name: " << root->GetStudent().GetName()

                << ", dateOfBirth: " << root->GetStudent().GetDateOfBirth() << ", GPA: " << root->GetStudent().GetGPA() << endl;

            LNR(root->Getleft());

            LNR(root->Getright());

        }

    }

    void AVL\_tree::TravelNLR()

    {

        NLR(root);

    }

main.cpp

#include <iostream>

#include <fstream>

#include "AVL\_tree.h"

#include "AVL\_tree.cpp"

#include "Node.h"

#include "Student.h"

using namespace std;

int main()

{

    AVL\_tree tree;

    // Inserting some students

    tree.insert(Student(101, "An", "2002-05-15", 3.0));

    tree.insert(Student(102, "Bo", "2003-02-28", 3.6));

    tree.insert(Student(103, "Ca", "2004-10-10", 3.7));

    tree.insert(Student(104, "De", "2005-06-20", 2.9));

    tree.TravelLNR();

    cout << "-------------\n";

    tree.TravelNLR();

    int search\_id = 102;

    bool found = tree.search(search\_id);

    if (found)

    {

        cout << "Student with ID " << search\_id << " found in the AVL tree." << endl;

    }

    else

    {

        cout << "Student with ID " << search\_id << " not found in the AVL tree." << endl;

    }

    return 0;

}

### **4. Tự tìm hiểu và cài đặt chức năng xóa một node ra khỏi cây AVL.**

AVL\_tree.h

    void deleteNode(int key);

AVL\_tree.cpp

    Node\* AVL\_tree::deleteNode(Node\* root, int key)

{

    if (root == nullptr) return root;

    if (key < root->GetStudent().GetStudentID())

        root->Setleft(deleteNode(root->Getleft(), key));

    else if (key > root->GetStudent().GetStudentID())

        root->Setright(deleteNode(root->Getright(), key));

    else

    {

        if (root->Getleft() == nullptr && root->Getright() == nullptr)

        {

                Node\* temp = root->Getleft() ? root->Getleft() : root->Getright();

                if (temp == nullptr)

                {

                    temp = root;

                    root = nullptr;

                }

                else

                {

                    \*root = \*temp;

                }

**delete** temp;

            }

            else

            {

                Node\* temp = root->Getright();

                while (temp->Getleft() != nullptr)

                    temp = temp->Getleft();

                root->SetStudent(temp->GetStudent());

                root->Setright(deleteNode(root->Getright(), temp->GetStudent().GetStudentID()));

            }

        }

        if (root == nullptr)

            return root;

        root->Setheight(1 + max(getHeight(root->Getleft()), getHeight(root->Getright())));

        int balance = checkBalance(root);

        if (balance > 1 && checkBalance(root->Getleft()) >= 0)

            rotateRight(root);

        if (balance > 1 && checkBalance(root->Getleft()) < 0)

        {

            Node\* nLeft = root->Getleft();

            rotateLeft(nLeft);

            rotateRight(root);

        }

        if (balance < -1 && checkBalance(root->Getright()) <= 0)

            rotateLeft(root);

        if (balance < -1 && checkBalance(root->Getright()) > 0)

        {

            Node\* nRight = root->Getright();

            rotateRight(nRight);

            rotateLeft(root);

        }

        return root;

    }

void AVL\_tree::deleteNode(int key)

    {

        root = deleteNode(root, key);

    }

main.cpp

#include <iostream>

#include "AVL\_tree.h"

#include "Node.h"

#include "Student.h"

using namespace std;

int main()

{

    AVL\_tree tree;

    // Inserting some students

    tree.insert(Student(101, "An", "2002-05-15", 3.0));

    tree.insert(Student(102, "Bo", "2003-02-28", 3.6));

    tree.insert(Student(103, "Ca", "2004-10-10", 3.7));

    tree.insert(Student(104, "De", "2005-06-20", 2.9));

    tree.TravelLNR();

    cout << "-------------\n";

    tree.TravelNLR();

    int search\_id = 102;

    bool found = tree.search(search\_id);

    if (found)

    {

        cout << "Student with ID " << search\_id << " found in the AVL tree." << endl;

    }

    else

    {

        cout << "Student with ID " << search\_id << " not found in the AVL tree." << endl;

    }

    cout << "\nCay AVL (Hien thi hinh anh):\n";

    tree.printAVLTree();

    cout << "-------------\n";

    int delete\_id;

    cout << "Nhap ID sinh vien can xoa: ";

    cin >> delete\_id;

    cout << "Xoa sinh vien co ID " << delete\_id << " khoi cay AVL.\n";

    tree.deleteNode(delete\_id);

    cout << "\nCay AVL sau khi xoa:\n";

    tree.TravelLNR();

    cout << "-------------\n";

    cout << "\nCay AVL (Hien thi hinh anh) sau khi xoa:\n";

    tree.printAVLTree();

    cout << "-------------\n";

    return 0;

}

## **BÀI TẬP THÊM**

### **1. Viết chương trình cho phép tạo, tra cứu và sửa chữa từ điển Anh-Việt.**

Word.h

#ifndef WORD\_H

#define WORD\_H

#include <string>

using namespace std;

class Word {

public:

    string english;

    string vietnamese;

    Word() = default;

    Word(const string& eng, const string& vie)

        : english(eng), vietnamese(vie) {}

};

#endif // WORD\_H

Node.h

#ifndef NODE\_H

#define NODE\_H

#include "Word.h"

class Node {

public:

Word data;

Node\* left;

Node\* right;

int height;

Node(const Word& w)

: data(w), left(nullptr), right(nullptr), height(1) {}

};

#endif // NODE\_H

AVLTree.h

#ifndef AVL\_TREE\_H

#define AVL\_TREE\_H

#include "Node.h"

#include <string>

using namespace std;

class AVLTree {

public:

    AVLTree();

    ~AVLTree();

    void insert(const Word& word);

    bool search(const string& english, string& vietnamese) const;

    bool modify(const string& english, const string& new\_vietnamese);

    void inOrderTraversal() const;

private:

    Node\* root;

    int getHeight(Node\* node) const;

    int getBalance(Node\* node) const;

    Node\* rightRotate(Node\* y);

    Node\* leftRotate(Node\* x);

    Node\* insert(Node\* node, const Word& word);

    Node\* minValueNode(Node\* node);

    Node\* search(Node\* node, const string& english) const;

    void inOrderTraversal(Node\* node) const;

    void destroy(Node\* node);

};

#endif // AVL\_TREE\_H

AVLTree.cpp

#include "AVLTree.h"

#include <iostream>

AVLTree::AVLTree() : root(nullptr) {}

AVLTree::~AVLTree() {

    destroy(root);

}

void AVLTree::destroy(Node\* node) {

    if (node != nullptr) {

        destroy(node->left);

        destroy(node->right);

**delete** node;

    }

}

int AVLTree::getHeight(Node\* node) const {

    if (node == nullptr)

        return 0;

    return node->height;

}

int AVLTree::getBalance(Node\* node) const {

    if (node == nullptr)

        return 0;

    return getHeight(node->left) - getHeight(node->right);

}

Node\* AVLTree::rightRotate(Node\* y) {

    Node\* x = y->left;

    Node\* T2 = x->right;

    x->right = y;

    y->left = T2;

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

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

    return x;

}

Node\* AVLTree::leftRotate(Node\* x) {

    Node\* y = x->right;

    Node\* T2 = y->left;

    y->left = x;

    x->right = T2;

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

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

    return y;

}

Node\* AVLTree::insert(Node\* node, const Word& word) {

    if (node == nullptr)

        return **new** Node(word);

    if (word.english < node->data.english)

        node->left = insert(node->left, word);

    else if (word.english > node->data.english)

        node->right = insert(node->right, word);

    else

        return node;

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

    int balance = getBalance(node);

    if (balance > 1 && word.english < node->left->data.english)

        return rightRotate(node);

    if (balance < -1 && word.english > node->right->data.english)

        return leftRotate(node);

    if (balance > 1 && word.english > node->left->data.english) {

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

        return rightRotate(node);

    }

    if (balance < -1 && word.english < node->right->data.english) {

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

        return leftRotate(node);

    }

    return node;

}

void AVLTree::insert(const Word& word) {

    root = insert(root, word);

}

Node\* AVLTree::minValueNode(Node\* node) {

    Node\* current = node;

    while (current->left != nullptr)

        current = current->left;

    return current;

}

bool AVLTree::search(const string& english, string& vietnamese) const {

    Node\* result = search(root, english);

    if (result != nullptr) {

        vietnamese = result->data.vietnamese;

        return true;

    }

    return false;

}

Node\* AVLTree::search(Node\* node, const string& english) const {

    if (node == nullptr || node->data.english == english)

        return node;

    if (english < node->data.english)

        return search(node->left, english);

    else

        return search(node->right, english);

}

bool AVLTree::modify(const string& english, const string& new\_vietnamese) {

    Node\* node = search(root, english);

    if (node != nullptr) {

        node->data.vietnamese = new\_vietnamese;

        return true;

    }

    return false;

}

void AVLTree::inOrderTraversal() const {

    inOrderTraversal(root);

}

void AVLTree::inOrderTraversal(Node\* node) const {

    if (node != nullptr) {

        inOrderTraversal(node->left);

        cout << "Tieng Anh: " << node->data.english

            << " | Tieng Viet: " << node->data.vietnamese << endl;

        inOrderTraversal(node->right);

    }

}

Dictionary.h

#ifndef DICTIONARY\_H

#define DICTIONARY\_H

#include "AVLTree.h"

#include <string>

using namespace std;

class Dictionary {

public:

    Dictionary();

    ~Dictionary();

    void addWord(const string& english, const string& vietnamese);

    bool lookupWord(const string& english, string& vietnamese) const;

    bool modifyWord(const string& english, const string& new\_vietnamese);

    void displayAll() const;

private:

    AVLTree tree;

};

#endif // DICTIONARY\_H

Dictionary.cpp

#include "Dictionary.h"

#include <fstream>

#include <iostream>

Dictionary::Dictionary() {}

Dictionary::~Dictionary() {}

void Dictionary::addWord(const string& english, const string& vietnamese) {

Word word(english, vietnamese);

tree.insert(word);

}

bool Dictionary::lookupWord(const string& english, string& vietnamese) const {

return tree.search(english, vietnamese);

}

bool Dictionary::modifyWord(const string& english, const string& new\_vietnamese) {

return tree.modify(english, new\_vietnamese);

}

void Dictionary::displayAll() const {

tree.inOrderTraversal();

}

main.cpp

#include "Dictionary.h"

#include "Node.h"

#include "AVLTree.h"

#include "Word.h"

#include <iostream>

#include <limits>

using namespace std;

int main() {

    Dictionary dict;

    int choice;

    string eng, vie;

    while (true) {

        cout << "\n===== Tu dien Anh-Viet =====\n";

        cout << "1. Them tu\n";

        cout << "2. Tim kiem tu\n";

        cout << "3. Sua tu\n";

        cout << "4. Hien thi tat ca tu\n";

        cout << "5. Thoat\n";

        cout << "Nhap lua chon: ";

        cin >> choice;

        if (cin.fail()) {

            cin.clear();

            cin.ignore(numeric\_limits<streamsize>::max(), '\n');

            cout << "Nhap sai. Vui long nhap so tu 1 den 5.\n";

            continue;

        }

        switch (choice) {

            case 1:

                cout << "Nhap tu Anh: ";

                cin.ignore(numeric\_limits<streamsize>::max(), '\n');

                getline(cin, eng);

                cout << "Nhap nghia tieng Viet: ";

                getline(cin, vie);

                dict.addWord(eng, vie);

                cout << "Them tu thanh cong.\n";

                break;

            case 2:

                cout << "Nhap tu Anh: ";

                cin.ignore(numeric\_limits<streamsize>::max(), '\n');

                getline(cin, eng);

                if (dict.lookupWord(eng, vie)) {

                    cout << "Nghia tieng Viet: " << vie << "\n";

                } else {

                    cout << "Tu khong tim thay.\n";

                }

                break;

            case 3:

                cout << "Nhap tu Anh: ";

                cin.ignore(numeric\_limits<streamsize>::max(), '\n');

                getline(cin, eng);

                cout << "Nhap nghia tieng Viet: ";

                getline(cin, vie);

                if (dict.modifyWord(eng, vie)) {

                    cout << "Sua tu thanh cong.\n";

                } else {

                    cout << "Tu khong tim thay.\n";

                }

                break;

            case 4:

                cout << "\nNoi dung tu dien:\n";

                dict.displayAll();

                break;

            case 5:

                cout << "Thoat chuong trinh. Tam biet!\n";

                return 0;

            default:

                cout << "Lua chon khong hop le. Vui long nhap lai.\n";

        }

    }

    return 0;

}